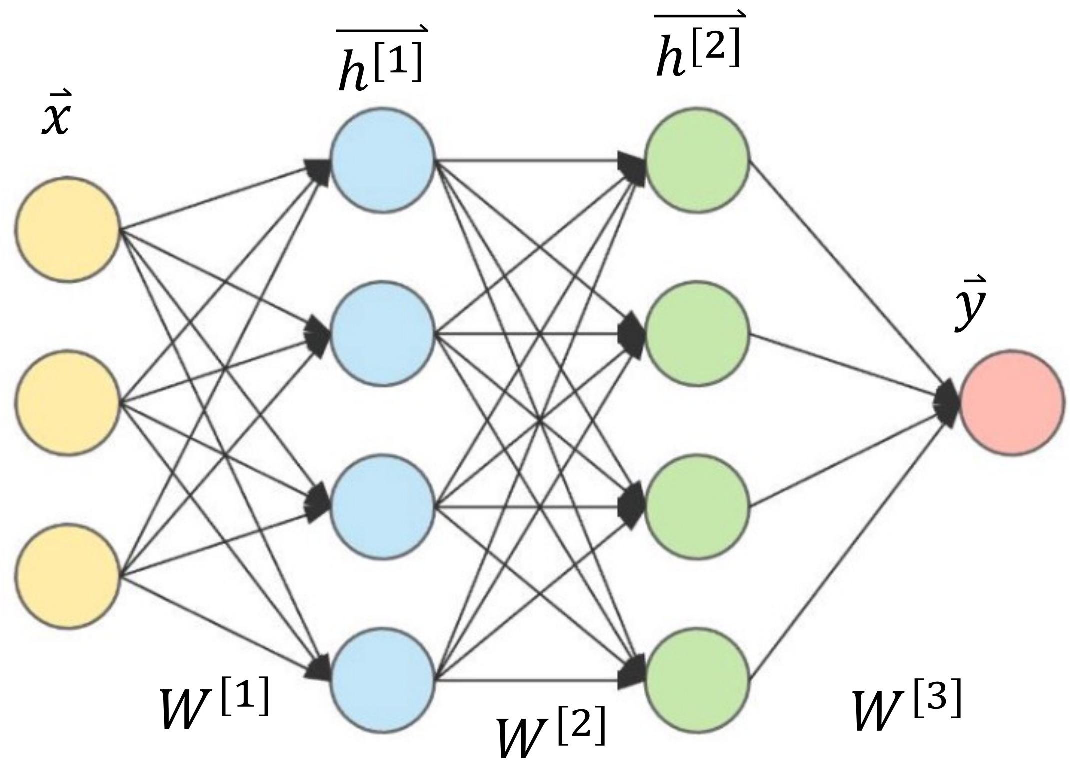
Optimization algorithms

Qingrun Zhang

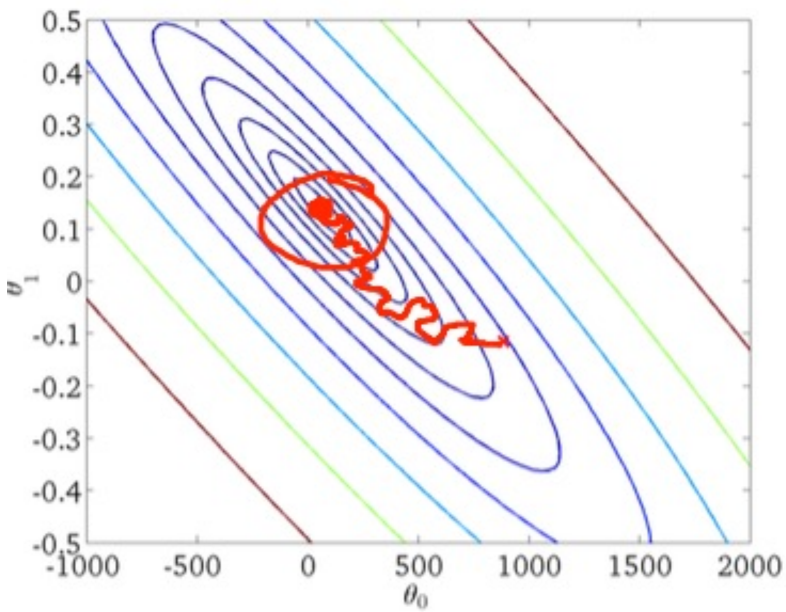
Stochastic Batch and Mini-Batch Gradient Descent

- Forward propagation
- $J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$
- $dw^{[l]} = \frac{\partial J}{\partial w^{[l]}}$
- $w^{[l]} = w^{[l]} - \alpha dw^{[l]}$

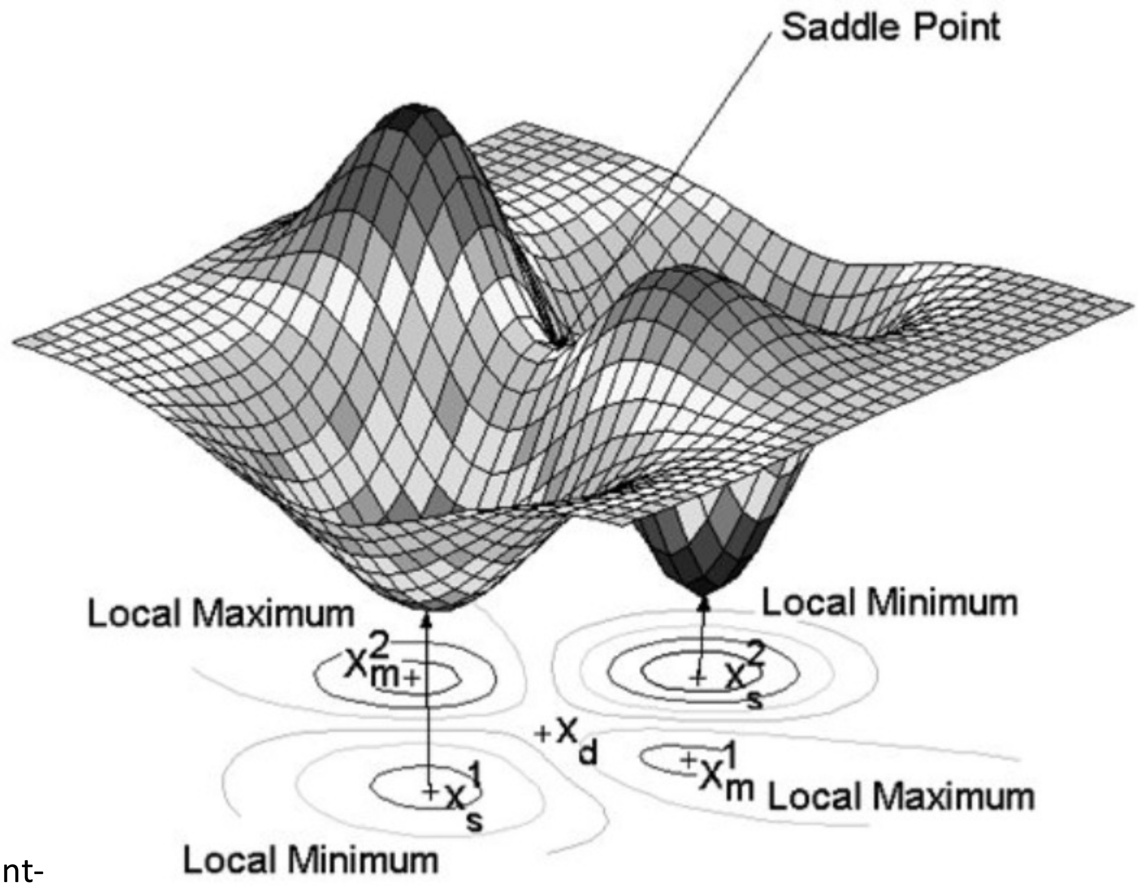


Common Problems when Training Neural Networks

- Local Minima and Saddle Points
- Noisy Gradients



https://stackoverflow.com/questions/35711315/gradient-descent-vs-stochastic-gradient-descent-algorithms



https://www.researchgate.net/profile/Hsiao-Dong-Chiang

Batch Gradient Descent

- $\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}(\widehat{y_i}, y_i)}{\partial \theta_j}$, $m = sample\ size$
- Pros:
  - Computationally Efficient: No updates are required after each sample.
  - Stable converge: Averaging all individual gradients over all samples, we get a good estimate of the true gradient
- Cons:
  - Slower Learning when sample size is big.
  - Local Minima and Saddle Points: we need some noisy gradients to allow gradient jump out of local minimum

Stochastic Gradient Descent

$$\theta_j = \theta_j - \alpha \frac{\partial \mathcal{L}(\widehat{y_i}, y_i)}{\partial \theta_j}$$

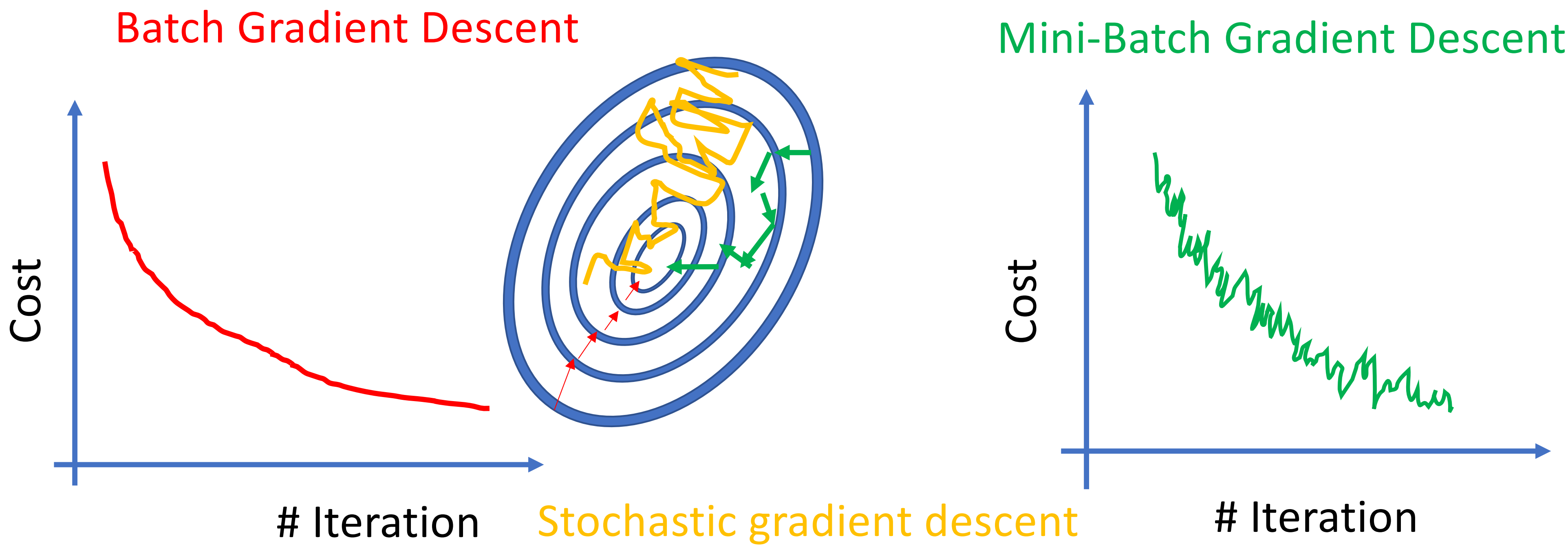
- Update the weights after each sample been processed by neural network
- Pros:
  - Immediate Performance Insights
  - Faster Learning
- Cons:
  - Noisy Gradients: The gradients for each sample can be very noisy. The gradients of each sample is only rough estimates of the true gradient
  - Computationally Intensive: the weight updates for each sample
  - Inability to settle on a global Minimum due to the noisiness

Mini-Batch Gradient Descent

- $\theta_j = \theta_j - \alpha \frac{1}{b} \sum_{i=k.b}^{(k+1)b} \frac{\partial \mathcal{L}(\widehat{y_i}, y_i)}{\partial \theta_j}$. $b = batch\ size$ $k = \left\{1, \frac{m}{b}\right\} (\#batches)$
- The gradient descent is calculated for each mini-batch of samples.
- Pros:
  - Computational Efficiency: in between batch gradient descent and SGD
  - Stable Convergence. Less noisy than SGD, but better than batch gradient descent
  - Faster Learning than batch gradient descent
- Cons:
  - The mini-batch size, b, becomes a new hyperparameter to tune.
- Mini-batch gradient descent is the default implement in DL

Stochastic Batch and Mini-Batch Gradient Descent

Size of minibatch=1, m, or in-between



https://colab.research.google.com/drive/1pikHsxLuZ7TZOrSe7iNt356mDZXra07I#scrollTo=6WpGXoD-9JCy

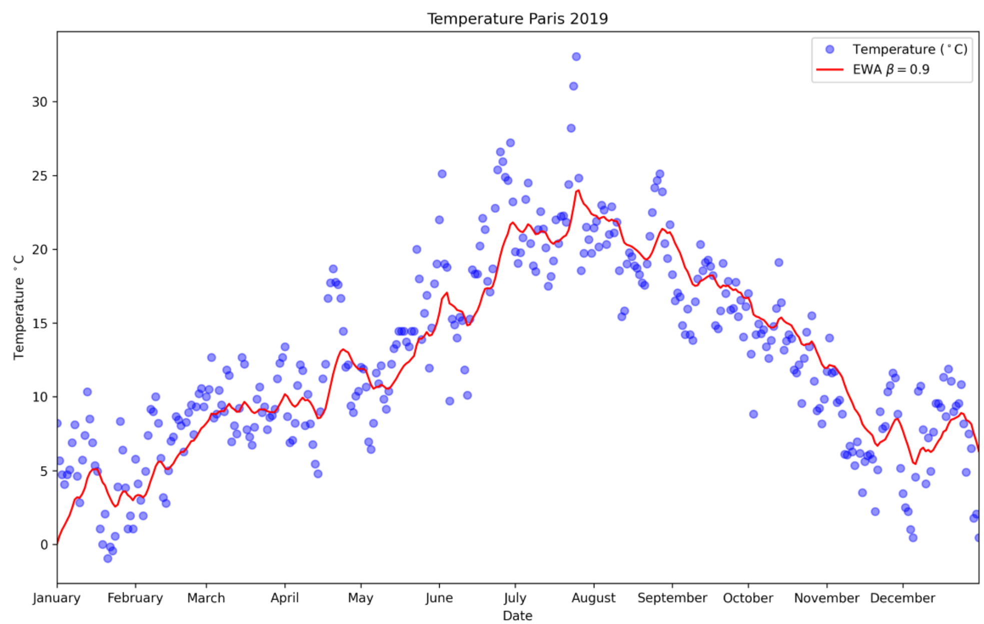
Exponentially Weighted Moving Averages

$$W_t = \begin{cases} 0 & t = 0 \\ \beta \cdot W_{t-1} + (1 - \beta) \cdot \theta_t & t > 0 \end{cases}$$

$$\begin{aligned} \beta &= \text{Weight parameter} \\ \theta_t &= \text{Temperature day } t \\ W_t &= \text{EWA for day } t \text{ (set } W_0 = 0) \end{aligned}$$

$\theta_1 = 4$°C
$\theta_2 = 8$**°C**
$\theta_3 = 7$**°C**
$\theta_4 = 14$**°C**
...
$\theta_{160} = 30$**°C**
$\theta_{161} = 22$**°C**

$$W_1 = 0.9 \cdot W_0 + 0.1 \cdot \theta_1$$
$$W_2 = 0.9 \cdot W_1 + 0.1 \cdot \theta_2$$
$$\vdots$$
$$W_t = 0.9 \cdot W_{t-1} + 0.1 \cdot \theta_t$$



Calculate the moving average of daily temperature.

https://medium.com/mlearning-ai/exponentially-weighted-average-5eed00181a09

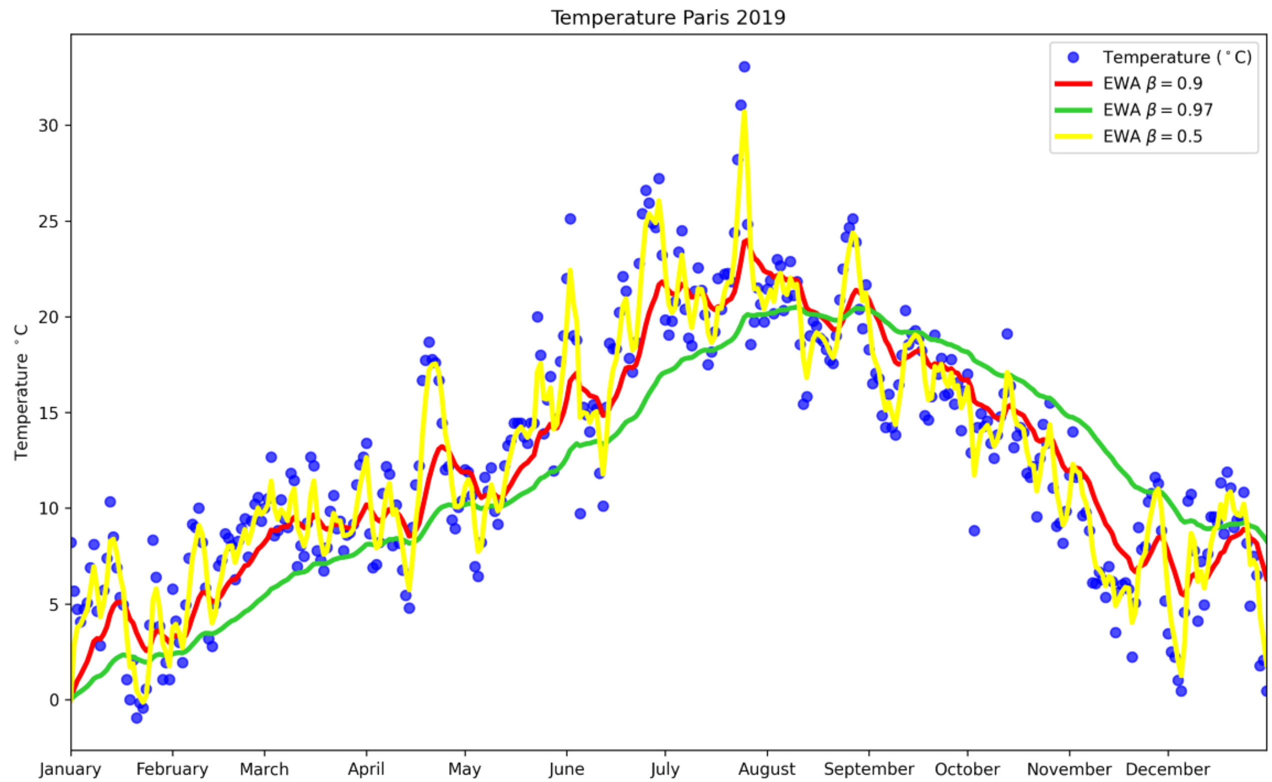
Understand Exponentially Weighted Averages

$$W_t = \underbrace{\color{red}{\beta} \cdot W_{t-1}}_{\text{trend}} + \overbrace{(\color{red}{1-\beta}) \cdot \theta_t}^{\text{current value}}$$

β determines how important the previous value is (the trend), and (1-β) how important the current value is.

$$n_\beta = \frac{1}{1-\beta}$$

β	n_β	**EWA**
0.9	10	Adapts normal
0.98	50	Adapts slowly
0.5	2	Adapts quickly



Intuition of Exponentially Weighted Averages

Lets expand the 3rd term (W_3) using the main equation

$$W_3 = 0.9 \cdot W_2 + 0.1 \cdot \theta_3$$
$$W_2 = 0.9 \cdot W_1 + 0.1 \cdot \theta_2$$
$$W_1 = 0.9 \cdot \underbrace{W_0}_{0} + 0.1 \cdot \theta_1$$

Plug in

$$W_3 = 0.9 \cdot \underbrace{(0.9\underbrace{(0.9 \cdot 0 + 0.1 \cdot \theta_1)}_{W_1} + 0.1 \cdot \theta_2)}_{W_2} + 0.1 \cdot \theta_3$$

Simplify

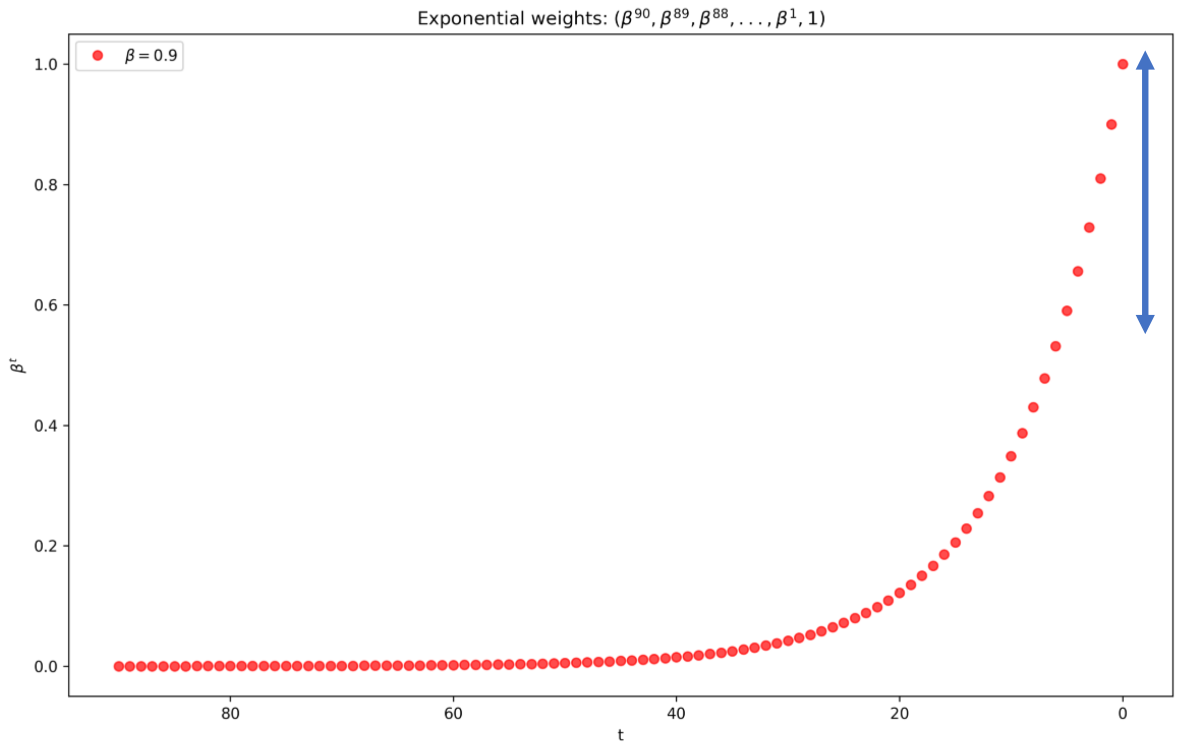
$$W_3 = 0.1(\theta_3 + 0.9\theta_2 + 0.9^2\theta_1)$$

General form

$$W_t = (1-\beta)\theta_t + (1-\beta)\beta\theta_{t-1} + (1-\beta)\beta^2\theta_{t-2} + \cdots + (1-\beta)\beta^{t-1}\theta_1 \Rightarrow W_t = (1-\beta) \cdot \sum_{k=1}^{t} \beta^{t-k}\theta_k$$

How the weights decay when t increase

$$W_t = \underbrace{(1-\beta)}\theta_t + \underbrace{(1-\beta)\beta}\theta_{t-1} + \underbrace{(1-\beta)\beta^2}\theta_{t-2} + \cdots + \underbrace{(1-\beta)\beta^{t-1}}\theta_1 \approx 1$$



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

$$\beta = 0.9$$

$$0.9^{10} \approx \frac{1}{e}$$

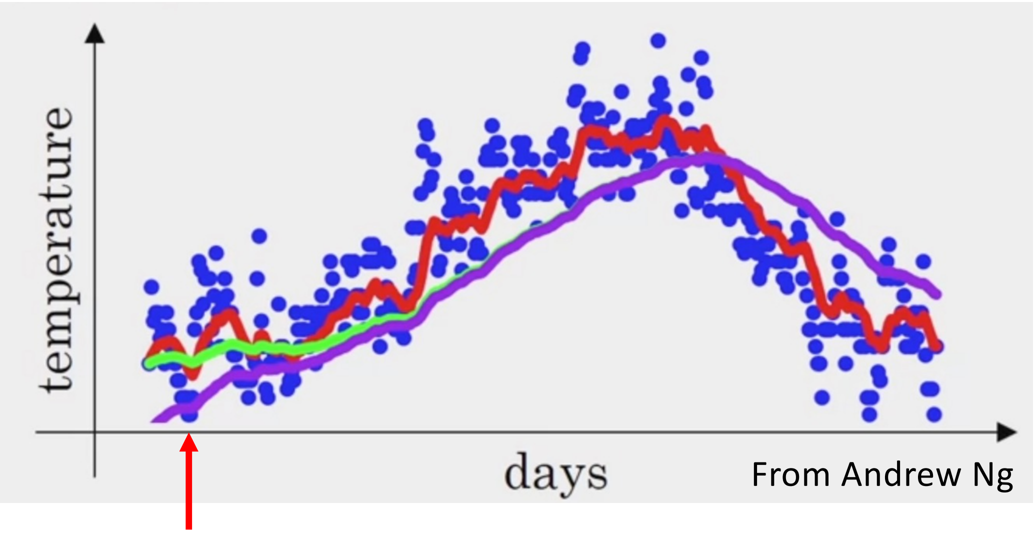
$$\beta = 0.98$$

$$0.98^{50} \approx \frac{1}{e}$$

https://medium.com/mlearning-ai/exponentially-weighted-average-5eed00181a09

Bias Correction in Exponentially Weighted Averages

$$W_t = \begin{cases} 0 & t = 0 \\ \beta \cdot W_{t-1} + (1-\beta) \cdot \theta_t & t > 0 \end{cases}$$



$$W_0 = 0$$

$$\begin{aligned} W_1 &= 0.98W_0 + 0.02\theta_1 \\ &= 0.98 * 0 + 0.02\theta_1 \\ &= 0.02\theta_1 \end{aligned}$$

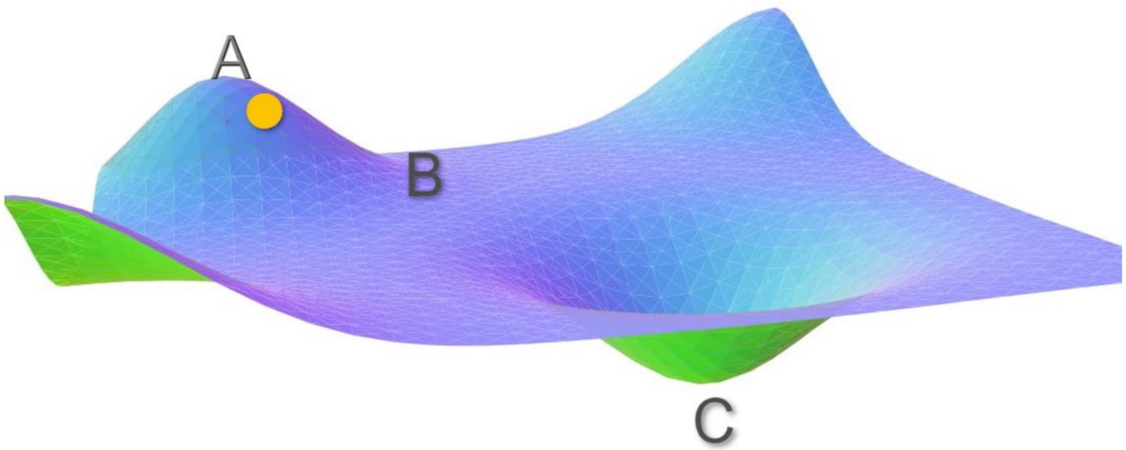
$$\begin{aligned} W_2 &= 0.98 * W_1 + 0.02*\theta_2 \\ &= 0.98 * 0.02\theta_1 + 0.02\theta_2 \\ &= 0.0196\theta_1 + 0.02\theta_2 \end{aligned}$$

$$\begin{aligned} W_2 &= (0.0196\theta_1 + 0.02\theta_2)/(1-0.98^2) \\ &= (0.0196\theta_1 + 0.02\theta_2)/0.0396 \end{aligned}$$

Bias correction: $\frac{W_t}{1-\beta^t}$

When **t** is bigger, $1-\beta^t \approx 1$

Gradient Descent with Momentum



Problem with gradient descent:

1. Saddle points lead to small or no weight updates. Learning stops
2. Weight update doesn't take into account past steps
3. The path followed by Gradient Descent is very jittery

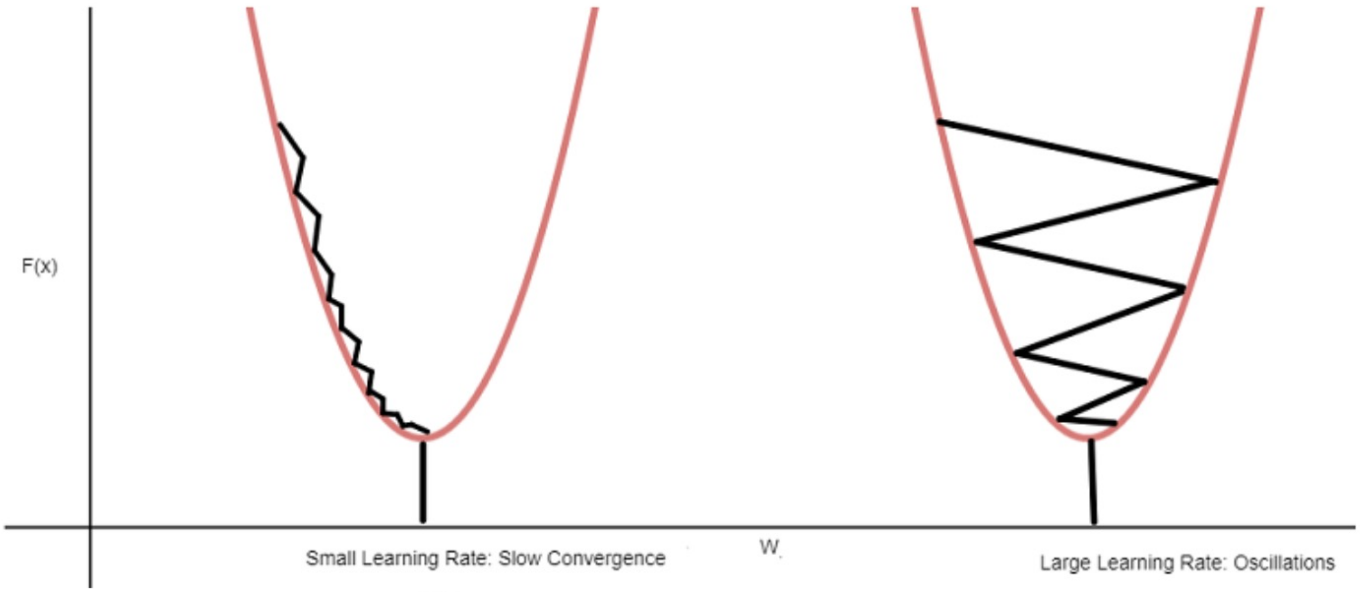
Solutions: We can gather **momentum** by taking a moving average over the past gradients. By using an Exponential Moving Average we can assign greater weight on the most recent values.

$$v(n) = (1-\beta)\sum_{t=1}^{n} \beta^{n-t}\delta(t)$$

Don't forget to correct the bias

https://towardsdatascience.com/gradient-descent-with-momentum-59420f626c8f

Gradient Descent with Momentum

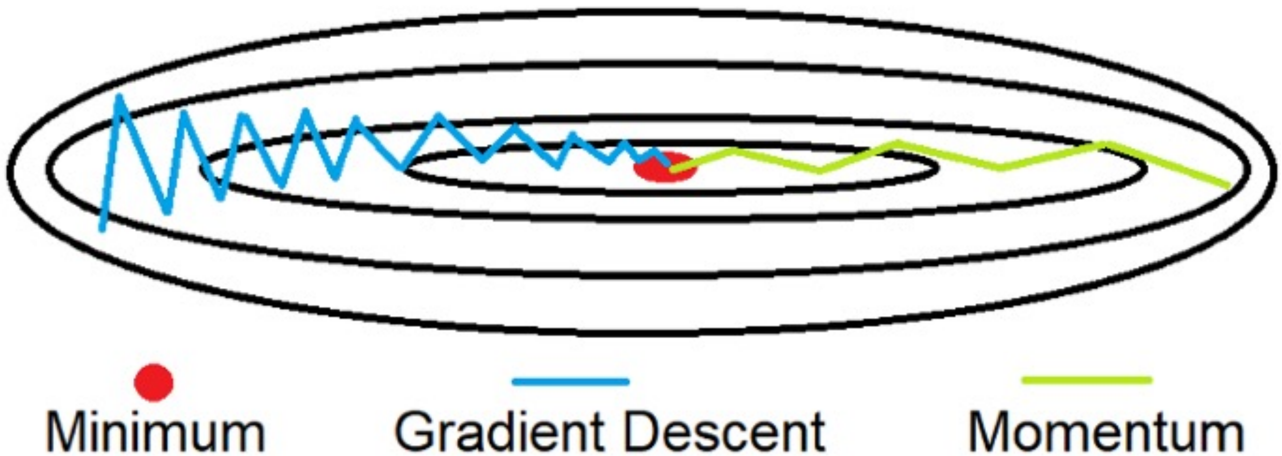


$$v(n) = (1-\beta)\sum_{t=1}^{n}\beta^{n-t}\delta(t)$$

Often $(1-\beta)$ is replaced by learning rate. When all the past gradients have the same sign, the summation term will become large

https://towardsdatascience.com/gradient-descent-with-momentum-59420f626c8f

Intuitive of Gradient Descent with Momentum



On iteration t:

Compute dW, db on the current mini-batch

$$v_{dW} = \beta \boxed{v_{dW}} + \boxed{(1-\beta) dW}$$

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

$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

Velocity

Acceleration

Conclusion

- Pros:
- Gradients accumulated from past iterations will push the cost further to move around a saddle point even when the current gradient is negligible or zero.
- Cons:
- The hyperparameter α (learning rate) has to be tuned manually.
- In some cases, where, even if the learning rate is low, the momentum term and the current gradient can alone drive and cause oscillations.
- Solutions:
- AdaptiveGradient and RMSprop can be used to solve the Learning rate problem
- A large momentum problem can be further resolved by using Nesterov Accelerated Gradient Descent.

- Source:
- https://towardsdatascience.com/gradient-descent-with-momentum-59420f626c8f

Adaptive Gradient (Adagrad)

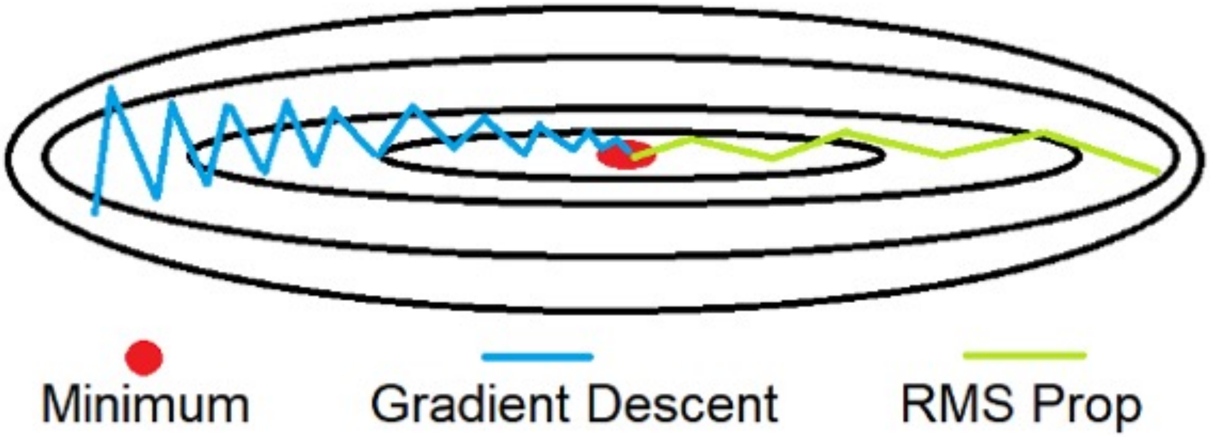
- SGD or SGD + Momentum, learning rate α is constant value.
- $W_t = W_t - \alpha dW_t$

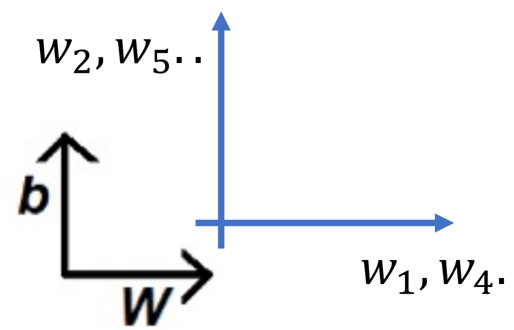
- Math for Adaptive Gradient:
- $W_t = W_t - \alpha' dW_t$
- $\alpha' = \frac{\alpha}{\sqrt{\gamma_t + \varepsilon}}$ $where\ \gamma_t = \sum_{i=1}^{t} dW_i^2$
- When iteration increase, γ_t will increase, α' will drop. Learning rate is different for each iteration.

Pros and Cons for Adagrad

- Pros:
- The sparse features will have a higher learning rate, while dense features will have lower learning rate
- Do not need tune the learning rate manually
- Cons:
- As number of iterations goes up, the learning will become slower, because the *sum of gradient squared* only grows and never shrinks.

Root-Mean-Square Propagation (RMS Prop)





Exponentially weighted average of square of derivatives

Elementwise square

$$s_{dw} = \beta S_{dw} + (1-\beta)dw^2 \qquad s_{db} = \beta S_{db} + (1-\beta)db^2$$

$$w = w - \underbrace{\alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}} \quad \alpha' \qquad b = b - \alpha \frac{db}{\sqrt{S_{db}}}$$

Small, speed up update on w direction

Large, slow down the update on b direction

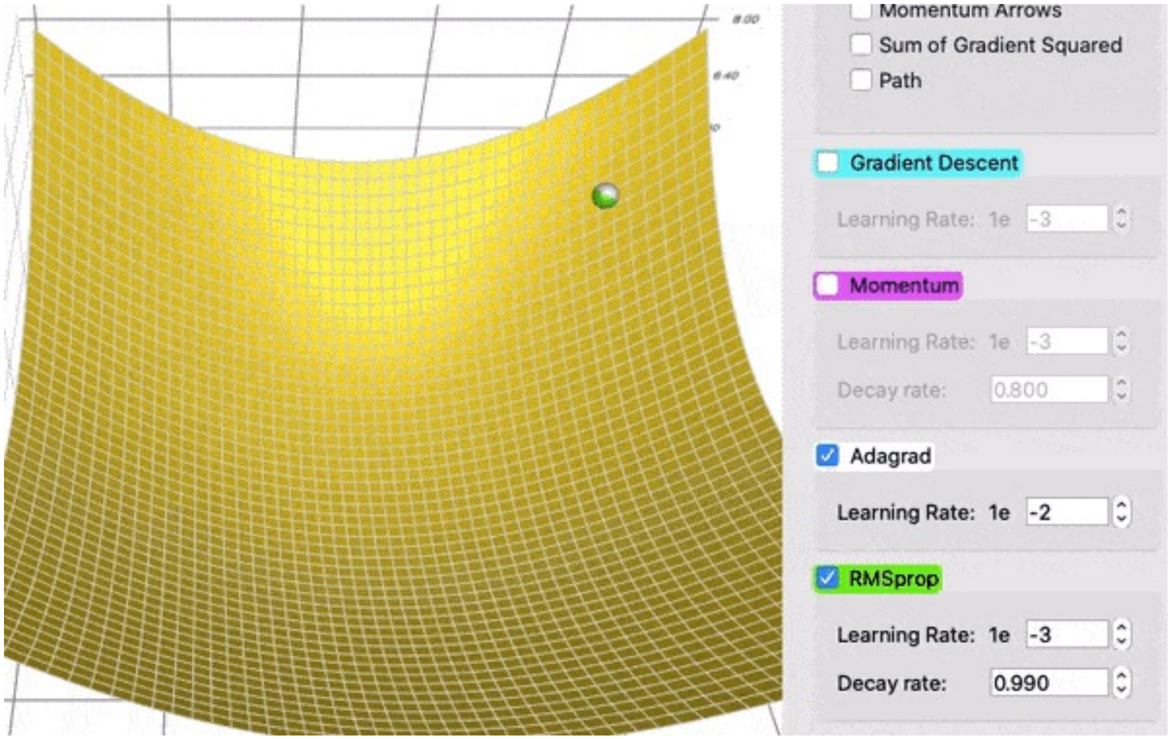
Root-Mean-Square Propagation (RMS Prop)

- $s_{dw} = \beta S_{dw} + (1-\beta)dw^2$ β *is the decay rate*
- *sum_of_gradient_squared = previous_sum_of_gradient_squared* **** decay_rate****+ gradient*^{2} **** (1- decay_rate)***
- The sum of gradient squared is actually the *decayed* sum of gradient squared.
- The recent dw^2 matters the most, the one from long ago are basically forgotten.
- The decay rate have two effects: a) decaying, b)Scaling effect. Scale down the whole term by $(1-\beta)$, the step is on the order of $\frac{1}{\sqrt{(1-\beta)}}$ larger for α

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

$$w = w - \alpha\frac{dw}{\sqrt{S_{dw}} + \varepsilon} \quad a'$$

RMSprop vs AdaGrad



RMSProp (green) vs AdaGrad (white). The first run just shows the balls; the second run also shows the sum of gradient squared represented by the squares.

https://towardsdatascience.com/a-visual-explanation-of-gradient-descent-methods-momentum-adagrad-rmsprop-adam-f898b102325c

Adaptive Moment Estimation (Adam)

- It combines the Momentum and RMS prop in a single approach.

$v_0 = 0, s_0 = 0$

$v_{t+1} = \beta_1 v_t + (1 - \beta_1) d\theta$ Momentum

$s_{t+1} = \beta_2 S_t + (1 - \beta_2) d\theta^2$ RMS Prop

$v_{t+1} = \frac{v_{t+1}}{1-\beta_1^t}, s_{t+1} = \frac{s_{t+1}}{1-\beta_2^t}$ Bias correction

$\theta_j = \theta_j - \frac{\alpha}{\sqrt{s_{t+1}}+\varepsilon} v_{t+1}$ Momentum + RMA Prop

Hyper-parameters in Adam optimizer

- Learning rate α
- β_1 default 0.9
- β_2 default 0.999
- ε default 1e-8

Pros and Cons

- **Advantages**:
- The method converges rapidly.
- Rectifies vanishing learning rate, high variance.
- **Disadvantages**:
- Computationally expensive.

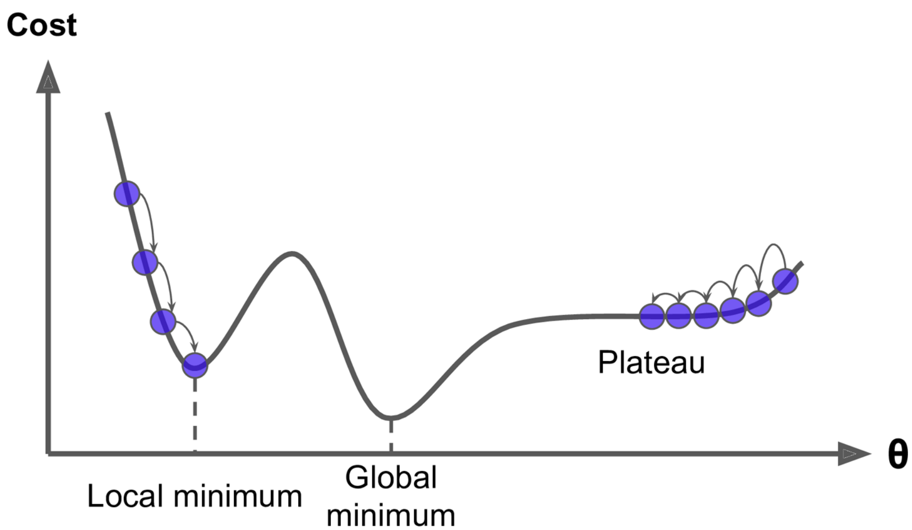
Learning rate decay

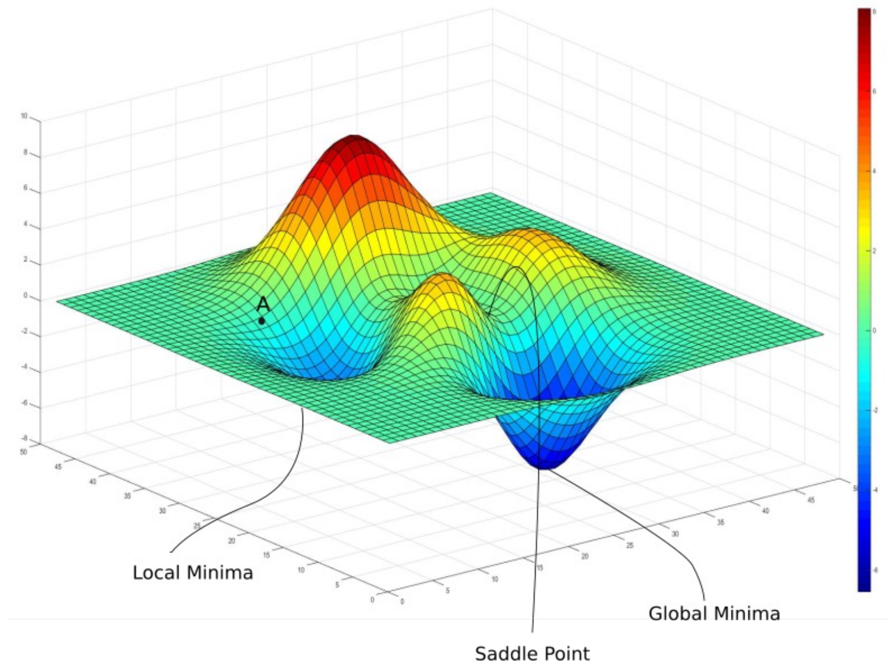
- ***Higher*** *the learning rate:*
  - The model converges quickly .
  - Higher risk of missing the optimal solution, or fail to converge.
- ***Lower*** *the learning rate:*
  - Higher accuracy.
  - Models converges very slow.
- 1 epoch means one pass through data. We can set learning rate decay as epoch number grows.
- $\alpha = \frac{1}{1+decay_{rate}*epoch\ \#}\alpha_0$
- $\alpha = 0.95^{epoch\#}\alpha_0$ Exponential decay

Implement exponential decay in Tensorflow

- ***tf.train.exponential_decay(learning_rate, global_step, decay_steps, decay_rate, staircase=False, name=None)***
  - **learning_rate:** Starting learning rate.
  - **global_step:** The successive learning step. The number of iterations (Training steps) since the beginning of training. We need to increment it by 1 every iteration, so the library can slowly decrease learning rate according to where in training phase we are.
  - **decay_steps**: The number of steps of different learning rates we want to have. *see picture_2*
  - **decay_rate:** How fast the training rate should drop. Ranges from **0.0** to **1.0** **bigger** the number, **faster** the rate decreases
  - **staircase:** Tells if we prefer to have decay in intervals (staircase=True), or that we prefer smooth line (staircase = False)
  - **Name**: Optional name for our operation

Local optima, saddle points and plateau





- In low dimensional, local minima are common
- In high dimensional, local minima are rare, saddle points are common
- It is exponentially unlikely to get stuck in a bad local minima (The Hessian have the same sign in high dim)
- Plateaus can make learning very slow.

https://www.oreilly.com/library/view/hands-on-machine-learning/9781491962282/ch04.html