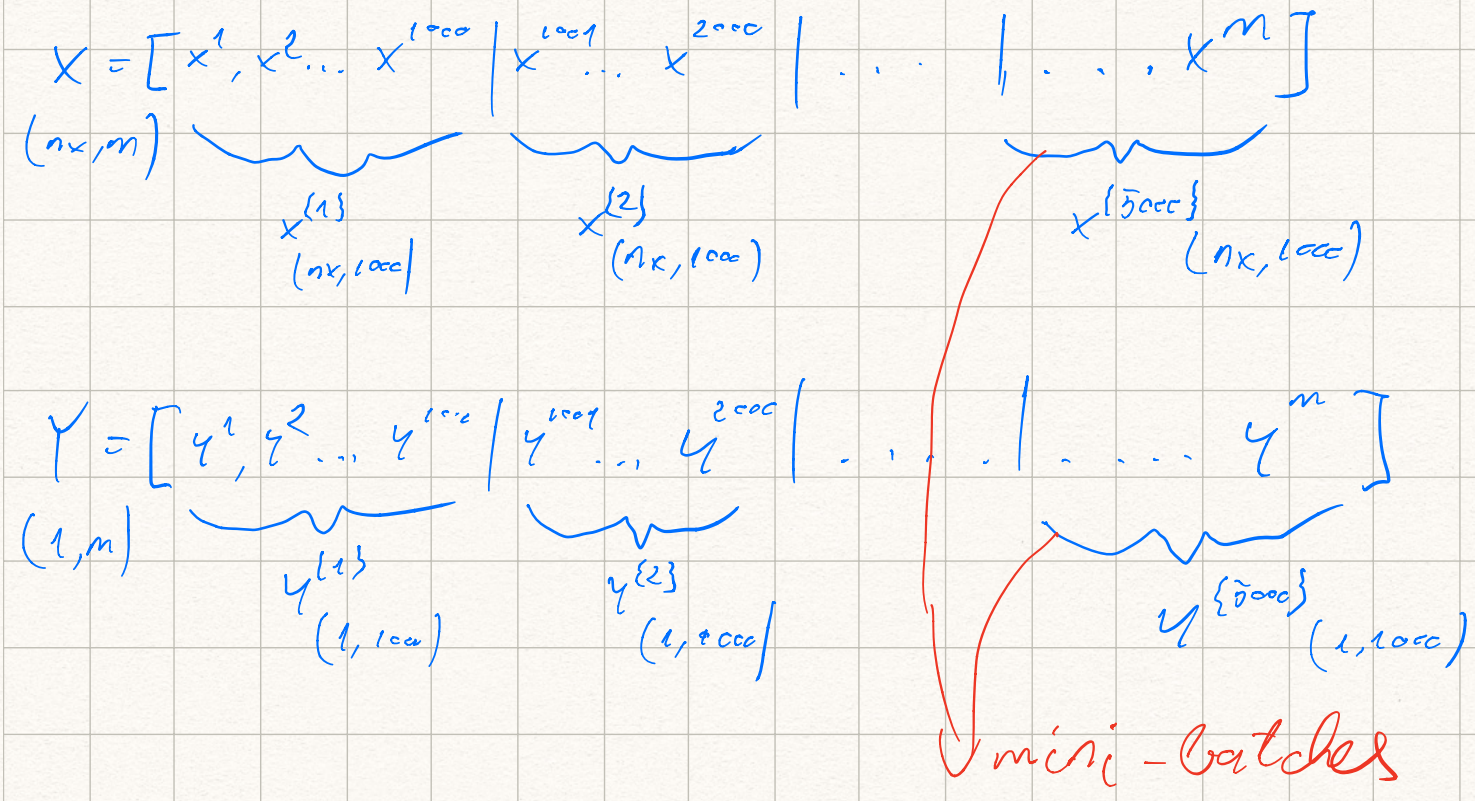
mini-batch gradient descent

Vectorization allowst to efficiently compute on m examples
=> however m is big => it can be slow



$m = 5000\,000$ => 5000 mini-batches each of 1000 examples

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

One step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$ mini-batches:

Repeat until concergency {

  for t = 1 ... 5000 {

    Forward propagation on $X^{\{t\}}$

$$z^1 = w^1 x^{\{t\}} + b^1$$
$$a^1 = f^1(z^1)$$
$$\vdots$$
$$a^l = f^l(z^l)$$

} use vectorized implementation

compute cost: $\sqrt{}$ for $x^{\{t\}}, y^{\{t\}}$

$$J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{1000} \mathcal{L}(\hat{y}^i, y^i) + \frac{\lambda}{2 \cdot 1000} \sum_l \|w^l\|_F^2$$

Backprop to compute gradient cost $J^{\{t\}}$

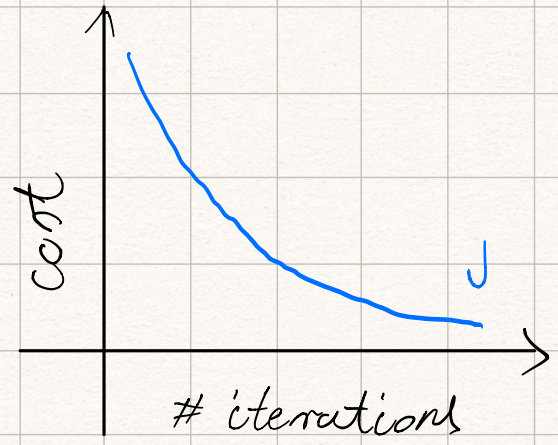
$$w^l := w^l - \alpha \, dw^l \qquad b^l := b^l - \alpha \, db^l$$

} -> 1 epoch (single pass through the training set)

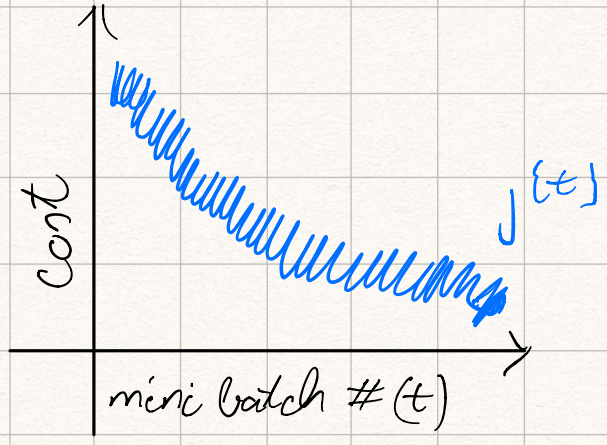
}

Training with mini-batch gradient descent

Batch gradient descent



Mini-batch gradient descent



$X^{\{1\}}, Y^{\{1\}}$ mini batch may be harder than $X^{\{2\}}, Y^{\{2\}}$ =>

=> oscillation

Choosing mini-batch size

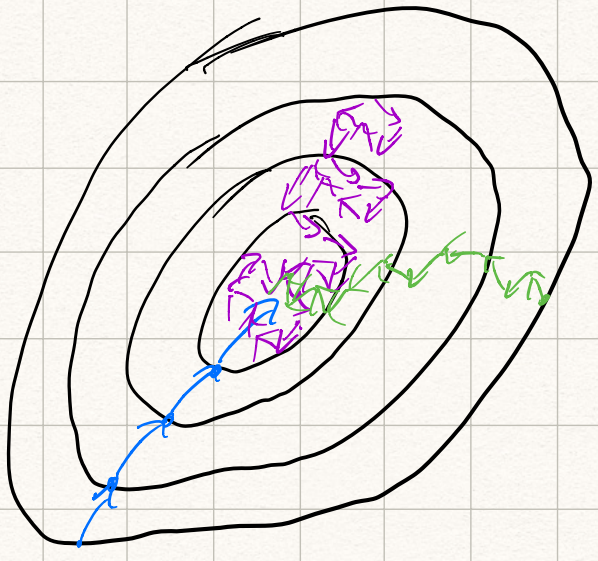
If mini-batch size = m => Batch gradient descent

If mini-batch size = 1 => Stochastic gradient descent

In practice => $1 \leq \text{size} \leq m$

SGD: lose speedup from vectorization

BGD: Too long per iteration



- Fastest learning from vectorization
- make progress without process the entire dataset

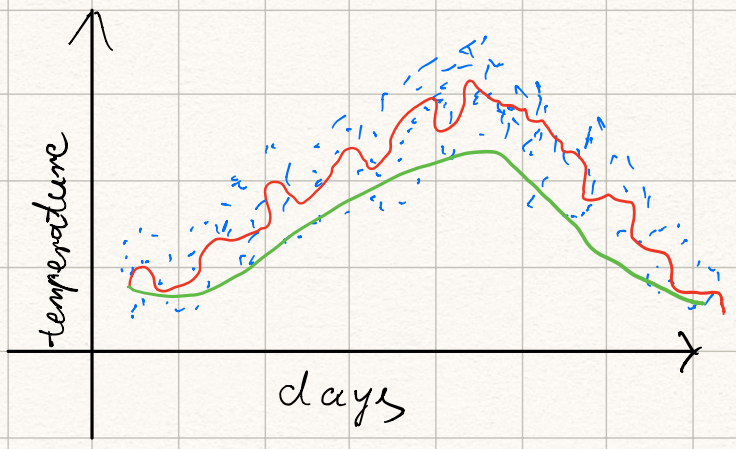
If small training set => use batch gradient descent

else => use mini-batch

with size: 64, 128, 256, 512

Make sure mini-batch fit in CPU/GPU memory

Exponentially weighted averages



$v_0 = 0$

$v_1 = 0.9 v_0 + 0.1 \theta_1$

$v_2 = 0.9 v_1 + 0.1 \theta_2$

$\vdots$

$v_t = 0.9 v_{t-1} + 0.1 \theta_t$

hyperparameter

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t \qquad \approx \frac{1}{1-\beta} \text{ days}$$

$\beta = 0.9 \Rightarrow$ 10 days average

$\beta = 0.98 \Rightarrow$ 50 days average

math stuff

$v_{100} = 0.9 v_{99} + 0.1 \theta_{100}$

$v_{99} = 0.9 v_{98} + 0.1 \theta_{99}$

$v_{98} = 0.9 v_{97} + 0.1 \theta_{98}$

$\vdots$

$v_{100} = 0.1 \theta_{100} + 0.9 v_{99}$

$0.1 \theta_{99} + 0.9 v_{98}$

$0.1 \theta_{98} + 0.9 v_{97}$

$$= 0.1\theta_{100} + 0.1 \cdot 0.9 \theta_{99} + 0.1 \cdot (0.9)^2 \theta_{98} + 0.1 \cdot (0.9)^3 \theta_{97} + \dots$$

Implementing

$v_\theta = 0$ ⇐ only one variable to store

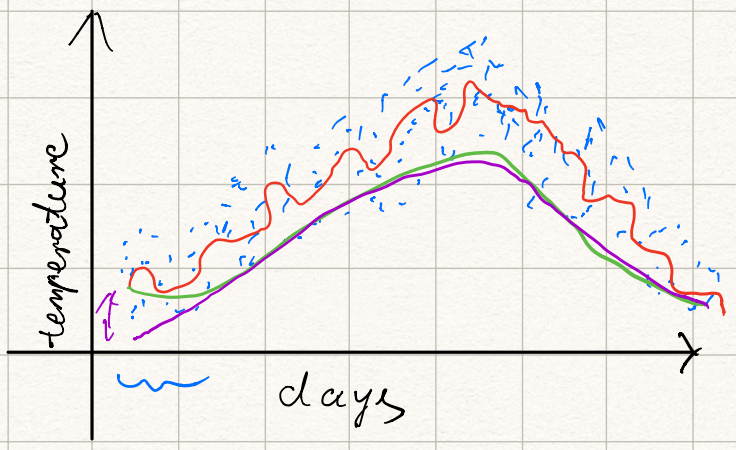
Repeat {

get next θ_t

$v_\theta := \beta v_\theta + (1-\beta)\theta_t$

}

Bias correction in exponentially weighted average



$v_t = \beta v_{t-1} + (1-\beta)\theta_t$

=> actually we get the purple curve

$v_0 = 0$

$v_1 = 0.98 v_0 + 0.02\theta_1$ => v_1 = much lower value than the valid one

$v_2 = 0.98 v_1 + 0.02\theta_2 = 0.98 \cdot 0.02\theta_1 + 0.02\theta_2$ =>

$\frac{v_t}{1-\beta^t}$ => $t = 2 : 1-\beta^t = 1 - 0.98^2 = 0.0396$

$\frac{v_2}{0.0396} = \frac{0.0196\theta_1 + 0.02\theta_2}{0.0396}$ => removing bias

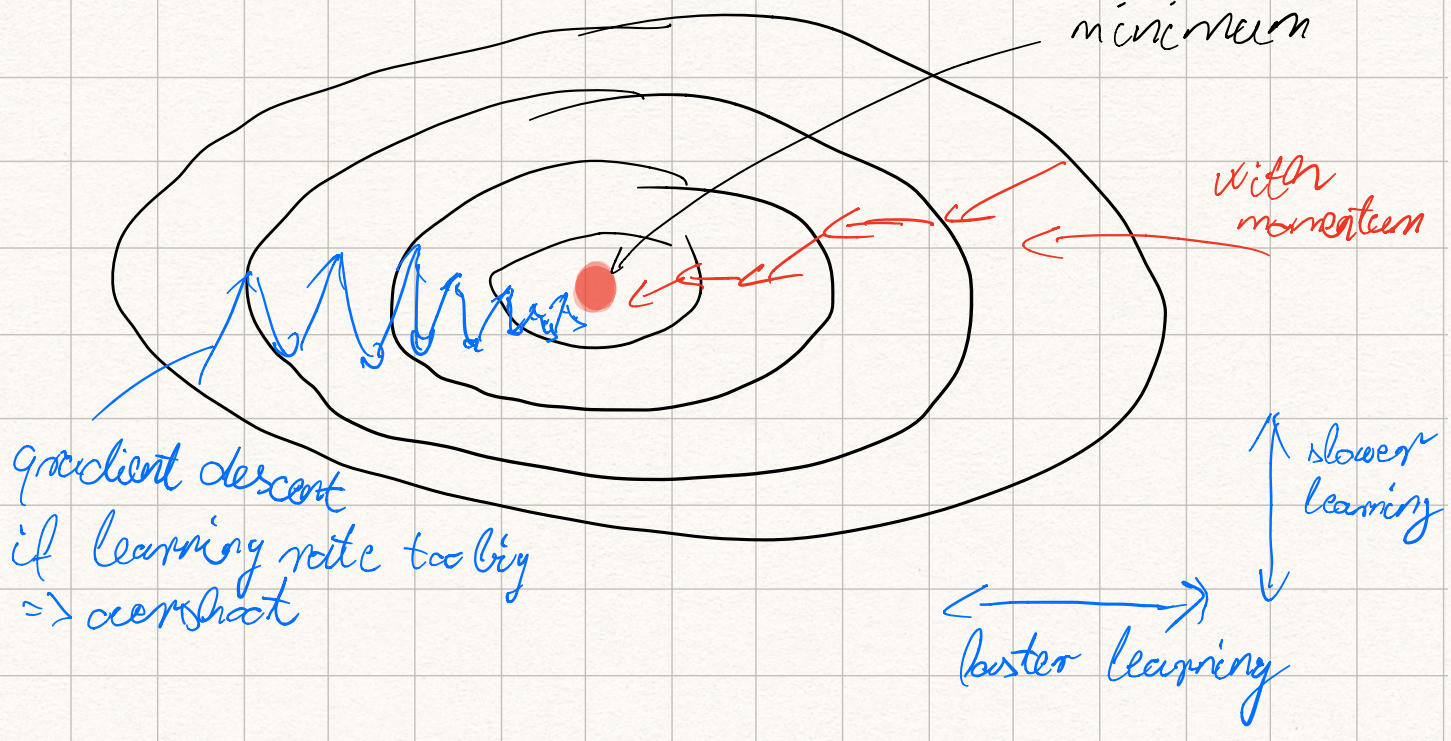
if t is large => $\beta^t \approx 0$ => no effect to bias

=> purple and green line are pretty overlap

=> using exponentially weighted average to build better optimizer

Gradient descent with momentum

↳ use exponentially weighted average to compute derivatives
↳ almost always faster than gradient descent



Momentum: friction velocity

iterate t:

compute dw, db on current mini-batch

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

=> smooth out gradient descent

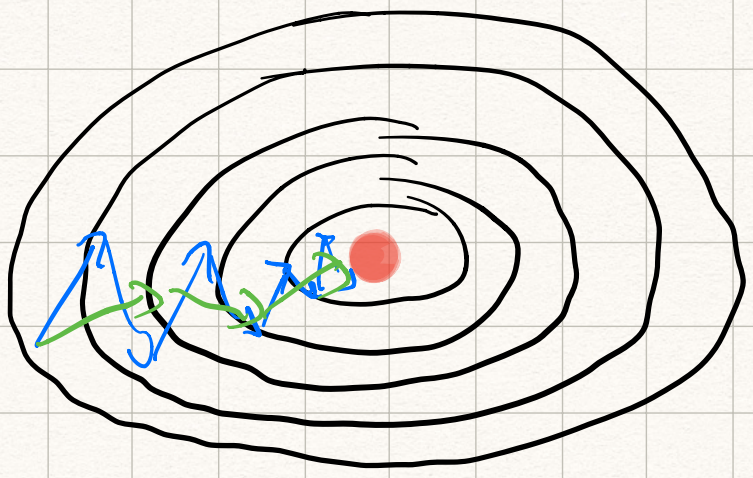
$$w := w - \hat{\alpha} v_{dw}, \quad b := b - \hat{\alpha} v_{db}$$

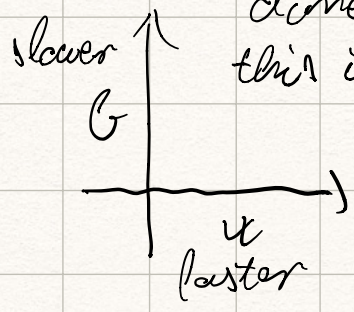
acceleration

Hyperparameters: $\hat{\alpha}, \beta$ → usually $\beta = 0.9$

≈ last 10 example average

RMSprop (root mean square propagation)





w, b are high dimensional parameters this is just an intuition

iterate t:

compute dw, db on current mini-batch (element wise)

$$Sdw = \beta_2 Sdw + (1-\beta_2)\, dw^2 \quad \leftarrow \text{small}$$

$$Sdb = \beta_2 Sdb + (1-\beta_2)\, db^2 \quad \leftarrow \text{large}$$

$$w := w - \alpha \frac{dw}{\sqrt{Sdw} + \varepsilon}, \qquad b := b - \alpha \frac{db}{\sqrt{Sdb} + \varepsilon}$$

$\varepsilon = 10^{-8}$ small number

=> in order to do not devide by zero

Adam optimization algorithm

↳ work well on wide range of architectures

→ use momentum and RMSprop

$Vdw = 0$, $Sdw = 0$, $Vdb = 0$, $Sdb = 0$

iterate t:

compute dw, db on current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)\, dw$$
$$V_{db} = \beta_1 V_{db} + (1-\beta_1)\, db$$
} momentum like

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\, dw^2$$
$$S_{db} = \beta_2 S_{dw} + (1-\beta_2)\, db^2$$
} RMSprop like

$$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t)$$
$$V_{db}^{corrected} = V_{db} / (1-\beta_1^t)$$
} bias correction

$$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t)$$
$$S_{db}^{corrected} = S_{db} / (1-\beta_2^t)$$
} bias correction

$$w := w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon} \quad , \quad b := b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

=> combine the effect of gradient descent with momentum and with RMSprop

↳ we can use larger learning rate

Hyperparameters choice:

α : needs to be tune

β_1 : ~ 0.9

β_2 : ~ 0.999

ε : 10^{-8}

Adam => Adaptive moment estimation

learning rate decay

Noisy mini-batch gradient descent -> never convergence, just oscillate around the minimum

=> learning rate decay =>

take smaller steps => better convergence

other hyperparameter

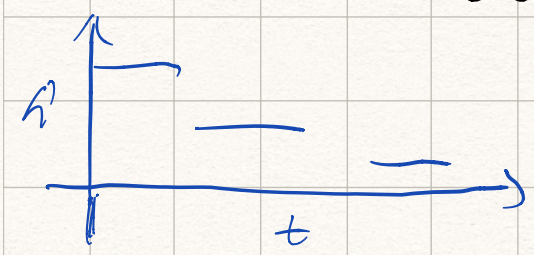
$$\alpha = \frac{1}{1 + \text{decay\_rate} \cdot \text{epoch\_num}} \cdot \alpha_0$$

$$\alpha = 0.95^{\text{epoch\_num}} \cdot \alpha_0 \quad \Rightarrow \text{exponentially decay}$$

$$\alpha = \frac{k}{\sqrt{\text{epoch\_num}}} \cdot \alpha_0$$

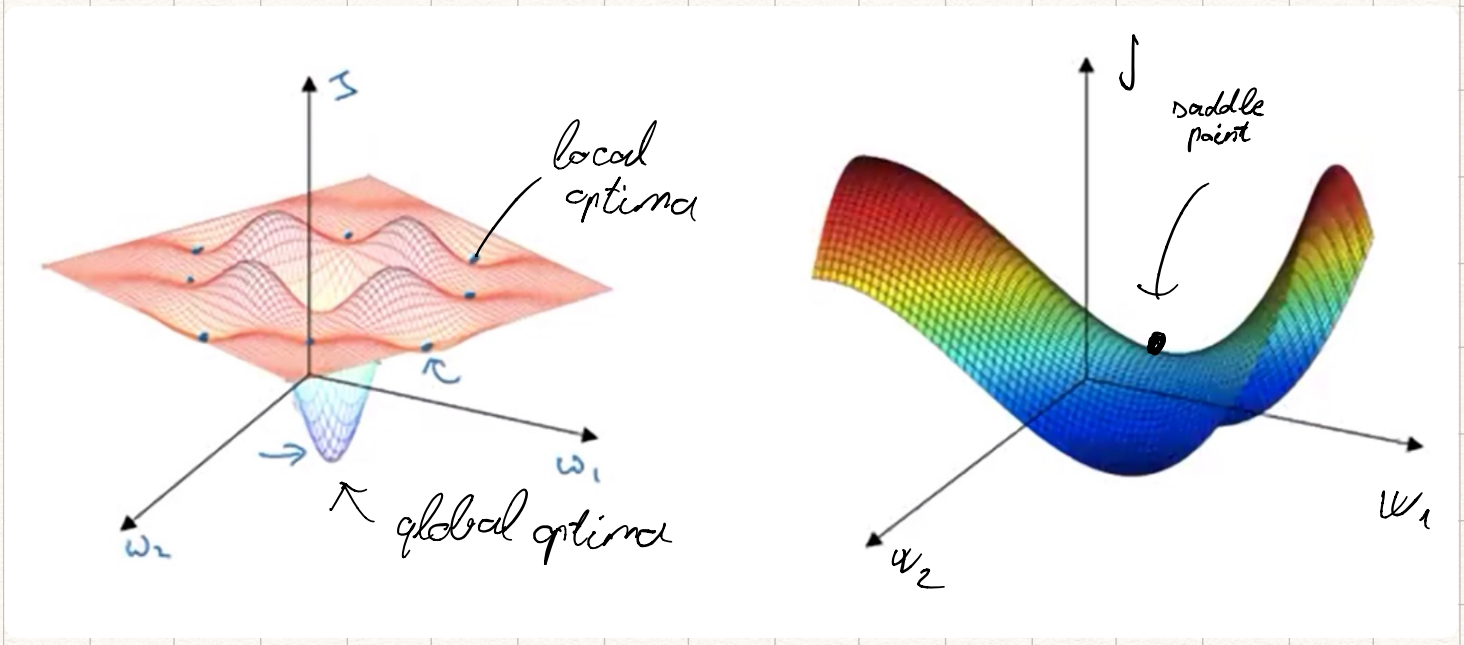
$$\alpha = \frac{k}{\sqrt{t}} \cdot \alpha_0$$

t: mini batch number



=> discrete decay

The problem of local optima

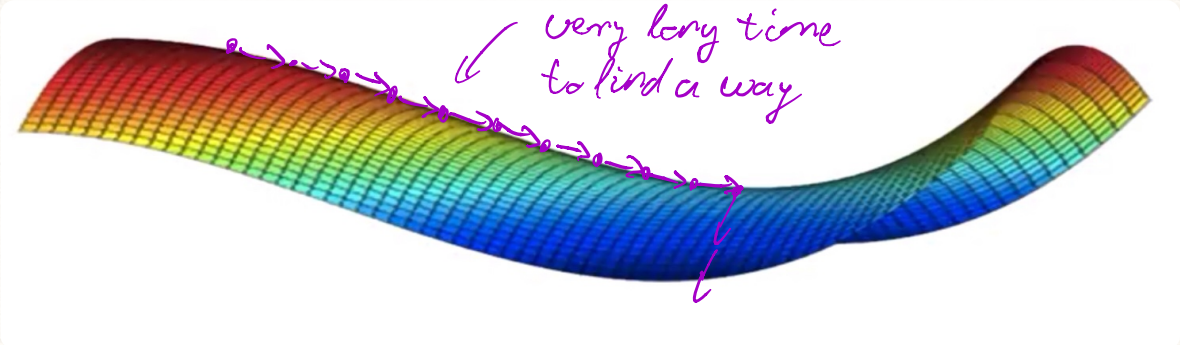


In two dimensions easy to create local optima

- but in high dimension space most points with Zero gradient it is a saddle point
- to be a local optima in each direction have to be a convex or concave curve

=> high dim $\approx$ 20000 => $p = 2^{-20000}$ the chance to be a local optima

Plateaus problem (derivative is close to zero for a long time)



unlikely to get stuck in a bad local optima => use large NN with many parameters

Plateaus can make learning slow => momentum, Adam