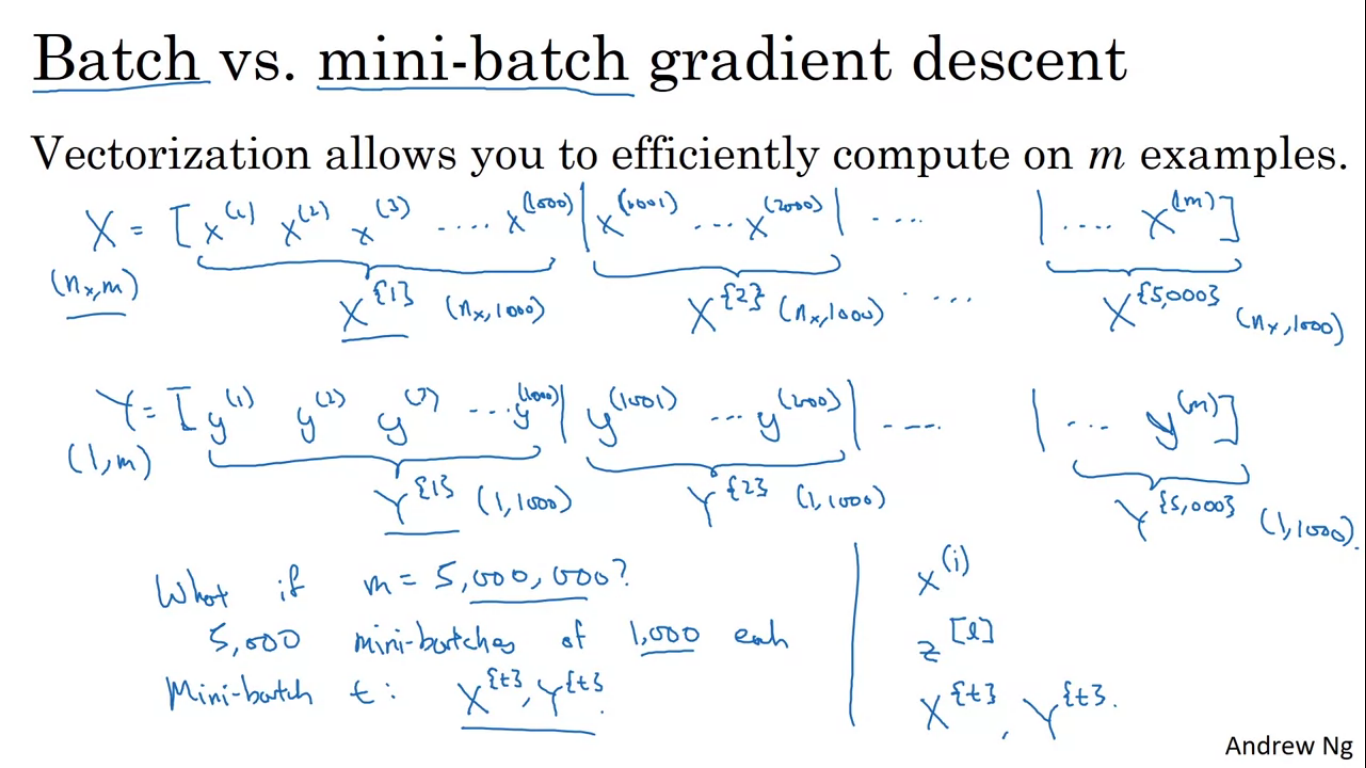
**Optimization Algorithms**

It enables us to train our neural networks much faster. Deep learning works best in the regime of big data. Training on a large data set is just slow. So having fast optimization algorithms help us to train our model quickly.

**Mini – Batch Gradient Descent**



Vectorisation allows us to efficiently compute on all m examples without an external for loop.

So we take our training examples and put them in the matrix X. Similar for Y.

**Dimensions of X is (nx,m) and Y is (1,m).**

Although vectorisation increases speed, but if m is very large, it would still be slow.

So before we take one step of gradient descent we have to process an entire training set and again for the next step of gradient descent you would again have to do it all above.

We can do a little faster if we start to make some progress on gradient descent even before we finish processing our entire training set of large size say 5 million examples.

To do above we split our entire training set into baby training sets and these baby training sets are called as mini batches. Let’s say each mini batch has 1000 examples each.

So x{1} = [x1,...x1000], x{2} = [x1001,...x2000]

So if we have 5 million training examples so we have 5000 mini batches.

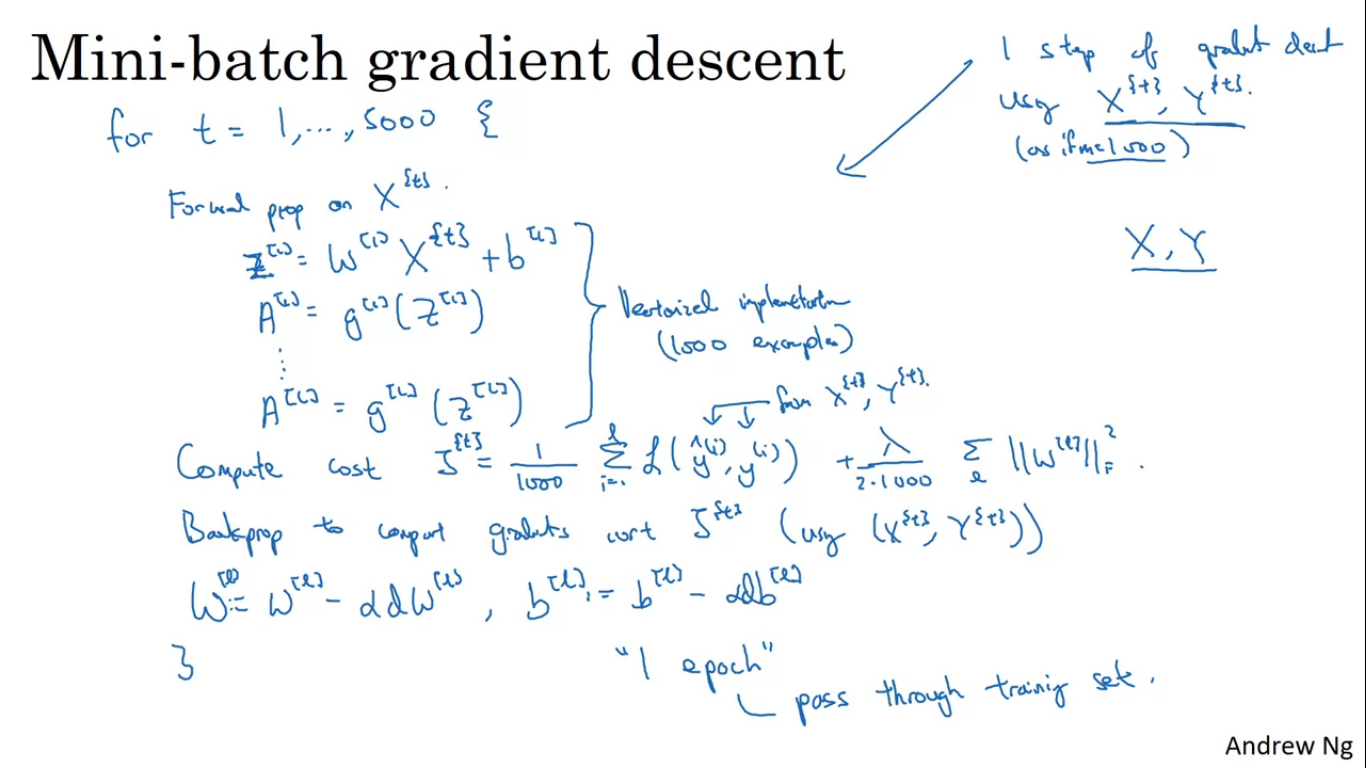
Similarly for y{1} = [y1,..y1000],y{2} = [y1001,...y2000]

Mini batch t: x{t} and y{t} 1000 training examples with a corresponding input output pairs.

* **X(i) – ith training example**
* **X[l] – to index into different layers of the NN**
* **X{t} – to index into mini batches – x{t} dimensions are (nx,1000) y{t} dimensions are (1,1000)**

In batch gradient descent we process a mini batch x{t}, rather than processing entire training et at the same time.

**Algorithm of mini batch gradient descent**



For loop limit 1-5000, since there are 5000 batches. Inside the for loop, we implement one step of gradient descent for x{t} and y{t}. It is as if you had the training set of size 1000 examples. The process can be vectorized for training of 1000 examples.

It is as follows:

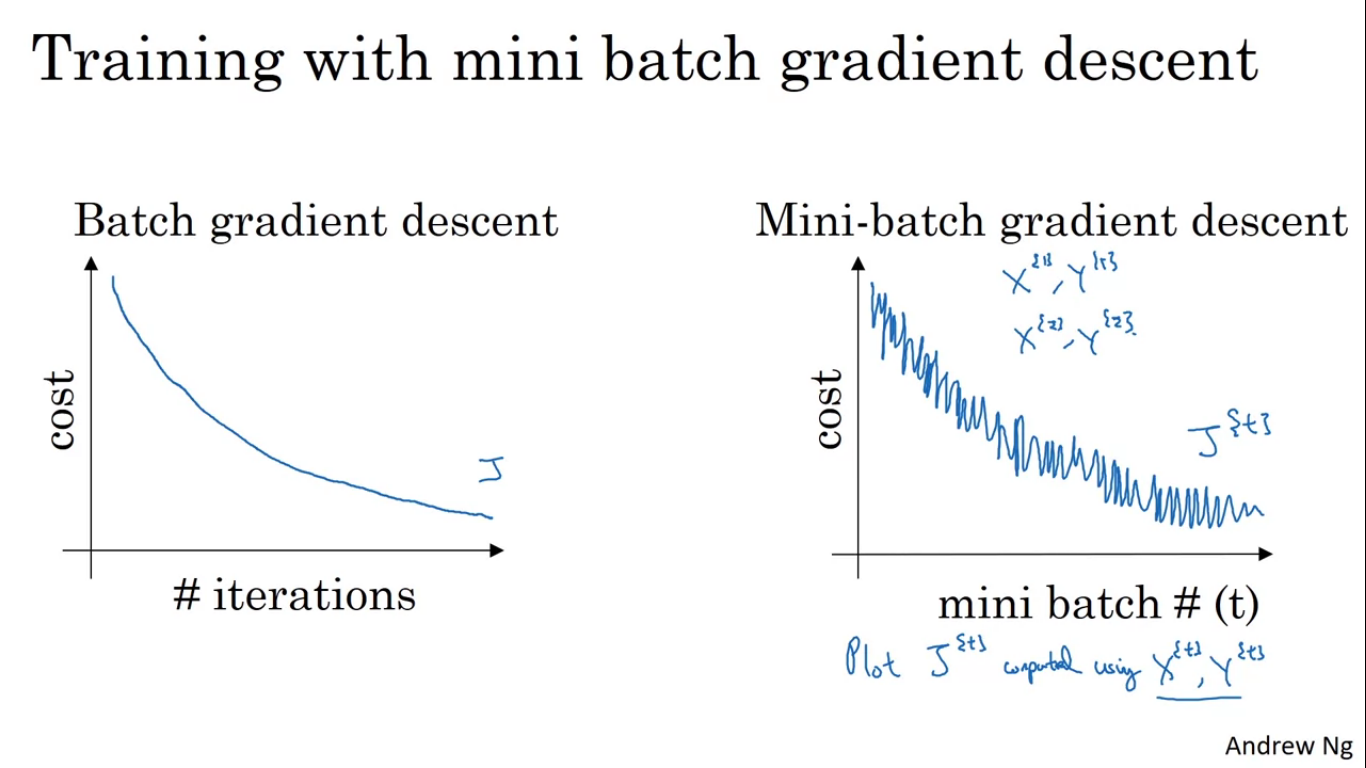
First implement forward propagation as shown this forward prop is a vectorized version with only difference of 1000 training examples instead of 5 million training examples.

Then we find out the cost function but it is of this specific batch so J{t}.

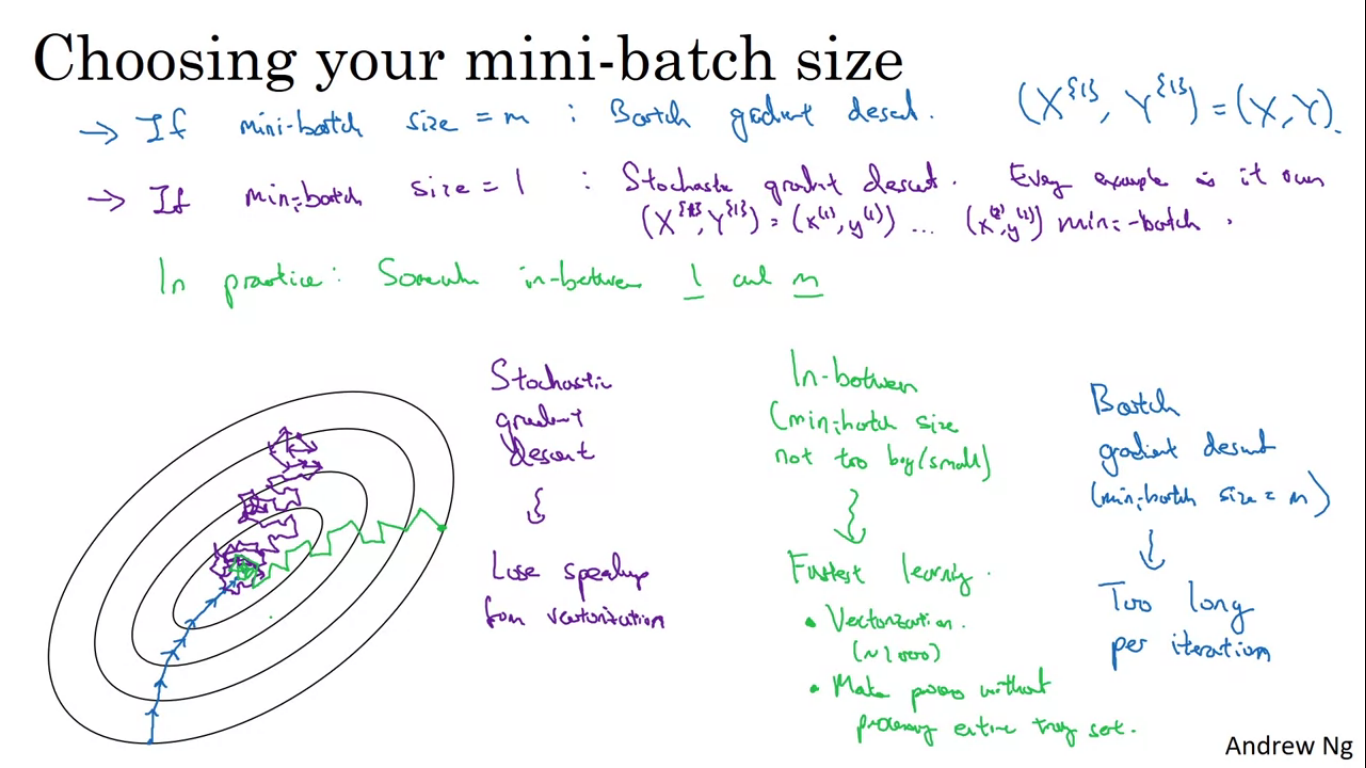
Next back prop to compute gradients with respect to J{t}.

The done is one epoch of training set, an epoch means a single pass through the training set.

With Batch gradient descent you go through entire training set on every iteration and expect the cost to go down on every single iteration, so if the cost function J goes up on any single iteration, then there is something wrong.



On mini batch gradient descent if you plot the cost function then it may not decrease on every iteration. It is as if on every iteration we are training on a different training set. So upon plotting the cost function J it should look like as shown, it should trend downwards but it is going to be a little bit noisier. The reason of noise is that x{1} and y{1} is a little bit noisier is that maybe this batch is a little bit easier to train so cost is downwards but next batch maybe difficult to train so noise.



The main parameter to choose here is the size of the mini batch. M is the training set size, if mini batch size ==m then it is batch gradient descent. Main disadvantage of this is takes too much time, too long per iteration

If mini batch size == 1 then it gives an algorithm called stochastic gradient descent. Every example is its own mini batch. Noisiness can be reduced by using a smaller learning rate. Main disadvantage is we lose all the speed up received from vectorisation.

Blue = batch gradient descent, and purple is stochastic gradient descent.

Stochastic gradient descent is extremely noisy and it won’t ever converge, it just kinds of keeps oscillating around the minimum and never stay there.

In practice the mini batch size we use will be somewhere in between 1 and m. Uses as an advantage both vectorisation and making progress without needing to wait till we process the entire training set.

Guidelines for choosing the mini batch size

* If you have a small dataset just use batch gradient descent.(if less than 2000 then its alright to use batch gradient descent).
* Typical batch size is between 64-512. Because of the way memory is addressed, the code runs faster if we have a mini batch size in powers of 2.
* Make sure the mini batch x{t} y{t} fits in the CPU/GPU memory.

Mini batch size is another hyper parameter which we need to find out to keep our cost function to J.

Optimization algorithms faster than gradient descent

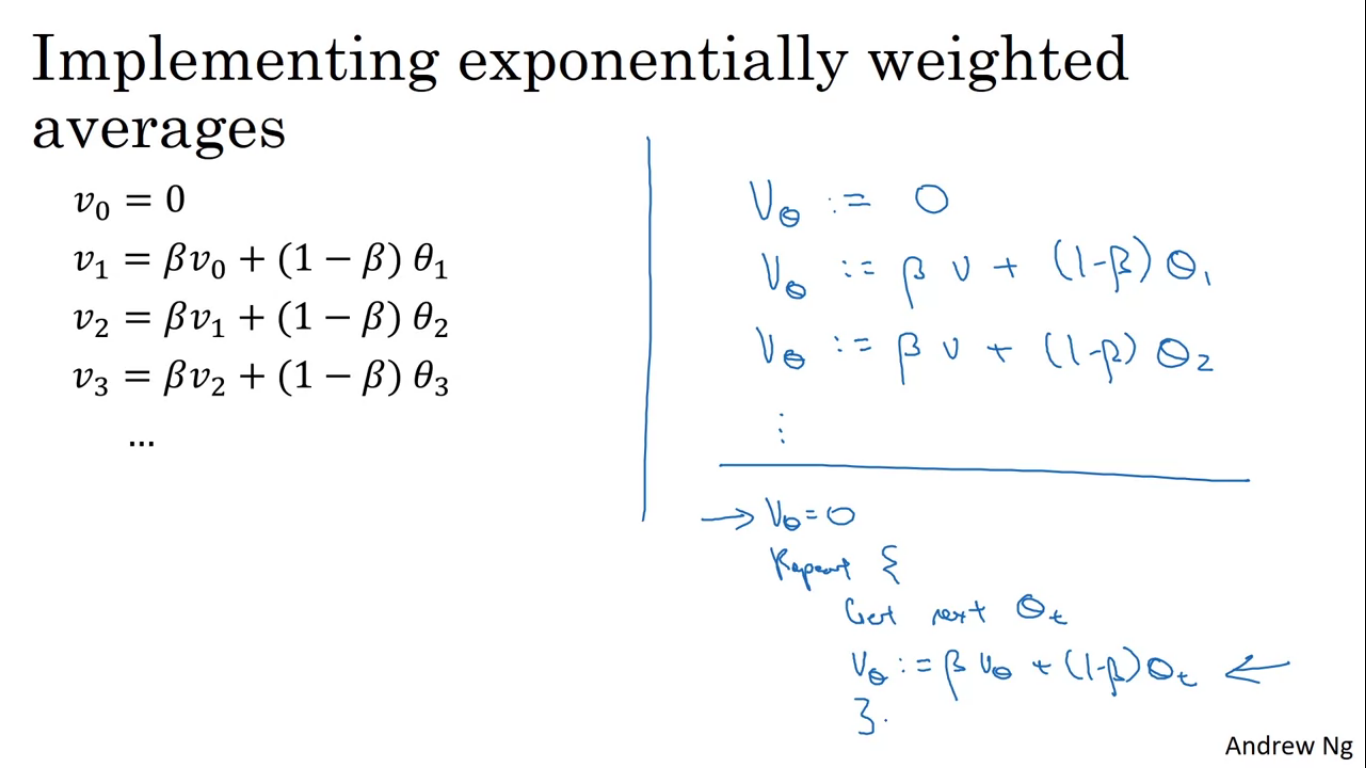
To use those algorithms we need to be able to use something called **exponentially weighted averages.**

So v0 = 0

V1 = 0.9v0 + 0.1theta1

V2 = 0.9v1 + 0.1theta2

Vt = 0.9v(t-1) + thetat



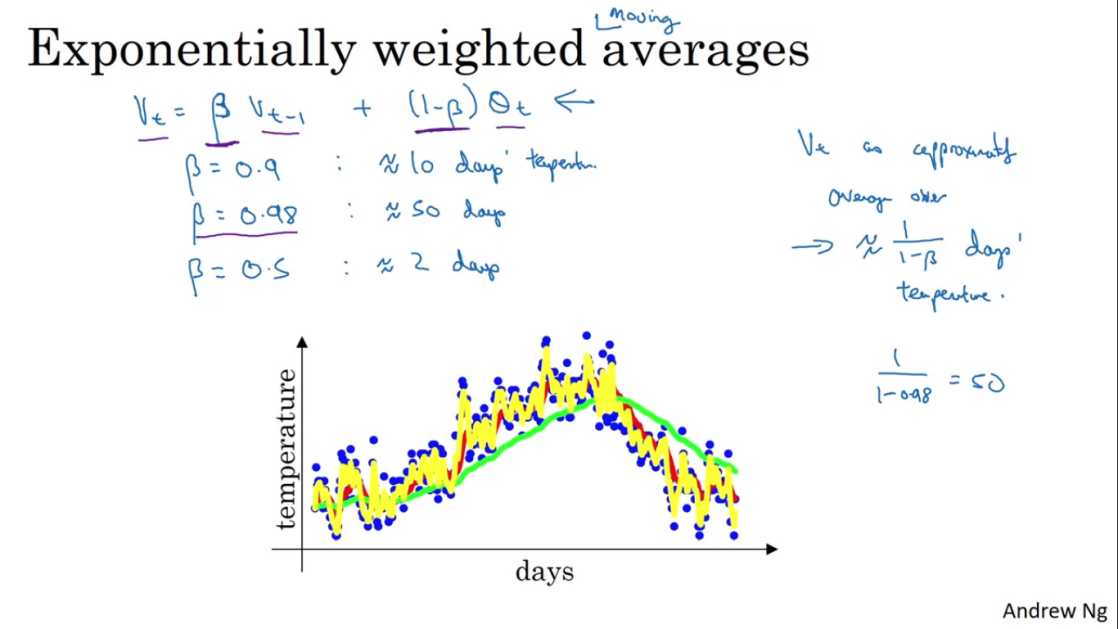
So if we plotted this we would get a moving average/ exponentially weighted average of the daily temperature.

Vt as approximately average over 1/(1-beta) days, where beta=0.9 so its the average over the last 10 days temperature. If beta=0.98, then its roughly averaging over last 50 days temperature.

With high value of beta the plot we get is much smoother because we are now averaging over more days of temperature. When the averaging window becomes large the formula adapts more slowly as it depends more on previous value and less on the current value.

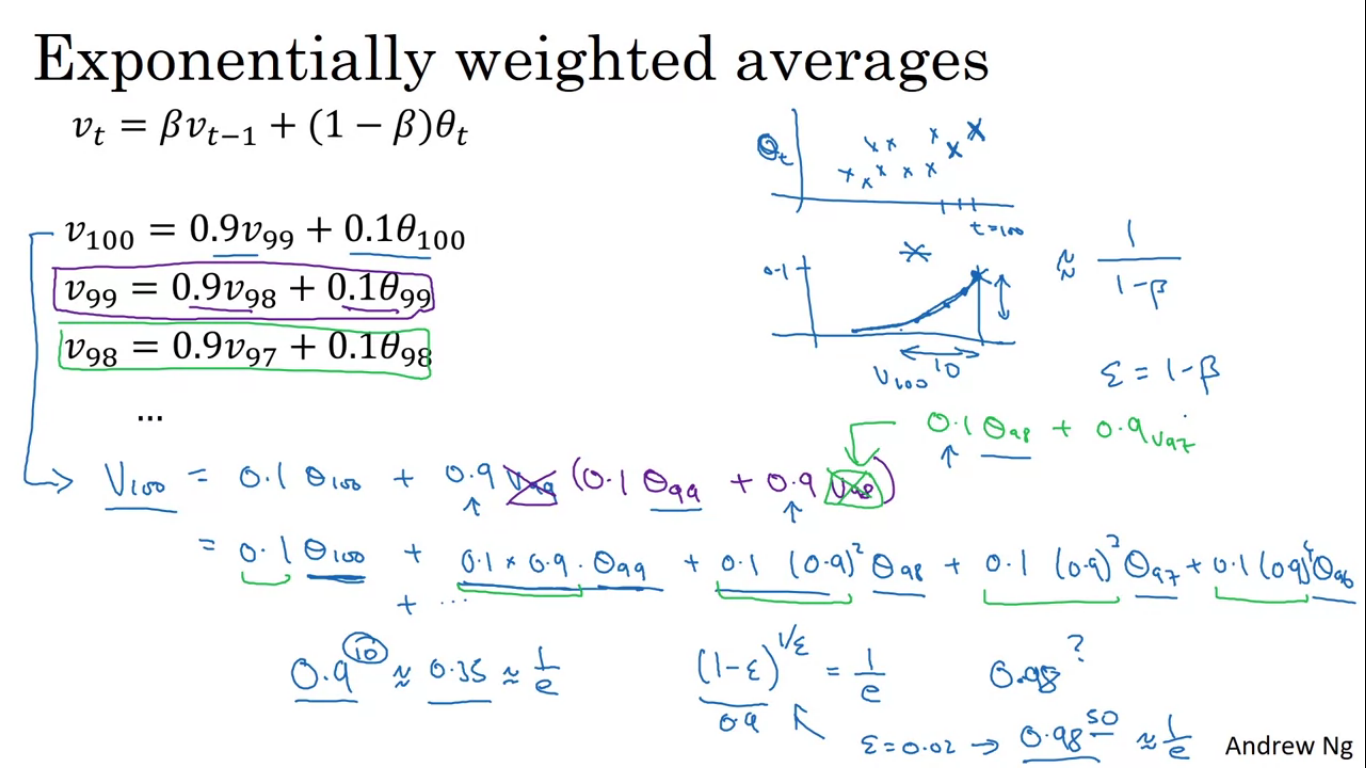
Averaging over shorter window is much more noisy and susceptible to outliers.

**Equation for implementing exponentially weighted averages**



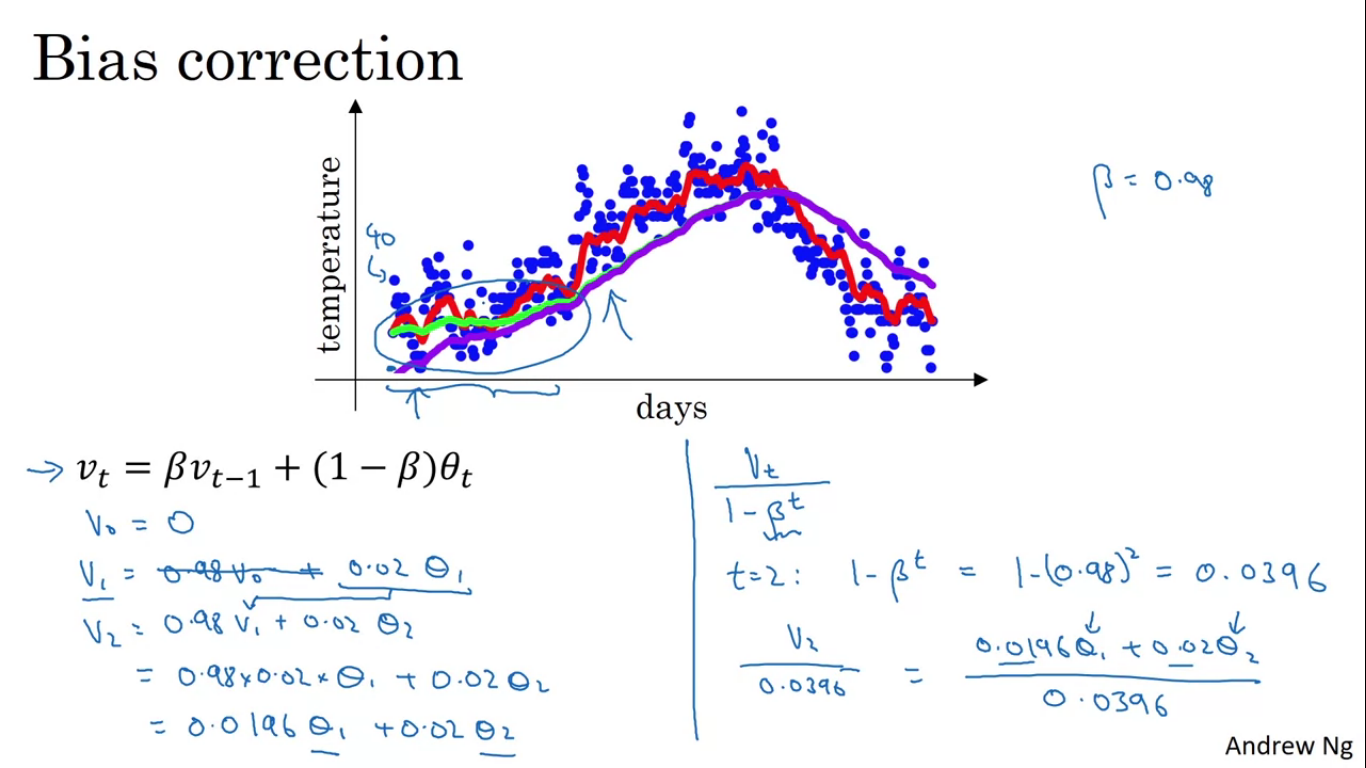
Vt = beta\*v(t-1) + (1-beta)\*thetat

Next whole is in the video and image itself.



Exponential weighted averages takes a very little memory.

**Bias correction**

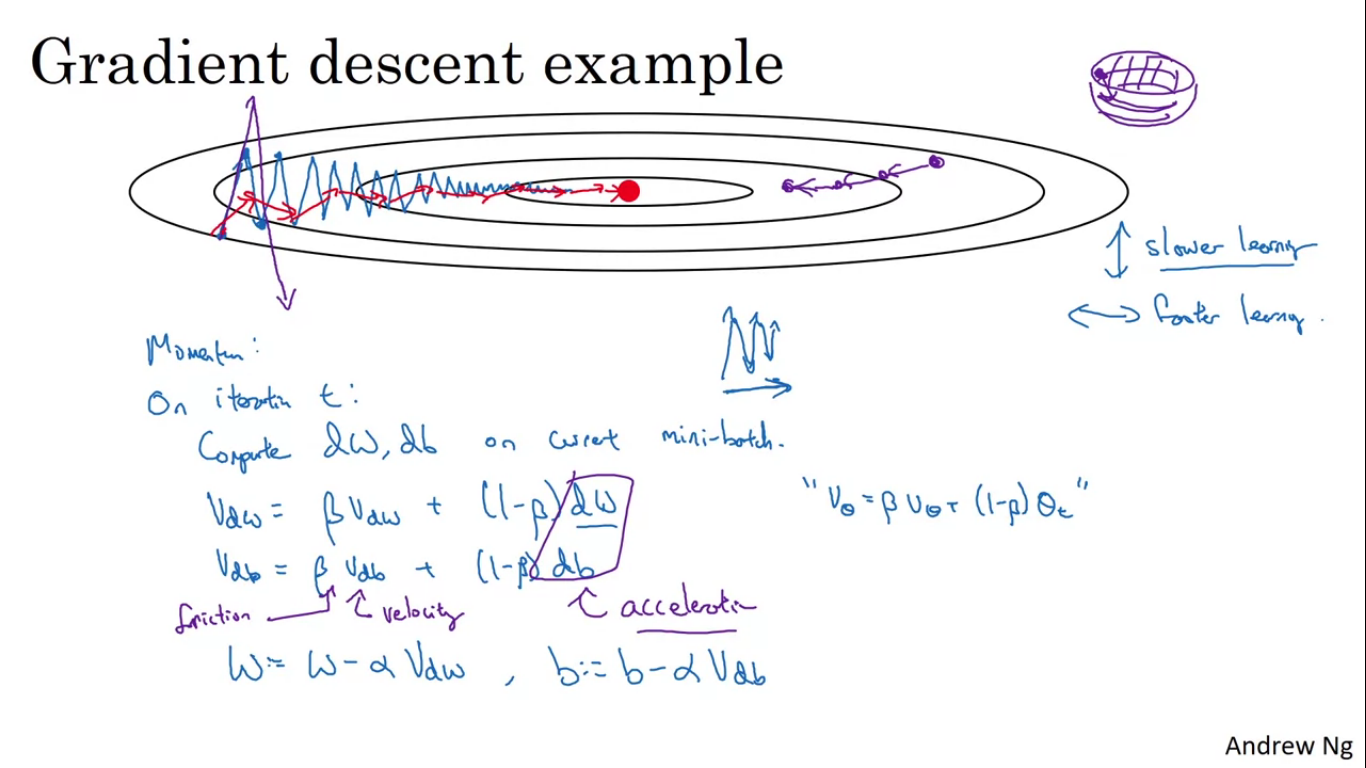
****

Initial phase of estimate does not work very good as shown here, we do not get the green line for beta=0.98 but rather we get the purple line which is very low start. So we need a correction term.

So instead of putting the value of vt, we put the value of vt/(1-beta^t)

So when t is large enough the bias correction makes no difference, as when t is large the purple line and green line overlaps but during initial phase bias correction helps us get better estimates.

**Gradient Descent with Momentum**



Always works faster than the standard gradient descent. The basic idea is to compute exponentially weighted average of the gradients and use that gradient to update our weights .

The up and downs (randomness) of gradient descent slows down our learning and prevents us from using a much larger learning rate. So if we use a larger learning rate we might end up overshooting, as shown in purple pen.

On the vertical axis we want the learning to be slower, since we don’t want to overshoot, but on the horizontal axis we want faster learning.

What’s the change with the case of momentum?



On each iteration t we compute usual dW, db on the current mini batch, we compute moving averages for the derivatives we are getting. The whole computation is shown on the image. What this does is that it smooth out the gradient descent.

In the vertical the derivatives are both positive and negative, so averaging out will give zero/slow learning and in the horizontal direction will be pretty big since all the values are positive.

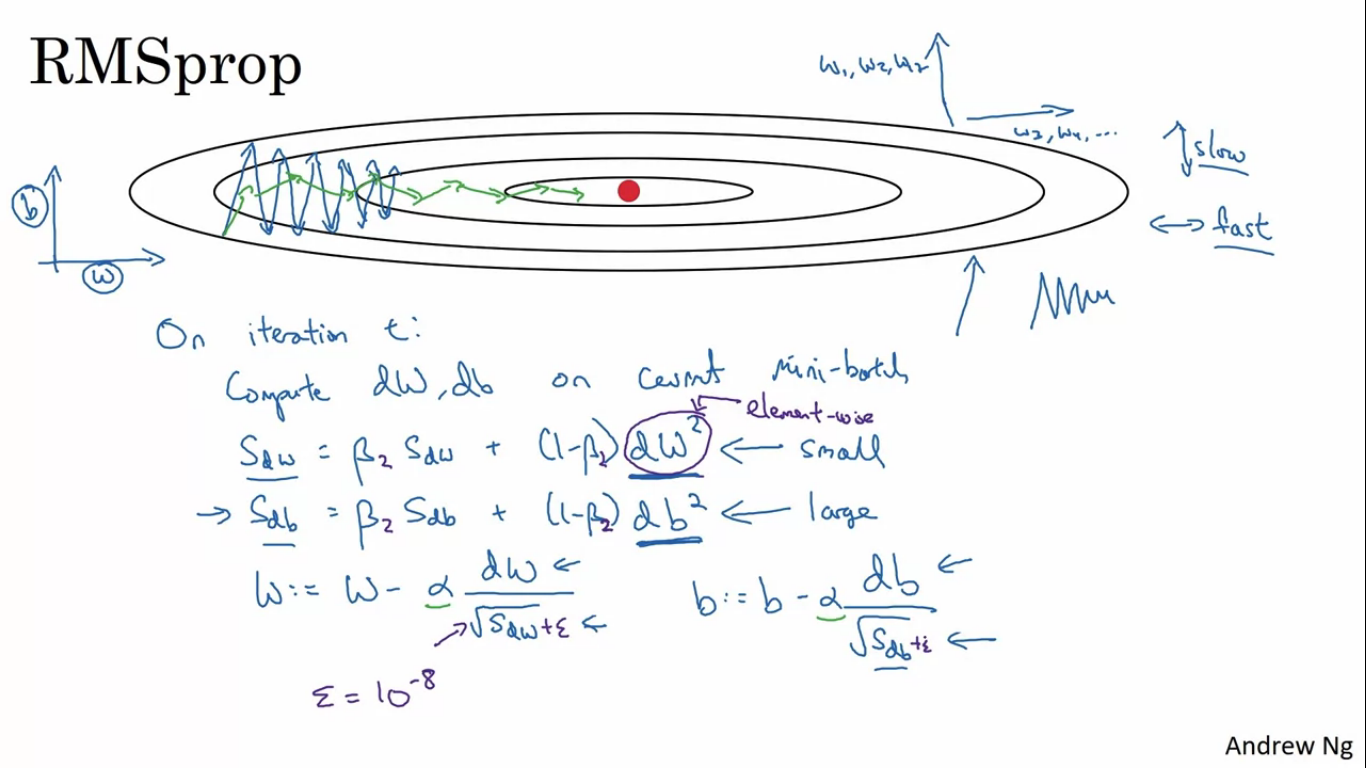
So gradient descent with momentum takes much quick actions in the horizontal direction and converges the cost.

Now we have two hyper parameters- learning rate alpha and the parameter beta which controls exponentially weighted average. Most common value of beta=0.9

No need to do bias correction here because in practice moving average gets warmed up after 10 iterations.

vdW is a mtrix of zeros with the same dimensions as dW,W, and vdb is a matrix with the dimensions of db,b.

**RMSProp (Root Mean Square Prop)**



It also speeds up our gradient descent. With normal gradient descent you may get huge oscillations in the vertical direction while also making progress in the horizontal direction slowly.

Let’s say vertical axis is the parameter b and horizontal axis is parameter w, so we want to slow learning in b direction and speed up learning in the horizontal direction.

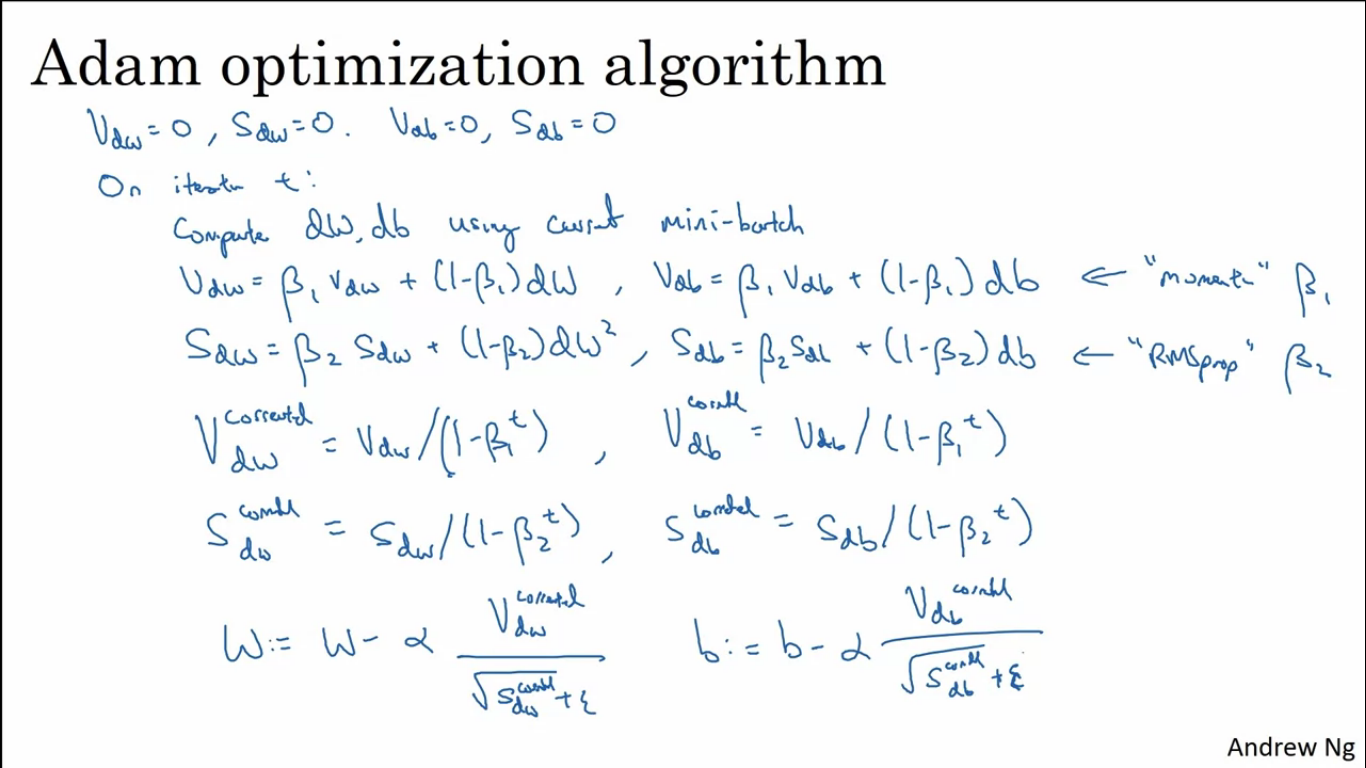
On iteration t it computes dW, db on current mini batch so we define instead of vdw as Sdw . so what we are doing is that we are keeping exponentially weighted average of the squares of the derivatives. The squaring is an element-wise operation.

If sdw is relatively small, in the w equation we would be dividing by small number in order to fasten the update of weights in the horizontal direction and we want sb to be relatively large so when we divide by sdb it would be small and it would slow down b, just as we wanted. Indeed the derivatives are much larger than the vertical direction than in the horizontal direction. The function is sloed very much steeply in the vertical direction(b) in comparison to the horizontal(W) direction.

Updates in the vertical direction are divided by much large number damping the oscillations and updates in the horizontal direction are divided by a smaller number .

Another advantage is that you can also use a larger learning rate alpha without diverging in the vertical direction.

**Adam Algorithm**



Basically taking momentum and RMSprop and putting them together. To initialise vdw=0, sdw=0, vdb=0, sdb=0. Then compute dw, db using current mini batch. And then do the momentum exponentially weighted average. Beta1 is for momentum and beta2 is for rmsprop portion. In the implementation of adam we do implement bias correction.

This algorithms has a number of hyper parameters

Learning rate (alpha)

Moving average of dw (beta1 = 0.9)

Moving weighted average of dw^2 (beta2 = 0.999)

Epsilon = 10^-8

**Adam stands for adaptive moment estimation**

Beta1 to compute the mean of the derivatives, called as first moment and beta 2 is used to calculate the mean of derivatives squared and it is called as second moment.

**Learning Rate Decay**



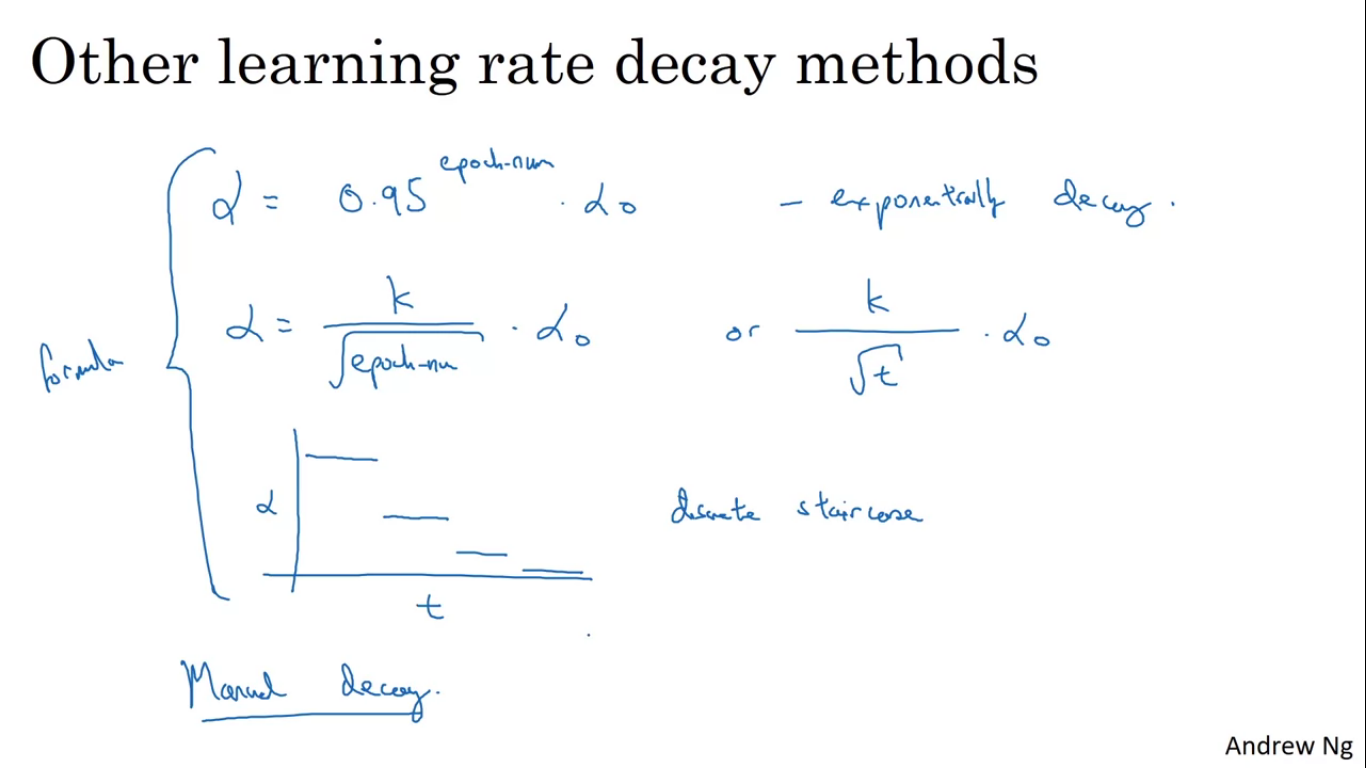
One thing that might speed up our learning process is to slowly reduce our learning rate over time.

Why do we want to implement learning rate decay?

Suppose we are implementing mini batch gradient descent with a pretty small mini batch like 64 examples. So our algorithm steps will be noisy and will tend towards the minimum but it wont exactly converge as we are using some fixed value of alpha.

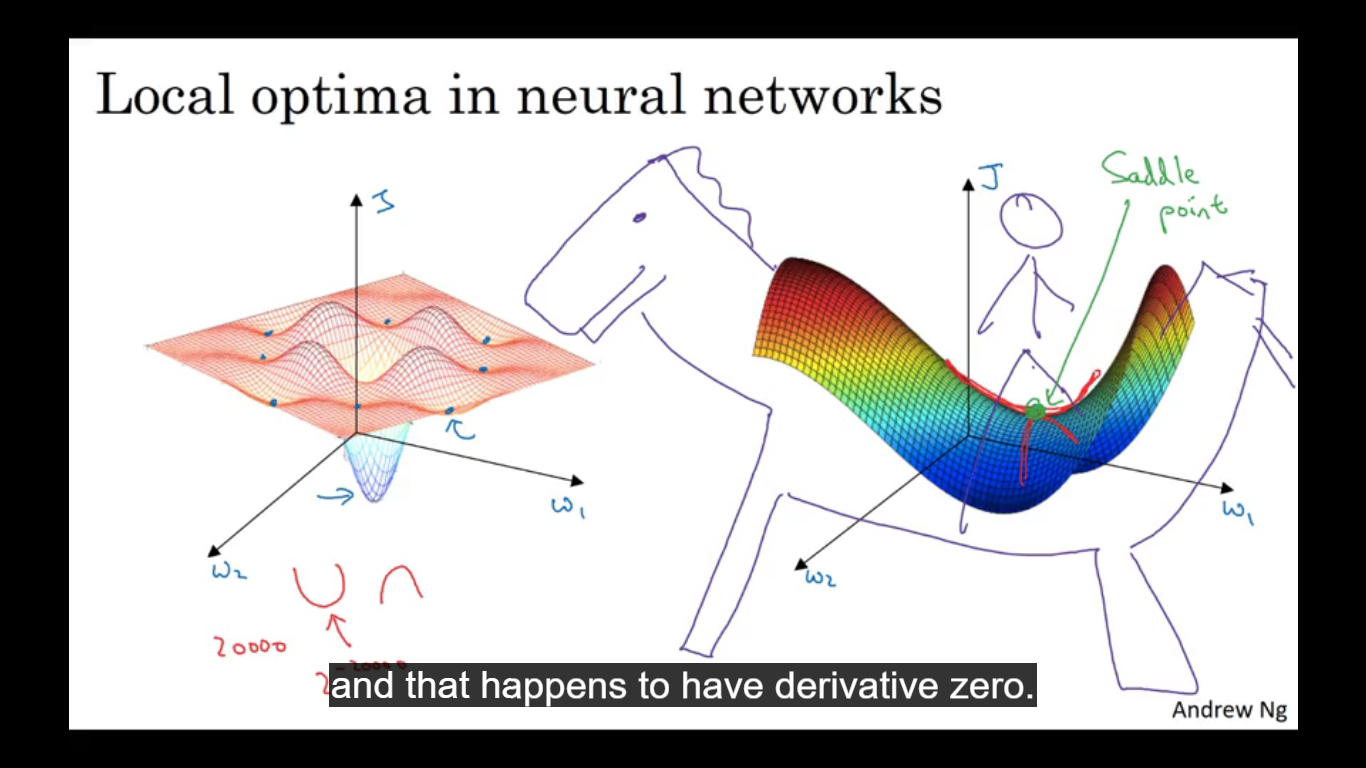
But if we were to slowly reduce our learning rate alpha then during initial stages we can have relatively fast learning as our alpha is large but as alpha becomes smaller the steps to take will become slower and we end up oscillating in the closer region to the convergence point.

1 epoch is 1 pass through the training set. So we set alpha accordingly to the image formula. Decay rate here is another hyper parameter which we might need to tune.



Another learning rate decay is exponential learning rate decay.

**Problem of Local Optima**



It is possible for gradient descent to get stuck in local optima rather than finding the global optima.

Most points of zero gradients in a cost function are saddle points. But if the gradient is zero it can either be convex like function or a concave like function.

Plateau is a region where the derivative is close to 0 for a long time. Convergence to the global minima on a plateau actually takes a lot more time. Algos like momentum, RMSprop and adam help a lot with our leaning algorithm in case of plateau.

We are unlikely to get stuck in a bad local optima provided we train a large enough network and cost function J is defined over relatively high dimensional space