



deeplearning.ai

Optimization Algorithms

Mini-batch gradient descent

- Speeds up GD because it allows network updates even before having processed the whole dataset.
- Breakdown the DB in several non-overlapping minibatches. Perform GD for each minibatch in the dataset (1 epoch = over entire DB) and repeat until convergence.

Batch vs. mini-batch gradient descent

x, y

$x^{\{t\}}, y^{\{t\}}$

Vectorization allows you to efficiently compute on m examples.

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(1000)} & | & x^{(1001)} & \dots & x^{(2000)} & | & \dots & | & \dots & x^{(m)} \end{bmatrix}$$

(n_x, m)

$X^{\{1\}} (n_x, 1000)$ $X^{\{2\}} (n_x, 1000)$ $X^{\{5,000\}} (n_x, 1000)$

CURLY BRACES = minibatch number

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \dots & y^{(1000)} & | & y^{(1001)} & \dots & y^{(2000)} & | & \dots & | & \dots & y^{(m)} \end{bmatrix}$$

$(1, m)$

$Y^{\{1\}} (1, 1000)$ $Y^{\{2\}} (1, 1000)$ $Y^{\{5,000\}} (1, 1000)$

What if $m = \underline{5,000,000}$?

5,000 mini-batches of 1,000 each

Mini-batch t : $x^{\{t\}}, y^{\{t\}}$

$$\begin{array}{c} x^{(i)} \\ z^{[l]} \\ x^{\{t\}}, y^{\{t\}} \end{array}$$

Mini-batch gradient descent

repeat {
for $t = 1, \dots, 5000$ {

Forward prop on $X^{\{t\}}$.

$$Z^{\{t\}} = W^{\{t\}} X^{\{t\}} + b^{\{t\}}$$

$$A^{\{t\}} = g^{\{t\}}(Z^{\{t\}})$$

⋮

$$A^{\{t\}} = g^{\{t\}}(Z^{\{t\}})$$

Vectorized implementation
(1000 examples)

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{m^{\{t\}}} \ell(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l \leftarrow \text{layers}} \|W^{(l)}\|_F^2$

samples in minibatch
 $m^{\{t\}}$
from $X^{\{t\}}, Y^{\{t\}}$

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

$$W^{\{t\}} := W^{\{t\}} - \alpha dW^{\{t\}}, \quad b^{\{t\}} := b^{\{t\}} - \alpha db^{\{t\}}$$

}

}

"1 epoch"

pass through training set.

1 step of grad desc
using $X^{\{t\}}, Y^{\{t\}}$.
(as if $m=1000$)

X, Y



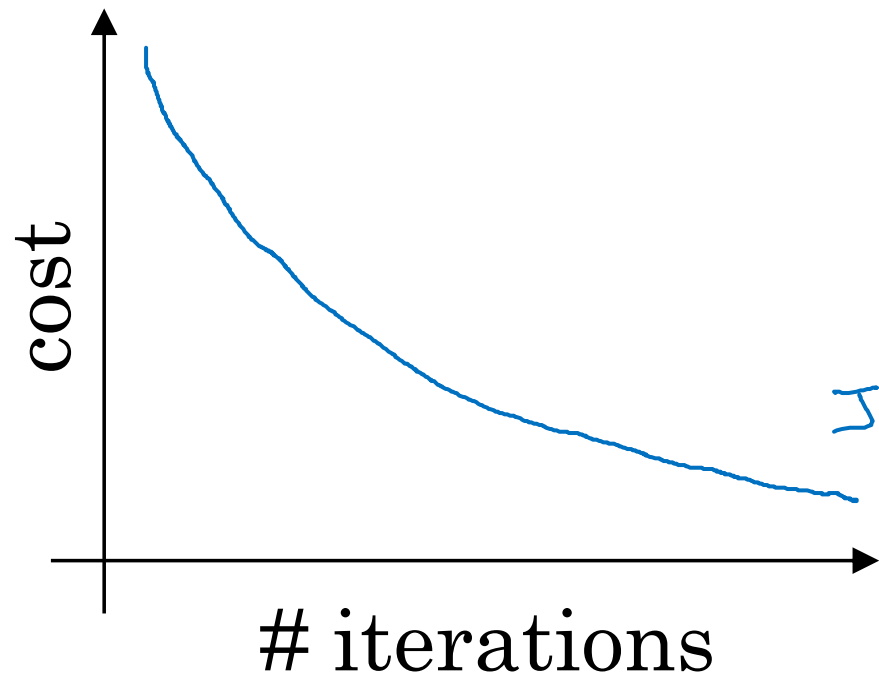
deeplearning.ai

Optimization Algorithms

Understanding
mini-batch
gradient descent

Training with mini batch gradient descent

Batch gradient descent

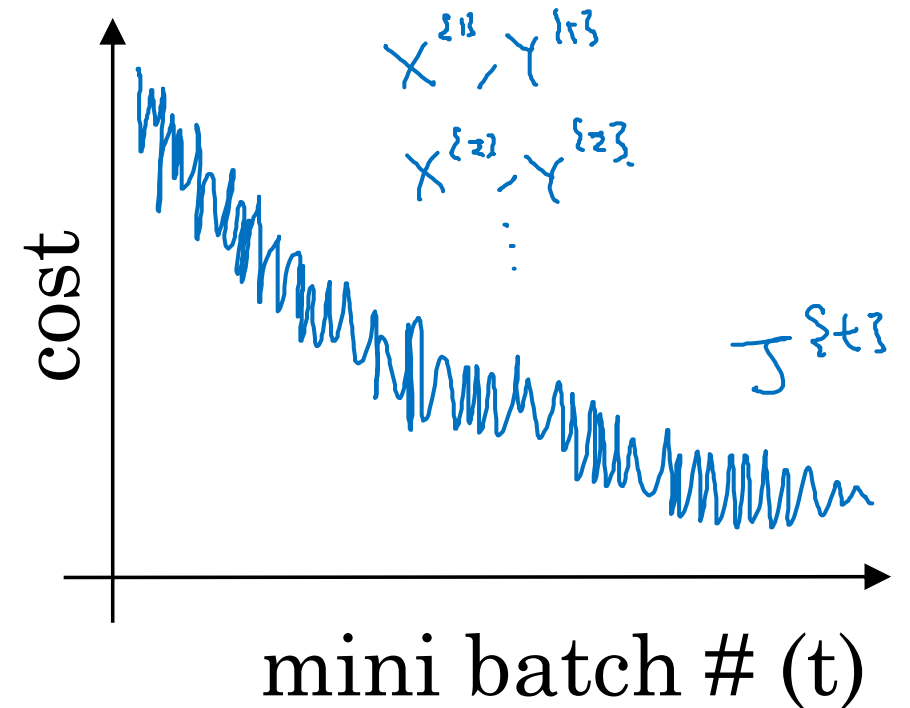


$$V_1 = 0.5 V_0 + 0.5 \cdot 30 = 15$$

$$V_2 = 0.5 \cdot 15 + 0.5 \cdot 15 = 15$$

conv
 $15/0.5 =$

Mini-batch gradient descent



Plot $J^{(t)}$ computed using $X^{(t)}, Y^{(t)}$

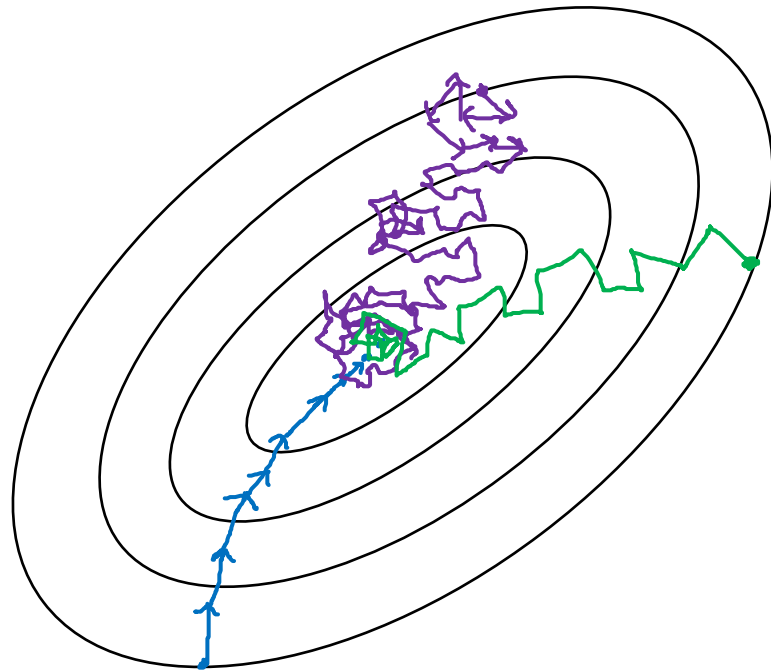
Choosing your mini-batch size

→ If mini-batch size = m : "Batch" gradient descent.

→ If mini-batch size = 1 : "Stochastic" gradient descent. Every example is its own mini-batch.

In practice: Somewhere in-between 1 and m

- if minibatch size = 1 \Rightarrow SGD : disadvantage of ~~both~~ vectorization.
 - if minibatch size = $m \Rightarrow$ batch GD : disadvantage need to observe the whole dataset before updating the model.
- $(X^{(1)}, Y^{(1)}) = (X, Y)$



Stochastic
gradient
descent

↓
Use ~~speed~~ ^{speedup}
from vectorization

In-between
(mini-batch size
not too big/small)

↓
Fastest learning.

- Vectorization.
($n=1000$)
- Make passes without
processing entire training set.

Batch
gradient descent
(mini-batch size = m)

↓
Too long
per iteration

Choosing your mini-batch size

- If small toy set: Use batch gradient descent. \rightarrow Important note
($m \leq 2000$)

- Typical mini-batch sizes: powers of 2

$$\rightarrow \underbrace{64, 128, 256, 512}_{\substack{2^6 \quad 2^7 \quad 2^8 \quad 2^9}} \quad \frac{1024}{2^{10}}$$

- TIP: Make sure mini-batch fit in CPU/GPU memory.
 $X^{(t)}, Y^{(t)}$



deeplearning.ai

Optimization Algorithms

Exponentially moving weighted averages

→ At the basis of optimization algorithms faster than GD.

Temperature in London

$$\theta_1 = 40^\circ\text{F} \quad 4^\circ\text{C} \quad \leftarrow$$

$$\theta_2 = 49^\circ\text{F} \quad 9^\circ\text{C}$$

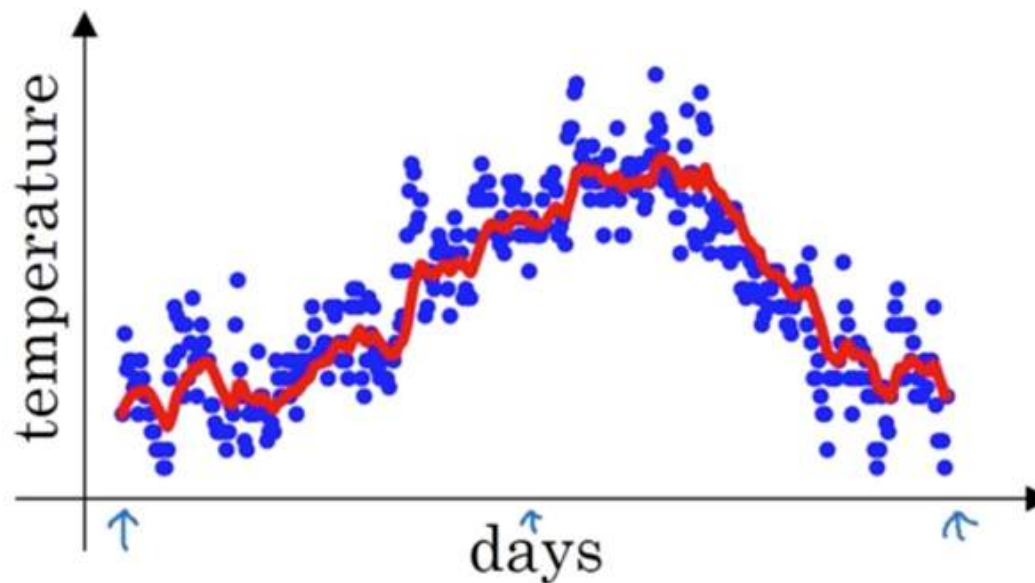
$$\theta_3 = 45^\circ\text{F} \quad \vdots$$

\vdots

$$\theta_{180} = 60^\circ\text{F} \quad 15^\circ\text{C}$$

$$\theta_{181} = 56^\circ\text{F} \quad \vdots$$

\vdots



$$V_0 = 0$$

$$V_1 = 0.9 V_0 + 0.1 \theta_1$$

$$V_2 = 0.9 V_1 + 0.1 \theta_2$$

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

\vdots

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

Exponentially weighted averages ^{moving}

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

$\beta = 0.9$: averaging over ≈ 10 days' temperature.

$\beta = 0.98$: averaging over ≈ 50 days

$\beta = 0.5$: ≈ 2 days

V_t is approximately

average over

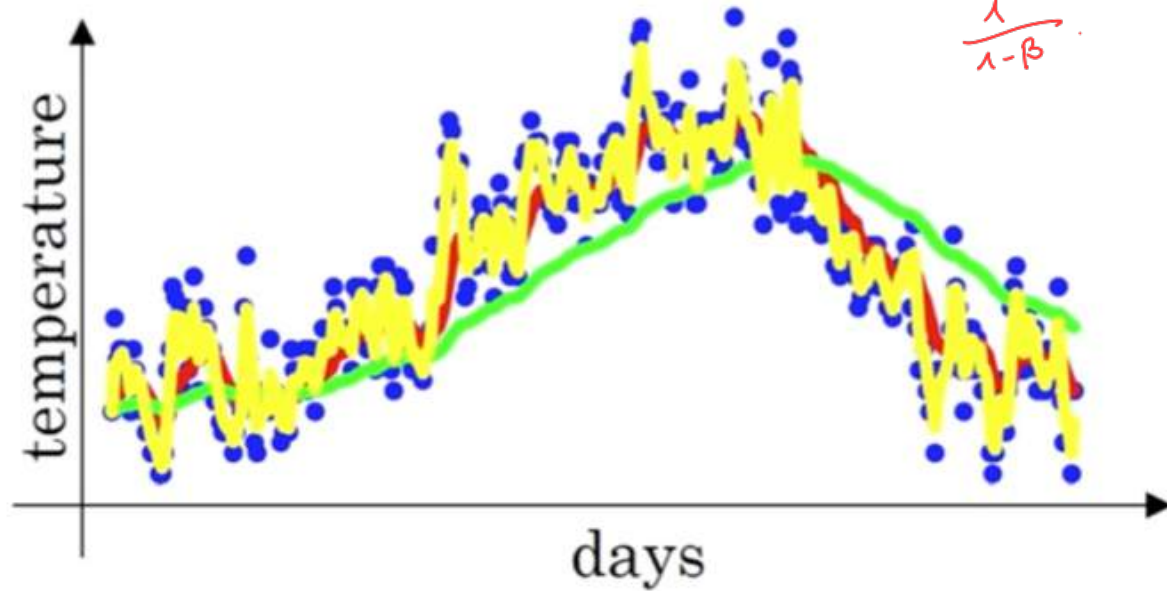
$\approx \frac{1}{1-\beta}$ days' temperature.

exponential averaging with forgetting $(1-\beta)$ has an exponential decay of $\frac{1}{1-\beta}$.

$$\frac{1}{1-0.98} = 50$$

The decay happens when the weight decays to $\frac{1}{e} \approx 0.35$.

For $0.9 \rightarrow 0.9^x \approx 0.35 \rightarrow x = \frac{\log 0.35}{\log 0.9} \approx 40$
 $0.98^x \approx 0.35 \rightarrow x = 50$





deeplearning.ai

Optimization Algorithms

Understanding
exponentially
weighted averages

Exponentially weighted averages

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

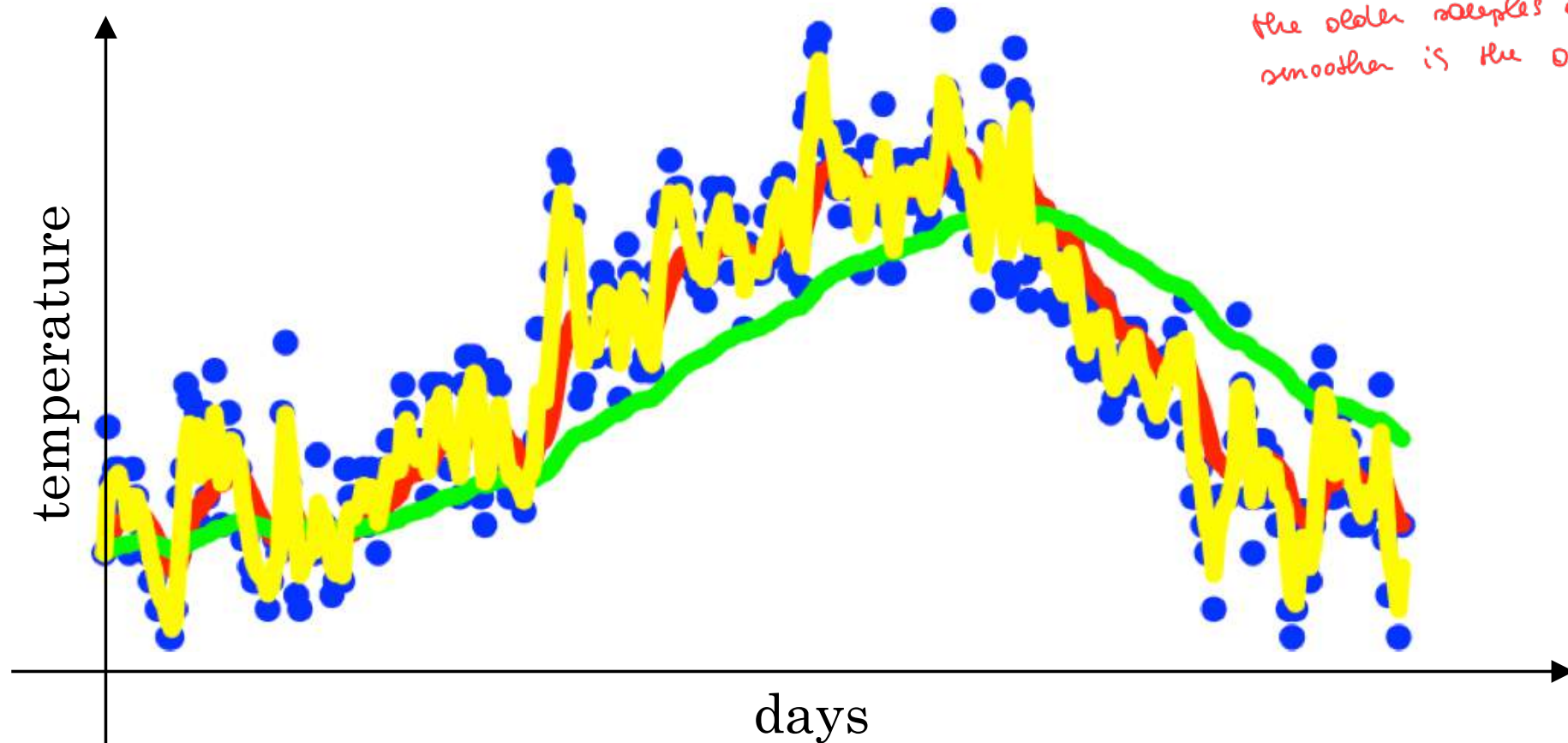
$$\beta = 0.9$$

$$0.98$$

$$0.5$$

$$\beta = \text{MEMORY}$$

the higher, the more influential the older samples are, the smoother is the average.



Exponentially weighted averages

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

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

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

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

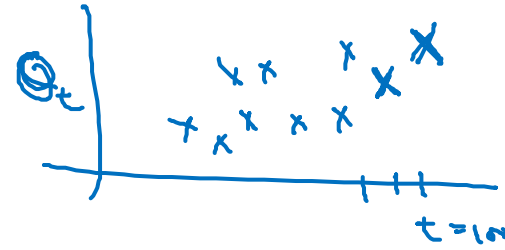
...

$$\begin{aligned} \rightarrow v_{100} &= 0.1\theta_{100} + 0.9 \cancel{v_{99}} (0.1\theta_{99} + 0.9 \cancel{v_{98}}) \\ &= \underbrace{0.1\theta_{100}} + \underbrace{0.1 \times 0.9 \cdot \theta_{99}} + \underbrace{0.1 (0.9)^2 \theta_{98}} + \underbrace{0.1 (0.9)^3 \theta_{97}} + \underbrace{0.1 (0.9)^4 \theta_{96}} + \dots \end{aligned}$$

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

$$\frac{(1-\epsilon)^{1/\epsilon}}{0.9} \approx \frac{1}{e}$$

$$\epsilon = 0.02 \rightarrow \underbrace{0.98^{50}} \approx \frac{1}{e}$$



$$\approx \frac{1}{1-\beta}$$

$$\epsilon = 1 - \beta$$

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

Implementing exponentially weighted averages

$$v_0 = 0$$

$$v_1 = \beta v_0 + (1 - \beta) \theta_1$$

$$v_2 = \beta v_1 + (1 - \beta) \theta_2$$

$$v_3 = \beta v_2 + (1 - \beta) \theta_3$$

...

$$V_\theta := 0$$

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

$$V_\theta := \beta v + (1 - \beta) \theta_2$$

\vdots

$$\rightarrow V_\theta = 0$$

Repeat {

Get next θ_t

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

}



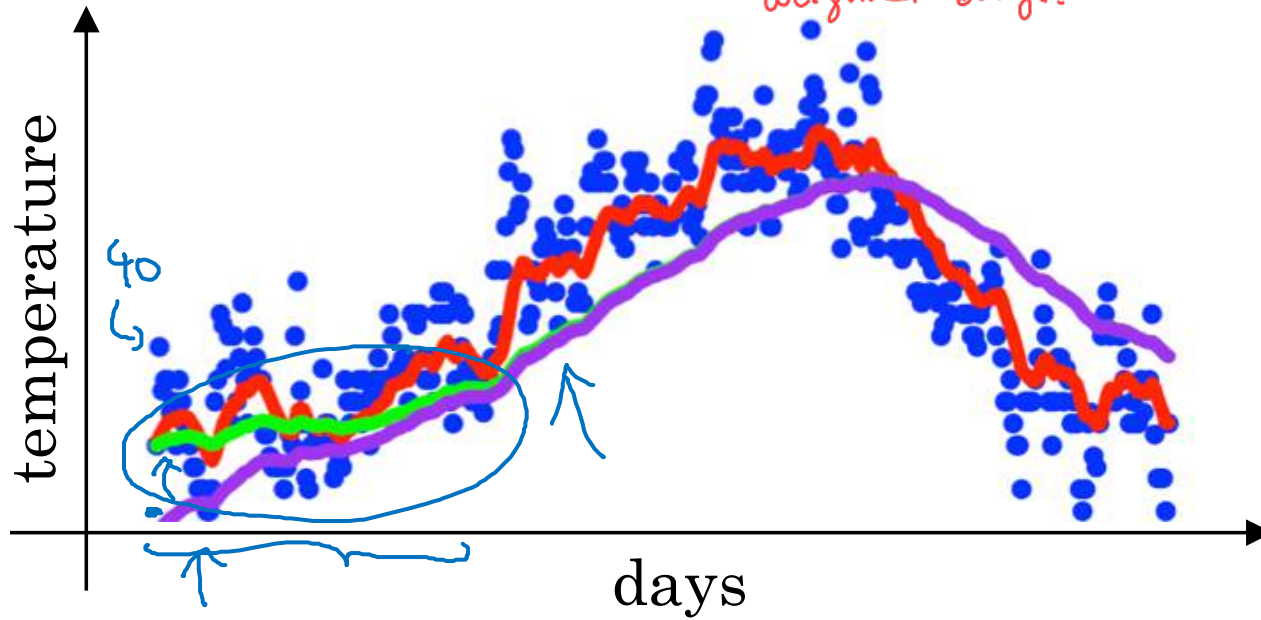
deeplearning.ai

Optimization Algorithms

Bias correction
in exponentially
weighted average

Bias correction

- Problem: at the beginning the weighted average is very small because there is not much history to contribute to the weighted average.



$$\beta = 0.98$$

- A correction of this bias is obtained scaling the estimate by $(1-\beta^t)$:

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

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

$$v_0 = 0$$

$$v_1 = \cancel{0.98 v_0} + \underbrace{0.02 \theta_1}$$

$$v_2 = 0.98 v_1 + 0.02 \theta_2$$

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

$$= \underline{0.0196 \theta_1} + \underline{0.02 \theta_2}$$

$$\frac{v_t}{1-\beta^t}$$

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

$$\frac{v_2}{0.0396}$$

$$= \frac{\underset{\downarrow}{0.0196} \theta_1 + \underset{\downarrow}{0.02} \theta_2}{0.0396}$$

- In practice this problem is often ignored.



deeplearning.ai

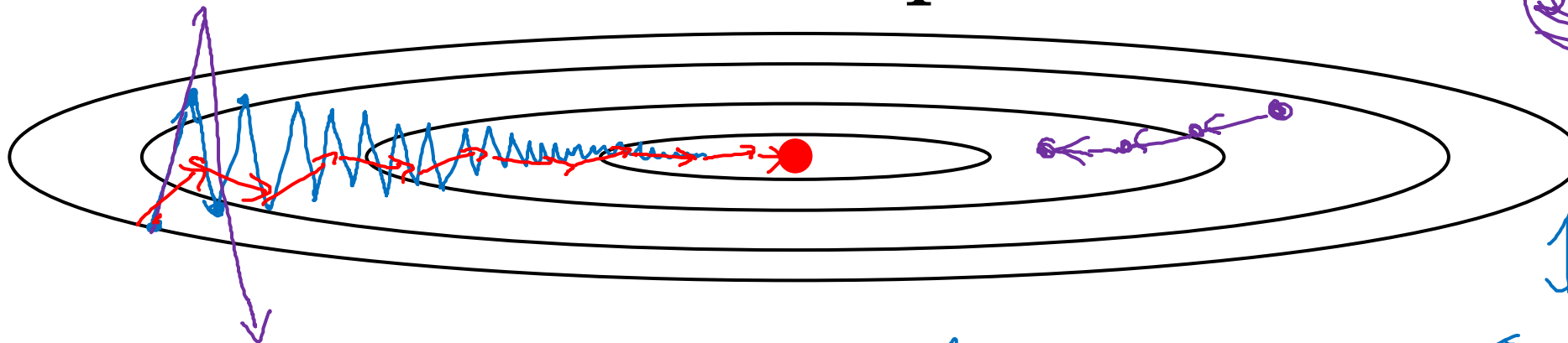
Optimization Algorithms

Gradient descent with momentum

= GD with smoothed gradient (exponentially weighted average).

Gradient descent example

- IDEA: GD in which the gradient is smoothed via weighted average before applying the update.



↑ slower learning
↔ faster learning.

Momentum:

On iteration t :

Compute $\Delta W, \Delta b$ on current mini-batch.

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

smoothed ΔW

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

smoothed Δb

friction → ↑ velocity

↑ acceleration

$$W := W - \alpha V_{\Delta W}, \quad b := b - \alpha V_{\Delta b}$$



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

Implementation details

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

On iteration t :

Compute dW, db on the current mini-batch

$$\begin{aligned} \rightarrow v_{dw} &= \beta v_{dw} + (1 - \beta) dW \\ \rightarrow v_{db} &= \beta v_{db} + (1 - \beta) db \end{aligned}$$

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

$$v_{dw} = \beta v_{dw} + dW \leftarrow$$

NOTE: in the literature, this version is often used.

This would correspond to the old formulation if β was scaled by $(1-\beta)$ instead.

$$\begin{aligned} v_{dw} &= \beta v_{dw} + (1-\beta) dW \\ \frac{v_{dw}}{1-\beta} &= \frac{\beta}{1-\beta} v_{dw} + dW = \hat{\beta} v_{dw} + dW \end{aligned}$$

...?... Ng prefers the previous

$$\cancel{v_{dw} = \beta v_{dw} + dW}$$

NOTE: bias correction is not used in practice with GD with momentum

Hyperparameters: α, β

learning rate

momentum

$$\beta = 0.9$$

corresponds to average over last ≈ 10 gradients



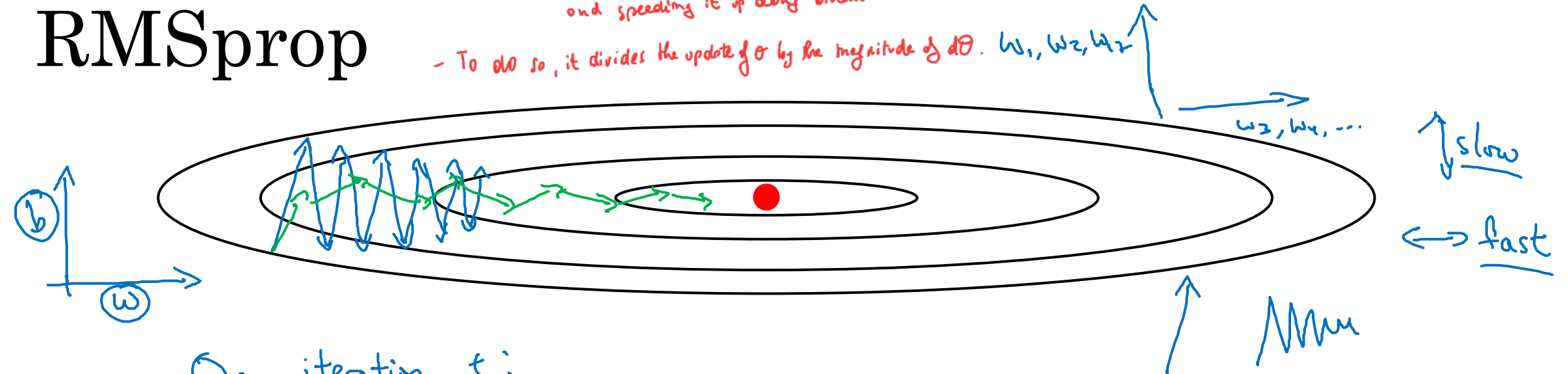
deeplearning.ai

Optimization Algorithms

RMSprop

RMSprop

- Intuition: we might want to slow down the update along certain dimensions and speeding it up along others.
- To do so, it divides the update of θ by the magnitude of $d\theta$. w_1, w_2, w_3



On iteration t :

Compute dw, db on current mini-batch

$$S_{dw} = \beta_2 S_{dw} + (1 - \beta_2) \underbrace{dw^2}_{\text{element-wise}} \leftarrow \text{small}$$

$$\rightarrow \underline{S_{db}} = \beta_2 S_{db} + (1 - \beta_2) \underline{db^2} \leftarrow \text{large}$$

$$w := w - \alpha \frac{dw}{\sqrt{S_{dw} + \epsilon}}$$

$$b := b - \alpha \frac{db}{\sqrt{S_{db} + \epsilon}}$$

the update's numerator is the gradient $d\theta$ while in the momentum it was the smoothed gradient $\hat{d\theta}$.

$$\epsilon = 10^{-8}$$

the huge S_{db} at the denominator makes the update along θ large/smaller if θ is smaller/larger.



deeplearning.ai

Optimization Algorithms

Adam optimization algorithm

"ADAPtive Moment estimation"

Adam optimization algorithm

- Mixes momentum and RMS in the update rule.

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

On iteration t :

Compute dw, db using current mini-batch

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

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

$yhat = np.array([.9, 0.2, 0.1, .4, .9])$

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

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

} ← bias correction

$$W := W - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected} + \epsilon}} \quad \left. \begin{array}{l} \text{momentum} \\ \text{rms} \end{array} \right\}$$

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

idea:

- correct dw with both momentum and RMS
→ V_{dw}, S_{dw}

- perform the update mixing the two

$\theta := \theta - \alpha \frac{V_{dw}}{\sqrt{S_{dw} + \epsilon}}$
 ← momentum
 ← RMS
 in RMS this is dw
 in momentum this is θ

Hyperparameters choice:

→ α : needs to be tuned

→ β_1 : 0.9 → (dw)

→ β_2 : 0.999 → (dw^2)

→ ϵ : 10^{-8}

recommended by the authors

Adam: Adaptive moment estimation



Adam Coates



deeplearning.ai

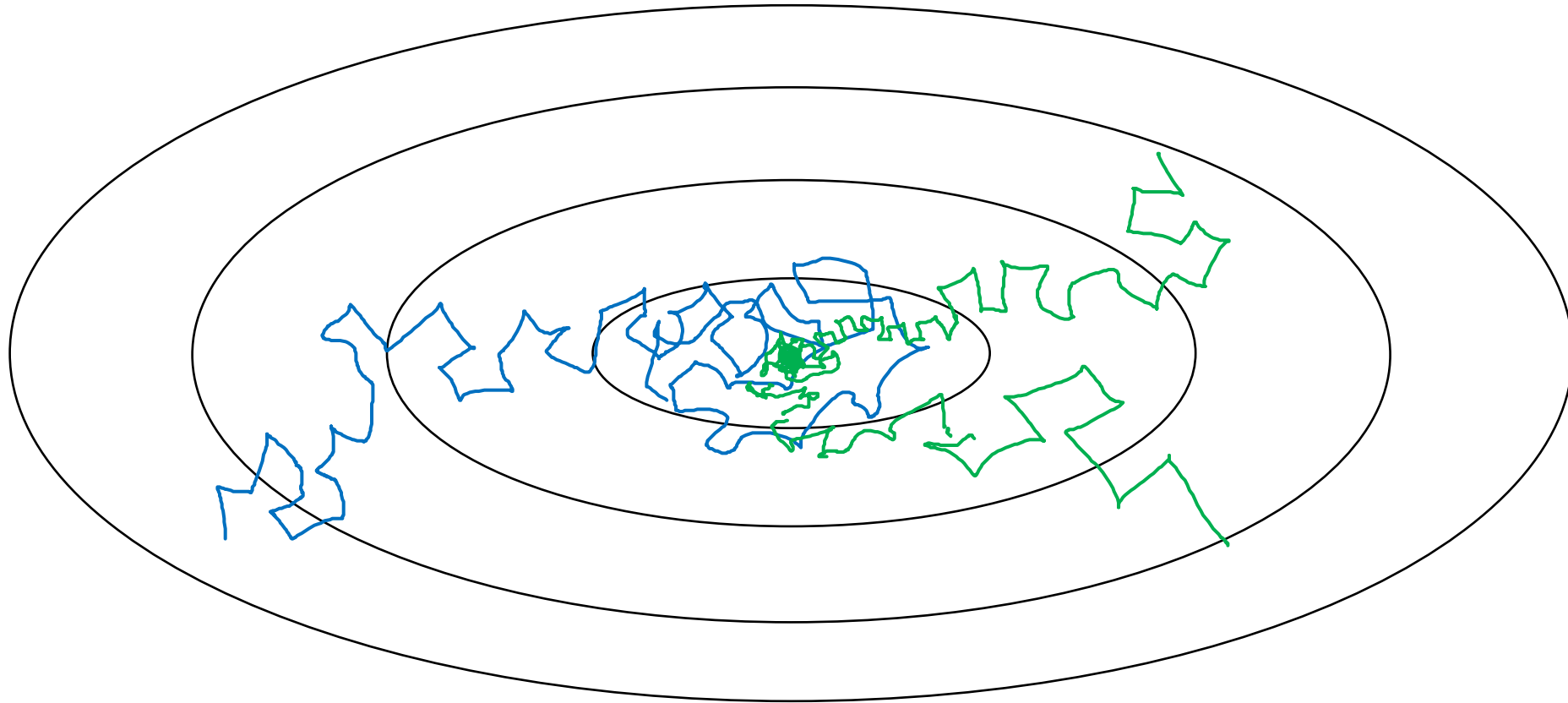
Optimization Algorithms

Learning rate decay

- It helps speeding up the convergence, but it is not as critical as the choice of the optimizer and of the other parameters.

Learning rate decay

Slowly reduce α

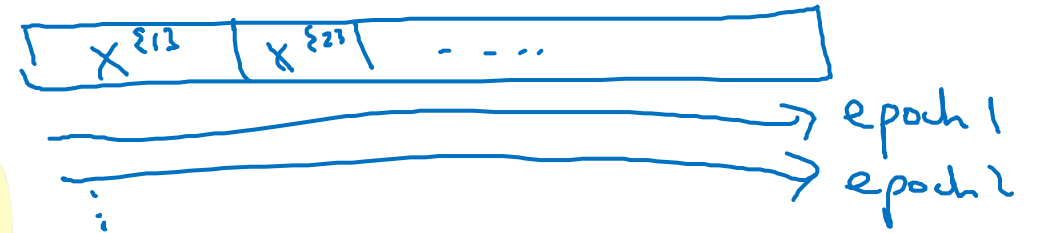


Learning rate decay

Idea: reduce the learning rate at every epoch.

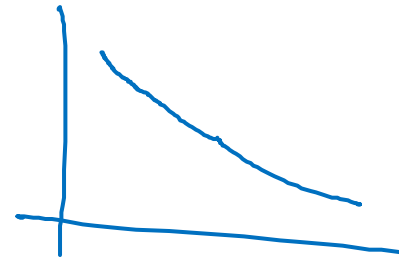
1 epoch = 1 pass through data.

$$\alpha = \frac{1}{1 + \text{decay-rate} * \text{epoch-num}} \alpha_0$$



$$\alpha_0 = 0.2$$
$$\text{decay-rate} = 1$$

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
\vdots	\vdots

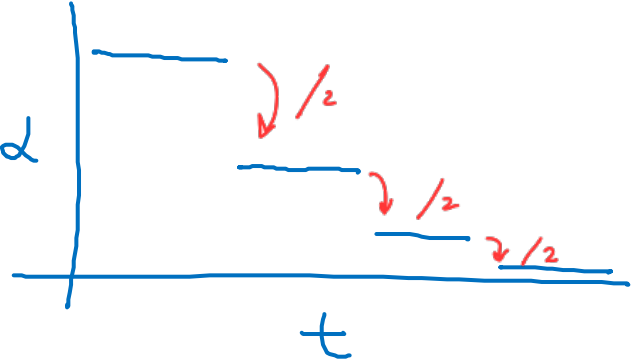


Other learning rate decay methods

formula {

$$\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0$$

← "exponential decay".

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


discrete staircase

Manual decay.



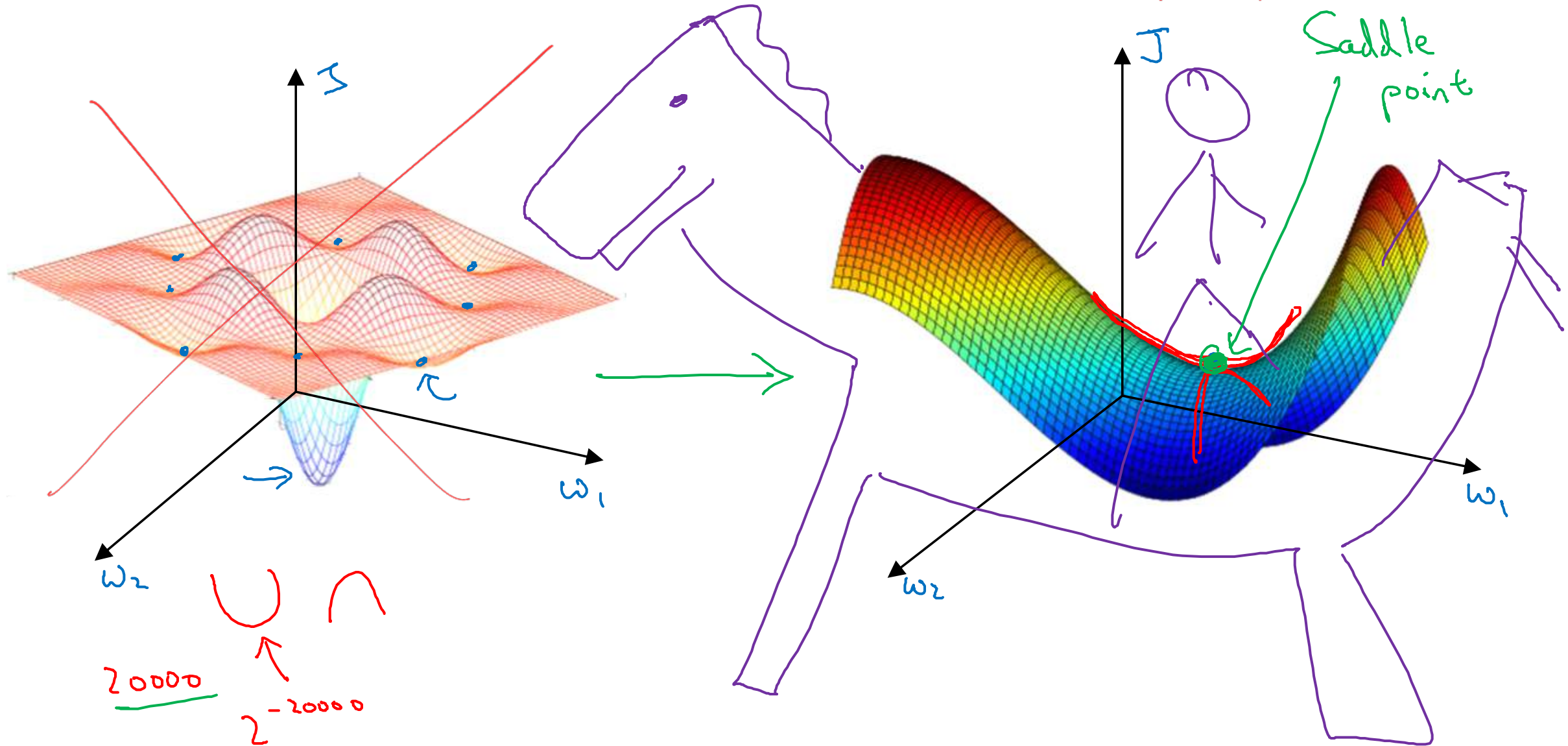
deeplearning.ai

Optimization Algorithms

The problem of local optima

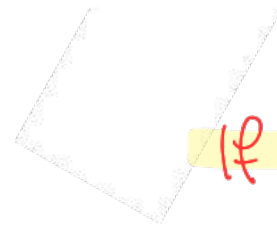
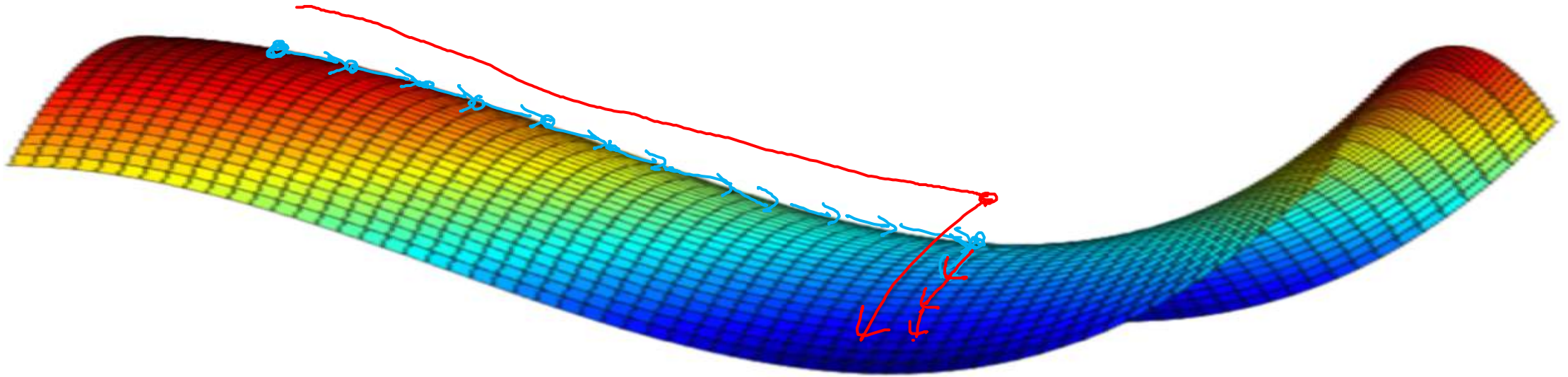
Local optima in neural networks

- In many dimensions, most local minima are saddle points (because over many dimensions there would be some with $\text{grad} > 0$ and others with $\text{grad} < 0$).



Problem of plateaus

Therefore



If the cost function has many parameters:

- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow