**Improving neural network, deciding upon hyper parameters, learn in a reasonable amount of time.**

Making good choices as to how we set our training, testing set can help us find a good performance on neural net.

4 things to note:

* Number of layers
* Number of hidden units
* Learning rates
* Activation function

It is almost impossible to find the correct choice on the first attempt.

Applied ML process includes – Idea, code and experimentation iteratively till we get our best values.

Setting up of train, development (hold out cross validation) and test set, we can make much more efficient use of that.

For training algorithms train set is used. Dev set is to find out which of our many different models performed best. In order to get a unbiased estimate as to how your model performs we use test set.

Traditional ML – 70/30 – 70% train and 30% test

Or 60/20/20 – 60% train and 20% dev and 20% test

But modern big data era, dev and test set are smaller. We need the dev set to be that big only on which we can evaluate two or more learning algorithms to find out which one performed better.

So if you have a million examples in your dataset just having 10000 samples in dev set is enough to evaluate which model performs better. Similarly 10000 examples is more than enough for your test set. So our ratio becomes 98% train and 1% dev and 1% test.

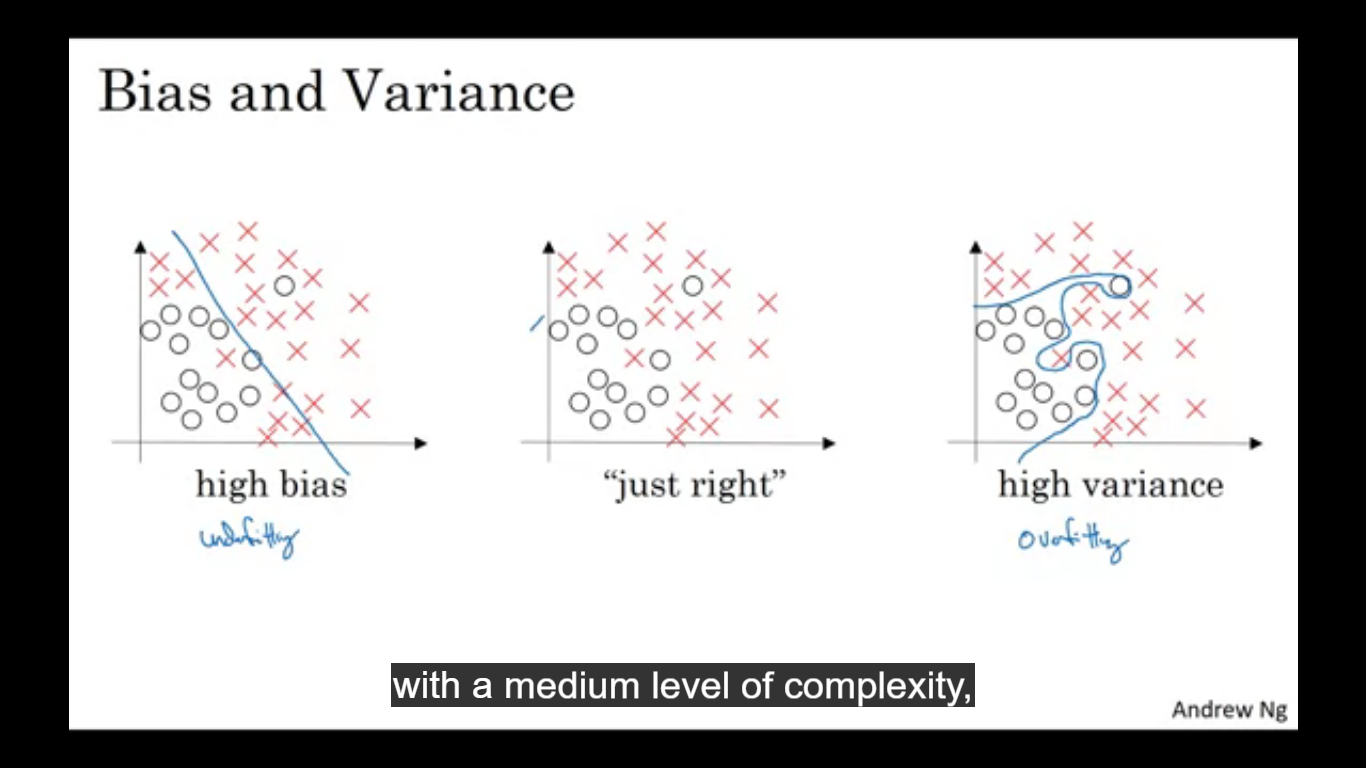
More and more people train on mismatched train and test distributions. For example a cat classifier where training dataset is from web-pages- high quality framed cat images, whereas dev/test data is cat images of user from cell phone cameras, so train and test/dev distributions are very different.

We should make sure that the dev and the test set must come from the same distribution.

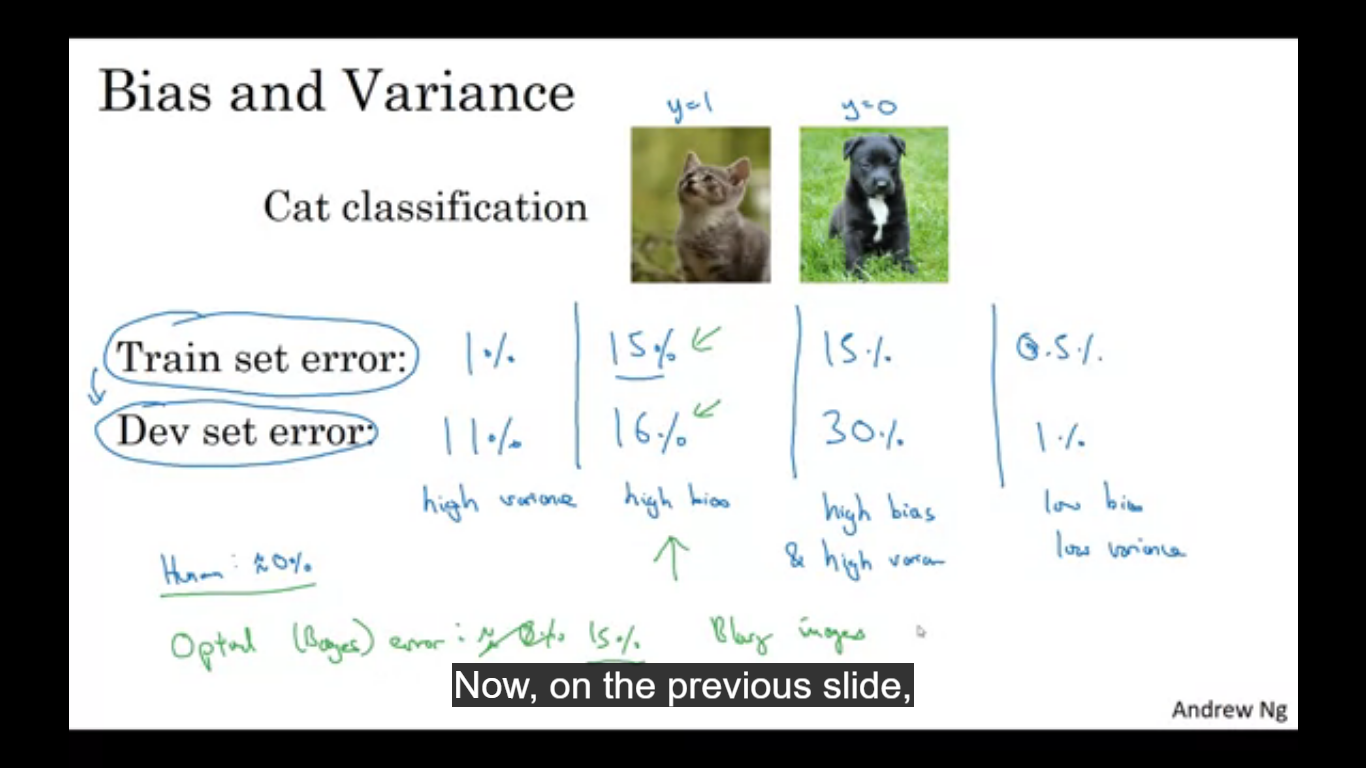
**Bias-Variance Trade-off**

In 2D problems we can plot the data and visualise the division boundary, but in the cases of high dimensional data, this might not be possible.

Two key terms to understand bias and variance –



Train set error and test/dev set error.



Let’s say train set error is 1% and dev set error is 11% - in this example you are doing very well on the train set and doing relatively poor on dev. Looks like overfitting and in cases of overfitting there is high variance.

Let’s say that training set error is 15% and dev set error is 16% then this data is high bias since it doesn’t perform well even on training set.

Let’s say 15% training error and 30% test error, this model has high bias as it is not doing very well on training set and high variance (worst)

If training set error is 0.5% and dev set error is 1% then the model has low bias and low variance.

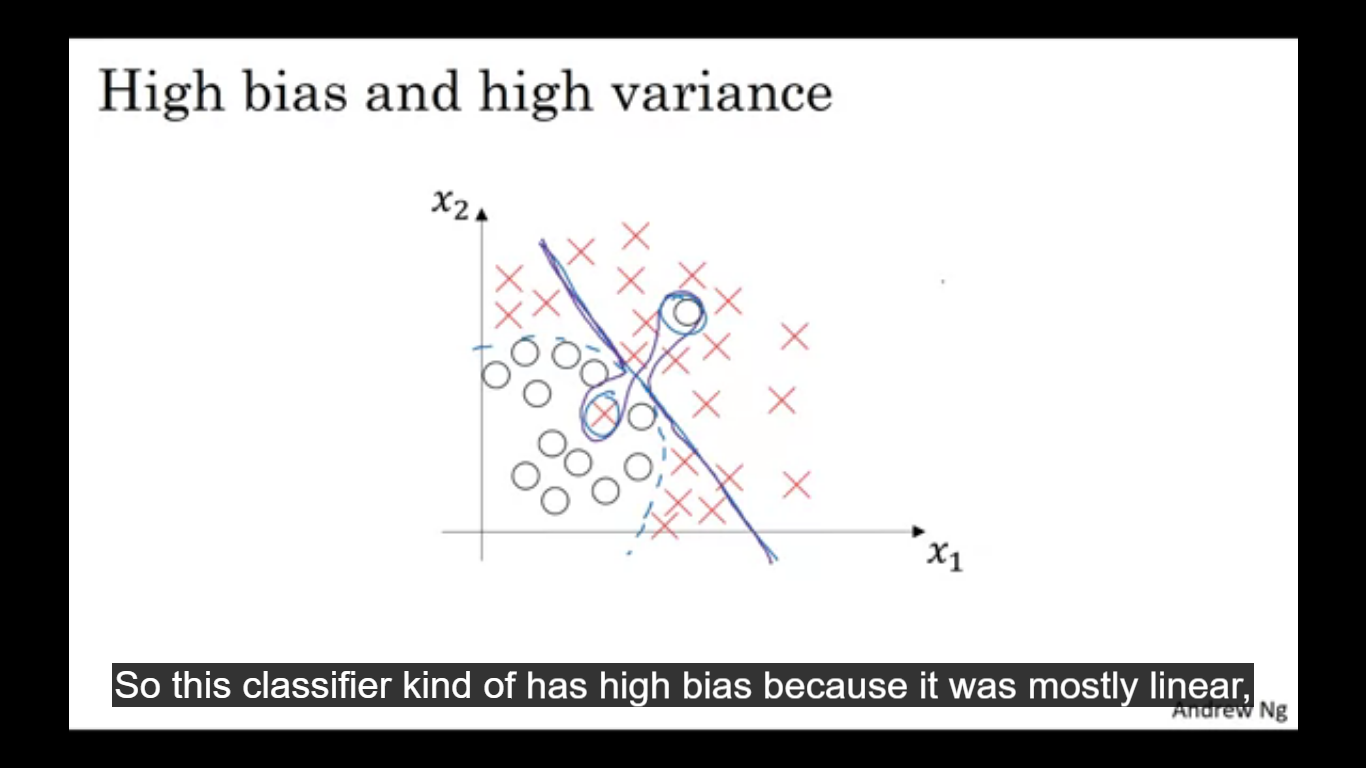
The above analysis is made on the fact that Bayesian error is 0%.

So in case of how to analyse bias and variance when no classifier can do very better, as in cases of blurry images so there no system could do very well so there the bayes error is much higher.

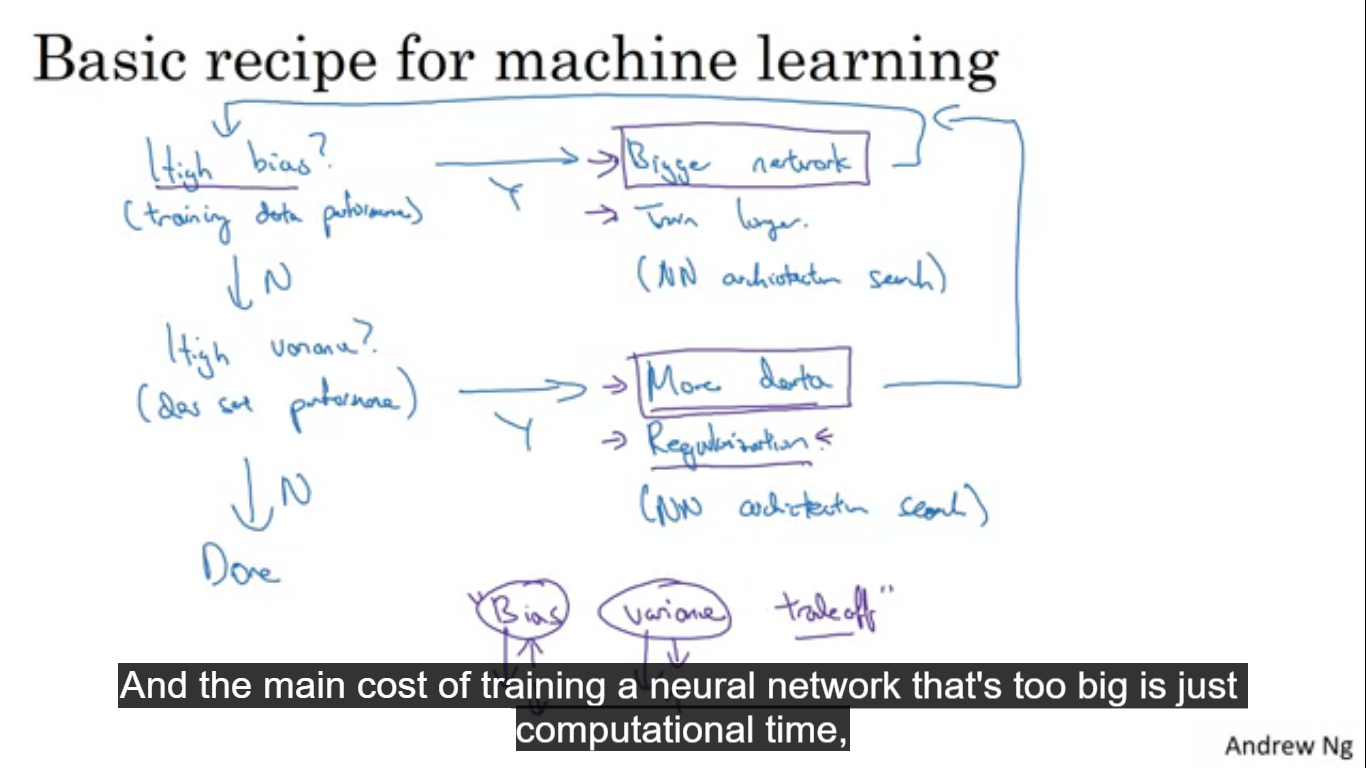
Underfits has high bias, overfits has high variance, high bias and high variance is the case when it cant generalise but tries to generalise a specific set of points by overfitting, so kind of both over fitting and under fitting in the same problem.

**Four categories:**

* **High bias**
* **High variance**
* **Both high bias and high variance(worst)**
* **Low bias and low variance (best)**

****

**Basic recipe for Machine Learning**

****

After initial training we will first check if our model has high bias? It is not fitting the training et well, try bigger network like more hidden layers or more hidden units or increasing the training time, or finding a neural network that is better suited for this problem.

Once the high bias problem is solved, ask do we have a high variance problem?

To find that we look at dev set performance. If we have high variance, best way to solve it is to get more data, or we could try regularization, or find a NN which is better suited for this task.

So once we reach low bias and low variance we would be done.

**Bias Variance Trade-off**

**For all the things we could try, we could either increase the bias and reduce the variance or increase the variance and reduce the bias.**

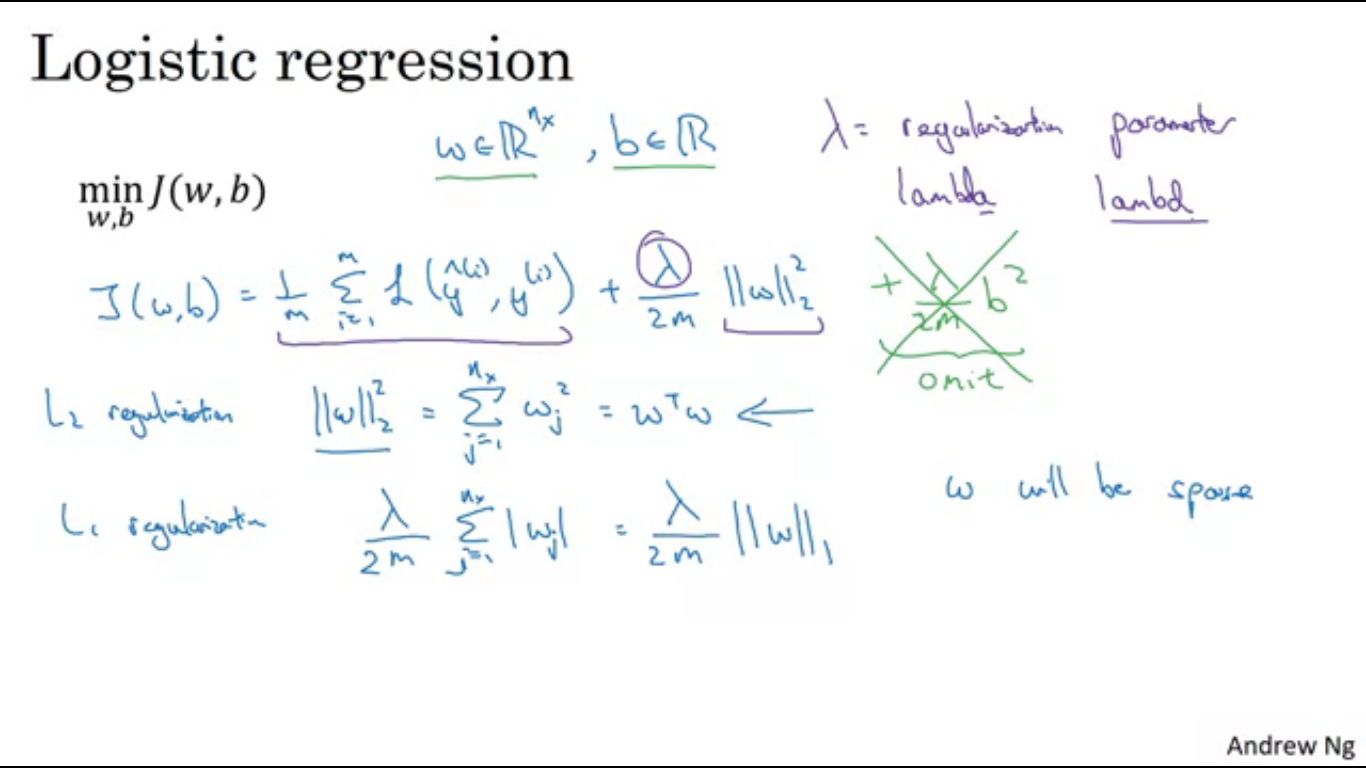
But in this deep learning era, getting a bigger network always reduces bias without increasing the variance, and getting more data reduces the variance without increasing the bias.

So trade-off has been much reduced.

**Regularization**

If we suspect our model has a high variance problem, our model is overfitting, the first thing to try is regularisation.

For Logistic Regression



L2 regularisation as here we are using L2 norm with the prime to vector W

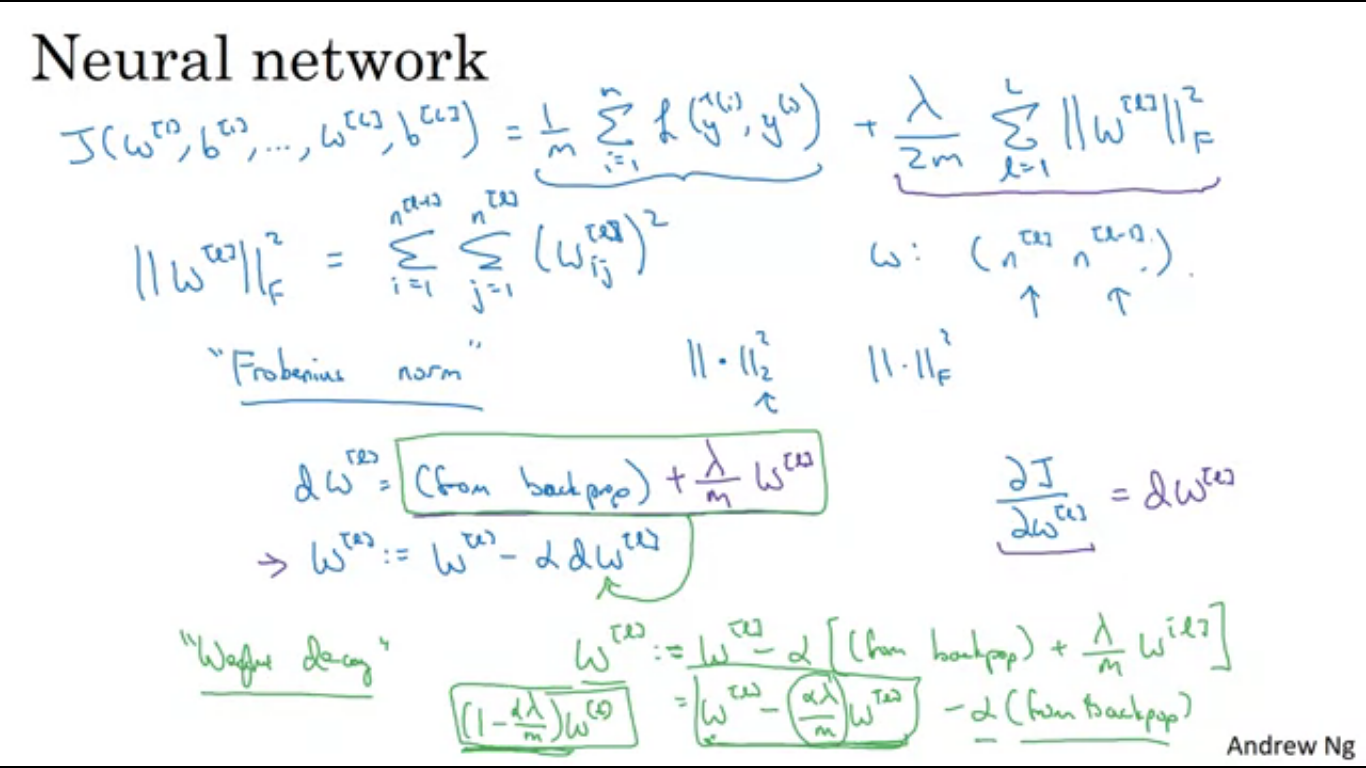
Why do we regularise only W and not b? The answer is that we can regularise b as well but it is not really required, since W is usually very high dimensional, whereas b is just a single number or one parameter out of very large number of parameters.

L1 regularisation is the L1 norm of vector W

If we use L1 regularisation then W will end up being sparse meaning W vector will have a lot of 0s in it. It does not work for the purpose of compressing your model, it’s more of a myth.

Lambda used in the image is called the regularization parameter, another hyper parameter we might have to tune.

**For Neural Network**



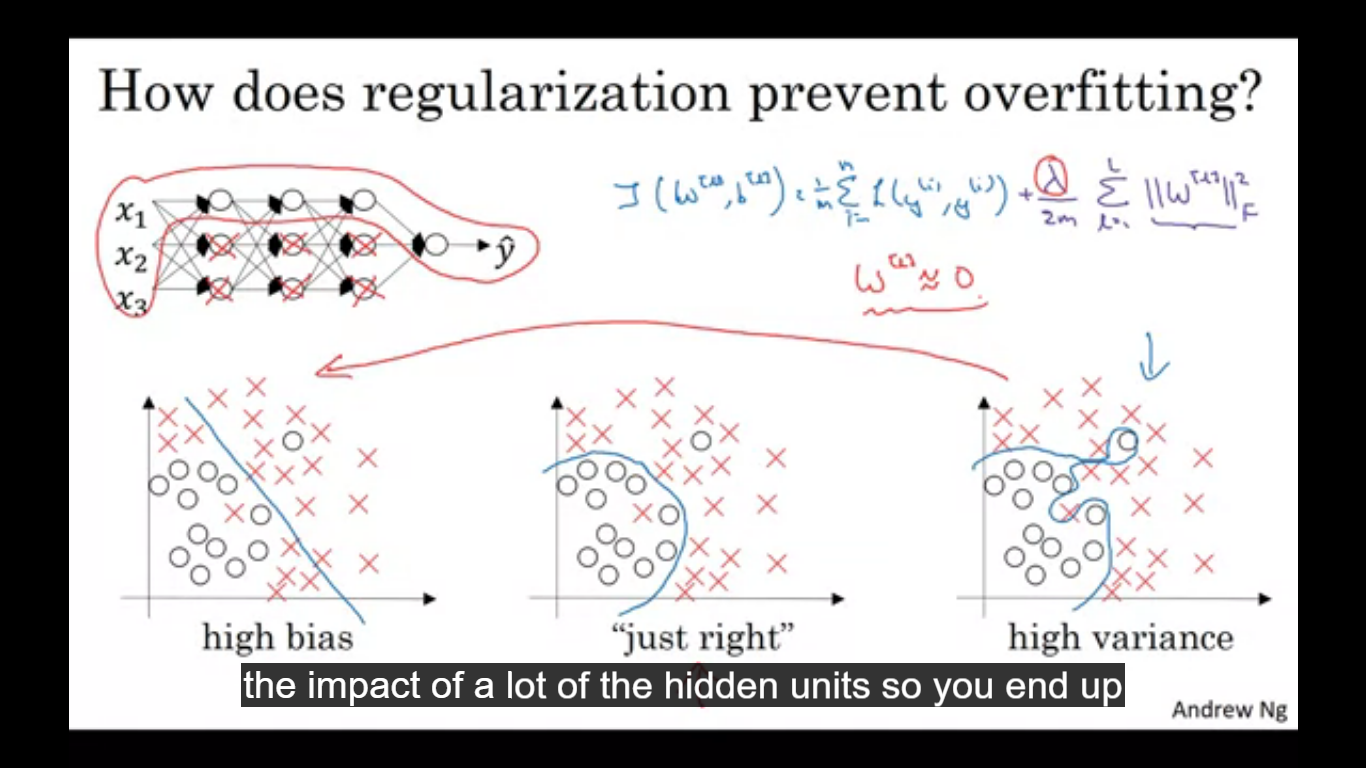
W dimension is (number hidden units in this layer, number hidden units in the previous layer)

The norm of W is called as the Frobenius norm of a matrix.

∣∣*w*[*l*]∣∣2=∑*i*=1*nl*​∑*j*=1*n*[*l*−1]​(*wi*,*j*[*l*]​)2

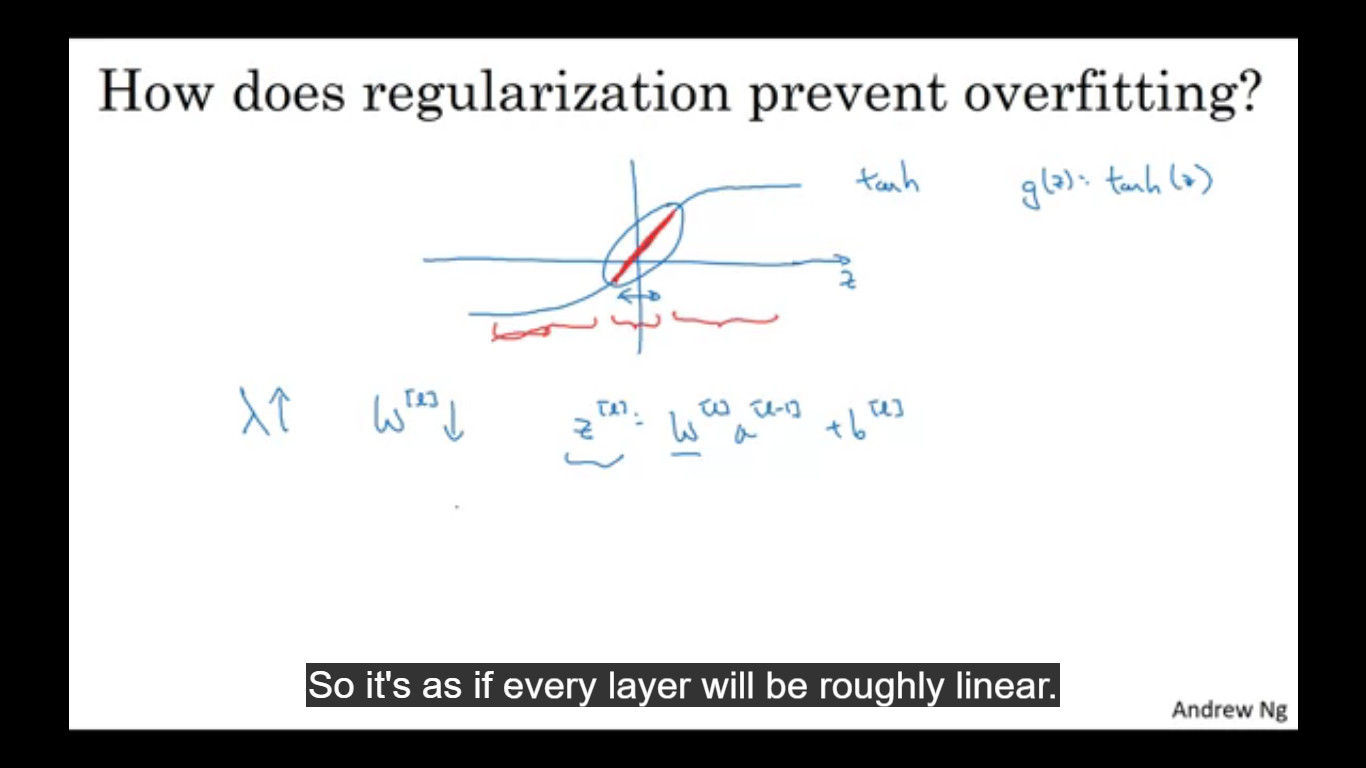
L2 regularisation is also called as the “Weight decay”. So as shown in the image without using regularisation and with using the regularisation there is a difference. Now the W is further subtracted by an additional term.

Why does Regularisation reduce overfitting?



The extra term we added penalises the weight matrix for being too large. So why is that shrinking the frobeinous norm might reduce over fitting?

One piece of intuition is if lambda is really big then it intensifies the weight matrix W to be close to 0. So that would be zeroing out the impact of lot of hidden units simplifying our neural network. This would take us from high variance to high bias, so we need to find an intermediate value of lambda, to get the right case. The intuition of zeroing out is not quite right, as all the hidden units would still be used but their effect would be reduced.



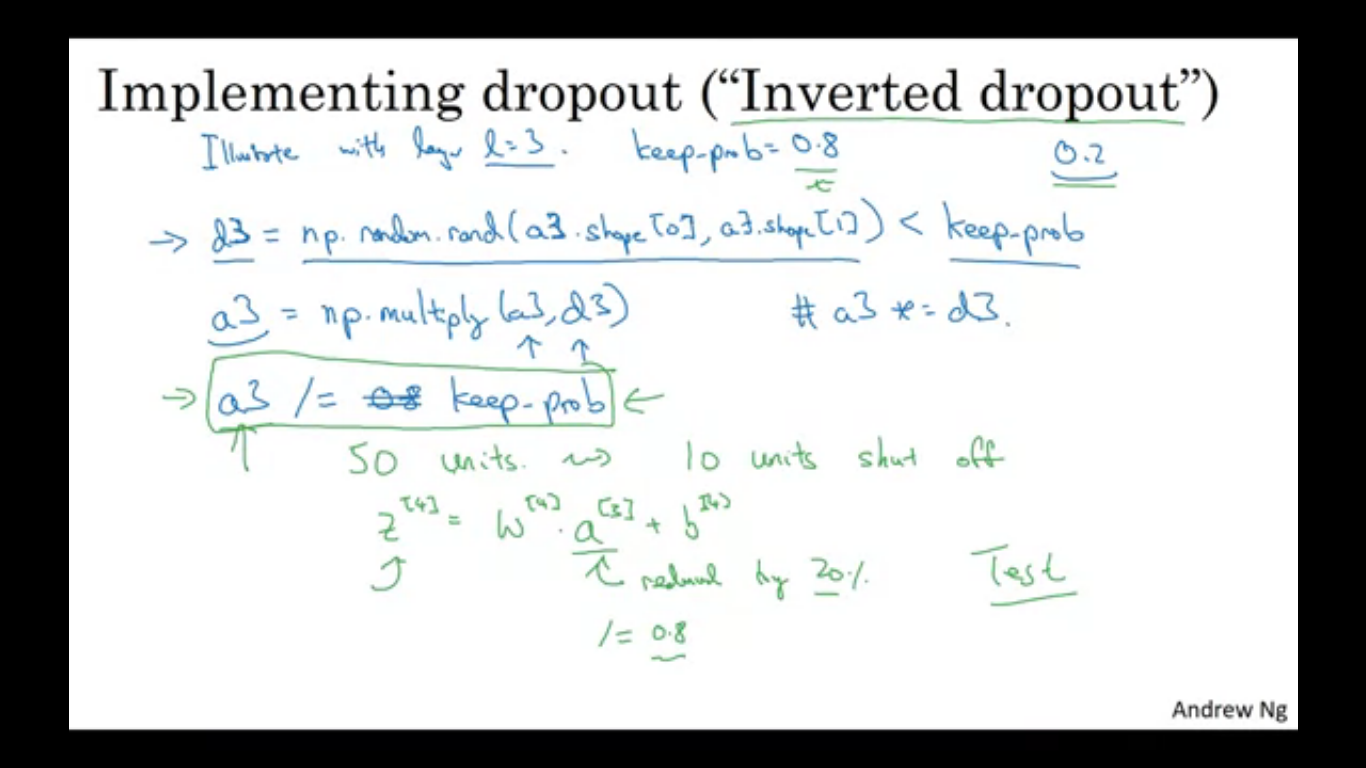
Another intuition is that if we increase lambda, W reduces and Z is W[l]a[l-1]+b[l], so as W reduces Z reduces and g(z) comes to the linear part of the activation function, making our each layer of the NN roughly linear.

Implementation tip – when you implement gradient descent, one of the steps to debug gradient descent is to plot cost function J as a function of number of iterations and see that cost function J decreases monotonically after every iteration of Gradient descent.

**Dropout**

Go through each of the layers of the network, and set some probability of eliminating a node in a neural network. Let’s say dropout is 0.5, that is 0.5 chance of keeping a node and 0.5 chance of removing each node. The nodes which are dropped we remove all the incoming and outgoing connections from it ending up with a much smaller NN.

Implementing dropout



Keep\_prob is the probability the given hidden unit will be kept. Keep\_prob = 0.8 means 0.8 chance the unit will be kept and 0.2 chance the unit mat be eliminated.

D3 is a matrix where for each training example it stores value of 80% chance of 1 and 20% chance of 0. A3 are the activations you computed. How the complete dropout is happening is explained in the image below.

Example let a3 has 50 units out of which 10 units will be zeroed out, Z[4] will be reduced by 20%.

Inverted dropout ensures that the expected value remains the same

For different training examples you zero out different hidden units. Vector d3 is used to decide what to zero out both in forward as well as back propagation.

For making predictions at test time we not use dropout because at test time we do not want our output to be random. Implementing dropout at test time just adds noise to our predictions.

Inverted dropout se division by keep\_prob so even if you don’t use dropout at test time we don’t need to add a scaling constant so as to keep the training and testing on the same phase.

Dropout randomly knocks out units in our network so it’s as if on every iteration you are working with a smaller neural network.

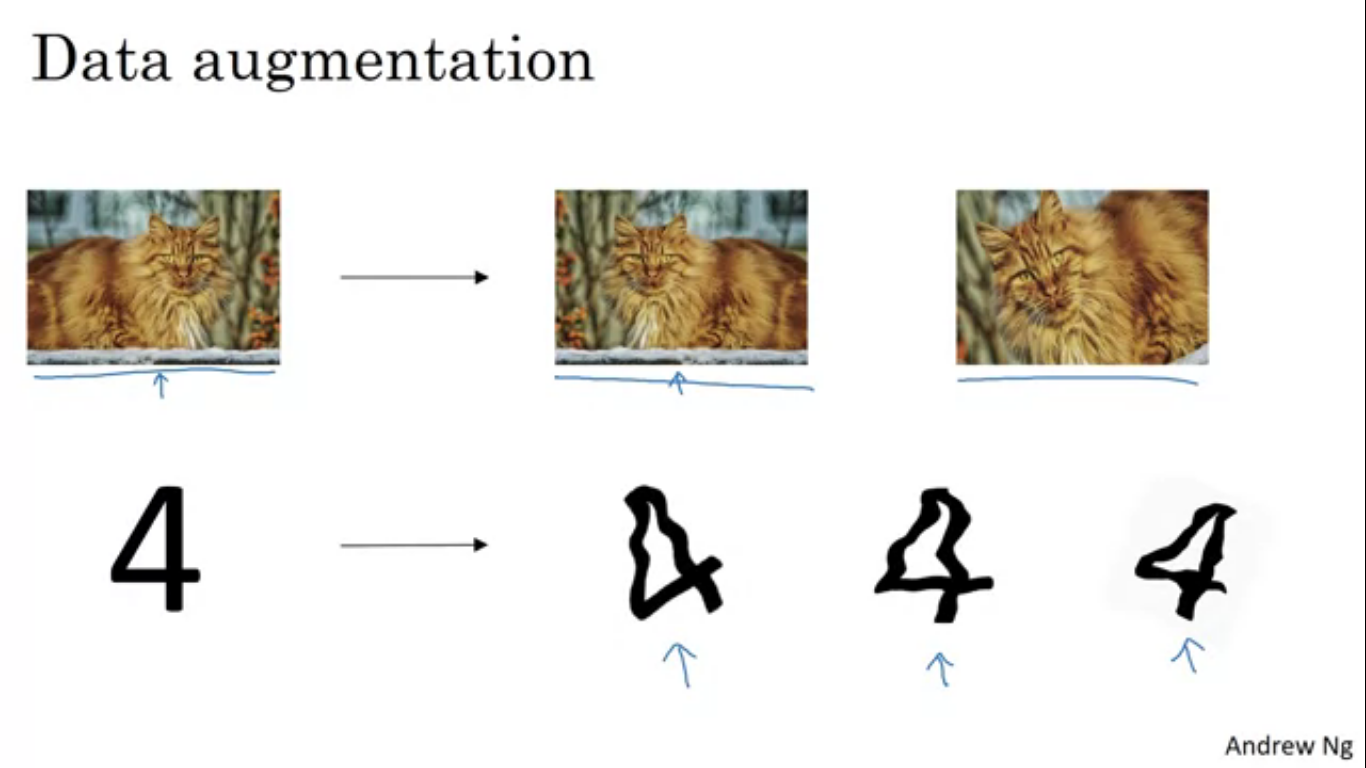
We are reluctant to put too much weight on any one input since it can go away, so spreading of weights will happen. This will tend to have an effect of shrinking the squared norms of the weights. Dropout has a similar effect to the L2 regularization. It is feasible to vary keep\_prob by layer. To prevent overfitting due to large number of parameters we keep\_prob to relatively low and for different layers where there is a lesser chance of overfitting you may keep a higher keep\_prob.

Keep\_prob of 1.0 means you are keeping every unit, and not really using dropout for that layer.

If we are more worried out some layers overfitting over the others, we can keep a lower keep\_prob for that layer. Alternative is one layer you apply dropout and other layer yo don’t apply dropout.

One big downside of dropout is that the cost function J is no longer well defined. So it is difficult to cross check that we have a well defined cost function J that is going downhill on every iteration as cost function we have defined is less well defined.

In addition to L2 and dropout, there are few more techniques to reduce overfitting in your neural network, so the technique include data augmentation.



Sometimes we require more data but since we can’t get more data we can flip the image horizontally, so by flipping the images horizontally we can double the size of the training set. Other than flipping horizontally you can also take random crops of the image to make more fake training examples. So data augmentation is also a good form of regularization.

Early Stopping – as you run gradient descent you plot either the training error or cost function J optimising. With early stopping we also plot the dev set error, what we will see os that the dev set error usually goes down for a while and then it increases. So early stopping is just stoppint he training process midway. Why does this work?

When you haven’t run many iterations for NN yet your parameters W will be close to zero. As before training for a long time W is quite small. Similar to L2 regularization by picking the neural network with smaller norm for W, hopefully the neural net is overfitting less.

Downside of early stopping

Orthogonalization – want to think about one task at a time. First we find out the algorithm, run t and then we check and reduce overfitting. Early stopping couples the two task and you no longer can work independently. As early stopping you are breaking your optimising of cost function J and also you are trying not to overfit. Mixing the two problems instead of working upon it one by one.

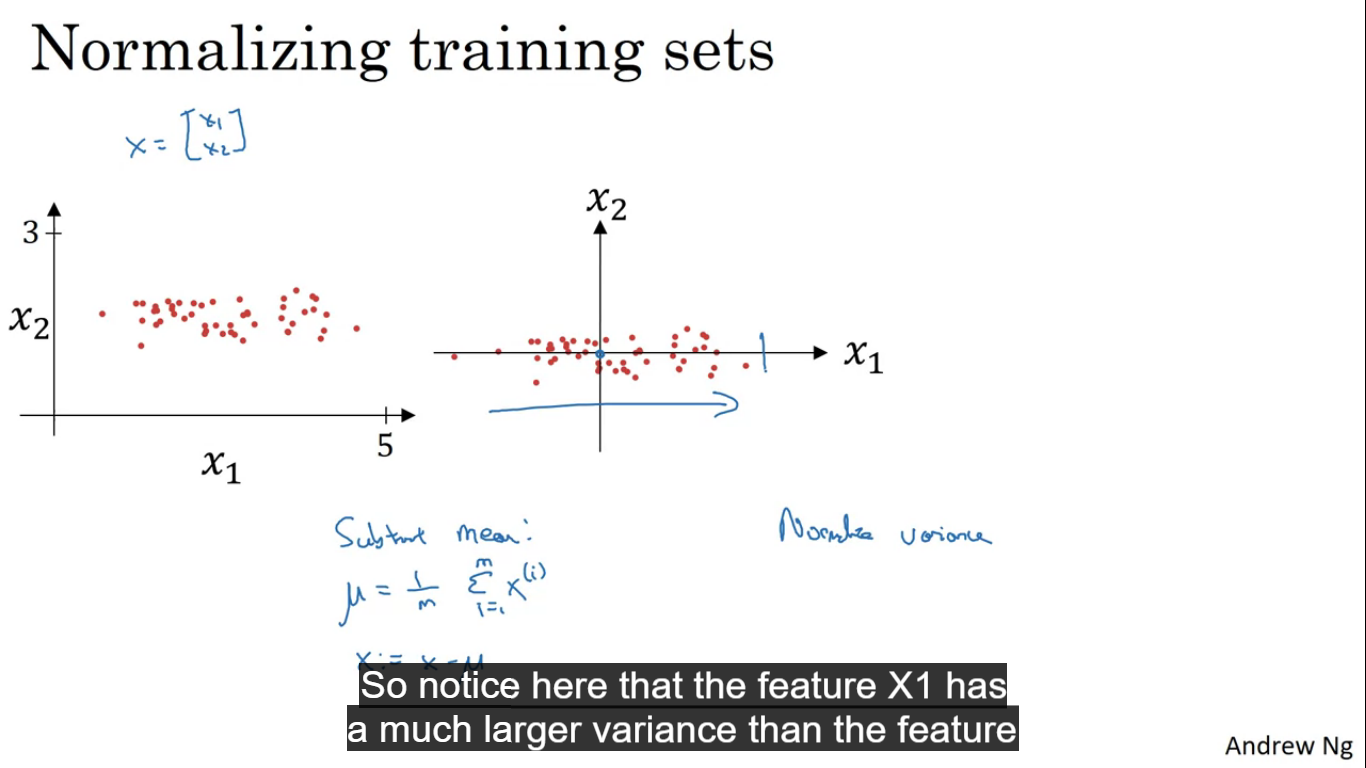
Alternatively you can use L2 regularisation, this makes the search space of hyper parameters easier to search over but you might have to try a lot of values of regularisation parameter lambda. Early stopping gets us small, medium and large sized W without having to try lot of values of lambda and getting a similar effect to L2 regularisation.

**Normalizing Inputs**

One technique that will speed up the training is when you normalise the inputs.

Normalizing the inputs corresponds to two steps,

* Subtract mean moving the training set until it has zero mean.
* Normalize the variances.



Mean = (1/m)\*sum(x(i))

X = X – mean

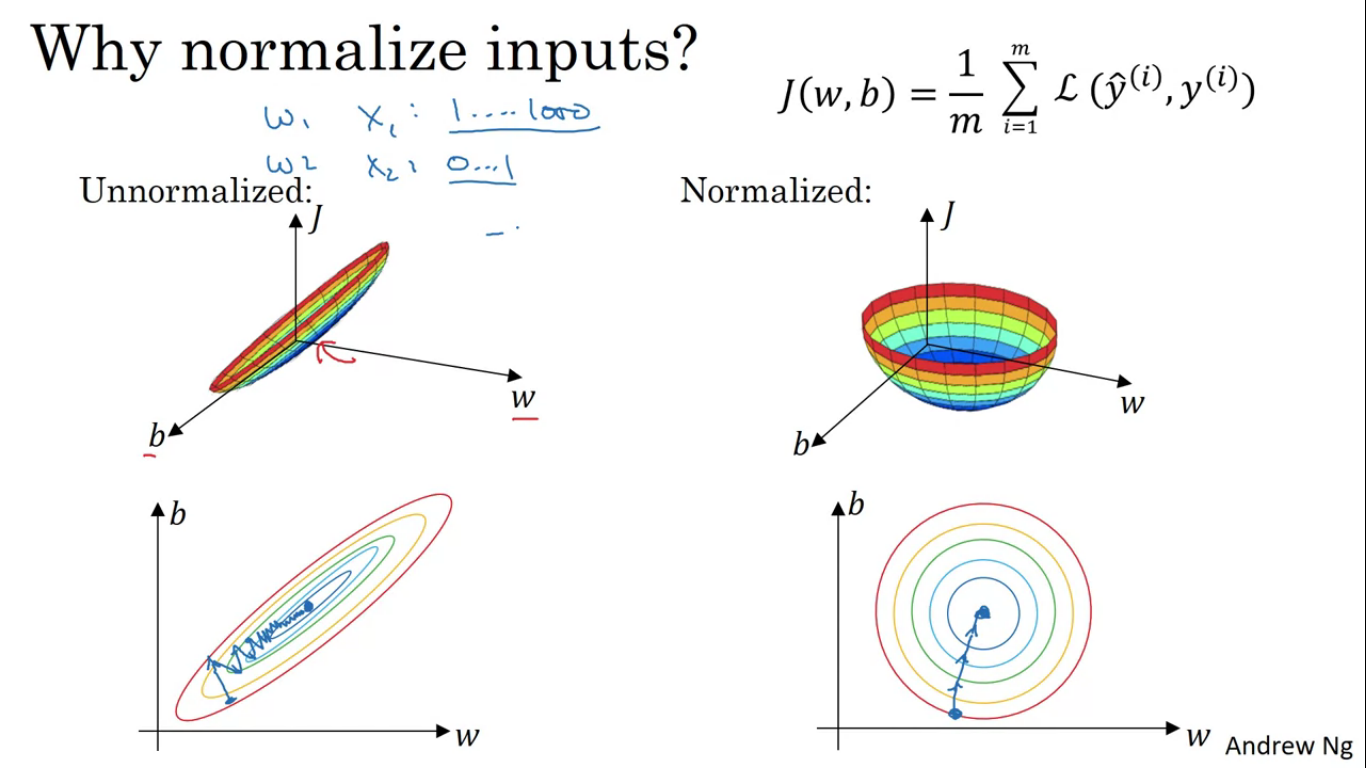
Sigma^2 = (1/m) sum(x(i) \*\* 2)

x/= sigma

After normalising the variance of x1 and x2 are both equal to one.

Don’t normalise the training and test set differently, use the same value of mean and sigmasquare.

**Why do we want to normalise the input features?**

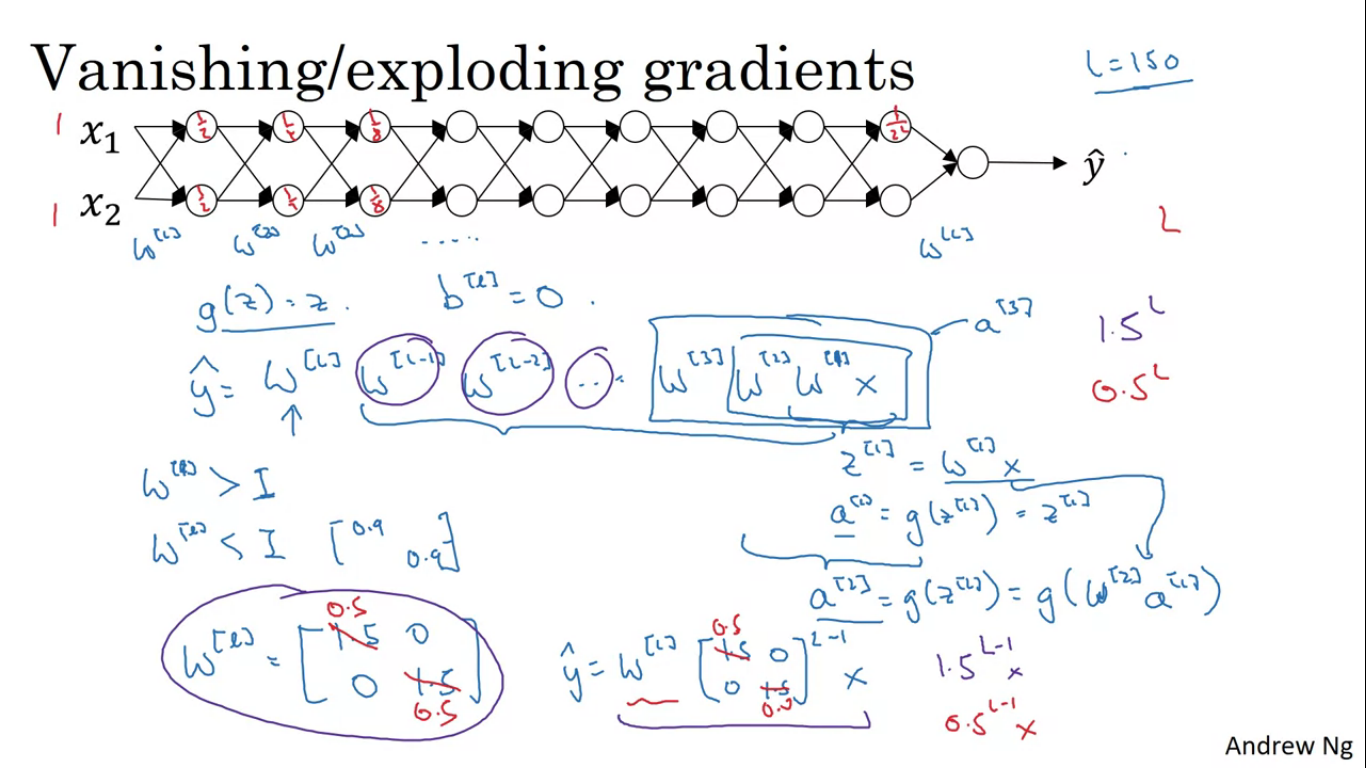


Un-normalised input features will make our cost function look like this. If our features are on very different scales like x1 ranges from 1-1000 and x2 from 0-1, then the range of parameters for w1 and w2 will end up taking many different values. So we would have very elongated function.

If you normalise the cost function would look more symmetric.

For un normalised we might have to use a very small learning rate as the gradient descent would need a lot of steps before it finally finds a way to the minimum. Whereas in spherical contours wherever you may start you’ll get straight to minimum.

**Vanishing and Exploding Gradients**

****

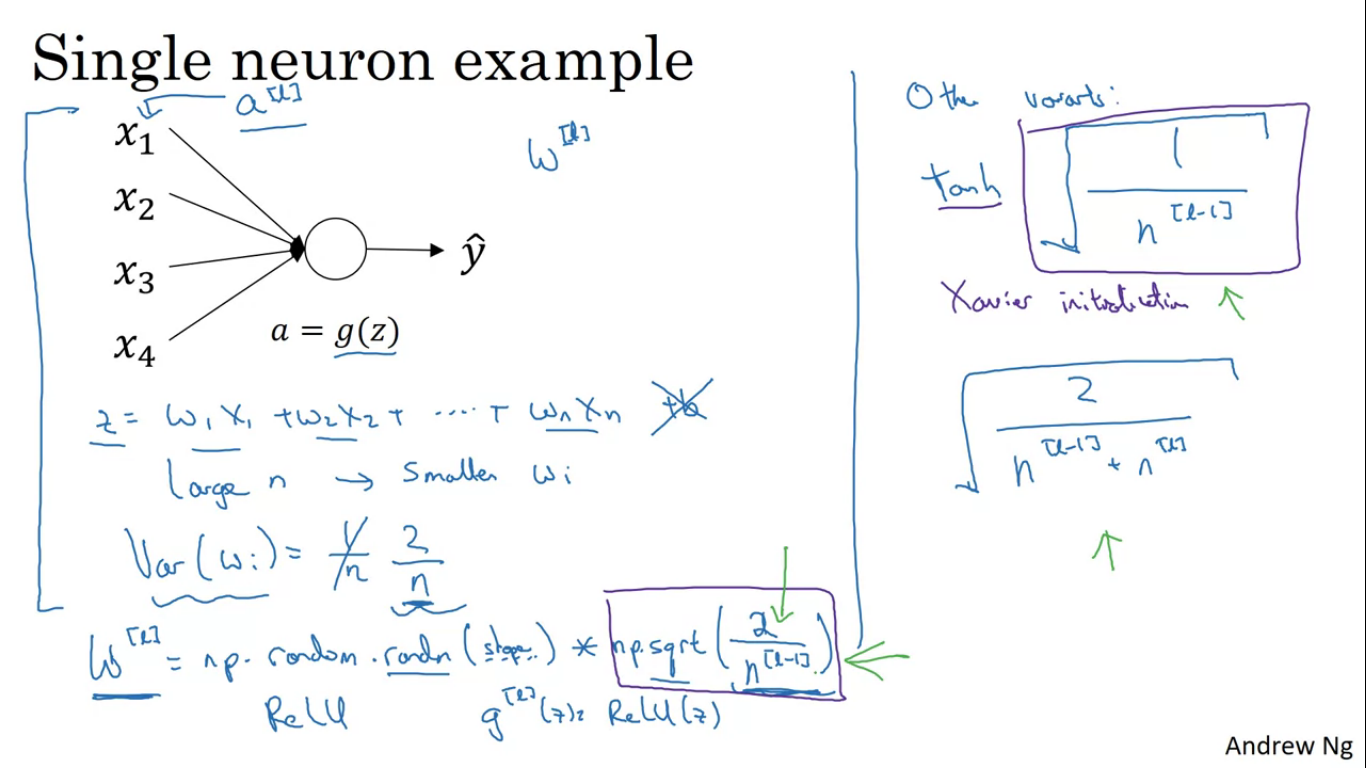
When training deep networks, the slope can sometimes get very small or big making the training difficult. How we can use careful choices of random weight initialisation to significantly reduce this problem.

W[l] is just a little larger than the identity, let’s say 1.5 so activations will increase 1.5^l exponentially and in a deep network this will become very large, similarly w[l] = 0.5 then the activations would keep o decreasing to 0.5,0.25,0.05... and in deep network it exponentially decreases.

So W[l] > I then the activations and explode

And W[l] < I then the activations can vanish.

**Weight initialisation for Deep Networks**

****

Partial solution to exploding or vanishing gradients is better or more careful choice of random initialisation for our NN.

Z = w1x1 + w2x2 + ... + wnxn

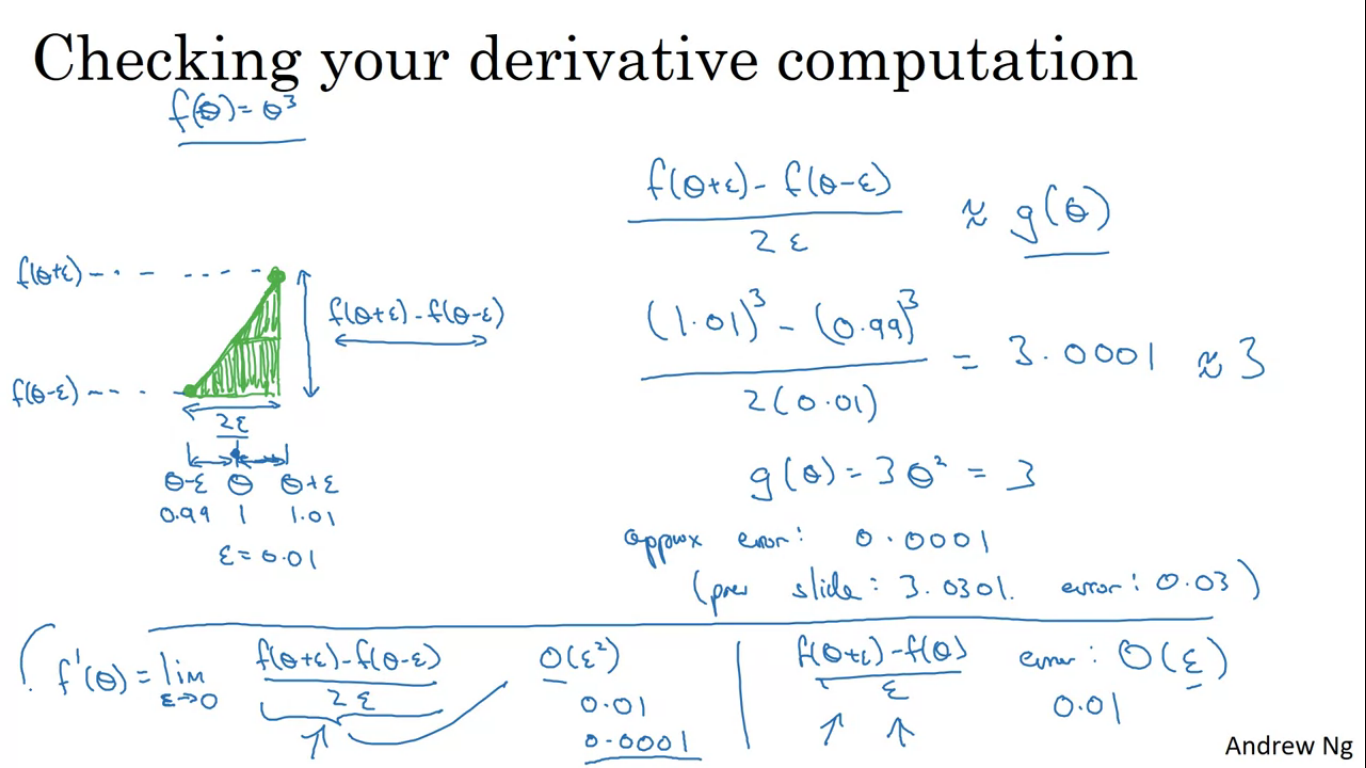
So larger n, we want smaller wi, since we have to add many wi and still want our Z to be small so wi need to be small. One reasonable thing to do is to set variance of w = (1/n) where n is the number of input features going into the neuron. So

W[l] = np.random.randn(shape) \* np.sqrt(1/n^[l-1])

If working with relu then setting the variance as (2/n) works better.

So this is not too much bigger or smaller than 1 so it doesn’t explode or vanish.

**Numerical approximate computation of gradients**

****

Sometimes we write all the equations and we are not 100%sure whether we have implemented the back propagation correctly. In order to build up to gradient checking lets first talk about how to numerically approximate computations of gradients.

So we find out theta + epsilon and theta – epsilon and if we find the height over width of the bigger triangle then we find out that this gives a better approximate of the derivative.

So derivative formula = (f(theta+epsilon) – f(theta-epsilon)) /2\*epsilon

F(theta) = theta^3 and theta = 1

= 1.01^3 – 0.99^3 / 2/0.01

=3.0001

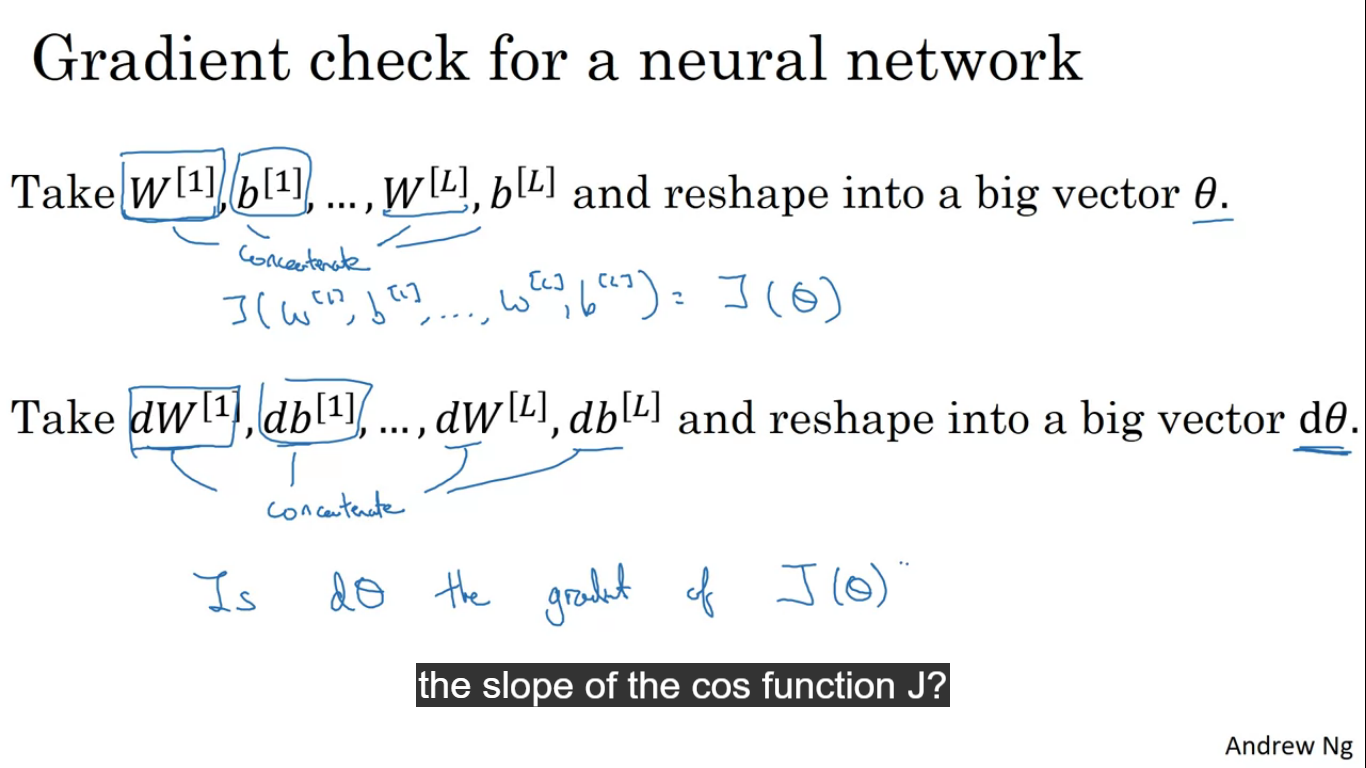
Whereas if we actually find the derivative through differentiation = 3\*theta^2

= 3\*1 = 3

So approximation error is 0.0001.

When we had taken only theta and theta+epsilon the error was 0.03 but now it has reduced to 0.0001.

**Gradient Checking**

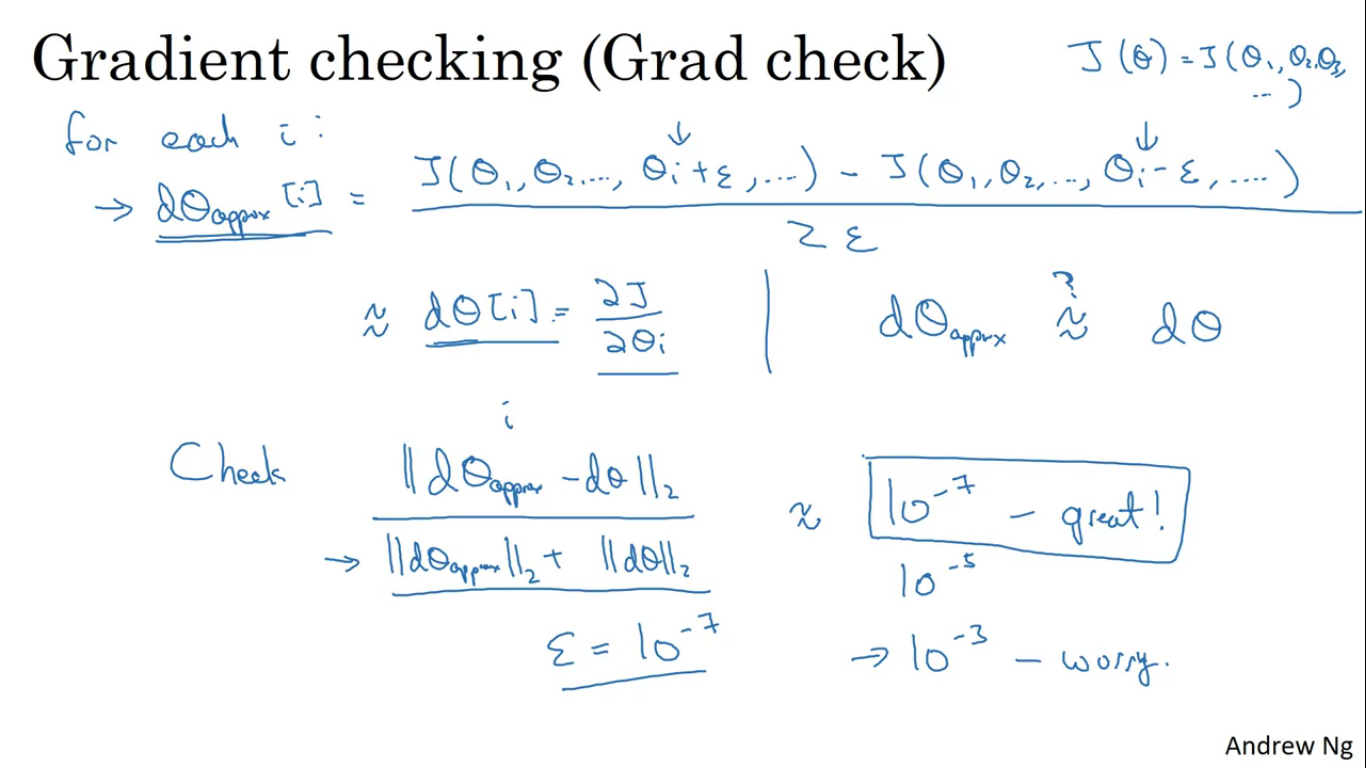
****

Our NN will have some parameters W[1],b[1]...w[n],b[n] etc. To implement gradient checking first you should take all parameters and reshape them into a giant vector data, that is taking all W and converting them to vector and then concatenating all of them together.

Also reshape dW[1]...dW[n] to a big vector dtheta.

Is dtheta the gradient of cost function J?

To do that we do as in the image



If the formula gives a value of 10^-7 or smaller then that means the derivative approximation is correct.

If the formula gives 10^-3 then its worry since it may have a bug.

Implementation notes for Grad Check

Don’t use Grad Check in training, use only for debugging as it is very slow.

If algorithm fails grad check, look at the components and try to identify the bug. If dthetaapprox is really very different from dtheta , find out the bug.

When doing grad check remember your regularization term, so when computing dtheta it is calculated with the regularization term added.

Grad check does not work with dropout. As dropout me every iteration, random neurons are dropped so there is not a easy to find out cost function that dropout is doing gradient descent on.

Run grad check at random initialisation. W and b close to zero but as the forward prop happens W and b rises but it is possible that backprop works with W and b nearly equal to 0, and becomes inaccurate when W and b becomes large.