Course 2: Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

Author:	Pradeep K. Pant
URL:	https://www.coursera.org/learn/deep-neural-network/home/welcome

****

Course 2: Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

****

This course will teach you the "magic" of getting deep learning to work well. Rather than the deep learning process being a black box, you will understand what drives performance, and be able to more systematically get good results. You will also learn TensorFlow.
After 3 weeks, you will:
- Understand industry best-practices for building deep learning applications.
- Be able to effectively use the common neural network "tricks", including initialization, L2 and dropout regularization, Batch normalization, gradient checking,
- Be able to implement and apply a variety of optimization algorithms, such as mini-batch gradient descent, Momentum, RMSprop and Adam, and check for their convergence.
- Understand new best-practices for the deep learning era of how to set up train/dev/test sets and analyze bias/variance
- Be able to implement a neural network in TensorFlow.

Week 1: Introduction to Deep Neural Networks

Learning Objectives

- Recall that different types of initializations lead to different results
- Recognize the importance of initialization in complex neural networks.
- Recognize the difference between train/dev/test sets
- Diagnose the bias and variance issues in your model
- Learn when and how to use regularization methods such as dropout or L2 regularization.
- Understand experimental issues in deep learning such as Vanishing or Exploding gradients and learn how to deal with them
- Use gradient checking to verify the correctness of your backpropagation implementation

Setting up your Machine Learning Application

Train/Dev/Test sets

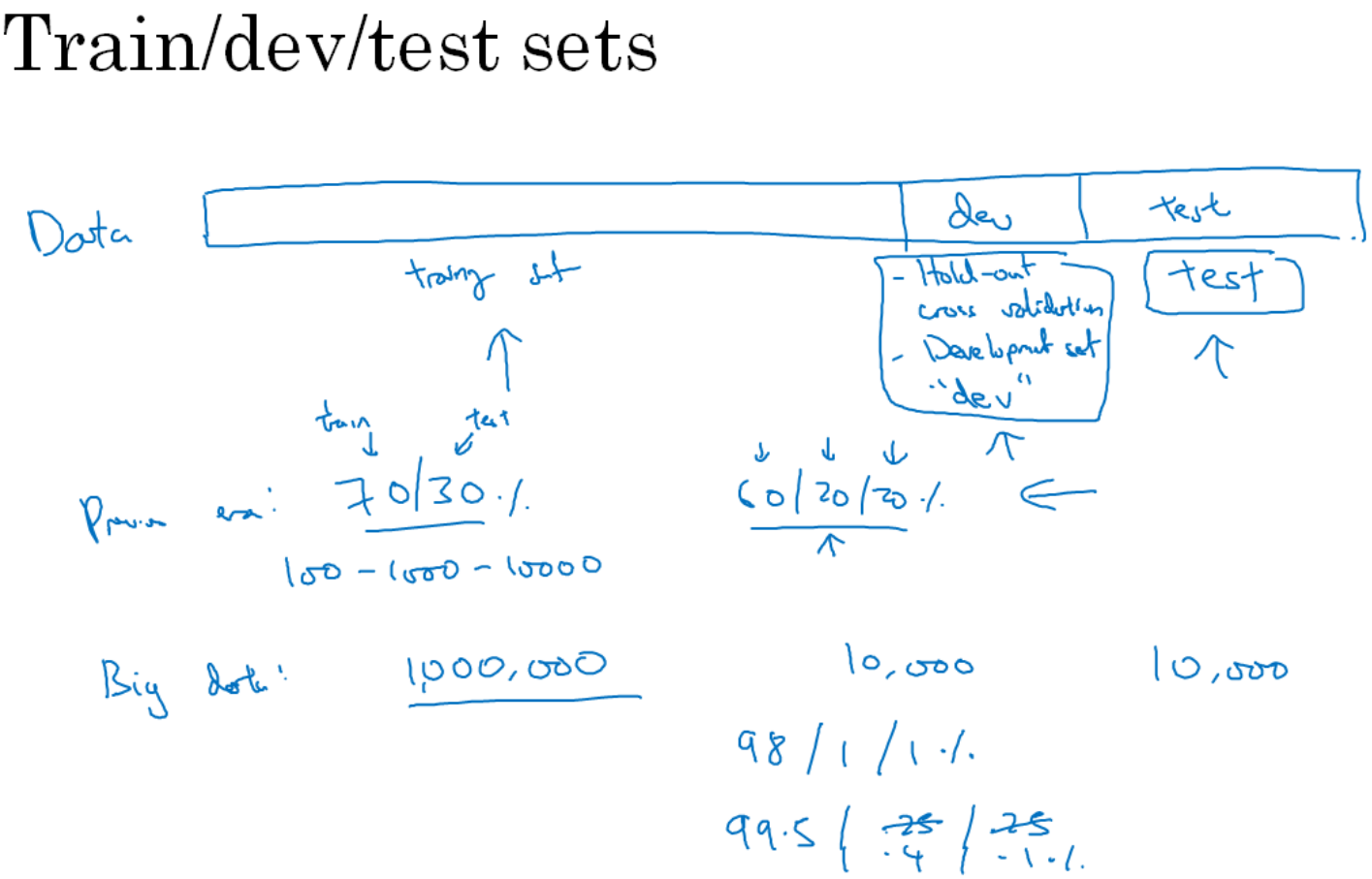
This course teaches about the practical aspects of deep learning. We just learned in last course how to implement a neural network. In this section we'll learn the practical aspects of how to make your neural network work well. Ranging from things like hyperparameter tuning to, how to set up your data, to how to make sure your optimization algorithm runs quickly so that you get your learning algorithm to learn in a reasonable time. In this section we'll first talk about the how to set a machine learning problem, then we'll talk about randomization. And we'll talk about some tricks for making sure your neural network implementation is correct.

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 rates? 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, 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 and 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 websites that wants 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 try to do something in computer vision. Or maybe a researcher with a lot of experience in speech

recognition might 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.

Let's see the below diagram:



So if this is your training data, then traditionally we 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 sets. 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 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 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 99.5% train and 0.25% dev, 0.25% test. Or maybe a 0.4% dev, 0.1% test.

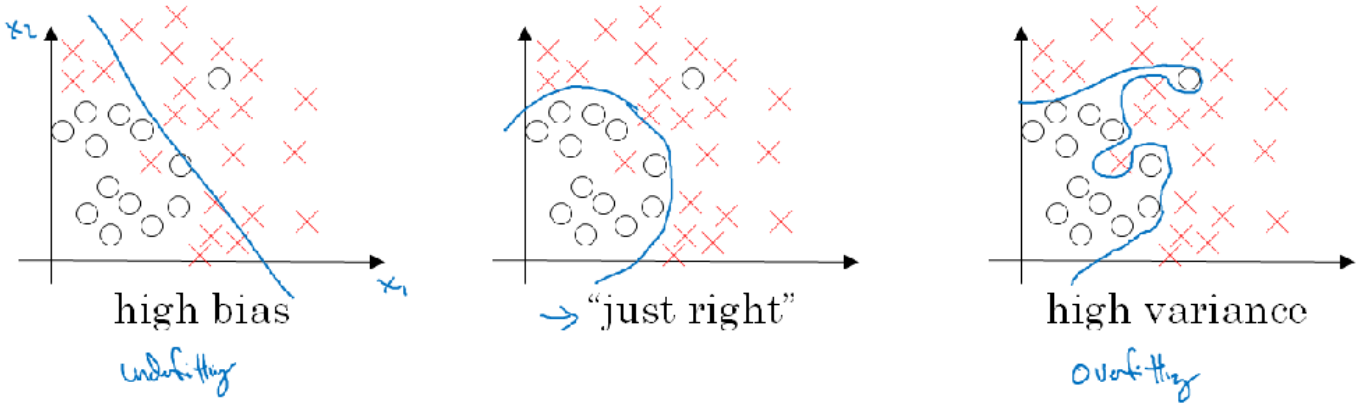
So just to recap, when setting up your machine learning problem, We'll often set it up into a train, dev and test sets, and if you have a relatively small dataset, these traditional ratios might be okay. But if you have a much larger data set, it's also fine to set your dev and test sets to be much smaller 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 our 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 blurrier, lower res images just taken with a cell phone camera in a more casual condition. And so these two distributions of data may be different. The rule of thumb I'd encourage you to follow in this case is to make sure that the **dev and test sets come from the same distribution**. We'll say more about this particular guideline as well, but because you will be using the dev set to evaluate a lot of different models and trying really hard to improve performance on the dev set. It's nice if your dev set comes from the same distribution as your test set. But because deep learning algorithms have such a huge hunger for training data, one trend I'm seeing is that you might use all sorts of creative tactics, such as crawling webpages, in order to acquire a much bigger training set than you would otherwise have. Even if part of the cost of that is then that your training set data might not come from the same distribution as your dev and test sets. But you find that so long as you follow this rule of thumb, that progress in your machine learning algorithm will be faster.

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. So having set up a train dev and test set will allow you to integrate more quickly. It will also allow you to more efficiently measure the bias and variance of your algorithm so you can more efficiently select ways to improve your algorithm.

Bias/Variance

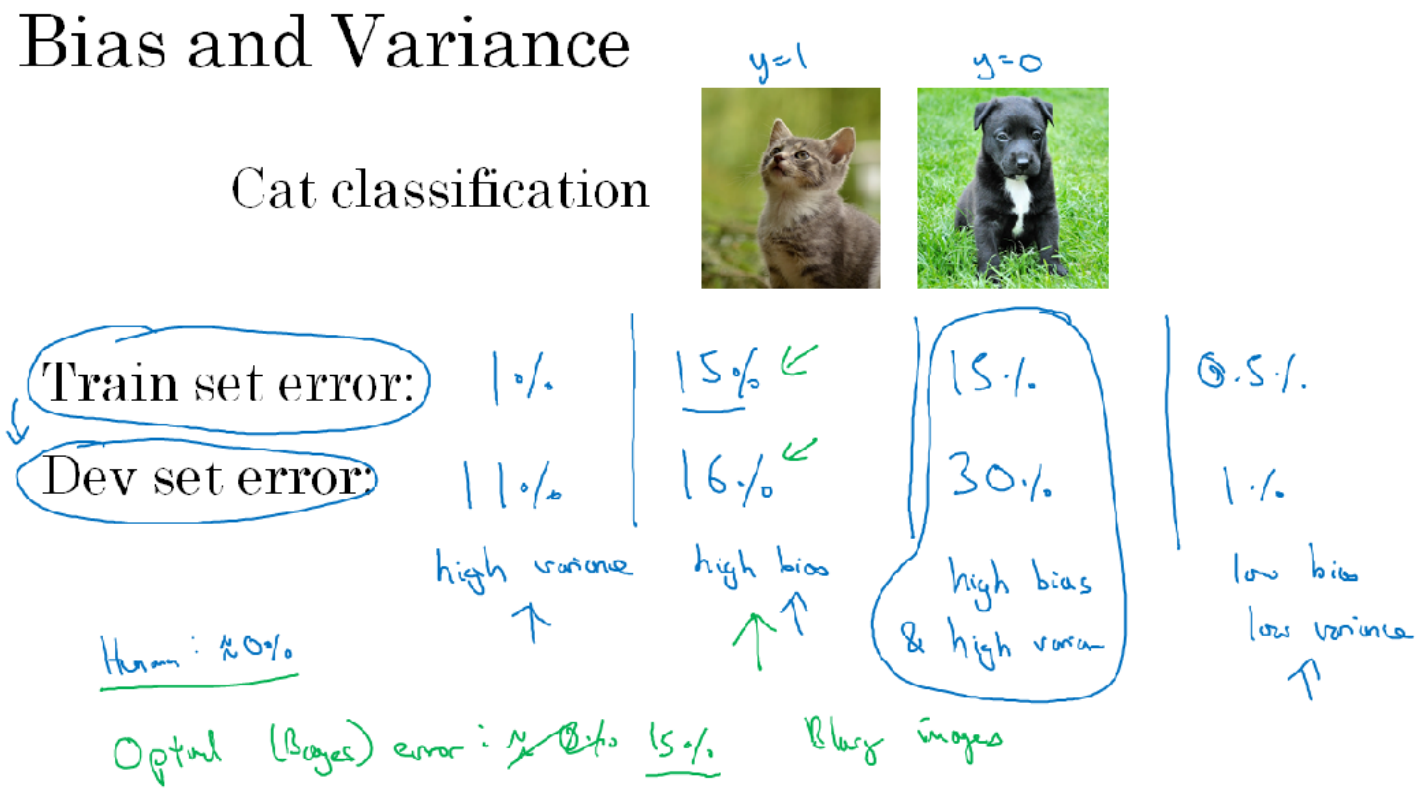
Bias and Variance is one of those concepts that's easily learned but difficult to master. Even if you think you've seen the basic concepts of Bias and Variance, there's often more new ones to it than you'd expect. In the Deep Learning era, another trend is that there's been less discussion of what's called the bias-variance trade-off. You might have heard this thing called the **bias-variance trade-off** but in Deep Learning era there's less of a trade-off, so we talk about the bias, we also talk about the variance but we just talk less about the **bias-variance trade-off**. Let's see what this means. Check the below diagram:

Bias and Variance



In first diagram, if you fit a straight line to the data, maybe you get a logistic regression fit to that. This is not a very good fit to the data and so this is class of a **high bias**, what we say that this is **underfitting** the data. On the opposite end (see 3rd diagram), if you fit an incredibly complex classifier, maybe a deep neural network, or neural network with all the hidden units, maybe you can fit the data perfectly, but that doesn't look like a great fit either. So there's a classifier of **high variance** and this is **overfitting** the data and there might be some classifier in between, with a medium level of complexity, that maybe fits it correctly like (see diagram 2). That looks like a much more reasonable fit to the data, so we call that just right. It's somewhere in between. So in a 2D example like this, with just two features, x1 and x2, we can plot the data and visualize bias and variance. In high dimensional problems, we can't plot the data and visualize decision boundary. Instead, there are couple of different metrics, which helps in understanding bias and variance.

So continuing our example of cat picture classification (check below diagram)



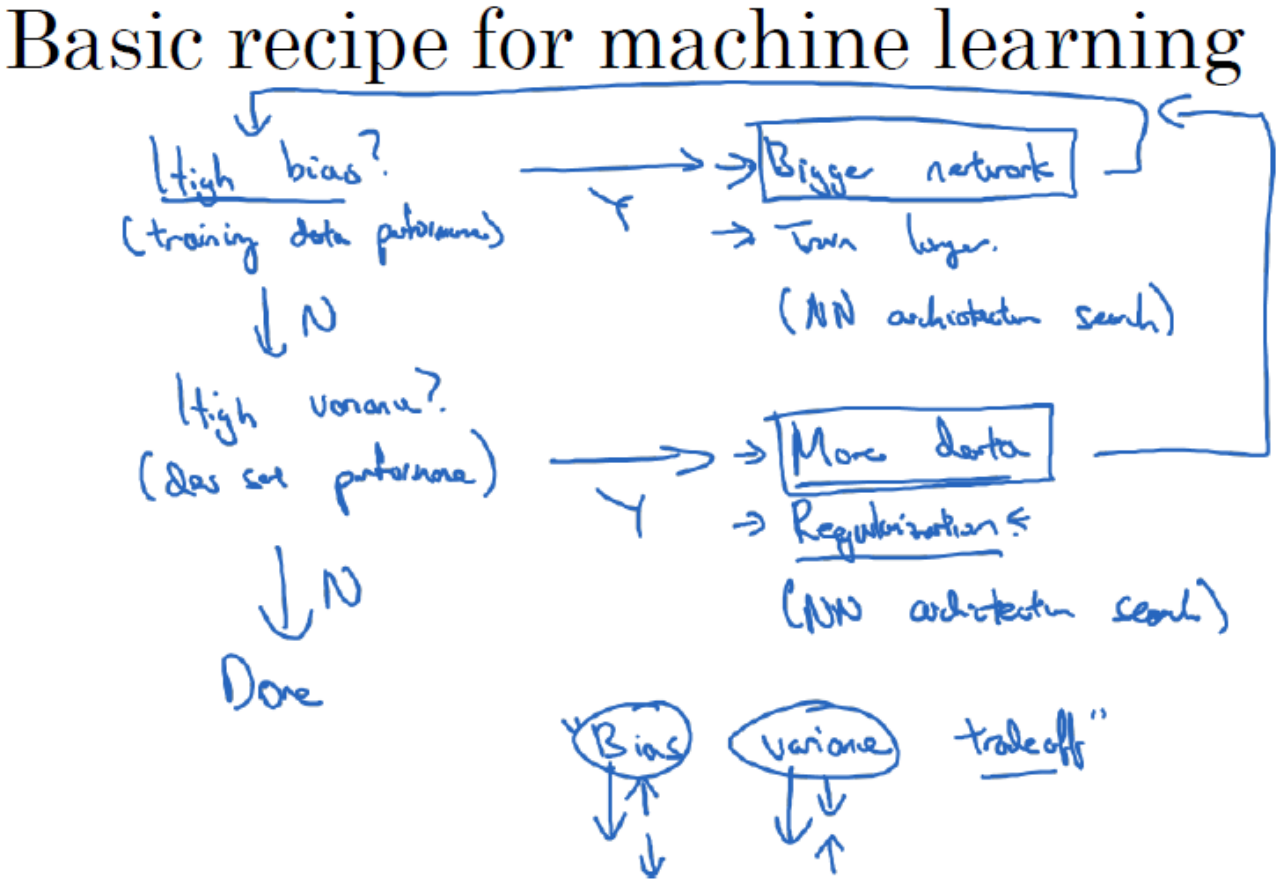
where found a cat is an positive example and found dog is a negative example, the two key numbers to look at to understand bias and variance will be the **train set error** and the **dev set** or the **development set error**. So for the sake of argument, let's say that you're recognizing cats in pictures, is something that people can do

nearly perfectly, right? So let's say, your training set error is 1% and your **dev set error** is, for the sake of argument, let's say is 11%. So in this example, you're doing very well on the training set, but you're doing relatively poorly on the development set. So this looks like you might have **overfit** the training set, that somehow you're not generalizing well, to this whole cross-validation set which is a development set. And so if you have an example like this, we would say this has **high variance**. So by looking at the training set error and the development set error, you would be able to render a diagnosis of your algorithm having high variance. Now, let's say, that you measure your training set and your dev set error, and you get a different result. Let's say, that your training set error is 15%. I'm writing your training set error in the top row, and your dev set error is 16%. In this case, assuming that humans achieve roughly 0% error, that humans can look at these pictures and just tell if it's cat or not, then it looks like the algorithm is not even doing very well on the training set. So if it's not even fitting the training data seam that well, then this is **underfitting** the data so this algorithm has high bias. But in contrast, this actually generalizing at a reasonable level to the dev set, whereas performance in the dev set is only 1% worse than performance in the training set. So this algorithm has a problem of high bias, because it was not even fitting the training set. Well, this is similar to the leftmost plots we had on the previous diagram.

Now, here's another example. Let's say that you have **15%** training set error, so that's pretty high bias, but when you evaluate to the dev set it does even worse, maybe it does **30%**. In this case, I would diagnose this algorithm as having high bias, because it's not doing that well on the training set, and high variance. So this has really the worst of both worlds. And one last example, if you have **0.5% training set error**, and **1% dev set error**, then maybe our users are quite happy, that you have a cat classifier with only 1% error, than just we have low bias and low variance. One subtle thing to mention is that this analysis is predicated on the assumption, that human level performance gets nearly **0%** error or, more generally, that the optimal error, sometimes called **bayes error**, so the base in optimal error is nearly 0%. It turns out that if the **optimal error** or the **bayes error** were much higher, say, it were **15%**, then if you look at this classifier, **15%** is actually perfectly reasonable for training set and you wouldn't see it as high bias and also a pretty low variance. So the case of how to analyze bias and variance, when no classifier can do very well, for example, if you have really blurry images, so that even a human or just no system could possibly do very well, then maybe bayes error is much higher, and then there are some details of how this analysis will change. But leaving aside this subtlety for now, the takeaway is that by looking at your training set error you can get a sense of how well you are fitting, at least the training data, and so that tells you if you have a bias problem. And then looking at how much higher your error goes, when you go from the training set to the dev set, that should give you a sense of how bad is the variance problem, so you'll be doing a good job generalizing from a training set to the dev set, that gives you sense of your variance. All this is under the assumption that the bayes error is quite small and that your training and your dev sets are drawn from the same distribution. So to summarize, we've seen how by looking at your algorithm's error on the training set and your algorithm's error on the dev set you can try to diagnose, whether it has problems of high bias or high variance, or maybe both, or maybe neither. And depending on whether your algorithm suffers from bias or variance, it turns out that there are different things you could try.

Basic recipe for machine learning

In the previous section, we saw how looking at training error and dev error can help you diagnose whether your algorithm has a bias or a variance problem, or maybe both. It turns out that this information that lets you much more systematically using what they call a **basic recipe for machine learning** and lets you much more systematically go about improving your algorithms' performance. Let's take a look.



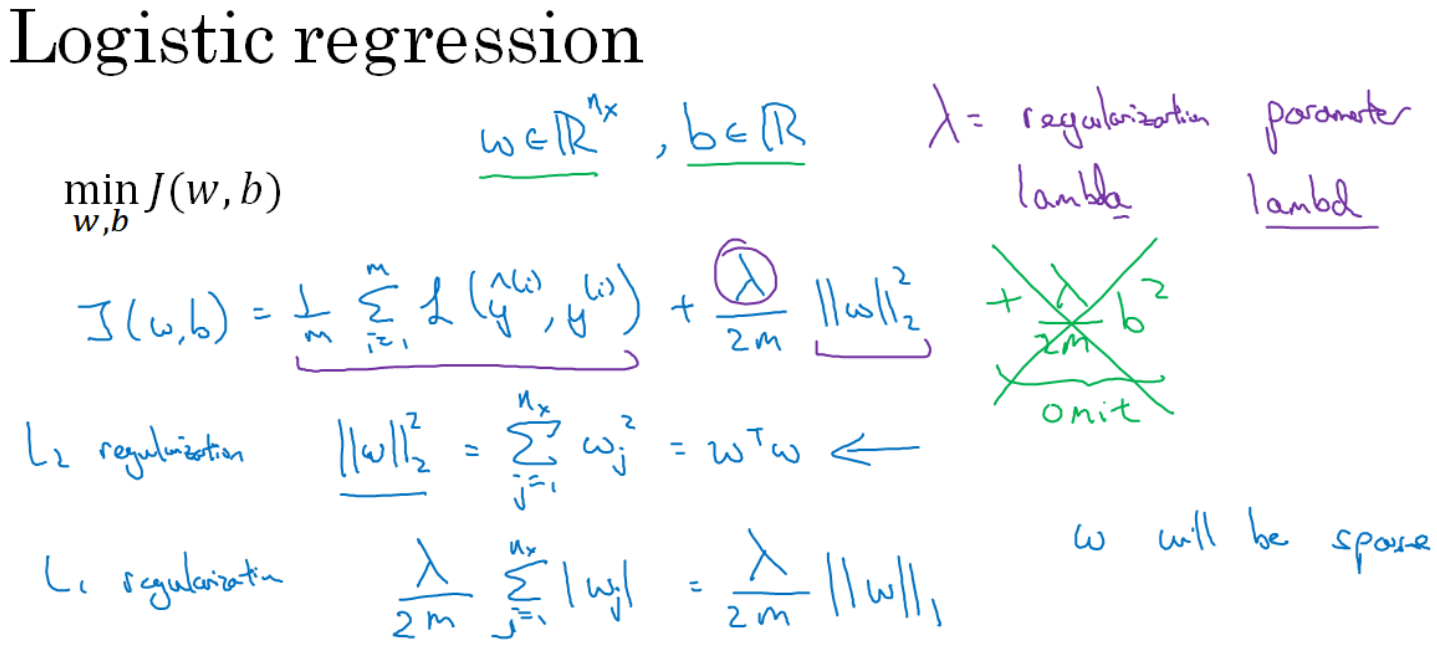
When training a neural network, here's a basic recipe we will use. After having trained an initial model, we will first ask, does our algorithm have high bias? And so to try and evaluate if there is high bias, you should look at, really, the training set or the training data performance and so, if it does have high bias, does not even fit 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. Maybe run trains longer or try some more advanced optimization algorithms,** we'll talk about optimization in upcoming sections or you can also try, not sure if works or not try but we can try a lot of different neural network architectures and maybe you can find a new network architecture that's better suited for this problem but there is no gurantee that this step will work but can be tried out. **Whereas getting a bigger network almost always helps and training longer doesn't always help, but it certainly never hurts**. So when training a learning algorithm, we should try these things until we can at least get rid of the bias problems, as in go back after we've tried this and keep doing that until we can fit, at least, fit the training set pretty well. And usually if you have a big enough network, you should usually be able to fit the training data well so long as it's a problem that is possible for someone to do. If the image is very blurry, it may be impossible to fit it. But if at least a human can do well on the task, if you think base error is not too high, then by training a big enough network you should be able to, hopefully, do well, at least on the training set. To at least fit or overfit the training set. Once you reduce bias to acceptable amounts then ask, do you have a variance problem? And so to evaluate that we would look at **dev set performance.** Are you able to generalize from a pretty good training set performance to having a pretty good dev set performance? And if you have high variance, well, best way to solve a **high variance problem is to get more data but** sometimes you can't get more data Or we can try regularization, which we'll talk about in the next section, to try to **reduce overfitting** but if you can find a more appropriate neural network architecture, sometimes that can reduce your variance problem as well, as well as reduce your bias problem. But how to do that? It's harder to be totally systematic how you do that. But so we can try these things and I kind of keep going back, until hopefully you find something with both low bias and low variance, where upon you would be done. So a couple of points to notice. First is that, depending on whether you have high bias or high variance, the set of things you should try could be quite different. So I'll usually use the training dev set to try to diagnose if you have a bias or variance problem, and then use that to select the appropriate subset of things to try. So for example, **if you actually have a high bias problem, getting more training data is actually not going to help.** Or at least it's not the most efficient thing to do. So being clear on how much of a bias problem or variance problem or both can help you focus on selecting the most useful things to try. Second, in the earlier era of machine learning, there used to be a lot of discussion on what is called the **bias variance**

tradeoff and the reason for that was that, for a lot of the things you could try, we can increase bias and reduce variance, or reduce bias and increase variance but back in the pre-deep learning era, we didn't have many tools, we didn't have as many tools that just reduce bias or that just reduce variance without hurting the other one. but in the modern deep learning, big data era, so long as you can keep training a bigger network, and so long as you can keep getting more data, which isn't always the case for either of these but if that's the case, then **getting a bigger network almost always just reduces your bias without necessarily hurting your variance, so long as you regularize appropriately and getting more data pretty much always reduces your variance and doesn't hurt your bias much**. So what's really happened is that, with these two steps, the ability to train, pick a network, or get more data, we now have tools to drive down bias and just drive down bias, or drive down variance and just drive down variance, without really hurting the other thing that much and I think this has been one of the big reasons that deep learning has been so useful for supervised learning, that there's much less of this tradeoff where you have to carefully balance bias and variance, but sometimes you just have more options for reducing bias or reducing variance without necessarily increasing the other one. And, in fact you have a well regularized network. We'll talk about regularization starting from the next section. Training a bigger network almost never hurts. And the main cost of training a neural network that's too big is just computational time, so long as you're regularizing. So I hope this gives you a sense of the basic structure of how to organize your machine learning problem to diagnose bias and variance, and then try to select the right operation for you to make progress on your problem. One of the things I mentioned several times in the section is **regularization**, is a **very useful technique for reducing variance**. There is a little bit of a bias variance tradeoff when you use regularization. It might increase the bias a little bit, although often not too much if you have a huge enough network.

Regularizing your neural network

Regularization

If you suspect your neural network is **over-fitting** your data this means that you have a high variance problem, one of the first things you should try is 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 the errors in your network. So let's see how regularization works. Let's develop these ideas using **logistic regression**.

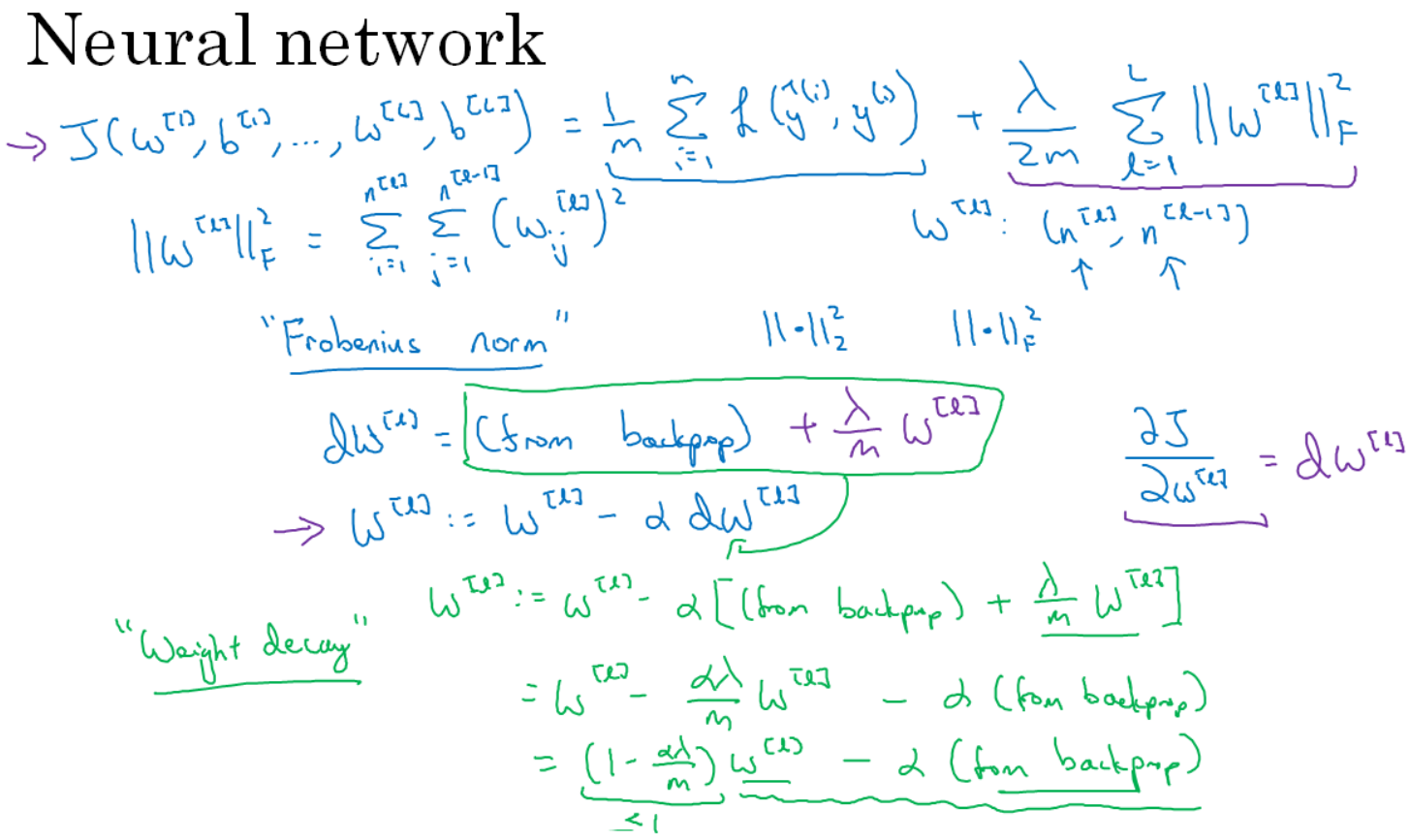


Recall that for logistic regression, we were trying to minimize the **cost function J**. Just to recall w and b in the logistic regression, are the parameters. So w is an x-dimensional parameter vector, and b is a real number. And so to add regularization to the logistic regression, what you do is add a thing called, **lambda**, which is called the **regularization parameter**. Check diagram for formula of regularization which is also called **L2 regularization**.

Now, why do you regularize just the parameter w? Why don't we add something here 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 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/m of sum over of this. And this is also called the L1 norm of the parameter vector w, 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, and 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. So just to add to notation ,**Lambda here is called the regularization, Parameter** and usually, you set this using **your development set, or using cross validation**. When you 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.

So this is how you implement L2 regularization for logistic regression. How about a neural network?



In a neural network, we have a cost function that's a function of all of our parameters, $w^{[1]}$, $b^{[1]}$ through $w^{[L]}$, $b^{[L]}$, where capital L is the number of layers in your neural network. Check above diagram for formula. So **the cost function, sum of the losses, summed over your m training examples. And says at 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, meaning the squared norm is defined as the sum of the i sum of j, of each of the elements of that matrix, squared.? Previously, we would complete **dw 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 $w^{[1]}$ - the learning rate times d. Check the diagram above. **L2 regularization** is sometimes also called **weight decay** because it's just like the ordinally gradient descent, where you update w by subtracting alpha times the original gradient you got from **backprop**.

Why regularization reduces overfitting?

Why does regularization help with overfitting? Why does it help with reducing variance problems? Few methods we can try:

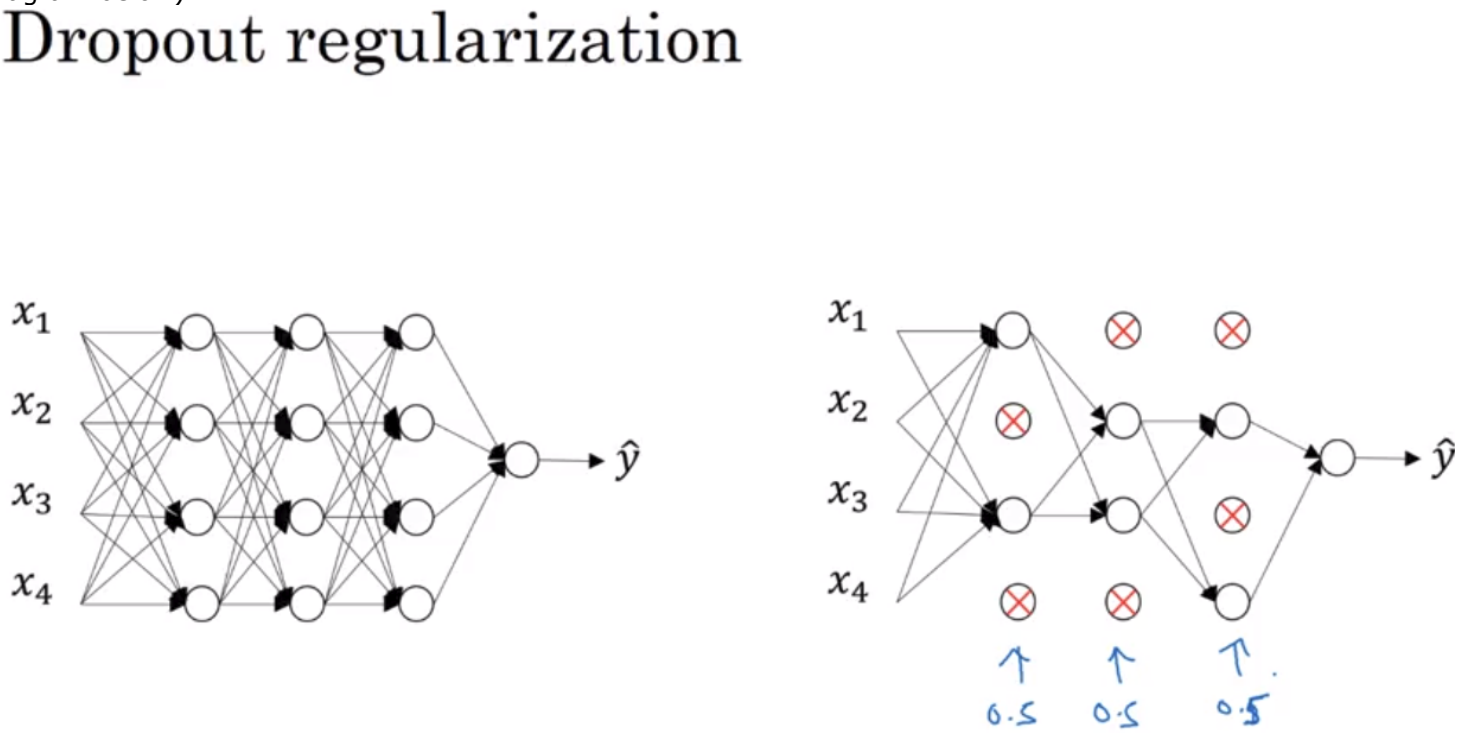
- Set the weight matrices W to be **reasonably close to zero**. So one piece of intuition is maybe it set the weight to be so close to zero for a lot of hidden units that's basically zeroing out a lot of the impact of these hidden units. And if that's the case, then this much simplified neural network becomes a much

smaller neural network. In fact, it is almost like a logistic regression unit, but stacked most probably as deep. And so that will take you from this overfitting case much closer to the left to other high bias case.

- Another trial we can do by using the **tanh**. If the regularization becomes very large, the parameters W very small, so Z will be relatively small, kind of ignoring the effects of b for now, so Z will be relatively small or, really it takes on a small range of values. And so the activation function if is **tanh**, 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**.

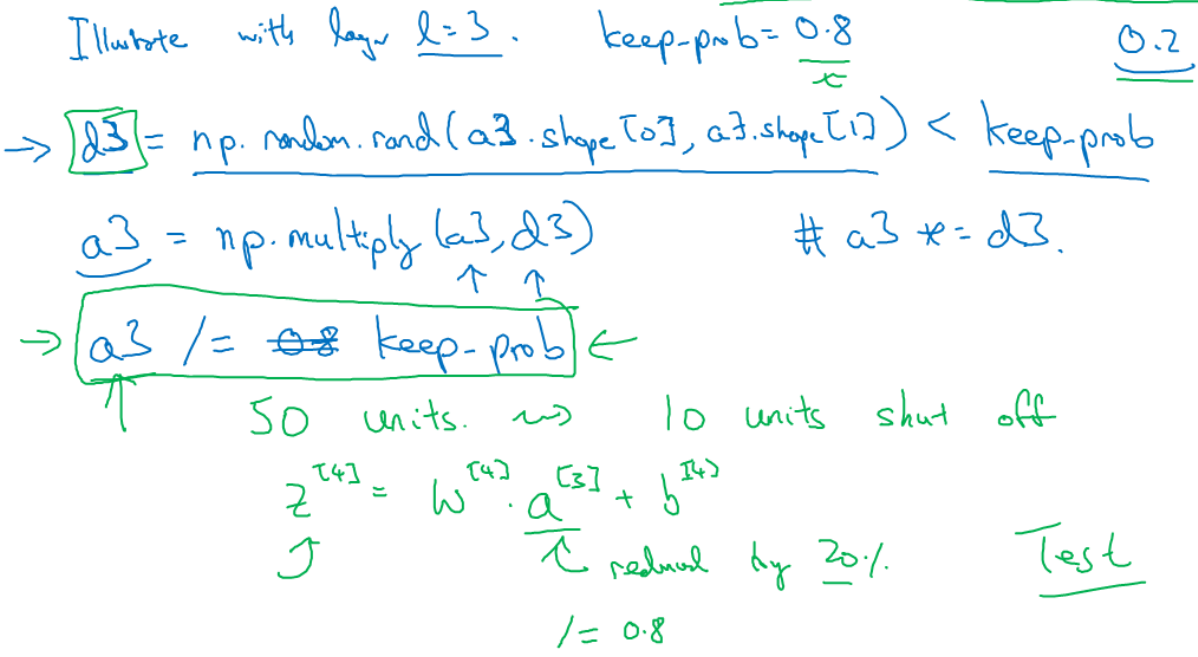
Dropout regularization

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. (Check the diagram below)



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. They just go around coding those are random, but this **actually works**. But you can imagine that because you're training a much smaller network on each example or maybe just give a sense for why you end up able to regularize the network, because these much smaller networks are being trained. Let's look at how you implement dropout. There are a few ways of implementing dropout. the most common one, which is technique called **inverted dropout**. Check diagram below:

Implementing dropout ("Inverted dropout")



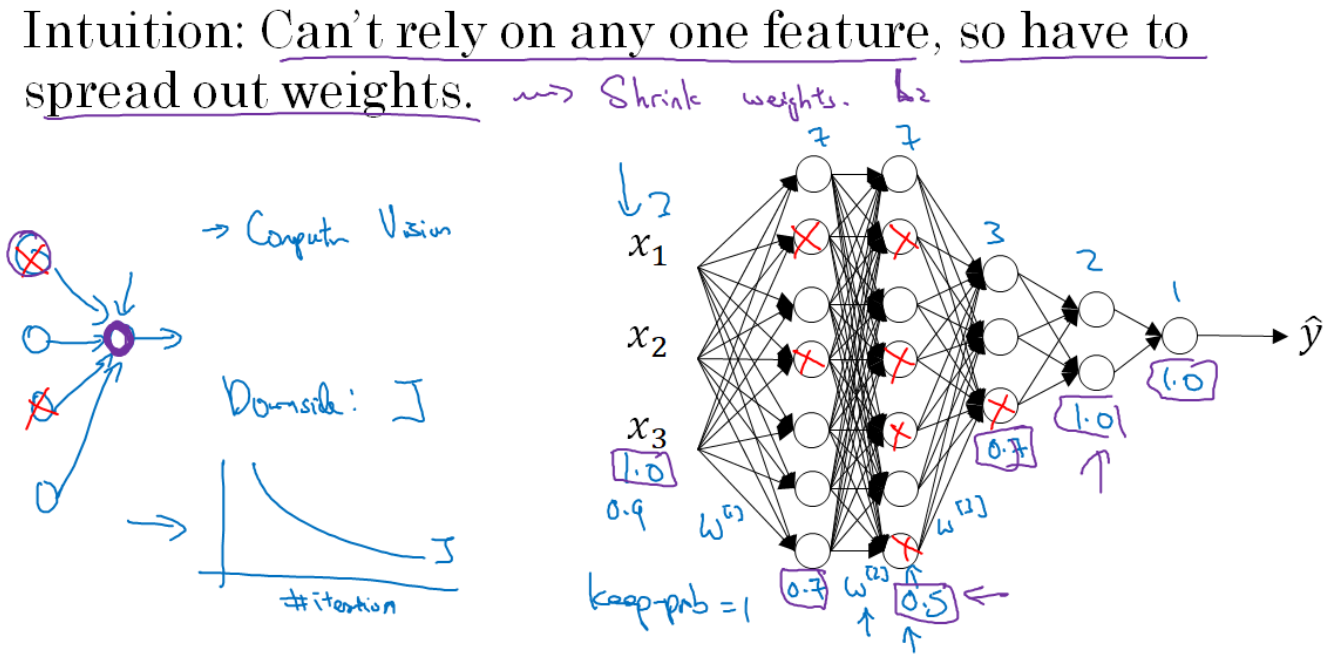
The last line is what's called the **inverted dropout technique** and its effect is that, no matter what you set to keep.prob to, whether it's 0.8 or 0.9 or even one, if it's set to one (1) then there's no dropout, because it's keeping everything or 0.5 or whatever, this **inverted dropout technique by dividing by the keep.prob**, it ensures that the expected value of a3 remains the same. And it turns out that at test time, when you trying to evaluate a neural network, this inverted dropout technique makes test time easier because you have less of a scaling problem. By far the most common implementation of dropouts today as far as I know is inverted dropouts.

Understanding Dropout

Drop out does this seemingly crazy thing of randomly knocking out units on your network. Why does it work so well with a regularizer? Let's gain some better intuition. In the previous section, we saw the intuition that drop-out 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 neural network seems like it should have a **regularizing effect**.

Why does drop-out work?



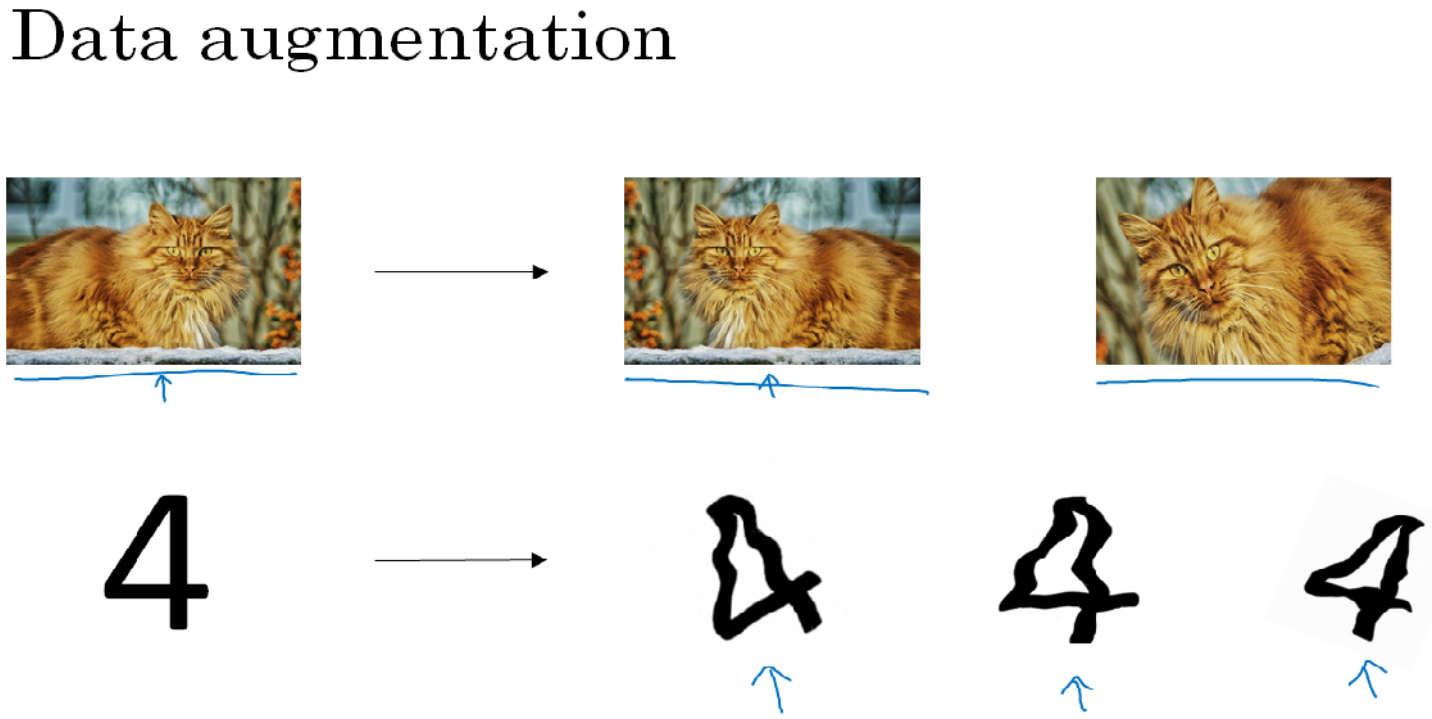
Here's a **Second intuition** which is, let's look at it from the perspective of a single unit. Let's take one unit, now, for this unit to do his job as for inputs and it needs to generate some meaningful output. Now with drop out, the inputs can get randomly eliminated. Sometimes other two units will get eliminated, sometimes a different unit will get eliminated. So, if we take example of a unit in diagram, it can't rely on any one feature because any one feature could go away at random or any one of its own inputs could go away at random. Some one would be reluctant to put all of its bets on, say, just one input, right? The weights, we're reluctant to put too much weight on any one input because it can go away. So this unit will be more motivated to spread out this way and give you a little bit of weight to each of the four inputs to this unit. And by spreading all the weights, 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 **drop out is that it shrinks the weights and does some of those outer regularization that helps prevent over-fitting.** 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.

One more detail for when you're implementing drop out. See the diagram where we have three input features. This is seven hidden units here, seven, three, two, one. So, one of the parameters we had to choose was the **keep-prob** which has a chance of keeping a unit in each layer. So, it is also feasible to vary **keep-prob** by layer so there could be different **keep-prob** for different layers. Notice that the **keep-prob** of 1.0 means that you're keeping every unit and so, you're really not using drop out 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 the **keep-prob** to be smaller to apply a more powerful form of drop out. It's kind of like cranking up the regularization parameter **lambda of L2 regularization** where you try to regularize some layers more than others. And technically, you can also apply drop out 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 that often. And so, a key prop of 1.0 was quite common for the input there. You can also use a very high value, maybe 0.9, but it's much less likely that you want to eliminate half of the input features. **So just to summarize**, if you're more worried about some layers overfitting 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 drop out and some layers where you don't apply drop out and then just have one hyper parameter, which is a **keep-prob** for the layers for which you do apply drop outs. Some implementational tips-> Many of the first successful implementations of drop outs were to **computer vision**. So in computer vision, the input size is so big, inputting all these pixels that you almost never have enough data. And so drop out is very frequently used by computer vision. And there's some **computer vision researchers 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 over-fitting**. And so, unless my algorithm is over-fitting, I

wouldn't actually bother to use drop out. So it's used somewhat less often than other application areas. There's 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 drop out. But their intuition 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 are randomly killing off a bunch of nodes. And so, if you are double checking the performance of grade descent, it's actually harder to double check that you have a well defined **cost function J** that is going downhill on every iteration. Because the cost function J that you're optimizing is actually less. Less well defined, or is certainly hard to calculate. So what we can try usually is to turn off drop out, means **keep-prob = 1.0** and then run the code code and make sure that it is **monotonically decreasing J**, and then turn on drop out and hope that I didn't introduce bugs into my code during drop out.

Other regularization methods

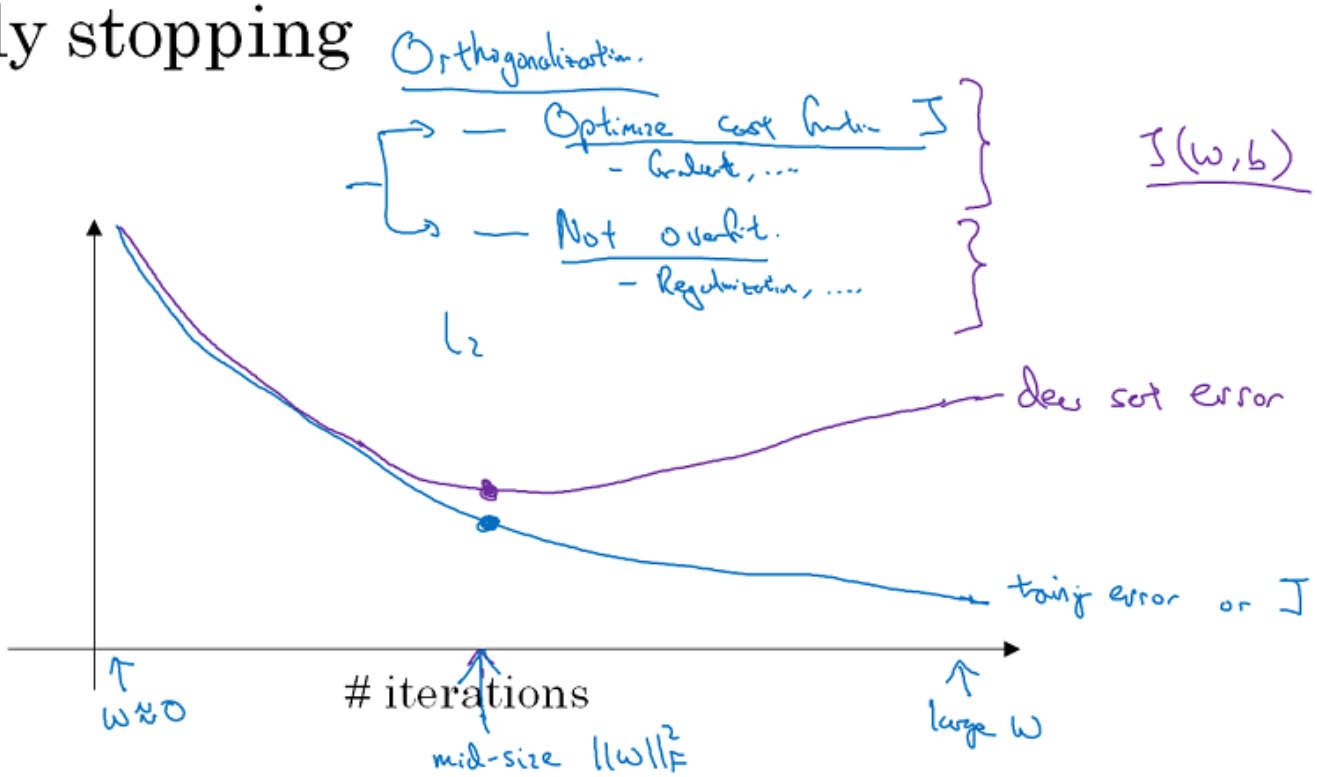
In addition to **L2 regularization** and **drop out regularization** there are few other techniques to reducing over fitting in your neural network. Let's take a look. Let's say you fitting a cat classifier, If you are **over fitting getting more training data can help**, but getting more training data can be expensive and sometimes you just can't get more data. But what you can do is **augment your training set** (Check the image below).



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 confrontational 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 we 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 force. So data augmentation can be used as a **regularization technique**, in fact similar to regularization.

There's one other technique that is often used called **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 0/1 classification error on the training set. Or just plot the cost function J optimizing, and that should decrease monotonically, so with **early stopping**, we plot like shown below:

Early stopping

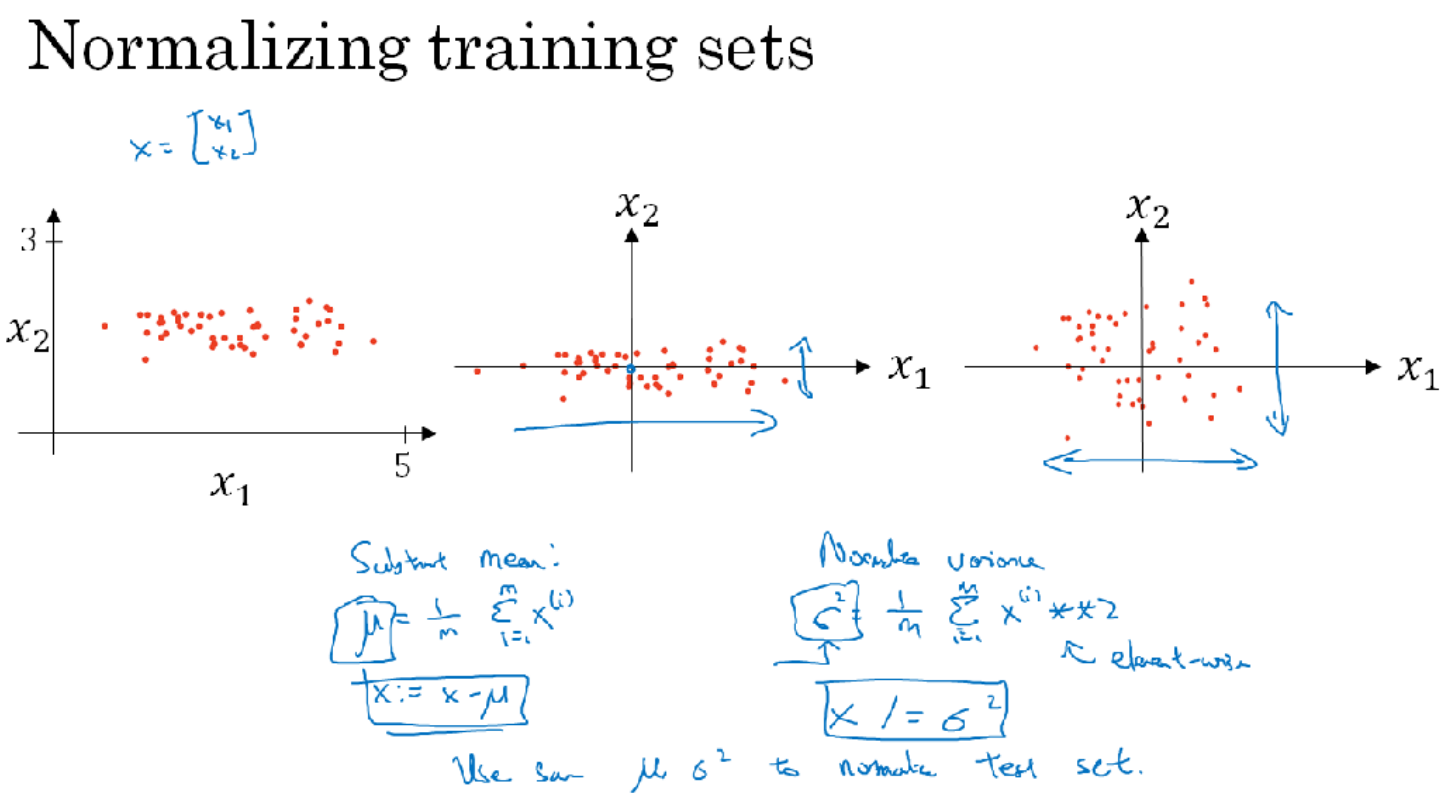


And again, this could be a classification error in a development sense, or something like the cost function, like the logistic loss or the log loss of the dev set. Now what you find is that your dev set error will usually go down for a while, and then it will increase from there.

So what early stopping does is that that look for the iteration where NN is doing best and stop trading on neural network halfway and take whatever value achieved this dev set error. So why does this work? Well when we've haven't run many iterations for our neural network yet our 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.** There is a downside of early stopping too, 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 **gradient descent** and then also 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 in 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 optimizing 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**. I'll say more about orthorganization in a later sections, so if you don't fully get the concept yet, don't worry about it. But, to me the main downside of early stopping is that this couples these two tasks. So you no longer can work on these two problems independently, because by stopping gradient decent early, you're sort of breaking whatever you're doing to optimize cost function J, because now you're not doing a great job reducing the cost function J. You've sort of not done that that well. And then you also simultaneously trying to not over fit. So instead of using different tools to solve the two problems, you're using one that kind of mixes the two. And this just makes the set of things you could try are more complicated to think about. **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.

Setting up your optimization problem
Normalizing inputs



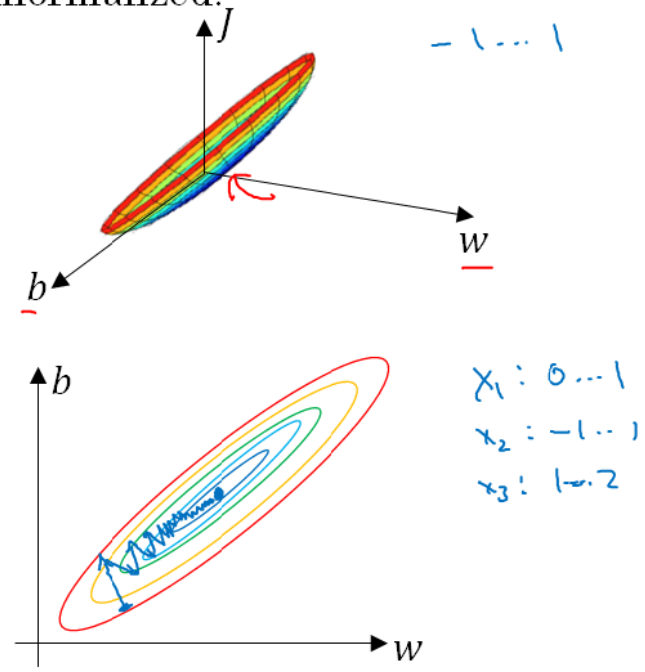
When training a neural network, one of the techniques that will speed up your training is if you normalize your inputs. Let's see what that means. Let's see if a training sets with two input features. So the input features x are two dimensional, and check the diagram for a scatter plot of our training set. Normalizing your inputs corresponds to two steps. The first is to subtract out or to zero out the mean. So you set mu = 1 over M sum over I of $X^{[i]}$. So this is a vector, and then X gets set as X- mu for every training example, so this means you just move the training set until it has 0 mean. And then the second step is to normalize the variances. So notice here that the feature X1 has a much larger variance than the feature X2 here. So what we do is set sigma = 1 over m sum of Xi**2. I guess this is a element y squaring. And so now sigma squared is a vector with the variances of each of the features, and notice we've already subtracted out the mean, so Xi squared, element y squared is just the variances. And you take each example and divide it by this vector sigma squared. And so in pictures, you end up with this. Where now the variance of X1 and X2 are both equal to one.
And one tip, if you use this to scale your training data, then use the same mu and sigma squared to normalize your test set, right? In particular, you don't want to normalize the training set and the test set differently. Whatever this value is and whatever this value is, use them in these two formulas so that you scale your test set in exactly the same way, rather than estimating mu and sigma squared separately on your training set and test set. Because you want your data, both training and test examples, to go through the same transformation defined by the same mu and sigma squared calculated on your training data. So, why do we do this? **Why do**

we want to normalize the input features?

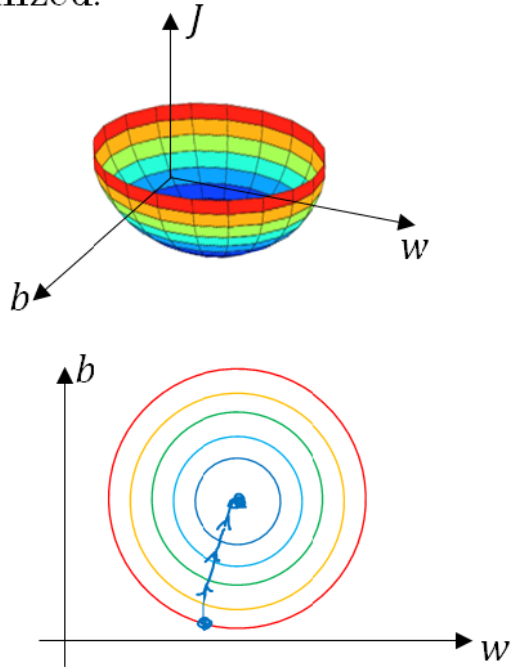
Why normalize inputs?

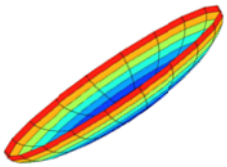
$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right)$$

Unnormalized:



Normalized:



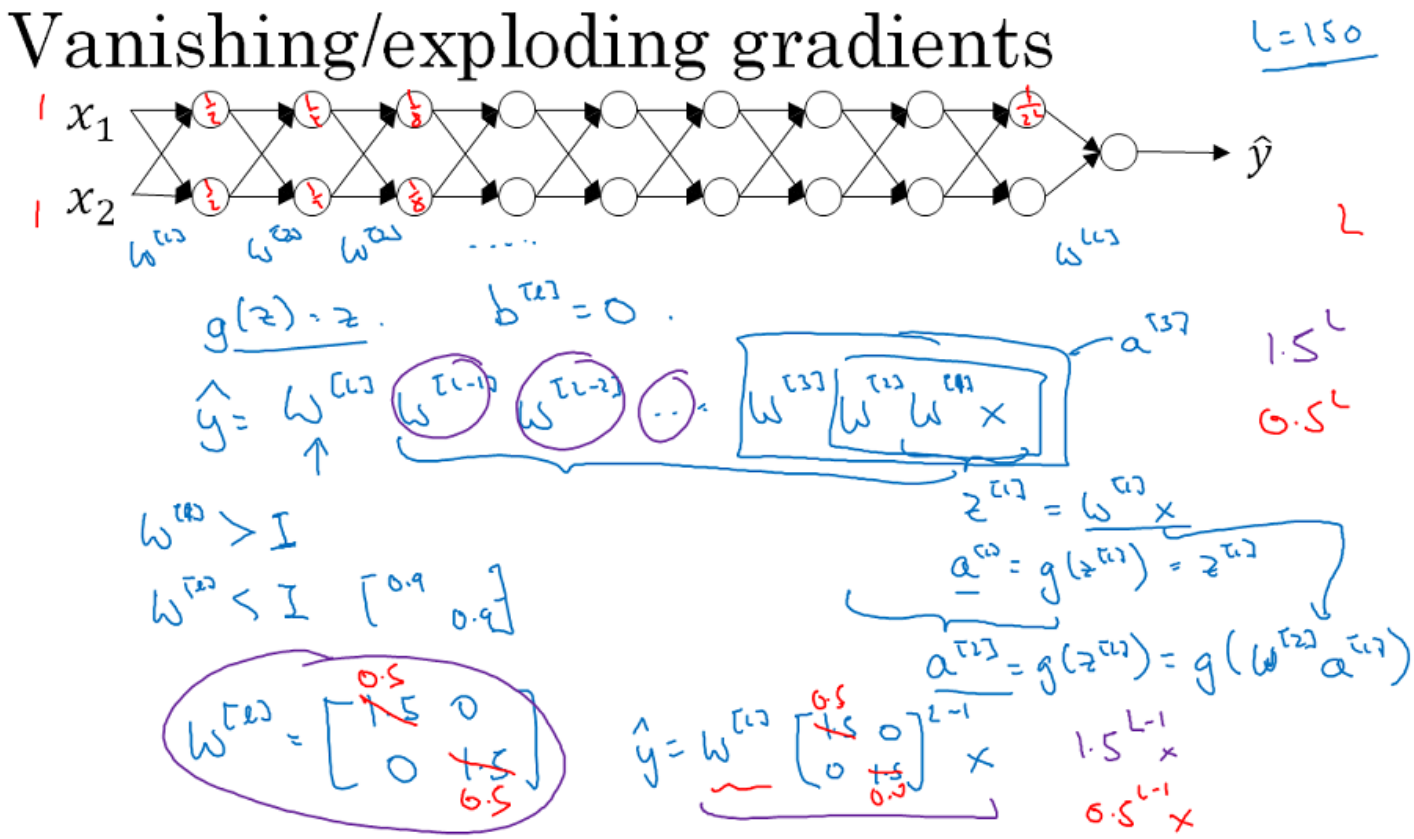


Recall that a cost function is defined as written on the top right. It turns out that if you use unnormalized input features, it's more likely that your cost function will look like this, it's a very squished out bowl, very elongated cost function, where the minimum you're trying to find is maybe over there. But if your features are on very different scales, say the feature X1 ranges from 1 to 1,000, and the feature X2 ranges from 0 to 1, then it turns out that the ratio or the range of values for the parameters w1 and w2 will end up taking on very different values. And so maybe these axes should be w1 and w2, but I'll plot w and b, then your cost function can be a very elongated bowl like that. So if you part the contours of this function, you can have a very elongated function like that. Whereas if you normalize the features, then your cost function will on average look more symmetric. And if you're running gradient descent on the cost function like the one on the left, then you might have to use a very small learning rate because if you're here that gradient descent might need a lot of steps to oscillate back and forth before it finally finds its way to the minimum. Whereas if you have a more spherical contours, then wherever you start gradient descent can pretty much go straight to the minimum. You can take much larger steps with gradient descent rather than needing to oscillate around like like the picture on the left. Of course in practice w is a high-dimensional vector, and so trying to plot this in 2D doesn't convey all the intuitions correctly. But the rough intuition that your cost function will be more round and easier to optimize when your features are all on similar scales. Not from one to 1000, zero to one, but mostly from minus one to one or of about similar variances of each other. **That just makes your cost function J easier and faster to optimize.** In practice if one feature, say X1, ranges from zero to one, and X2 ranges from minus one to one, and X3 ranges from one to two, these are fairly similar ranges, so this will work just fine. It's when they're on dramatically different ranges like ones from 1 to a 1000, and the another from 0 to 1, that that really hurts your optimization algorithm. But by just setting all of them to a 0 mean and say, variance 1, like we did in the last

part of --, that just guarantees that all your features on a similar scale and will usually help your learning algorithm run faster. So, if your input features came from very different scales, maybe some features are from 0 to 1, some from 1 to 1,000, then it's important to normalize your features. If your features came in on similar scales, then this step is less important. Although performing this type of normalization pretty much never does any harm, so I'll often do it anyway if I'm not sure whether or not it will help with speeding up training for your algebra.

Vanishing/exploding gradients

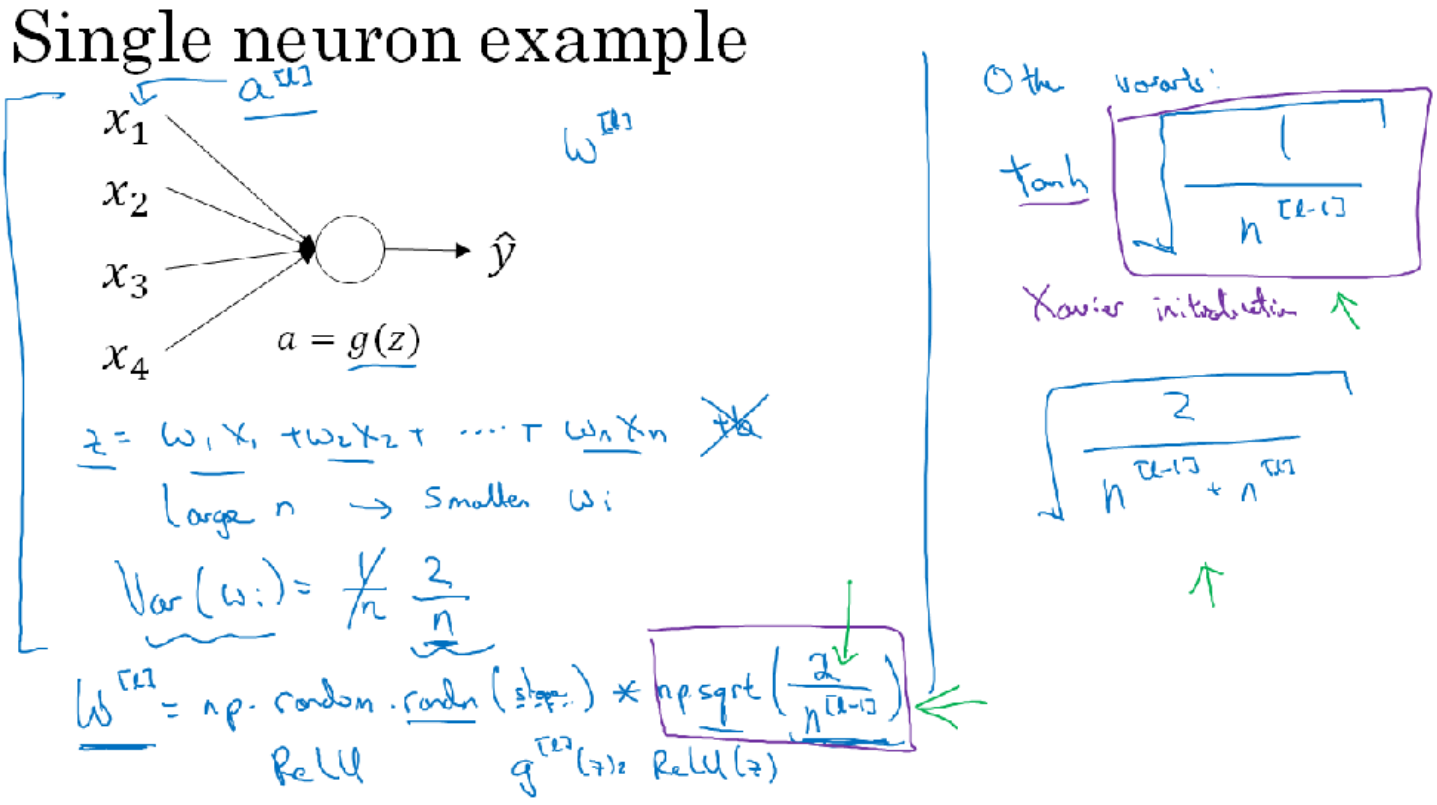
One of the problems of training neural network, especially very deep neural networks, is data **vanishing and exploding gradients**. What that means is that when you're training a very deep network your derivatives or your slopes can sometimes get either very, very big or very, very small, maybe even exponentially small, and this makes training difficult. In this section you will see what this problem of exploding and vanishing gradients really means, as well as how you can use careful choices of the **random weight initializatio**n to significantly reduce this problem. Check the diagram:



So the intuition you can take away from this is that at the weights W, if they're all just a **little bit bigger** than one or just a little bit bigger than the identity matrix, then with a very deep network the activations can explode. And if **W is just a little bit less** than identity. So this maybe here's 0.9, 0.9, then you have a very deep network, the activations will **decrease exponentially**.

Weight initialization for deep networks

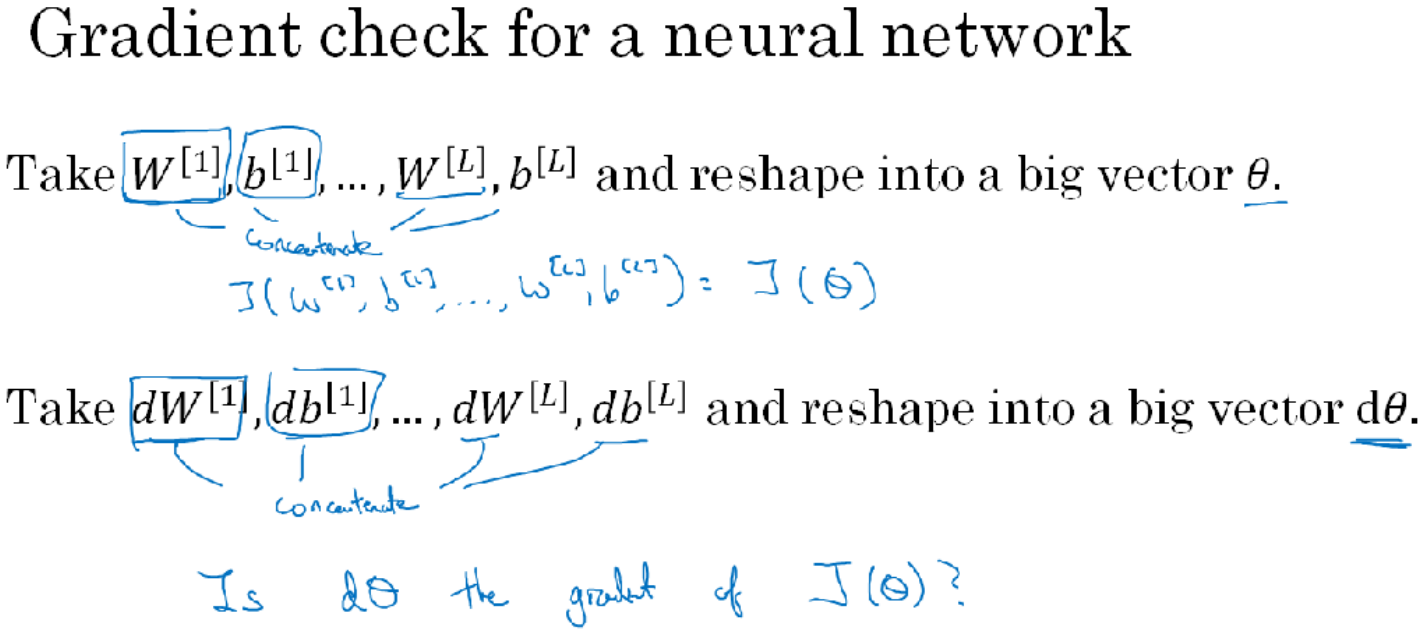
In the last section, we saw how very deep neural networks can have the problems of vanishing and exploding gradients. It turns out that a partial solution to this, doesn't solve it entirely but helps a lot, is better or more careful choice of the random initialization for your neural network. To understand this lets start with the example of initializing the ways for a **single neuron** and then we're going to generalize this to a deep network.



Looking into diagram we get some intuition about the problem of vanishing or exploding gradients as well as how choosing a **reasonable scaling for how you initialize the weights**. Hopefully that makes your weights not explode too quickly and not decay to zero too quickly so you can train a reasonably deep network without the weights or the gradients exploding or vanishing too much. When you train deep networks this is another trick that will help you make your neural networks trained much.

Gradient checking

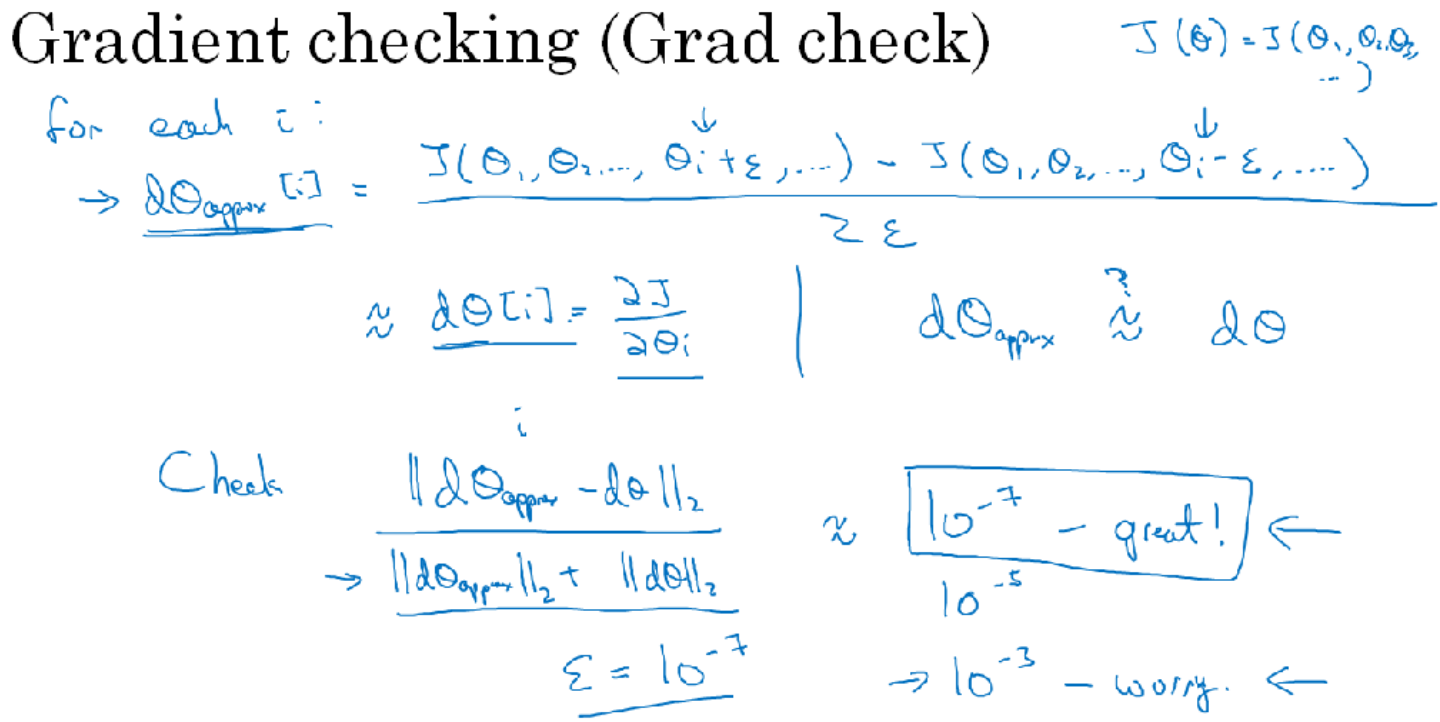
Gradient checking is a technique that can help in saving lots of time to find bugs in implementations of back propagation. In this section we'll see that how could we use it too to debug, or to verify that the implementation of back-prop is correct. Check the diagram below:



So your new network will have some sort of parameters, $W^{[1]}$, $b^{[1]}$ and so on up to $W^{[L]}$,$b^{[1]}$. So to implement gradient checking, the first thing you should do is take all your parameters and reshape them into a giant vector data. So what you should do is take **W which is a matrix, and reshape it into a vector**. You gotta take all of

these W's and reshape them into vectors, and then concatenate all of these things, so that you have a giant **vector theta**. Next, with W and b ordered the same way, you can also take $dW^{[1]}$, $db^{[1]}$ and so on, and initiate them into big, giant vector **d_theta** of the same dimension as **theta**. Remember, $dW^{[1]}$ has the same dimension as $W^{[1]}$ and $db^{[1]}$ has the same dimension as $b^{[1]}$. So the same sort of reshaping and concatenation operation, you can then reshape all of these derivatives into a giant vector **d_theta**. Which has the same dimension as theta.

So to implement grad check, what you're going to do is implements a loop so that for each i, so for each component of theta, let's compute **d_theta_approx**. Check the diagram below:

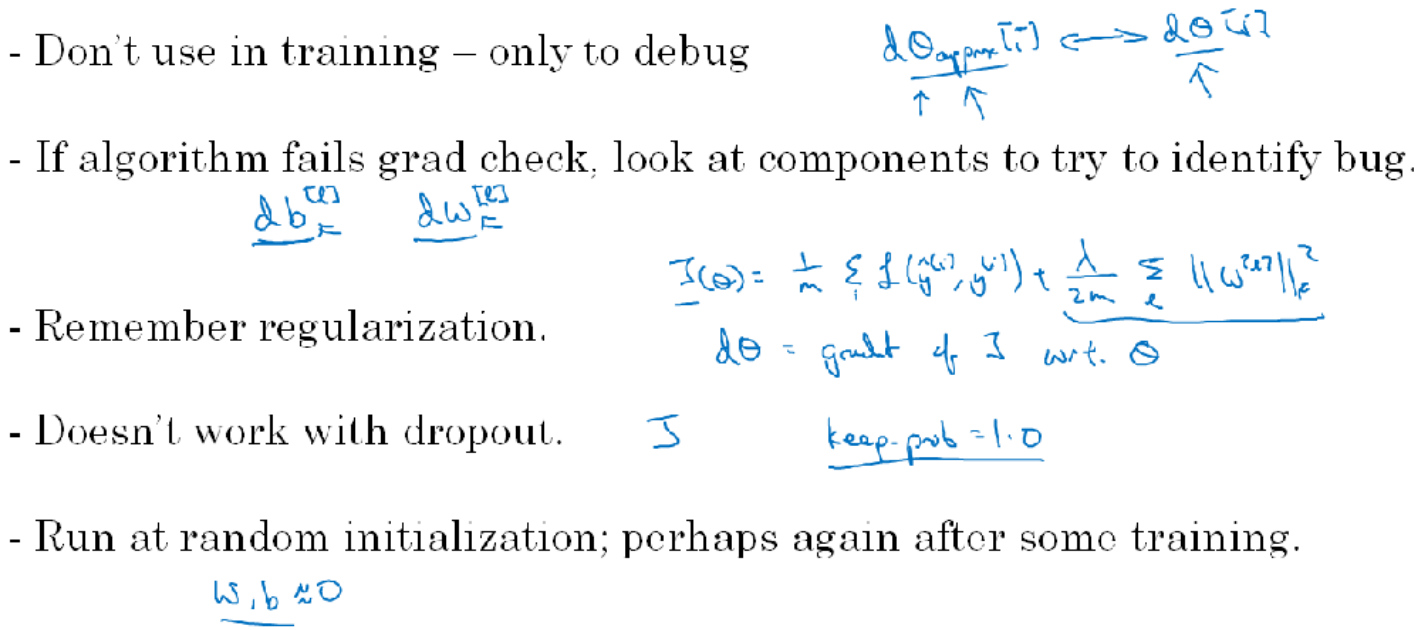


If the grad check has a relatively big value then there are great chances of having a bug, so go in debug, debug, debug and after debugging for a while, If we find that it passes grad check with a small value, then you can be much more confident that NN is correct.

Gradient checking implementation notes

This section we'll learn about some practical tips or notes on how to actually go about implementing grad-check for your neural network.

Gradient checking implementation notes



First, don't use grad check in training, only to debug. So what I mean is that, computing **d_theta_approx** i, for all the values of i, this is a very slow computation. So to implement gradient descent, you'd use backprop to compute d theta and just use backprop to compute the derivative. And it's only when you're debugging that you would compute this to make sure it's close to d theta. But once you've done that, then you would turn off the grad check, and don't run this during every iteration of gradient descent, because that's just much too slow. Second, if an algorithm fails grad check, look at the components, look at the individual components, and try to identify the bug. So what I mean by that is if d theta approx is very far from d theta, what I would do is look at the different values of i to see which are the values of d theta approx that are really very different than the values of d theta. So for example, if you find that the values of theta or **d_theta**, they're very far off, all correspond to dbl for some layer or for some layers, but the components for dw are quite close, right? Remember, different components of theta correspond to different components of b and w. When you find this is the case, then maybe you find that the bug is in how you're computing db, the derivative with respect to parameters b. And similarly, vice versa, if you find that the values that are very far, the values from d theta approx that are very far from d theta, you find all those components came from dw or from dw in a certain layer, then that might help you hone in on the location of the bug. This doesn't always let you identify the bug right away, but sometimes it helps you give you some guesses about where to track down the bug.

Next, when doing grad check, **remember your regularization** term if you're using regularization. Next, **grad check doesn't work with dropout, because in every iteration, dropout** is randomly eliminating different subsets of the hidden units. There isn't an easy to compute cost function J that dropout is doing gradient descent on. It turns out that dropout can be viewed as optimizing some cost function J, but it's cost function J defined by summing over all exponentially large subsets of nodes they could eliminate in any iteration. So the cost function J is very difficult to compute, and you're just sampling the cost function every time you eliminate different random subsets in those we use dropout. So it's difficult to use **grad check to double check your computation with dropouts**. So what we can usually do is implement grad check without dropout. So if you want, you can set **keep-prob and dropout to be equal to 1.0**. And then turn on dropout and hope that my implementation of dropout was correct. So the recommended setting is to turn off dropout, use grad check to double check that your algorithm is at least correct without dropout, and then turn on dropout.

Week 2: Optimization algorithms

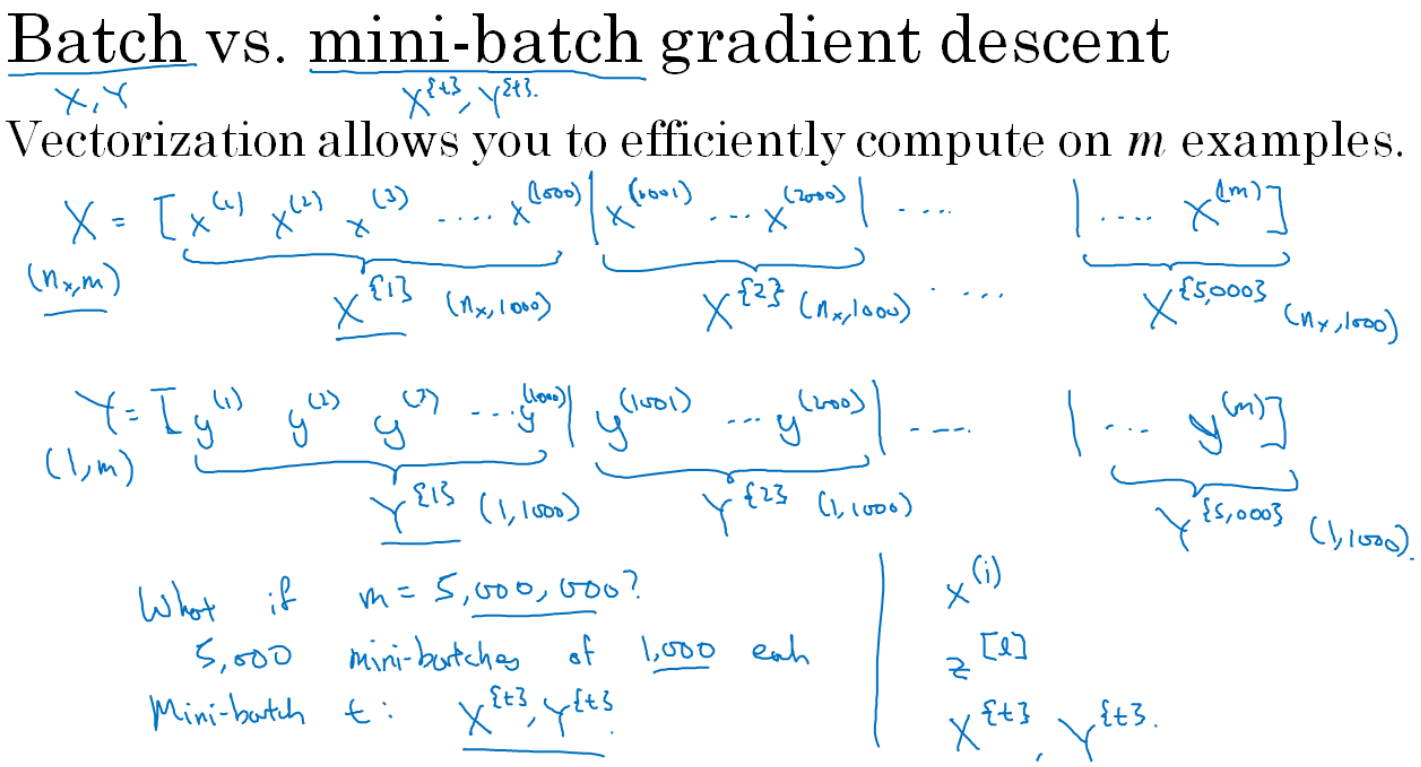
Learning Objectives

- Remember different optimization method such as (Stochastic) Gradient Descent, Momentum, RMSProp and Adam
- Use random minibatches to accelerate the convergence and improve the optimization
- Know the benefit of learning rate decay and apply it on your optimization

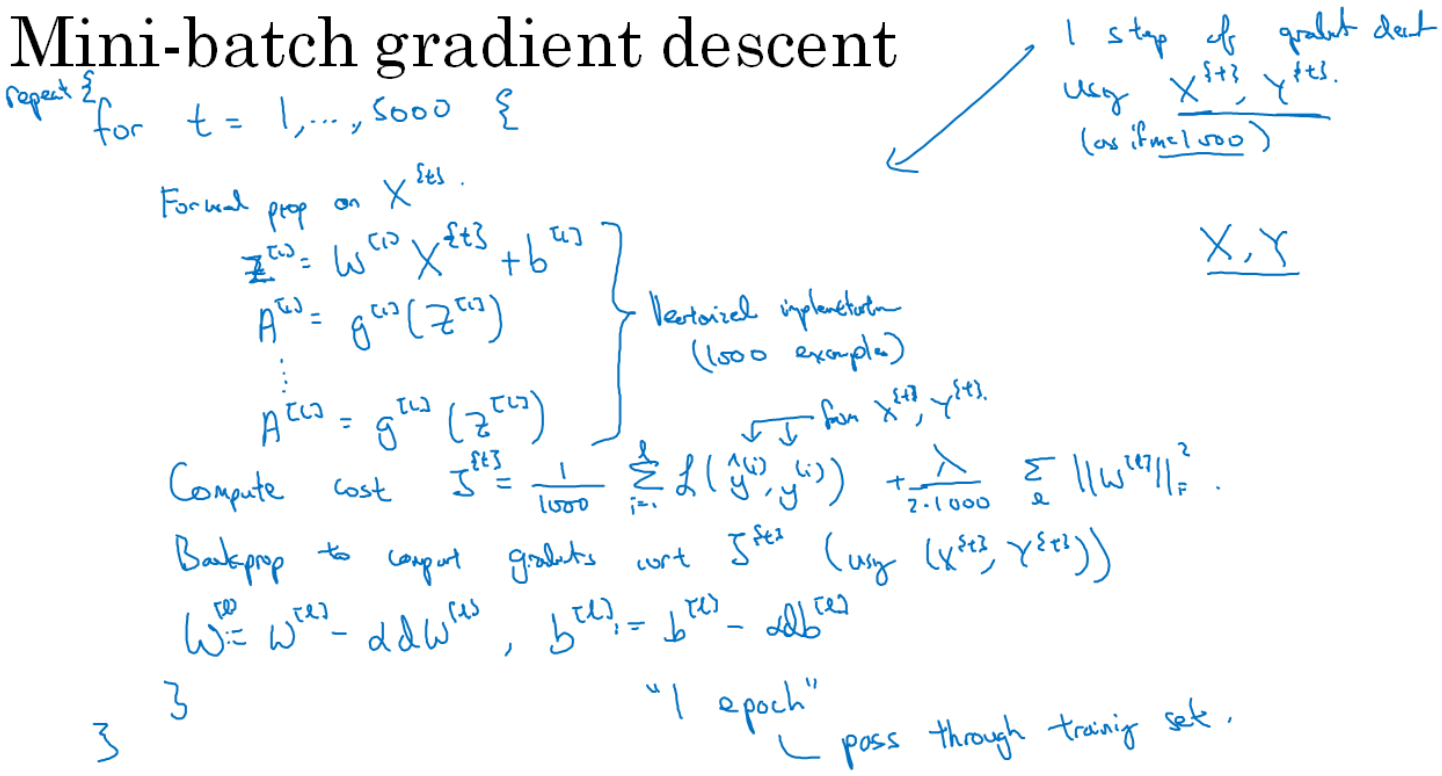
Optimization algorithms

Mini-batch gradient descent

In this section we'll learn about optimization algorithms that will enable user to train your neural network much faster. As mentioned before applying machine learning is a highly empirical process, is highly iterative process. In which you just had to train a lot of models to find one that works really well. So, it really helps to really train models quickly. One thing that makes it more difficult is that Deep Learning does not work best in a regime of big data. We are able to train neural networks on a huge data set and training on a large data set is just slow. So, what you find is that having fast optimization algorithms, having good optimization algorithms can really speed up the efficiency of you and your team. So, let's get started by talking about mini-batch gradient descent. We've learned previously that vectorization allows you to efficiently compute on all m examples, that allows you to process your whole training set without an explicit formula. That's why we would take our training examples and stack them into these huge matrix capsul. Check diagram below:

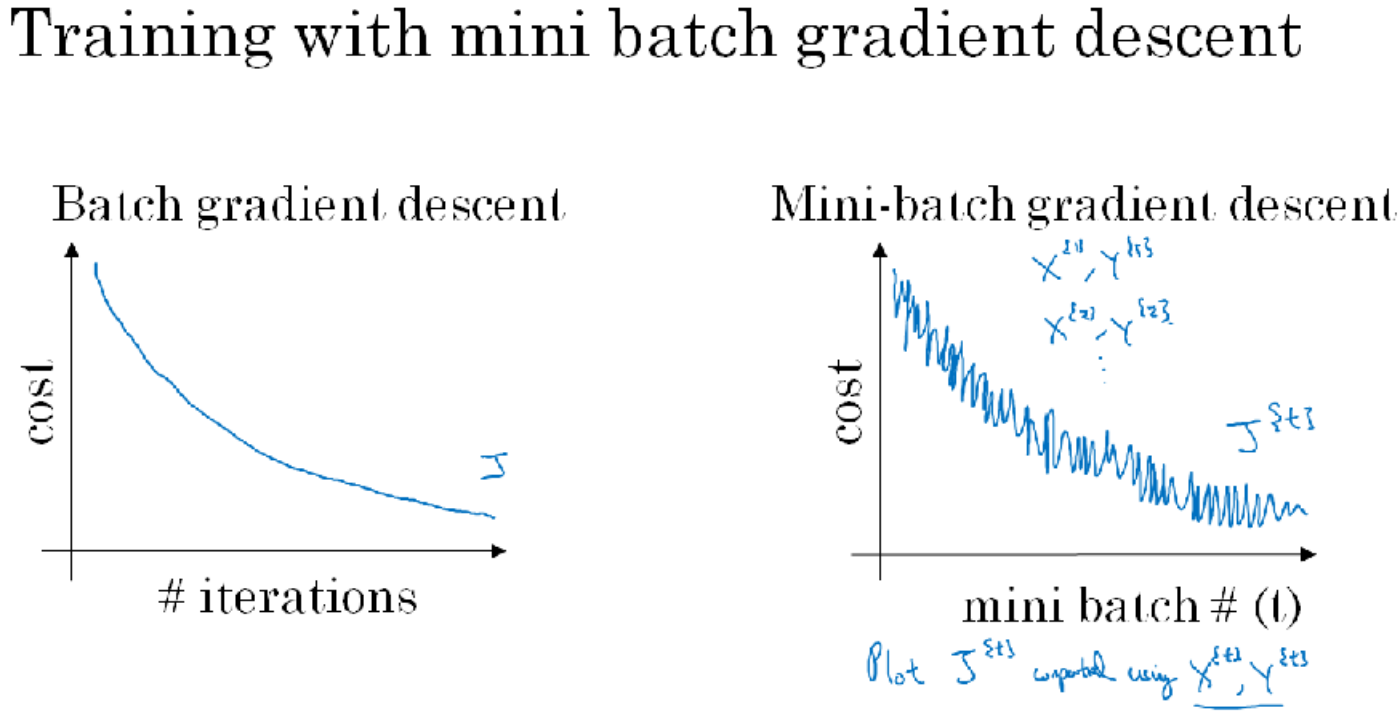


Vectorization allows you to process all M examples relatively quickly if M is very large then it can still be slow. For example what if M was 5 million or 50 million or even bigger. With the implementation of gradient descent on your whole training set, what you have to do is, you have to process your entire training set before you take one little step of gradient descent. And then you have to process your entire training sets of five million training samples again before you take another little step of gradient descent. So, it turns out that you can get a faster algorithm if you let **gradient descent start to make some progress even before you finish processing your entire, your giant training sets of 5 million examples**. In particular, here's what you can do. Let's say that you split up your training set into smaller, little baby training sets and these baby training sets are called **mini-batches**. Check diagram for detail explanation. To explain the name of this algorithm, **batch gradient descent**, refers to the gradient descent algorithm we have been talking about previously. Where you process your entire training set all at the same time. And the name comes from viewing that as processing your entire batch of training samples all at the same time. **Mini-batch gradient** descent in contrast, refers to algorithm which we'll talk about on the next section and which you process is single mini batch with mini-batch gradient descent, a single pass through the training set, that is one epoch, allows you to take 5,000 gradient descent steps. Now of course you want to take multiple passes through the training set which you usually want to, you might want another for loop for another while loop out there. So you keep taking passes through the training set until hopefully you converge with approximately converge. When you have a lost training set, mini-batch gradient descent runs much faster than batch gradient descent and that's pretty much what everyone in Deep Learning will use when you're training on a large data set.
Check this diagram too:



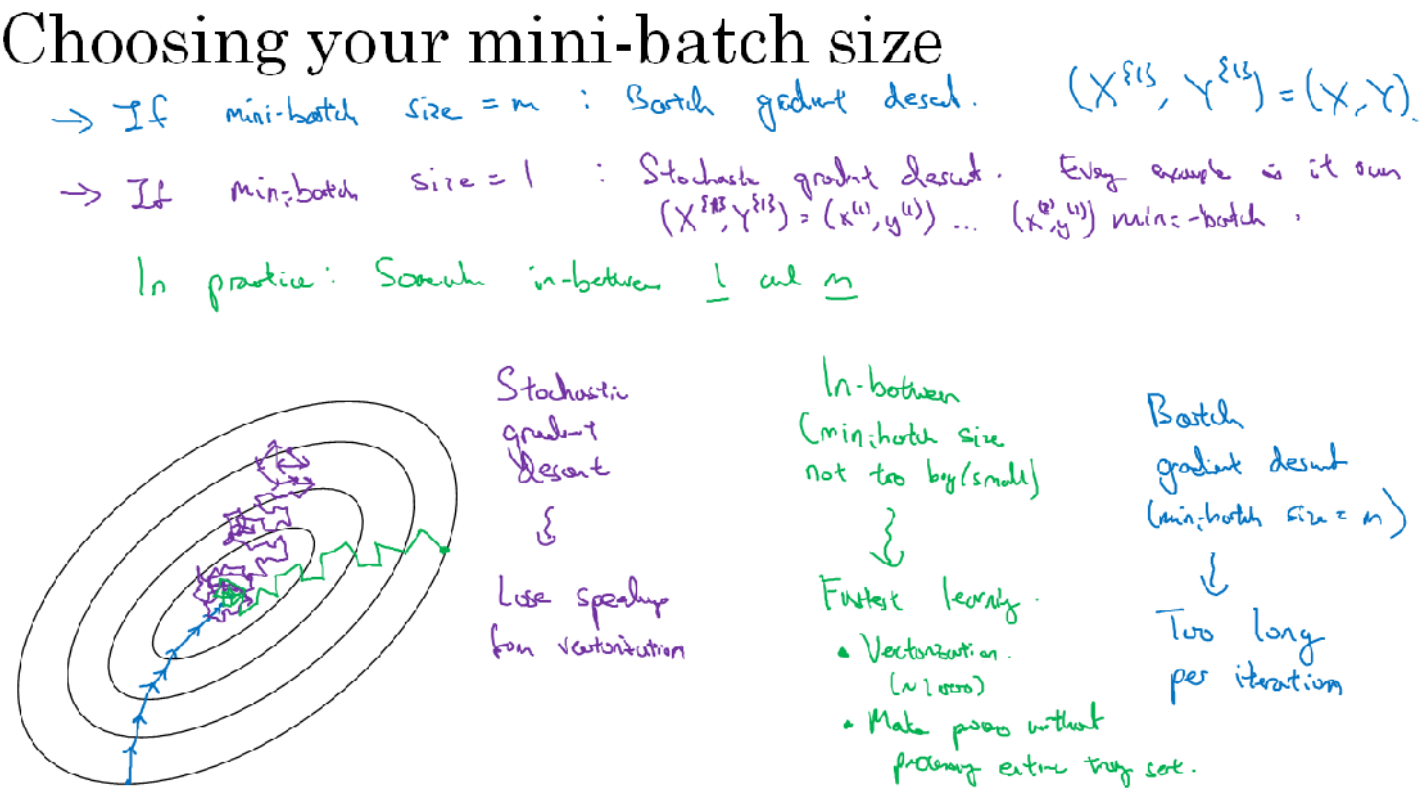
Understanding Mini-batch gradient descent

In this section, you learn more details of how to implement gradient descent and gain a better understanding of what it's doing and why it works. With batch gradient descent on every iteration you go through the entire training set and you'd expect the cost to go down on every single iteration. Check the diagram below:

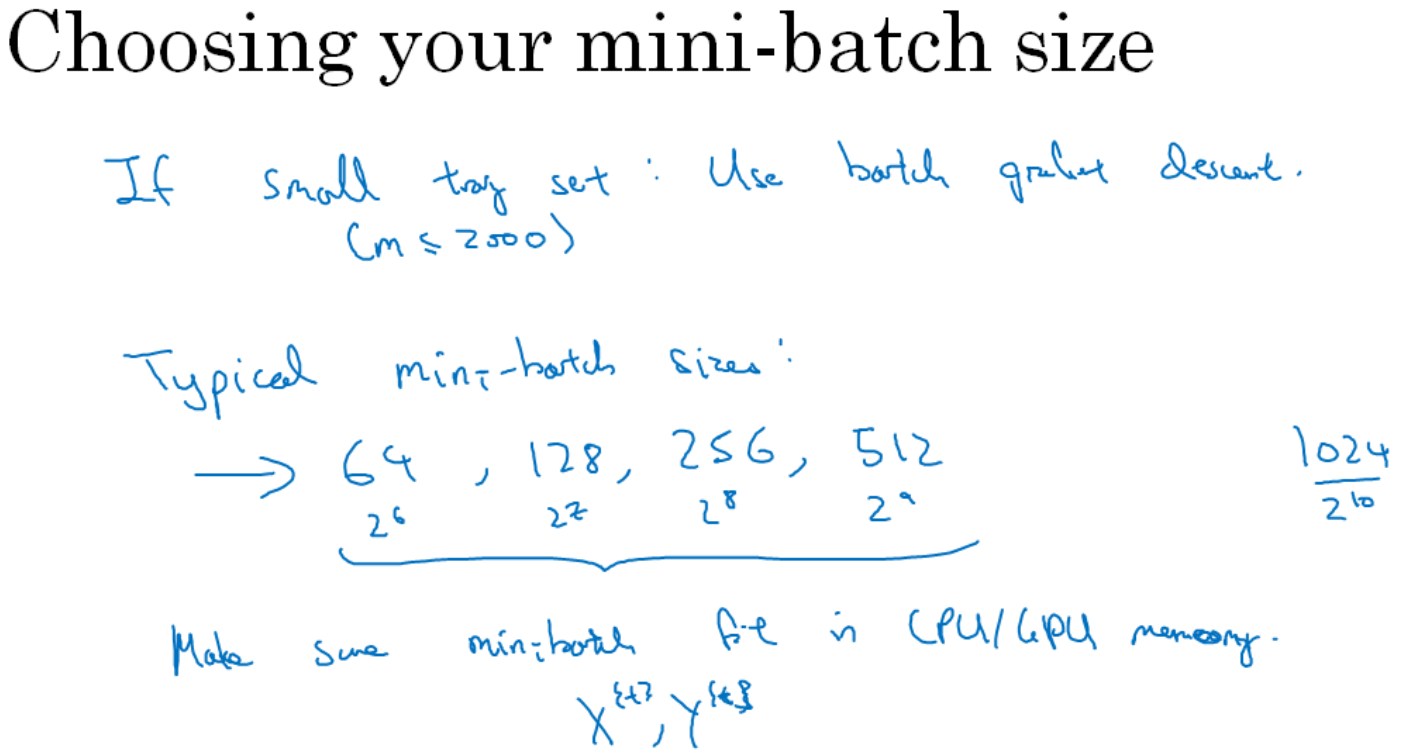


So if we've had the cost function j as a function of different iterations it should decrease on every single iteration. And if it ever goes up even on iteration then something is wrong. Maybe you're running ways to big. On mini batch gradient descent though, if you plot progress on your cost function, then it may not decrease on every iteration. In particular, on every iteration you're processing some $X^{\{t\}}$, $Y^{\{t\}}$ and so if you plot the cost function $J^{\{t\}}$, which is computer using just $X^{\{t\}}$, $Y^{\{t\}}$. Then it's as if on every iteration you're training on a

different training set or really training on a different mini batch. So you plot the cross function J, you're more likely to see something that looks like this. It should trend downwards, but it's also going to be a little bit noisier.



Some guidelines to choose mini-batch algorithm and size. First, if you have a small training set, Just use **batch gradient descent.** If you have a small training set then no point using **mini-batch gradient descent** you can process a whole training set quite fast. So you might as well use batch gradient descent. What a small ~ 2000 training set means. Otherwise, if you have a bigger training set, typical mini batch sizes would be anything from 64 up to maybe 512 are quite typical. And because of the way computer memory is laid out and accessed, sometimes your code runs faster if your mini-batch size is a power of 2.

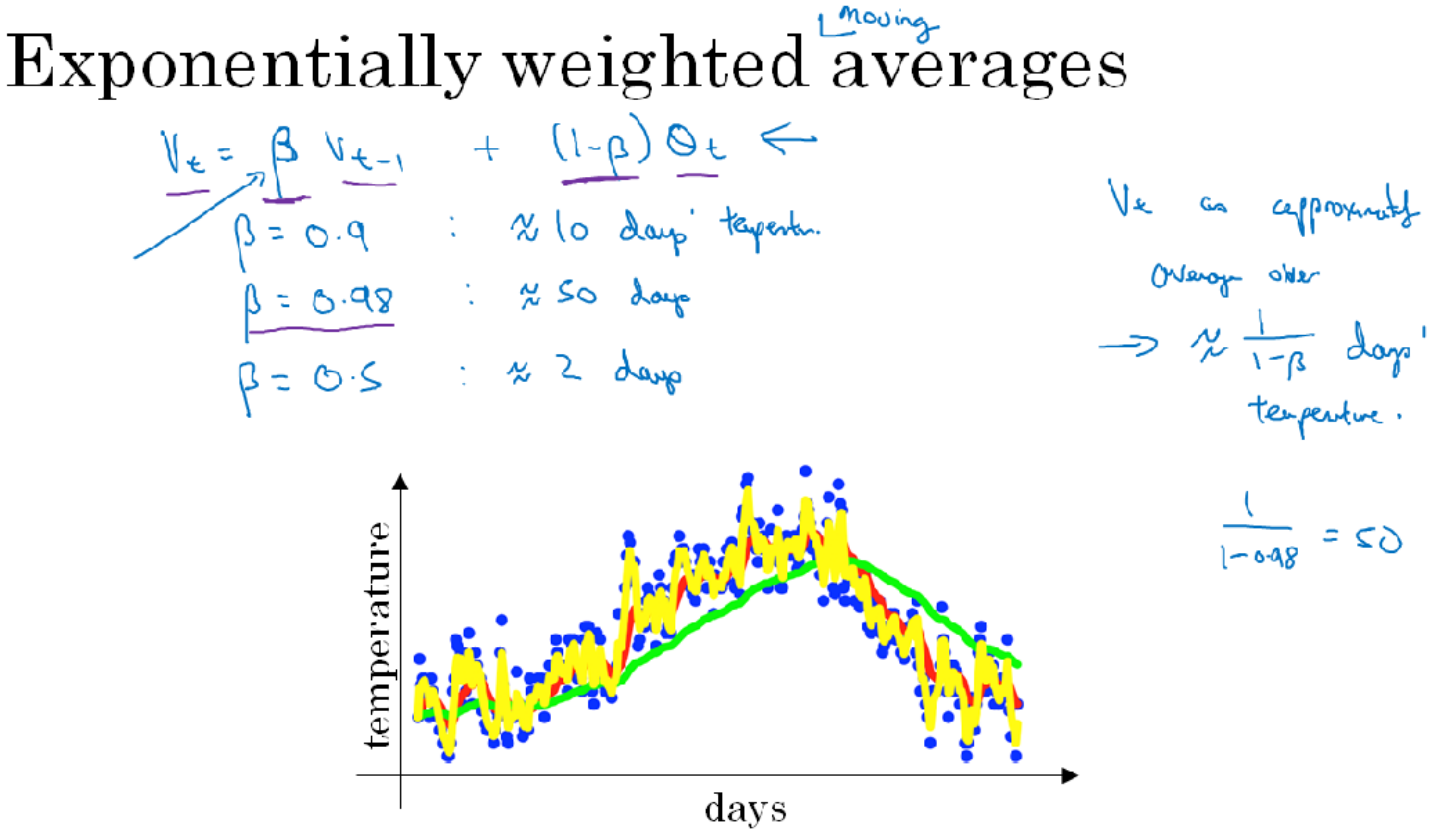


One more point, all of your $X^{\{t\}}$, $Y^{\{t\}}$ that that fits in CPU/GPU memory and this really depends on your application and how large a single training sample is. But if you ever process a mini-batch that doesn't actually

fit in CPU, GPU memory, whether you're using the process, the data. Then you find that the performance suddenly falls of a cliff and is suddenly much worse. So this gives a sense of the typical range of mini batch sizes that people use. In practice of course the mini batch size is another hyper parameter that you might do a quick search over to try to figure out which one is most sufficient of reducing the cost function J.

Exponentially weighted averages

There are some optimization algorithms which are faster than gradient descent. In order to understand those algorithms, you need to be able they use something called **exponentially weighted averages**. Also called exponentially weighted moving averages in statistics.



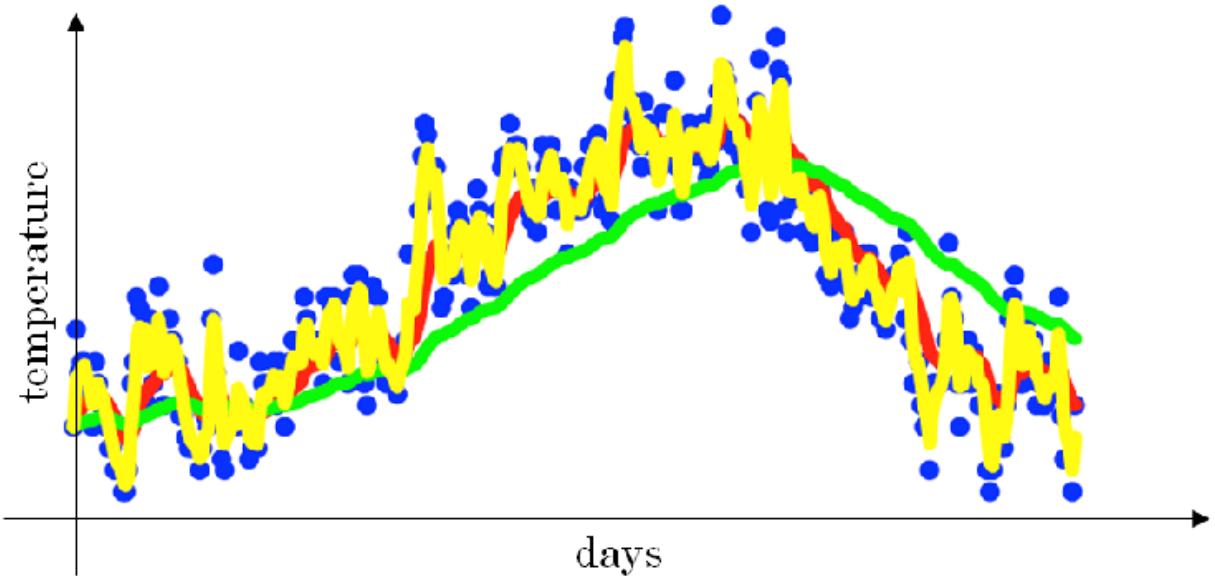
Understanding exponentially weighted averages

In the last section, we talked about **exponentially weighted averages**. This will turn out to be a key component of several **optimization algorithms** that you used to train your neural networks. So, in this section, we'll delve a little bit deeper into intuitions for what this algorithm is really doing.

Exponentially weighted averages

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$\beta = 0.9 \quad 0.98 \quad 0.5$



Implementing exponentially weighted averages

$$v_0 = 0$$
$$v_1 = \beta v_0 + (1-\beta)\,\theta_1$$
$$v_2 = \beta v_1 + (1-\beta)\,\theta_2$$
$$v_3 = \beta v_2 + (1-\beta)\,\theta_3$$
$$\ldots$$

$V_\theta := 0$

$V_\theta := \beta V + (1-\beta)\theta_1$

$V_\theta := \beta V + (1-\beta)\theta_2$

$\vdots$

$\rightarrow V_\theta = 0$

Repeat {

Get next θ_t

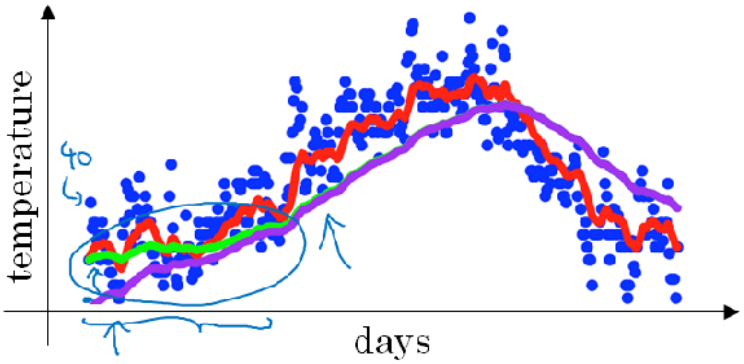
$V_\theta := \beta V_\theta + (1-\beta)\theta_t$

}

Bias correction in exponentially weighted averages

We've learned how to implement exponentially weighted averages. There's one technical detail called **bias correction** that can make you computation of these averages more accurately.

Bias correction



$\beta = 0.98$

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$v_0 = 0$

$v_1 = 0.98 v_0 + 0.02\theta_1$

$v_2 = 0.98 v_1 + 0.02\theta_2$

$= 0.98 \times 0.02 \times \theta_1 + 0.02\theta_2$

$= 0.0196\theta_1 + 0.02\theta_2$

$\frac{v_t}{1-\beta^t}$

$t=2: \quad 1-\beta^t = 1-(0.98)^2 = 0.0396$

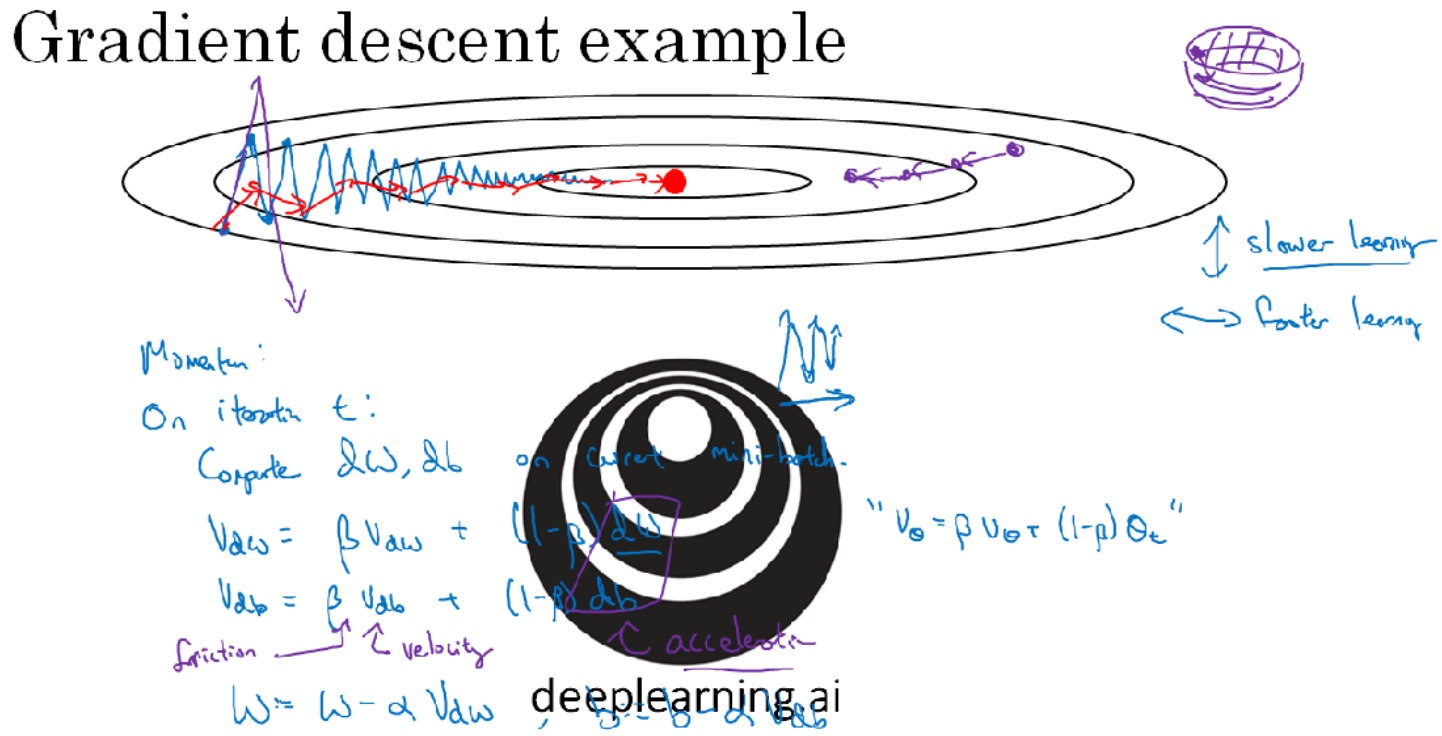
$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$

In machine learning, for most implementations of the **exponential weighted average**, people don't often bother to implement **bias corrections** because most people would rather just wait that initial period and have a slightly more biased estimate and go from there but if you are concerned about the bias during this initial phase, while your exponentially weighted moving average is still warming up. Then bias correction can help you get a better estimate early on.

Gradient descent with momentum

There's an algorithm called **momentum**, or **gradient descent with momentum that almost always works faster than the standard gradient descent algorithm**. In one sentence, the basic idea is to compute an **exponentially weighted average of your gradients, and then use that gradient to update your weights instead**. In this section, let's unpack that one sentence description and see how you can actually implement this. As a example let's say that you're trying to optimize a cost function which has contours like this.

So the red dot denotes the position of the minimum.



Looking at diagram we can see that the up and down **oscillations slows down gradient descent** and prevents us from using a much **larger learning rate**. In particular, if you were to use a much larger learning rate you might end up over shooting and end up diverging and so the need to prevent the oscillations from getting too big forces you to use a learning rate that's not itself too large. Another way of viewing this problem is that on the vertical axis you want your learning to be a bit slower, because you don't want those oscillations. But on the horizontal axis, you want **faster learning**. As we want to aggressively move from left to right, toward that minimum, toward that red dot (in diagram).To do that we can implement gradient descent with momentum.

On each iteration, or more specifically, during iteration you would compute the usual derivatives dw, db, and if you're using batch gradient descent, then the current mini-batch would be just your whole batch and this works as well off a batch gradient descent. So if your current mini-batch is your entire training set, this works fine as well.

Implementation details

$V_{dW} = 0$, $V_{db} = 0$

On iteration t:

Compute dW, db on the current mini-batch

$$v_{dW} = \beta v_{dW} + (1-\beta) dW$$
$$v_{db} = \beta v_{db} + (1-\beta) db$$

$V_{dW} = \beta V_{dW} + dW$

$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Hyperparameters: α, β

$\beta = 0.9$

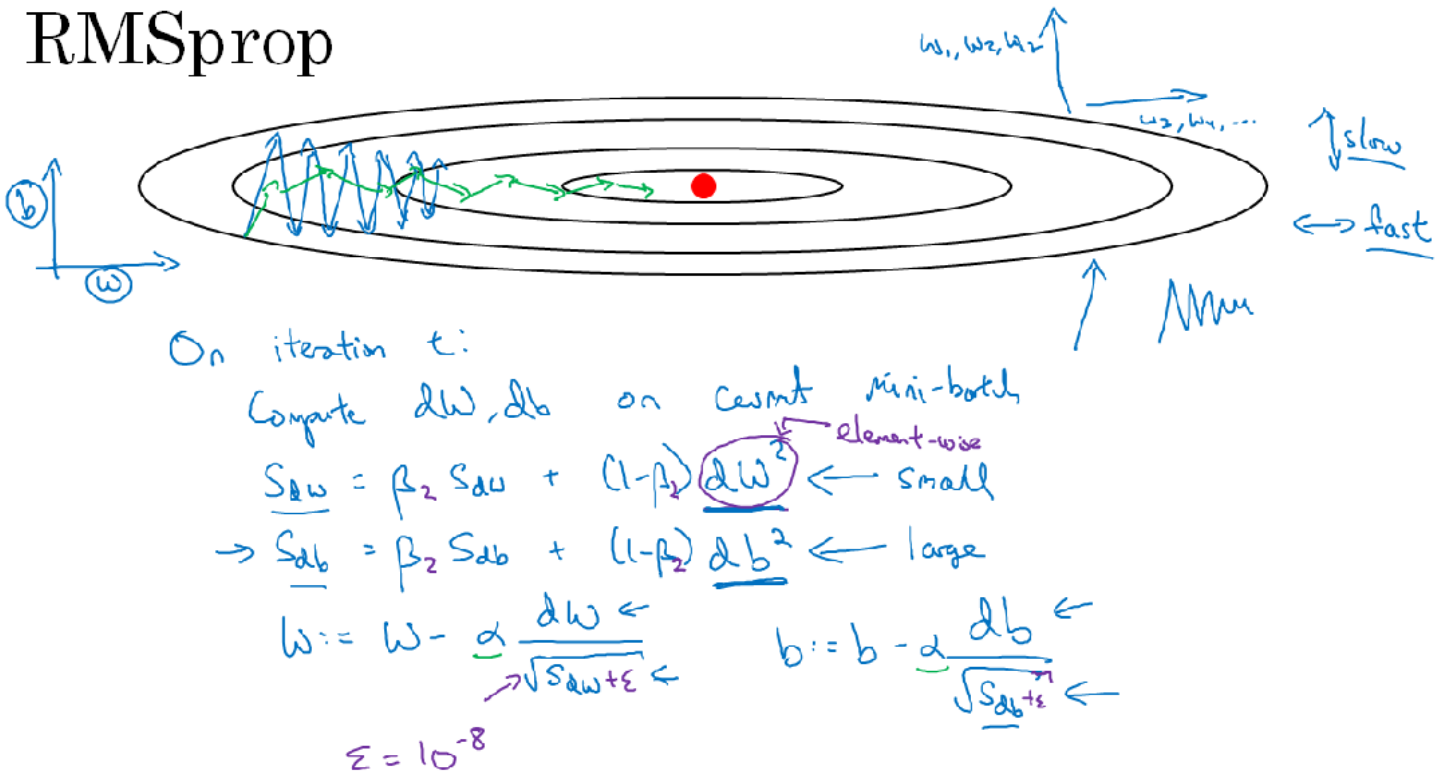
average over last ≈ 10 gradients

(Check the diagram for explanation)
If you average out these gradients, you find that the oscillations in the vertical direction will tend to average out to something closer to zero. So, in the vertical direction, where you want to slow things down, this will average out positive and negative numbers, so the average will be close to zero. Whereas, on the horizontal direction, all the derivatives are pointing to the right of the horizontal direction, so the average in the horizontal direction will still be pretty big. So that's why with this algorithm, with a few iterations you find that the gradient descent with momentum ends up eventually just taking steps that are much smaller oscillations in the vertical direction, but are more directed to just moving quickly in the horizontal direction. And so this allows your algorithm to take a more straightforward path, or to damp out the oscillations in this path to the minimum.

RMSProp

There's another algorithm called **RMSprop**, which stands for **root mean square prop**, that can also speed up gradient descent.
Let's see how it works. Recall our example from before, that if you implement gradient descent, you can end up with huge oscillations in the vertical direction, even while it's trying to make progress in the horizontal direction. In order to provide intuition for this example, let's say that the vertical axis is the parameter b and horizontal axis is the parameter w. It could be w1 and w2 where some of the center parameters was named as b and w for the sake of intuition. And so, you want to slow down the learning in the b direction, or in the vertical direction. And speed up learning, or at least not slow it down in the horizontal direction. So this is what the RMSprop algorithm does to accomplish this. On iteration t, it will compute as usual the derivative dW, db on the current

mini-batch.



Check the diagram for formulas.

Now let's gain some intuition about how this works. Recall that in the horizontal direction or in this example, in the W direction we want learning to go pretty fast. Whereas in the vertical direction or in this example in the b direction, we want to slow down all the oscillations into the vertical direction. So with this terms $\mathbf{S_{dW}}$ and $\mathbf{S_{db}}$, what we're hoping is that $\mathbf{S_{dW}}$ will be relatively small, so that here we're dividing by relatively small number. Whereas $\mathbf{S_{db}}$ will be relatively large, so that here we're dividing a relatively large number in order to slow down the updates on a vertical dimension and indeed if you look at the derivatives, these derivatives are much larger in the vertical direction than in the horizontal direction. So the slope is very large in the b direction, so with derivatives like this, this is a very large **db** and a relatively small **dw**. Because the function is sloped much more steeply in the vertical direction than as in the b direction, than in the w direction, than in horizontal direction and so, **db squared** will be relatively large. So $\mathbf{S_{db}}$ will relatively large, whereas compared to that **dW** will be smaller, or **dW squared** will be smaller, and so $\mathbf{S_{dw}}$ will be smaller. The one effect of this is also that you can therefore use a **larger learning rate** alpha, and get faster learning without diverging in the vertical direction. So that's RMSprop, and it stands for root mean squared prop, because here you're squaring the derivatives, and then you take the square root.

So finally, RMSprop is a another way for you to speed up your learning algorithm. One fun fact about RMSprop, it was actually first proposed not in an academic research paper, but in a Coursera course that Jeff Hinton had taught on Coursera many years ago.

Adam Optimization Algorithm

During the history of deep learning, many researchers including some very well-known researchers, sometimes proposed optimization algorithms and showed that they worked well in a few problems. But those optimization algorithms subsequently were shown not to really generalize that well to the wide range of neural networks you might want to train. So over time, I think the deep learning community actually developed some amount of skepticism about new optimization algorithms. And a lot of people felt that **gradient descent with momentum** really works well, was difficult to propose things that work much better. So, **rms prop** and the **Adam optimization algorithm**, which we'll talk about in this section, is one of those rare algorithms that has really stood up, and has been shown to work well across a wide range of deep learning architectures So, this is one of the algorithms that I wouldn't hesitate to recommend you try because many people have tried it and seen it work well on many problems and the **Adam optimization algorithm** is basically taking **momentum and rms prop and putting them together**.

Adam optimization algorithm

$V_{dw} = 0, S_{dw} = 0, V_{db} = 0, S_{db} = 0$

On iteration t:

Compute dw, db using current mini-batch

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1) db$ ← "momentum" β_1

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2) db$ ← "RMSprop" β_2

yhat = np.array([.9, 0.2, 0.1, .4, .9])

$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t), \quad V_{db}^{corrected} = V_{db} / (1-\beta_1^t)$

$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t), \quad S_{db}^{corrected} = S_{db} / (1-\beta_2^t)$

$$W := W - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon} \qquad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

Looking into equations in diagram we cab see that this algorithm combines the effect of **gradient descent with momentum** together with **gradient descent with rms prop** and this is a commonly used learning algorithm that is proven to be very effective for many different neural networks of a very wide variety of architectures. So, this algorithm has a number of hyper parameters. The learning with hyper parameter alpha is still important and usually needs to be tuned. So you just have to try a range of values and see what works. A common choice really the default choice for ß1 is 0.9, this is the moving average. The hyper parameter for ß2, the authors of the Adam paper, inventors of the Adam algorithm recommend 0.999. Again this is computing the moving weighted average of dw2 as well as db squares. And then Epsilon, the choice of epsilon doesn't matter very much. But the authors of the Adam paper recommended it 10 to the -8. But this parameter you really don't need to set it and it doesn't affect performance much at all. But when implementing Adam, what people usually do is just use the default value. So, ß1 and ß2 as well as epsilon. I don't think anyone ever really tunes Epsilon. And then, try a range of values of Alpha to see what works best. You could also tune ß1 and ß2 but it's not done that often among the practitioners I know.

Hyperparameters choice:

→ α : needs to be tune

→ β_1 : 0.9 → (dw)

→ β_2 : 0.999 → (dw^2)

→ ε : 10^{-8}

Adam : Adaptive moment estimation

Adam Coates

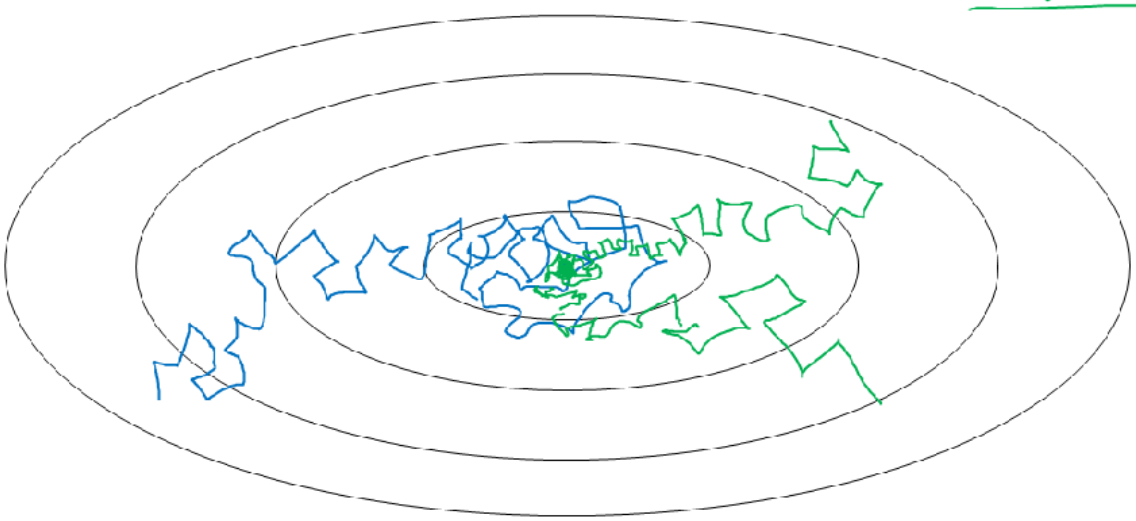
So, where does the term 'Adam' come from? Adam stands for **Adaptive Moment Estimation**. So ß1 is computing the mean of the derivatives. This is called the **first moment**. And ß2 is used to compute exponentially weighted average of the dw² and that's called the **second moment**. So that gives rise to the name adaptive moment estimation. But everyone just calls it the **Adam optimization algorithm**.

Learning rate decay

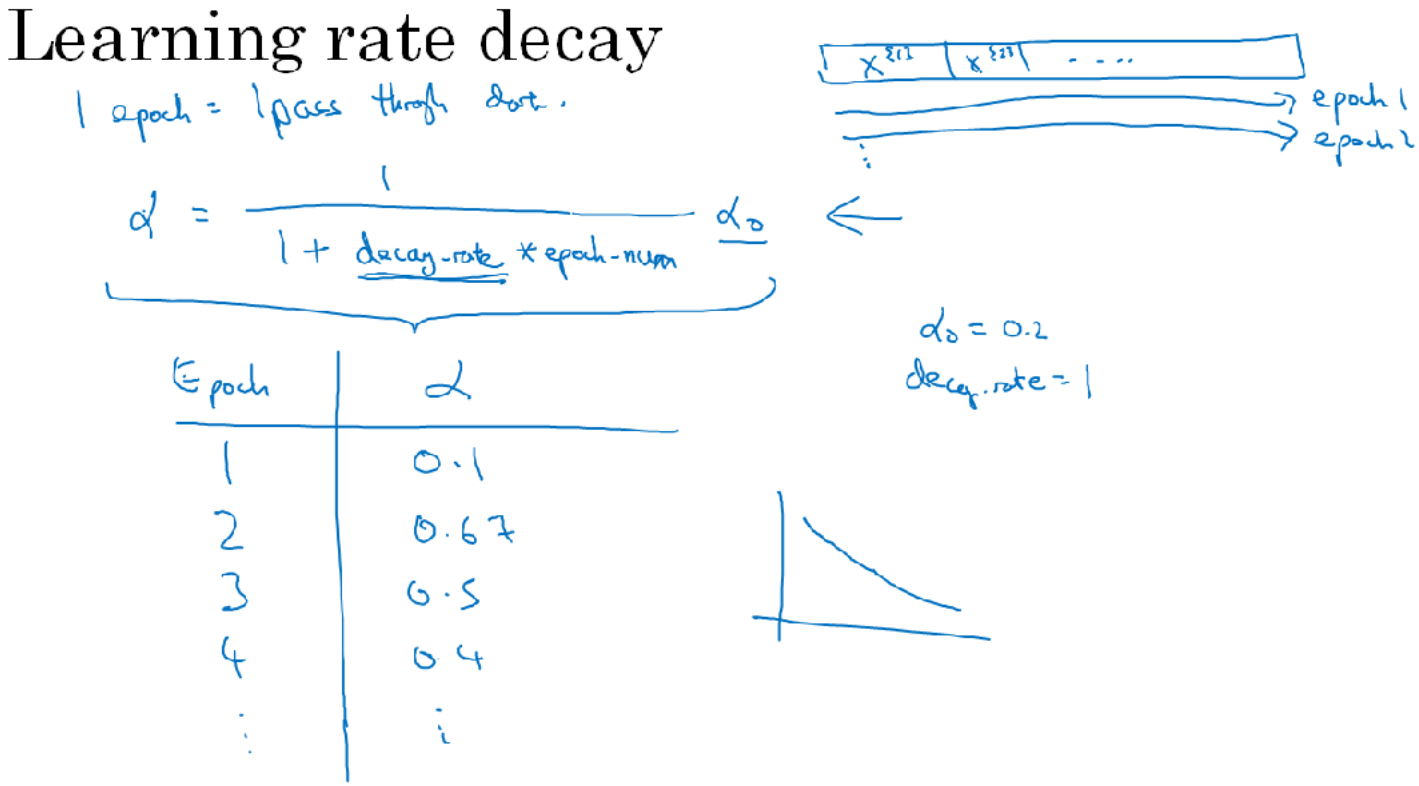
One of the things that might help speed up your learning algorithm, is to slowly reduce your learning rate over time. We call this **learning rate decay**. Let's see how you can implement this. Let's start with an example of why you might want to implement learning rate decay. Suppose you're implementing mini-batch gradient descent, with a reasonably small mini-batch. Maybe a mini-batch has just 64, 128 examples. Then as you iterate, your steps will be a little bit noisy. And it will tend towards this minimum (see the diagram)

Learning rate decay

Slowly reduce α



but it won't exactly converge but our algorithm might just end up wandering around, and never really converge, because we're using some fixed value for alpha and there's just some noise in your different mini-batches. But if you were to slowly reduce your learning rate alpha, then during the initial phases, while your learning rate alpha is still large, you can still have relatively fast learning but then as alpha gets smaller, your steps you take will be slower and smaller and so you end up oscillating in a tighter region (check diagram) rather than wandering far away, even as training goes on and on. So the intuition behind slowly reducing alpha, is that maybe during the initial steps of learning, you could afford to take much bigger steps but then as learning approaches converges, then having a slower learning rate allows you to take smaller steps. Check the diagram for steps to implement learning rate decay.

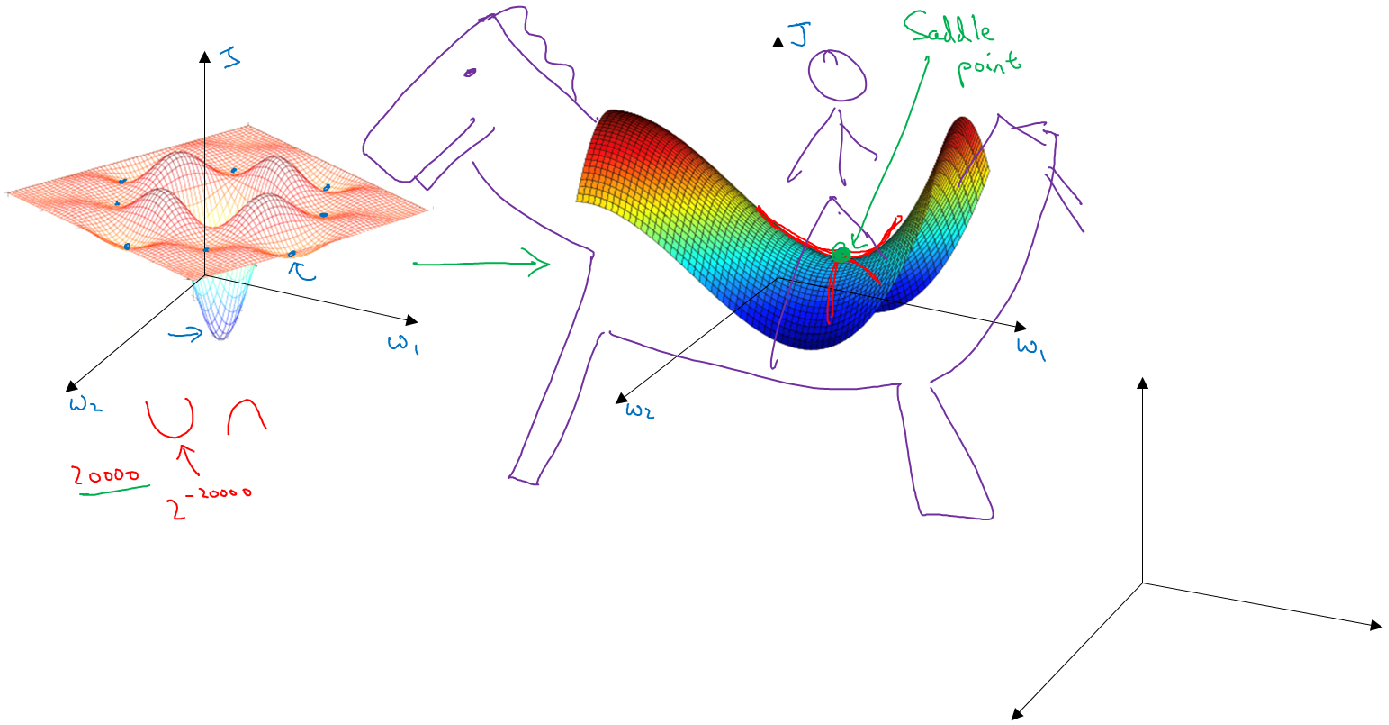


So so far, we've talked about using some formula to govern how alpha, the learning rate, changes over time. One other thing that people sometimes do, is **manual decay** and so if you're training just one model at a time, and if your model takes many hours, or even many days to train. What some people will do, is just watch your model as it's training over a large number of days and then manually say, it looks like the learning rate slowed down, I'm going to decrease alpha a little bit. Of course this works, this manually controlling alpha, really tuning alpha by hand, hour by hour, or day by day. This works only if you're training only a small number of models, but sometimes people do that as well. So now you have a few more options for how to control the learning rate alpha. Finally, **learning rate decay** is usually lower down on the list of things I try. **Setting alpha**, just a fixed value of alpha, and getting that to be well tuned, has a huge impact.

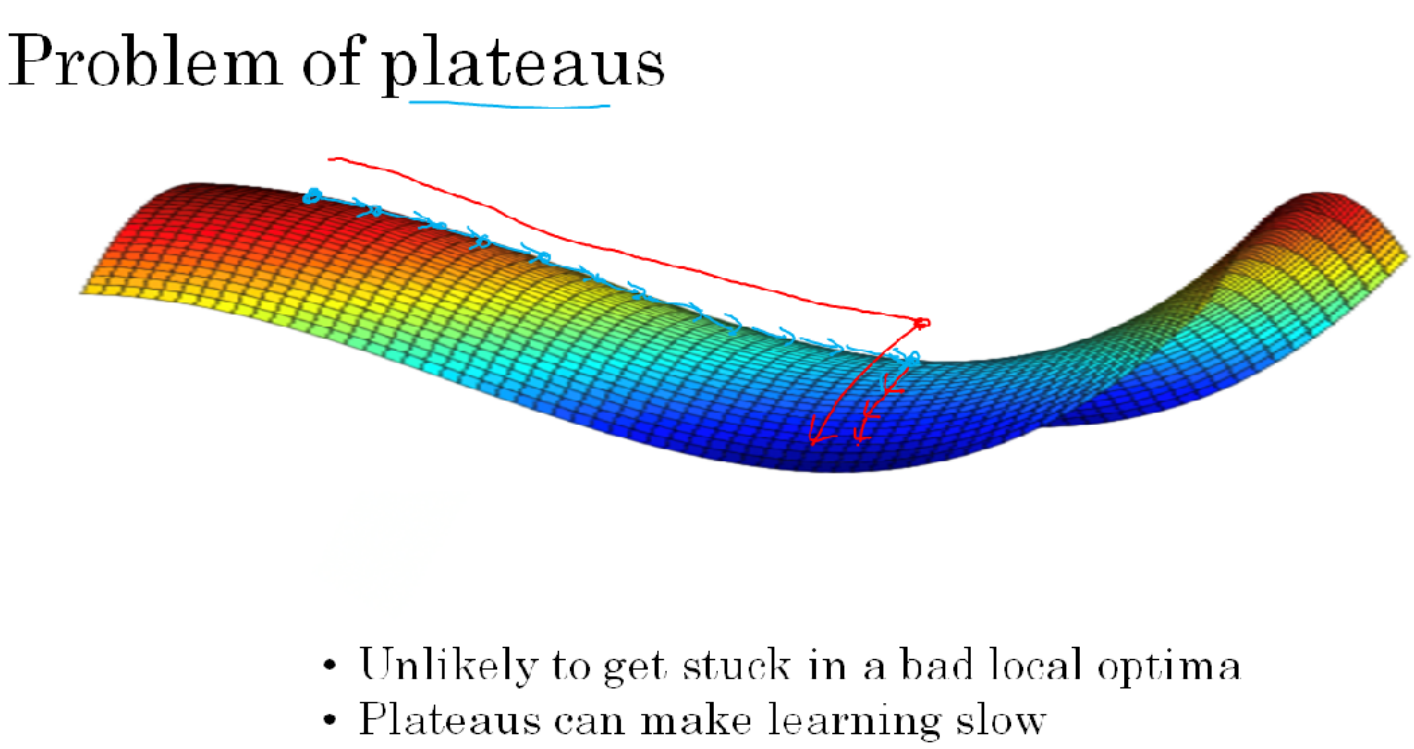
The problem of local optima

In the early days of deep learning, people used to worry a lot about the optimization algorithm getting stuck in bad local optima but as this theory of deep learning has advanced, our understanding of **local optima** is also changing. Let me show you how we now think about local optima and problems in the optimization problem in deep learning. This was a picture people used to have in mind when they worried about local optima.

Local optima in neural networks



Maybe you are trying to optimize some set of parameters, we call them W1 and W2, and the height in the surface is the cost function. In this picture, it looks like there are a lot of local optima in all those places. And it'd be easy for grading the sense, or one of the other algorithms to get stuck in a local optimum rather than find its way to a global optimum. It turns out that if you are plotting a figure like this in two dimensions, then it's easy to create plots like this with a lot of different local optima. And these very low dimensional plots used to guide their intuition. But this intuition isn't actually correct. It turns out if you create a neural network, most points of zero gradients are not local optima like points like this. Instead most points of **zero gradient in a cost function are saddle points**. So, that's a point where the zero gradient, again, just is maybe W1, W2, and the height is the value of the cost function J. But informally, a function of very high dimensional space, if the gradient is zero, then in each direction it can either be a **convex like** function or a **concave like** function. And if you are in, say, a 20,000 dimensional space, then for it to be a local optima, all 20,000 directions need to look like this. And so the chance of that happening is maybe very small, maybe two to the minus 20,000. Instead you're much more likely to get some directions where the curve bends up like so, as well as some directions where the curve function is bending down rather than have them all bend upwards. **So that's why in very high-dimensional spaces you're actually much more likely to run into a saddle point like that shown on the right, then the local optimum**. As for why the surface is called a saddle point, if you can picture, maybe this is a sort of saddle you put on a horse, right? Maybe this is a horse. This is a head of a horse, this is the eye of a horse. Well, not a good drawing of a horse but you get the idea. Then you, the rider, will sit here in the saddle. That's why this point here, where the derivative is zero, that point is called a saddle point. There's really the point on this saddle where you would sit, I guess, and that happens to have derivative zero and so, one of the lessons we learned in history of deep learning is that a lot of our intuitions about low-dimensional spaces, like what you can plot on the left, they really don't transfer to the very high-dimensional spaces that any other algorithms are operating over. Because if you have 20,000 parameters, then J as your function over 20,000 dimensional vector, then you're much more likely to see saddle points than local optimum. If local optima aren't a problem, then what is a problem?



It turns out that plateaus can really slow down learning and a plateau is a region where the derivative is close to zero for a long time. So if you're here, then gradient descents will move down the surface, and because the gradient is zero or near zero, the surface is quite flat. You can actually take a very long time, you know, to slowly find your way to maybe this point on the plateau. Let it take this very long slope off before it's found its way here and they could get off this plateau. So the takeaways from this section are, first, **you're actually pretty unlikely to get stuck in bad local optima so long as you're training a reasonably large neural network, save a lot of parameters, and the cost function J is defined over a relatively high dimensional space**. But second, **that plateaus are a problem and you can actually make learning pretty slow. And this is where algorithms like momentum or RmsProp or Adam can really help your learning algorithm as well**. And these are scenarios where more sophisticated optimization algorithms, such as Adam, can actually speed up the rate at which you could move down the plateau and then get off the plateau.

Week 3: Hyperparameter tuning

Learning Objectives

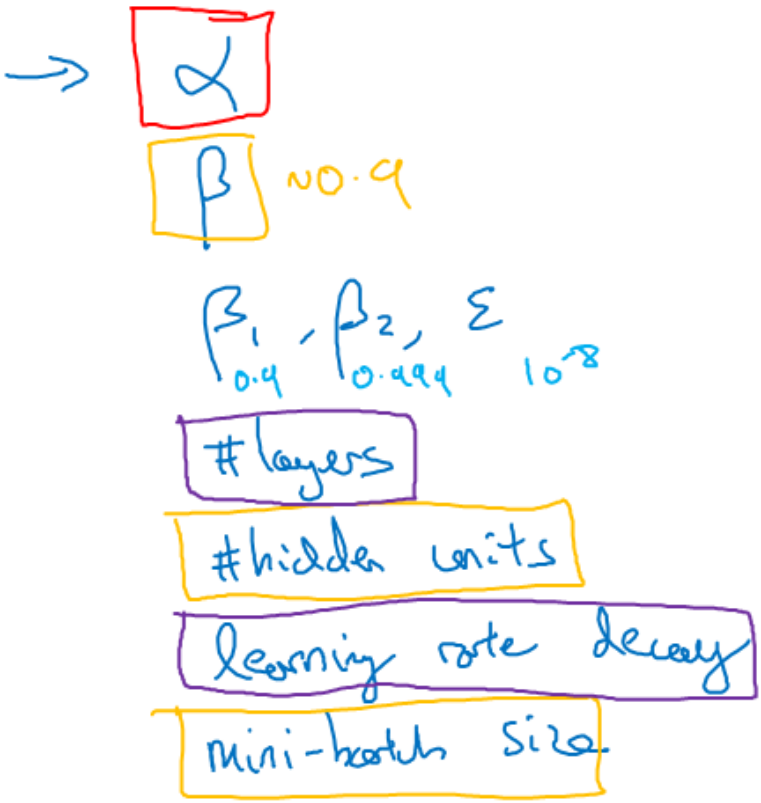
Master the process of hyperparameter tuning

Hyperparameter tuning

Tuning process

We've seen by now that changing neural nets can involve setting a lot of different hyperparameters. Now, how do you go about finding a good setting for these hyperparameters? In this section, we'll discuss some guidelines, some tips for how to systematically organize your hyperparameter tuning process, which hopefully will make it more efficient for you to converge on a good setting of the hyperparameters. One of the painful things about training deep nets is the sheer number of hyperparameters you have to deal with, ranging from the **learning rate alpha** to the **momentum term beta** (if using momentum), or the hyperparameters for the **Adam Optimization Algorithm** which are beta 1, beta 2, and epsilon. Maybe you have to pick the **number of layers**, maybe you have to pick the **number of hidden units** for the different layers, and maybe you want to use **learning rate decay**, so you don't just use a single learning rate alpha. And then of course, you might need to choose the **mini-batch size**. So it turns out, some of these hyperparameters are more important than others. The most learning applications I would say, **alpha, the learning rate is the most important hyperparameter to tune**. Other than alpha, a few other hyperparameters which we tune next, would be the **momentum term**, say, 0.9 is a good default. I'd also tune the **mini-batch size** to make sure that the optimization algorithm is running efficiently. Often we can fiddle around with the **hidden units**. Check the diagram below:

Hyperparameters



Of the ones we've circled in orange, these are really the three that we would consider second in importance to the learning rate alpha and then third in importance after fiddling around with the others, the number of layers can sometimes make a huge difference, and so can learning rate decay. So hopefully it does give us some rough sense of what hyperparameters might be more important than others, **alpha**, most important, for sure, followed maybe by the ones we've circle in **orange**, followed maybe by the ones we circled in **purple**.

Now, if you're trying to tune some set of hyperparameters, how do you select a set of values to explore? The two key takeaways are, use random sampling and adequate search and optionally consider implementing a coarse to fine search process.

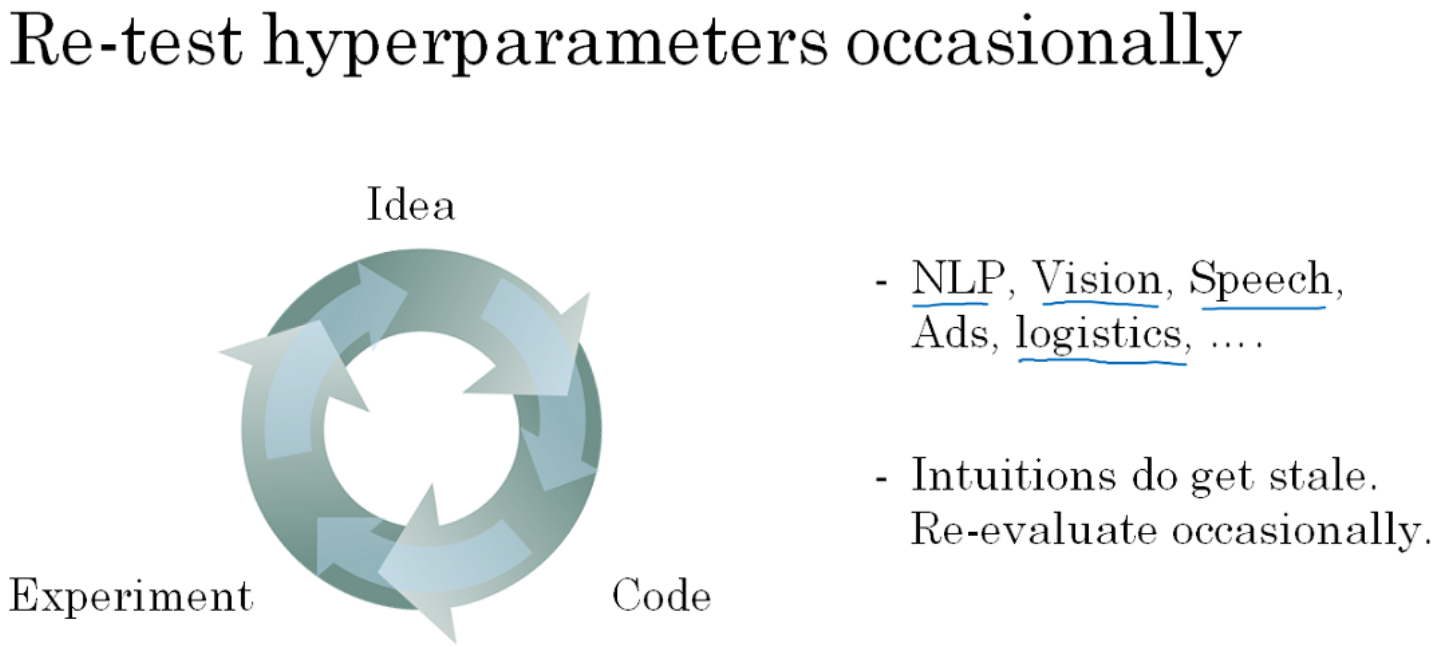
Using an appropriate scale to pick hyperparameters

In the last section, we saw how sampling at random, over the range of hyperparameters, can allow us to search over the space of hyperparameters more efficiently. But it turns out that sampling at random doesn't mean sampling uniformly at random, over the range of valid values. Instead, it's important to pick the appropriate scale on which to explore the hyperparamaters. Sampling uniformly at random over the range is not possible for all hyperparameters for example **learning rate alpha** where instead of linear scale, log scale is better choice. Finally, one other tricky case is sampling the hyperparameter beta, used for computing exponentially weighted averages.

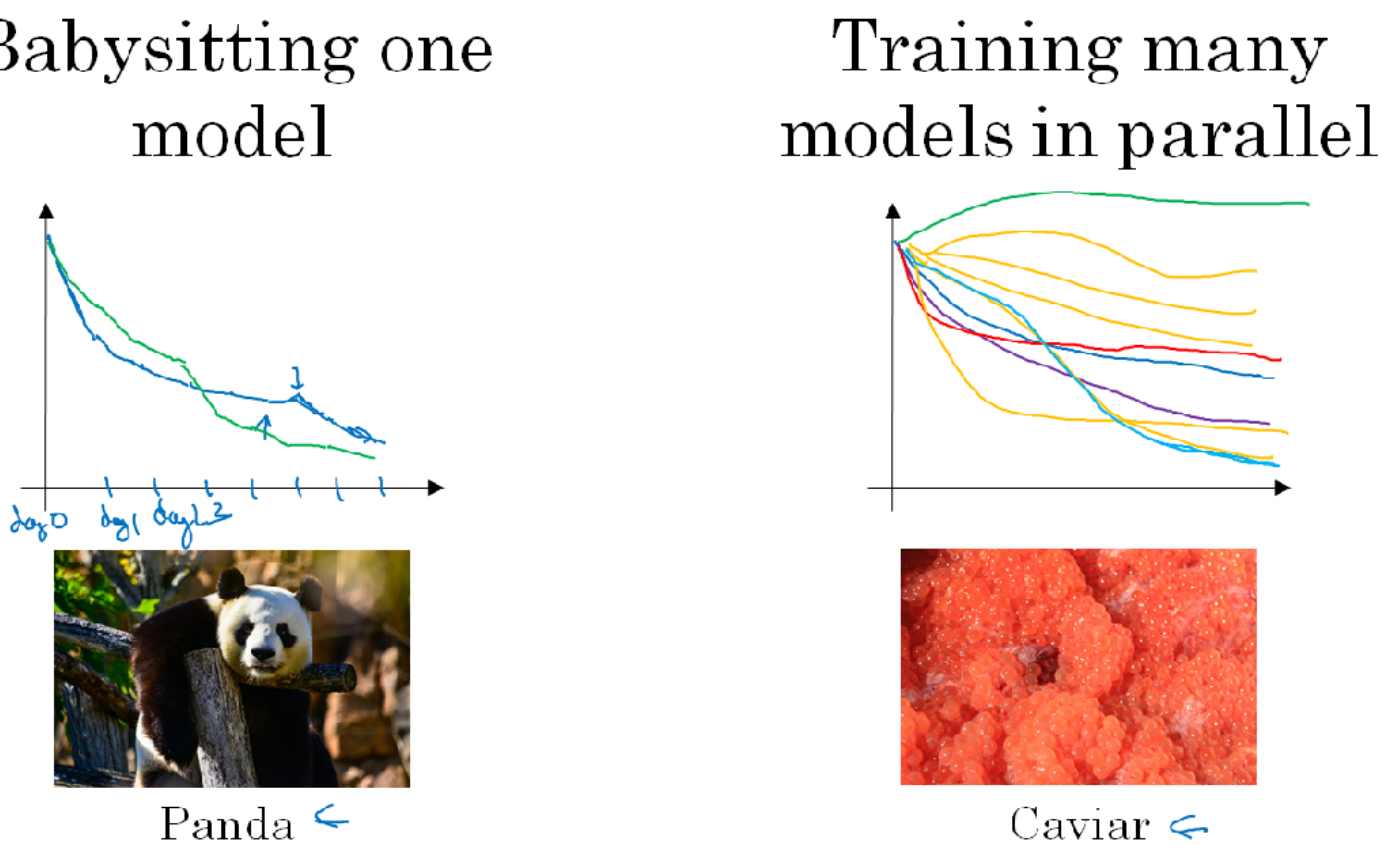
Hyperparameters tuning in practice: Pandas Vs. Caviar

Before wrapping up our discussion on hyperparameter search, here are just a couple of final tips and tricks for how to organize your hyperparameter search process. Deep learning today is applied to many different application areas and that intuitions about hyperparameter settings from one application area may or may not transfer to a different one. There is a lot of cross-fertilization among different applications' domains, so for example, we've seen ideas developed in the computer vision community, such as **Conv nets** or **ResNets**, (which we'll talk about later course), successfully applied to speech. We've seen ideas that were first developed in speech successfully applied in NLP, and so on. So one nice development in deep learning is that people from different application domains do read increasingly research papers from other application domains to look for

inspiration for cross-fertilization. In terms of your settings for the hyperparameters, though, we've seen that intuitions do get stale. So even if you work on just one problem, say logistics, you might have found a good setting for the hyperparameters and kept on developing your algorithm, or maybe seen your data gradually change over the course of several months, or maybe just upgraded servers in your data center. And because of those changes, the best setting of your hyperparameters can get stale. So recommendation is to just retesting or reevaluating your hyperparameters at least once every several months to make sure that you're still happy with the values you have.



Finally, in terms of how people go about searching for hyperparameters, I see maybe two major schools of thought, or maybe two major different ways in which people go about it. One way is if you **babysit one model**. And usually you do this if you have maybe a huge data set but **not a lot of computational resources**, not a lot of CPUs and GPUs, so you can basically afford to train only one model or a very small number of models at a time. In that case you might gradually babysit that model even as it's training. So, for example, on Day 0 you might initialize your parameter as random and then start training. And you gradually watch your learning curve, maybe the cost function J or your dataset error or something else, gradually decrease over the first day. Then at the end of day one you feel confident with the learning progress and try increasing the learning rate a little bit and see how it does. And then maybe it does better. And then that's your Day 2 performance. And after two days you say, okay, it's still doing quite well. Maybe I'll fill the momentum term a bit or decrease the learning variable a bit now, and then you're now into Day 3. And every day you kind of look at it and try nudging up and down your parameters. And maybe on one day you found your learning rate was too big. So you might go back to the previous day's model, and so on. But you're kind of babysitting the model one day at a time even as it's training over a course of many days or over the course of several different weeks. So that's one approach, and people that babysit one model, that is **watching performance and patiently nudging the learning rate up or down**. But that's usually what happens if you don't have enough computational capacity to train a lot of models at the same time.



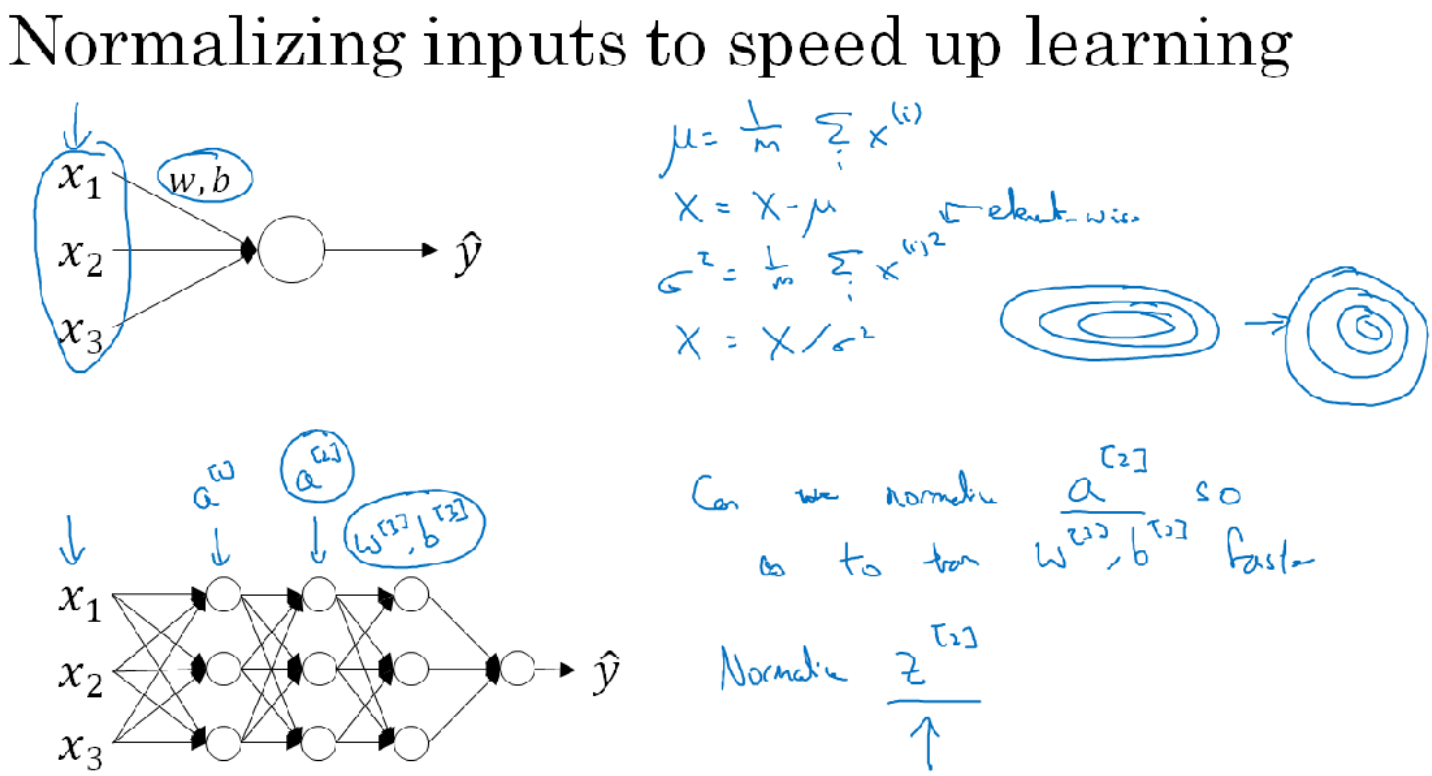
The other approach would be if you train many models in parallel. So you might have some setting of the hyperparameters and just let it run by itself ,either for a day or even for multiple days, and then you get some learning curve like (check right part of the diagram) and this could be a plot of the cost function J or cost of your training error or cost of your dataset error and then at the same time you might start up a different model with a different setting of the hyperparameters. And so, your second model might generate a different learning curve (check diagram). I will say that one looks better. And at the same time, you might train a third model, which might generate a learning curve (Check diagram), and another one that and so on. So this way you can try a lot of different hyperparameter settings and then just maybe quickly at the end pick the one that works best (curve in blue). Looks like in this example it was, maybe BLUE curve that look best. So to make an analogy, I'm going to call the approach on the left the **panda approach**. When pandas have children, they have very few children, usually one child at a time, and then they really put a lot of effort into making sure that the baby panda survives. So that's really babysitting. One model or one baby panda. Whereas the approach on the right is more like what fish do. I'm going to call this the **caviar strategy**. There's some fish that lay over 100 million eggs in one mating season. But the way fish reproduce is they lay a lot of eggs and don't pay too much attention to any one of them but just see that hopefully one of them, or maybe a bunch of them, will do well. But I'm going to call it the panda approach versus the caviar approach, since that's more fun and memorable. So the way to choose between these two approaches is really a function of how much computational resources you have. If you have enough computers to train a lot of models in parallel, then by all means take the caviar approach and try a lot of different hyperparameters and see what works. But in some application domains, I see this in some online advertising settings as well as in some **computer vision** applications, where there's just so much data and the models you want to train are so big that it's difficult to train a lot of models at the same time. It's really application dependent of course, but I've seen those communities use the panda approach a little bit more, where you are kind of babying a single model along and nudging the parameters up and down and trying to make this one model work. Although, of course, even the panda approach, having trained one model and then seen it work or not work, maybe in the second week or the third week, maybe I should initialize a different model and then baby that one along just like even pandas, I guess, can have multiple children in their lifetime, even if they have only one, or a very small number of children, at any one time.

Batch normalization

Normalizing activations in a network

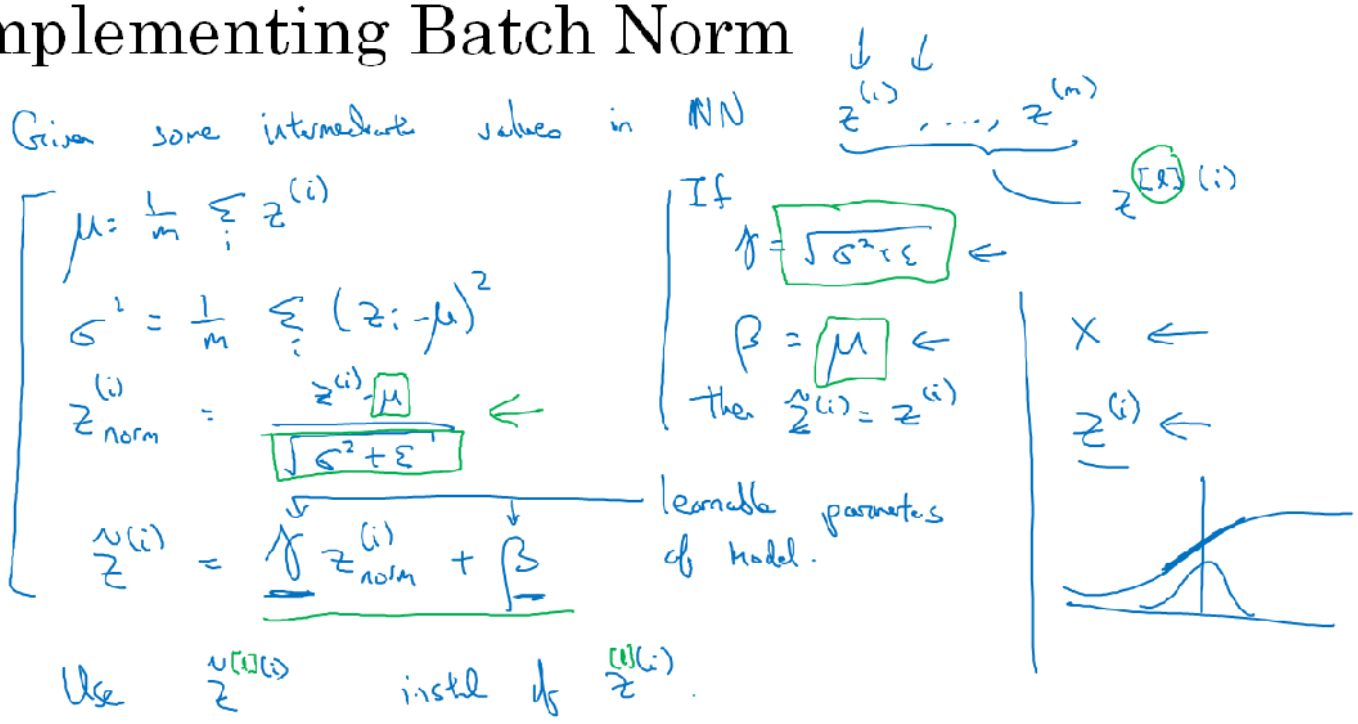
In the rise of deep learning, one of the most important ideas has been an algorithm called batch normalization, created by two researchers, Sergey Ioffe and Christian Szegedy. Batch normalization makes your **hyperparameter search problem** much easier, makes your neural network much more robust. The choice of

hyperparameters is a much bigger range of hyperparameters that work well, and will also enable you to much more easily train even very deep networks. Let's see how batch normalization works. When training a model, such as logistic regression, you might remember that normalizing the input features can speed up learnings.



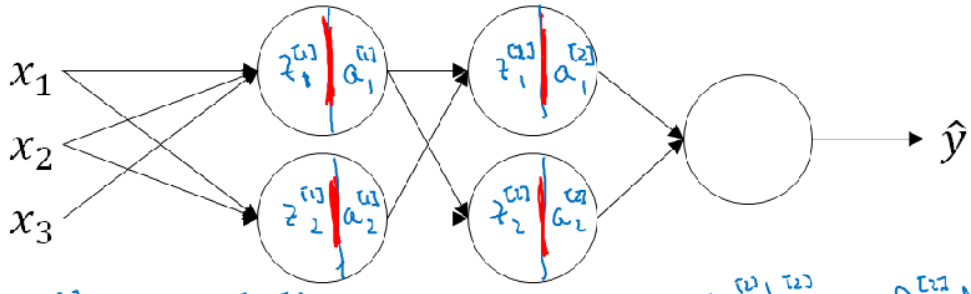
We can see in diagram that we can normalize our data set according to the variances and we saw in an earlier section how this can turn the contours of your learning problem from something that might be very elongated to something that is more round, and easier for an algorithm like **gradient descent to optimize**. So this works, in terms of normalizing the input feature values to a neural network. Now, how about a deeper model? You have not just input features x, but in this layer you have activations **a1**, in this layer, you have activations **a2** and so on. So if you want to train the parameters, say w3, b3, then wouldn't it be nice if you can normalize the mean and variance of **a2** to make the training of w3, b3 more efficient? In the case of logistic regression, we saw how normalizing **x1, x2, x3** maybe helps you train w and b more efficiently. So here, the question is, for any hidden layer, can we normalize, The values of a, let's say a2, in this example but really any hidden layer, so as to train w3 b3 faster, right? Since a2 is the input to the next layer, that therefore affects your training of w3 and b3. So this is what **batch norm does**, **batch normalization**, or batch norm for short, does. Although technically, we'll actually **normalize the values of not a2 but z2**. There are some debates in the deep learning literature about whether you should normalize the value before the activation function, so z2, or whether you should normalize the value after applying the activation function, a2. In practice, normalizing z2 is done much more often. So that's the version I'll present and what I would recommend you use as a default choice. Check below diagram for batch-norm implementation of single layer

Implementing Batch Norm



Fitting Batch Norm into a neural network

Adding Batch Norm to a network



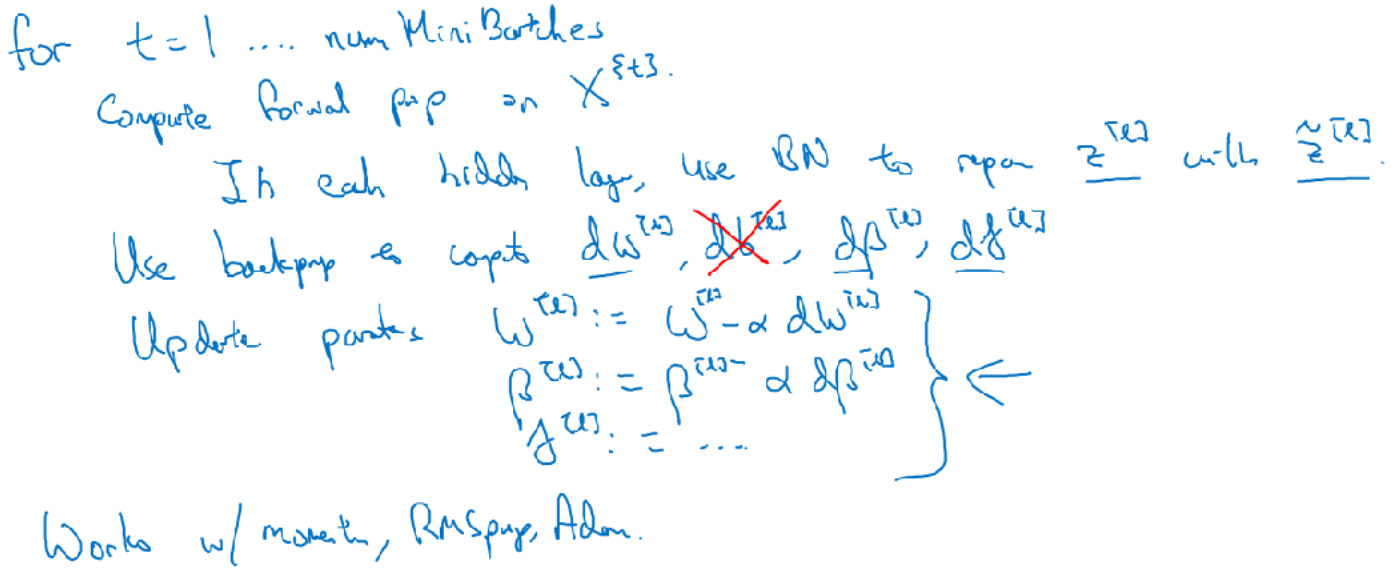
$X \xrightarrow{w^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{BatchNorm (BN)}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{w^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[BN]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \rightarrow a^{[2]} \rightarrow \dots$

Parameters: $w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}, \dots, w^{[L]}, b^{[L]}$
$\rightarrow \beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots, \beta^{[L]}, \gamma^{[L]}$
$\rightarrow \beta$

$d\beta^{[l]}$ $\quad \beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$

tf.nn.batch_normalization

Implementing gradient descent

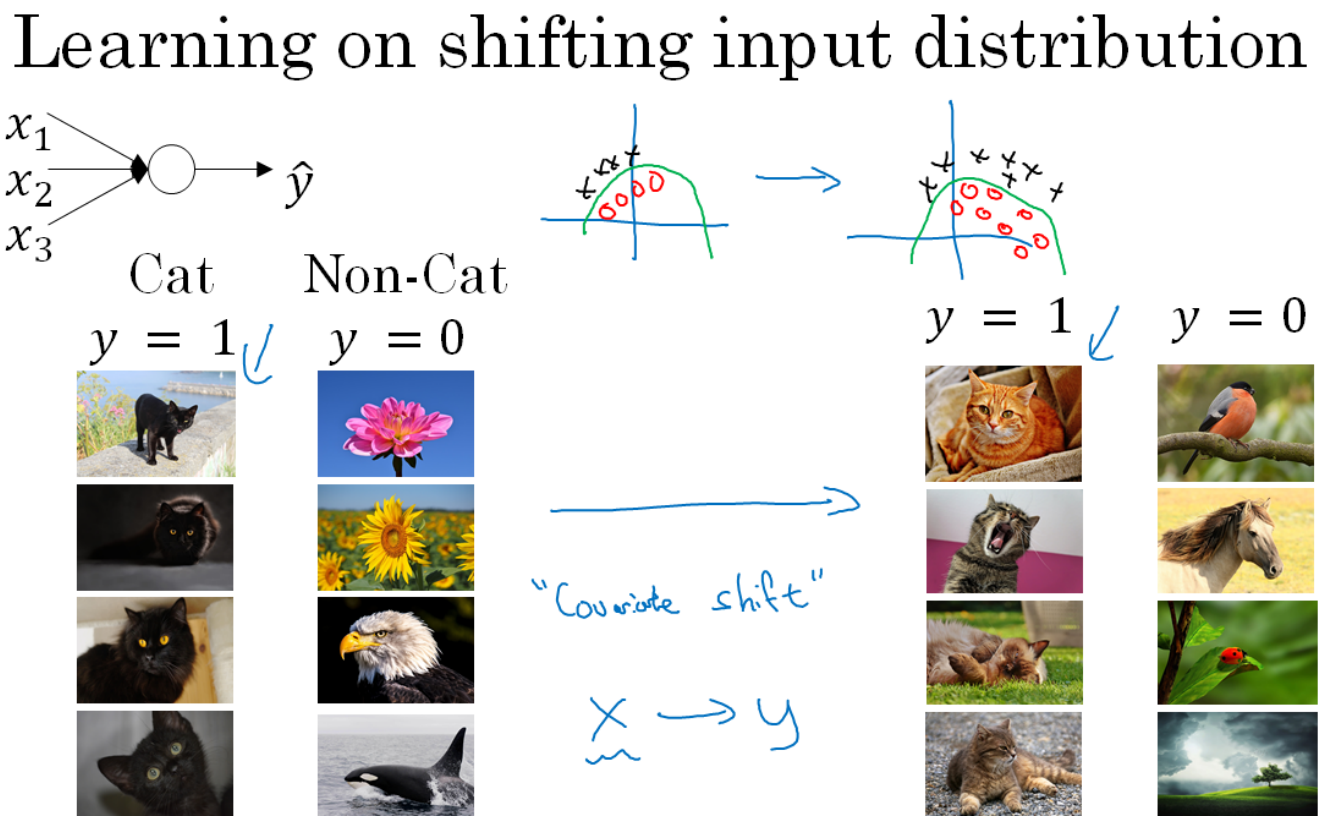


In programming framework like TF implementing Batch-Norm is just a single line of code.

Why does Batch Norm work?

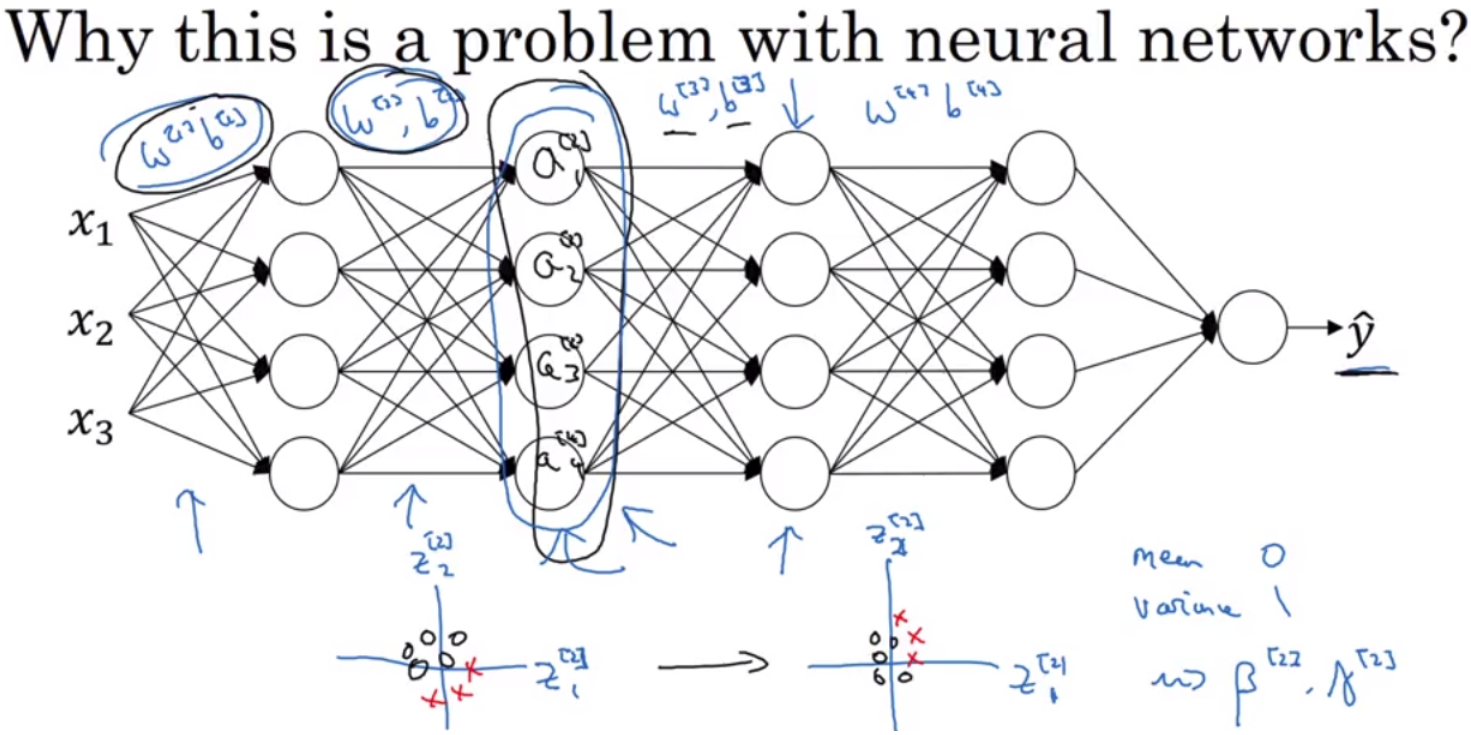
So, why does batch norm work? Here's one reason, we've seen how normalizing the input features, the X's, to mean zero and variance one, how that can speed up learning. So rather than having some features that range from zero to one, and some from one to a 1,000, by normalizing all the features, input features X, to take on a similar range of values that can speed up learning. So, one intuition behind why batch norm works is, this is doing a similar thing, but further values in your hidden units and not just for your input there. Now, this is just a partial picture for what batch norm is doing. There are a couple of further intuitions, that will help you gain a deeper understanding of what batch norm is doing.

Let's take a look at these in this section. A second reason why **batch norm** works, is it makes weights, later or deeper than your network, say the weight on layer 10, more robust to changes to weights in earlier layers of the neural network, say, in layer one. To explain what I mean, let's look at this most vivid example.

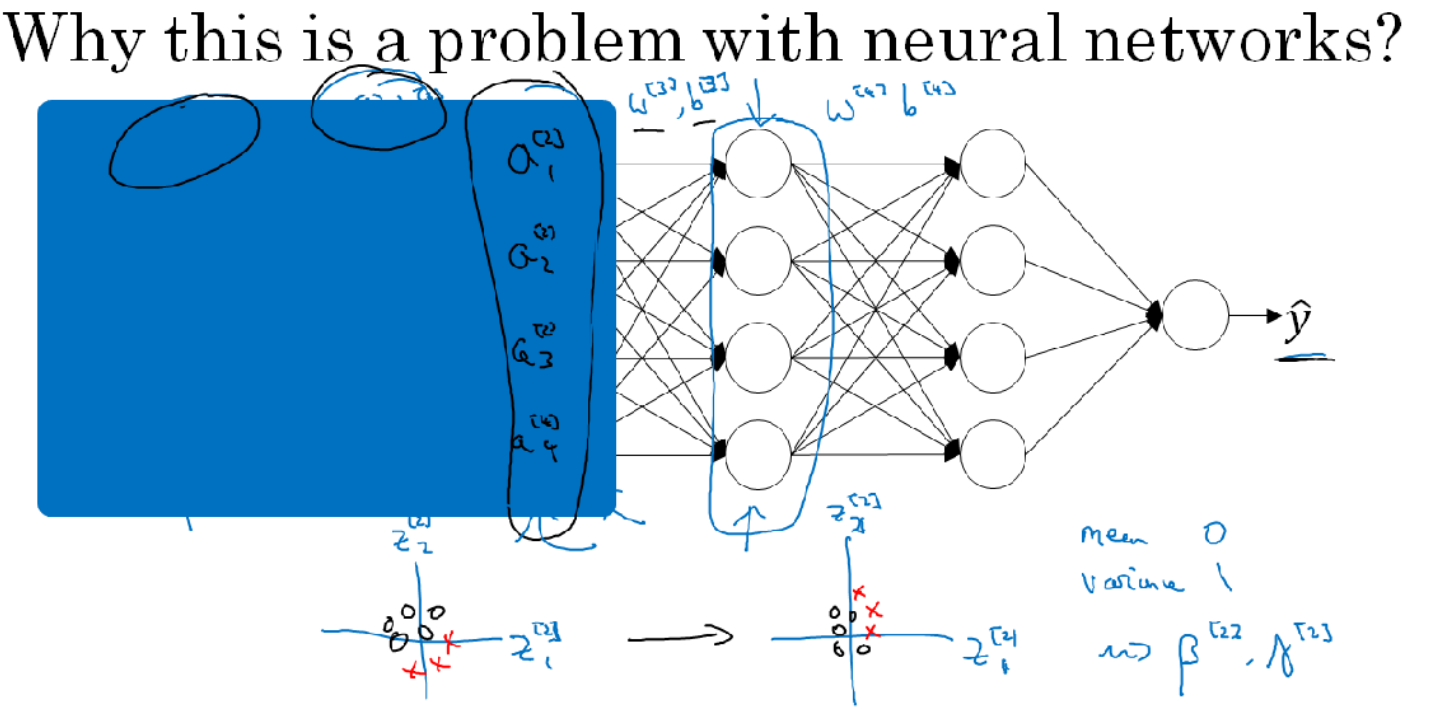


Let's see a training on network, maybe a shallow network, like logistic regression or maybe a deep network, on our famous cat detection toss. But let's say that you've trained our data sets on all images of black cats. If you now try to apply this network to data with colored cats where the positive examples are not just black cats like on the left, but to color cats like on the right, then your classifier might not do very well.

So in pictures, if your training set looks like (x and y axis drawing 1 in image), where you have positive examples (check image) and negative examples (check image), but you were to try to generalize it, to a data set where maybe positive examples are here and the negative examples are here, then you might not expect a module trained on the data on the left to do very well on the data on the right. Even though there might be the same function that actually works well, but you wouldn't expect your learning algorithm to discover that green decision boundary, just looking at the data on the left. So, this idea of your data distribution changing goes by the somewhat fancy name, **covariate shift**. And the idea is that, if you've learned some X to Y mapping, if the distribution of X changes, then **you might need to retrain your learning algorithm**. And this is true even if the function, the ground true function, mapping from X to Y, remains unchanged, which it is in this example, because the ground true function is, is this picture a cat or not. And the need to retain your function becomes even more acute or it becomes even worse if the ground true function shifts as well. So, how does this problem of covariate shift apply to a neural network? Consider a deep network like below

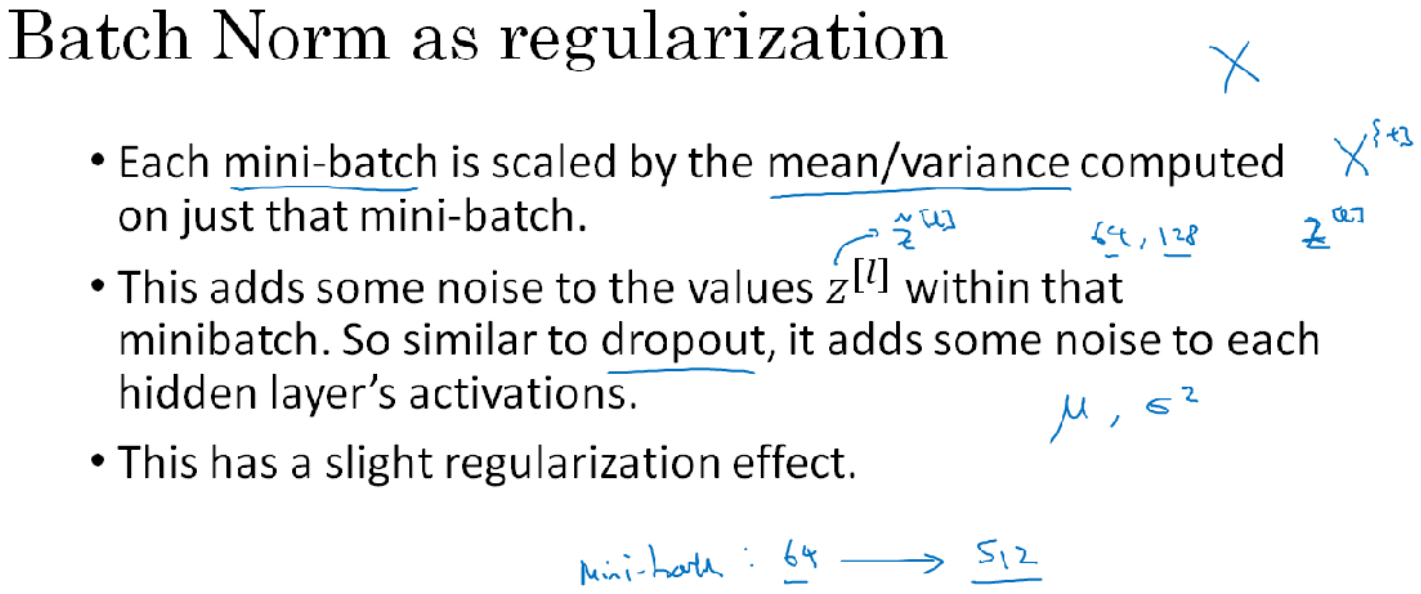


and let's look at the learning process from the perspective of this certain layer, the third hidden layer. So this network has learned the parameters $W^{[3]}$ and $B^{[3]}$. And from the perspective of the third hidden layer, it gets some set of values from the earlier layers, and then it has to do some stuff to hopefully make the output Y-hat close to the ground true value Y. So let me cover up the nose on the left for a second.



So from the perspective of this third hidden layer, it gets some values, let's call them $a1^{[2]}$, $a2^{[2]}$, $a3^{[2]}$, $a4^{[2]}$ but these values might as well be features x1, x2, x3, x4, and the job of the third hidden layer is to take these values and find a way to map them to Y-hat. So you can imagine doing great intercepts, so that these parameters $W^{[3]}$ $b^{[3]}$ as well as maybe $W^{[4]}$ $b^{[4]}$, and even $W^{[5]}$ $b^{[5]}$, maybe try and learn those parameters, so the network does a good job, mapping from the values we drew in black on the left to the output values Y-hat. But now let's uncover the left of the network again (check first diagram of the section). The network is also adapting parameters $W^{[2]}$ $b^{[2]}$ and $W^{[1]}$ $b^{[1]}$, and so as these parameters change, these values, $a^{[2]}$, will also change. So from the perspective of the third hidden layer, these hidden unit values are changing all the time, and so it's suffering from the **problem of covariate shift** that we talked about on the previous section. So what **batch norm does, is it reduces the amount that the distribution of these hidden unit**

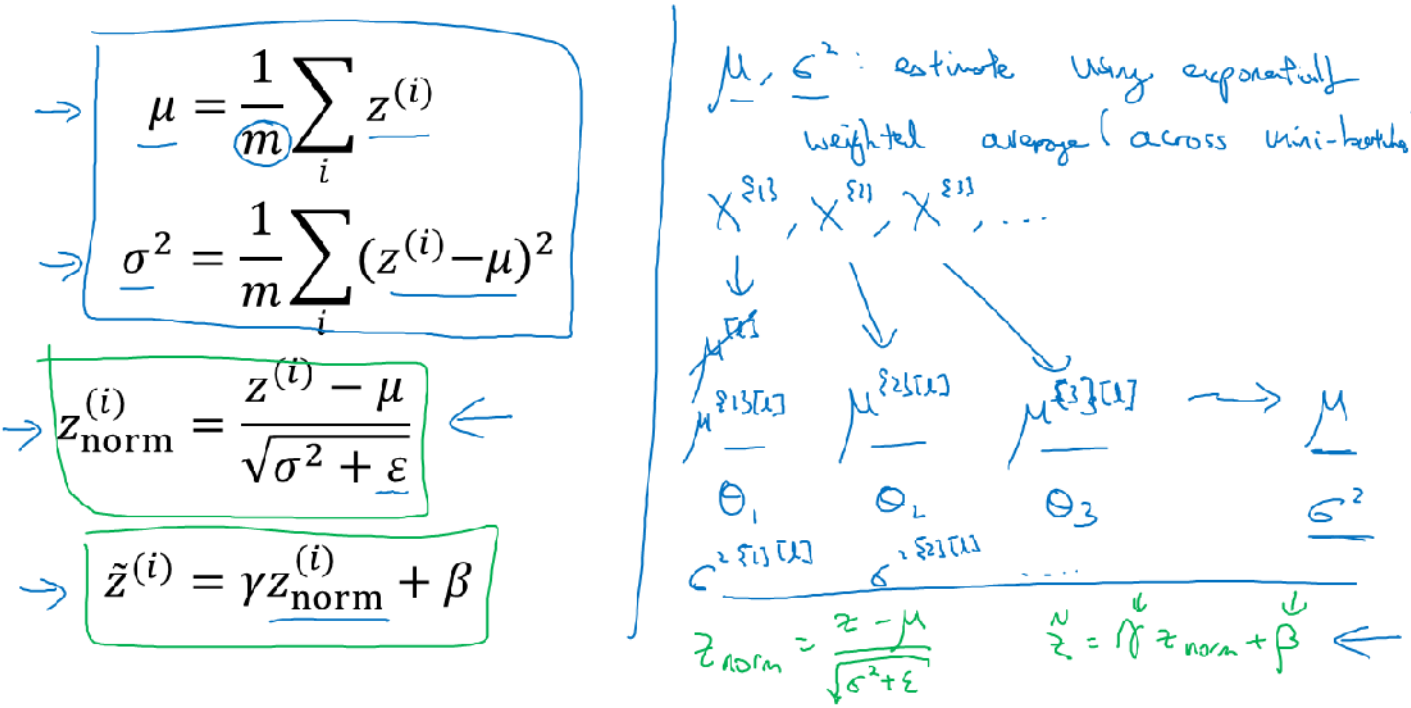
values shifts around. And if it were to plot the distribution of these hidden unit values, maybe this is technically we normalize Z, so this is actually $z1^{[2]}$ $z2^{[2]}$ and I also plot two values instead of four values, so we can visualize in 2D. **What batch norm is saying is that, the values for** $z1^{[2]}$ $z2^{[2]}$ can change, and indeed they will change when the neural network updates the parameters in the earlier layers. **But what batch norm ensures is that no matter how it changes, the mean and variance of $z1^{[2]}$ and $z2^{[2]}$ will remain the same**. **So even if the exact values of $z1^{[2]}$ and $z2^{[2]}$ change, their mean and variance will at least stay same mean zero and variance one**. Or, not necessarily mean zero and variance one, but whatever value is governed by beta 2 and gamma 2 Which, if the neural networks chooses, can force it to be mean=0 and variance=1 or really, any other mean and variance. But what this does is, it limits the amount to which updating the parameters in the earlier layers can affect the distribution of values that the third layer now sees and therefore has to learn on. And **so, batch norm reduces the problem of the input values changing, it really causes these values to become more stable, so that the later layers of the neural network has more firm ground to stand on**. And even though the input distribution changes a bit, it changes less, and what this does is, even as the earlier layers keep learning, the amounts that this forces the later layers to adapt to as early as layer changes is reduced or, if you will, it weakens the coupling between what the early layers parameters has to do and what the later layers parameters have to do. **And so it allows each layer of the network to learn by itself, a little bit more independently of other layers, and this has the effect of speeding up of learning in the whole network**. Takeaway is that batch norm means that, especially from the perspective of one of the later layers of the neural network, the earlier layers don't get to shift around as much, because they're constrained to have the same mean and variance. And so this makes the job of learning on the later layers easier. It turns out batch norm has a second effect, it has a slight regularization effect. So one non-intuitive thing of a batch norm is that each mini-batch, the mean and variance computed on just that mini-batch as opposed to computed on the entire data set, **that mean and variance has a little bit of noise in it**, because it's computed just on your mini-batch of, say, 64, or 128, or maybe 256 or larger training examples. Batch norm works with mini-batch.



Batch Norm at test time

Batch norm processes your data one mini batch at a time, but the test time you might need to process the examples one at a time. Let's see how you can adapt your network to do that. Recall that during training, here are the equations you'd use to implement batch norm.

Batch Norm at test time



So the takeaway from this is that during training time mu and sigma squared are computed on an entire mini batch of say 64 engine, 28 or some number of examples. But that test time, you might need to process a single example at a time. So, the way to do that is to **estimate mu and sigma squared from your training set** and there are many ways to do that. You could in theory run your whole training set through your final network to get mu and sigma squared. But in practice, what **people usually do is implement and exponentially weighted average where you just keep track of the mu and sigma squared values you're seeing during training and use and exponentially the weighted average, also sometimes called the running average, to just get a rough estimate of mu and sigma squared and then you use those values of mu and sigma squared that test time to do the scale and you need the head and unit values Z**. In practice, this process is pretty robust to the exact way you used to estimate mu and sigma squared. So, I wouldn't worry too much about exactly how you do this and if you're using a deep learning framework, they'll usually have some default way to estimate the mu and sigma squared that should work reasonably well as well. But in practice, any reasonable way to estimate the mean and variance of your hidden and unit values Z should work fine at test. So, that's it for batch norm and using it. I think you'll be able to train much deeper networks and get your learning algorithm to run much more quickly. B

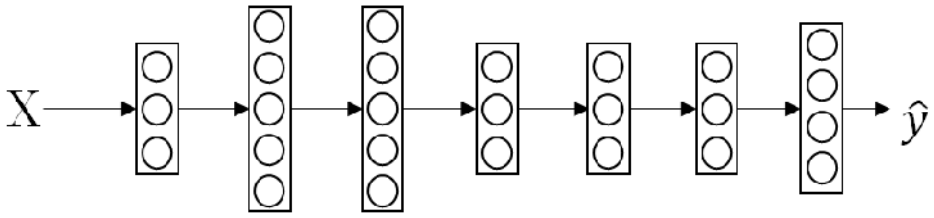
Multi-class classification

Softmax Regression

So far, the classification examples we've talked about have used binary classification, where you had two possible labels, 0 or 1. Is it a cat, is it not a cat? What if we have multiple possible classes? There's a generalization of logistic regression called **Softmax regression**. The less you make predictions where you're trying to recognize one of C or one of multiple classes, rather than just recognize two classes. Let's take a look. Let's say that instead of just recognizing cats you want to recognize cats, dogs, and baby chicks. So we are going to call cats class 1, dogs class 2, baby chicks class 3. And if none of the above, then there's an other or a none of the above class, which we are going to call class 0. So here's an example of the images and the classes they belong to.

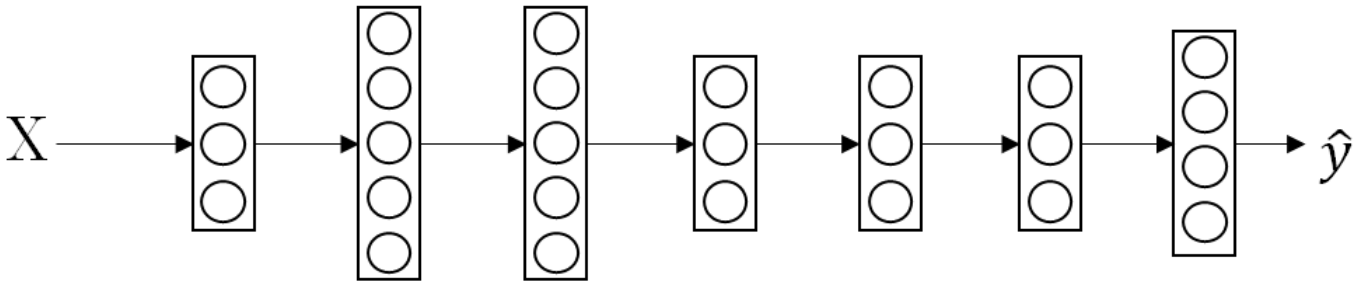
Recognizing cats, dogs, and baby chicks

3 1 2 0 3 2 0 1



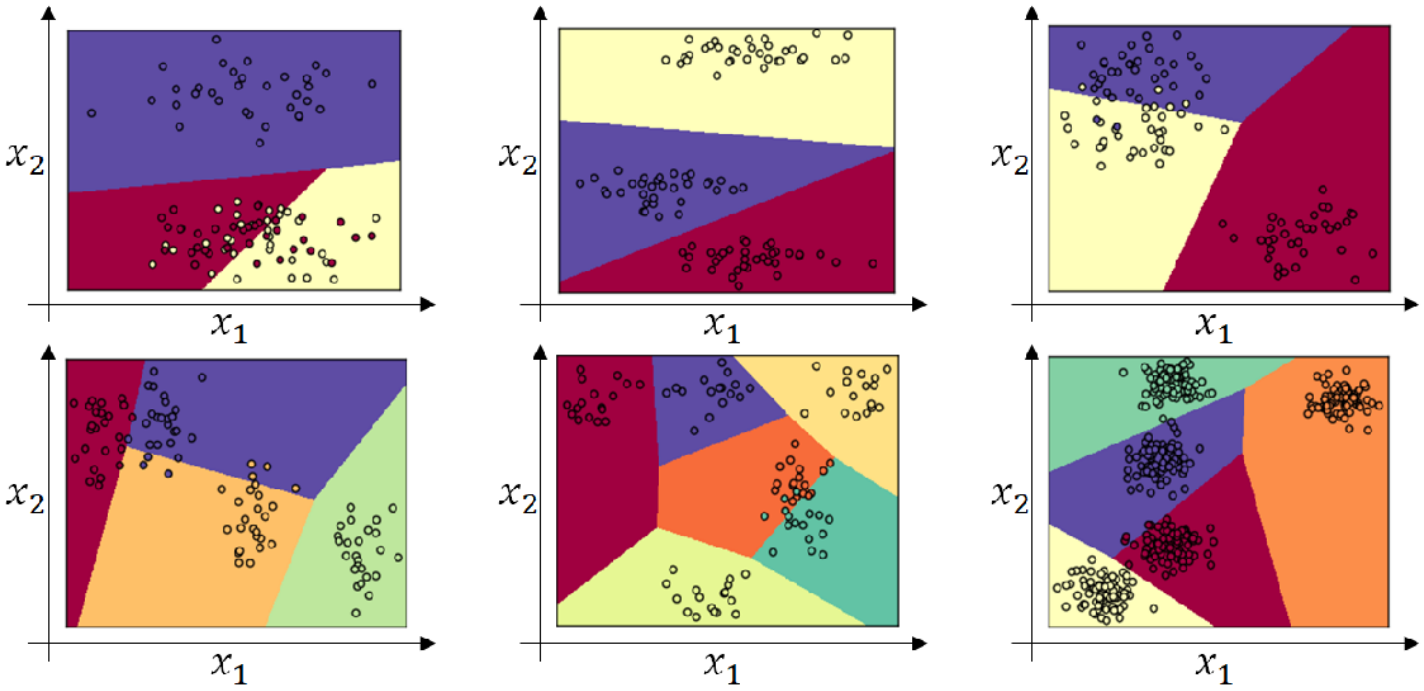
That's a picture of a baby chick, so the class is 3. Cats is class 1, dog is class 2, I guess that's a koala, so that's none of the above, so that is class 0, class 3 and so on. So the notation we're going to use is, we're going to use capital C to denote the number of classes you're trying to categorize your inputs into. And in this case, you have four possible classes, including the other or the none of the above class. So when you have four classes, the numbers indexing your classes would be 0 through capital C minus one. So in other words, that would be zero, one, two or three. In this case, we're going to build a new XY, where the upper layer has four, or in this case the variable capital alphabet C upward units. So N, the number of units upper layer which is layer L is going to equal to 4 or in general this is going to equal to C. And what we want is for the number of units in the upper layer to tell us what is the probability of each of these four classes. So the first node here is supposed to output, or we want it to output the probability that is the other class, given the input x, this will output probability there's a cat. Give an x, this will output probability as a dog. Give an x, that will output the probability. I'm just going to abbreviate baby chick to baby C, given the input x. So here, the output labels y hat is going to be a four by one dimensional vector, because it now has to output four numbers, giving you these four probabilities. And because probabilities should sum to one, the four numbers in the output y hat, they should sum to one. The standard model for getting your network to do this uses what's called a **Softmax layer,** and the output layer in order to generate these outputs. Then write down the map, then you can come back and get some intuition about what the Softmax there is doing.

Softmax layer



So in the final layer of the neural network, you are going to compute as usual the linear part of the layers. So z, capital L, that's the z variable for the final layer. So remember this is layer capital L. So as usual you compute that as wL times the activation of the previous layer plus the biases for that final layer. Now having computer z, you now need to apply what's called the **Softmax activation function**.
let's take a look at how you can train a neural network that uses a Softmax layer.

Softmax examples



The name softmax comes from contrasting it to what's called a **hard max** which would have taken the vector Z and matched it to a vector. So hard max function will look at the elements of Z and just put a 1 in the position of the biggest element of Z and then 0s everywhere else. And so this is a very hard max where the biggest element gets a output of 1 and everything else gets an output of 0. Whereas in contrast, a **softmax** is a more gentle mapping from Z to these probabilities. Softmax regression generalizes logistic regression to C classes. or we can say softmax regression is a generalization of logistic regression to more than two classes.