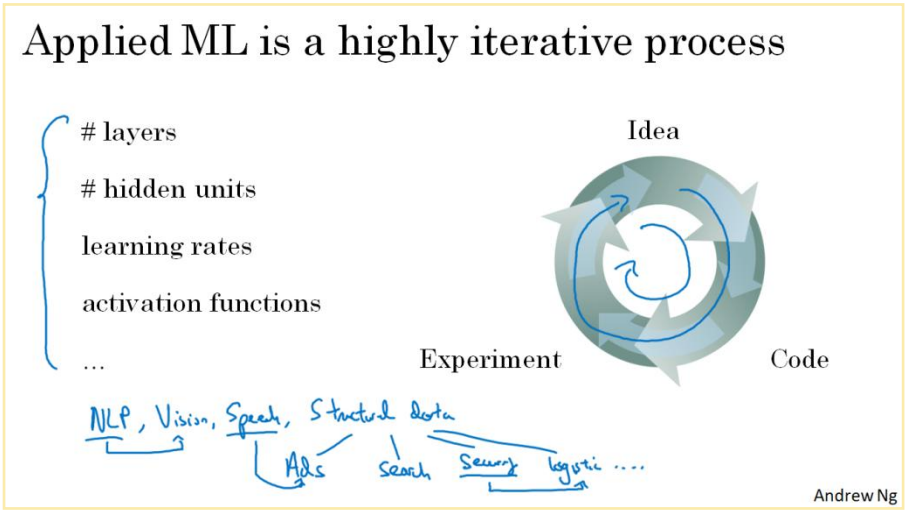
Course 2: Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

Content

1. Setting up your Machine Learning Application

We'll now see how to make NN work well. When training a neural network we have to make a lot of decisions:

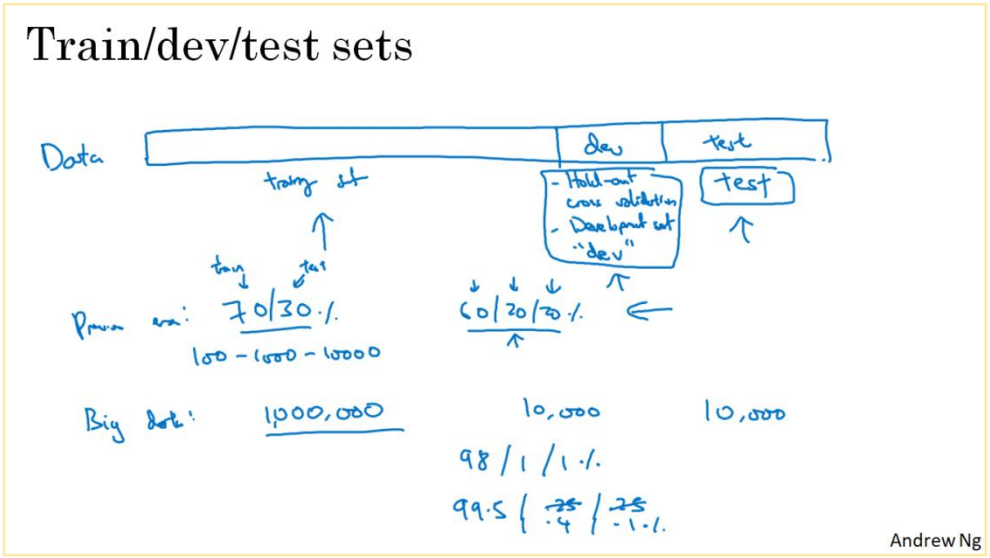


And it is also an iterative process. Deep Learning has succeeded in a lot of areas, but it turns out that the best choices in each problem or domain cannot be transferred to other ones. So we will usually need to go through the process and find the best combinations for our problems.

But what we can try is to go through this cycle **effectively**.

1.1. Train/dev/test set

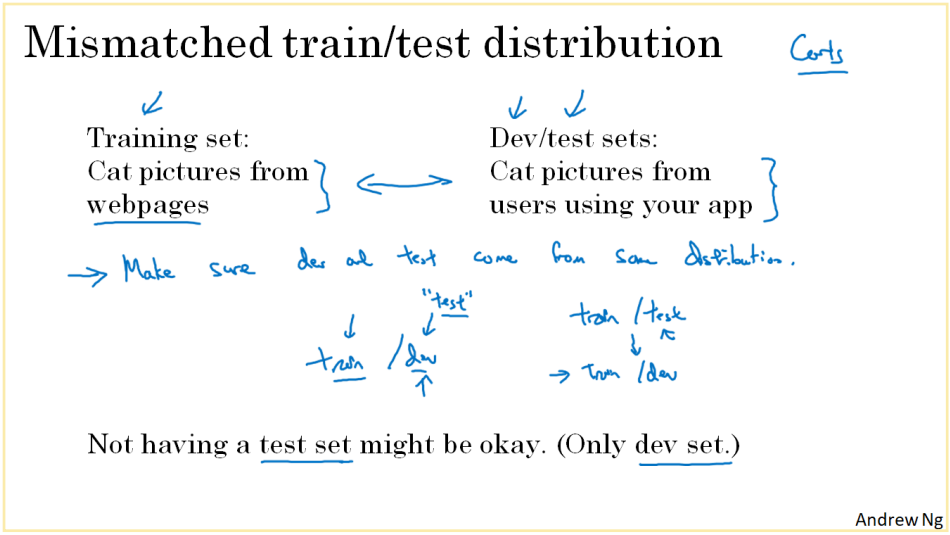
To begin with something, making a good choice when choosing between training, development and test set will help us find a high performance neural network.



We will train the different models on the training set and evaluate them on the dev set. Then we can keep our best model and test it on the test set.

A key difference from the traditional machine learning models is that since we expect to have many data, we no longer want a 60%/20%/20% distribution between sets. Maybe a 98%/1%/1% is adequate if we have a million data points and testing a model on 10.000 is enough.

Another issue we'll come across is **mismatched train/test set distributions.** That is, the data used in the sets is different (for example, we can train with high resolution photos but then test with low resolution pictures taken by users).

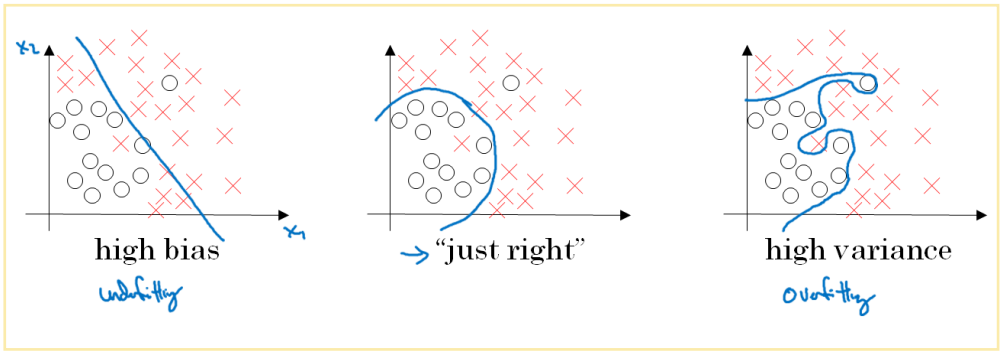


A single rule of thumb is that dev and test sets should come from the same distribution. Since we will be doing the hyperparameter tuning process on the dev set, we certainly want it to be like the test set.

Finally, it might be okay if we do not have a test set if we don't need an unbiased estimate of our accuracy.

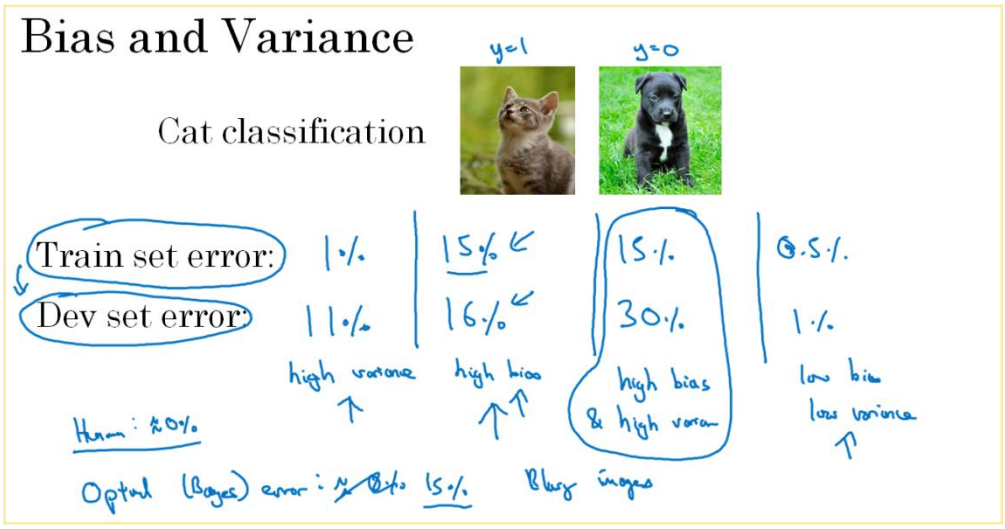
1.2. Bias – Variance

Let's show this example:



We can easily represent the concepts in 2-D space. But when we have more dimensions, we cannot. We will need to use some metrics.

Continuing with the example of cat classification:

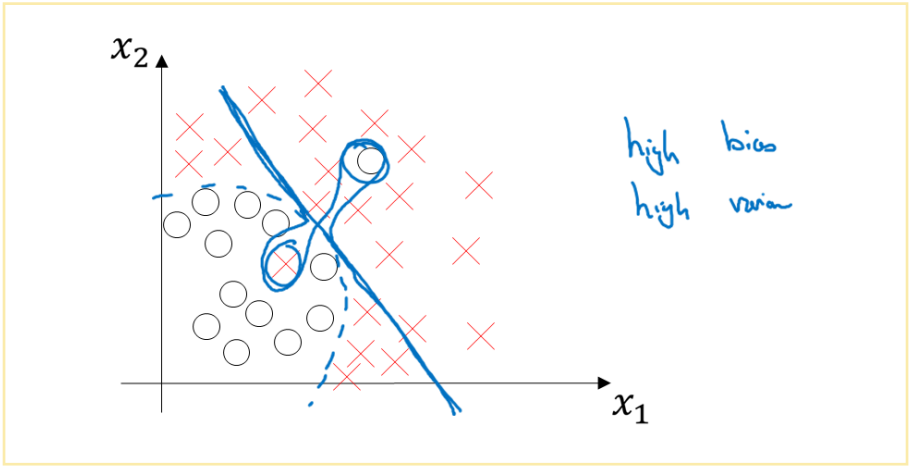


The two key metrics will be train set error and dev set error. If we have, for example, 1% error in train set and 11% error in dev set, we would say that we have high variance. Our model is overfitting the data. If we get high, similar errors then we have a biased model. On the third case, we are having both high bias and variance.

What we are trying to achieve is low train and dev errors, and it is important that both of them are similar.

This is assuming that the error rate we are trying to get near is 0% (i.e. humans can recognize cats with almost 0% error. In this case the optimal (Bayesian) error is 0%). If this rate was, for example, 15%, then the second classifier would be perfect.

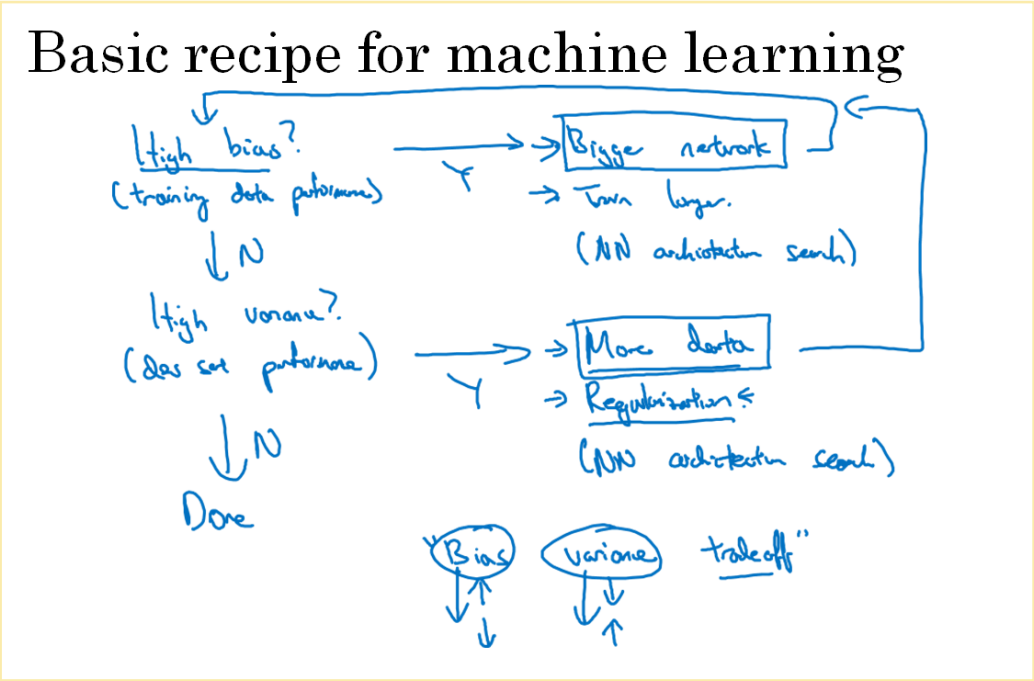
And, how high bias and high variance looks like?



It would be something like that: a classifier that misses several points and also does strange things at some points. Although this seems a bit strange on 2-D, in more dimensional spaces we can get models with both high bias and variance.

Now we know how to diagnose high bias or high variance in our model. The next step is to know what to do in those cases.

1.3. Basic Machine Learning Recipe



If we have high bias, meaning we are not being able to even fit well our training set, we will need to try a bigger network, or train it for a long time. In addition, we could try different NN architectures.

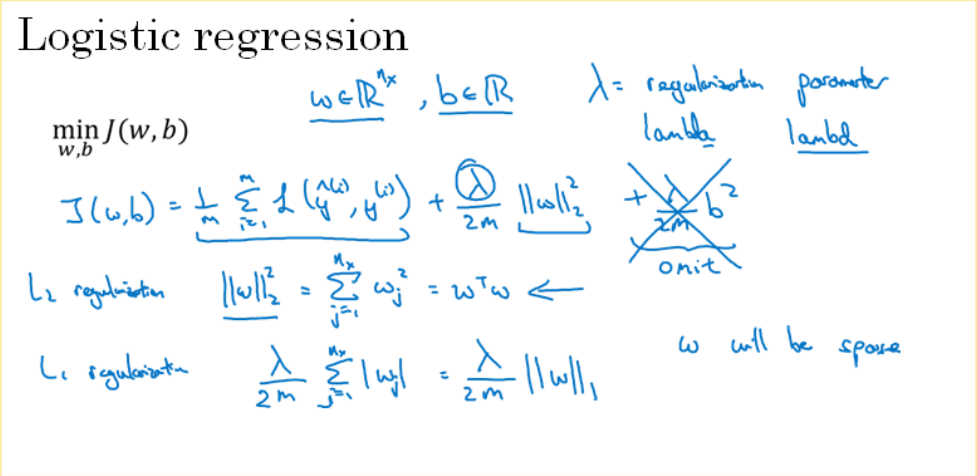
Once we don't have bias, if we have variance, we should go get more data or try regularization to prevent overfitting. We could also try different architectures.

One last thing to note is that the famous bias-variance trade off does not hold in the modern deep learning era. In the best cases, that trade off does not exist, meaning we can get low bias and low variance by getting more data or trying bigger networks. This means, if we have high bias and low variance, we can train a bigger network that will reduce it and will also not hurt the variance.

2. Regularization

Let's see how regularization works. We'll develop these ideas using logistic regression.

2.1. L-2 Regularization



In logistic regression, we can add that term to the cost function *J*. We will usually omit that term regarding to the *b* vector, since most parameters will be in the *w* matrix, and it won't make much of a difference.

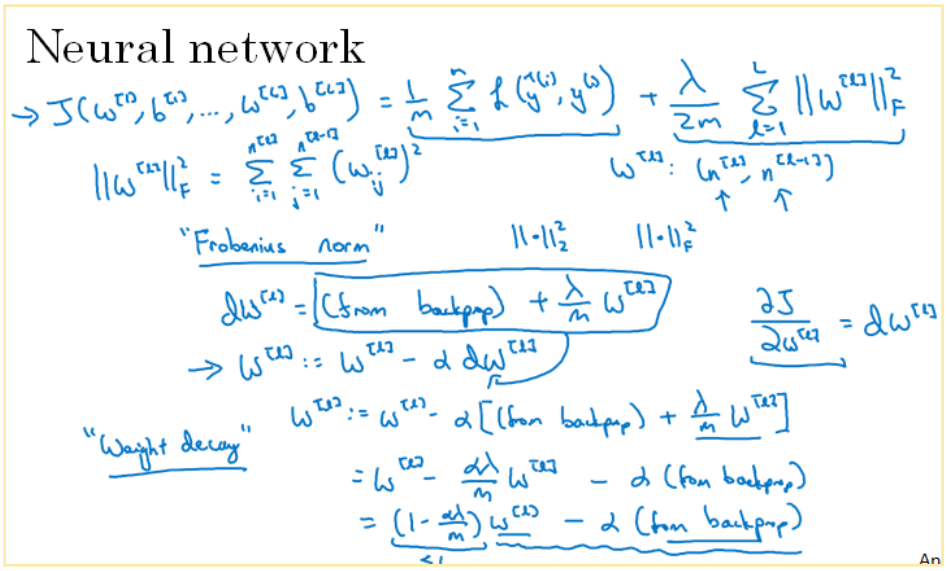
If we add the term with the L2-norm we get L2 regularization. But we can add a L1-norm term.

Adding L2 regularization will help *w* values decrease. Adding L1 regularization will make *w* a sparse matrix.

However, when tuning NN, L2 regularization is much more used.

Lambda is called the regularization parameter. We will determine it with cross-validation with the dev test.

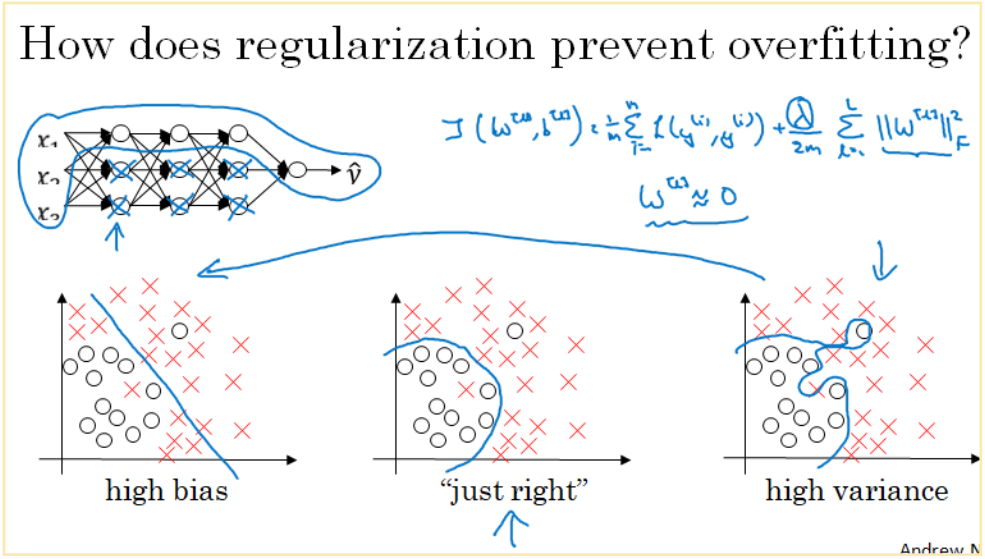
So, this is how we implement regularization in logistic regression: adding that term to the cost function. And how about the NN?



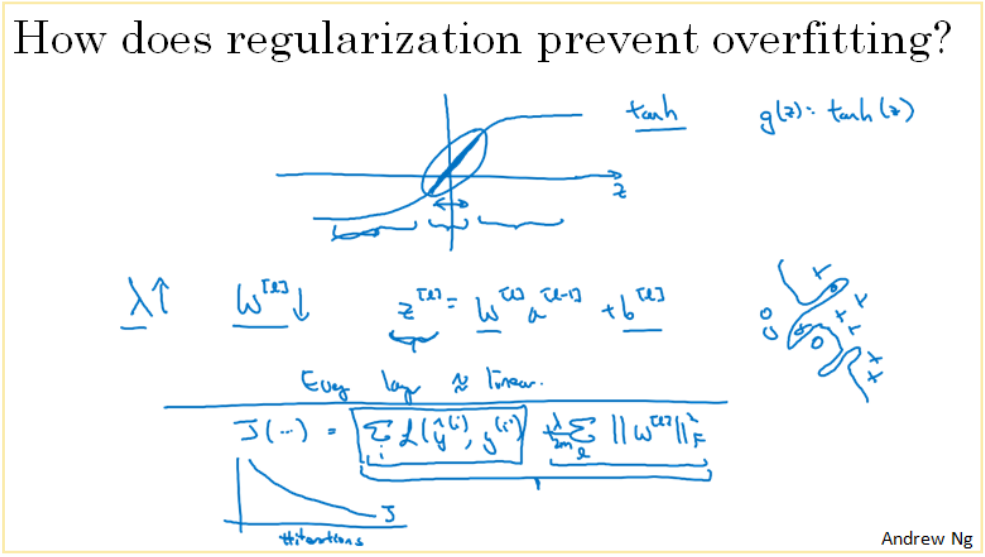
Now the term we add is the constant times the sum of the Frobenius norm over all layers. To implement it in the gradient descent, we just need to add the term as we can see in the slide. What we are really doing is taking *w* matrix and reducing its values. That's why it is also called *weight decay*.

But, why does reducing the *w* elements prevent overfitting? The intuition is simple: if we reduce the weights (let's think about reducing some of them to zero), we are disabling some neurons, and that leads us to a simpler network that may not be as prone to overfitting as the previous one.

Strictly, with the L2 regularization we don't set to zero the neurons, but in fact we reduce their effect.



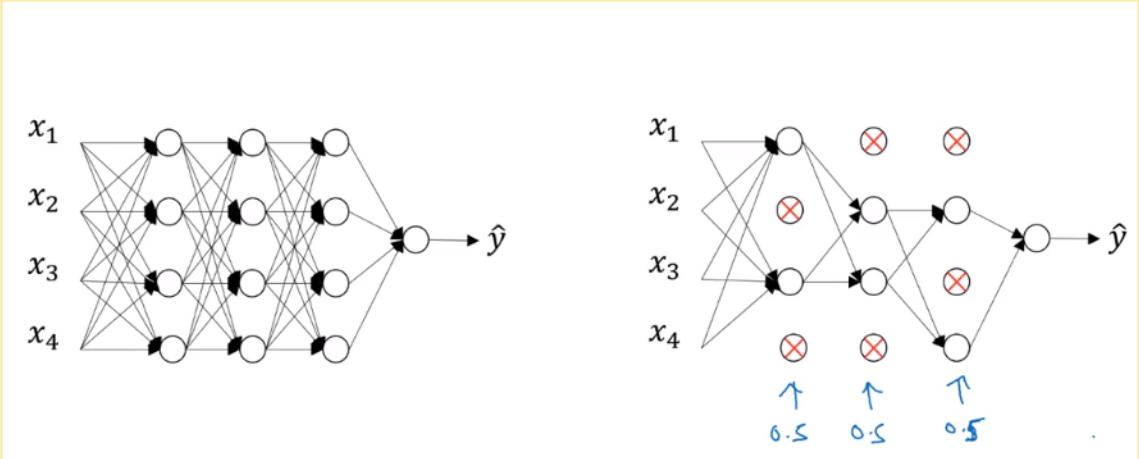
Let's see another intuition of regularization:



If we use *tanh* activation function, if lambda is large, then the weights will be small, which will cause z to be small as well, so we will move in the part of the domain of the function where it is roughly linear. And recall from course 1 that if the activation functions are linear, then the NN represents a simple linear transformation.

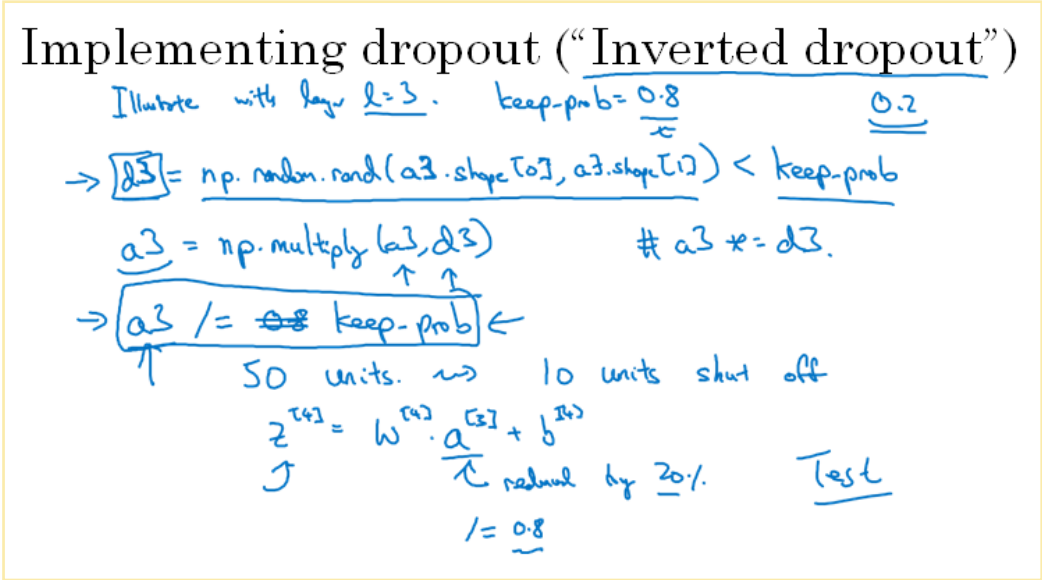
2.2. Drop-out regularization

We can use another method for preventing overfitting called dropout regularization:



We can just simply assign a probability of deleting nodes to each layer. For example, if we assign a 0.5 probability to each layer, we'll get the right-side NN, which turns out to be a much simpler one.

There are different techniques to implement dropout. We'll see the **inverted dropout**.

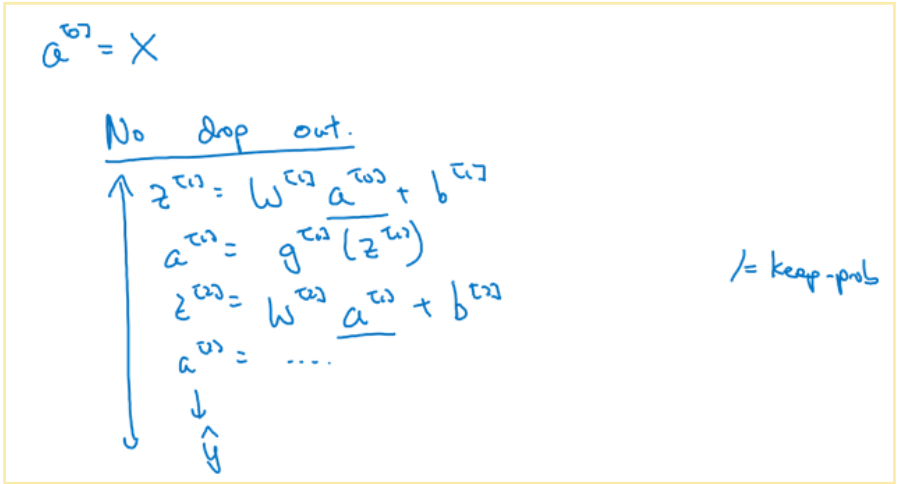


We are seeing how to implement it on layer 3. $d3$ is the dropout vector of the layer 3. We'll create a *keep_prob* variable representing the probability of keeping a node.

Then, we'll define the $a3$ activations as the multiplication of $a3$ times $d3$. Finally, we have to scale up $a3$ by dividing by *keep_prob*. This turns out to work well when predicting test data.

The point of this is that we'll change the probability on every iteration of gradient descent. This means that on each iteration we will be shutting off different neurons.

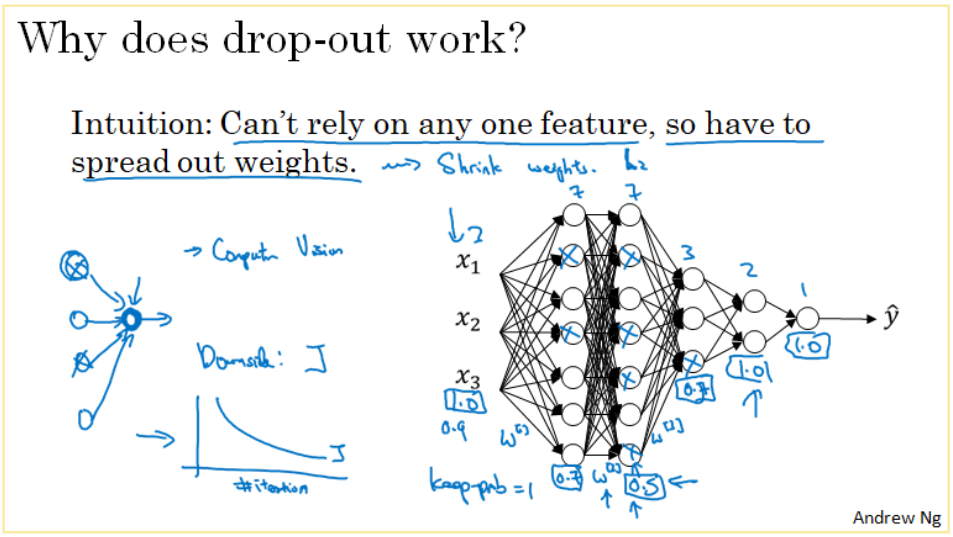
Then, having trained the algorithm, what should we do on test set?



We are **not** going to use dropout on test data, since it may introduce noise to our predictions. We could run a prediction process many times with different hidden units randomly dropped out and then take the average, but that's computationally inefficient and will give us roughly the same results.

We also have to note that because we have scaled the activations, we are using the effect of dropout when predicting test data (see slide).

Let's gain more intuition on why dropout works:



Let's think about a single neuron from the slide. Normally, it would take 4 inputs and return and output. But with dropout, at every iteration some of the inputs disappear. So that neuron will not rely on a single feature, but in fact spread the weights, which will lead to reducing the L2 norm of the weights.

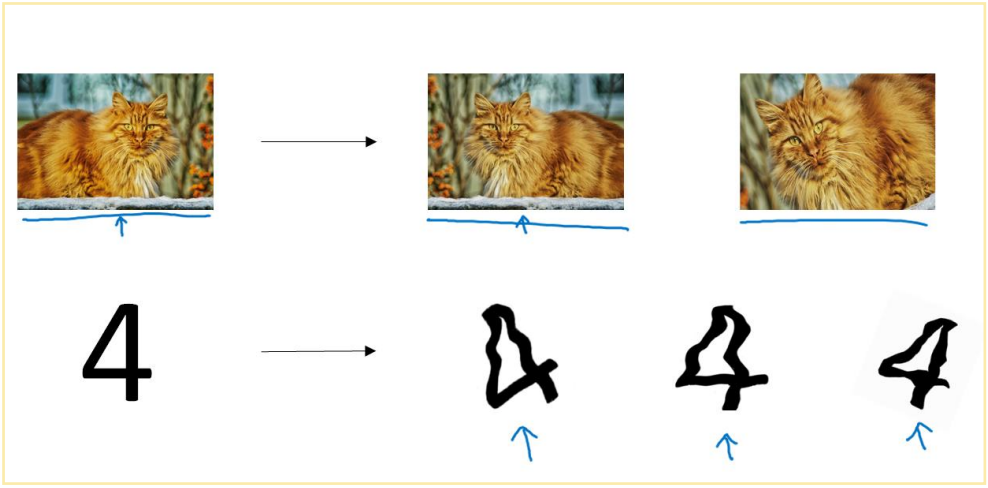
As we can see in the slide, we can use different *keep_prob* values for every layer. For the bigger layers we may have more overfitting since we may have a lot of parameters.

In addition, we don't usually use dropout for the input layer.

One big downside of drop out is that the cost function J is no longer well-defined. For this reason we can't rely on a graph of number of iterations vs cost as a debugging tool. So, what we can do is run the network without dropout and check the graph is descending. If it is, we turn on drop out and hope we have not had implementation errors.

2.3. Other regularization techniques

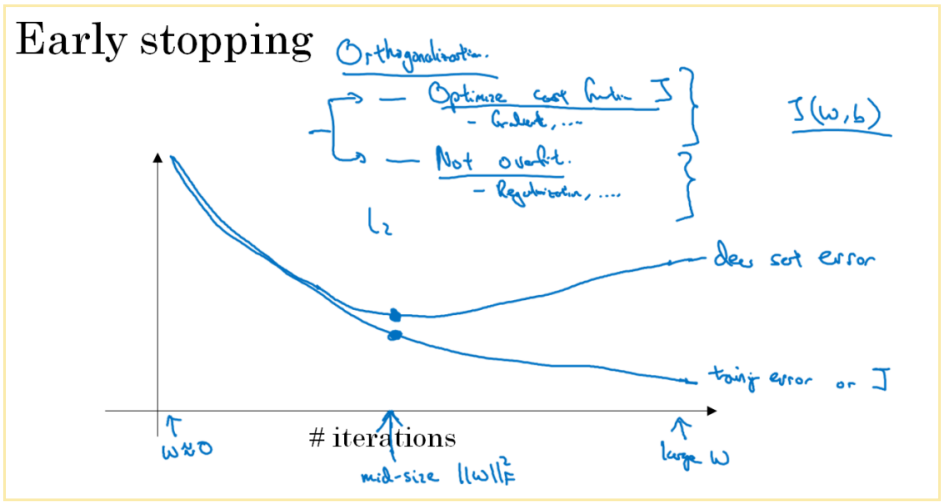
2.3.1. Data augmentation



This is not as good as getting more data, but can help.

2.3.2. Early stopping

We will plot the cost on training and dev set vs the number of iterations:



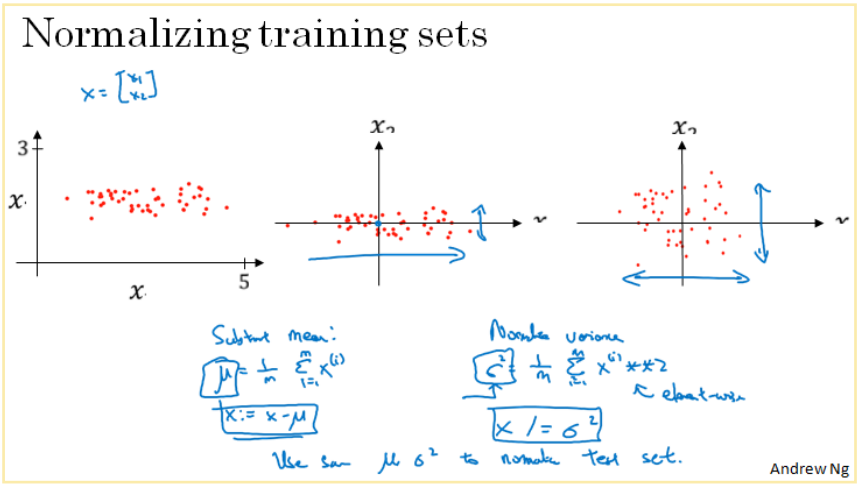
We could stop at the point where errors begin to evolve differently. But this has a downside, and it is related with the principle of **orthogonalization.** This principle divides machine learning in two independent tasks: reducing the cost (gradient descent) and not overfitting (regularization, etc…). And early stopping couples these two tasks, and you no longer can work on them separately.

Once we have seen methods to prevent overfitting, let's some techniques for setting up our optimization problem to make training go quickly.

3. Optimization

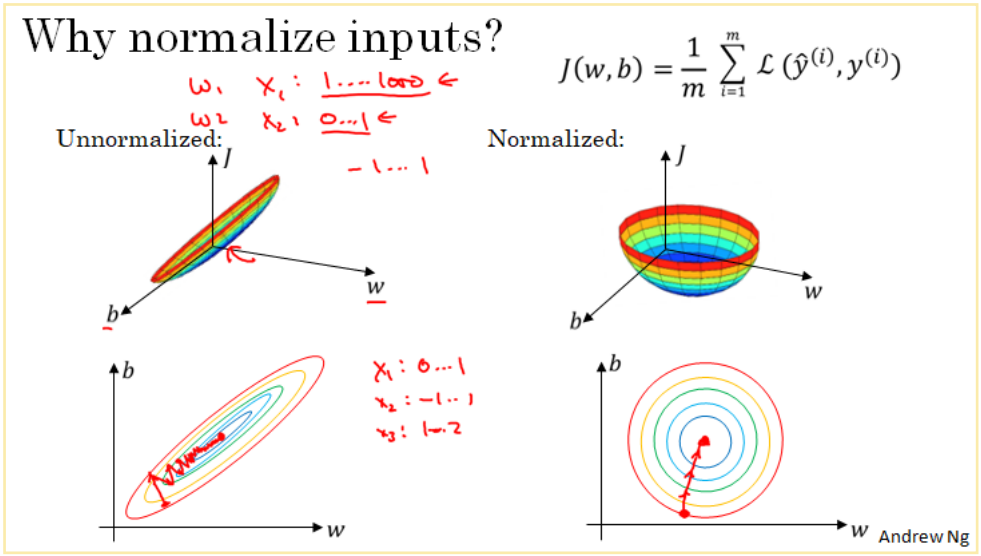
3.1. Normalizing inputs

Normalizing the inputs can speed up the training process. By normalizing we are doing the following:



Now, the variance of both variables is equal to 1 and they are centered (mean = 0). Remember that we should use the same parameters on both training and test sets, since we want them to go through the same transformations.

So why normalize inputs?

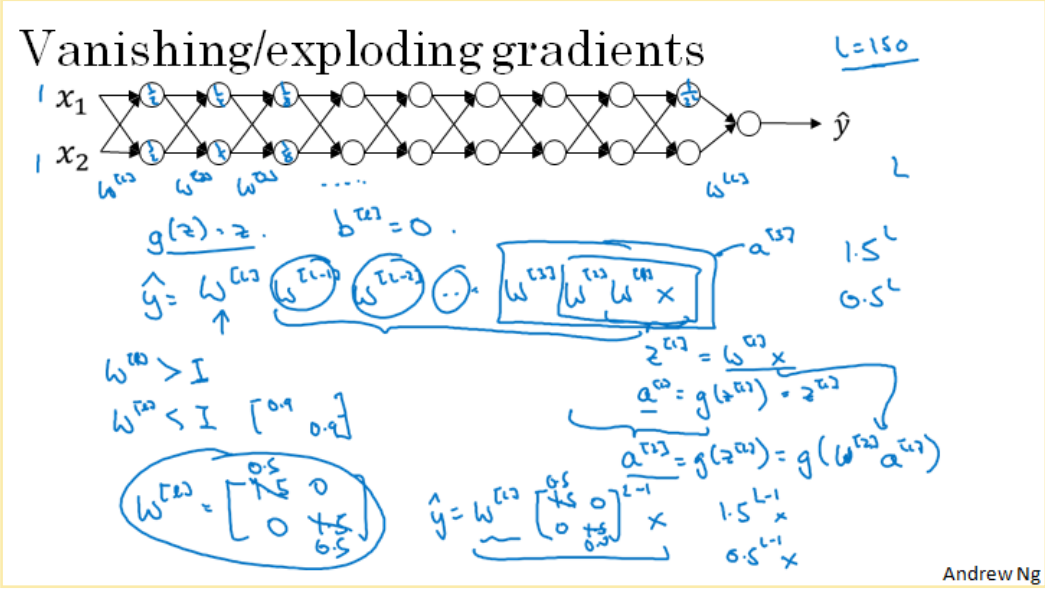


The cost function will look more symmetric, which will ease the training process as we can use higher learning rates.

If our features are in different scales we should use normalization. If they don't, we won't make much of a difference, but normalizing won't never do us harm, so we should do it anyway.

3.2. Vanishing / exploding gradients

One of the problems of training neural networks, specially very deep neural networks, is data vanishing and exploding gradients. What that means is that when you're training a very deep network derivatives or slopes can sometimes get either very, very big or very, very small, maybe even exponentially small, and this makes training difficult.

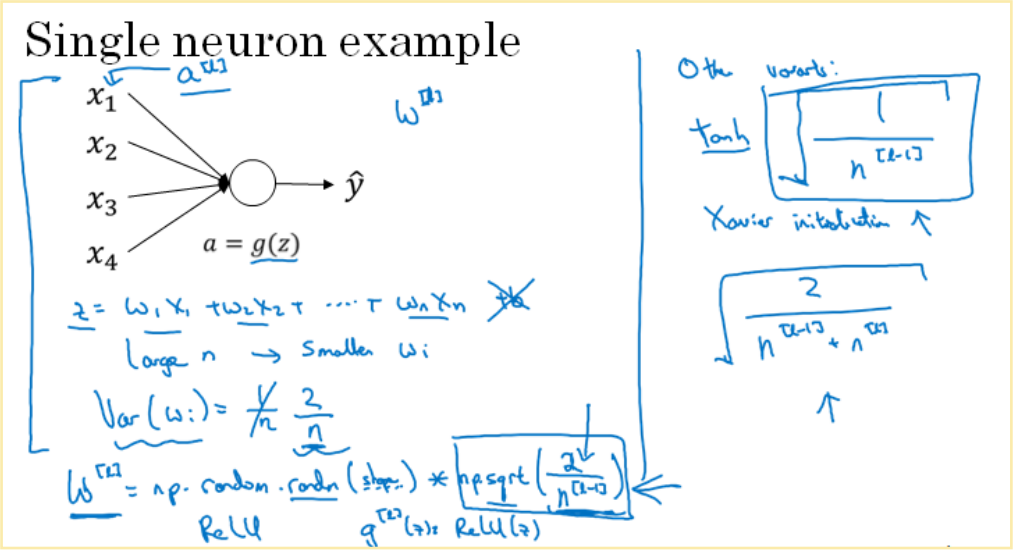


Let's work out an example. Let's assume we have the network of the slide, with linear activation functions and $b = 0$ on every layer. Under that conditions, $\hat{y}$ equals the formula above. If we assume the W matrices are a bit over or a bit under the identity matrix, then we can see that $\hat{y}$ becomes a function of the power of L. This means activations decrease or increase exponentially.

This is actually a problem. There is a partial solution consisting of carefully choosing the initial weights.

3.3. Weight initialization for deep networks

Let's first work this example with a single neuron:



If we want z not to be very large, the larger n is, the smaller each weight should be. We can force the variance to be $2/n$ (decreasing as n increases) (the 2 instead of a 1 is because in some cases it works better). Then, calculating the weights as the formula at the bottom of the slide would help to maintain the scale.

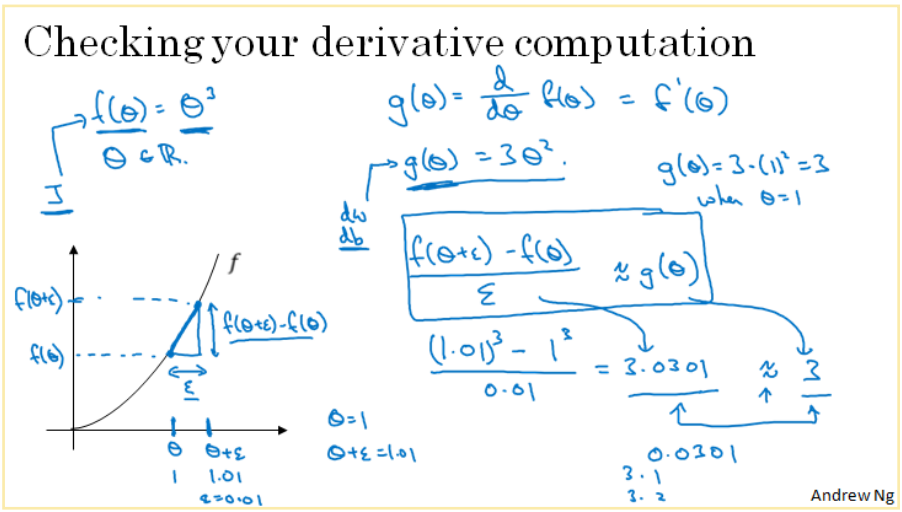
On the right side we have another ways of initialization.

- If we are using ReLu functions, we should use $\frac{2}{n}$
- If we are using tanh functions, we should use the Xavier initialization

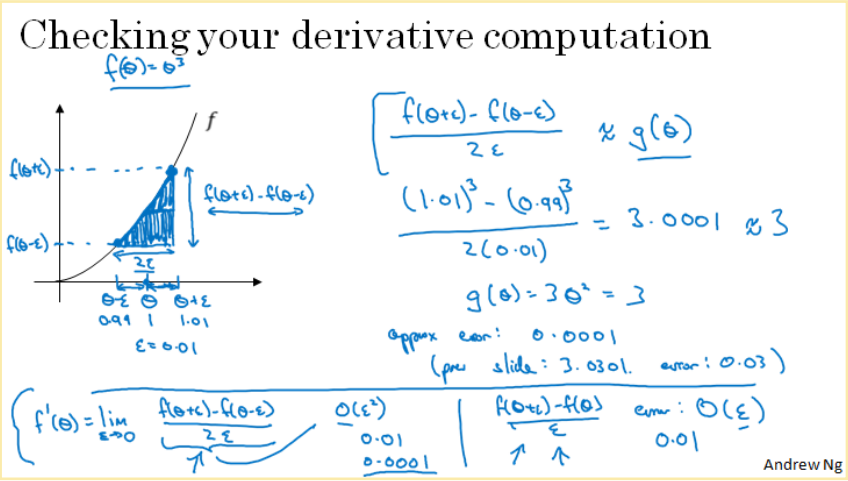
3.4. Gradient checking

Gradient checking is a technique that can help us make sure our implementation of backpropagation is correct. But before discussing it, let's first talk about how to numerically approximate computations of gradients.

We can approximate the derivative at a point as the highlighted formula:



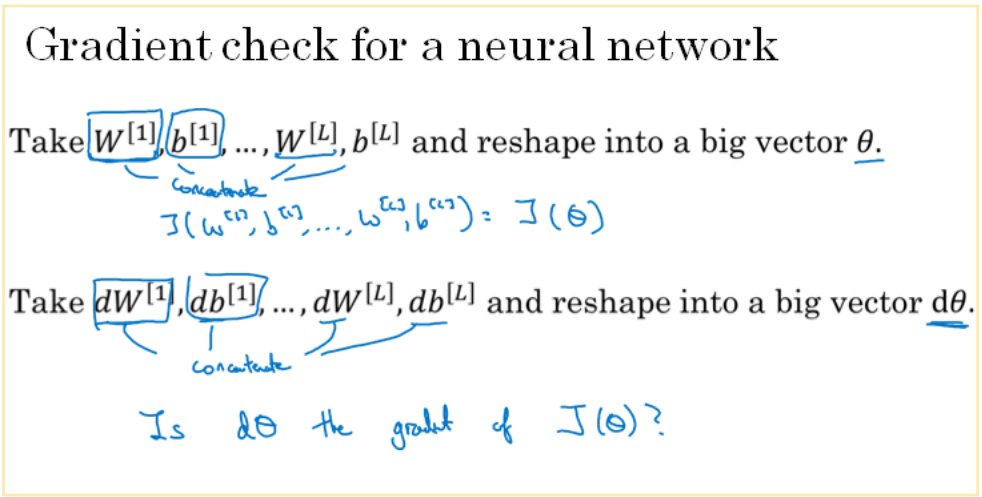
If we go in both ways with epsilon, we get a mathematically better approximation of the gradient:



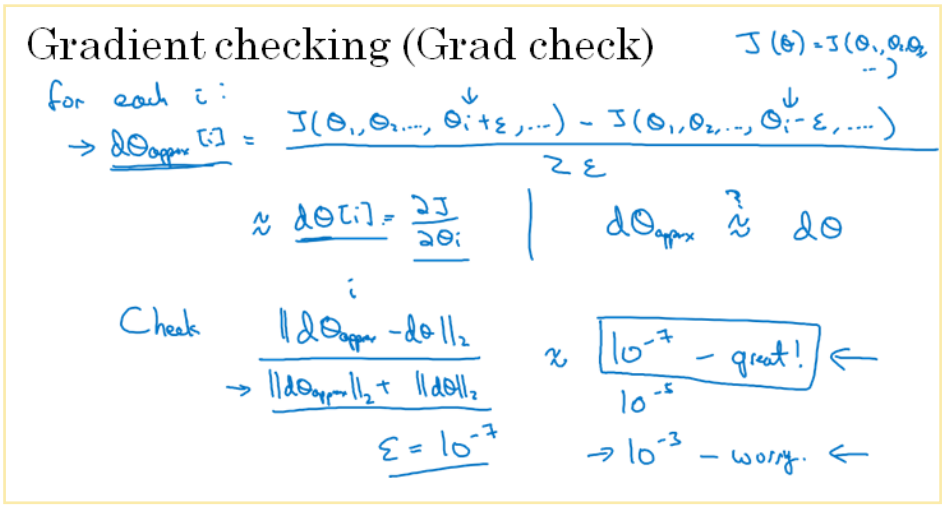
Using this, we can verify whether or not a function g is a correct implementation of the derivative of f at a point. Let's now see how we can use this to ensure our back propagation is correct.

First of all, we will take the weight matrices and reshape them into a big vector theta. This will allow us to express J as a function of a single variable.

Then, we should do the same with dθ, and ask ourselves the question: is dθ the gradient of J(θ)?

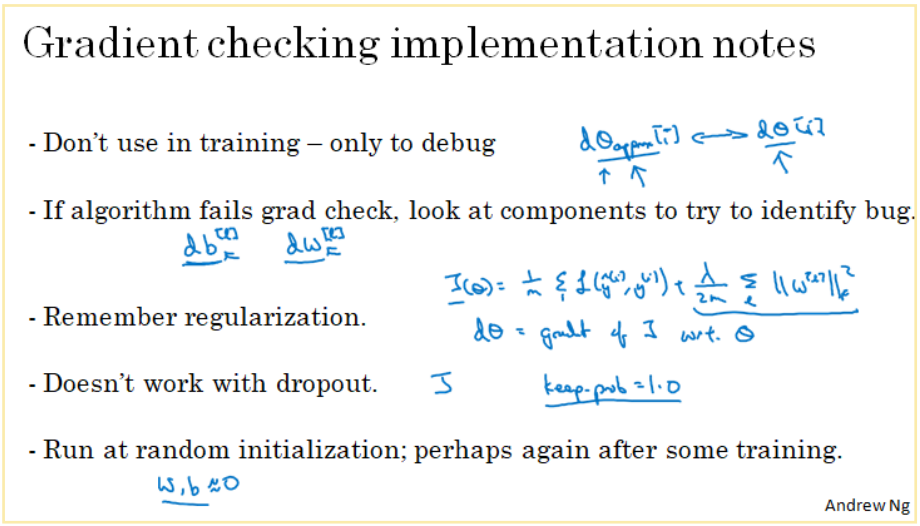


We can now apply the calculus concept we have just seen to answer this question:



For each I (each component of theta), we will implement the formula above and check if we are doing it correctly by calculating the distance between those two vectors.

Finally, let's see some tips on gradient checking:



- Only to debug
- We should look at the components of θ and dθ. If we see db are really different but dw not, perhaps we have a bug when computing b.
- If we are using regularization, we should include the regularization term when computing gradients.
- Because dropout randomly eliminates units, J does not have an analytical expression. That's why we cannot compute its derivative analytically. What we can do is carry grad check without dropout and then introduce it later.

4. Optimization algorithms

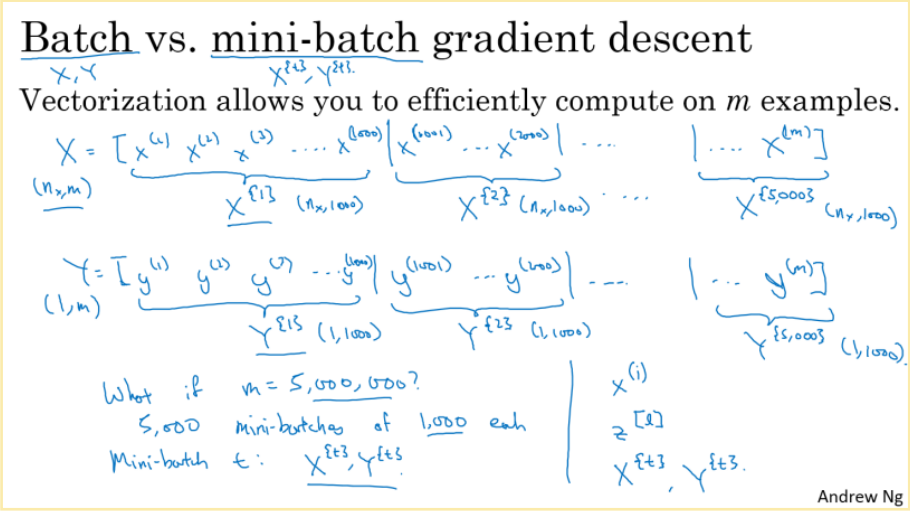
In this module we will learn optimization algorithms that will help us train our models much faster. Machine learning, as we have said, is a highly iterative process: we just have to try many combinations and stick to the better one. So it really helps to train models quickly.

In addition, deep learning does not work pretty well in a regime of big data. Training on a large dataset is just slow. So having good optimization algorithms will help us save a lot of time.

We'll begin with mini-batch gradient descent.

4.1. Mini-batch gradient descent

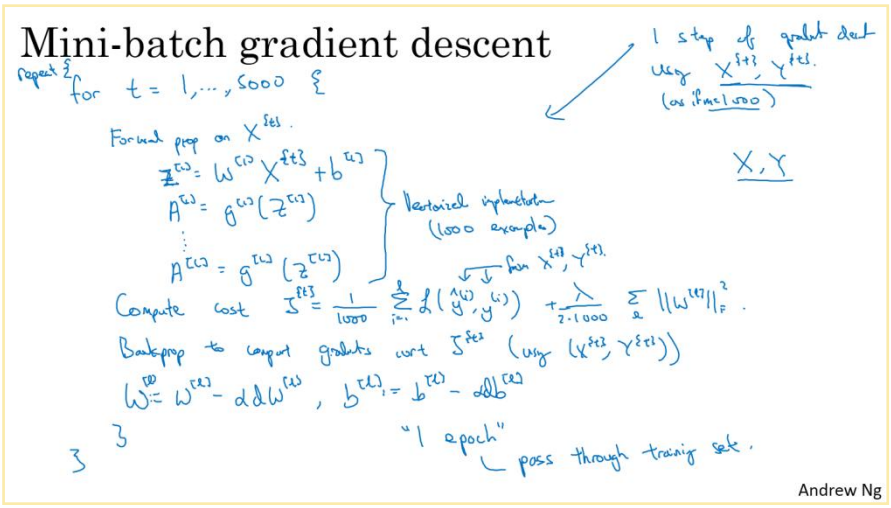
We have seen that vectorization allows us to efficiently compute m examples without an explicit for loop.



As we have seen, we would stack our examples into a matrix and perform the algebra operations.But what if we have a lot of examples? With the implementation of gradient descent on the whole training set, we would have to process the entire dataset on every step of gradient descent.

So we could get a faster algorithm if we don't process the entire dataset but a fraction of it. Suppose we divide our training set into subsets and call them mini-batches (check the above slide notation for clearance). Doing this is called **mini-batch gradient descent**. Taking the whole dataset at a time is called **batch gradient descent.**

Let's see how mini-batch gradient descent works:



We get a single batch and run gradient descent on it, as if it was the full training set. This means, we perform forward propagation, compute the cost function and then perform backpropagation to get the updated weights, but all of this with the data of a single batch. Then we run this for the whole set of batches (in the example, we have divided our 5.000.000 points training set into 5.000 batches of 1.000 points each.

Completing this for loop (meaning we have passed over the full training set – or, what is the same, all the batches) is called doing one epoch of training.

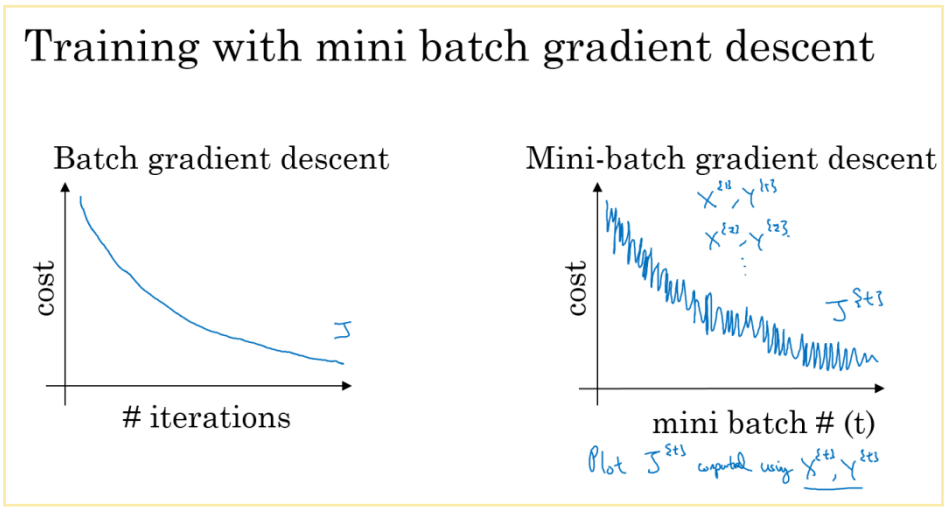
We can see that:

- With batch gradient descent, a single pass over the whole dataset lets us take one single step on the gradient.
- With mini-batch gradient descent, in this example, a single pass over the whole dataset (one epoch) has allowed us to take 5.000 steps.

However, we usually want to make multiple passes through the training set, so we will run a number of epochs.

It turns out that mini-batch gradient descent is much faster and it is what deep learning practitioners use when dealing with large datasets.But why?

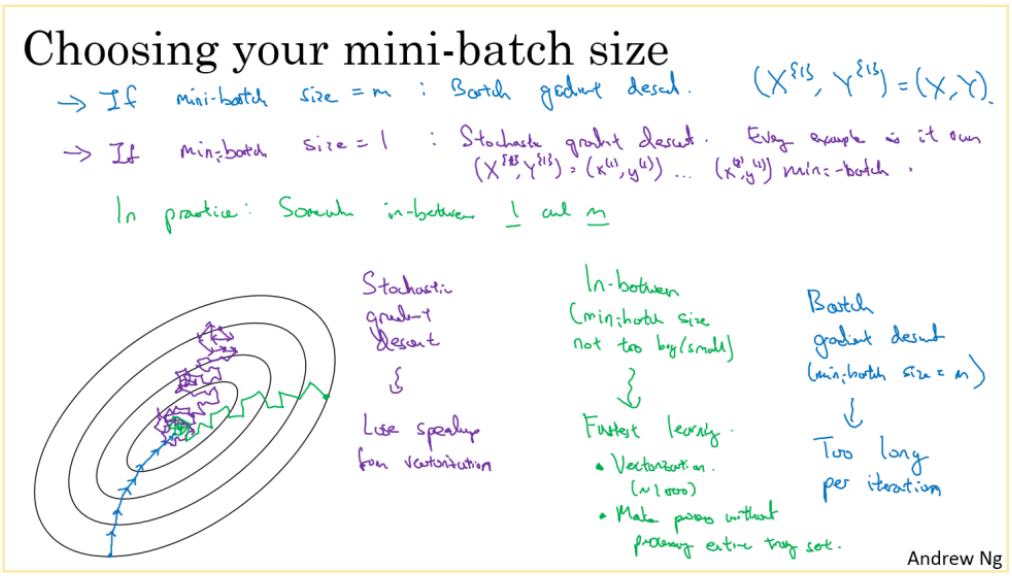
Understanding mini-batch gradient descent



With batch gradient descent, we would expect the cost to be decreasing on every iteration. If it goes up on a single iteration, it's wrong. Perhaps our learning rate is too big.

On mini-batch gradient descent, on every iteration we're processing a different subset of the data points, so we should see something like as in the figure.

One of the parameters we need to choose is the size of the mini-batches.



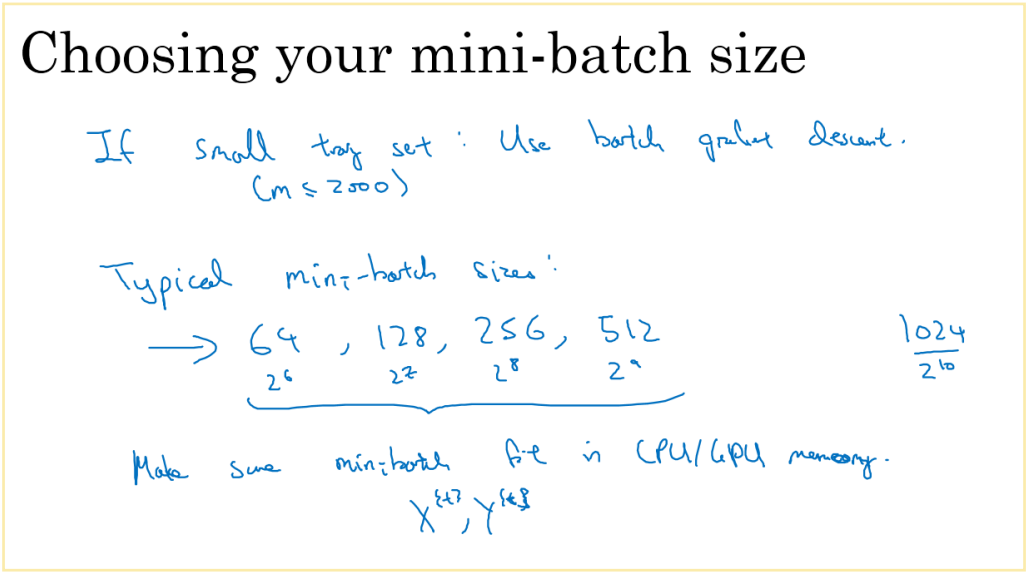
- If mini-batch size is m, then we have the batch gradient descent.
- If mini-batch size is equal to 1, this is called stochastic gradient descent. Every example is a mini-batch.

On a contour plot, these two extremes would look like as the image in the above image. SGD is very noisy but will on average lead us on the right direction. However, it will never converge. It won't get to the minimum and stay there. It will be oscillating on the minimum zone.

In practice, the mini-batch size will be in between of these two extremes:

- If we use batch gradient descent, our processing will take too long on every iteration if we have a very large dataset.
- If we use SGD, the main disadvantage is that we lose almost all of the speed-up vectorization gives us.

For this reason, laying in between makes us take advantage of vectorizationand also reduce the training time. So, the mini-batch size has to lay between 1 and m, which value should we choose? Here are some guidelines:



If we have a small dataset (let's say less than 2.000 examples) we should just use batch gradient descent.

If we don't, some typical sizes are 64, 128, 256 and 512. They are all powers of 2, because of the way computer memory is layed out and accessed.

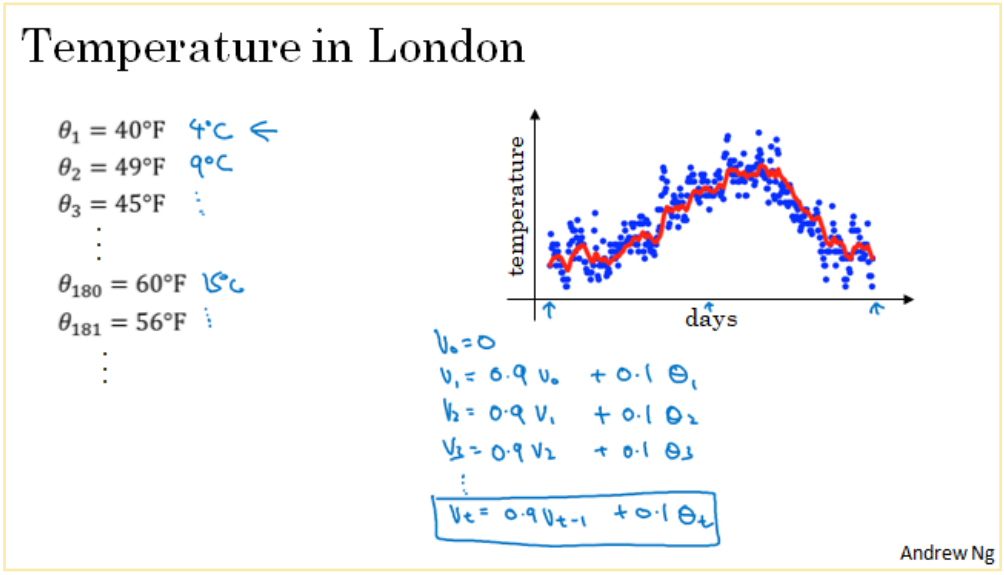
What we should to is just try: try several values and see which one makes the process more effective. In addition, we should make sure the mini-batch size fits into memory. If not, we will see a clear reduction of the performance.

It turns out that there are even more efficient algorithms than gradient descent or mini-batch gradient descent. Let's see them:

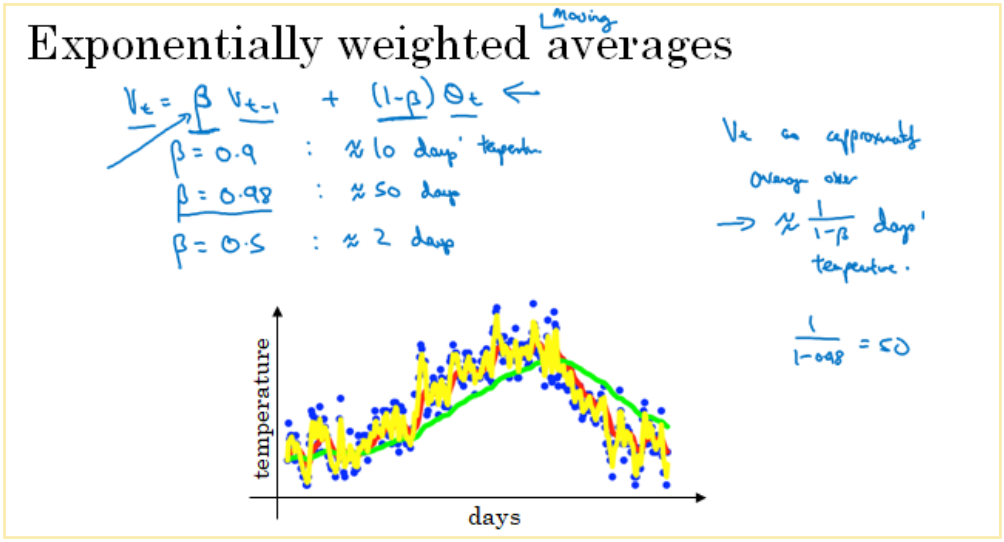
4.2. Exponentially weighted averages

We are going to see a few optimization algorithms that are faster than gradient descent. But in order to understand them, we need to be able to understand exponentially weighted averages.

Let's suppose we have the different temperatures in London:



This time series is noisy, so in order to extract the trend or the moving average, we could do as in the above slide. If we implement the formula, we get an exponentially weighted average of the daily temperature.

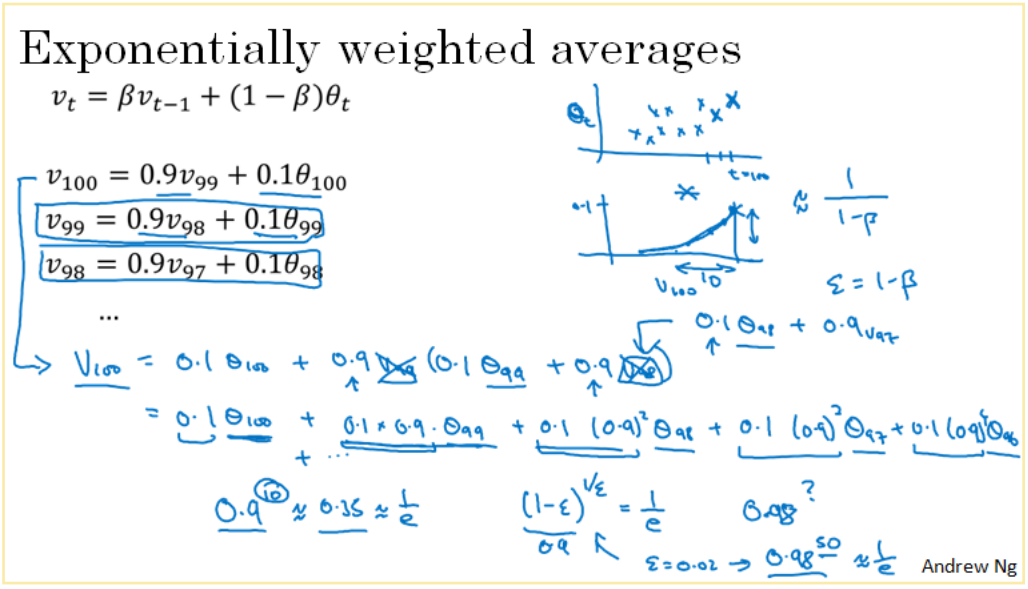


If we call ß the coefficient, it turns out that Vt is approximately averaging over $\frac{1}{1-\beta}$ days. For example, if ß is 0.9, we could think about averaging over the last 10 days.

If we set ß close to 1, we are averaging over a lot more days, which leads to a smoother plot. If we set it to 0.5, we ara averaging over two days, so we are much

noisy. Setting a lower ß lets us adapt more quickly to the current value. If ß is high, we will have latency, as the present value will not influence pretty much the value.

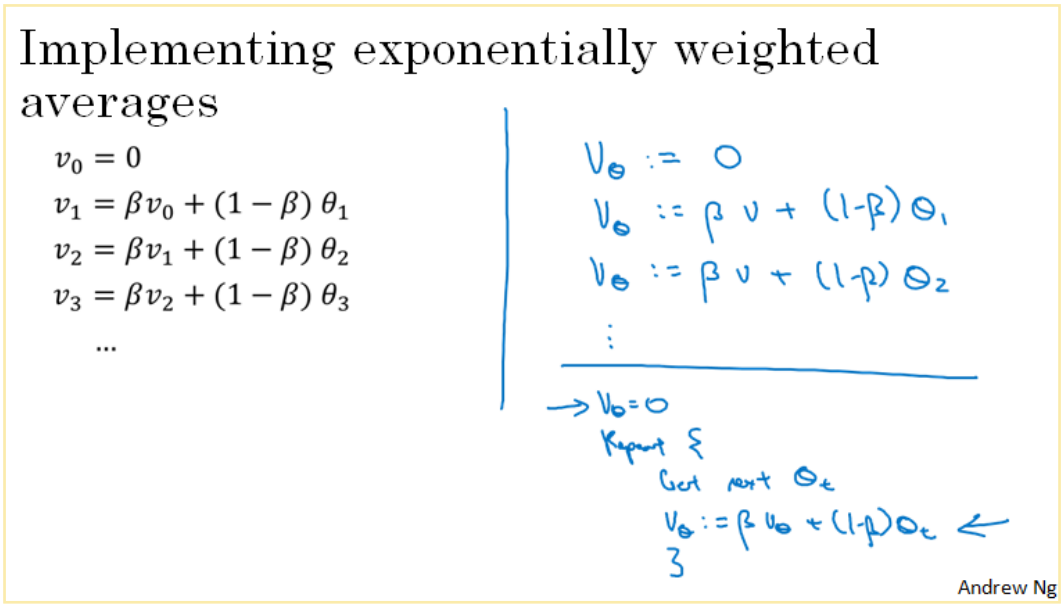
Understanding weighted averages



Expanding the equation, we can express $V100$ as the sum of the θt values multiplied by the exponentially evolving factors. If we plot the two graphics, we can see V100 as the product of both functions: the decay function at the actual values of θt.

The fact that we say that taking a beta value of 0.9 equals to averaging over 10 days relies on the formulas at the bottom of the slide.

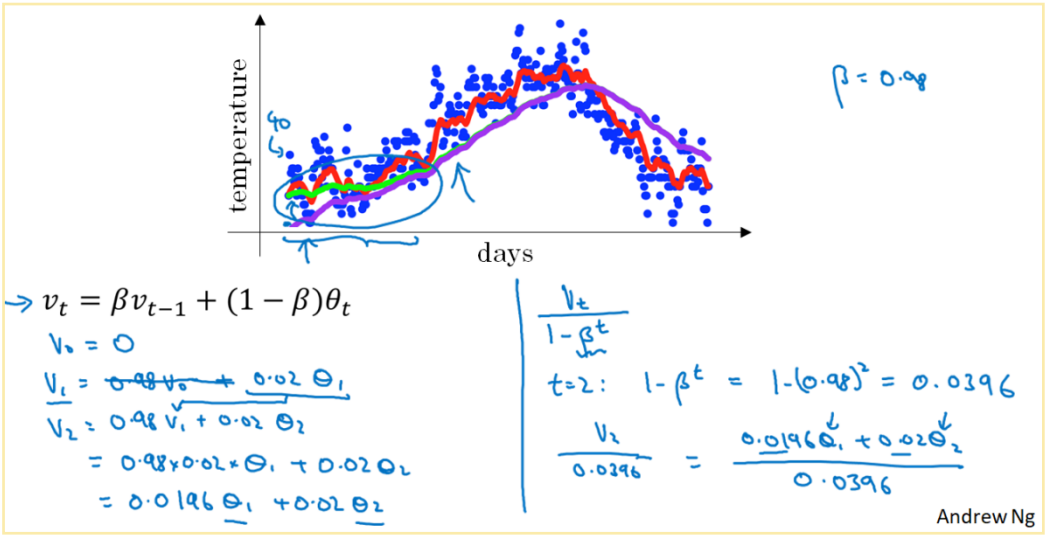
Let's see how we actually implement this:



One of the advantages of this formula is that it takes very little memory. This will help us when implementing optimization algorithms.

4.3. Bias correction

There's one technical detail called bias correction that can make the computation of these averages more accurately.

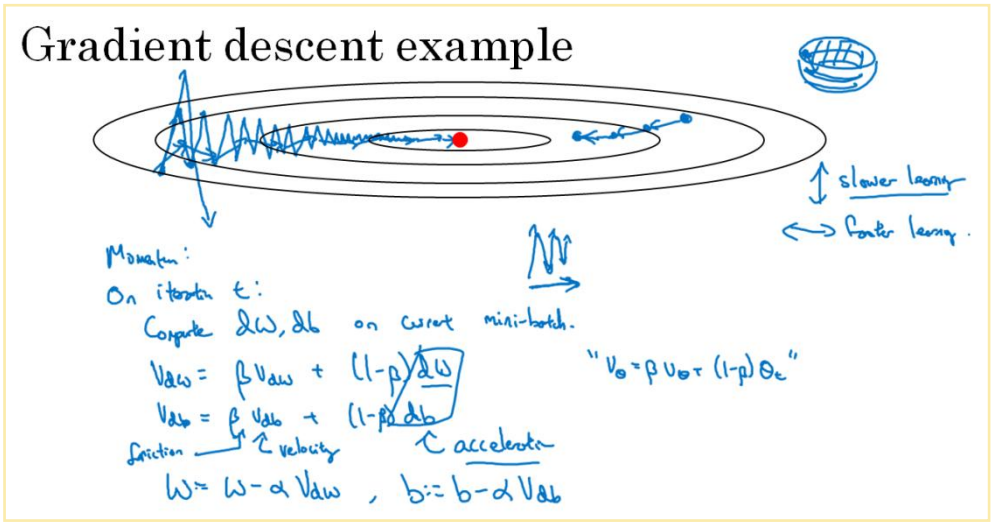


Because the first term V0 is set equal to 0, the first values of the smooth curve are biased. A way to deal with this is to take $\frac{Vt}{1-\beta t}$ instead of Vt.

4.4. Gradient descent with momentum

This algorithm performs almost always faster than the standard gradient descent. The basic idea is to compute an exponentially weighted average of the gradient, and then use that gradient to update the weights instead.

Let's say we want to reach the minimum of a cost functions with contours like this:

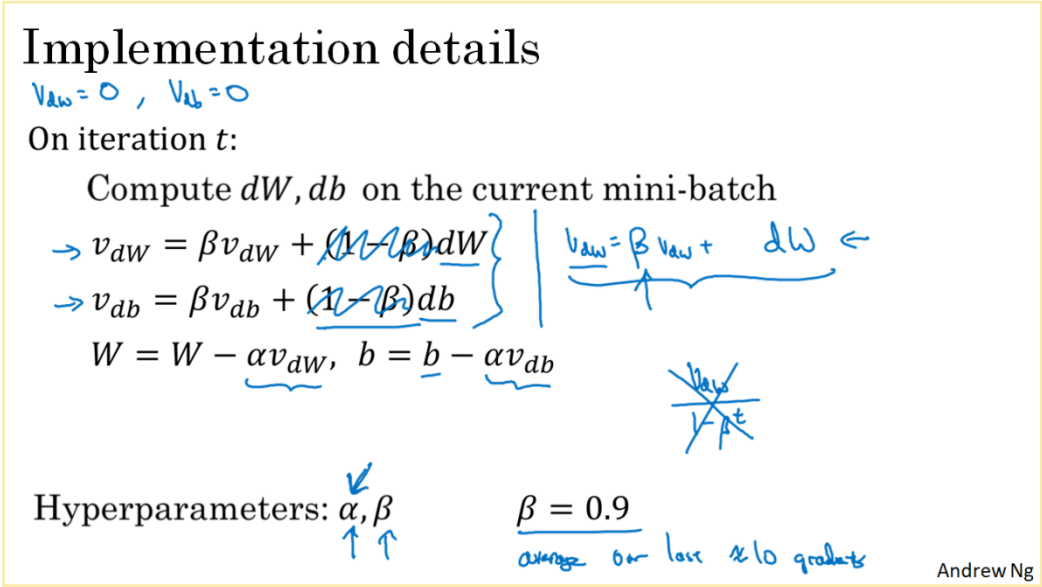


On this kind of cost functions we will oscillate a lot (although using batch gradient descent). We won't also be able to use a high learning rate, so the learning process will take a lot of time.

We could view this problem as the following: we are interested in learning slower on the vertical axis (so as not to oscillate that much) and faster on the horizontal axis (to faster get towards the minimum).

So we can implement the gradient descent process as in the above slide (basically computing V_{dW}). If we do this, if we have a lot of oscillations approximately symmetric, the average of them will be near zero, so we will have no movement on the vertical axis. However, we will still move on the horizontal axis since all the derivatives have the same direction and hence the average does.

This will allow the gradient descent process to be faster. We have to mention that this works on both batch and mini-batch gradient descent. Let's see now implementation details:



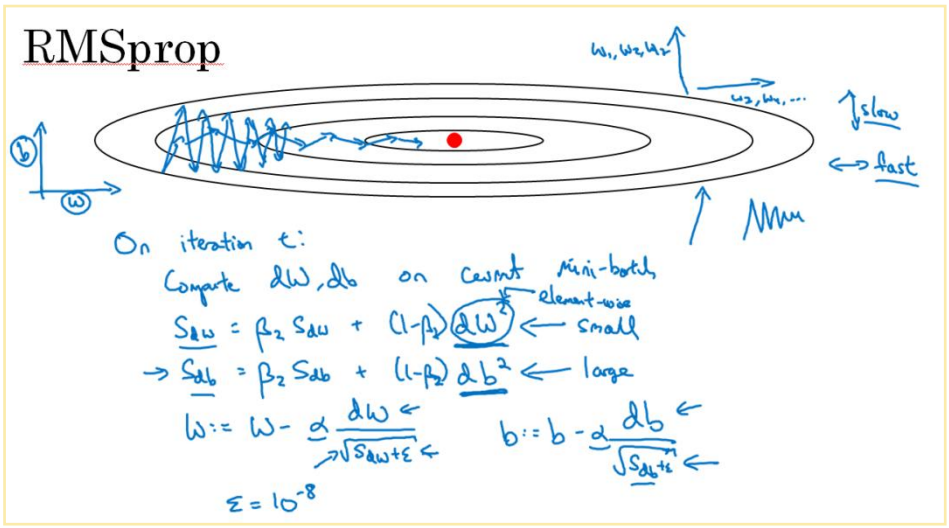
The most common ß value is 0.9, but it turns out to be a new hyperparameter. In addition, people don't usually bother to implement bias correction since after a few iterations the smooth curve will be the same with or without it.

Finally, we can see on the literature that the (1 –ß) term is omitted. Both ways will work fine.

This is the gradient descent with momentum algorithm. It will almost always be faster than the standard algorithm. However, we still have some more options to fasten our learning process.

4.5. RMSprop

This is another algorithm (root mean squared prop) that can also speed up gradient descent. Let's see how it works:



We are again in the same case as before: let's imagine a cost function with the shape of the slide. Let's also suppose that we have the w parameter on the x axis and the b parameter on the y axis. We would like to slow down the learning in the b direction and speed it up in the w direction.

Instead of V_{dW}, we are going to use S_{dW} notation. We will implement the formulas of the slide. What they are actually doing is keeping an exponentially weighted average of the squares of the derivatives. When updating the weights, we are adding the root terms at the denominator of the fractions.

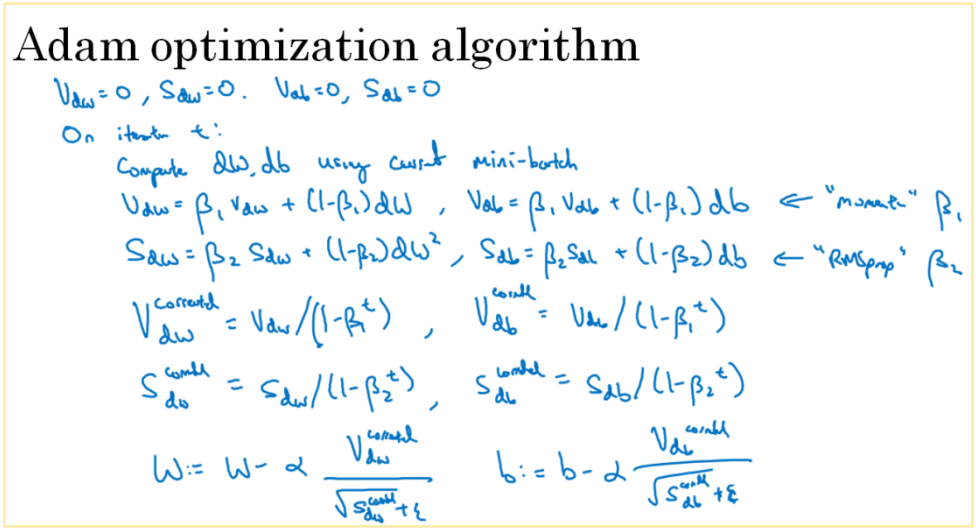
Let's gain some intuition. If we want to slow down learning on the b axis and speeding it up on the w axis, S_{dW} should be very small and S_{db} should be large in order to the weights to be updated fastly and slowly respectively.

Recall that although we have seen this example on a 2 dimensional space, this intuition holds on to larger spaces.

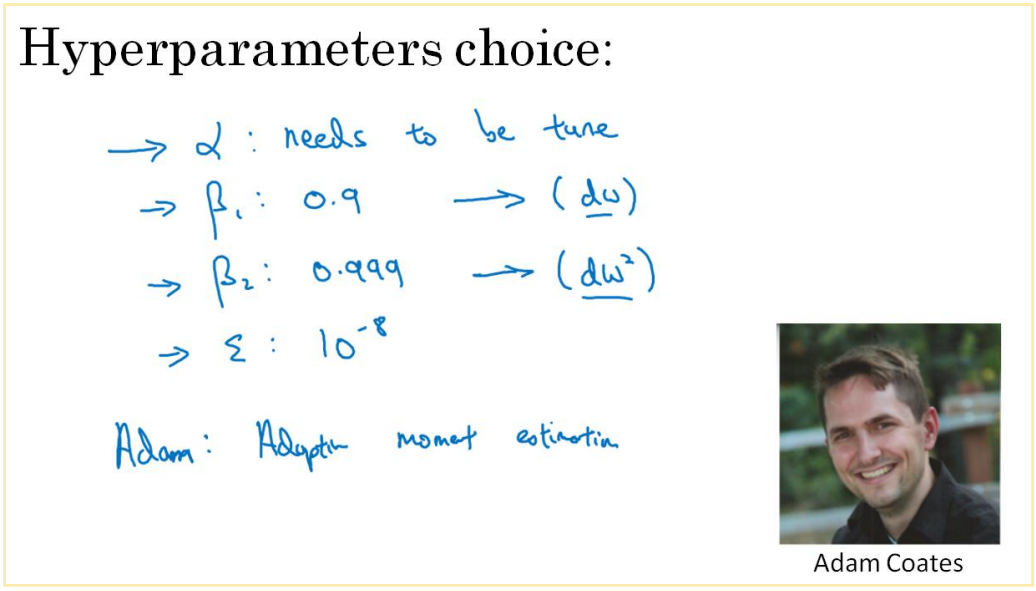
Finally, we have called the betas with the subscript two because in the next point we will combine RMSprop with momentum. In addition, the epsilon term at the denominator corresponds to the need of them to be different of zero.

4.6. Adam optimization algorithm

Adam optimization algorithm has shown to work well across a wide range of deep learning architectures. It basically consists of taking momentum and RMSprop and putting them together.



The implementation is as shown in the above slide. Note that we are introducing bias correction. This algorithm has a few hyperparameters:



- α: learning rate. We need to tune it (we try different values and see which one works better).
- for the rest of hyperparameters, we can see the most used values on the slide.

So the popular procedure is to tune alpha and give the rest of hyperparameters those default values. We could tune all four hyperparameters, but it's not seen very often.

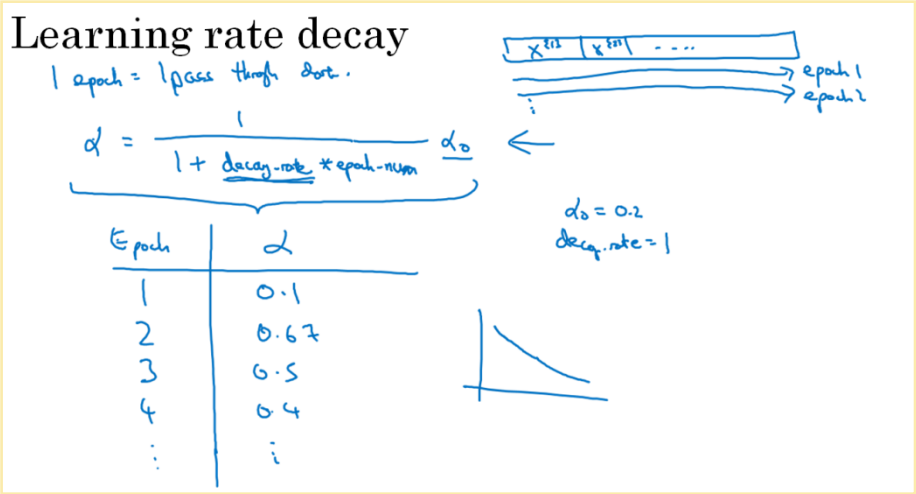
4.7. Learning rate decay

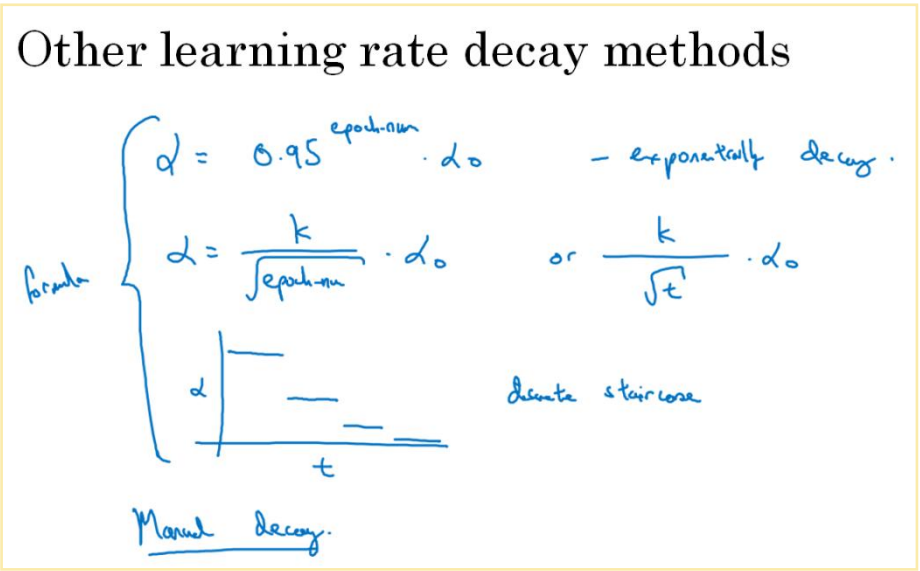
One of the things that might help speed up our learning algorithm is to slowly reduce our learning rate over time. This is called learning rate decay.

The reason why we would be interested in using this learning rate decay is the following: as we go on with more and more iterations, we will be presumably nearer to the minimum than at the beginning. So we would like the learning rate to be smaller at that stage, so as to converge easier instead of be wandering around the minimum.

To summarize, we can use higher learning rates at the first steps of the gradient descent and then use lower ones to easily converge.

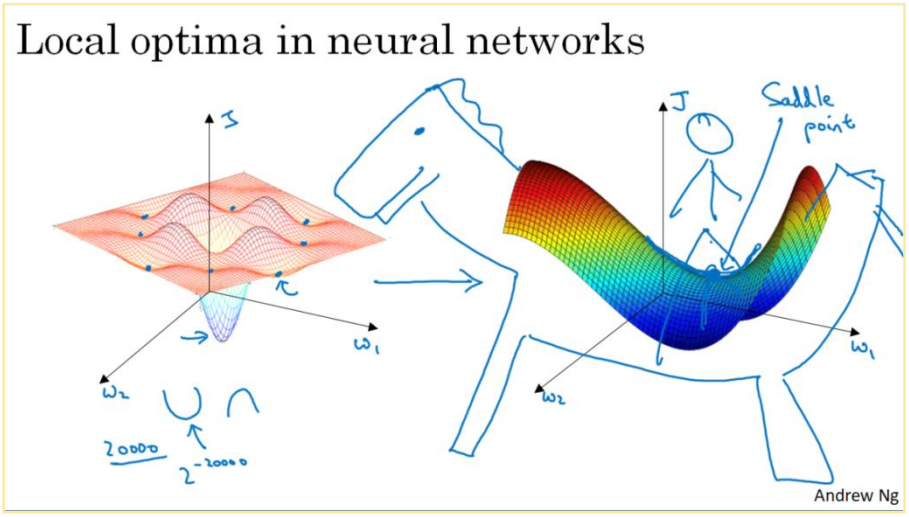
There are different ways to implement learning rate decay:





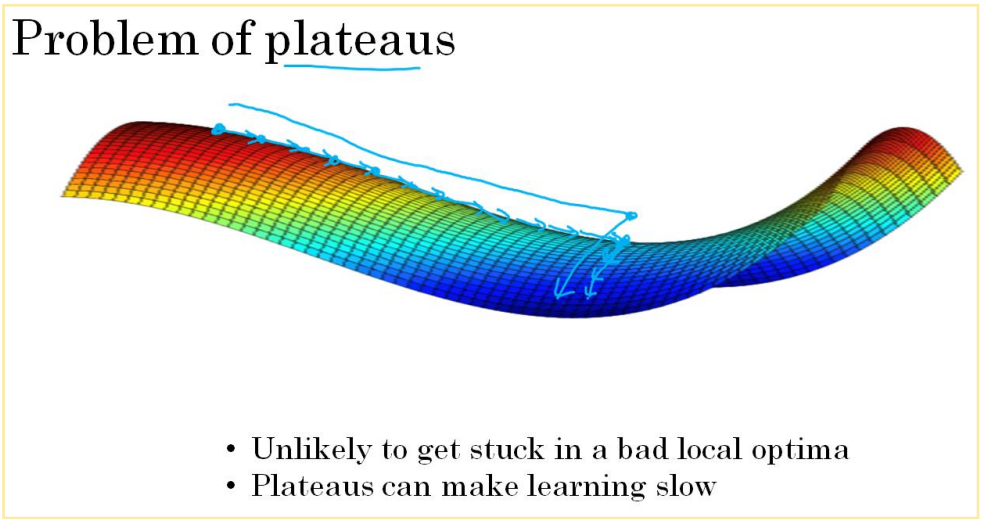
4.8. The problem of local optima

Throughout the training process we may get stuck on local optima. Let's talk about these points. 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.



People used to think of the cost function as the one in the left, with plenty of local minima. But it turns out that it is much more probable for us to come across a cost function like the one on the right. The intuitions of low dimensional spaces don't transfer to high dimensional spaces. In fact, the local minima turn out to be saddle points. They happen to have derivative equal to zero, but the second derivative isn't.

SO, it turns out that saddle points are not a problem. And what is a problem? **Plateaus.**



So in conclusion, on hig dimensional spaces it is unlikely to get stuck in bad local optima, but plateaus can make learning slow. This is when sophisticated algorithms such as Adam or RMSprop can help us.

It has to be noted that at the day of today no one has a complete understanding of the behavior in really high dimensional spaces. Our understanding of them is still evolving.

5. Hyperparameter tuning

5.1. Searching over hyperparameters

Until now we have seen that training neural nets involves setting a lot of different hyperparameters. But how we find good values of these? We'll see some tips for how to systematically organize the hyperparameter tuning process.

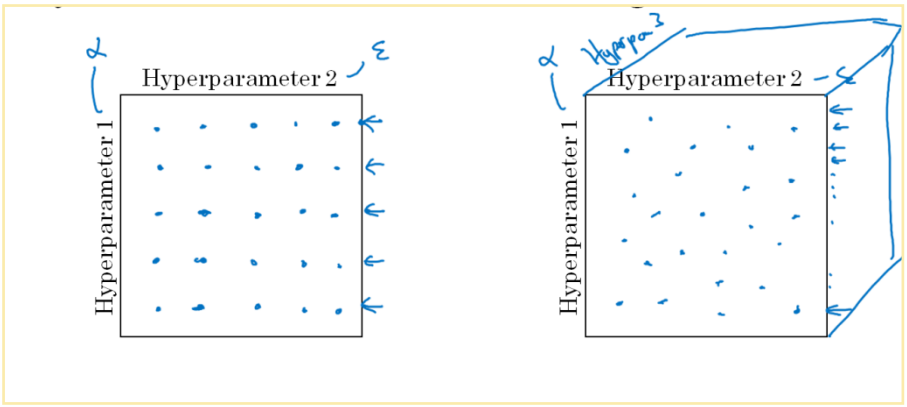
When training a net we have a lot of hyperparameters involved, with the order of importance between brackets:

- Learning rate (I)
- Momentum beta (II)
- Adam beta1 and beta2. Normally we won't even tune these.
- Number of layers (III)
- Number of hidden units (II)
- Learning rate decay (III)
- Mini-batch size (II)

It turns out that some hyperparameters are more important than others. But this is not a hard rule.

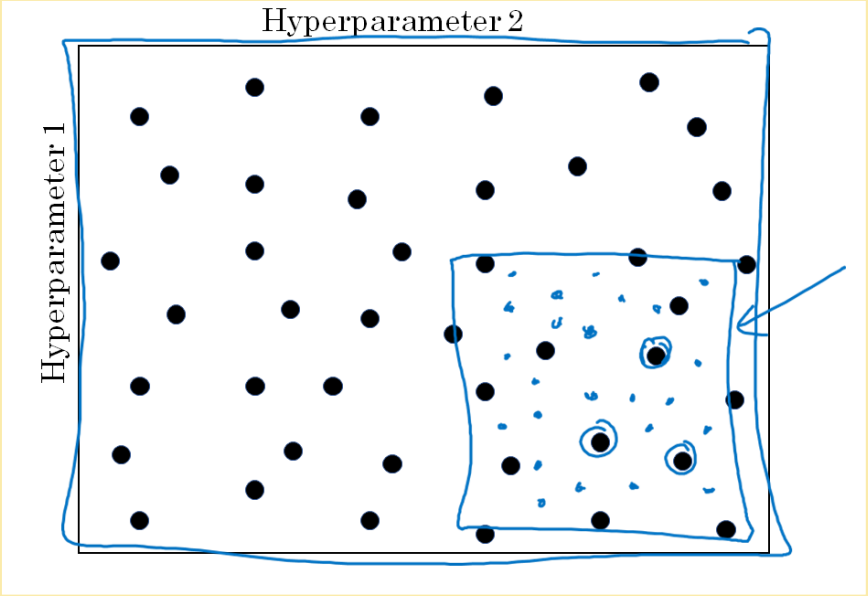
Once we have a set of hyperparameters to tune, how do select a set of values to explore?

Instead of performing a grid search, on deep learning we usually perform a **random search**. And the reason is the following:



We have more important hyperparameters than others. Let's imagine an example where we perform a search over alpha and epsilon. Alpha is much more important than epsilon, and if we do a grid search we will trying 25 combinations but only with 5 different values of alpha. When doing a random search, we will search over 25 different values of alpha.

In addition, we can use a coarse to fine scheme: we can detect a zone where we are getting high performance and then perform a more exhaustive search into that zone.

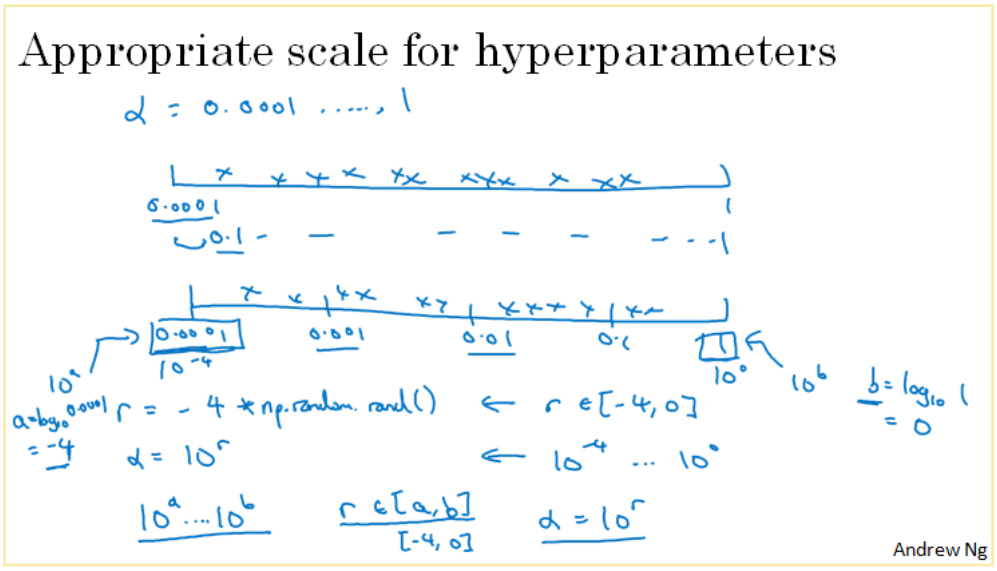


But there is much more involved in hyperparameter tuning. Let's see how to choose an appropriate scale to pick hyperparameters:

On some hyperparameters, it seems reasonable to just perform a random search over the space. For example:

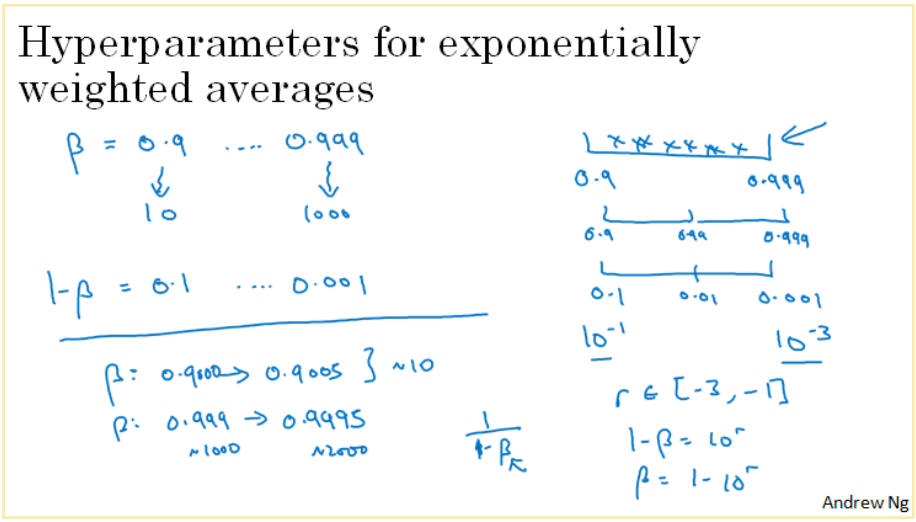
- Number of units on each layer: a 50-100 range seems reasonable.
- Number of layers: for example, 2-4, 2-6, etc…

But this is not true for all hyperparameters. On some of them, such as the learning rate, we should use a logarithmic scale.



In python, we can implement this by generating a random value and then calculating 10^r.

Another tricky case is sampling the hyperparameter beta when talking about exponentially weighted averages. We can perform the same process as in the previous slide and then calculate the complementary:

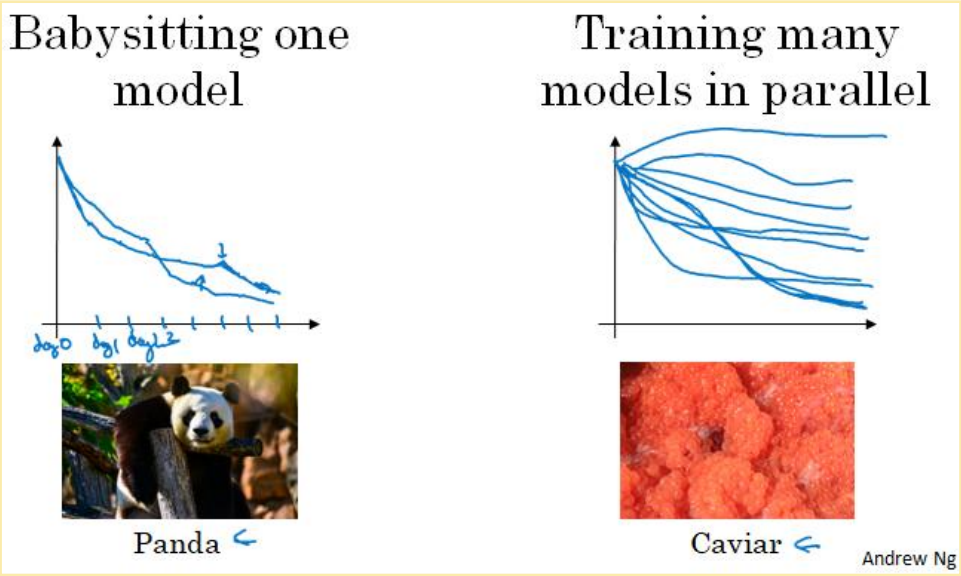


Before wrapping up the hyperparameter tuning process, let's discuss a few tips:

- Hyperparameter intuitions don't hold across different domains.
- Even in the same problem, intuitions do get stale. That's way it's a good practice to re-evaluate our models occasionally.

In addition, in terms of how people go about searching for hyperparameters, there are two schools of thought:

- If computational resources are limited, we can stick to a model and improve it throughout the days (babysit one model).
- If they are not limited, we could run several models in parallel.



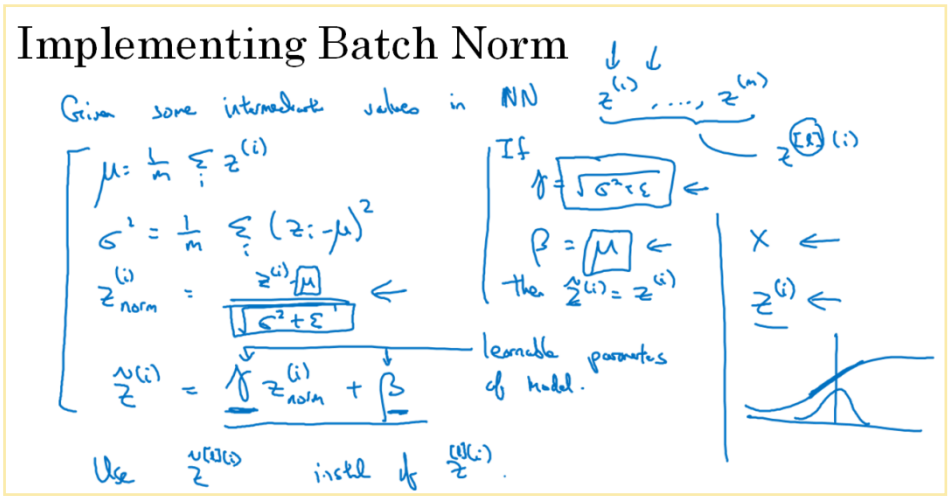
In addition to this, it turns out that that there's one other technique that can make our neural network much more robust to the choice of hyperparameters. It doesn't work for all neural networks, but when it does, it can make the hyperparameter search much easier and also make training go much faster.

5.2. Batch normalization

In the rise of deep learning, one of the most important ideas has been an algorithm called batch normalization. Batch normalization makes the hyperparameter search problem much easier and makes our neural network much more robust. The choice of hyperparameters is a much bigger range of hyperparameters that work well, and will also enable us to much more easily train even very deep networks.

Let's remember that normalizing the inputs helped to speed up the training process. But what happens with a deeper model? We not only have input features, but also activations.

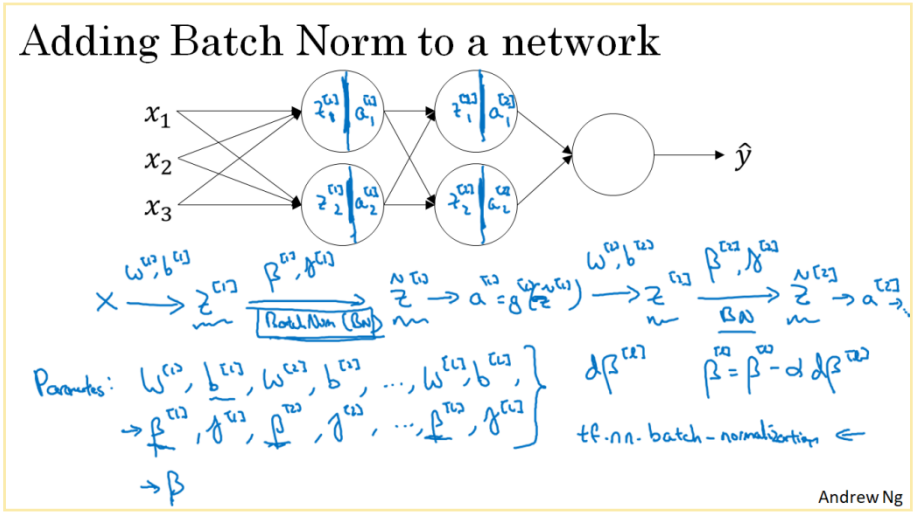
It turns out that we can normalize the activations (that are the input values for the next layer), and this is what batch normalization does. With a little nuance: we will normalize Z instead of A. It is much more often done, although some people normalize after the activation.



We will normalize every Z from 1 to m on every layer. But we might not want all the Z to have mean 0 and variance 1, so we will calculate them as gamma times Z plus beta, which gamma and beta being learnable parameters of our model (just like the weights, they will be learnt).

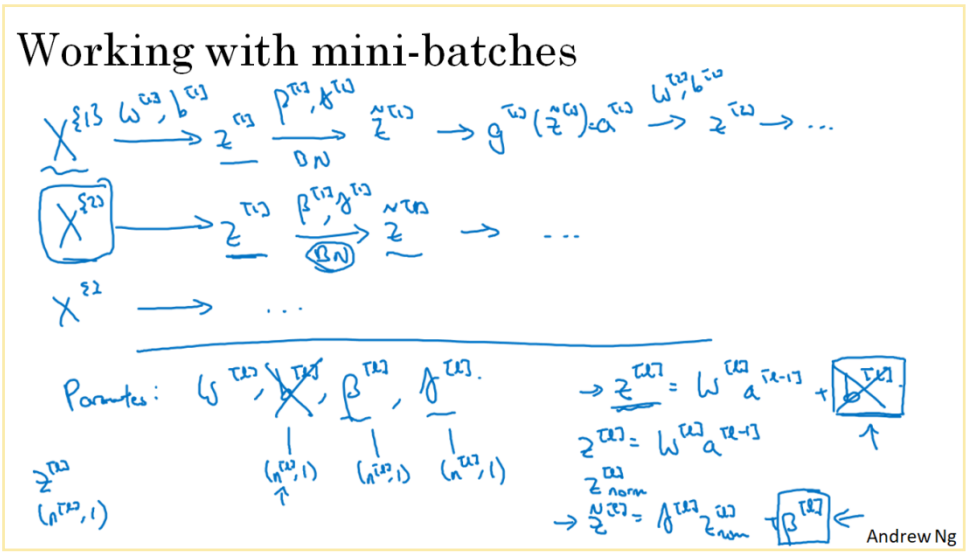
Remark that gamma and betta allow us to set the variance and the mean of Z. If gamma and beta where the squared values at the right of the slide, then Z would retain its original value.

Let's see how we can implement this:



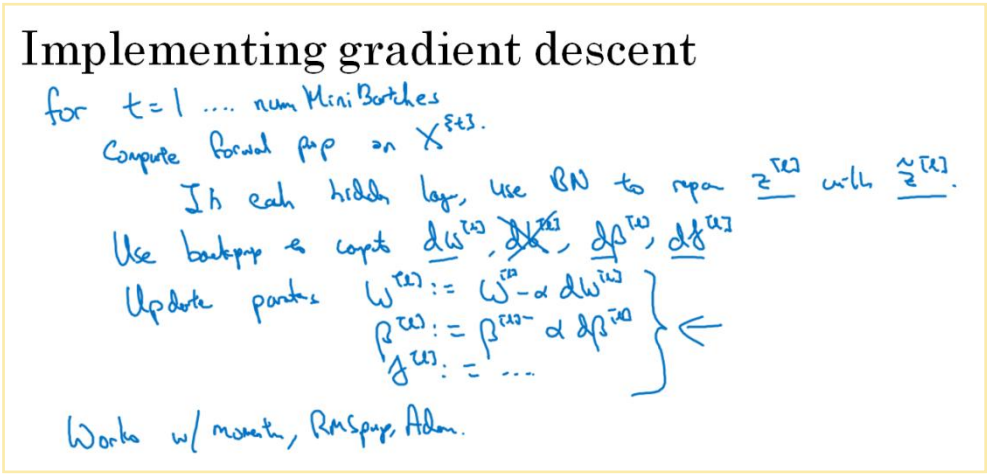
It is simply normalizing Z on each layer using beta and gamma on each one. Beta and gamma will be updated just as the weights are on each step of gradient descent.

However, batch norm is usually applied to mini-batches.



On each step, we are calculating mean and variance of the mini-batch. There is something that has to be noted: Z is computed on each layer l with the formula we can see at the bottom right of the slide. Because what batch norm does is normalize Z to have mean 0 and 1 variance, and then rescale it with beta and gamma, the b value will get subtracted no matter what value it takes. So if we are performing batch normalization, we can eliminate the b parameter.

Let's put it all together:



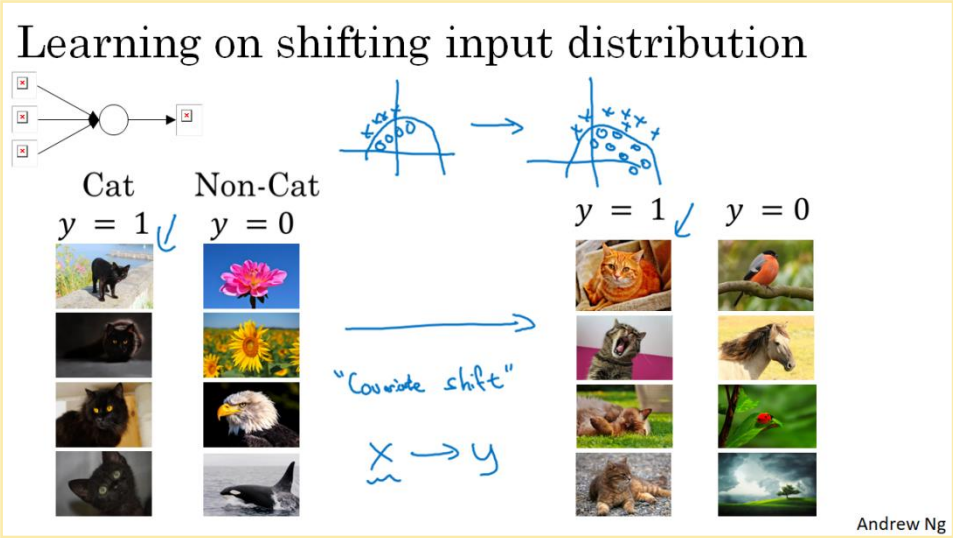
This also works with other optimization algorithms such as RMSprop, Adam, etc…

So, why does batch normalization work?

We have seen that normalizing the inputs and setting them on the same scale can speed up learning. The intuition is that doing so on all the hidden layers can also help speeding up the learning process. But there are a couple further intuitions:

Let's suppose that we have trained a cat vs non-cat model on images of only black cats as positive examples. If we then feed the trained model with images of cats that aren't black, we would expect that the model doesn't predict very well. In other

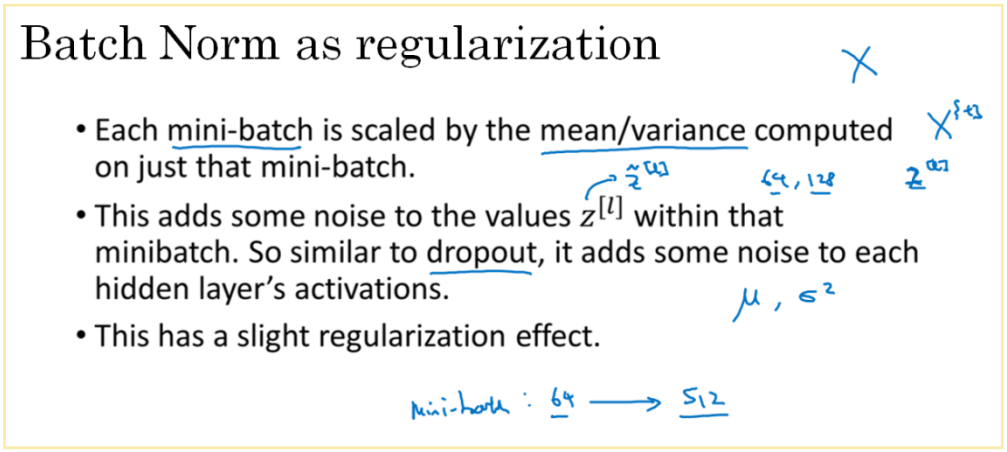
words, if the distribution of the X changes, then we might need to re-train our model.



This is called **covariate shift**.

So, what batch normalization does is that it reduces the amount that the activations fed to each layer change because of distribution changes. So, if input features change, batch norm will make them to change "less".

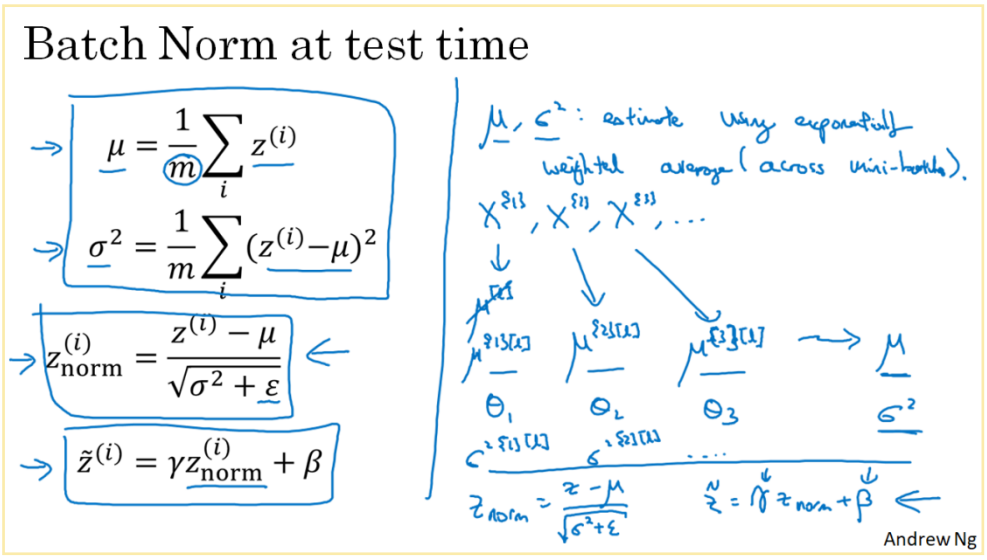
In addition, it turns out that batch norm has a second effect:it has a slight regularization effect.



But we shouldn't see batch norm as a way to regularize. It is an algorithm that can help speed up the training process, and regularization is a side effect.

One important thing before wrapping up batch norm is: how do we do with new data? Because we are using mean and variance from each mini-batch, how can we do when we are making predictions on new data?

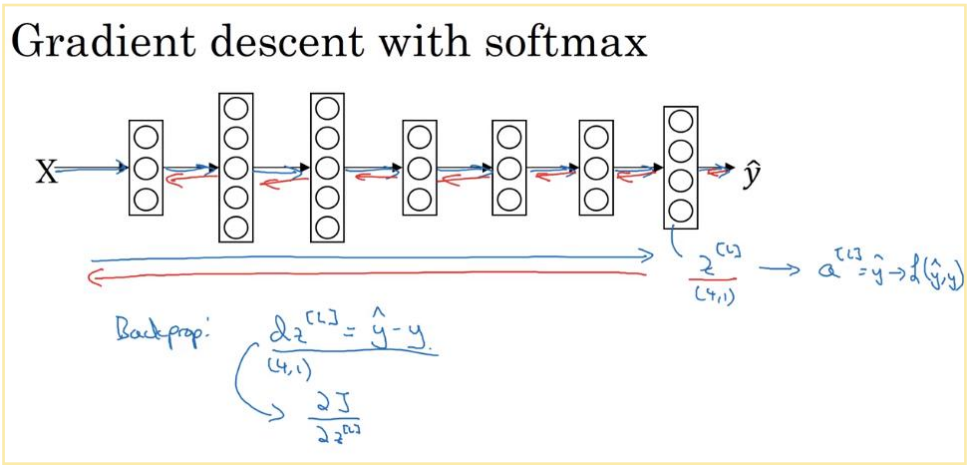
For new data, we are estimating the mean and the variance as an exponentially weighted average across the mini-batches.



6. Multi-class classification

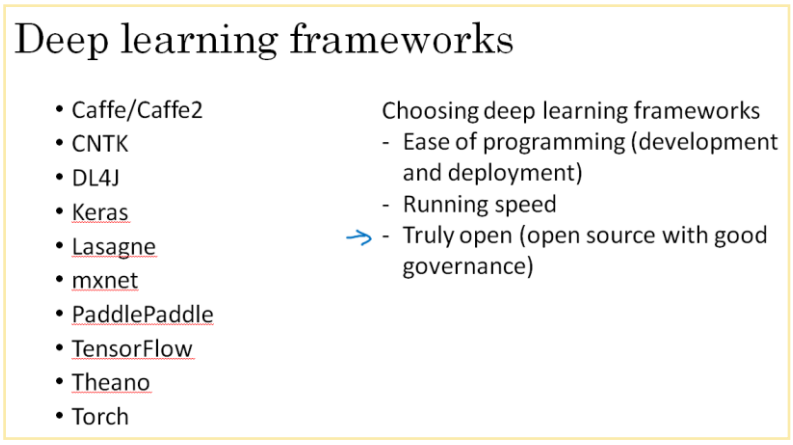
Until now we have seen binary classification examples. But what happens if we have a multi-class classification problem?

The thing we have to remember is that the output layer needs to have as many units as classes we have and **softmax** activation functions. Also, we need to beign the implementation of backprop with:



7. Deep learning programming frameworks

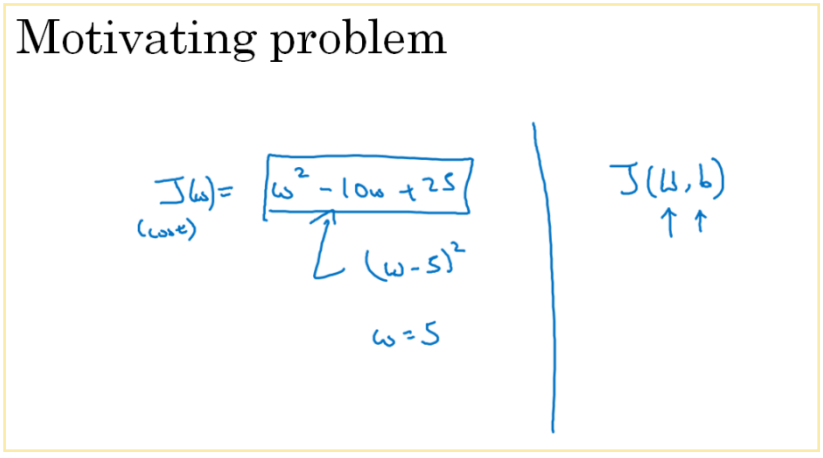
Until now we have implemented our models from scratch with python and numpy. But there are many good deep learning software frameworks that will help us implement these models.



7.1. Tensorflow

We will start to use TensorFlow. We will see the basic structure of a TensorFlow program and then do practice exercises in the assignments.

Let's get a motivating problem and then see how we can implement it in TF:



The code is available on 10. TensorFlow example.