Deep learning

Dr. Aissa Boulmerka
a.boulmerka@centre-univ-mila.dz

2023-2024

CHAPTER 5
OPTIMIZATION ALGORITHMS

Mini-batch gradient descent

- Training NN with a large data is slow. So to find an optimization algorithm that runs faster is a good idea.
- Suppose we have $m = 5\ million$. To train this data it will take a huge processing time for one step.
  - because 5 million won't fit in the memory at once we need other processing to make such a thing.
- You can make a faster algorithm to make gradient descent process some of your items even before you finish the 5 million items.
- In Batch gradient descent we run the gradient descent on the whole dataset.
- While in Mini-Batch gradient descent we run the gradient descent on the mini datasets.

Mini-batch gradient descent

- Suppose we have split m to mini batches of size 1000.

$$X = \underbrace{[x^{(1)}x^{(2)}x^{(3)} \ \dots x^{(1000)}}_{X^{\{1\}}\ (n_x, 1000)}|\underbrace{x^{(1001)} \ \dots x^{(2000)}}_{X^{\{2\}}\ (n_x, 1000)}|\dots|\underbrace{\dots x^{(m)}]}_{X^{\{5000\}}\ (n_x, 1000)}$$

(n_x, m)

- We similarly split X & Y :

$$Y = \underbrace{[y^{(1)}y^{(2)}y^{(3)} \ \dots y^{(1000)}}_{Y^{\{1\}}\ (1,1000)}|\underbrace{y^{(1001)} \ \dots y^{(2000)}}_{Y^{\{2\}}\ (1,1000)}|\dots|\underbrace{\dots y^{(m)}]}_{Y^{\{5000\}}\ (1,1000)}$$

$(1, m)$

If $m = 5\,000\,000$:
5000 mini-batches of 1000 each
mini-batch t : $X^{\{t\}}, Y^{\{t\}}$

Mini-batch gradient descent

- Mini-Batch algorithm pseudo code:

Repeat

for t = 1,…,5000 # this is called an epoch

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

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = g^{[1]}\left(Z^{[1]}\right)$$
$$\vdots$$
$$A^{[l]} = g^{[l]}\left(Z^{[l]}\right)$$

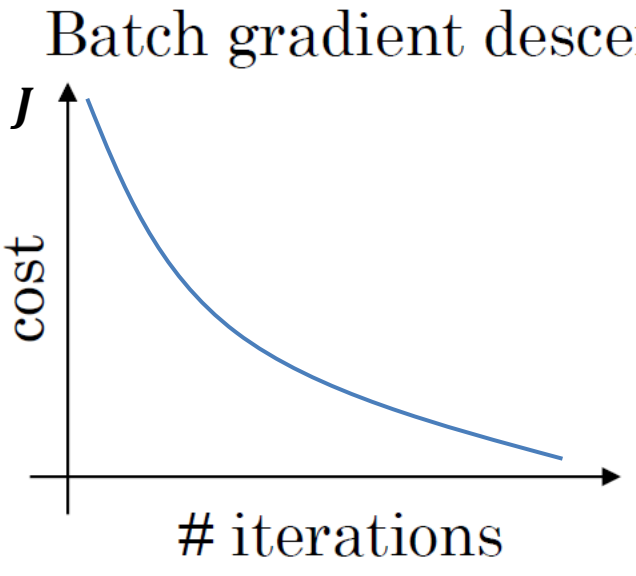
compute cost $J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{1000}\mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) + \frac{\lambda}{2.1000}\left\|w^{[l]}\right\|_2^2$

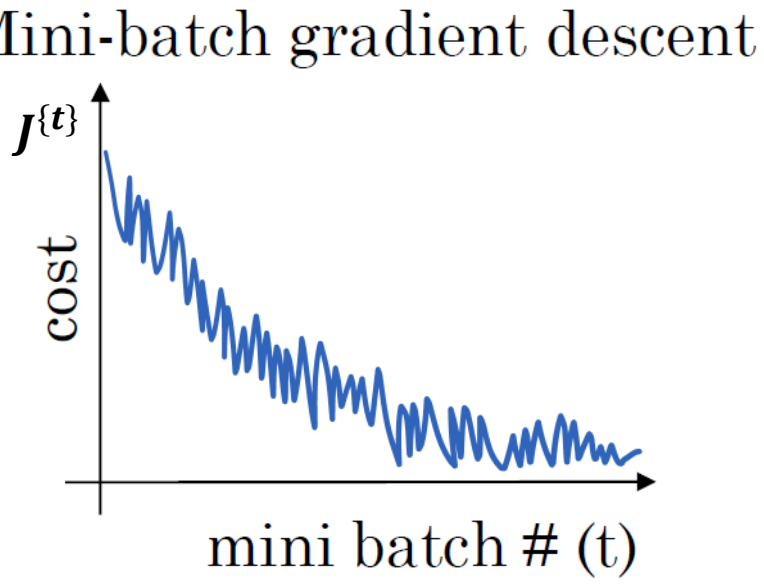
backward propagation to compute gradients

$w^{[l]} = w^{[l]} - \alpha dw^{[l]}$, $b^{[l]} = b^{[l]} - \alpha db^{[l]}$

- The code inside an epoch should be vectorized.
- Mini-batch gradient descent works much faster in the large datasets.

Understanding mini-batch gradient descent

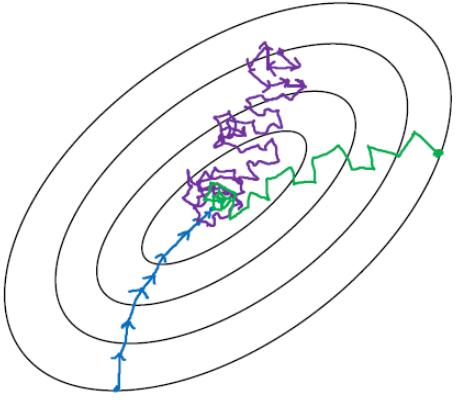




- In mini-batch algorithm, the cost won't go down with each step as it does in batch algorithm.
- It could contain some ups and downs but generally it has to go down (unlike the batch gradient descent where cost function descreases on each iteration)..

Choosing your mini-batch size

- If mini-batch size $= m \Longrightarrow$ Batch gradient descent
- If mini-batch size $= 1 \Longrightarrow$ Stochastic gradient descent (SGD)
- If $1 \leq$ mini-batch size $\leq m \Longrightarrow$ Mini-batch gradient descent



Stochastic gradient descent (SGD)	Mini-batch gradient descent	Batch gradient descent
▪ too noisy regarding cost minimization ▪ won't ever converge ▪ lose speedup from vectorization	▪ faster learning ▪ make progress without waiting to process the entire training set	▪ too long per iteration (epoch)

Guidelines for choosing mini-batch size

I. If small training set (< 2000 examples): use batch gradient descent.

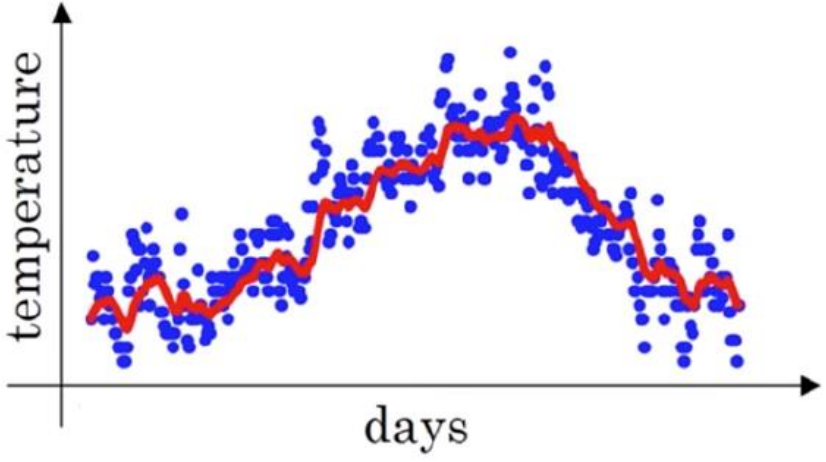
II. It has to be a power of 2 (because of the way computer memory is layed out and accessed, sometimes your code runs faster if your mini-batch size is a power of 2): 64, 128, 256, 512, 1024, ...

III. Make sure that mini-batch fits in CPU/GPU memory.

Note: Mini-batch size is a hyperparameter.

Exponentially weighted averages

$\theta_1 = 40°F$
$\theta_2 = 49°F$
$\theta_3 = 45°F$
$\vdots$
$\theta_{180} = 60°F$
$\theta_{181} = 56°F$
$\vdots$



Now lets compute the Exponentially weighted averages:

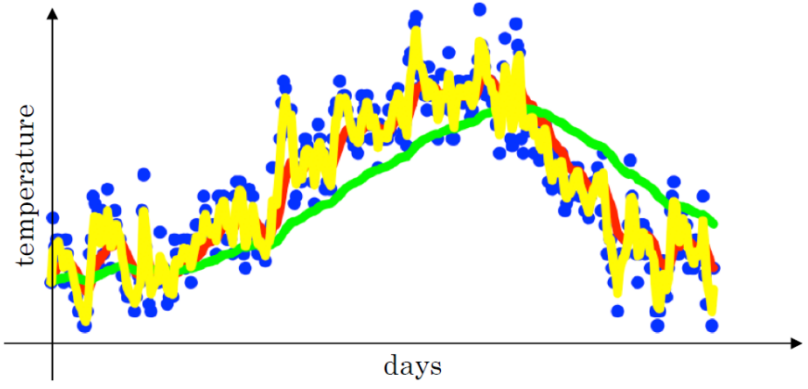
$V_0 = 0$
$V_1 = 0.9V_0 + 0.1\theta_1$
$V_2 = 0.9V_1 + 0.1\theta_2$
$V_3 = 0.9V_2 + 0.1\theta_3$
$\vdots$
$V_t = 0.9V_{t-1} + 0.1\theta_t$

Exponentially weighted averages

- General equation:

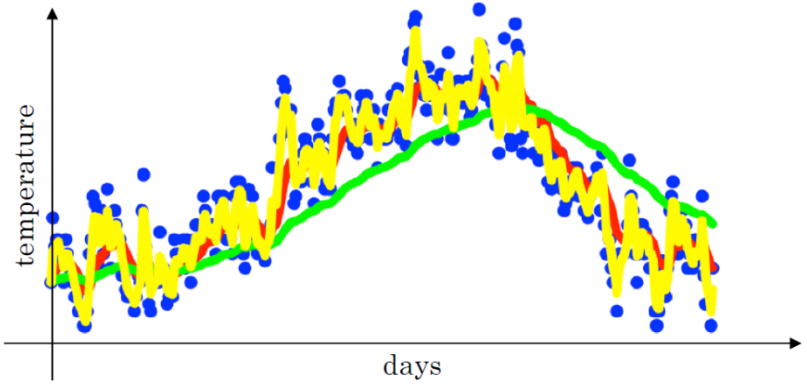
$$V_t = \beta V_{t-1} + (1 - \beta)\theta_t$$

- If we plot this it will represent averages over $\approx \frac{1}{1-\beta}$ entries:
  - $\beta = 0.9$ will average last 10 entries
  - $\beta = 0.98$ will average last 50 entries
  - $\beta = 0.5$ will average last 2 entries
- Best beta average for our case is between 0.9 and 0.98



Understanding exponentially weighted averages

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



- **Intuition:** exponentially weighted averages can give different weights to recent data points (theta) based on value of beta. If beta is high (around 0.9), it smoothens out the averages of skewed data points (oscillations w.r.t. Gradient descent terminology). So this reduces oscillations in gradient descent and hence makes faster and smoother path towards minima.

Bias correction in exponentially weighted averages

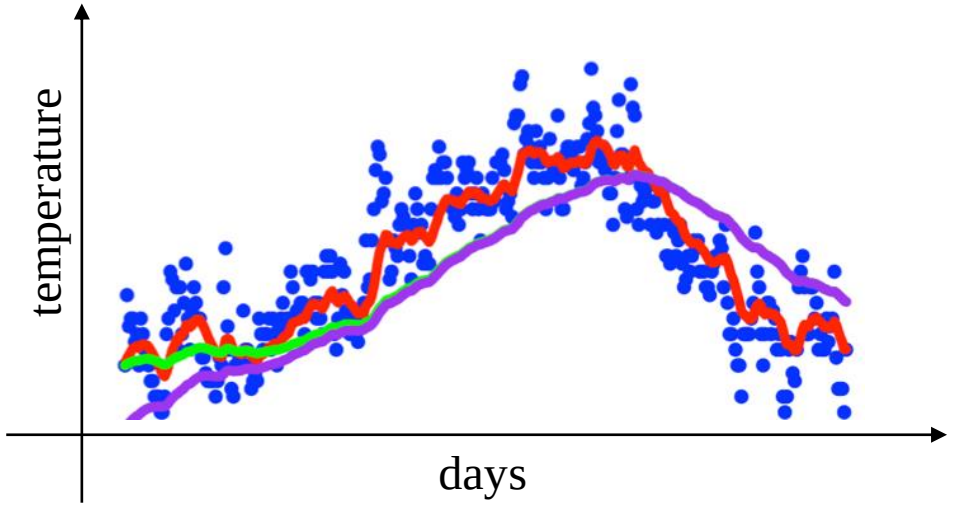
- The bias correction helps make the exponentially weighted averages more accurate.
- Because $V_0 = 0$, the bias of the weighted averages is shifted and the accuracy suffers at the start.
- To solve the bias issue we have to use this equation:

$$V_t = \frac{\beta}{1 - \beta^t} V_{t-1} + \frac{1 - \beta}{1 - \beta^t} \theta_t$$

- As t becomes larger the $(1 - \beta^t)$ becomes close to 1

Bias correction

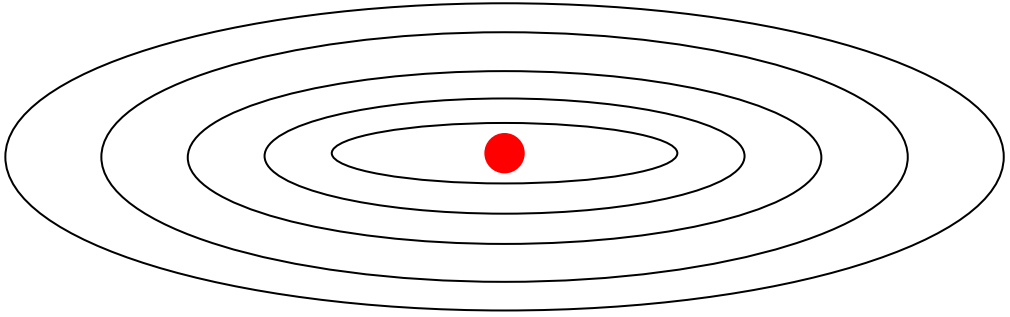
$\beta = 0.98$



$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$
$$V_0 = 0$$
$$V_1 = 0.98V_0 + 0.02\theta_1 = 0.02\theta_1$$
$$V_2 = 0.98V_1 + 0.02\theta_2$$
$$V_2 = 0.98 * 0.02\theta_1 + 0.02\theta_2$$
$$V_2 = 0.0196\theta_1 + 0.02\theta_2$$

$$\frac{V_t}{1-\beta^t}$$
$$t = 2: \quad 1-\beta^t = 1-0.98^2 = 0.0396$$
$$\frac{V_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$$
$$\frac{V_2}{0.0396} = 0.4949\theta_1 + 0.5051\theta_2$$

Gradient descent with momentum



- The momentum algorithm almost always works faster than standard gradient descent.
- The simple idea is to calculate the exponentially weighted averages for your gradients and then update your weights with the new values.

$vdW = 0, vdb = 0$

on iteration t:

```
  # can be mini-batch or batch gradient descent
  compute dw, db on current mini-batch
```

$vdW = \beta * vdW + (1 - \beta) * dW$

$vdb = \beta * vdb + (1 - \beta) * db$

$W = W - \alpha * vdW$

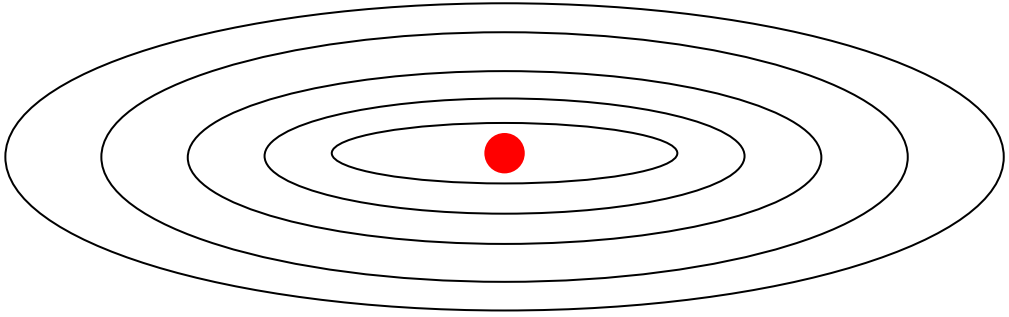
$b = b - \alpha * vdb$

α : *learning rate*

Implementation details

- Momentum helps the cost function to go to the minimum point in a more fast and consistent way.
- beta is another **hyperparameter**. $\beta = 0.9$ is very common and works very well in most cases.
- In practice people don't bother implementing bias correction.

RMSprop (Root Mean Squered)



- Stands for **Root Mean Square prop.**
- This algorithm speeds up the gradient descent.
- Pseudo code:

$sdW = 0, sdb = 0$
on iteration t:
  # can be mini-batch or batch gradient descent
  compute dw, db on current mini-batch
  $sdW = (\beta * sdW) + (1 - \beta) * dW\text{^}2$ #squaring is element-wise
  $sdb = (\beta * sdb) + (1 - \beta) * db\text{^}2$ #squaring is element-wise
  $W = W - \alpha * dW / sqrt(sdW)$
  $b = B - \alpha * db / sqrt(sdb)$

α : $learning\ rate$

Adam optimization algorithm

- Stands for **Adaptive Moment Estimation**.
- Adam optimization and RMSprop are among the optimization algorithms that worked very well with a lot of NN architectures.
- Adam optimization simply puts RMSprop and momentum together!

$vdW = 0, vdW = 0, sdW = 0, sdb = 0$
on iteration t:
can be mini-batch or batch gradient descent
compute dw, db on current mini-batch
$vdW = \beta_1 * vdW + (1 - \beta_1) * dW$ # momentum
$vdb = \beta_1 * vdb + (1 - \beta_1) * db$ # momentum
$sdW = \beta_2 * sdW + (1 - \beta_2) * dW^2$ # RMSprop
$sdb = \beta_2 * sdb + (1 - \beta_2) * db^2$ # RMSprop
$vdW = vdW / (1 - \beta_1^t)$ # fixing bias
$vdb = vdb / (1 - \beta_1^t)$ # fixing bias
$sdW = sdW / (1 - \beta_2^t)$ # fixing bias
$sdb = sdb / (1 - \beta_2^t)$ # fixing bias
$W = W - \alpha * vdW / (\sqrt{sdW} + \varepsilon)$
$b = b - \alpha * vdb / (\sqrt{sdb} + \varepsilon)$

α : *learning rate*

Hyperparameters for Adam

- Learning rate (α): needed to be tuned.
- Parameter of the momentum (β_1): 0.9 is recommended by default.
- Parameter of the RMSprop (β_2) : 0.999 is recommended by default.
- ε: 10^{-8} is recommended by default.

Learning rate decay

- Slowly reduce learning rate.
- As mentioned before mini-batch gradient descent won't reach the optimum point (converge). But by making the **learning rate decay** with iterations it will be much closer to it because the steps (and possible oscillations) near the optimum are smaller.
- One technique equations is:

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

  - epoch_num is over all data (not a single mini-batch).

Other learning rate decay methods

- Other learning rate decay methods (continuous):

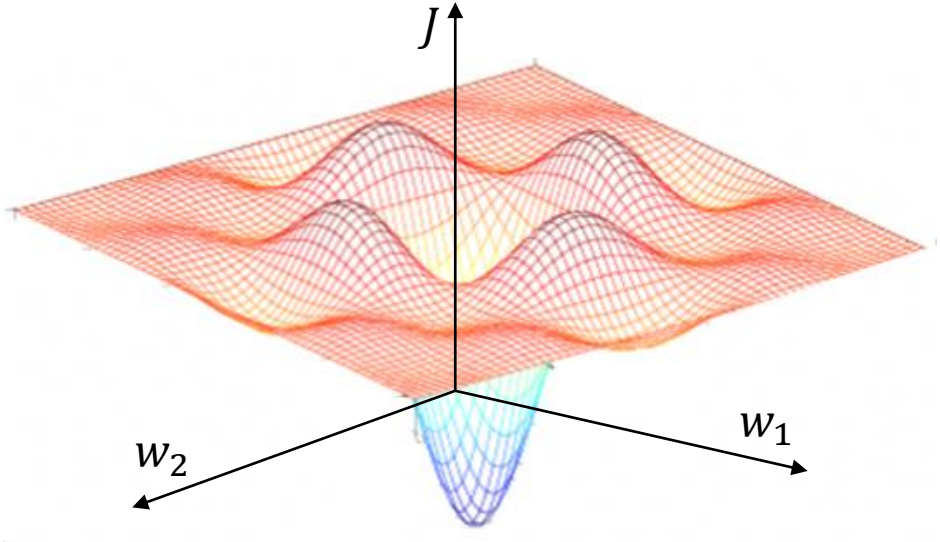
$$\alpha = \left(0.95^{epoch\_num}\right) * \alpha_0$$

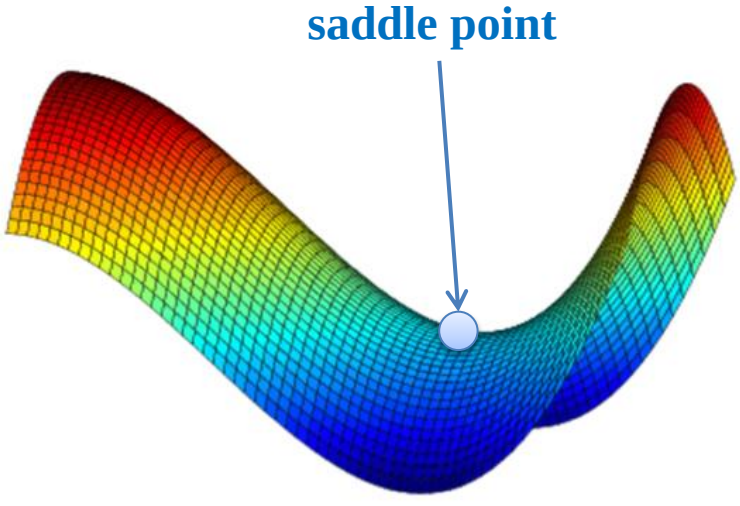
$$\alpha = \left(\frac{k}{\sqrt{epoch\_num}}\right) * \alpha_0 \text{ or } \alpha = \left(\frac{k}{\sqrt{t}}\right) * \alpha_0$$

- Some people perform learning rate decay discretely - repeatedly decrease after some number of epochs.
- Some people are making changes to the learning rate manually.
- Decay method or $decay\_rate$ is another hyperparameter .
- Learning rate decay has less priority.

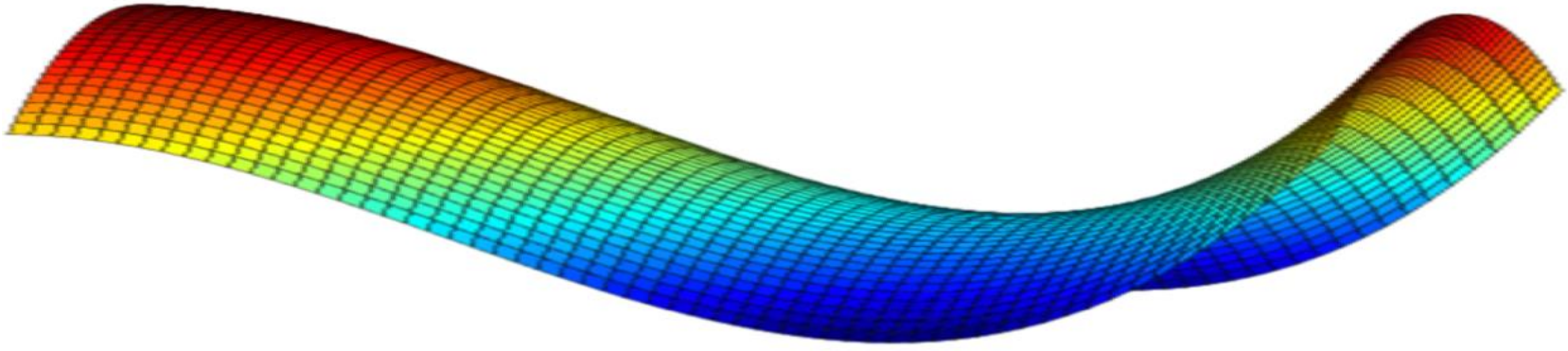
The problem of local optima

- The normal **local optima** is not likely to appear in a deep neural network because data is usually **high dimensional**. For point to be a local optima it has to be a local optima for each of the dimensions which is highly unlikely.
- It's unlikely to get stuck in a bad local optima in high dimensions, it is much more likely to get to the **saddle point** rather to the **local optima**, which is not a problem.





Problem of plateaus



- Plateaus can make **learning slow**:
  - Plateau is a region where the **derivative is close to zero** for a long time.
  - This is where algorithms like momentum, RMSprop or Adam can help.

References

- Andrew Ng. Deep learning. Coursera.
- Geoffrey Hinton. Neural Networks for Machine Learning.
- Kevin P. Murphy. Probabilistic Machine Learning An Introduction. MIT Press, 2022.
- MIT Deep Learning 6.S191 (http://introtodeeplearning.com/)