



deeplearning.ai

Optimization Algorithms

Mini-batch gradient descent

Batch vs. mini-batch gradient descent

X, Y

$X^{\{t\}}, Y^{\{t\}}$

Vectorization allows you to efficiently compute on m examples.

$$X = \begin{bmatrix} X^{(1)} & X^{(2)} & X^{(3)} & \dots & X^{(500)} & | & X^{(1001)} & \dots & X^{(2000)} & | & \dots & | & \dots & X^{(m)} \end{bmatrix}$$

(n_x, m)

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

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

$(1, m)$

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

What if $m = 5,000,000$?

5,000 mini-batches of 1,000 each

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

$X^{(i)}$
 $Z^{[l]}$
 $X^{\{t\}}, Y^{\{t\}}$

Mini-batch gradient descent

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

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

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

$$A^{(l)} = g^{(l)}(Z^{(l)})$$

:

$$A^{(L)} = g^{(L)}(Z^{(L)})$$

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^L f(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_l \|W^{(l)}\|_F^2$.

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

$$W^{(l)} := W^{(l)} - \alpha \nabla W^{(l)}, \quad b^{(l)} := b^{(l)} - \alpha \nabla b^{(l)}$$

3 } 3 }

"1 epoch"
└ pass through training set.

1 step of gradient descent
using $\frac{X^{\{t+1\}}}{Y^{\{t+1\}}}$
(as if $t=5000$)

X, Y



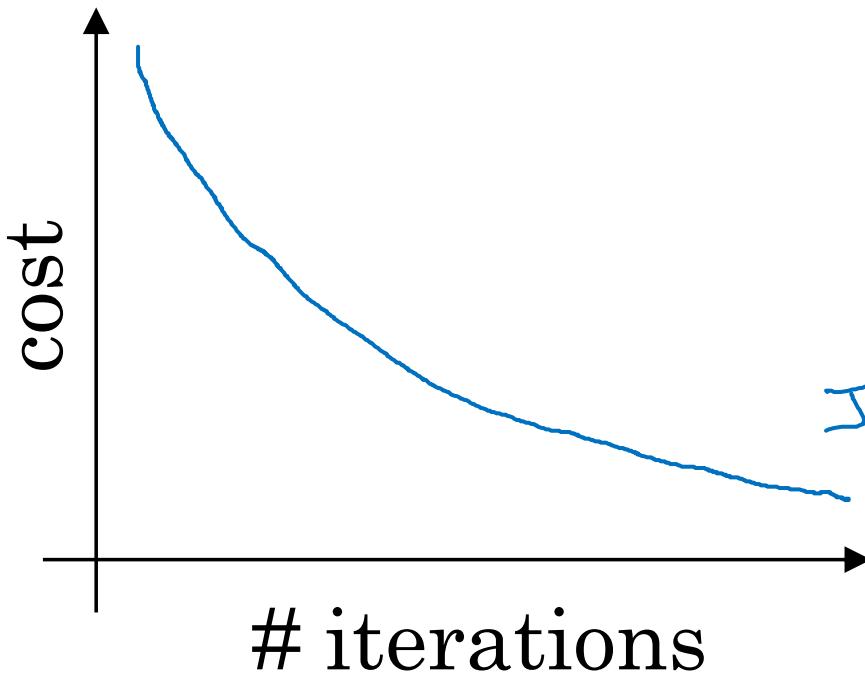
deeplearning.ai

Optimization Algorithms

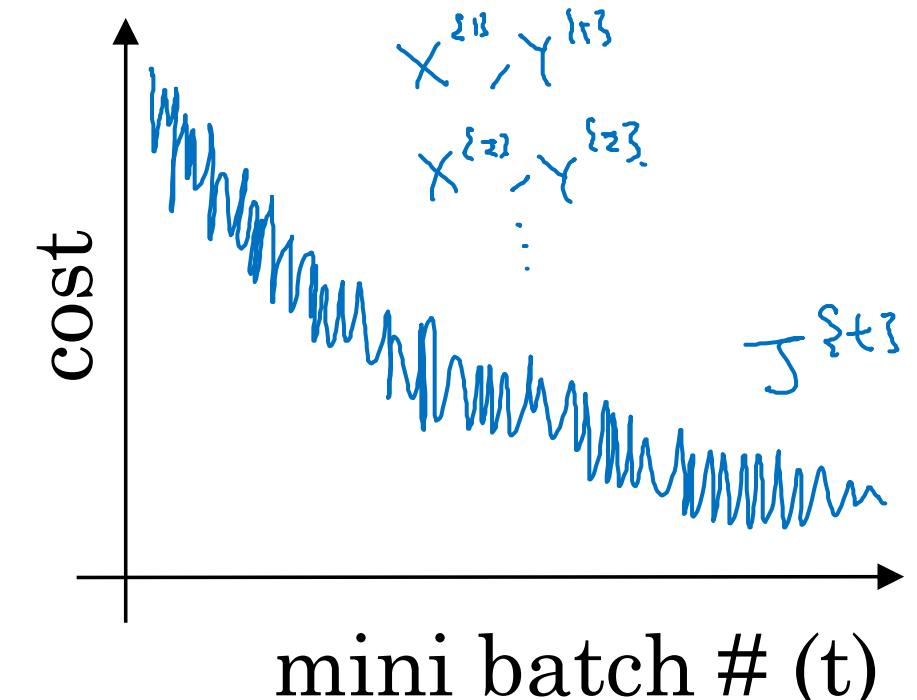
Understanding mini-batch gradient descent

Training with mini batch gradient descent

Batch gradient descent



Mini-batch gradient descent



Plot J^{st} computed using $x^{\{t\}}, y^{\{t\}}$

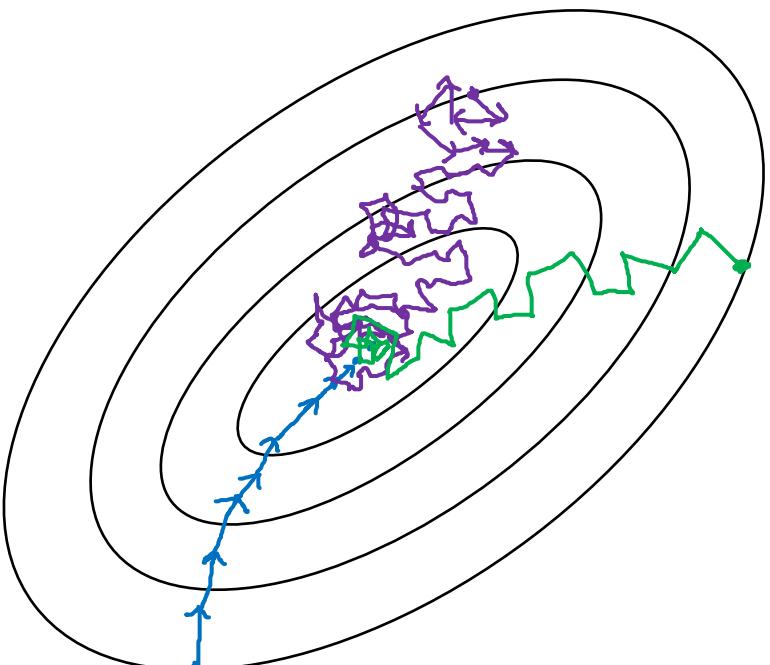
Choosing your mini-batch size

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

$$(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$$

→ If mini-batch size = 1 : Stochastic gradient descent. Every example is its own
 $(X^{(1)}, Y^{(1)}) = (x^{(1)}, y^{(1)}) \dots (x^{(n)}, y^{(n)})$ mini-batch.

In practice: Somehw in-between 1 and m



Stochastic
gradient
descent

{
Use speedup
from vectorization

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

{
Faster learning.

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

Batch
gradient descent
(mini-batch size = m)

{
Two long
per iteration

Choosing your mini-batch size

If small training set : Use batch gradient descent.
 $(m \leq 2000)$

Typical mini-batch sizes:

$$\rightarrow 64, 128, 256, 512 \quad \frac{1024}{2^{10}}$$

$2^6 \quad 2^7 \quad 2^8 \quad 2^9$



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

$$X^{\{t\}}, Y^{\{t\}}$$



deeplearning.ai

Optimization Algorithms

Exponentially weighted averages

Temperature in London

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

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

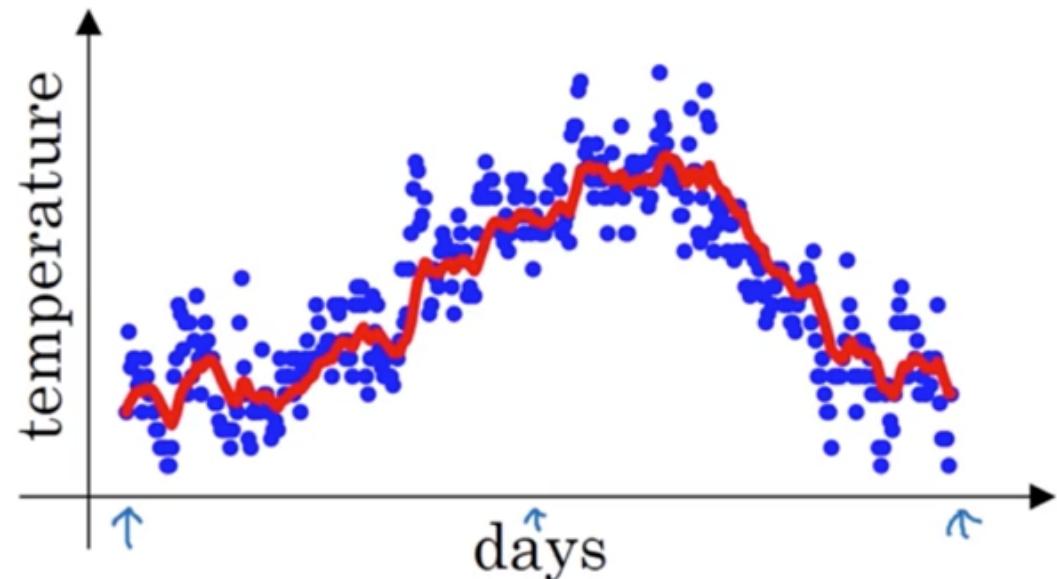
$$\theta_3 = 45^{\circ}\text{F} \quad \vdots$$

\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

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

$\beta = 0.9$: ≈ 10 days' temper.

$\beta = 0.98$: ≈ 50 days

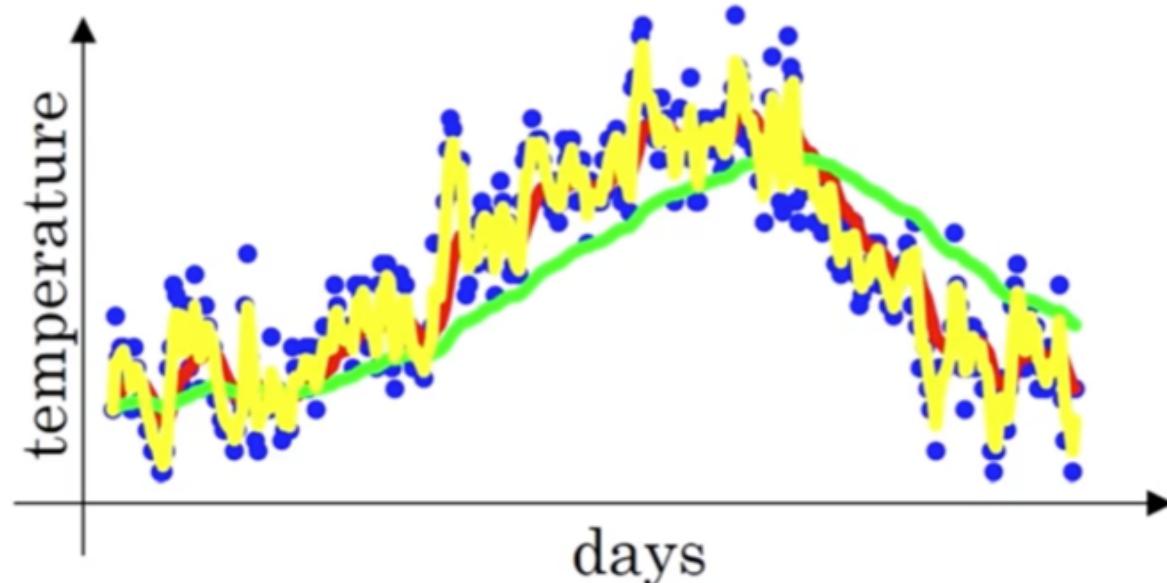
$\beta = 0.5$: ≈ 2 days

V_t is approximately

average over

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

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





deeplearning.ai

Optimization Algorithms

Understanding exponentially weighted averages

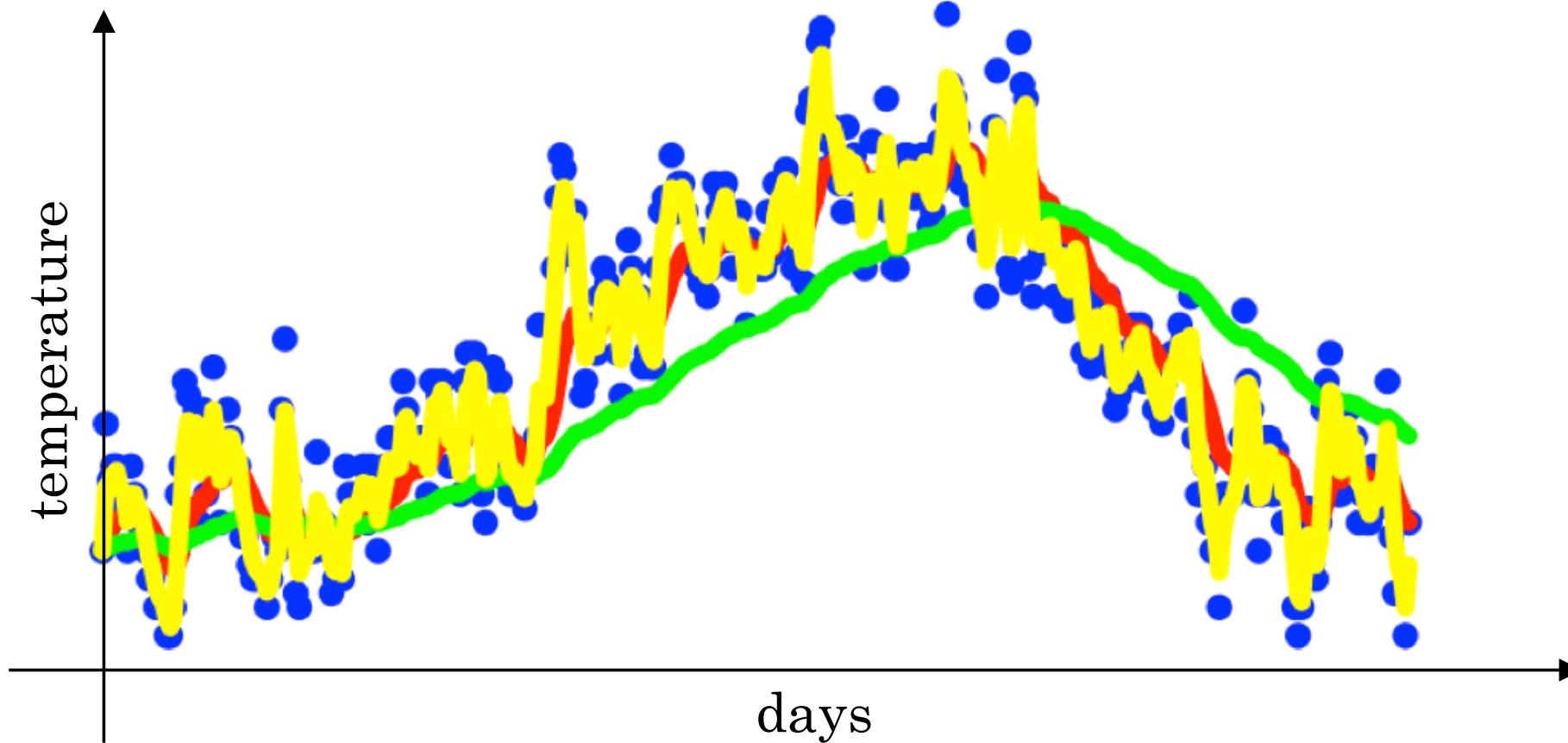
Exponentially weighted averages

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

$$\beta = 0.9$$

$$0.98$$

$$0.5$$



Exponentially weighted averages

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

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

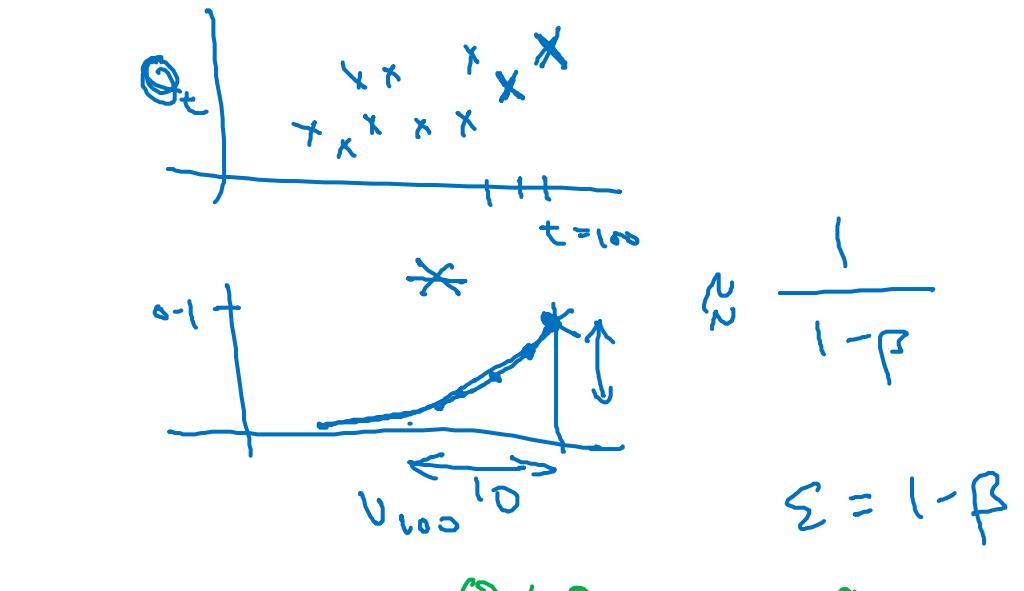
$$v_{99} = 0.9v_{98} + 0.1\theta_{99}$$

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

...

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

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



$$\frac{(1-\varepsilon)^{1/\varepsilon}}{0.9} = \frac{1}{e}$$

$0.98 ?$

$$\varepsilon = 0.02 \rightarrow \underline{0.98^{50}} \approx \frac{1}{e}$$

Implementing exponentially weighted averages

$$v_0 = 0$$

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

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

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

...

$$V_0 := 0$$

$$V_0 := \beta V + (1-\beta) \theta_1$$

$$V_0 := \beta V + (1-\beta) \theta_2$$

:

$$\rightarrow V_0 = 0$$

Repeat {

Get next θ_t

$$V_0 := \beta V_0 + (1-\beta) \theta_t \leftarrow$$

}

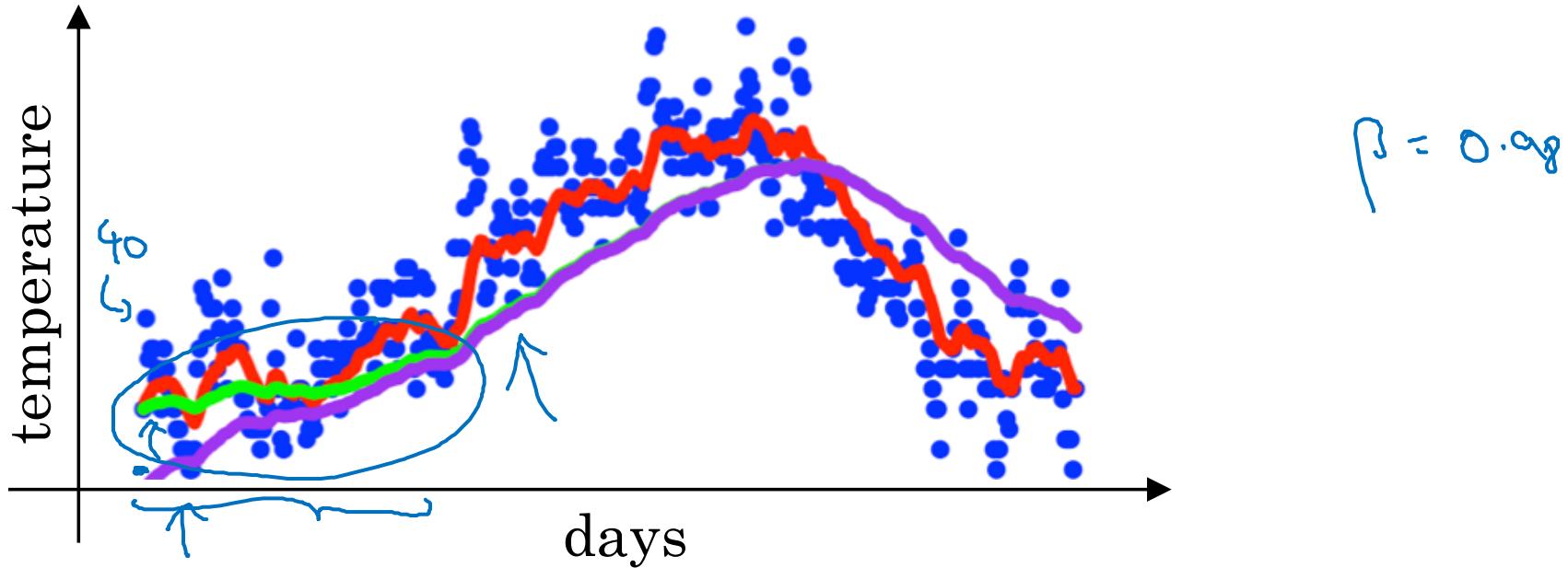


deeplearning.ai

Optimization Algorithms

Bias correction
in exponentially
weighted average

Bias correction



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

$$v_0 = 0$$

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

$$\begin{aligned} v_2 &= 0.98 v_1 + 0.02 \theta_2 \\ &= 0.98 \times 0.02 \times \theta_1 + 0.02 \theta_2 \\ &= \underline{0.0196\theta_1} + \underline{0.02\theta_2} \end{aligned}$$

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

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

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

$$\frac{\underline{0.0196\theta_1} + \underline{0.02\theta_2}}{0.0396}$$

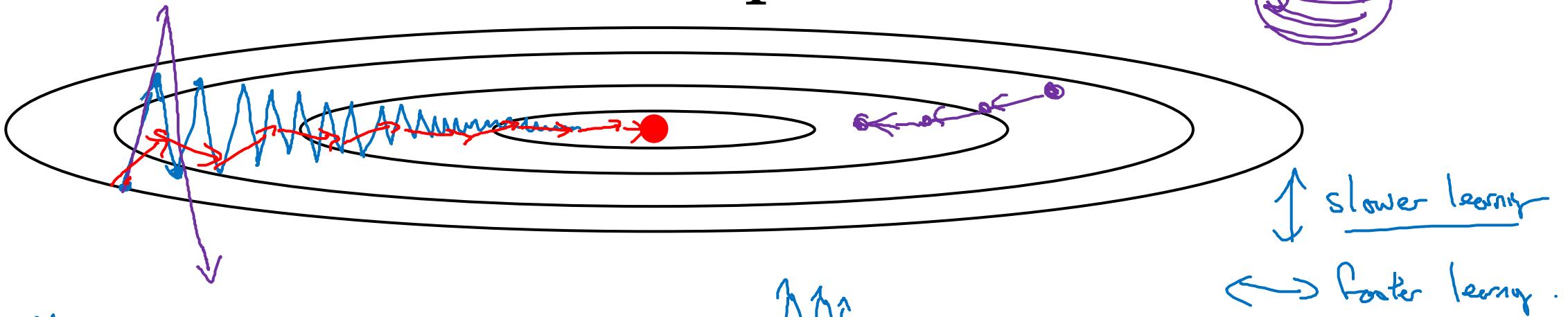


deeplearning.ai

Optimization Algorithms

Gradient descent with momentum

Gradient descent example



Momentum:

On iteration t :

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

$$v_{dw} = \beta v_{dw} + (1-\beta) \frac{\Delta w}{\text{acceleration}}$$

$$v_{db} = \beta v_{db} + (1-\beta) \frac{\Delta b}{\text{acceleration}}$$

Friction ↑ velocity

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

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

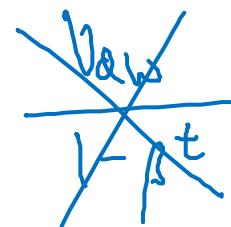
Implementation details

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

On iteration t :

Compute dW, db on the current mini-batch

$$\begin{aligned} \rightarrow v_{dw} &= \beta v_{dw} + (1 - \beta) dW \\ \rightarrow v_{db} &= \beta v_{db} + (1 - \beta) db \end{aligned} \quad \left. \right\} \quad \begin{aligned} v_{dw} &= \beta v_{dw} + dW \leftarrow \\ v_{db} &= \beta v_{db} + db \end{aligned}$$
$$W = W - \underbrace{\alpha v_{dw}}_{\text{Hyperparameter } \alpha}, b = b - \underbrace{\alpha v_{db}}_{\text{Hyperparameter } \alpha}$$



Hyperparameters: α, β

$$\beta = 0.9$$

average over last ≈ 10 gradients

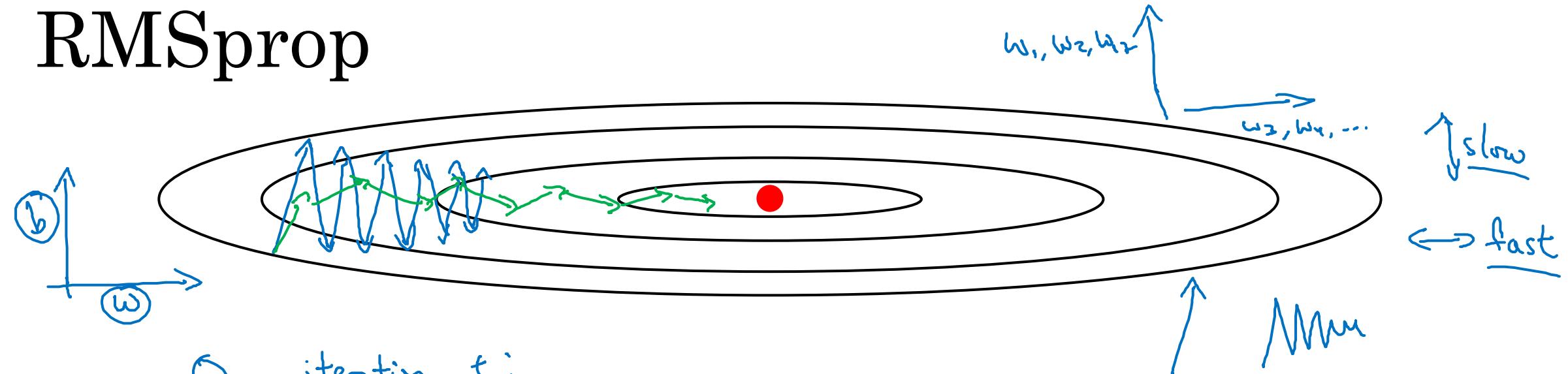


deeplearning.ai

Optimization Algorithms

RMSprop

RMSprop



On iteration t :

Compute dW, db on current mini-batch

$$\underline{S_{dw}} = \beta_2 \underline{S_{dw}} + (1-\beta_2) \underline{dW^2} \leftarrow \begin{matrix} \text{element-wise} \\ \text{small} \end{matrix}$$

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

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

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

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



deeplearning.ai

Optimization Algorithms

Adam optimization algorithm

Adam optimization algorithm

$$V_{dw} = 0, S_{dw} = 0. \quad V_{db} = 0, S_{db} = 0$$

On iteration t :

Compute $\delta w, \delta b$ using current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1 - \beta_1) \delta w, \quad V_{db} = \beta_1 V_{db} + (1 - \beta_1) \delta b \quad \leftarrow \text{"moment"} \beta_1$$

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

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

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

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

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

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

Hyperparameters choice:

- α : needs to be tune
- β_1 : 0.9 $\rightarrow (\underline{dw})$
- β_2 : 0.999 $\rightarrow (\underline{dw^2})$
- ϵ : 10^{-8}

Adam: Adaptive moment estimation



Adam Coates



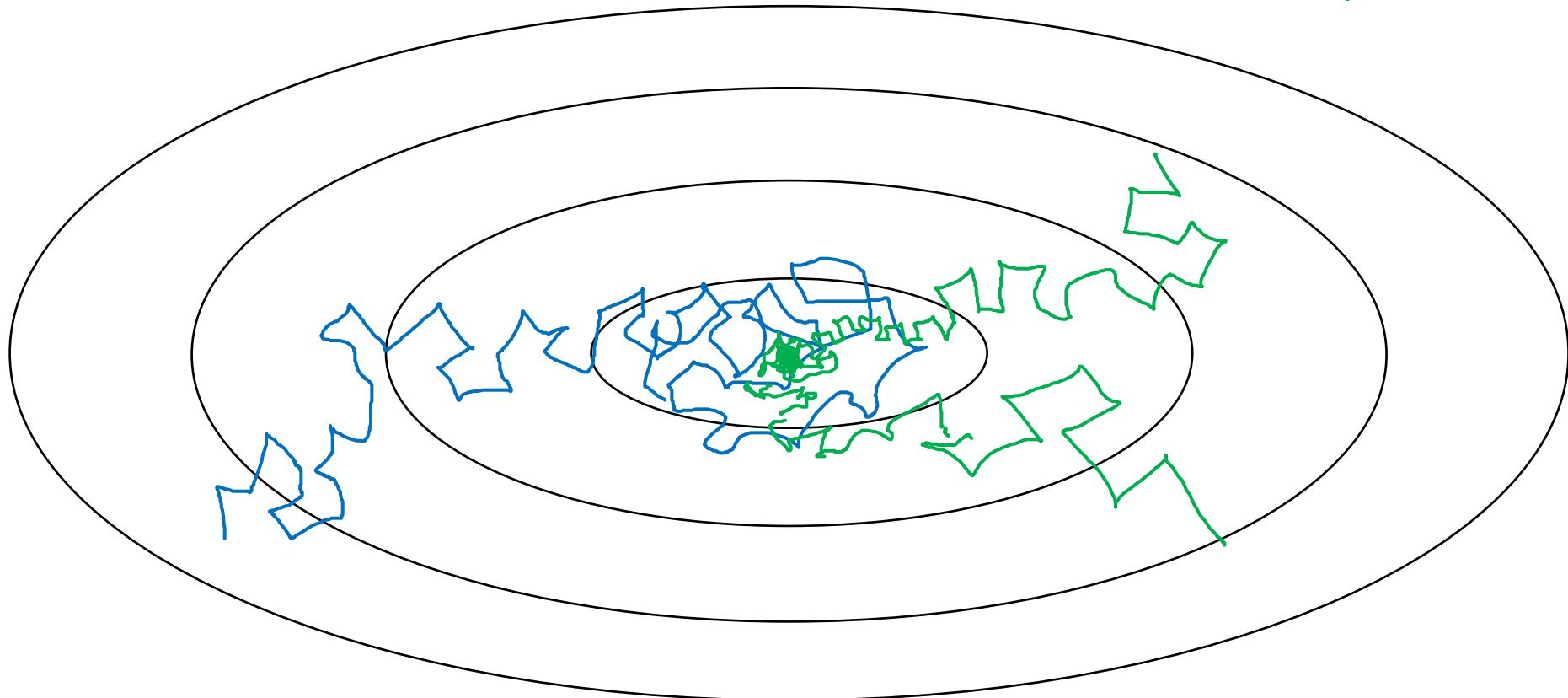
deeplearning.ai

Optimization Algorithms

Learning rate decay

Learning rate decay

Slowly reduce λ

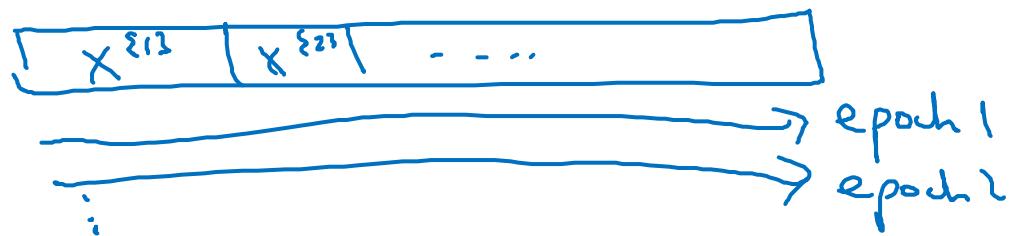


Learning rate decay

1 epoch = 1 pass through data.

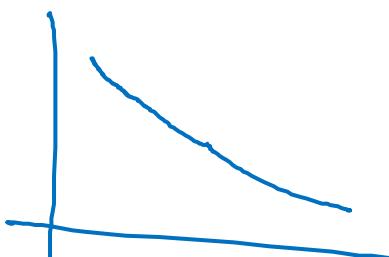
$$\alpha = \frac{\alpha_0}{1 + \text{decay-rate} * \text{epoch-num}}$$

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
:	:



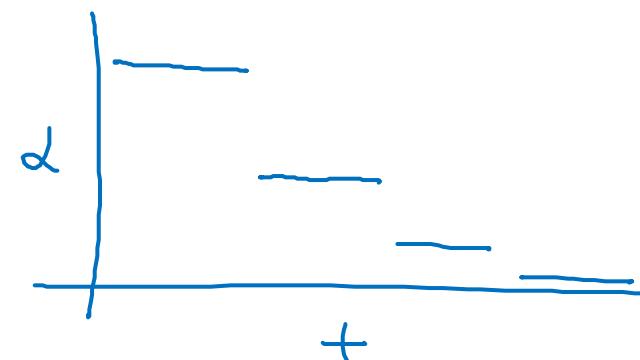
$$\alpha_0 = 0.2$$

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



Other learning rate decay methods

formulas

$$\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0 \quad - \text{exponentially decay.}$$
$$\alpha = \frac{k}{\sqrt{\text{epoch-num}}} \cdot \alpha_0 \quad \text{or} \quad \frac{k}{\sqrt{t}} \cdot \alpha_0$$


discrete staircase

Manual decay.

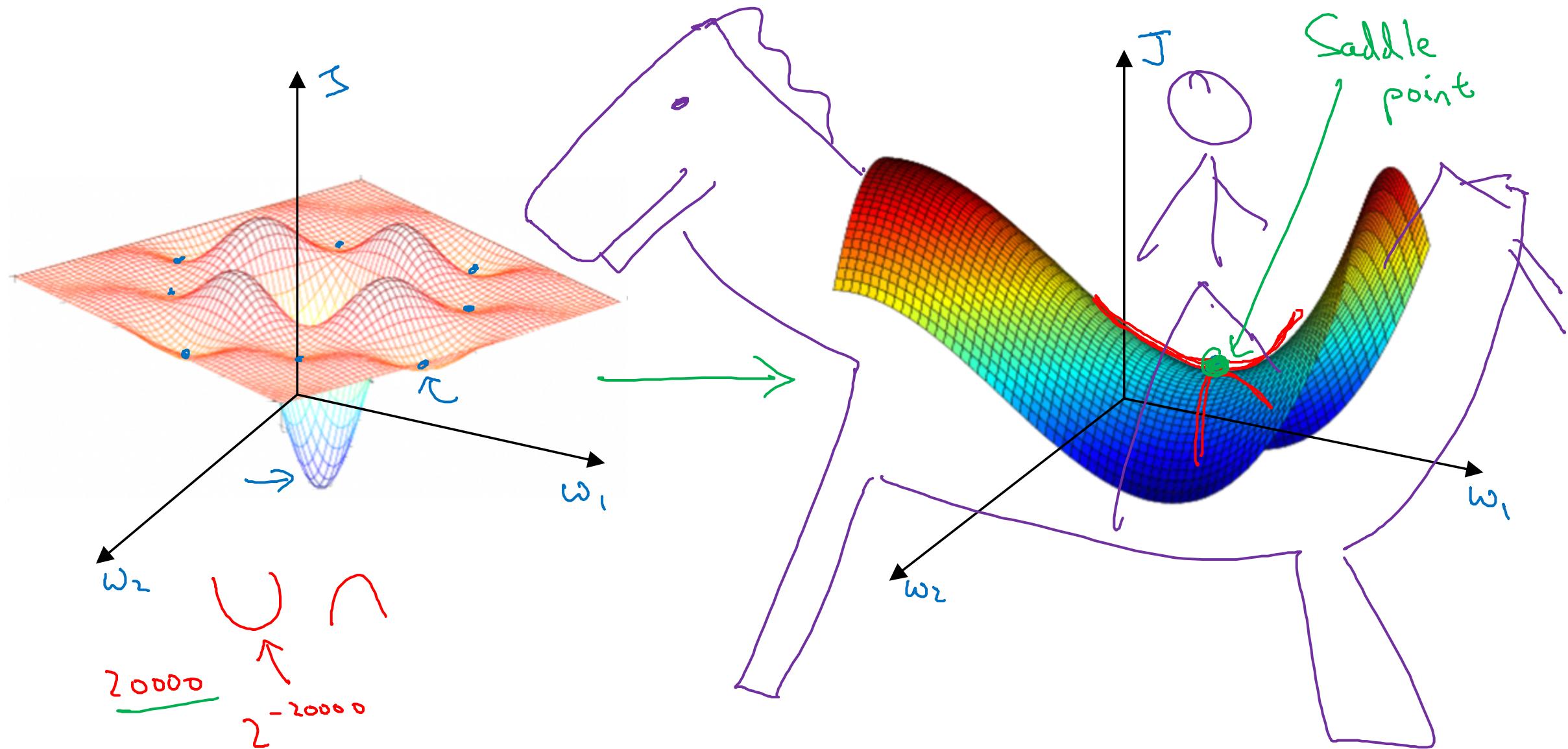


deeplearning.ai

Optimization Algorithms

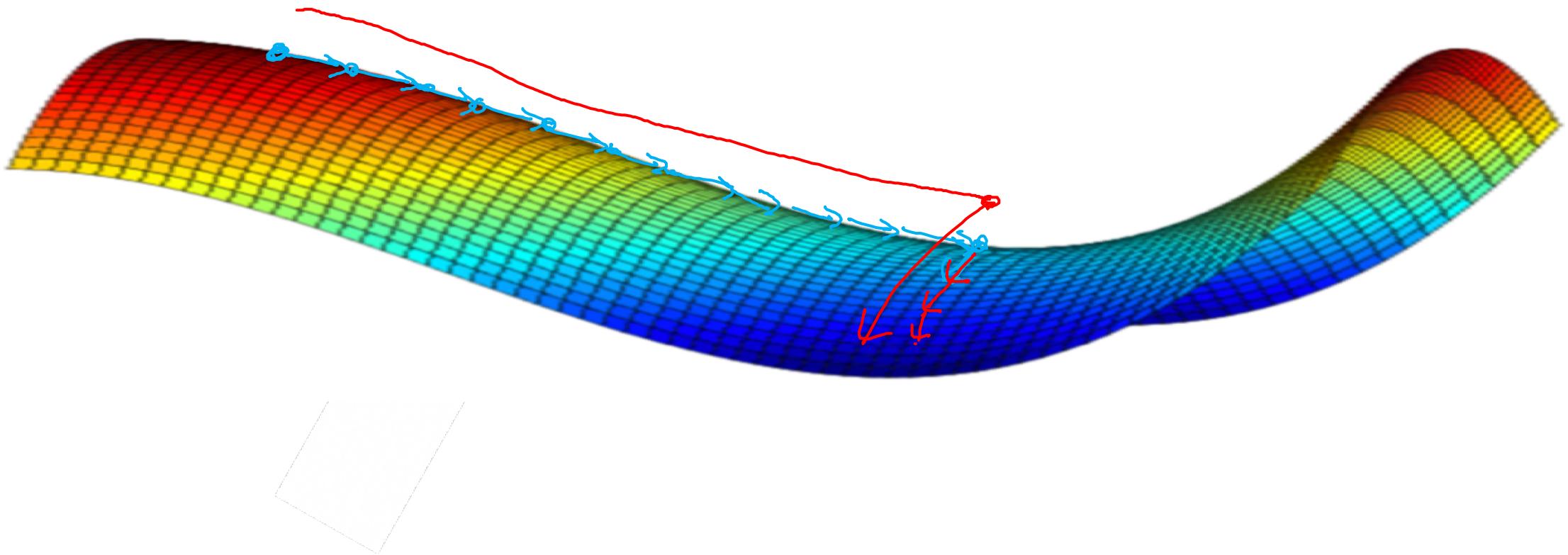
The problem of local optima

Local optima in neural networks



Andrew Ng

Problem of plateaus



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