CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

TensorFlow

Machine Learning

Machine Learning

Lecture: TensorFlow

Ted Scully

Optimization

- So far we have considered a vectorised solution for neural networks where we push **all of our training data** through the network at one time.

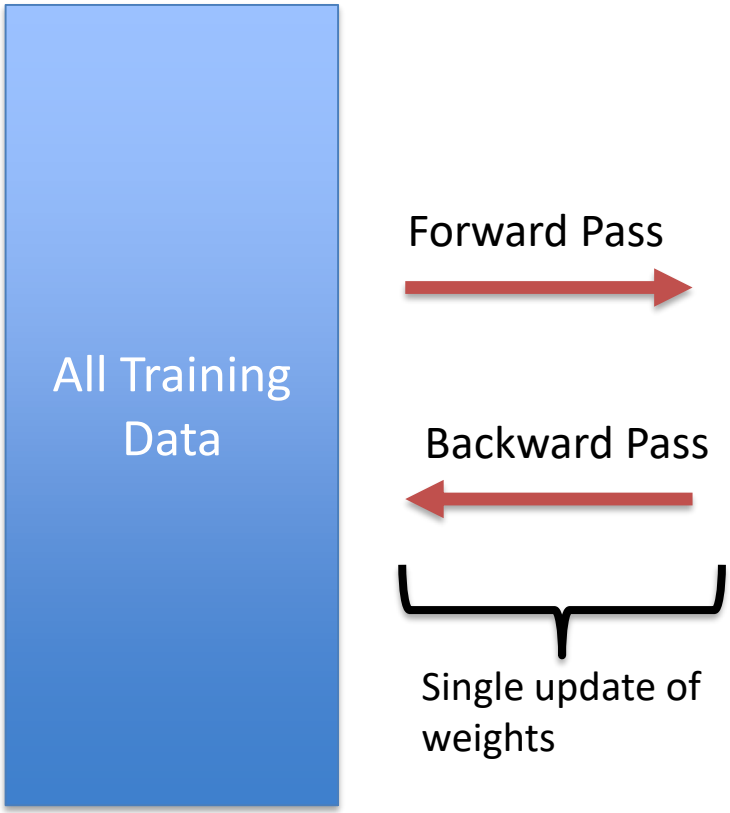
- Also we have only considered updating the weights using our **standard gradient descent** update rule.

- Over the next few slides we will look at widely used variants around both the **forward pass process** and the **update of weights** in the backward pass.

- **Mini-batch gradient descent.**
- Learning Rate Decay
- Adaptive Learning Rates

Batch v's Mini-Batch Gradient Descent

- Consider a situation where you have a very large number of train examples (**4,000,000**).
- The issue that arises in such scenarios is that we must push all 4M data points through our graph in order to determine a single update for our weights.
- This can take a very long time and can mean that training the model takes a very very long time. Given that the process of gradient descent machine learning is so iterative and we try **many different model configurations** this represents a serious problem.
- This approach is often referred to as **batch processing** (we take the entire batch (training set) for each update of our weights)



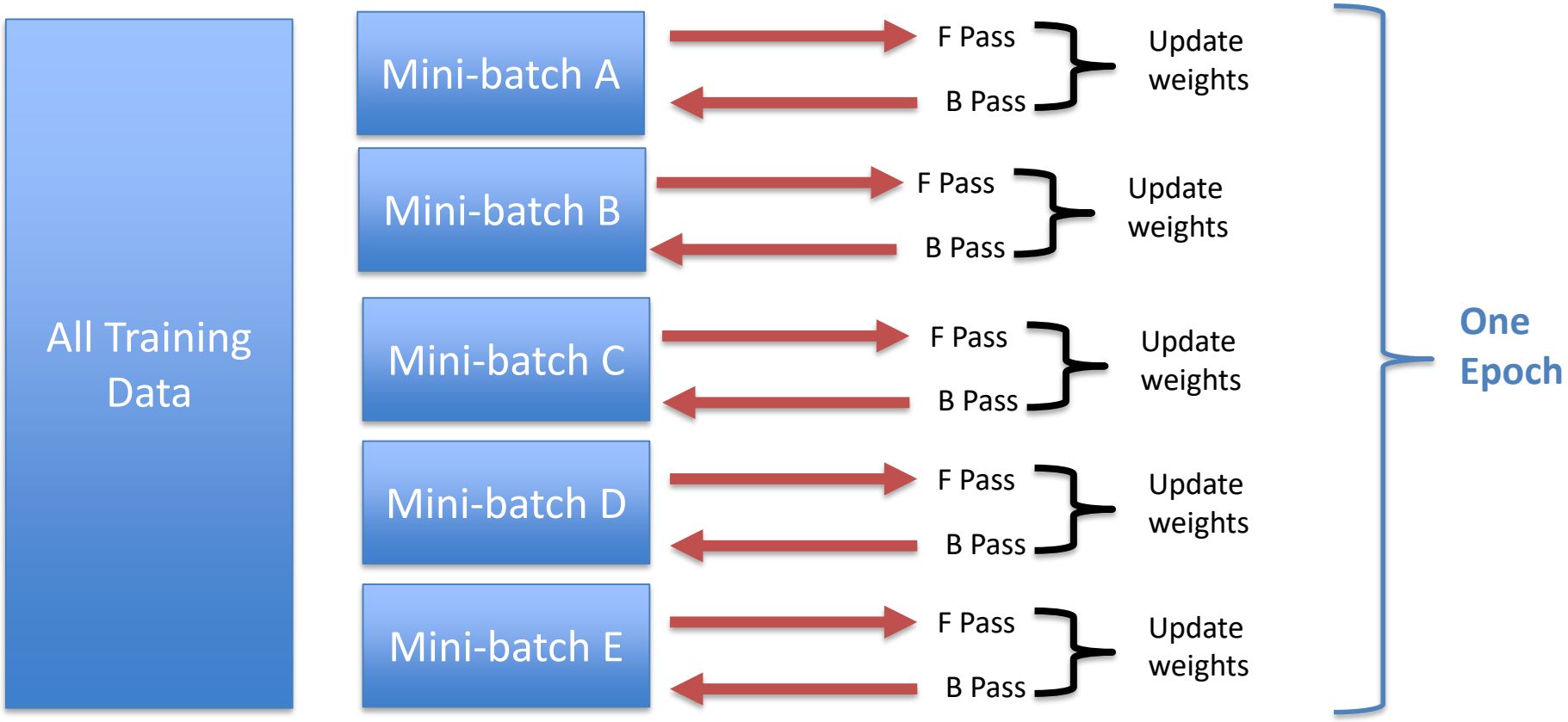
Batch v's Mini-Batch Gradient Descent

- An alternative approach that significantly alleviates the problem outlined in the previous slide is to **break-up the training data** into exclusive partitions, referred to as **mini-batches**.

- For example, we might break our **4M** training examples in **8000 mini-batches** with **500 training instances** in each mini-batch.

- Subsequently, we take the first mini-batch and complete a single forward and backward pass and use it to update the weights once. We then take the next mini-batch and complete a single forward and backward pass and use it to update the weights once. We continue this process until we have processed all 8000 mini-batches.
- At this stage we have completed **one epoch** but the weights have been updated 8000 times (as opposed to just once with batch processing). All training examples have been pushed through the model once. We may then have many epochs before we reach a convergence.

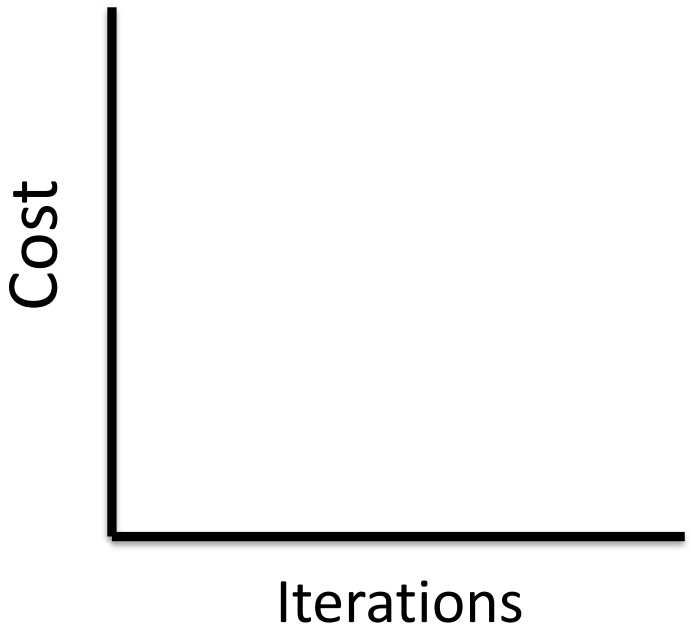
Batch v's Mini-Batch Gradient Descent



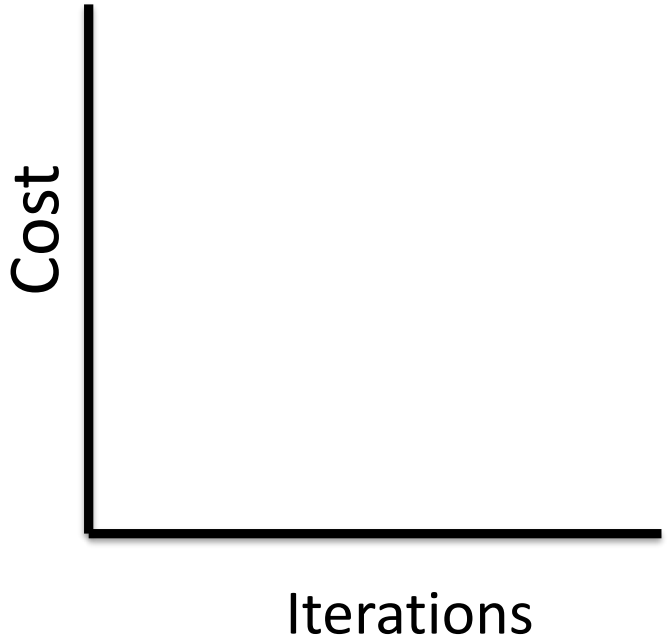
- The main difference is that when using mini-batch, after one epoch the weights will have been updated mini-batch number of times … in our example the weights would have been updated 8000 times.

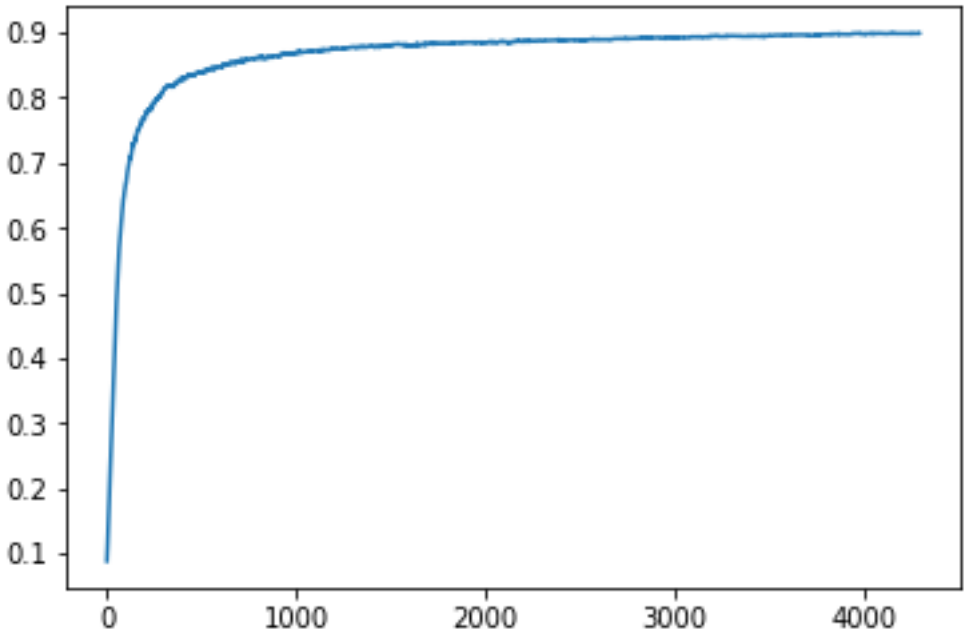
- However, after a single epoch with batch gradient descent the weight have only been updated once.

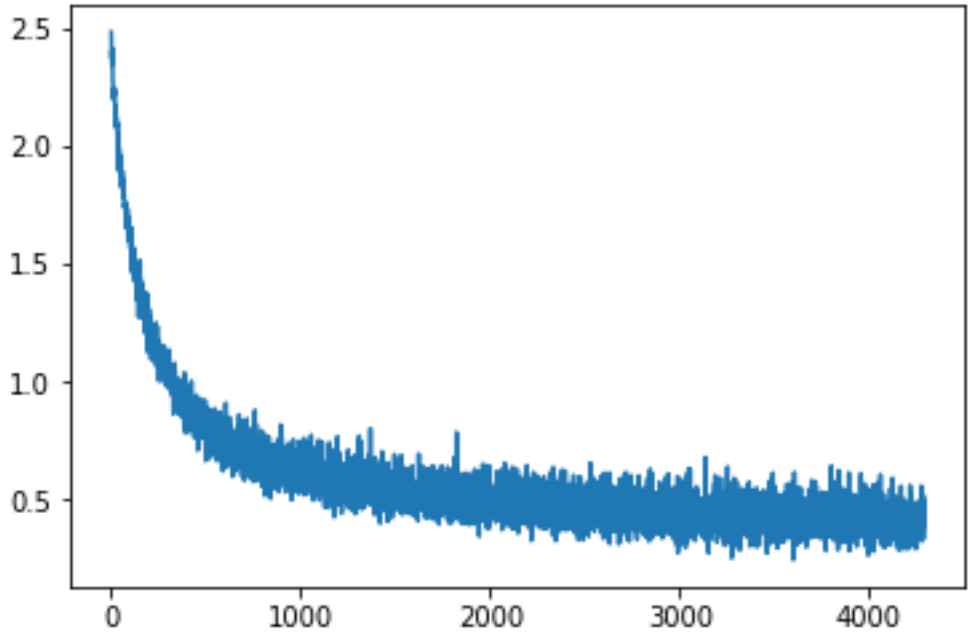
Batch GD



Mini-Batch GD




0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1000
2000
3000
4000


2.5
2.0
1.5
1.0
0.5
0
1000
2000
3000
4000

Selecting a size for your Mini-Batch

- If your batch size is the same as your training set size then you are using **batch GD**.
  - You would typically only use batch GD when you have a small training set (typically $m < 4000$)
  - We have already outlined the problem with this option. **Too slow for large training sets**.

Size of Mini-Batch

- If the ***batch size*** **= 1** we are using what is called **stochastic GD**.
  - The drawback with this option is it can be **very noisy**. Each step will on average take you closer to the minimum but some will also step in the opposite direction.
  - An additional consequence of the noisy learning process is that stochastic gradient descent can find it **difficult to converge on an true minimum** for a specific problem and may end up circling the value (but will obtain a good approximation).
  - The other problem with stochastic GS is that you **lose the speed advantage** provided by vectorization because you are just processing one training example at a time.

Size of Mini-Batch

- If ***batch size*** **> 1 and < m** then we are using what is called **mini-batch gradient descent**.
  - Typically sizes for mini-batch is 32, 64, 128, 256, 512, 1024.
  - The advantage of this technique is that we can achieve a relatively high frequency of weight updates while still retain the benefit of **vectorised speed** up advantages.
  - The behaviour of the loss function is still **noisy** although not as noisy as pure stochastic GD. The larger the batch size the less noisy the behaviour.

Optimization

- We have only considered updating the weights using our standard gradient descent update rule.

- Mini-batch gradient descent.
- **Learning Rate Decay**
- Adaptive Learning Rates

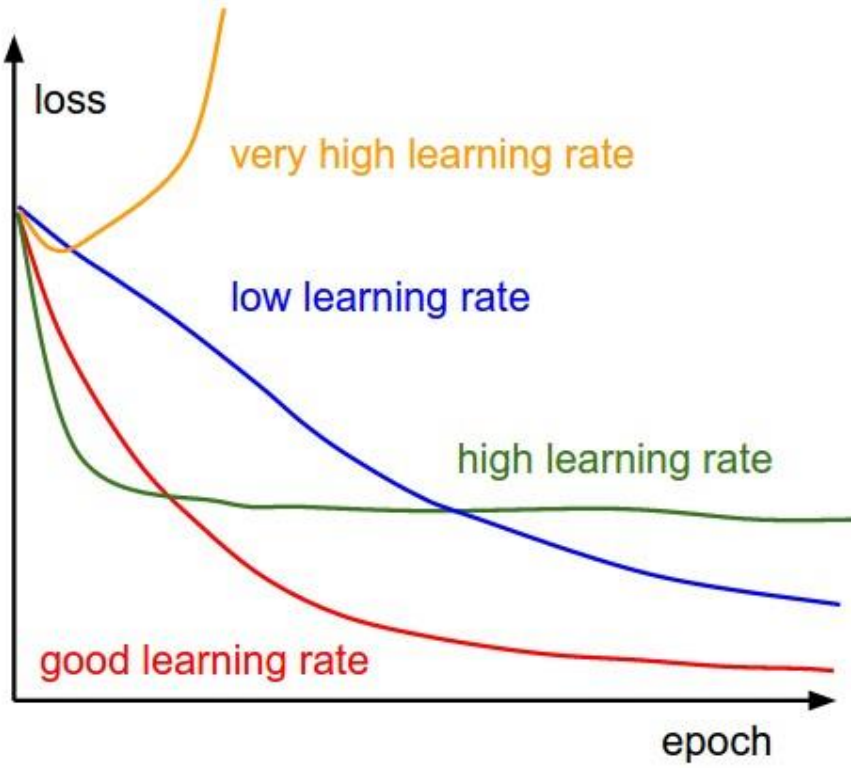
repeat

$$\lambda_1 = \lambda_1 - \alpha\,(d\lambda_1)$$
$$\lambda_2 = \lambda_2 - \alpha\,(d\lambda_2)$$
$$b = b - \alpha\,(db)$$

Learning Rate Decay

- Learning Rate Decay is a simple concept that **gradually reduces the learning rate (alpha)** as we iterate through gradient descent.

- High learning rates may reduce the loss quickly, but they can often get stuck at values of loss (green line). They end up jumping around the minimum rather than converging properly.



- The idea behind learning rate decay is that we want to **initially move quickly in the direction of the true minimum** but as we approach we want to **gradually reduce the learning rate** to ensure smaller steps and better convergence.

Learning Rate Decay – Step Decay

- There are a range of methods that can be adopted for implementing learning rate decay.

- One of the simplest and most effective is called **step decay**.
- In step decay you **reduce the learning rate by some factor** every few epochs.
- For example, you might **multiply the learning rate by 0.5 after each five epochs**. Please note that the appropriate value can vary significantly from one problem to another.

- Another, more manual, approach is to run your model initially with a fixed learning rate. Observe the epoch where the validation loss **stops reducing**.
- You can then apply a decay of the learning rate at this epoch. In other words you can reduce the learning rate by some value (e.g. 0.5) when the validation stops decreasing.

Learning Rate Decay

- Another common technique is an exponential rate decay.
- For each epoch we update our learning rate according to the following rule:
- $\alpha_{epoch} = \alpha_0 \, e^{-epoch}$

Epoch	α_{epoch}
0	0.5
1	0.18
2	0.06
3	0.02
4	0.009
5	0.003

In this example α_0 = 0.5

As you can see the learning rate decreases **rapidly**. This rate of decay can be adjusted by adding an additional hyper-parameter h to the equation.

$$\alpha_{epoch} = \alpha_0 \, e^{-epoch*h}$$

Optimization

- So far we have considered a vectorised solution for neural networks where we push all of our training data through the network at one time.
- Also we have only considered updating the weights using our standard gradient descent update rule.

- Over the next few slides we will look at widely used variants around both the forward pass process and the update of weights in the backward pass.

- Mini-batch gradient descent.
- Learning Rate Decay
- **Adaptive Learning Rates**

Adaptive Learning Rates

- You may notice when we use learning rate decay we adjust the learning rate. However, that **single adjusted learning rate** is used to update the learning rate for **all training parameters**.
- A preferable approach could be to employ an adaptive rate of update that is specific to each of the training parameters.
- There has been a very significant amount of research work invested in the area of adaptive learning rates over the last number of years.

repeat

$$\lambda_1 = \lambda_1 - \alpha\,(d\lambda_1)$$
$$\lambda_2 = \lambda_2 - \alpha\,(d\lambda_2)$$
$$b = b - \alpha\,(db)$$

Exponentially Moving Average

- To understand the operation of the majority of algorithms that employ adaptive learning you first need to grasp the idea of exponentially weighted moving average (also called exponentially weighted moving average (EMA)).
- EMA allows you to track the moving average of a time series over time.
- Assume we have a continuous sequence of time series data **Y** containing **n data points**.
- **EMA is a weighted average of these n data points, where the weighting applied to the data points decreases exponentially over time**.
- That is EMA ensures that more recent data points obtain a higher waiting.

Exponentially Moving Average

- Assume we have a continuous sequence of time series data **Y** containing **n data points**.

- The equation for EMA for the series data Y (Y^1, Y^2 ..) is as follows:

- $$E^t = \begin{cases} Y^1 & when\ \ t = 1 \\ ((1-\beta) * Y^t) + (\beta * E^{t-1}) & when\ t > 1 \end{cases}$$

- What is the impact of a very high value of β

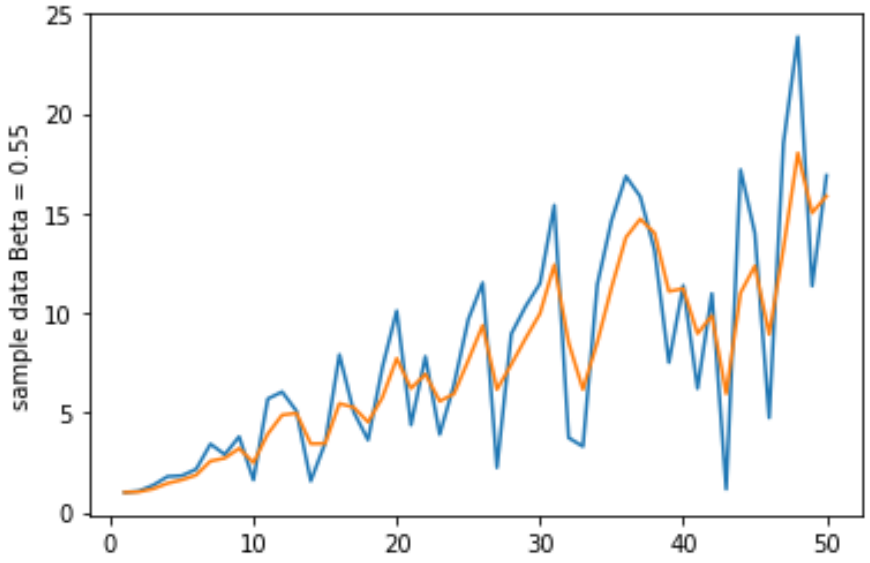
Exponentially Moving Average

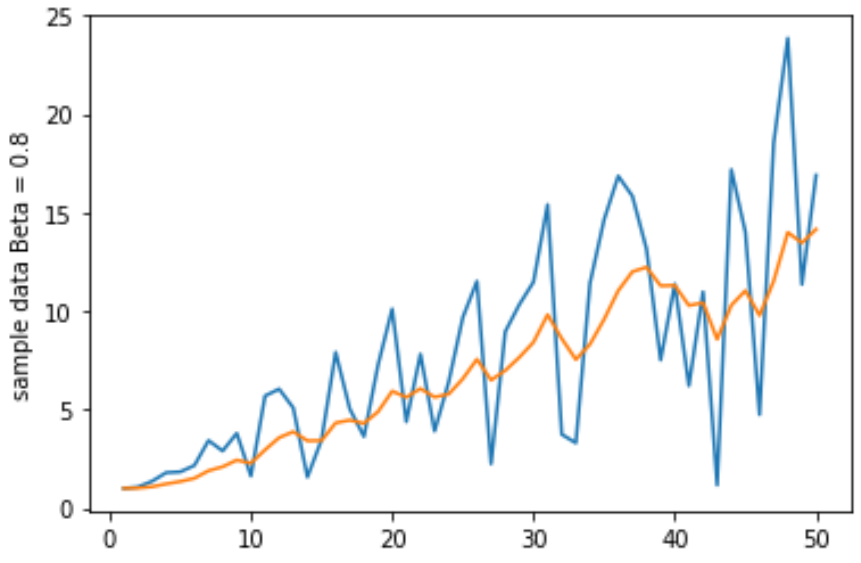
- Assume we have a continuous sequence of time series data **Y** containing **n data points**.

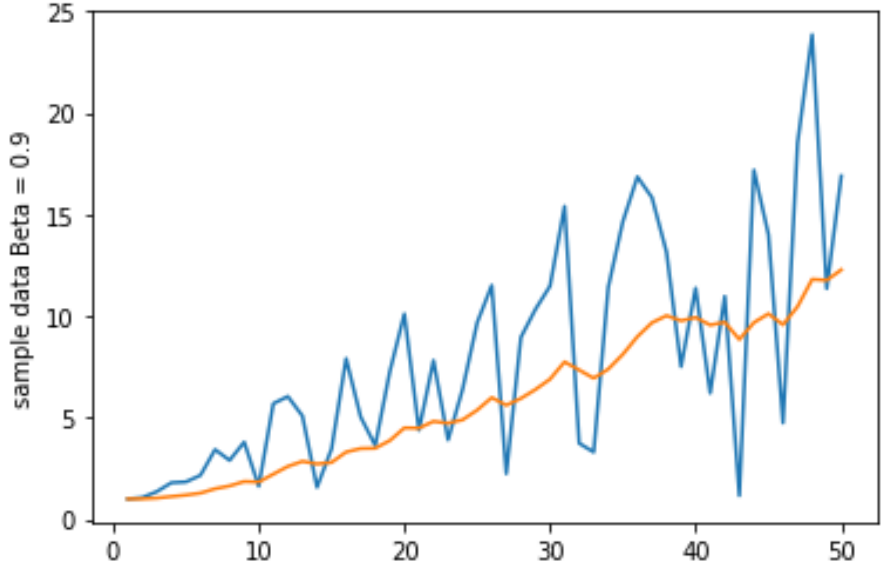
- The equation for EMA for the series data Y (Y^1, Y^2 ..) is as follows:

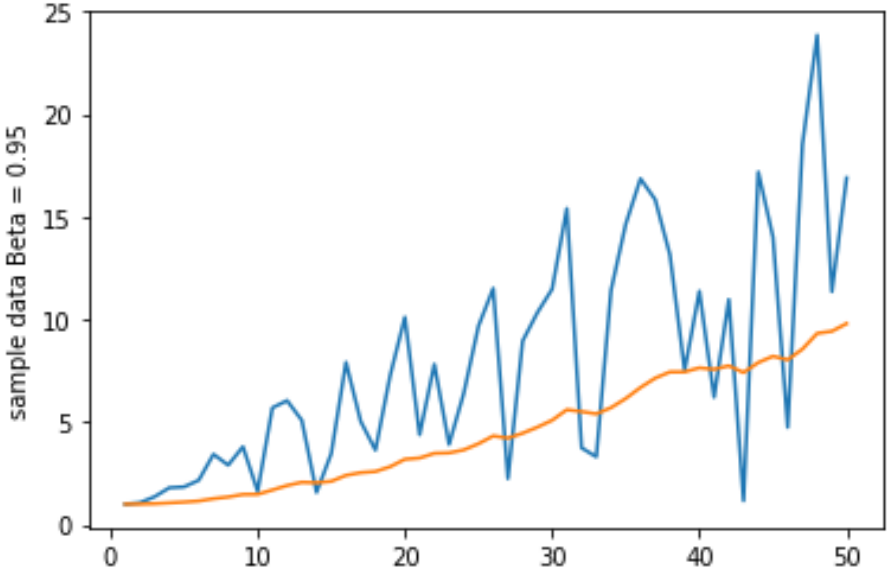
- $$E^t = \begin{cases} Y^1 & when\ \ t = 1 \\ ((1-\beta) * Y^t) + (\beta * E^{t-1}) & when\ t > 1 \end{cases}$$

- The **higher** the value of the variable $\boldsymbol{\beta}$ the slower the old observations are discounted and less emphasis put on the new observation.


sample data Beta = 0.55
0
5
10
15
20
25
0
10
20
30
40
50


sample data Beta = 0.8
0
5
10
15
20
25
0
10
20
30
40
50


sample data Beta = 0.9
0
5
10
15
20
25
0
10
20
30
40
50


sample data Beta = 0.95
0
5
10
15
20
25
0
10
20
30
40
50

Exponentially Moving Average

- One of the benefits of using the exponentially moving average is that we don't keep track of each average value of time. Instead we just keep track of **one variable** the current average E. Therefore, when using it in ML it becomes

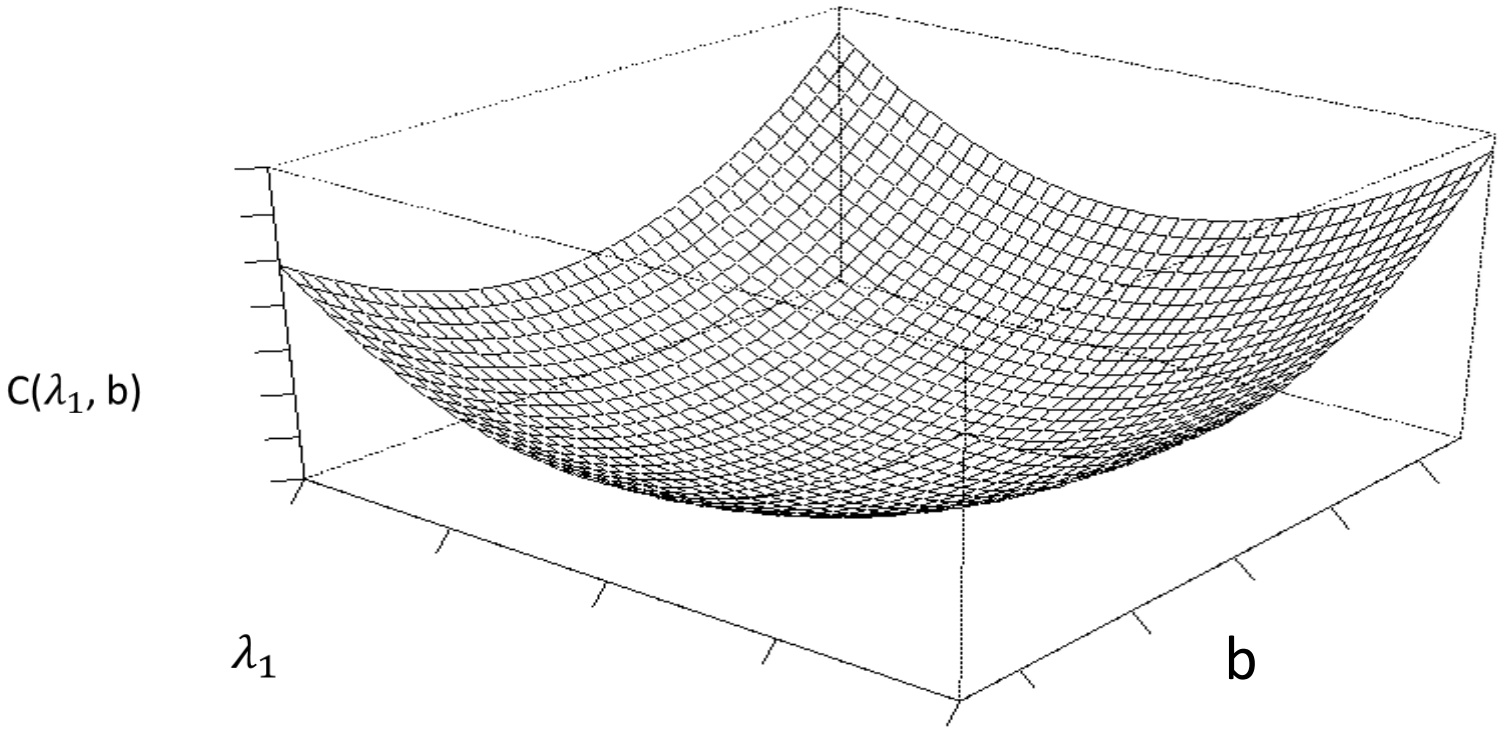
- $$E = \begin{cases} Y^1 & when\ \ t = 1 \\ ((1-\beta) * Yt) + (\beta * E) & when\ t > 1 \end{cases}$$

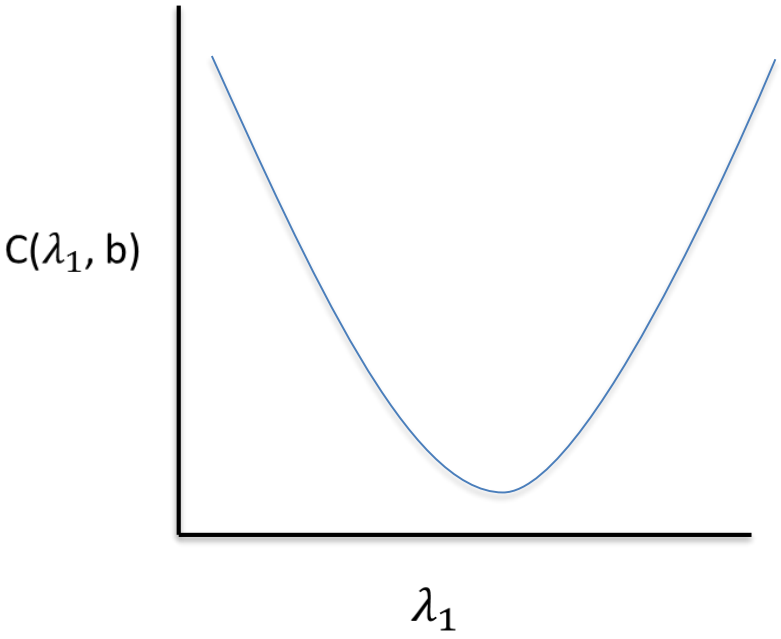
- This is one of the benefits of using EMA. It has a **low memory footprint** (and is computationally inexpensive).

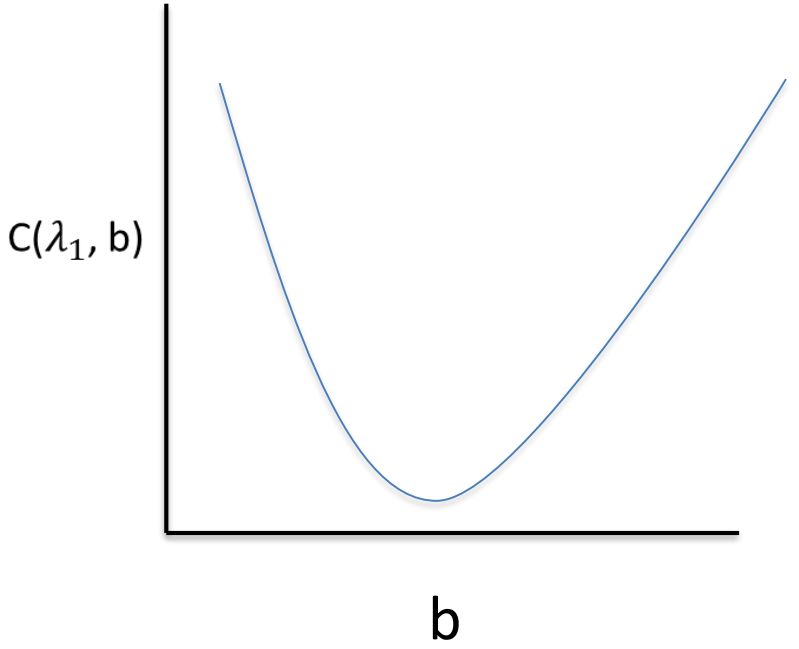
- One of the benefits of EMA that you see in the previous slide is that is can **smoothen out large oscillations** in the value of a variable.

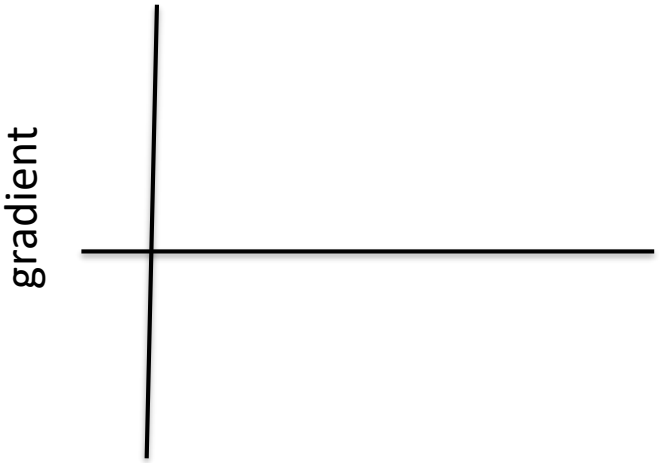
Adaptive Learning Rates

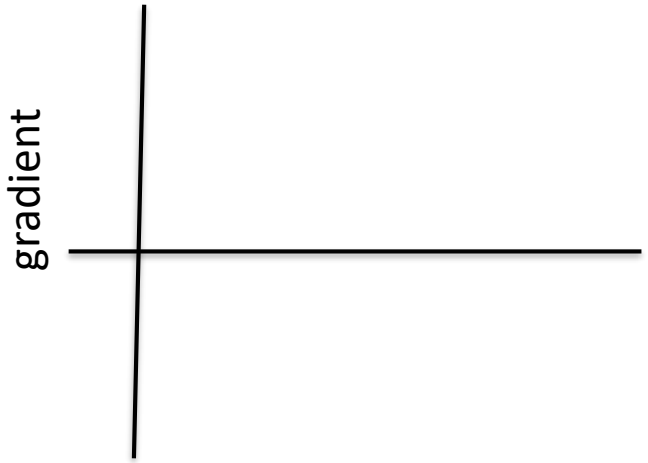
- You may notice when we use learning rate decay we adjust the learning rate. However, that **single adjusted learning rate** is used to update the learning rate for **all training parameters**.
- A preferable approach could be to employ an adaptive rate of update that is specific to each of the training parameters.
- There has been a very significant amount of research work invested in the area of adaptive learning rates over the last number of years.




C(λ_1, b)
λ_1


C(λ_1, b)
b


gradient


gradient

Gradient Descent with Momentum

- The idea behind gradient descent with momentum is to **calculate an exponential moving average of your gradients for each trainable parameter** and use that average to update your weights.
- So let's look at it for the simple problem where just have two learnable parameters our bias b and a single coefficient λ_1.

repeat

$$\lambda_1 = \lambda_1 - \alpha\,(d\lambda_1)$$

$$b = b - \alpha\,(db)$$

Gradient Descent with Momentum

- Notice in the code we maintain an **EMA for the gradients for both trainable parameters**. It is this value that now becomes core to the update of these parameters.
- Notice if there is a **high level of oscillation for one parameter** such as bias then it will **smoothen out this value** and slow down the rate of update.
- In turn this can allow us to use a large learning rate and obtain convergence faster.

repeat

$$E_{\lambda_1} = \beta E_{\lambda_1} + (1-\beta) d\lambda_1$$

$$E_b = \beta E_b + (1-\beta) db$$

$$\lambda_1 = \lambda_1 - \alpha (E_{\lambda_1})$$

$$b = b - \alpha (E_b)$$