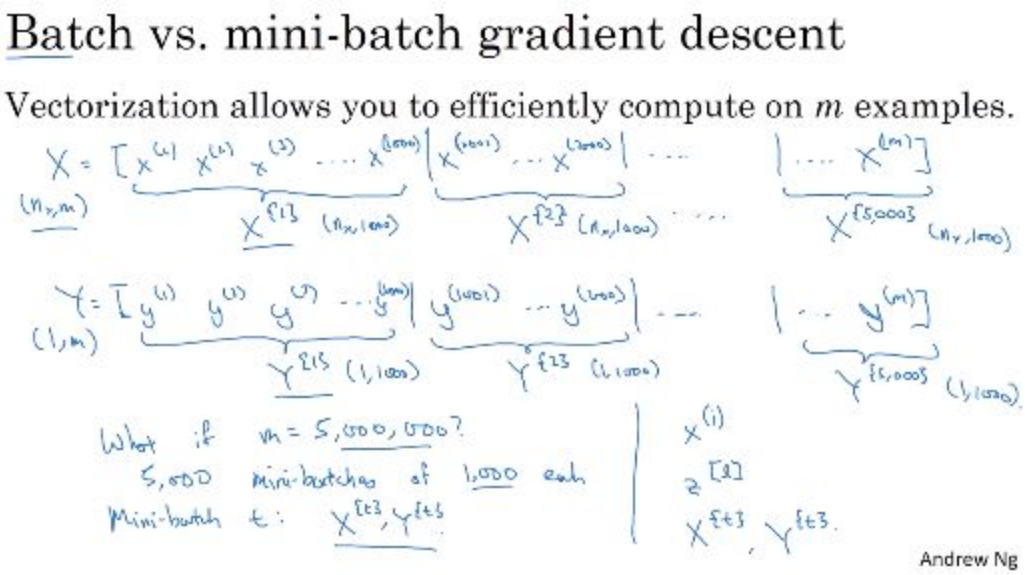
**Andrew Ng's Improving Deep Neural Networks Class**

# Week 2: Practical Aspects of Deep Learning

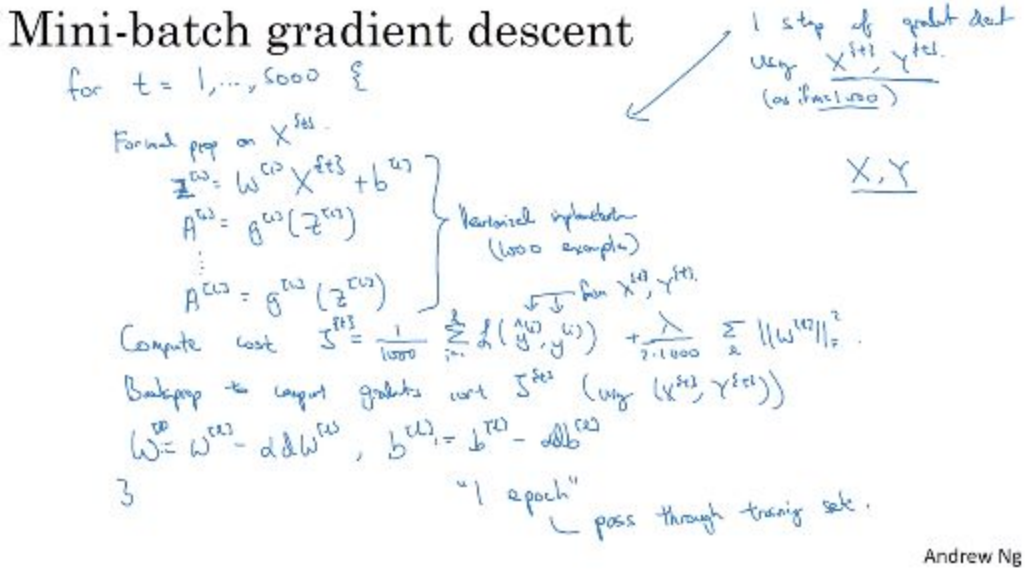
*Mini Batch Gradient Descent*

- Let's start looking at mini batch gradient descent by looking at the X and Y matrices. X contains all the training examples and is therefore equal to $[x^{(1)}, x^{(2)}, x^{(3)}, ..... , x^{(m)}]$. Y is also similar but it contains the labels so it is equal to $[y^{(1)}, y^{(2)}, y^{(3)}...., y^{(m)}]$. We live in the age of big data so it is common to have large datasets; imagine if our data set has 5 million training examples. Batch gradient descent will need to use all 5 million examples on every step making this process very slow. mini-batch gradient descent, on the other hand, only requires a few examples 'per batch'; in the example below we are using batches of 1,000 examples so for X we have $X^{\{1\}} = the\,first\,1,000\,examples$ and it has dimensions $(n_x, 1000)$, $X^{\{2\}} = [x^{(1001)}, ..., x^{(2000)}]$ and so on. Y goes through the same process. In our example we then end up with 5,000 mini batches of 1,000 examples each. Each mini batch t then is composed of $X^{\{t\}}, Y^{\{t\}}$.



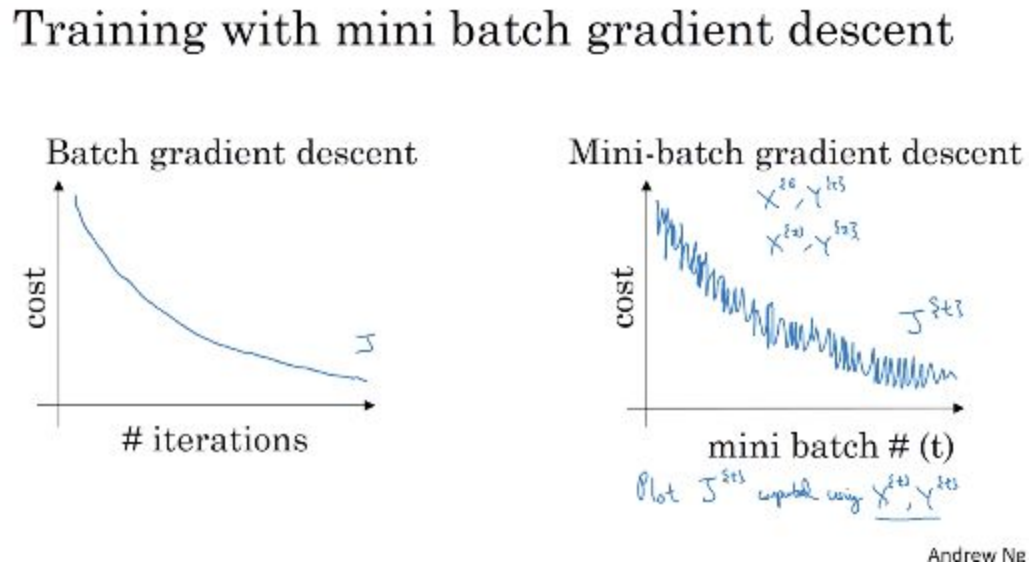

- How do we implement mini batch gradient descent? We start with a for-loop for all the training batches, using our previous examples we had 5,000 so that's the index for the for-loop, we then compute Z as $Z^{(l)} = w^{(l)}X^{\{t\}} + b^{(l)}$, A, until we get $A^{[L]}$. We then compute the cost, do backprop to compute our gradients using $(X^{\{t\}}, Y^{\{t\}})$ and update our gradients as follows: $w^{[l]} = w^{[l]} - \alpha dw^{[l]}\ and\ b^{[l]} = b^{[l]} - \alpha db^{[l]}$. This is one pass of gradient descent, also known as epoch.

- Summarizing, mini batch gradient descent is the preferred choice when we have a large training set. It runs much faster than gradient descent and is what everybody uses when training on a large data set.



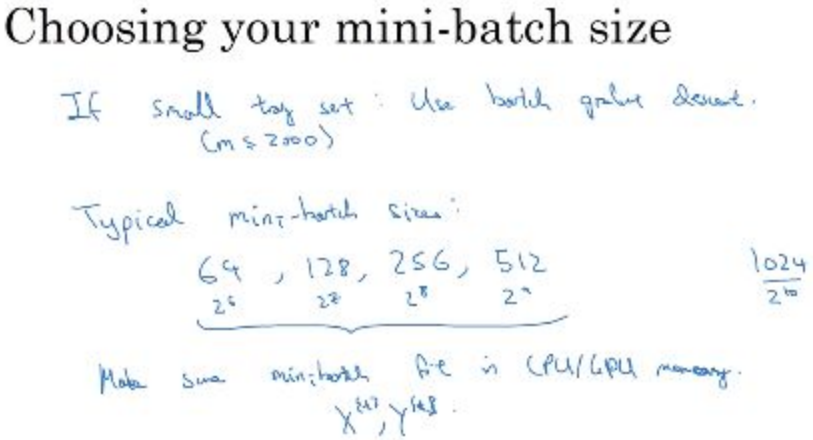

*Understanding Mini Batch Gradient Descent*

- When we train with regular batch gradient descent we expect to see a cost reduction every time. If we see an increase even in one pass we know there is something wrong. That is not the case with mini-batch gradient descent. Mini batch gradient descent is noisier because it is like training on a different training set every time. So one batch might be an 'easy' batch $X^{\{t\}}$, $Y^{\{t\}}$ and another batch $X^{\{t\}}$, $Y^{\{t\}}$ might be 'harder' (meaning it might contain some unlabeled examples). In general mini batch gradient descent should also trend lower over time but it will be noisier.



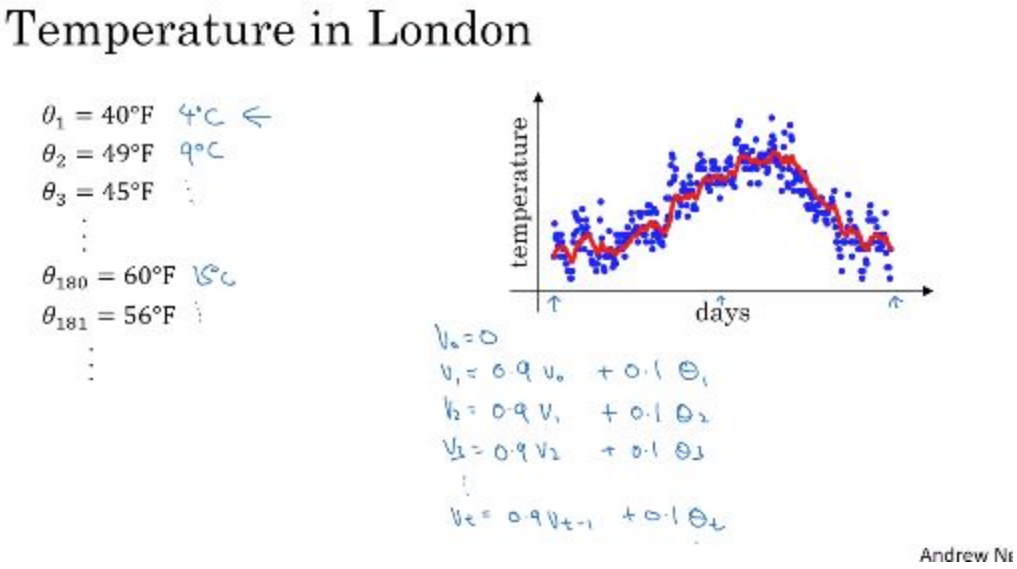
- How do we choose the mini batch size? Let's start by looking at the extremes: if we have minibatch size of m then we have batch gradient descent where $(X^{\{t\}}, Y^{\{t\}}) = (X, Y)$. If we have minibatch size = 1 then we have stochastic gradient descent where each training example is its own mini batch. In practice we choose something in between.


  In general, batch gradient descent takes a straightforward to the global minimum, which is good, but it takes too long per iteration because each iteration uses the whole training set. Stochastic gradient descent takes a very noisy road to the global minimum and will never converge, rather it will 'hang out' around that global minimum. The disadvantage of stochastic gradient descent is that we lose the speed from vectorization. Choosing a batch size of somewhere in between enables the fastest learning because we are still using vectorization and we don't need to wait to process the entire training set.
- Here are some guidelines for how to choose the mini batch size:
  - If we have a small training set: use batch gradient descent
  - Typical batch sizes are multiples of 2 (64, 128, 256, 512) because of the way computer memory is laid out and accessed
  - Finally, make sure the mini batch size fits in CPU/GPU memory

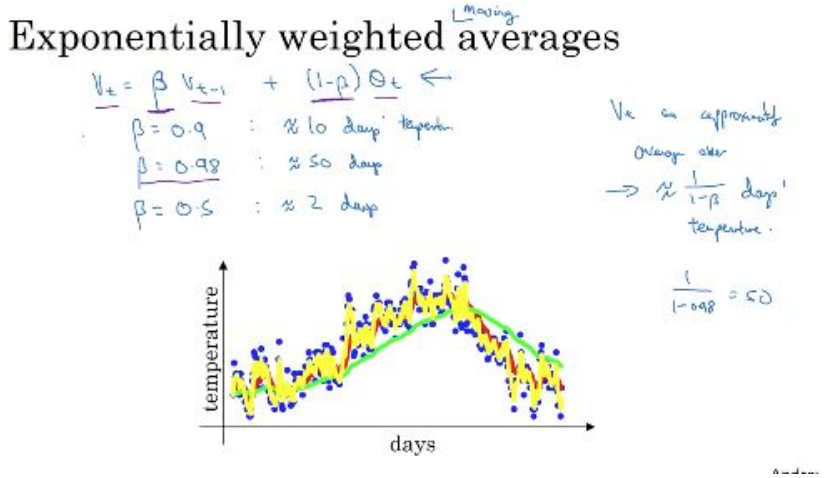


*Exponentially Weighted Averages*

- Now we will take a look at other optimization algorithms that are faster than gradient descent. In order to understand these algorithms we need to understand something called exponentially weighted averages. We'll illustrate it using the example below. We have the temp in London for one year. When we plot it we have something like the blue dots. We then do the following: set $v_0 = 0,\ v_1 = 0.9 * v_0 + 0.1 * \theta_1$ where $v_0$ is the previous day v and $\theta_1$ is the temperature that day. The general formula is then $v_t = 0.9 * v_{t-1} + 0.1 * \theta_t$. If we plot that we then end up with the red line below, a less noisier line. This is what is called an exponentially weighted average of the daily temperature.



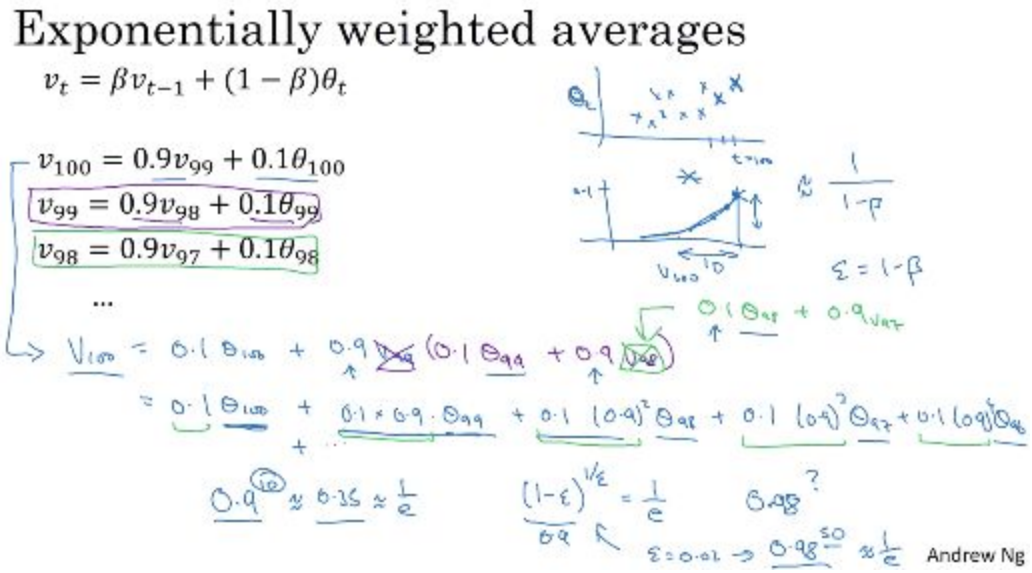

- The general formula is then $v_t = \beta * v_{t-1} + (1-\beta) * \theta_t$. We can also think of $v_t$ as approximately averaging over $1/(1-\beta)$ days' temperature. If $\beta = 0.9$ we are averaging over $\approx 10\ days'\ temp$. If $\beta = 0.98$ we are averaging over $\approx 50\ days'\ temp$ and our plot is the green line below. If $\beta = 0.5$ we are averaging over $\approx 2\ days'\ temp$ and our plot is the yellow one below,




Averaging over a small number of days adapts fast to a new day temp but is very noisy. Averaging over a big number of days gives us a smoother curve but gives less weight to a new day's temp.

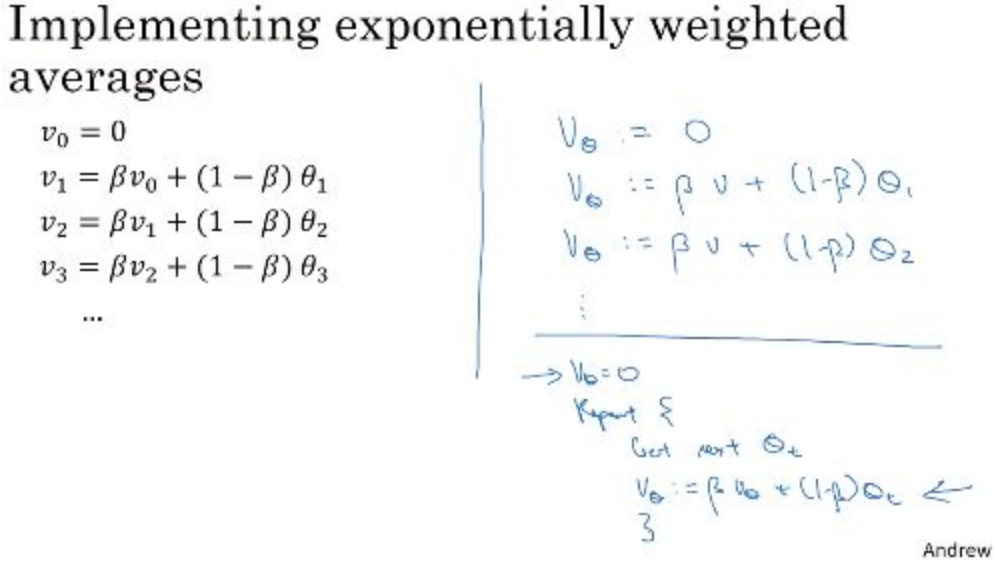
## *Understanding Exponentially Weighted Averages*

- Let's go a little deeper into how exponentially weighted averages work. Remember our equation is $v_t = \beta * v_{t-1} + (1-\beta) * \theta_t$. If we take that equation and plug in the values from day 100 we see the equations on the left side below. We can then do some algebra using beta =0.9, we know $v_{100} = 0.1 * \theta_{100} + 0.9 * v_{99}$ but we can substitute $v_{99}$ with its equation so we then have $v_{100} = 0.1 * \theta_{100} + 0.9 * (0.1 * \theta_{99} + 0.9 * v_{98}$. We can then substitute $v_{98}$ with its equation and so on. This equation ends up as a weighted sum/average of the day's temp + and the previous day temp and the previous day temp and so on.




We can draw this equation on the top right corner and we end up with an exponentially decaying function. How many days is this equation averaging over? It turns out that $0.9^{10} \approx 0.35 \approx 1/e$. This means it takes about 10 days for the height of this to decay ⅓ so we say we are computing the average that focuses on the last 10 days because after 10 days the weight decays to less than about a ⅓ of the weight of the current day.

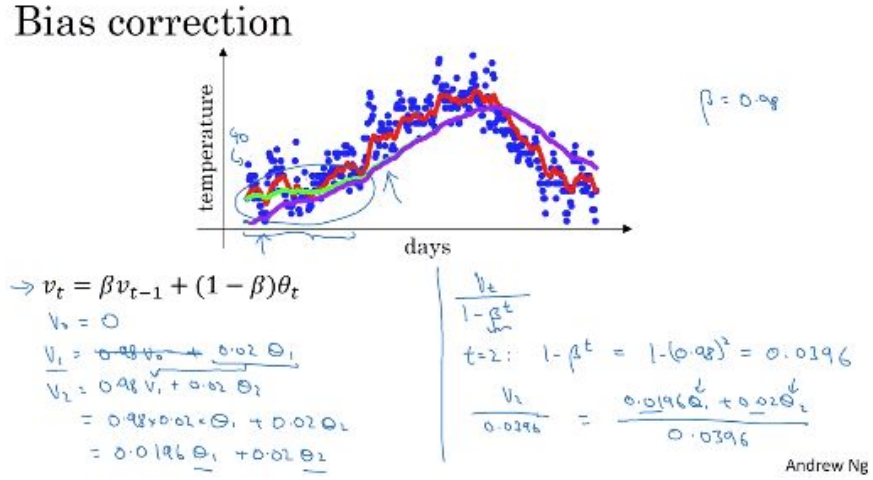
- How do we actually implement exponentially weighted averages? What we do in practice is shown on the right side below. We initialize $v_\theta = 0$ then update it to $v_\theta = \beta v + (1-\beta) * \theta_1$ and so on. We can do the same using a for loop.




Exponentially weighted averages is one line of code and very efficient from a computation and memory point of view which is one reason why it is commonly used in ML.

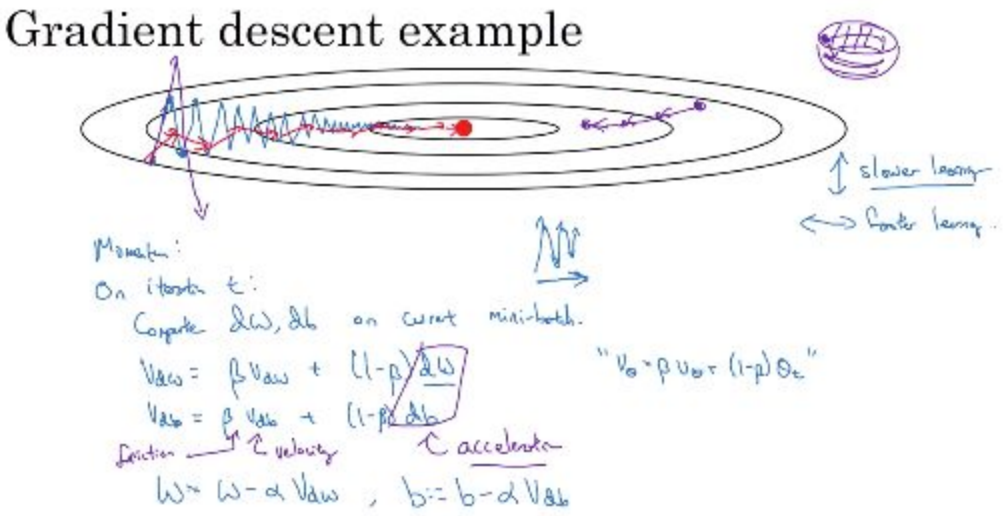
*Bias Correction in Exponentially Weighted Averages*

- In the last session we saw how different beta values give us different curves as shown below. It turns out we do not obtain the green line below, we get the purple line. Notice how this line is defaced to the right. The reason for this is shown on the bottom left corner: the computation for day 1, $\theta_1$, is a low value and not a good average; we get the same case for day 2. We correct this with bias correction. What we do is the following: $v_t/(1-\beta^t)$. If t-=2 then we have $1-\beta^t = (1-0.98)^2 = 0.0396$ so $v_t/0.0396$ will give us a better average.

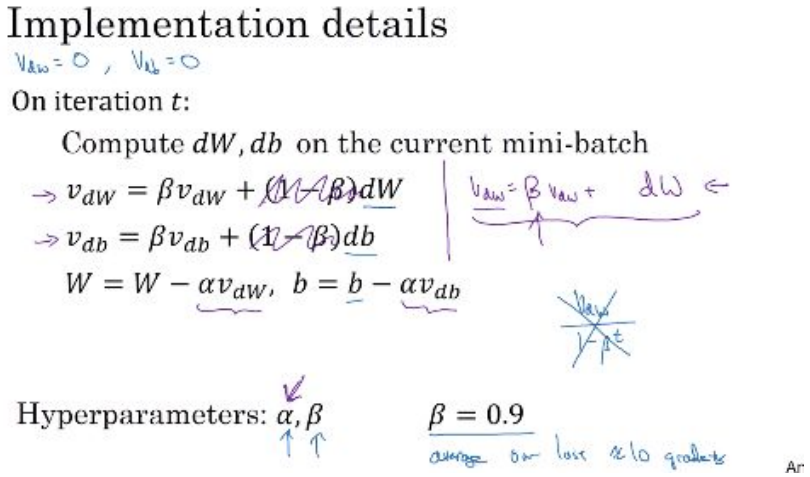



*Gradient Descent with Momentum*

- We have an algorithm called gradient descent with momentum that works faster than normal gradient descent. The basic idea is to compute an exponentially weighted average of the gradients and use that instead. Let's see how it works: gradient descent usually has a somewhat erratic way to get to the global minimum as shown below. What we want with gradient descent is slower learning vertically and faster learning horizontally; in other words, a smoother and faster path to the global minimum. Momentum provides this in the following manner:
  - On each iteration t, we compute $dw \; and \; db$ as always
  - We then compute $v_{dw} = \beta * v_{dw} + (1-\beta)dw$ and $v_{db} = \beta * v_{db} + (1-\beta)db$
  - We then update our weights using $v_{dw}$ and $v_{db}$: $w = w - \alpha * v_{dw}, \; b = b - \alpha v_{db}$




- Let's look at a couple of implementation details: below are the steps needed to do gradient descent with momentum. In practice $\beta = 0.9$ is a good value to use and it means that we are averaging over the last 10 gradients. Notice we know have an additional hyperparameter, $\beta$, which controls the exponentially weighted average.

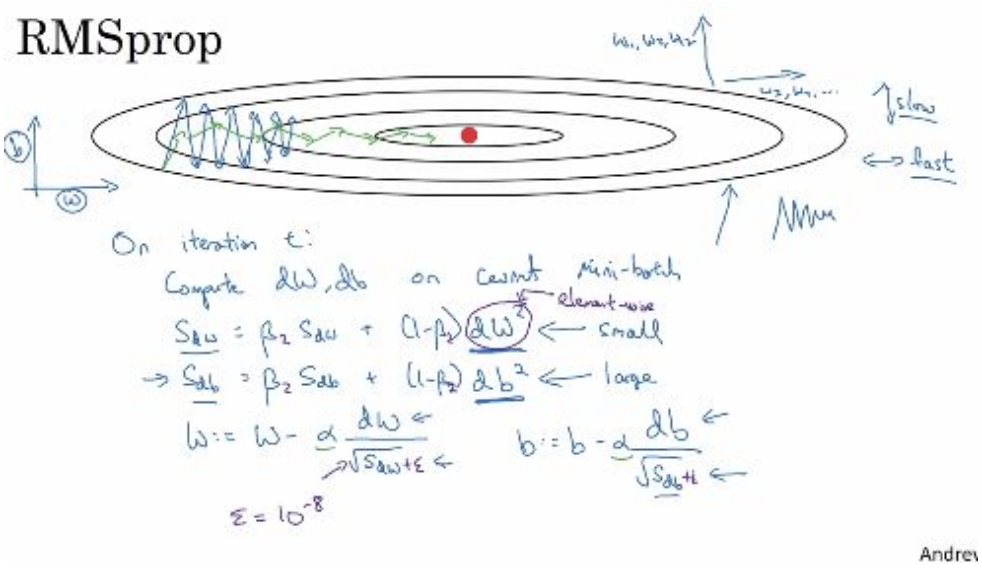



Often we will see vdw as $v_{dw} = \beta v_{dw} + dw$. This also works fine but Professor Ng prefers the other version because this one is less intuitive. In practice we also see people not bothering with bias correction.

## *RMS Prop*

- There is another algorithm that can speed up gradient descent: Root Mean Square Prop or RMSProp. Similar to Momentum, RMS prop smooths out the gradient descent steps by trying to minimize the vertical learning and maximizing the horizontal learning. The steps are:
    - On each iteration t, compute $dw$ *and* $db$ as always

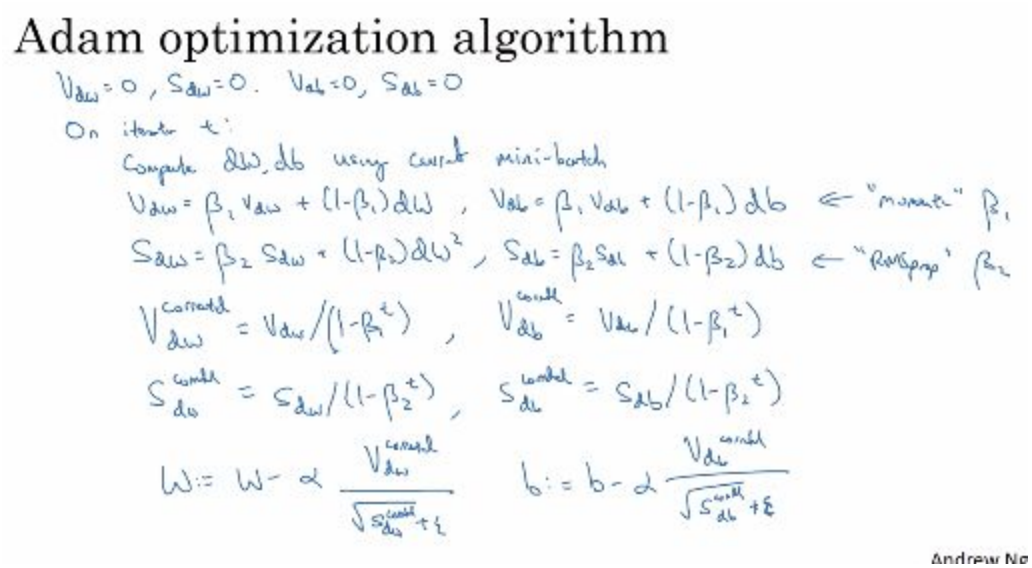
- We then compute $s_{dw} = \beta * s_{dw} + (1-\beta)dw^2$ and $s_{db} = \beta * s_{db} + (1-\beta)db^2$
- We then update our gradients as follows:
  $w = w - \alpha * (dw/\sqrt{s_{dw}})$ *and* $b = b - \alpha * (db/\sqrt{s_{db}})$



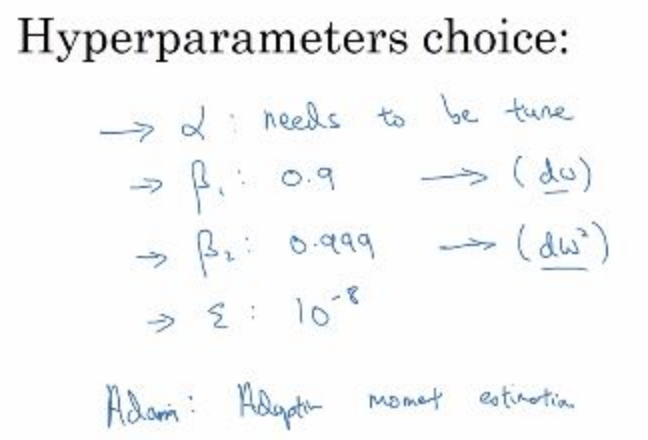

The idea is that dw is a small value and db is a large value so when we update the gradients using those values we get the smooth steps we are looking for.

### *Adam Optimization Algorithm*

- Adam Optimization basically combines Momentum & RMS Prop. The calculations are:
  - For iteration t we compute $dw$ *and* $db$ as always
  - We then compute
    $v_{dw} = \beta_1 * v_{dw} + (1-\beta)dw,\ v_{db} = \beta_1 * v_{db} + (1-\beta)db,\ s_{dw} = \beta_2 * s_{dw} + (1-\beta_2)dw^2$
    and $s_{db} = \beta_2 * s_{db} + (1-\beta_2)db^2$
  - Here we implement bias correction so we end up with
    $v_{dw}^{corrected} = v_{dw}/(1-\beta_1^t),\ v_{db}^{corrected} = v_{db}/(1-\beta_1^t),\ s_{dw}^{corrected} = s_{dw}/(1-\beta_2^t),\ s_{db}^{corrected} = s_{db}/(1-\beta_2^t)$
  - We then update our gradients as follows: $w = w - \alpha * (v_{dw}^{corrected}/\sqrt{s_{dw}^{corrected} + \varepsilon})$ and
    $b = b - \alpha * (v_{db}^{corrected}/\sqrt{s_{db}^{corrected} + \varepsilon})$




- This algorithm has a few hyperparameters to use:
  - $\alpha$ : *needs to be tuned*
  - $\beta_1$ : 0.9
  - $\beta_2$ : 0.999
  - $\varepsilon$ : $10^{-8}$

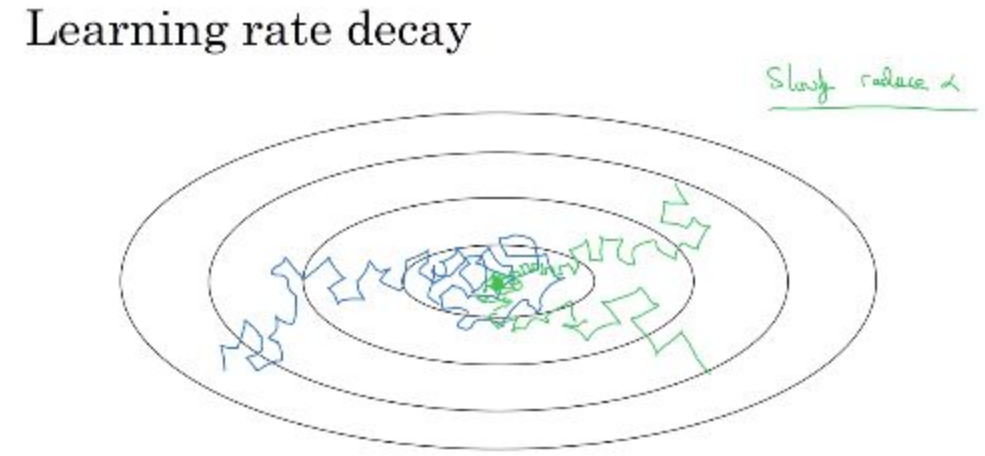



Adam, by the way, stands for Adaptive moment estimation.

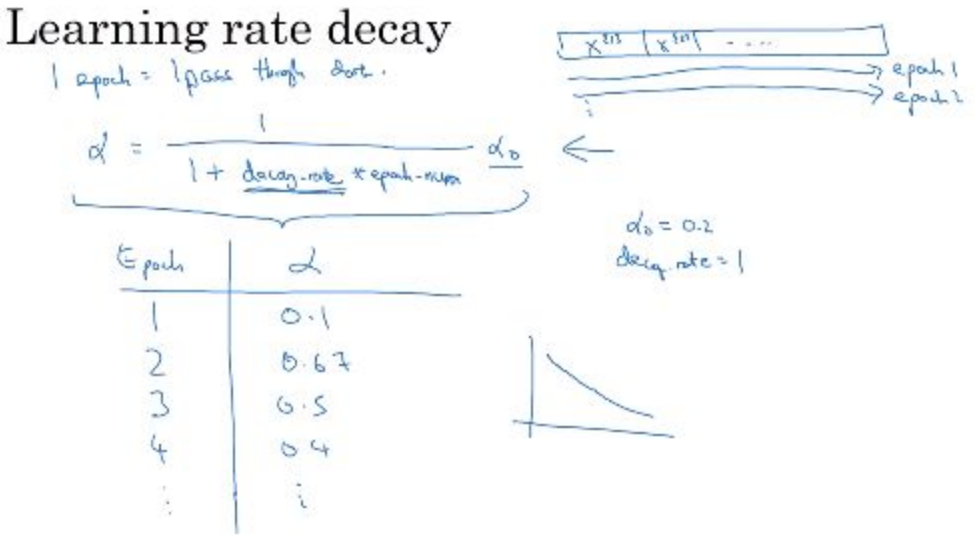
### *Learning Rate Decay*

- One of the things that can help our learning algorithm is to slow down the learning rate over time, why? Consider the example below. When doing mini batch gradient descent, we have a noisy way to the global minimum and we never converge. By slowly reducing the learning rate we can have somewhat big steps at the beginning, as shown in the

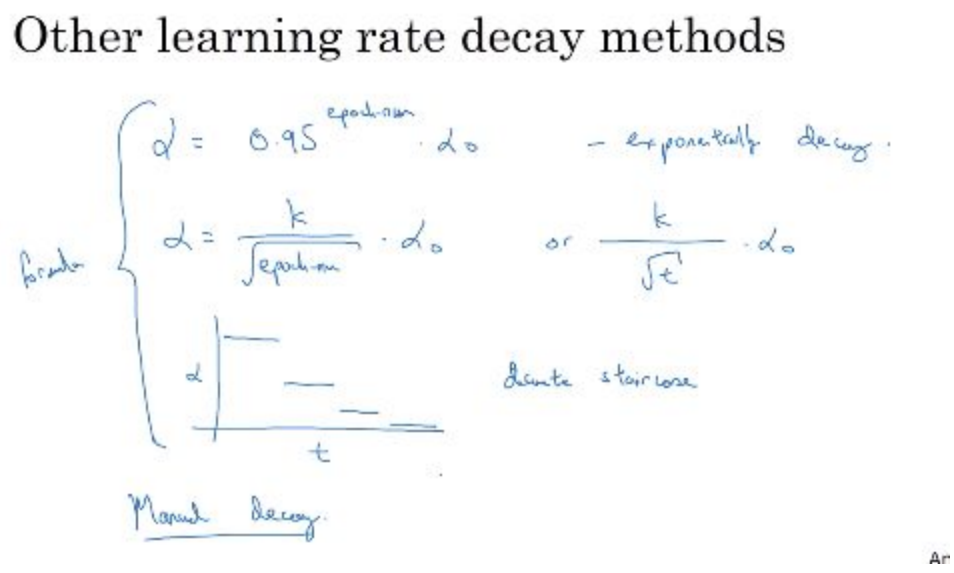
green trajectory, and smaller ones at the end so our algorithm is tighter and 'hangs out' closer to the global minimum.




- How do we implement learning rate decay? Remember that 1 epoch = 1 pass through the training set. What we do is we set $\alpha = 1/(1 + decay\ rate * epoch\ number)$. In this way we get a learning rate that decreases as we go through the epochs, which is exactly what we want.

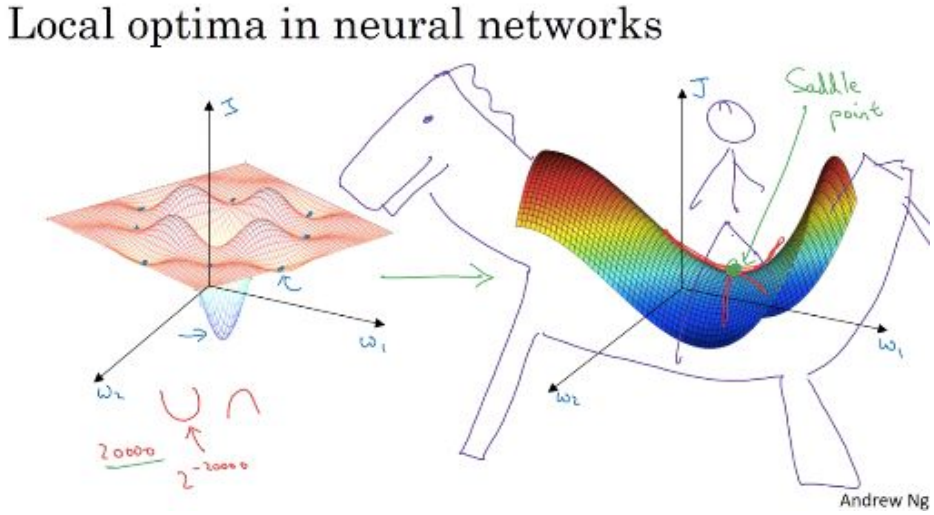



- There are other ways of calculating the learning rate decay: other formulas and manually. Other formulas are: $\alpha = 0.95^{epoch\ number} * \alpha$, $\alpha = (k/\sqrt{epoch\ number}) * \alpha$ or a staircase where we cut alpha in half after a number of steps. Manual decay is done when you have a small number of models that take days to train and you are observing them and changing the learning rate as appropriate (i.e. the learning rate has slowed down, let's cut the value of alpha).



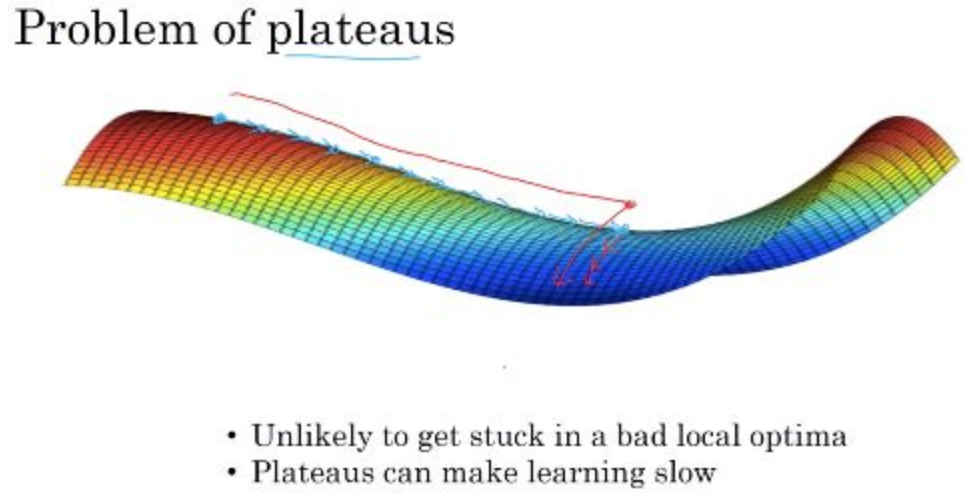

*The problem of local optima*

- In the early days of deep learning there was a lot of worry about local optima and optimization algorithms getting stuck there. As deep learning has advanced this is less of a concern. The picture on the left is what we had in mind when we were afraid of getting stuck in local optima. However, the picture on the right is a better representation of what happens with neural networks. In such a case it is more likely you encounter a 'saddle' point (where the derivative = 0) rather than a local optima. Neural networks usually have many, many parameters so the chances of getting a local optima are very small. The lesson is then that a lot of our intuitions about low dimensional spaces do not translate to high dimensional situations. Therefore, local optima is less of a problem.




- If local optima is not a problem, what is a problem? Plateaus. Plateaus can slow down learning and are regions where the derivative is close to 0 for a long time. As shown

below, an optimization algorithm might take a while to get off the plateau and continue to go down towards the global minimum. More sophisticated algorithms such as Adam or RMSprop help us to go quickly through the plateau.




Summarizing: local optima is not a problem in deep learning and plateaus can make learning slow.

*Summary from programming exercise: Gradient Descent*

- The difference between gradient descent, mini batch gradient descent and stochastic gradient descent is the number of examples used in each step.
- You have to tune $\alpha$.
- A well-tuned mini batch size usually outperforms gradient descent and stochastic gradient descent, especially when the training set is large.

*Summary from programming exercise: Mini-Batch Gradient Descent*

- Shuffling & Partitioning are two steps required to create mini batches.
- Mini batch sizes are usually powers of two: 64, 128, 256, 512.

*Summary from programming exercise: Momentum*

- Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied to all 3 gradient descent types.
- With momentum you have to tune $\beta$ *and* $\alpha$.

*Summary from programming exercise: Momentum*

- Some advantages of Adam are relatively memory requirements & that it usually works well even with little tuning of hyperparameters (except $\alpha$).