Week #2 :- Optimization Algorithms.

Mini-Batch Gradient Descent :-

→ Let's explore different numerical method for optimization that will enable you to train your neural network much faster.

→ As we already discussed that, machine learning is a very iterative process and the whole process works well when we train our network on very large datasets. Training on very "large datasets" can be very slow and frankly quite a frustrating process. Having an efficient optimization algorithm can really help you to speed up your process. So let's get started.

Batch vs mini-batch gradient descent :-

As we have seen that, vectorization allows you to efficiently compute on m examples.

So let's stack examples in a vector.

$$X = [x^1 \; x^2 \; x^3 \; \ldots\ldots \; x^m]$$

$$Y = [y^1 \quad y^2 \quad y^3 \quad \dots \quad y^m]$$

$$X \in \mathbb{R}^{(n_x, m)} \qquad Y \in \mathbb{R}^{(m, 1)}$$

What you did in gradient descent is that, you used to process the whole dataset before taking a step of gradient descent.

→ Now with small datasets its not a problem, but with datasets like where m = 5 million or fifty million then you will need to wait for a lot of time even on a fairly good machine.

Turns out that, you can get a faster algorithm if you improve your gradient descent that start working even before you finish processing your data.

Here's what you do...

Let's say you divide your giant dataset into little baby sets and these small baby sets are called **mini-batches**. Say you have 1000 examples each in these baby sets

$$X = [\underbrace{x^{(1)} \; x^{(2)} \; \dots \; x^{(1000)}}_{X^{[1]}} \mid \underbrace{x^{(1001)}, x^{(1002)}, \dots \; x^{(2000)}}_{X^{[2]}} \mid \dots \mid \underbrace{\dots \; x^{(m)}}_{X^{[5000]}}]$$

Considering original data contains 5 million data points.

Do the same with Y

$$Y = [\underbrace{y^{(1)}\ y^{(2)}\ y^{(3)} \dots y^{(1000)}}_{Y^{\{1\}}} \mid \underbrace{y^{(1001)}\ y^{(1002)} \dots y^{(2000)}}_{Y^{\{2\}}} \mid \dots \underbrace{\dots y^{(m)}}_{Y^{\{n\}}}]$$

Now you have:-

Mini-batch t: $X^{\{t\}}, Y^{\{t\}}$

Each example with i/p and o/p pairs.

$$X^{\{t\}} \in \mathbb{R}^{(n_x, 1000)}; \quad Y^{\{t\}} \in \mathbb{R}^{(1, 1000)}$$

Functionality of Mini-batch Gradient Descent:-

While(Converge){ for $t = 1, \dots, 5000$:

What are you gonna do inside the loop is 1 step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$

Vectorized implementation (1000 examples)

forward prop on $X^{\{t\}}$

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\vdots$$
$$A^{[L]} = g^{[L]}(Z^{[L-1]})$$

Compute cost $J = \frac{1}{1000} \sum_{i=1}^{L} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \times 1000} \sum_l \|w^{[l]}\|_F^2$

Backprop to compute gradients w.r.t $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

$$W^{[l]} := W^{[l]} - \alpha dW^{[l]}, \quad b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

end.

}.

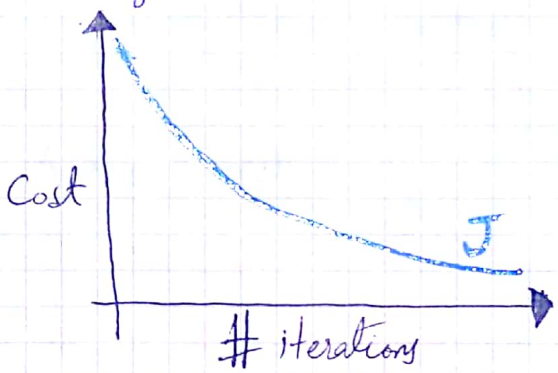
One time of computation of this loop is also called "1- epoch"

Epoch:- Single pass through the training set

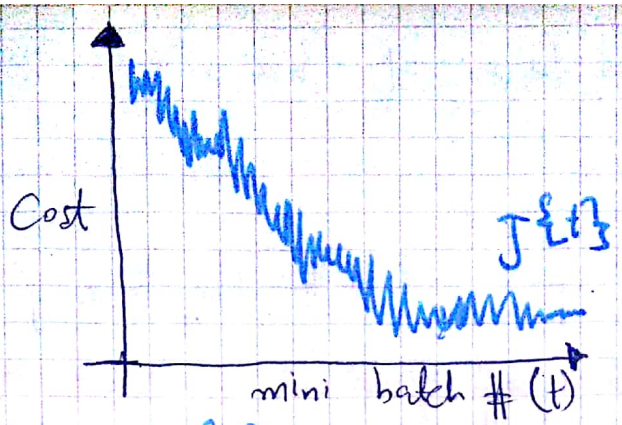
★ With the version of gradient descent that we used earlier, by going through dataset one time, you take one gradient step, but with mini batch in this case you take 5000 step by going through the set one time.

Understanding Mini-batch gradient descent:-

With batch gradient descent, on every iteration you go through the dataset and expect cost to go down on every single single step like this:-



In mini-batch gradient descent, if you plot the cost on every iteration, the cost may not decrease on every single step. Because in min-batch algorithm, its like you are training your model on different training set everytime and the plot will go downwards but it will be a bit noisy.



Plots $J^{\{t\}}$ computed using $X^{\{t\}}, Y^{\{t\}}$

The reason for it being noisy is that may be there is batch 1 with easy data so the cost is going down but the second batch is not that easy and there are mislabeled examples there.

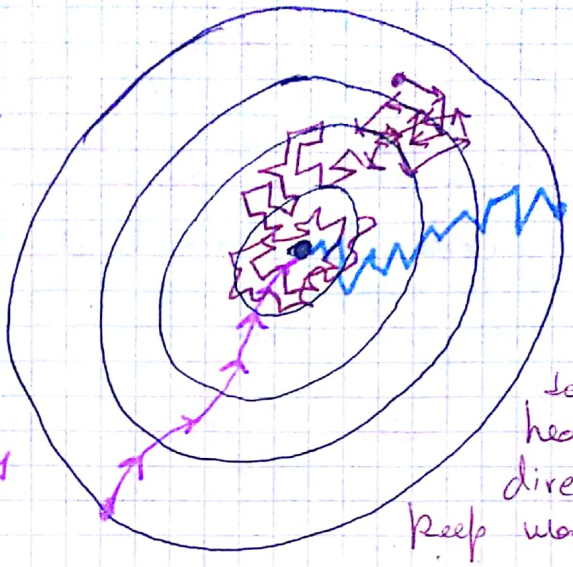
Choosing your mini-batch size :-

- On one extreme if your mini-batch size = m
  ↳ This will give you batch gradient descent
- On the other extreme : if mini-batch size = 1 :
  This will give you an algorithm called
  Stochastic Gradient Descent.
  and in this case every example is a mini batch itself.

So it means with every training example you take a gradient descent step.

Let's look at, what these two extremes will do on optimizing this cost function.

Assume following is the Contour plot of your Cost function :-



With first case gradient descent will take steps and converge to the minimum after some iterations

but with stochastic gradient descent the journey of the optimization will be complex and sometimes it will head in the wrong direction, and it will keep wandering around the minimum but won't converge.

In practice :- The mini-batch size you will use will be somewhere in between 1 and m.

Stochastic Gradient Descent	In - between	Batch gradient Descent
Lose speedup from vectorization Because you process every example individually.	(mini-batch size) not too big or not too small. ⇒ Fastest learning. Following are the benefits :- → Vectorization. (~1000) → Make progress without needing to wait to process the entire training set.	(mini batch size = m) Takes too long per iteration

Now how do you go about choosing the mini batch size :-

- Here are some guidelines :-

→ If you have a small dataset :- Use batch Gradient Descent.

→ Otherwise :- Typical minibatch size :

64, 128, 256, 512

Sometimes if your code has mini-batch size in a power of 2. Your code runs faster.

→ Make sure minibatch fit in CPU/GPU memory.

★ It is another hyperparameter which you tune, during development of your model in order to optimize the cost function J.

Exponentially weighted averages :-

In order to train the model faster, we need sophisticated optimization algorithms. Let's first discuss the concept of exponentially weighted averages or exponentially weighted moving averages. Let's first understand these, then we will move on to develop more sophisticated algorithms.

→ Let's motivate this by an example :-

Suppose we have data of temperature of London at different time of the years.

$\theta_1 = 40°F \rightarrow 4°C$

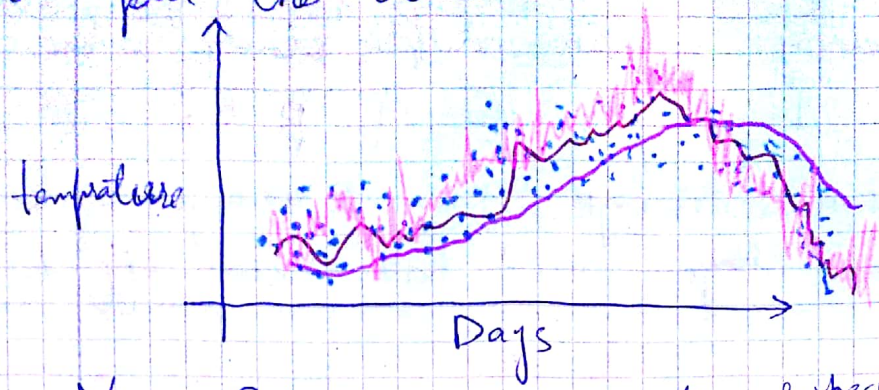
$\theta_2 = 49°F \quad 9°C$

$\theta_3 = 45°F$

$\theta_{180} = 60°F \quad 15°C$

$\theta_{181} = 56°F$

we fold the dataset.



$V_0 = 0$

$V_1 = 0.9 V_0 + 0.1 \theta_1$; $V_2 = 0.9 V_1 + 0.1 \theta_2$ (Temperature of specific day)

$V_3 = 0.9 V_2 + 0.1 \theta_3$ .....

$$V_t = 0.9 V_{t-1} + 0.1 \theta_t$$

If you compute this and plot it in red you get the red curve, the moving average or weighted average of daily temperature.

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

$V_t \Rightarrow$ an approximately averaging over $\approx \frac{1}{1-\beta}$ days' temperature.

e.g $\beta = 0.9 \approx$ 10 day temperature.

If $\beta = 0.98 \Rightarrow \frac{1}{1-0.98} \approx 50$ So approximately last 50 days' temperature. — Color plot.

→ with β being high the curve is lot more smoother, because you are averaging over more days. But on the flipside, the curve is now shifted further to the right, because you are now averaging over much larger window of temperatures.

So by taking larger window this exponentially weighted averaging formula adapts slowly when the temperature changes because $(1-\beta)\theta_t$ will have much smaller weight given β is high so θ_t i.e current change will have low effect.

if $\beta = 0.5$ then for we averaging over 2 days. So, your curve will have much more noise but it adapts quickly

Understanding exponentially weighted averages:-

Recall the equation for weighted averages:-

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

Let's dig a bit deeper into the math and understand how it is computing the average of the daily temperatures.

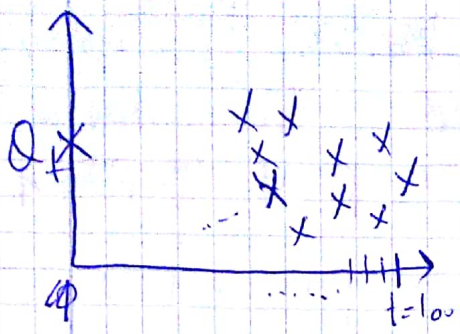
Let's set $\beta = 0.9$ and write out some equation corresponding to it.

$$V_{100} = 0.9 V_{99} + 0.1 \theta_{100}$$
$$V_{99} = 0.9 V_{98} + 0.1 \theta_{99}$$
$$V_{98} = 0.9 V_{97} + 0.1 \theta_{98}$$
$$\dots\dots$$

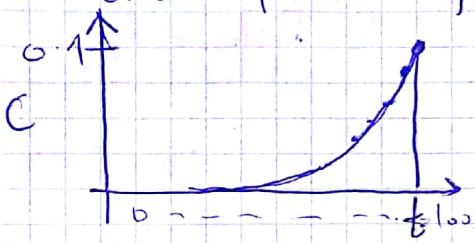
$$V_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9 V_{97}))$$

$$= 0.1\theta_{100} + 0.1\times 0.9\,\theta_{99} + 0.1\times 0.9^2\theta_{98} + 0.1\times 0.98^3\theta_{97} + 0.1\times 0.98^4\theta_{96} + \dots\dots$$

Now if we were to plot the θ_t and visualize it



then we have this function of exponential decay



$\Rightarrow (1-\beta) \times \beta^{99} \ldots\ldots$

$(1-\beta) \cdot (\beta)^0$

- we take elementwise product of these two functions and then summing it up.

→ all of the underlined coefficients add up to 1 or very close to 1, upto a detail called Bias Correction.

→ Now question is how many days is this averaging over. turnsout that:

$$(0.9)^{10} \approx 0.35 \approx \frac{1}{e}$$

$$(1-\beta)^{1/\beta} = \frac{1}{e}$$

→ So it takes about 10 to decay 1/3.

what if $0.98 \Rightarrow \beta$

$(0.98)^{50} \approx 1/e$ will take 50 days to decay to 1/3rd

- Hence $(\beta)^? = 1/e$ tell you how many days it is averaging over.

Finally let's see how do you actually implement this:

$V_0 = 0$

$V_1 = \beta V_0 + (1-\beta)\theta_1$

$V_2 = \beta V_1 + (1-\beta)\theta_2$

⋮

In practice

Initialize $V_\theta = 0$

$V_\theta := \beta V + (1-\beta)\theta_1$

$V_\theta := \beta V_\theta + 1-\beta\ \theta_2$

.....

$V_\theta = 0$

Repeat {

Get next θ_t

$V_\theta = \beta V_\theta + (1-\beta)\ \theta_t$ }

This saves you a lot of memory, but this is not as better estimate as explicitly taking m days' temperature adding them up and divide by 10.

Bias Correction in Exponentially weighted averages:-

→ It can make computation of these averages more accurate.

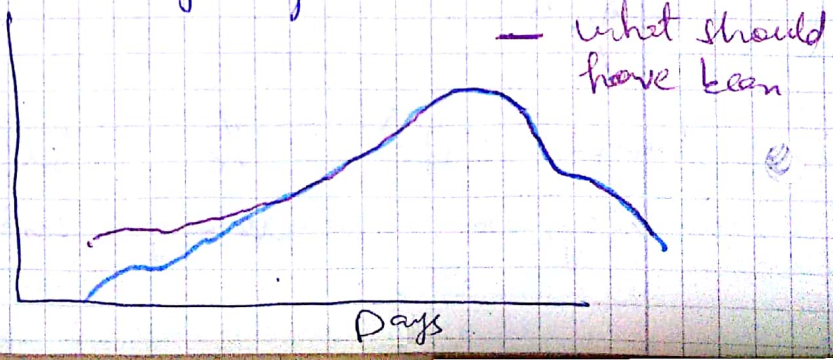
Let's explore how this works.

Previously we saw weighted average plot for different β values so keeping that in mind let's discuss further:

S. when $\beta = 0.98$, you don't actually get that purple curve but you get a curve like this with this formula:

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

in blue.



It starts pretty low, let's see how to fix that: when you implement the moving average, you initialize it with $V_0 = 0$.

$$V_1 = 0.98 V_0 + 0.02 \theta_1$$

$$\text{So } V_1 = 0.02 \theta_1$$

That's why you get much lower value. So it's not a very good estimate of first day's temperature.

$$V_2 = 0.98 V_1 + 0.02 \theta_2$$

$$V_2 = 0.98 \times 0.02 \theta_1 + 0.02 \theta_2$$

$$= 0.0196 \theta_1 + 0.02 \theta_2$$

assuming θ_1 and θ_2 are positive then V_1 and V_2 will much less then θ_1 or θ_2, which is not a good estimate of first ~~stars~~ days' temperature.

Turns out, there is another way to get around this problem.

→ Instead of taking V_t, take $\frac{V_t}{(1-\beta^t)}$, where t is the current date you are on.

let's take a concrete example to understand this:

$$t = 2 : \quad 1 - \beta^t = 1 - (0.98)^2 = 0.0396$$

$$\text{So } \frac{V_2}{0.0396} = \frac{0.0196 \theta_1 + 0.02 \theta_2}{0.0396}$$

So this becomes the weighted average of θ_1 and θ_2 and the denominator removes the bias.

→ So when t is large $1-(\beta)^t \approx 1$ as $(\beta)^t \approx 0$ so in the graph after sometime both graphs matches.

Gradient Descent with Momentum :-

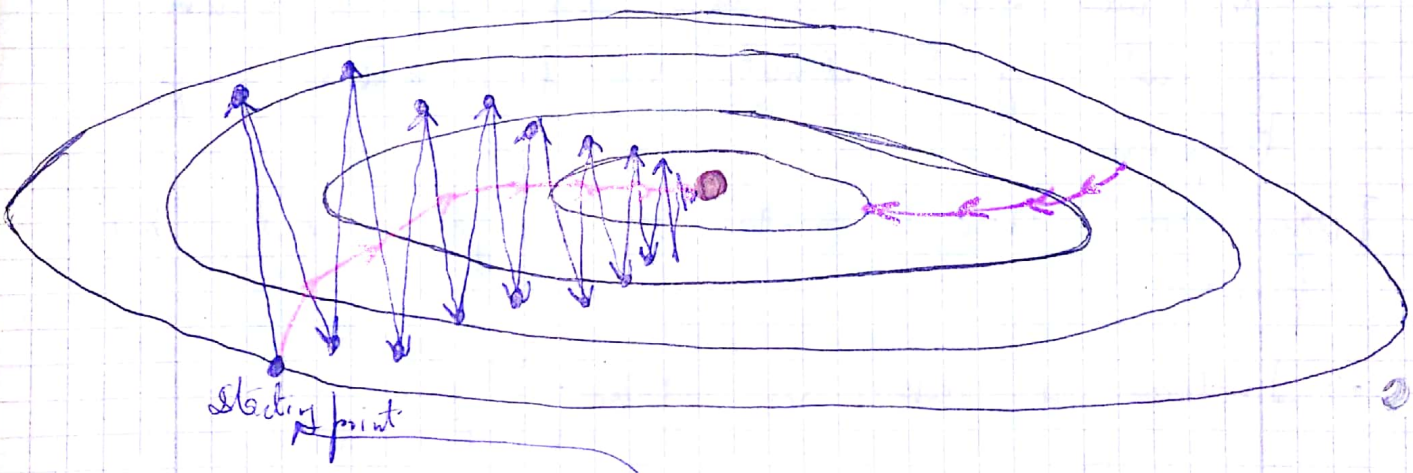
It works better than the simple Gradient descent algorithm.

In one sentence :-

The basic Idea is to Compute the exponentially weighted average of your gradients and then use that gradient to update your weights instead.

Let's motivate this by this example :-

Let's say you are trying to optimize a Cost function, who's Contour plot looks like this : in blue



Suppose you start from here and following is the shape of Convergence. This kind of behaviour slows down gradient descent and prevents you from using larger learning rates.

If you were to use larger learning rates, you might end up overshooting or diverging.

So in order to prevent the oscillation, you use a learning rate which is not too large.

→ On the vertical axis you want slow learning obviously to prevent oscillation, but on the horizontal axis you need faster learning.

→ Here is what you can do to implement Gradient descent with momentum.

Momentum:-

On iteration t:

Compute dw, db on current mini-batch

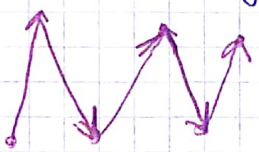
$$V_{dw} = \beta V_{dw} + (1-\beta) dw$$

$$V_{db} = \beta V_{db} + (1-\beta) db$$

$$w = w - \alpha V_{dw} \quad ; \quad b = b - \alpha V_{db}$$

Now what this does is smoothout the steps of gradient descent.

Now how this works is that, let's say previous few derivatives that you computed were like this



Now if we take the weighted average of all of this in vertical direction it will be close to zero because the positive value will cancel out the negative one. But in horizontal direction are the slopes are in the same direction so the average in horizontal direction will still be big.

So, that's why gradient descent with momentum yields much faster convergence because the oscillations in vertical direction will be smoothed out. Illustration is shown in Pink.

+ One intuition about it is that, when you are trying to optimize a ball shaped curve.

then these derivative terms (dw, db) you can think of as providing acceleration to a ball that you rolling down hill

The momentum term V_{dw}, V_{db} are representing the velocity.

⇒ You take a ball and derivative imparts acceleration to a ball which is rolling down hill and it is getting faster and faster (shown in Purple in the Contour) because of acceleration. As β is less than 1, so this plays the role of friction and it prevents your ball from speeding up.

$V_{dw} = 0$, $V_{db} = 0$

Implementation Details:-

On iteration t:

Compute dw, db on the current mini-batch

$$V_{dw} = \beta V_{dw} + (1-\beta) dW$$

$$V_{db} = \beta V_{db} + (1-\beta) db$$

$$W = W - \alpha V_{dw} \quad , \quad b = b - \alpha V_{db}$$

Hyperparameters : α, β $\beta = 0.9$

$(0.9)^{10} = 1/e \Rightarrow$ So averaging over last 10 iteration gradients.

How about bias Correction ?

→ Do you wanna take $\frac{V_{dw}}{1-\beta^t}$ and $\frac{V_{db}}{1-\beta^t}$ instead of V_{dw} and V_{db}, turns out in practice people don't use it, because after 10 iterations your moving average will have warmed up and there will be no longer a bias estimate.

Note:- In most of literature on gradient descend with momentum, you see $(1-\beta)$ term being omitted from $(1-\beta)\,dw$ and $(1-\beta)\,db$, and the net effect of this is that your V_{dw} and V_{db} will be scaled by the factor of $1-\beta$.

→ Both of these works fine but it affects what's the best value of the learning rate alpha.

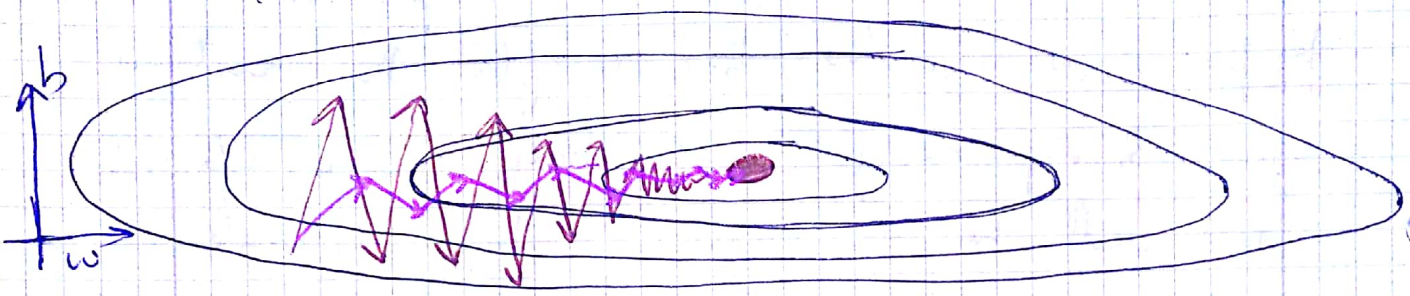
But this formulation seems a little less intuitive., because one impact of this is that, if you end up tuning your hyperparameter β then this affects the scaling of V_{dw} and V_{db} and in the end you'll also have to end up learning rate tuned.

RMSprop:

Root mean square propagation.

Let's see how it works.

Let's use the same Contour example from last topic:-



In order to provide intuition, let's say that, vertical axis is parameter b and horizontal one is w.

You want to slow down learning in b direction and speed up or at least not slow down in w direction.

→ Now this is what RMS prop does to accomplish this:-

On iteration t:

Compute dw, db on current mini batch

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\, dw^2 \quad \text{(elementwise)}$$

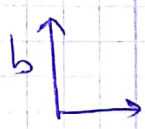
$$S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2 \quad \text{(elementwise)}$$

$$w := w - \alpha \frac{dw}{\sqrt{S_{dw}}} \qquad b := b - \alpha \frac{db}{\sqrt{S_{db}}}$$

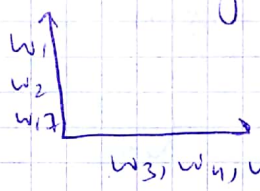
Let's gain some intuition of how this works:-

So, as we established that in vertical direction we want to slow down the learning so what we are hoping is that, S_{dw} will be relatively small and S_{db} will be relatively large so dw gets divided by small number and we get large step whereas db gets divided by large number and b's step is small.

and we get the following shape of optimization curve (shown in purple).



⇒ This axis is just taken to get the intuition. In practice, your data can be very high dimensional and your axis can be like this



etc.

As we are dividing in this algorithm so, to ensure numerical stability i.e. prevention of potential division from zero we add $\varepsilon = 10^{-8}$ or 10^{-7} in the denominator just to make sure that our algorithm doesn't blow up.

→ We talked about momentum and as well as RMS prop, but if we put them together, we get an even better algorithm. Let's talk about that.

Adam Optimization algorithm:-

This algorithm take momentum and RMS prop and puts them together to form a new one.

To implement:

$V_{dw} = 0$, $S_{dw} = 0$, $V_{db} = 0$, $S_{db} = 0$

On iteration t

Compute dw, db using current mini batch.

Then momentum:-

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$$

$$V_{db} = \beta_1 V_{db} + (1-\beta_1) db$$

RMS Prop

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$$

$$S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$$

In typical implementation of Adam, you do implement bias Correction.

$V_{dw}^{Corrected}$ ← after bias Correction $= V_{dw} / (1-\beta_1^t)$, $V_{db}^{Corrected} = V_{db} / (1-\beta_1^t)$

$S_{dw}^{Corrected}$ ← Corrected after bias Correction $= \frac{S_{dw}}{(1-\beta_2^t)}$, $S_{db}^{Corrected} = \frac{S_{db}}{(1-\beta_2^t)}$

Finally update

$$W := W - \alpha \frac{V_{dw}^{Corrected}}{\sqrt{S_{dw}^{Corrected}} + \varepsilon}$$

$$b := b - \alpha \frac{V_{db}^{Corrected}}{\sqrt{S_{db}^{Corrected}} + \varepsilon}$$

This algorithm has no. of hyperparameters :-

α :- needs to be tuned

β_1 :- 0.9 ⇒ default choice [Computing weighted average of dw, db]

β_2 :- 0.999 ⇒ default choice [Computing weighted average of dw^2 and db^2]

ε :- Doesn't matter, usually 10^{-8}.

Adam ⇒ Adaptive moment estimation.

β_1 is used to compute weighted average of dw, db

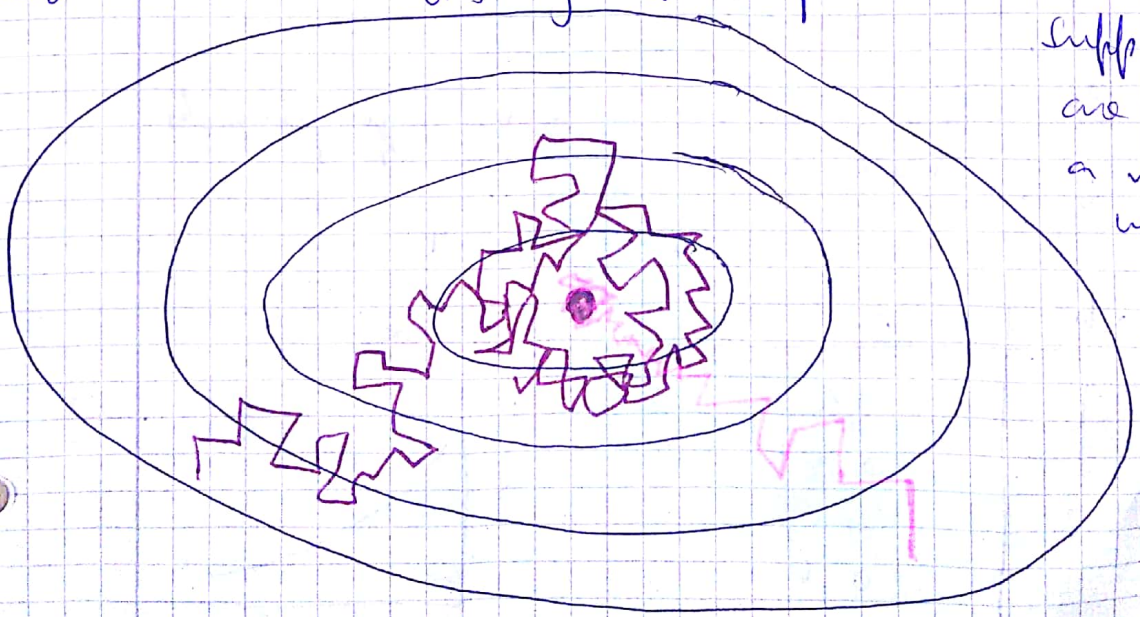
β_2 is for square of dw and db

So these are 2 moments.

Learning Rate Decay :-

One of the things that slightly might helps you to speed up your learning algorithm is to slowly reduce your learning rate. This is called "Learning rate Decay"

Let's motivate this by an example :-



Suppose you are implementing a mini batch with small dataset like 64

Following is the trend shown in red depicted by the optimization algorithm i.e. it is converging but not getting to zero but just wander around it.

That's may be because your learning rate is fixed or there is some noise in your mini-batch.

⇒ We can avert this difficulty by slowly reducing α.

Now what is the intuition behind it?

- May be at the start you take big steps, but now as you are converging to zero you slowly decrease α and now near to zero the oscillations will not be that big and will be in a much tighter region.

Here is how you implement it:

1 epoch = 1 pass through the data.

$$\alpha = \frac{1}{1 + \text{decay\_rate} * \text{epoch\_num}} * \alpha_0$$

→ Now decay rate becomes another hyperparameter which you need to tune.

e.g. $\alpha_0 = 0.2$ decay_rate = 1

epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
⋮	⋮

Other learning rate decay methods:-

Exponential decay

$$\alpha = 0.95^{epoch\_num} \cdot \alpha_0$$

$$\alpha = \frac{k}{\sqrt{epoch\_num}} \cdot \alpha_0 \quad \text{or} \quad \frac{k}{\sqrt{mini\ batch\ num = t}} \cdot \alpha_0$$

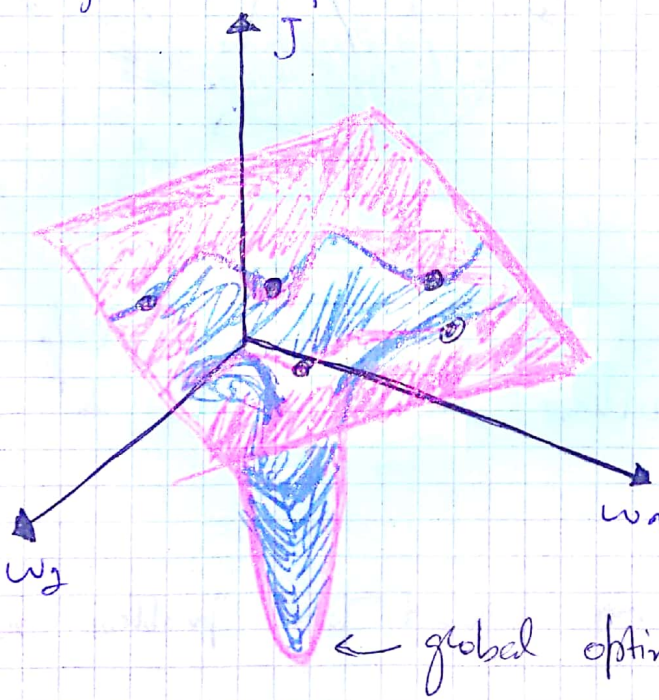
Discrete staircase decay like Armijo stepping

→ Manual Decay:- also works if your model takes days to train.

The Problem of local Optima:-

In early days of deep learning, people used to worry about getting stuck in bad local optima.

Let's visualize that problem.



← global optimum.

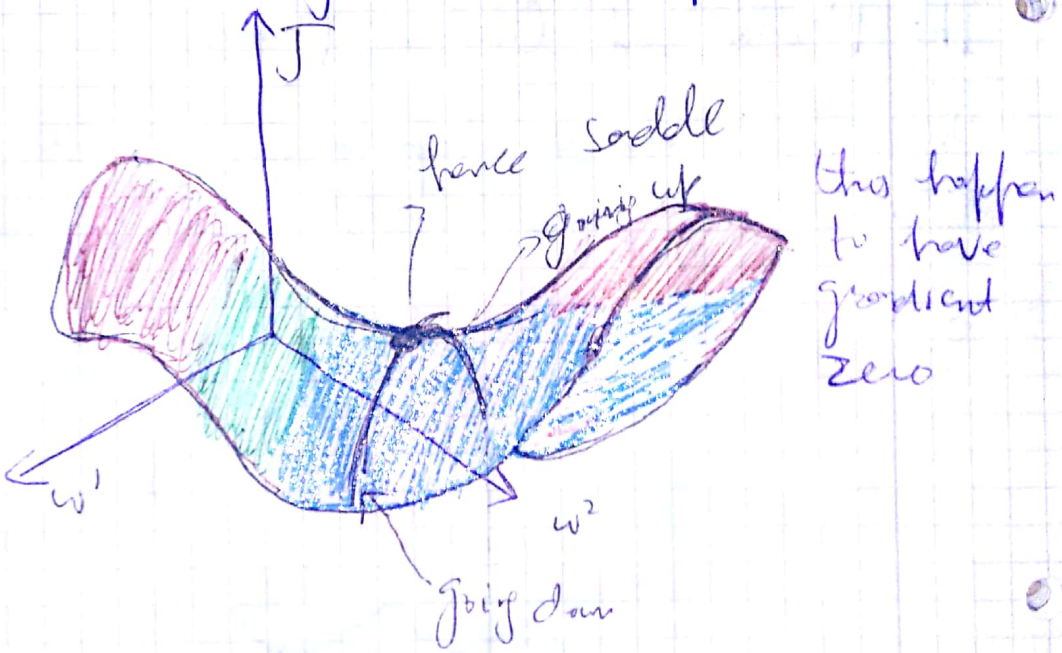
In this picture there are a lot of local optima and it will be easy for an optimization algorithm to get stuck there.

In plots like this, it turns out that most points are not local optima, but they are saddle points.

- In high dimensional function, if a gradient is zero, then in each direction it can either be convex function ∪ or concave like function ∩

So if you are dealing with 20000 dimensions then all need to look like the same which has a low probability.

May be it like that in one dimension it is bending down but on other dimension it is rising up. So most likely its <u>saddle point</u>.

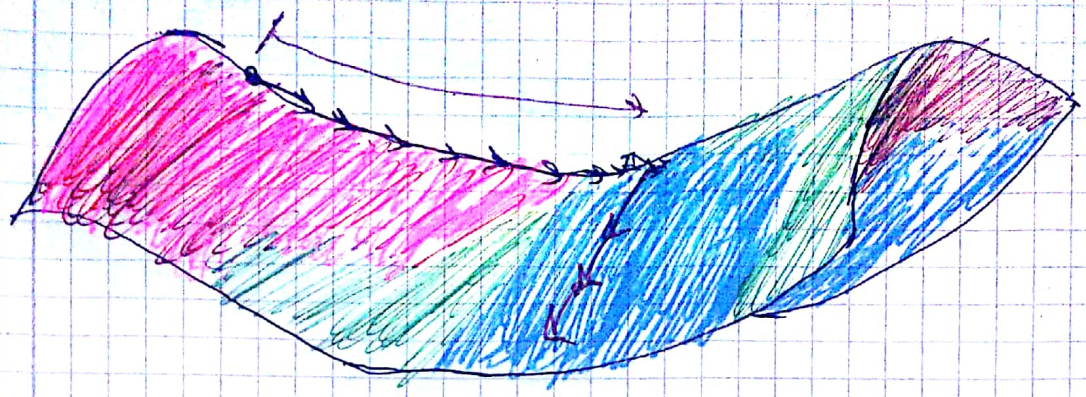


this happen to have gradient zero

→ basic lesson from history of deep learning is that lot of intuitions don't translate to higher dimensional space.

→ If local optima aren't the problem then what is the problem?

Problem of Pleateus



It turns out that problem is of pleateu like region on the curve where derivative remains zero for a long long time.

→ Takeaway is that, you are unlikely to get stuck in a bad local optima when you are working with a high dimensional space.

→ Pleateus can make learning very slow and this is where algorithms like, momentum and Adam can help.