

Copyright Notice

These slides are distributed under the Creative Commons License.

[DeepLearning.AI](#) makes these slides available for educational purposes. You may not use or distribute these slides for commercial purposes. You may make copies of these slides and use or distribute them for educational purposes as long as you cite [DeepLearning.AI](#) as the source of the slides.

For the rest of the details of the license, see <https://creativecommons.org/licenses/by-sa/2.0/legalcode>

LAB : 1-Implementing momentum, adam, and mini-batches.



deeplearning.ai

Optimization Algorithms

Mini-batch gradient descent

It allows to converge faster. Considering all examples every time is time consuming especially when we have large data sets, so we divide our training set into

Batch vs. mini-batch gradient descent

X, Y

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

Vectorization allows you to efficiently compute on m examples.

$$X = [x^{(1)} \ x^{(2)} \ x^{(3)} \ \dots \ x^{(500)} \ | \ x^{(1001)} \ \dots \ x^{(2000)} \ | \ \dots \ | \ \dots \ x^{(m)}]$$

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

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

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

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

When using batch mode, the cost should decrease in every single iteration while it will not be the case in the mini batch . In mini batch if we set size = m, t



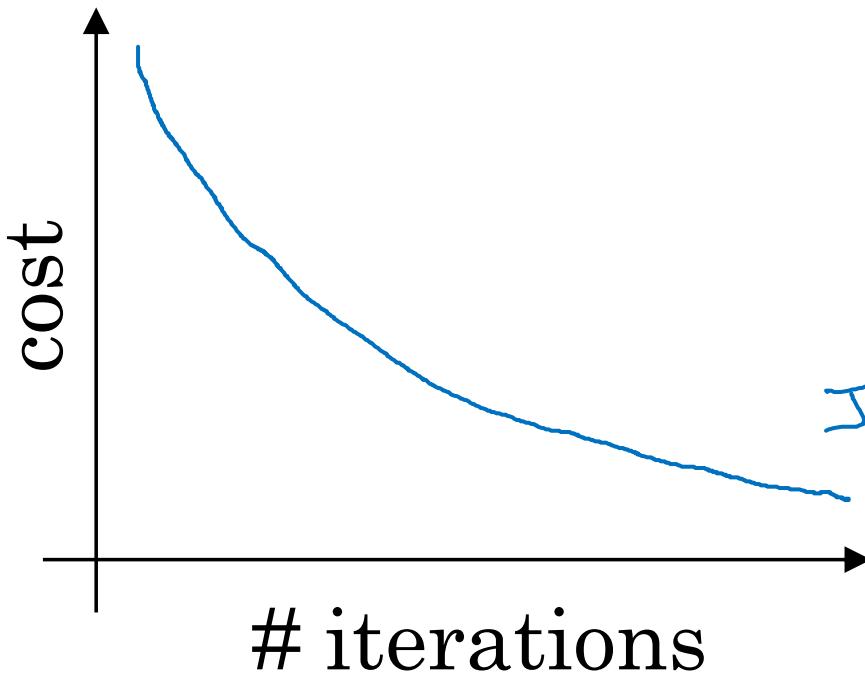
deeplearning.ai

Optimization Algorithms

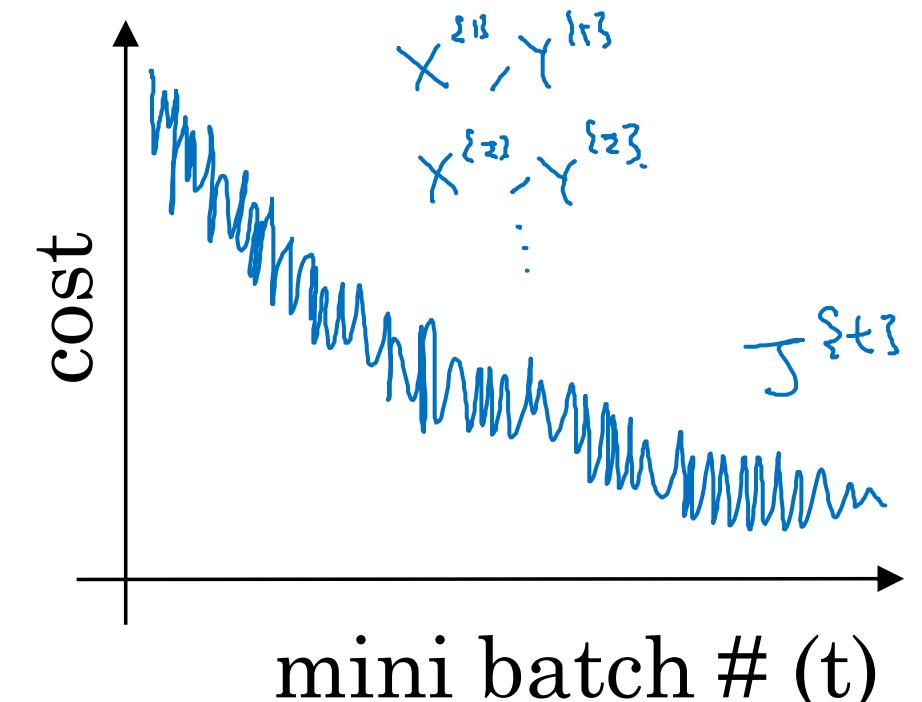
Understanding
mini-batch
gradient descent

Training with mini batch gradient descent

Batch gradient descent



Mini-batch gradient descent



Plot J^{st} computed using $X^{\{t\}}, Y^{\{t\}}$

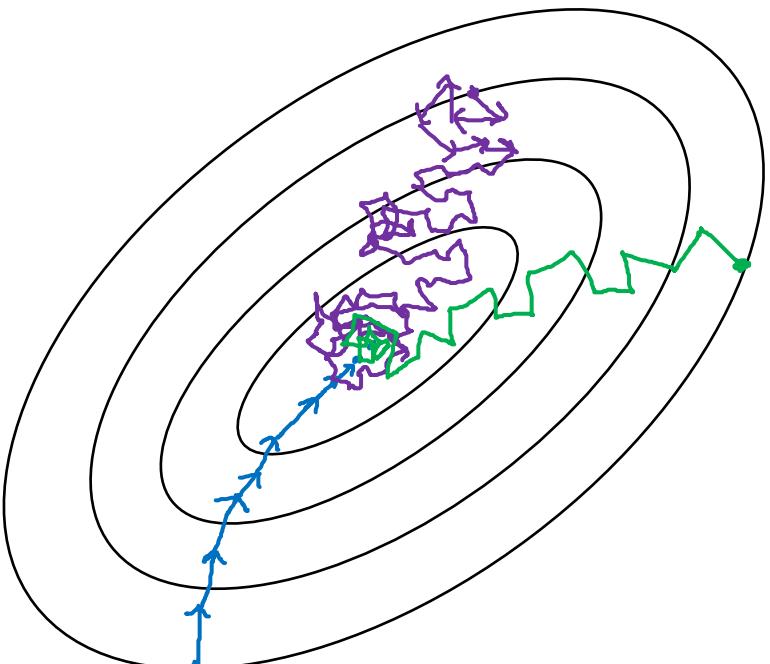
Choosing your mini-batch size

→ 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

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

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

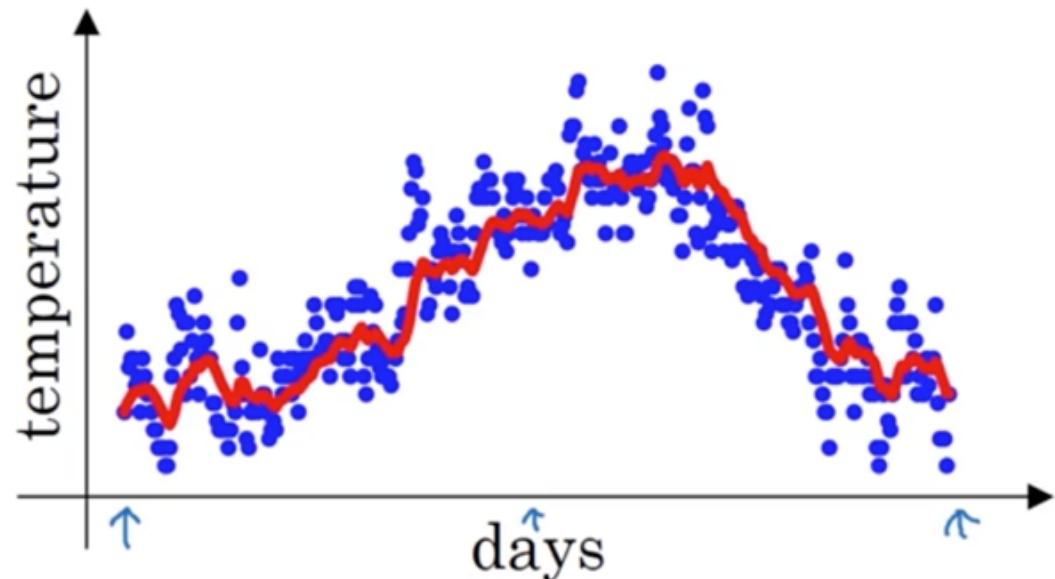
$$\theta_3 = 45^{\circ}\text{F} \quad \vdots$$

\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

$$\underline{V}_t = \beta \underline{V}_{t-1} + (1-\beta) \underline{\theta}_t \leftarrow$$

$\beta = 0.9$: ≈ 10 days' temper.

$\beta = 0.98$: ≈ 50 days

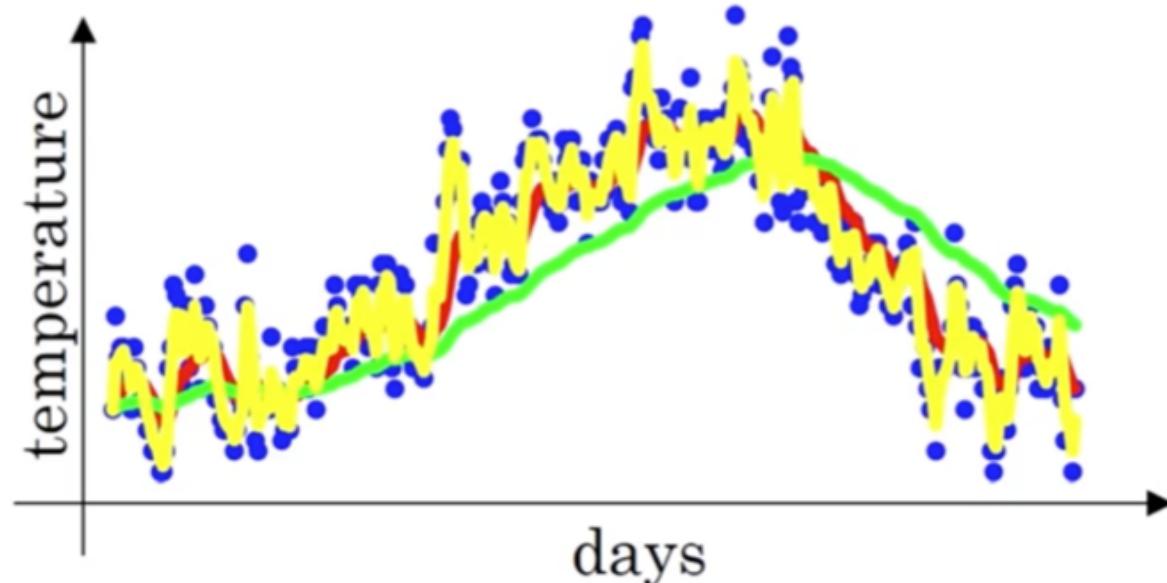
$\beta = 0.5$: ≈ 2 days

V_t is approximately

average over

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

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



We can say that beta = 0.9 indicates that it is roughly taking average of previous 10 days ($0.9^{10} = 0.35$), and $(1/e \approx 0.37)$, so after 10 days contrib



deeplearning.ai

Optimization Algorithms

Understanding exponentially weighted averages

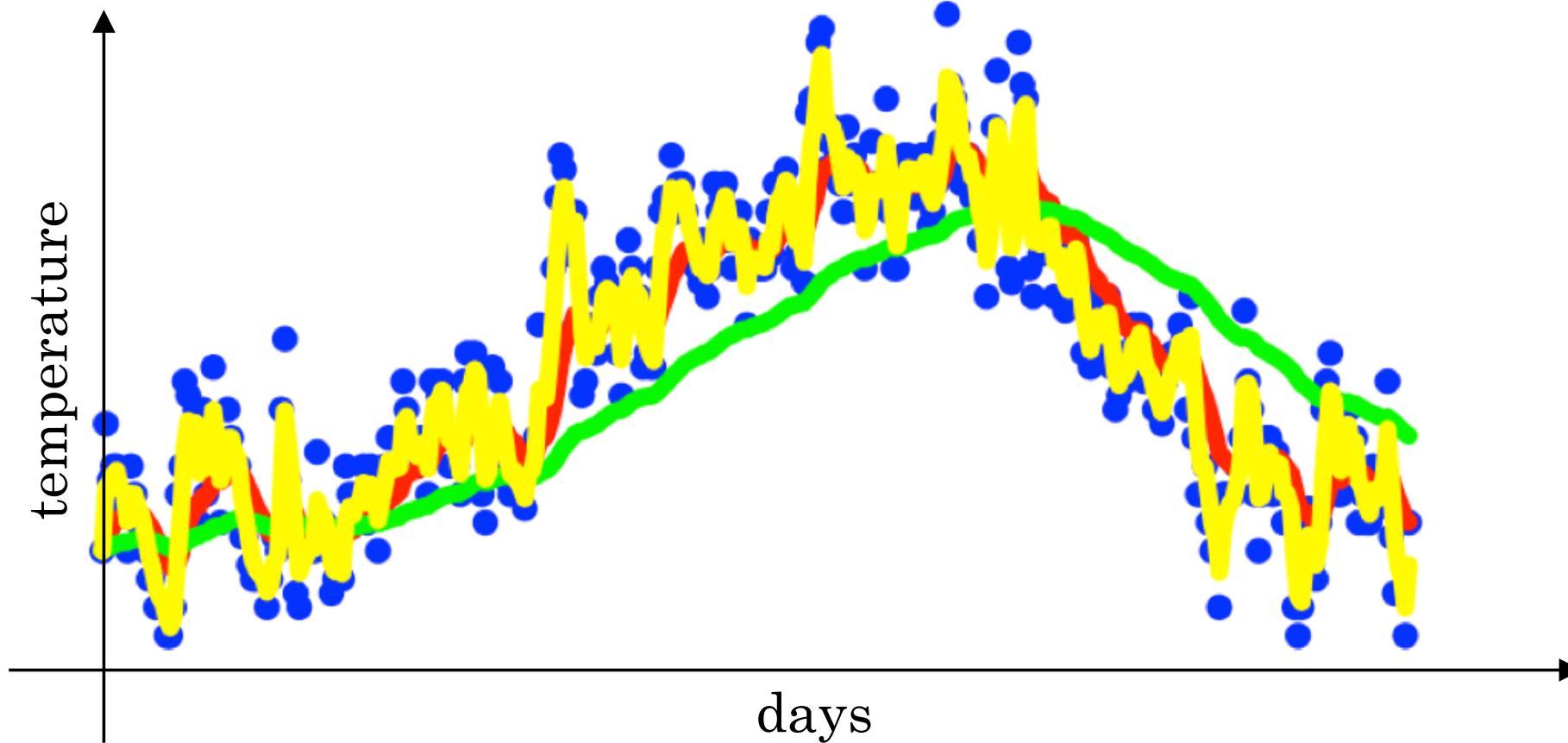
Exponentially weighted averages

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

$$\beta = 0.9$$

$$0.98$$

$$0.5$$



Exponentially weighted averages

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

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

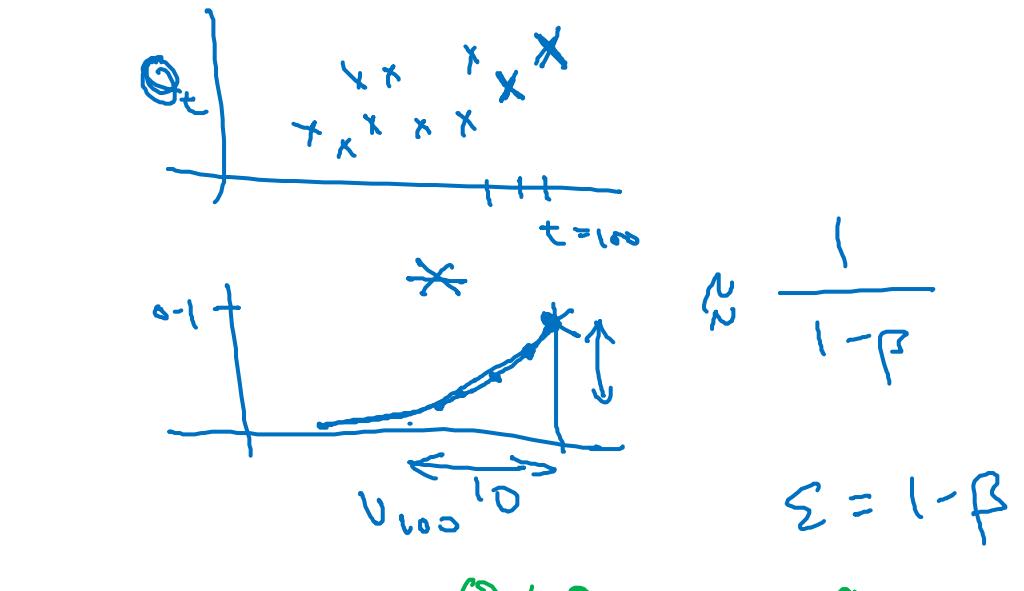
$$v_{99} = 0.9v_{98} + 0.1\theta_{99}$$

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

...

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

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



$$\Sigma = 1 - \beta$$

$$\begin{aligned} (1-\epsilon)^{1/\epsilon} &= \frac{1}{e} \\ \frac{0.9}{0.9} &\uparrow \quad 0.98 ? \\ \epsilon = 0.02 \rightarrow \underline{0.98^{50}} &\approx \frac{1}{e} \end{aligned}$$

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

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

}

As during starting phase our V_0 is 0, and if 1st day temperature is 40 degree, but beta = 0.98 so we will estimate it as 8 , and so for starting values. but we

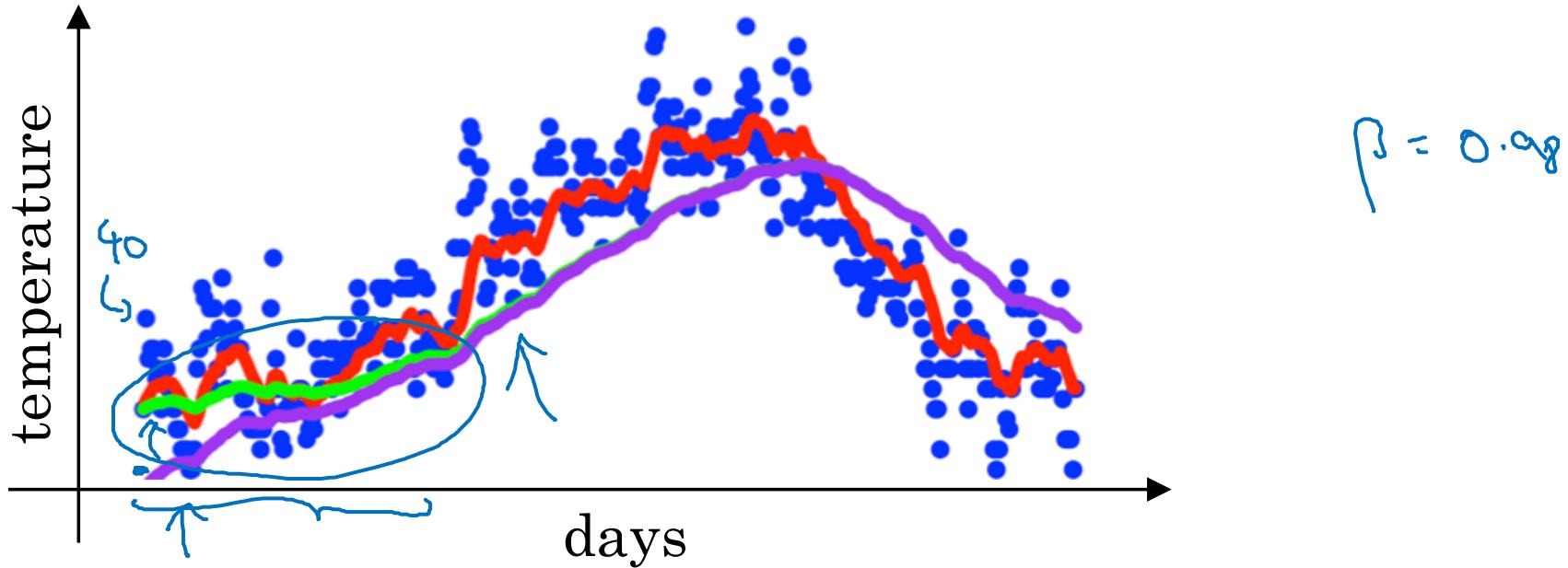


deeplearning.ai

Optimization Algorithms

Bias correction
in exponentially
weighted average

Bias correction



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

$$v_0 = 0$$

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

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

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

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

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

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

In one Sentence we can call it as idea to compute exponentially weighted avg of our gradients and then to use that gradient to update our weights instead

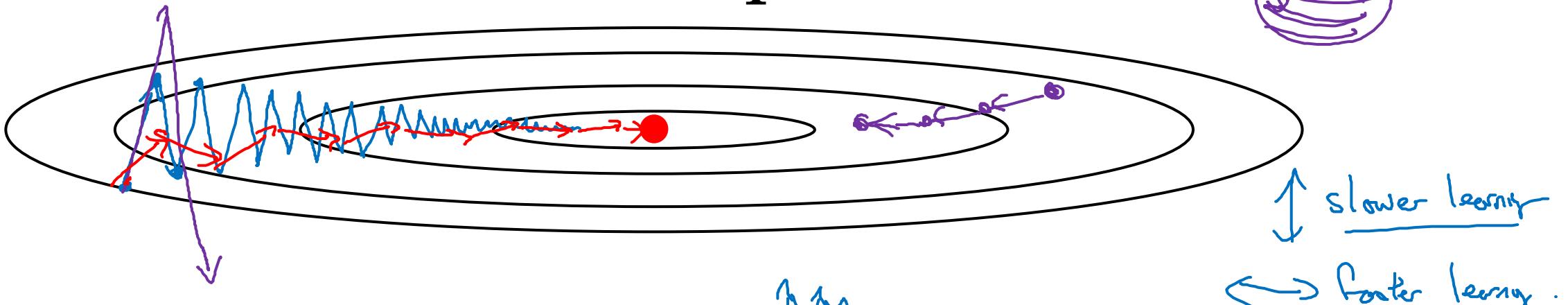


deeplearning.ai

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

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



We want learning to be slow in y axis (to prevent oscillations)

$$v_\theta = \beta v_\theta + (1-\beta) \Delta_\theta$$

Implementation details

$$V_{dw} = 0, \quad V_{ab} = 0$$

Initialisation if not using bias correction, but generally people just don't bother about bias correction. Here d

On iteration t :

Compute dW, db on the current mini-batch

$$\rightarrow v_{dW} = \beta v_{dW} + \underbrace{(1-\beta) \underline{dW}}_{\text{Error term}} \quad |$$

$$\rightarrow v_{db} = \beta v_{db} + \underbrace{(1-\beta) \underline{db}}_{\text{Error term}}$$

$\overbrace{\qquad\qquad\qquad}^{\text{Average error term}}$

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

The graph shows a blue wavy line representing the observed variable v_{dW} . A straight red line above it represents the estimated trend βv_{dW} . The difference between them is a small purple wavy line labeled dW , which is labeled as the "Error term".



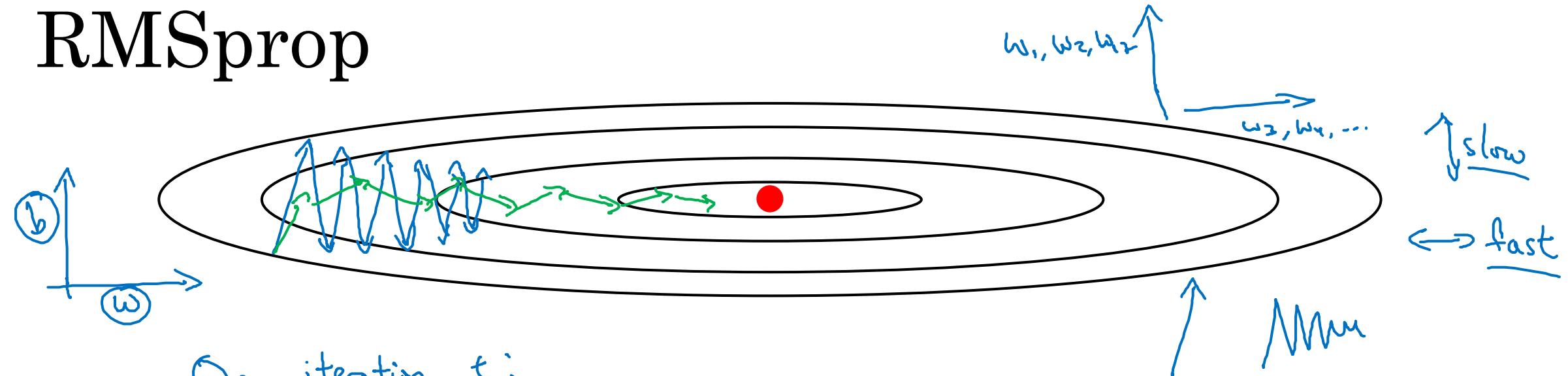
deeplearning.ai

Optimization Algorithms

RMSprop

So db would be large, and hence it would get updated slowly while W will be updated by lar

RMSprop



On iteration t :

Compute dW, db on current mini-batch

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

$\leftarrow \text{small}$

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

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

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

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

Adding this epsilon ensures that we don't divide by 0.



deeplearning.ai

Optimization Algorithms

Adam optimization algorithm

It basically combines momentum with RMSprop.

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])

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

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

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

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

We can consider V_{dw} as the running avg of the gradient

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

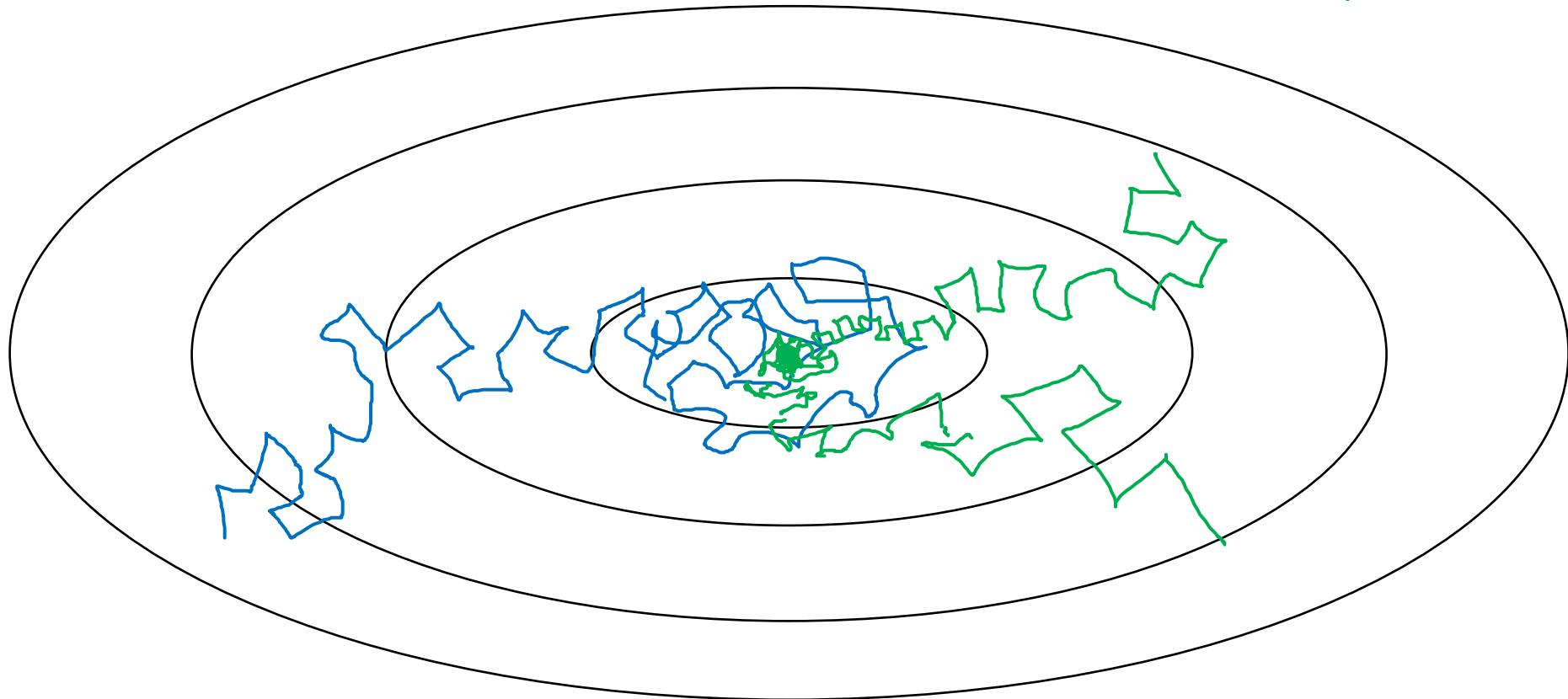
Optimization Algorithms

Learning rate decay

Slowly and slowly reduce the learning rate during later epochs .

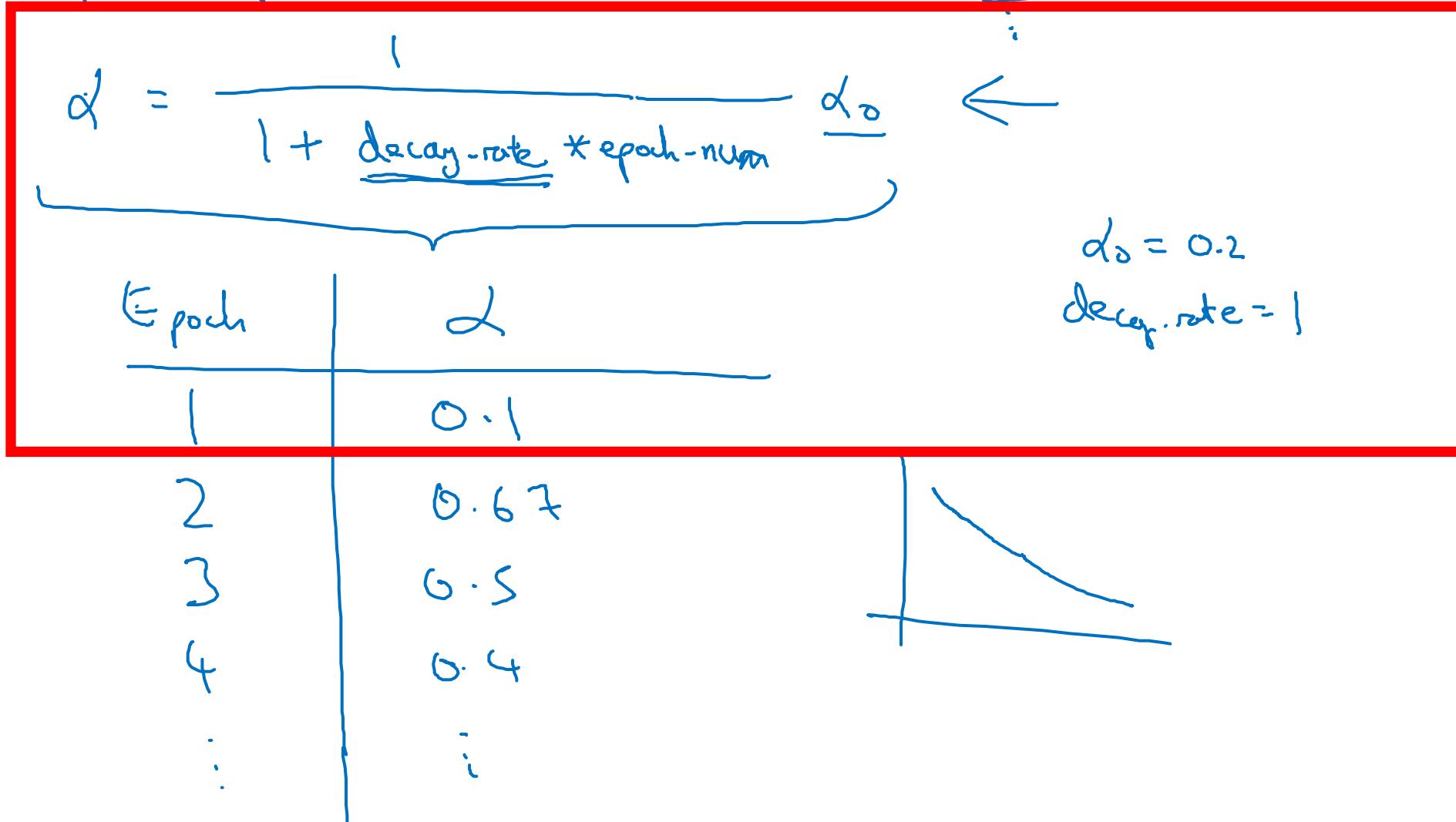
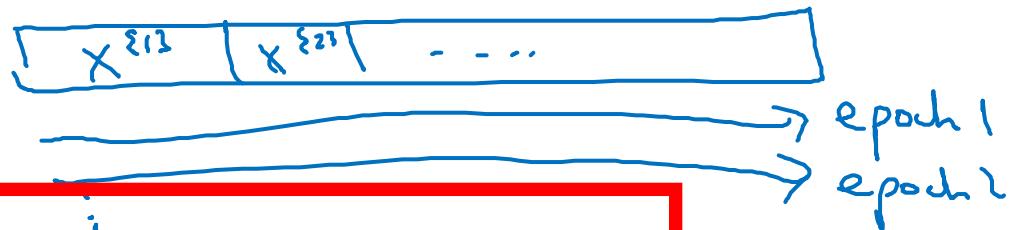
Learning rate decay

Slowly reduce λ



Learning rate decay

1 epoch = 1 pass through data.



Other learning rate decay methods

$$\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0 \quad - \text{exponentially decay.}$$

formula

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



discrete staircase

Manual decay.

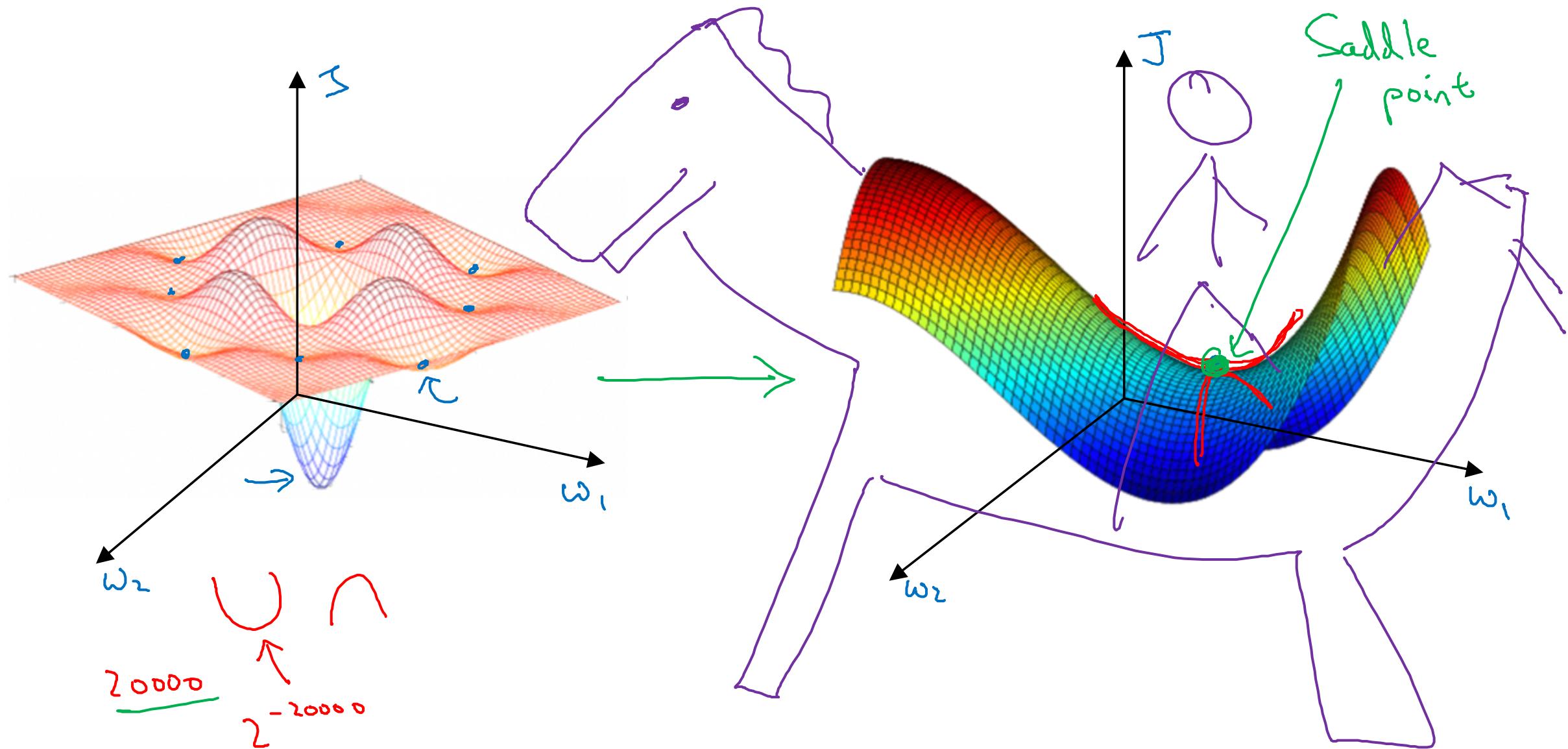


deeplearning.ai

Optimization Algorithms

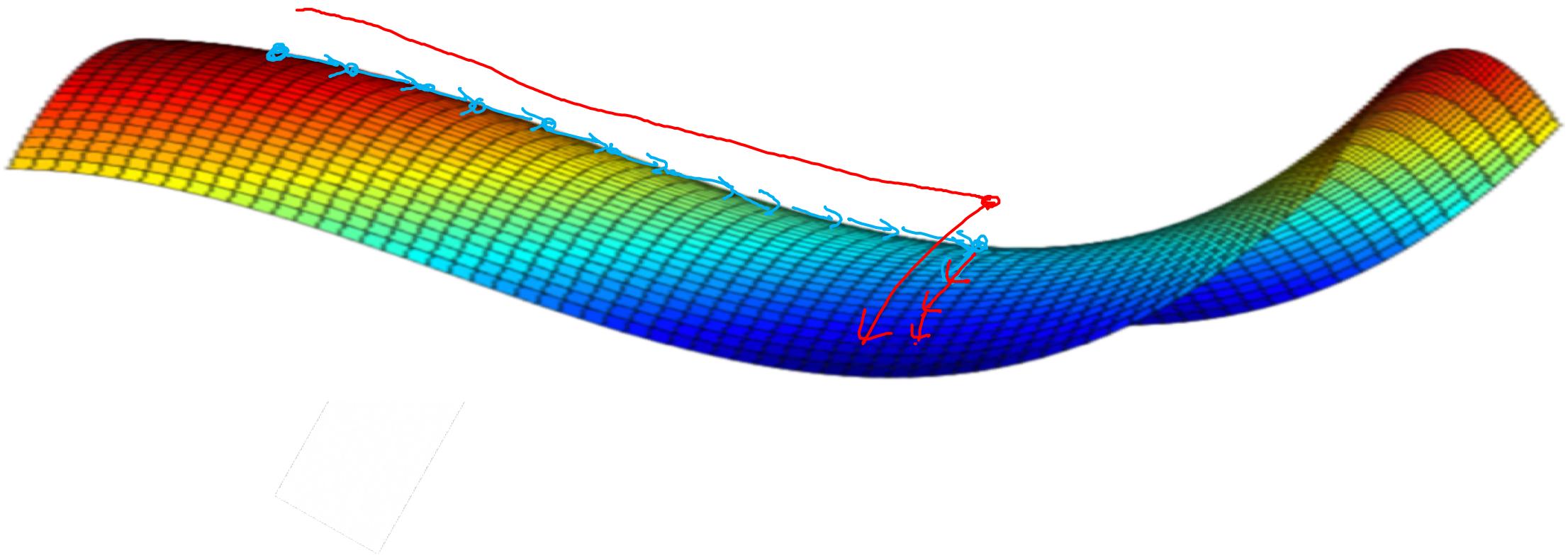
The problem of local optima

Local optima in neural networks



Andrew Ng

Problem of plateaus



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