**Week 2 – Notes**

**Optimization Algorithms**

**Mini-batch Gradient Descent**

If there are too many training examples, a cost function update is made extremely slow

The idea is to use instead of batch gradient descent (the gradients being updated only after learning all the examples) a mini-batch gradient descent; thus you update the gradients after learning only a part of the training examples

A picture containing handwriting, font, typography, calligraphy

Description automatically generated - notation for batch t

Mini-batch -> perform one step of gradient descent for each batch (forward prop, cost computation and back prop)

One epoch means a pass through training set, so if you use the mini-batch, you have to pass through each mini-batch

**Understanding Mini-batch Gradient Descent**

When using batch gradient descent, the cost function decreases monotonically, but when the mini-batch gradient descent is used, the cost oscillates because some mini-batches are “harder” to learn

Batch gradient descent: (mini-batch size = m) -> takes a lot of time for each iteration, but it converges to the local minima smoothly; it can be used for small training sets (<= 2000 examples)

Stochastic gradient descent: (mini-batch size = 1) -> you loose the vectorization speed, for many iterations it can go towards wrong directions and it wanders around the local optimum

Mini-batch gradient descent: (the in-between solution) -> faster learning, uses the vectorization, however it may go towards not the best directions, but in the end it will be very close to the local optimum; typical mini-batch sizes are 64, 128, 256, 512 and rarely 1024 (all of them are powers of 2 for optimization)

Pay attention that the mini-batch fits in the CPU/GPU memory

**Exponentially Weighted Averages**

A technique to compute the moving average is to use exponentially weighted moving average that for each step computes:

A picture containing handwriting, font, calligraphy, line

Description automatically generated, where beta is a parameter and theta t is the current point

This method takes into consideration less and less the oldest data points

Vt is approximatively an average over 1 / (1 – beta) data points

For examples if beta = 0.9 -> ~ 10 days; beta 0.98 -> ~ 50 days; beta 0.5 -> ~ 0.5 days

**Understanding Exponentially Weighted Averages**

You can think of this algorithm as you multiply the values with an exponential function, so that older values matter less

A picture containing handwriting, text, font, line

Description automatically generatedA picture containing handwriting, line, font, diagram

Description automatically generated

This moving average can be considered to take into account the last k values, where Beta^k ~ 1 / e ~ 0.35 (the exponential function)

A close-up of a white board

Description automatically generated with low confidence

This EWMA is used more than the classical MA because it doesn’t require to store a number of values equal to the length of the window and it’s extremely easy and fast to implement in only one line

**Bias Correction in Exponentially Weighted Averages**

The EWMA has a problem when there is no history and the estimates are biased

A picture containing line, diagram, font

Description automatically generated - green line – with bias correction; purple line – without bias correction

The solution is to divide Vt with (1 – beta^t); this denominator will become 0 as t gets larger, but initially it helps the values to be a bit bigger

**Gradient Descent with Momentum**

When you are using the gradient descent it can oscillates a lot across some dimensions and because of this oscillation we cannot use large learning rates

A picture containing text, handwriting, line, circle

Description automatically generated

The solution is to use the momentum: in this way we average out the oscillations, but we do not slow the learning process, because the steps taken towards the correct direction aren’t averaged out

A picture containing text, font, white, screenshot

Description automatically generated

The algorithm is called momentum because we can think that dW and db represent the acceleration, VdW and Vdb the velocity, and beta the friction

A common value for beta is 0.9 (the average of the last 10 gradients)

In practice some do not implement the (1 – beta) part, but this implies more fine-tuning for the learning rate alpha

Additionally, the bias correction is not usually implemented because after several iteration the problem solves by itself

**RMSprop**

Root Mean Square prop is another optimizer that uses the EWMA in order to minimize the oscillations and speed up the learning process (because we don’t have oscillations we can increase the learning rate)

It’s based on that fact it minimizes large gradients (so that high oscillations which have a high slope)

A whiteboard with writing on it

Description automatically generated with low confidence

It also uses a small epsilon that is added to the denominator to avoid division by 0

**Adam Optimization Algorithm**

It combines gradient descent with momentum and RMSprop in one optimization algorithm

Additionally, it includes bias estimation for VdW, Vdb, SdW, and Sdb

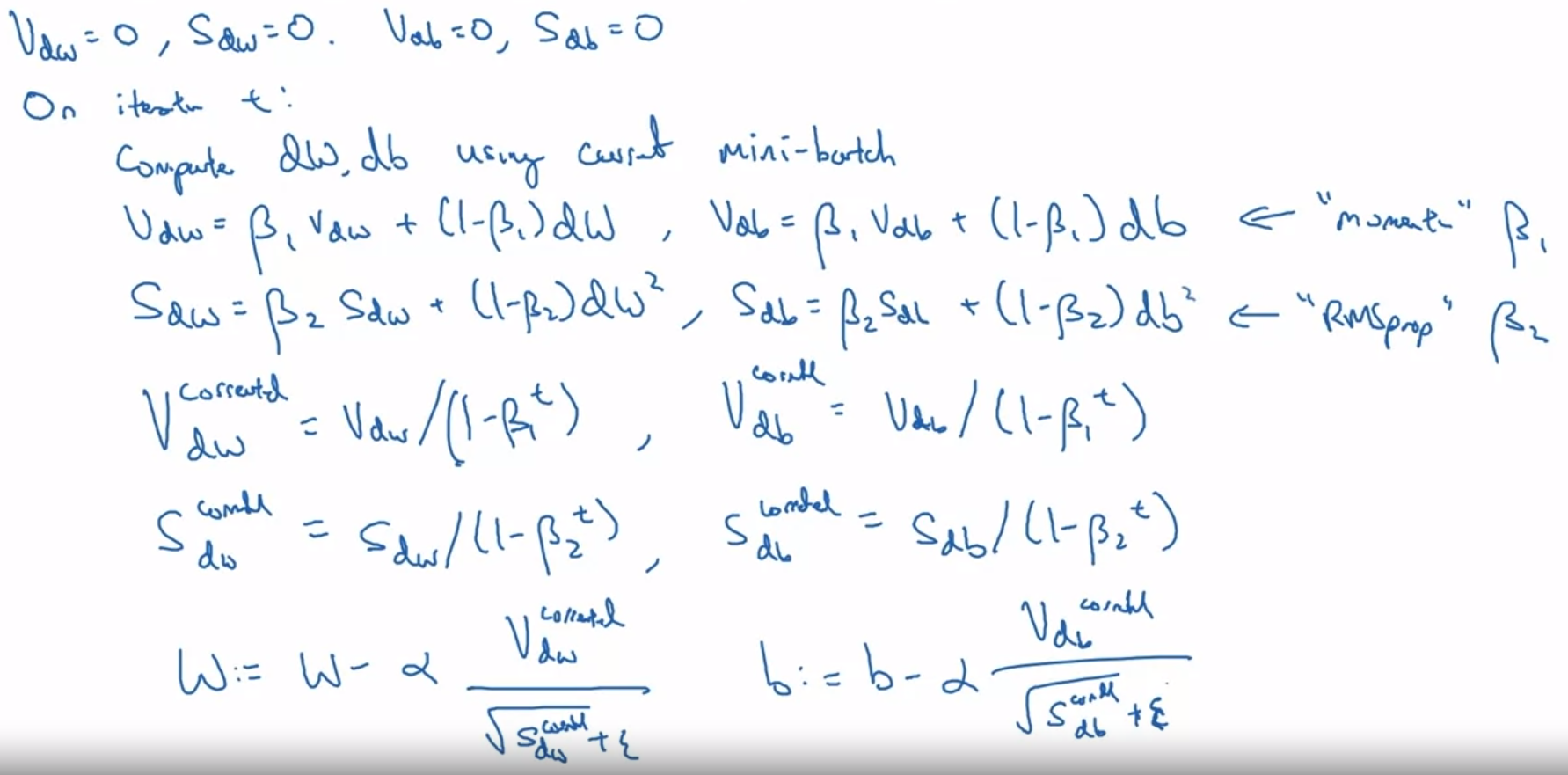
Beta 1 – from gradient descent with momentum – 0.9

Beta 2 – from RMSprop – 0.999

Usually, these values are not tweaked

Epsilon – 10^-8

Alpha – learning rate which has to be tuned



Adam stands for: Adaptive Moment Estimation

**Learning Rate Decay**

Is used to lower the learning rate in time, so that in the end we wander in a tighter region around the optimum

The formula for learning rate decay is:

A picture containing font, text, line, white

Description automatically generated

There are many decay methods:

Alpha = value^epoch\_num \* alpha\_zero – (exponential decay)

Alpha = value / sqrt(epoch\_num) \* alpha\_zero

Alpha = value / sqrt(batch\_num) \* alpha\_zero

Alpha = discrete staircase – you keep dividing the previous alpha by a fixed value

Manual decay – if you train a few models on a very large data set and the training takes days / weeks

**The Problem of Local Optima**

This problem is changed for deep learning because before (for machine learning) the optimization of the cost function was done on a limited number of parameters, so the space had a lower number of dimensions => there could be many local optima

However, this is different for deep learning, where we want to optimize a cost function that has thousands of dimensions. To have a local optima it means that all the dimensions in that point are concave / convex, which is very unlikely; very likely is to have saddle points, where the derivatives in orthogonal directions are zero

A picture containing origami, art

Description automatically generated with medium confidence

[Saddle points - YouTube](https://www.youtube.com/watch?v=8aAU4r_pUUU)

A saddle point is not a local maximum, nor a local minimum

However, it can be detected if you use a second derivate test

So, it’s unlikely to get stuck in a bad local optima

The real problem is represented by plateaus that can make learning slow

A picture containing boomerang

Description automatically generated

**Optimization methods assignment**

When you compute the mini-batch, the data is first shuffled

From one epoch to another, you may want to shuffle the mini-batches

Momentum takes into account past gradients to smooth out the update

If beta = 0 for gradient descent with momentum, then it will become a plain gradient descent

Common values for the beta are between 0.8 and 0.999, but if you don't want to fine-tune it, just pick beta = 0.9

Learning rate decay can be achieved by using adaptive methods or pre-defined learning rate schedules

If you set the decay to occur at every epoch, the learning rate goes to zero too quickly, so the solution is to decay the learning rate every few steps