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 One can introduce a memory factor to the gradient descent update rule. Instead of v = v - alpha * dv $X^{\{t\}}, Y^{\{t\}}$

Vectorization allows you to efficiently compute on *m* examples.

$$X = [\underbrace{x^{(1)}\ x^{(2)}\ x^{(3)} \dots x^{(1000)}}_{X^{\{1\}}\ (n_x, 1000)} \mid \underbrace{x^{(1001)} \dots x^{(2000)}}_{X^{\{2\}}\ (n_x, 1000)} \mid \dots \mid \underbrace{\dots x^{(m)}}_{X^{\{5,000\}}\ (n_x, 1000)}]$$

(n_x, m)

CURLY BRACES = minibatch number

$$Y = [\underbrace{y^{(1)}\ y^{(2)}\ y^{(3)} \dots y^{(1000)}}_{Y^{\{1\}}\ (1, 1000)} \mid \underbrace{y^{(1001)} \dots y^{(2000)}}_{Y^{\{2\}}\ (1, 1000)} \mid \dots \mid \underbrace{\dots y^{(m)}}_{Y^{\{5,000\}}\ (1, 1000)}]$$

$(1, m)$

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

5,000 mini-batches of 1,000 each

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

$x^{(i)}$

$z^{[l]}$

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

Andrew Ng

Mini-batch gradient descent

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

X, Y

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

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

$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$

$\vdots$

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

Vectorized implementation (1000 examples)

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

samples in mini-batch

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

layers

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

$W^{[l]} := W^{[l]} - \alpha dW^{[l]}, \quad b^{[l]} := b^{[l]} - \alpha db^{[l]}$

}

}

"1 epoch"

pass through training set.

Andrew Ng

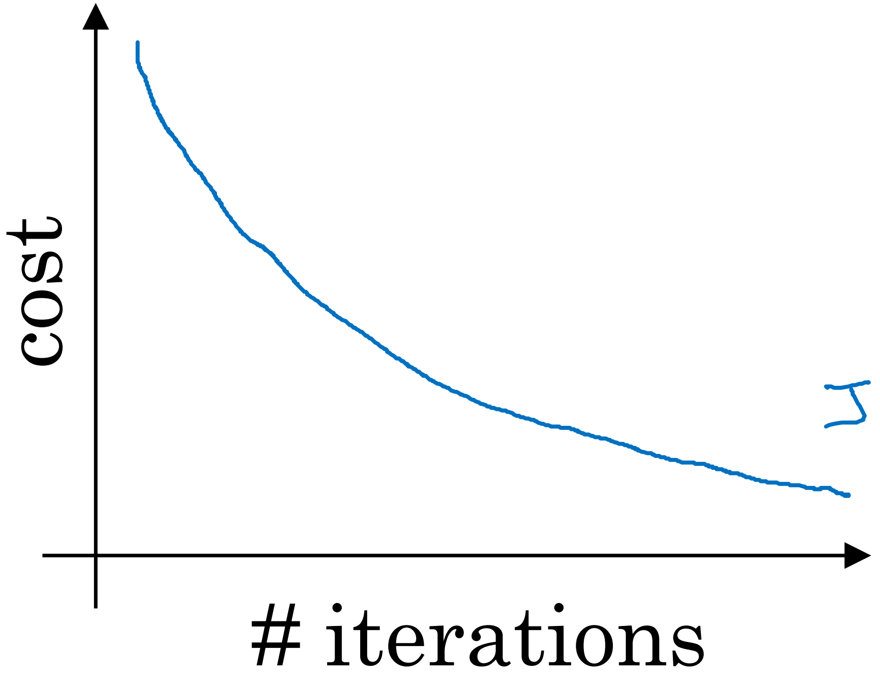
deeplearning.ai

Optimization Algorithms

Understanding mini-batch gradient descent

Training with mini batch gradient descent

Batch gradient descent



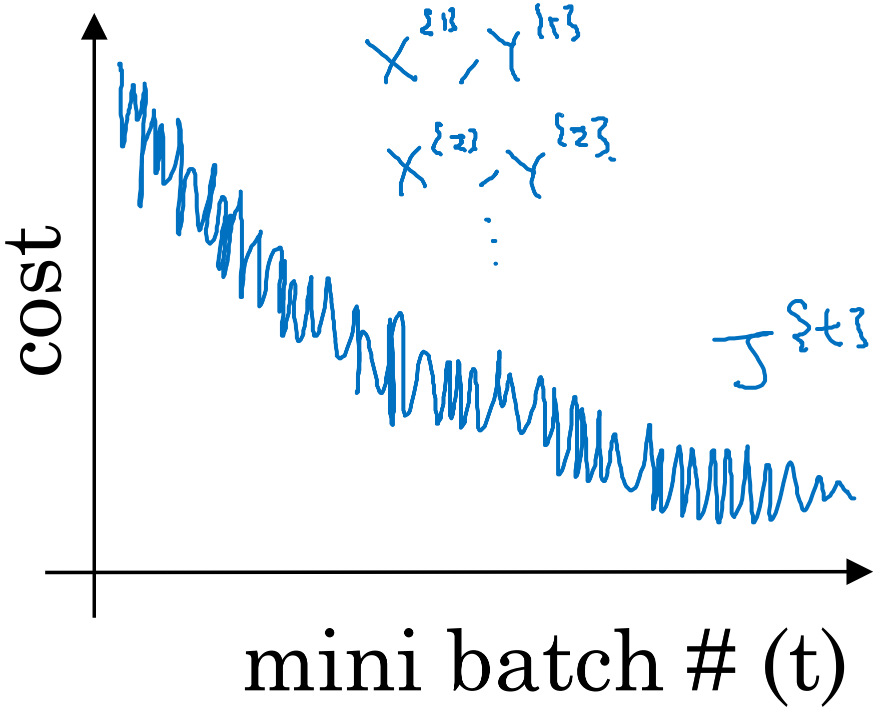
$V_1 = 0.5\, V_0 + 0.5 \cdot 30 = 15$

conv

$15/0.5 =$

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

Mini-batch gradient descent



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

Choosing your mini-batch size

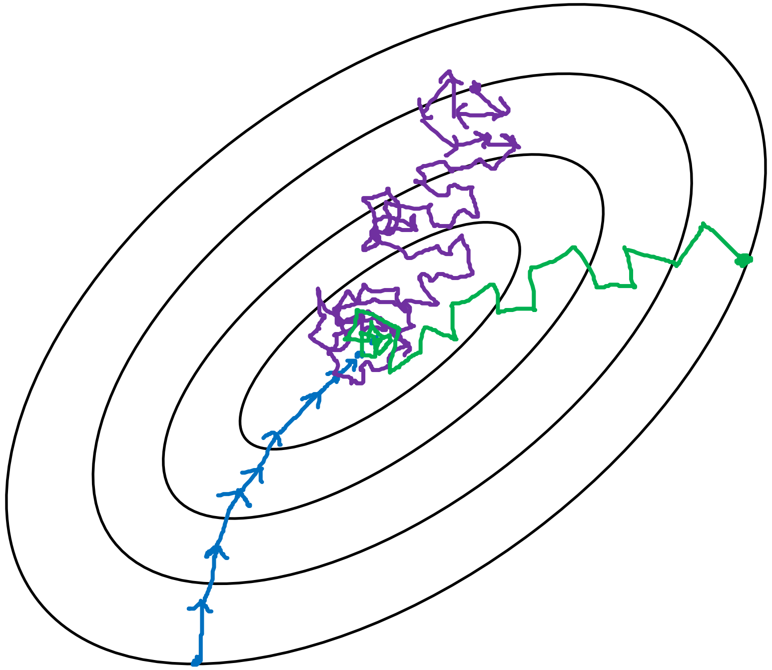
- if minibatch size = 1 => SGD : disadvantage of loosing vectorization.
- if minibatch size = m => Batch GD : disadvantage need to observe the whole dataset before updating the model.

→ 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 it own mini-batch.

$(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)}) \ldots (x^{(2)}, y^{(2)})$ mini-batch.

In practice: Somewhere in-between 1 and m



Stochastic gradient descent

↓

Lose ~~speed~~ speedup from vectorization

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

↓

Fastest learning.

- Vectorization. (~1000)
- Make progress without processing entire training set.

Batch gradient descent (mini-batch size = m)

↓

Too long per iteration

Andrew Ng

Choosing your mini-batch size

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

- Typical mini-batch sizes: powers of 2

  → $\underbrace{\underset{2^6}{64}, \underset{2^7}{128}, \underset{2^8}{256}, \underset{2^9}{512}}$ $\quad \underset{2^{10}}{1024}$

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

One can introduce a memory factor to the gradient descent update rule. The idea is to smooth out the dv using historical values. dv_smooth = beta * dv_old + (1 - beta) * dv.
By doing so, one obtains "gradient descent with momentum": W = W - alpha * dW_smooth where dW_smooth is the smoothed version of dW

deeplearning.ai

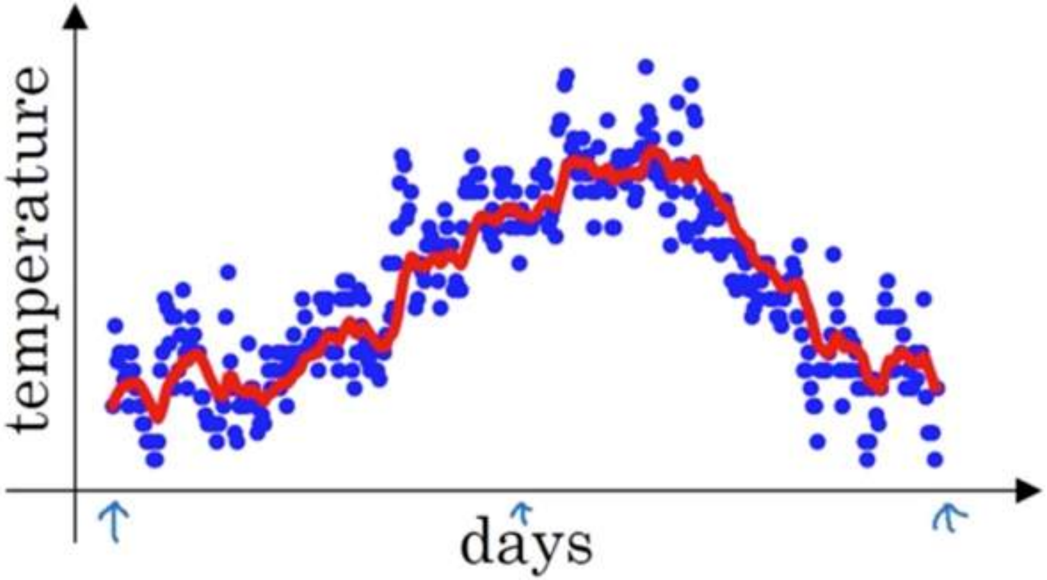
Optimization Algorithms

Exponentially weighted moving averages

→ At the basis of optimization algorithms faster than GD.

Temperature in London

$\theta_1 = 40°F$ 4°C ←
$\theta_2 = 49°F$ 9°C
$\theta_3 = 45°F$ ⋮
⋮
$\theta_{180} = 60°F$ 15°C
$\theta_{181} = 56°F$ ⋮
⋮



$V_0 = 0$
$V_1 = 0.9 V_0 + 0.1 \theta_1$
$V_2 = 0.9 V_1 + 0.1 \theta_2$
$V_3 = 0.9 V_2 + 0.1 \theta_3$
⋮
$V_t = 0.9 V_{t-1} + 0.1 \theta_t$

Andrew Ng

Exponentially weighted (moving) averages

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

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

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

$\beta = 0.5$: ≈ 2 days

V_t as approximately average over $\approx \frac{1}{1-\beta}$ days' temperature.

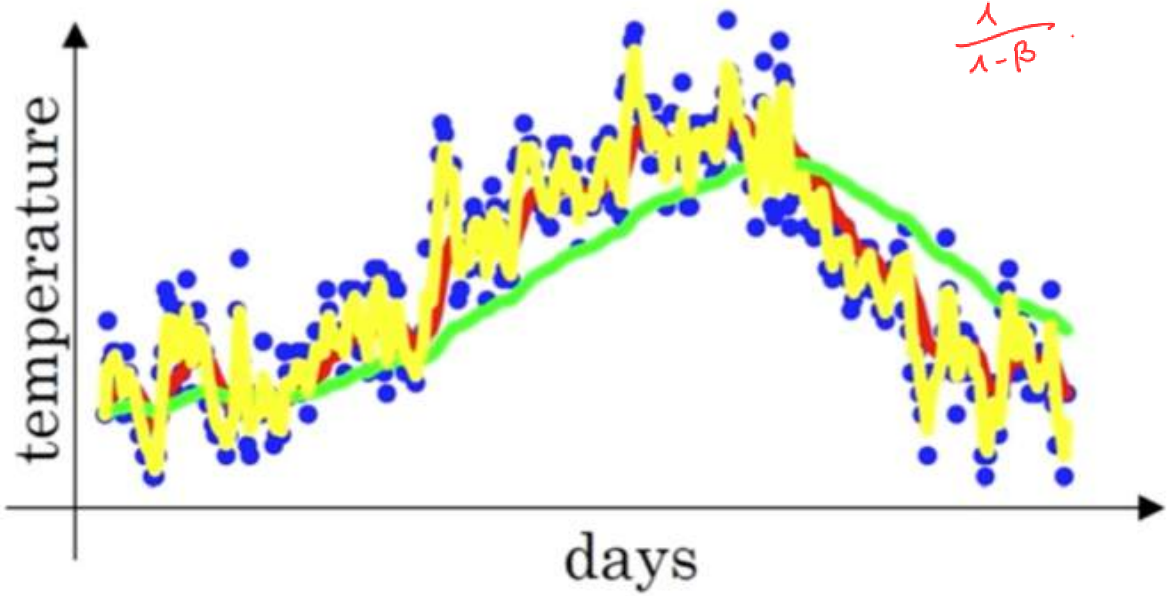
exponential averaging with forgetting $(1-\beta)$ has an exponential decay of $\frac{1}{1-\beta}$.

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

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

For $0.9 \rightarrow 0.9^x \approx 0.35 \rightarrow x = \log_{0.9} 0.35 = 10$

$0.98^x \approx 0.35 = x = 50$



deeplearning.ai

Optimization Algorithms

Understanding exponentially weighted averages

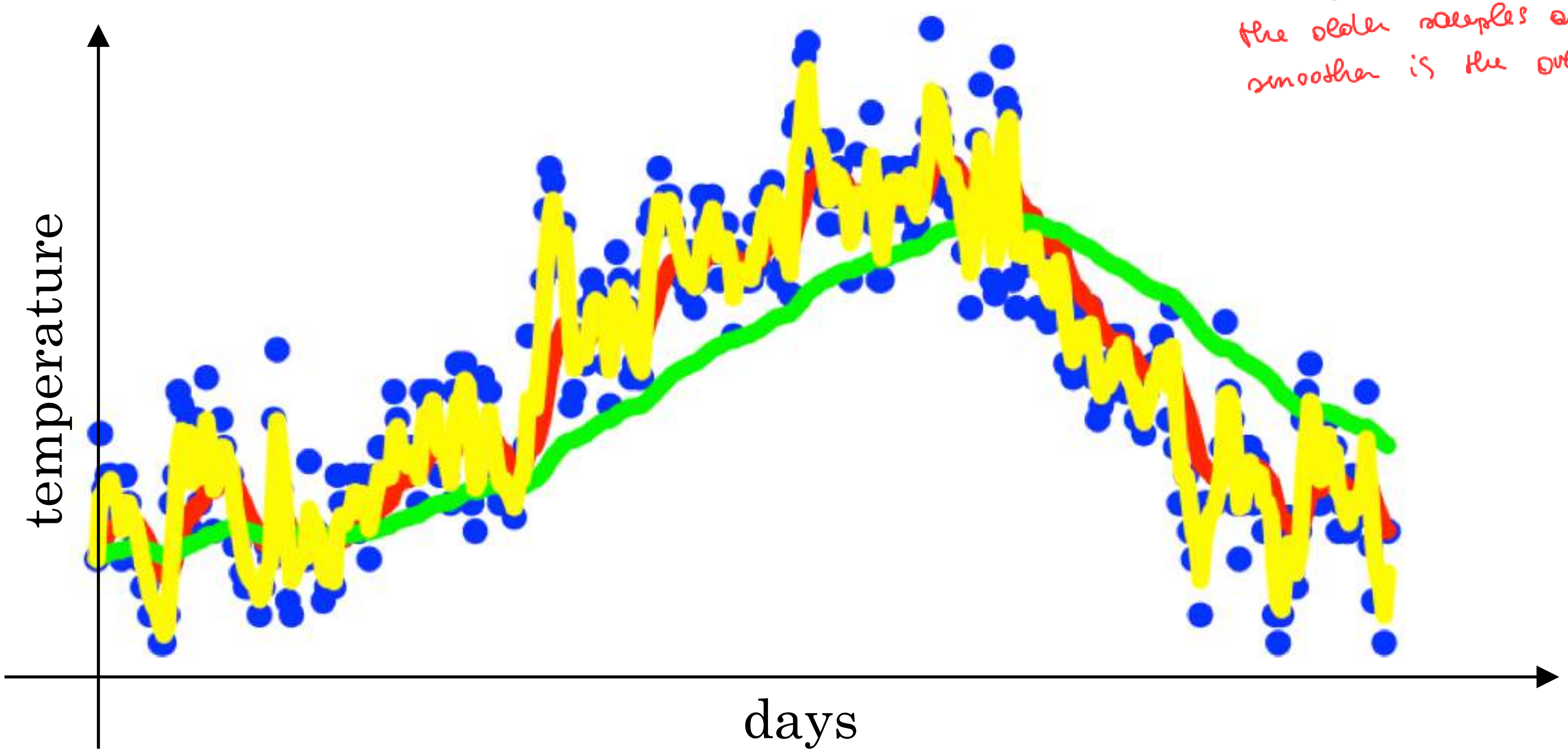
Exponentially weighted averages

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

$\beta = 0.9$ 0.98 0.5

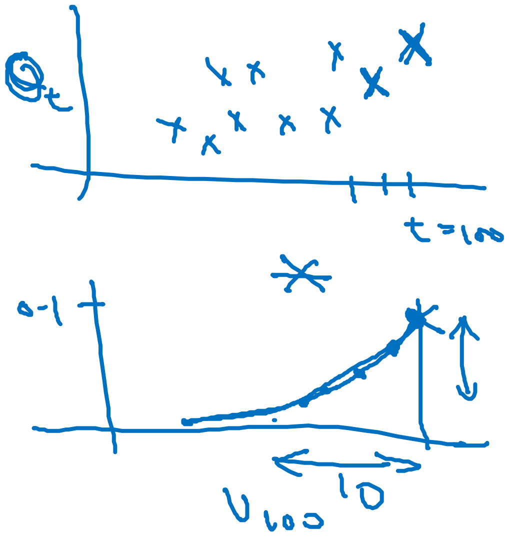
β = MEMORY

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



Exponentially weighted averages

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



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

$$\varepsilon = 1-\beta$$

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

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

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

...

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

$$v_{100} = 0.1\,\theta_{100} + 0.9\,\cancel{v_{99}}\,(0.1\,\theta_{99} + 0.9\,\cancel{v_{98}})$$

$$= 0.1\,\theta_{100} + 0.1 \times 0.9 \cdot \theta_{99} + 0.1\,(0.9)^2\,\theta_{98} + 0.1\,(0.9)^3\,\theta_{97} + 0.1\,(0.9)^4\,\theta_{96} + \dots$$

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

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

$$0.98\ ?$$

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

Implementing exponentially weighted averages

$$v_0 = 0$$
$$v_1 = \beta v_0 + (1-\beta)\,\theta_1$$
$$v_2 = \beta v_1 + (1-\beta)\,\theta_2$$
$$v_3 = \beta v_2 + (1-\beta)\,\theta_3$$
$$\ldots$$

$V_\theta := 0$

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

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

$\vdots$

$V_\theta = 0$

Repeat {

Get next θ_t

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

}

Andrew Ng

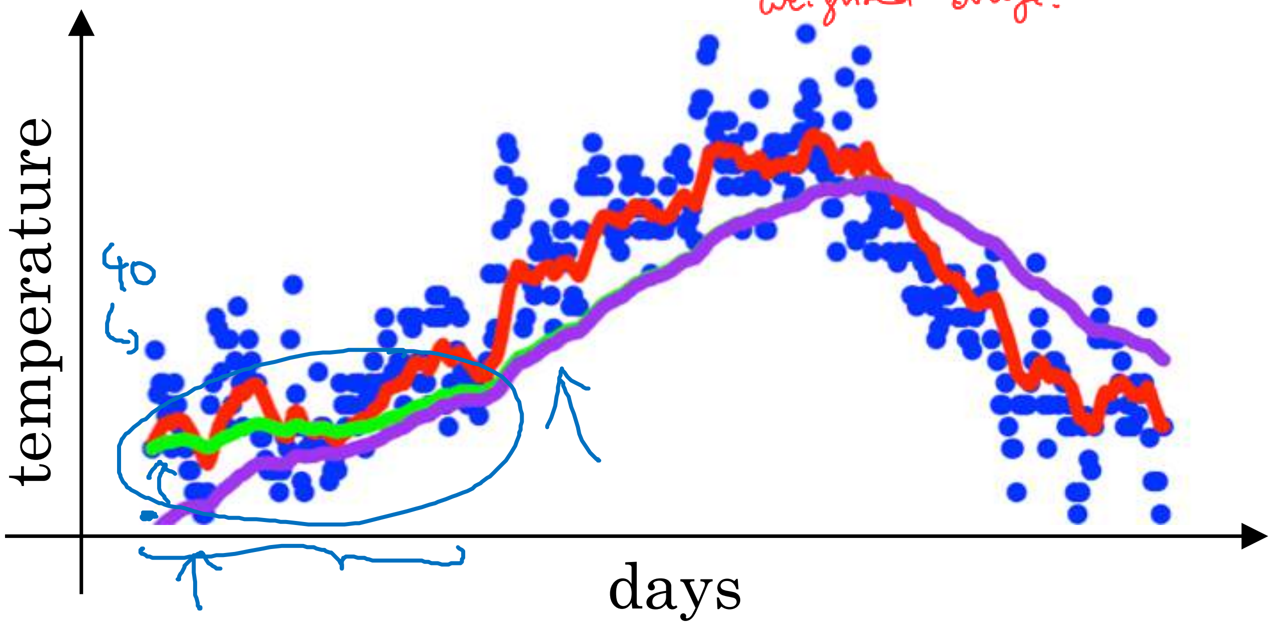
deeplearning.ai

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 $(1-\beta^t)$:

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

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

$v_0 = 0$

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

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

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

$= 0.0196\,\theta_1 + 0.02\,\theta_2$

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

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

$$\frac{v_2}{0.0396} = \frac{0.0196\,\theta_1 + 0.02\,\theta_2}{0.0396}$$

- In practice this problem is often ignored.

deeplearning.ai

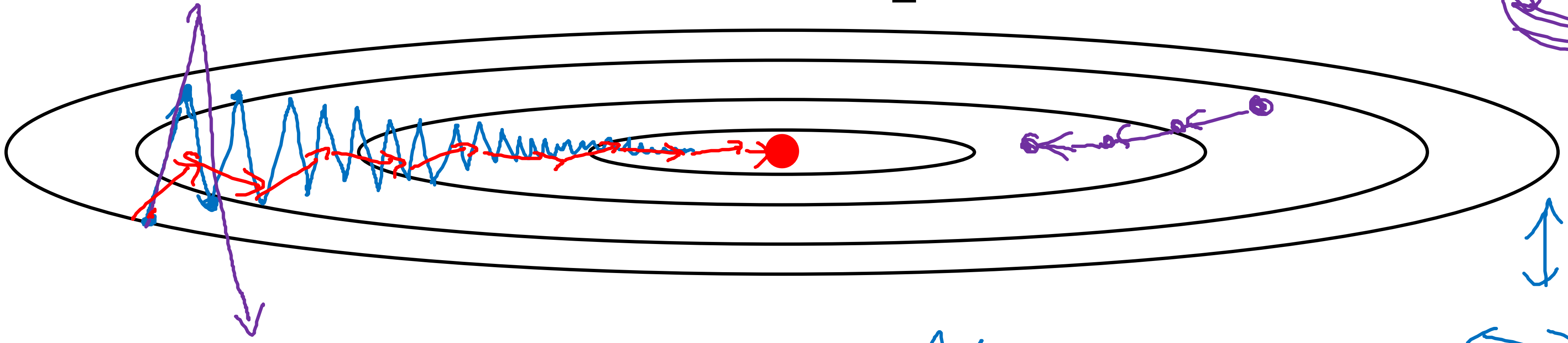
Optimization Algorithms

Gradient descent with momentum

= GD with smoothed gradient (exponentially weighted average).

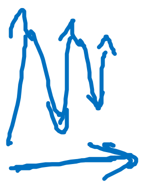
Gradient descent example

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



slower learning

faster learning.



Momentum:

On iteration t:

Compute dW, db on current mini-batch.

$$\underbrace{V_{dW}}_{\text{smoothed } dW} = \beta V_{dW} + (1-\beta)\, dW$$

$$\underbrace{V_{db}}_{\text{smoothed } db} = \beta V_{db} + (1-\beta)\, db$$

friction — velocity — acceleration

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

$W := W - \alpha V_{dW}$, $b := b - \alpha V_{db}$

Implementation details

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

On iteration t:

Compute dW, db on the current mini-batch

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

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

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

$\underbrace{\underline{V_{dw}} = \beta V_{dw} + \quad dW}$

NOTE: in the literature, this version is often used. This would correspond to the old formulation if β was scaled by $(1-\beta)$ instead.

$V_{dw} = \beta \cdot V_{dw} + (1-\beta) dw$

$\frac{V_{dw}}{1-\beta} = \frac{\beta}{1-\beta} V_{dw} + dw = \hat{\beta} V_{dw} + dw$

... ? ... Ng prefers the previous

$\cancel{\frac{V_{dw}}{1-\beta^t}}$

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

Hyperparameters: α, β

learning rate — momentum

$\underline{\beta = 0.9}$

corresponds to average over last ≈ 10 gradients

Andrew Ng

deeplearning.ai

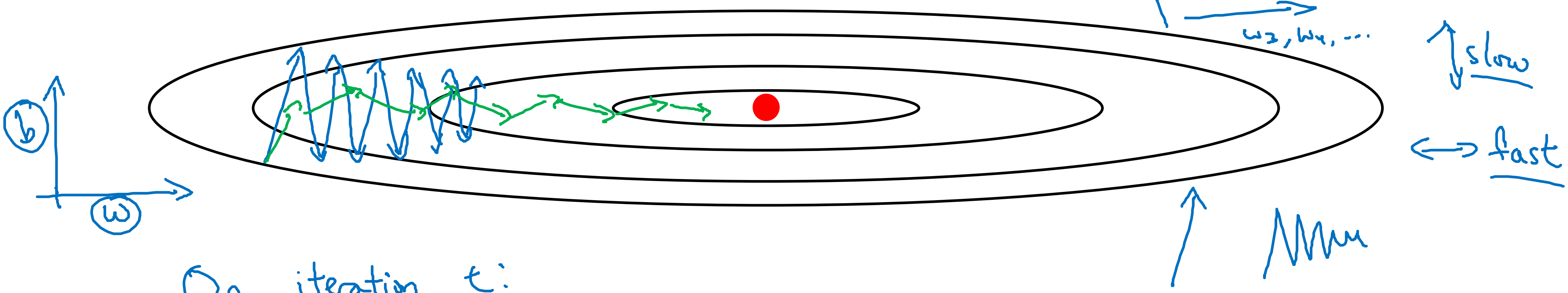
Optimization Algorithms

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

element-wise

$$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)\, dW^2 \leftarrow \text{small}$$

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

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}+\varepsilon}} \qquad b := b - \alpha \frac{db}{\sqrt{S_{db}+\varepsilon}}$$

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

$\varepsilon = 10^{-8}$

the term $S_{d\theta}$ at the denominator makes the update along θ larger/smaller if θ is smaller/larger.

Optimization Algorithms

Adam optimization algorithm

"ADAptive Moment estimation"

Adam optimization algorithm

- Mixes momentum and RMS in the update rule.

- momentum helps smoothing out gradient oscillations, while RMS allows to dampen "strong directions" (directions in which the gradient is big and enhance "weak directions").

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

On iteration t:

Compute dW, db using current mini-batch

$V_{dw}=\beta_1 V_{dw}+(1-\beta_1)dW,\quad V_{db}=\beta_1 V_{db}+(1-\beta_1)db$ ← "moment" β_1

$S_{dw}=\beta_2 S_{dw}+(1-\beta_2)dW^2,\quad S_{db}=\beta_2 S_{db}+(1-\beta_2)db$ ← "RMSprop" β_2

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

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

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

← bias correction

$$W:=W-\alpha\frac{\overbrace{V^{corrected}_{dw}}^{\text{momentum}}}{\underbrace{\sqrt{S^{corrected}_{dw}}+\varepsilon}_{\text{rms}}}\qquad b:=b-\alpha\frac{V^{corrected}_{db}}{\sqrt{S^{corrected}_{db}}+\varepsilon}$$

idea:
- correct $d\theta$ with both momentum and RMS → $V_{d\theta}, S_{d\theta}$
- perform the update mixing the two

$\theta:=\theta-\alpha\frac{V_{d\theta}}{\sqrt{S_{d\theta}}+\varepsilon}$ ← moment ← RMS

in RMS this is $d\theta$

in moment this is 1

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

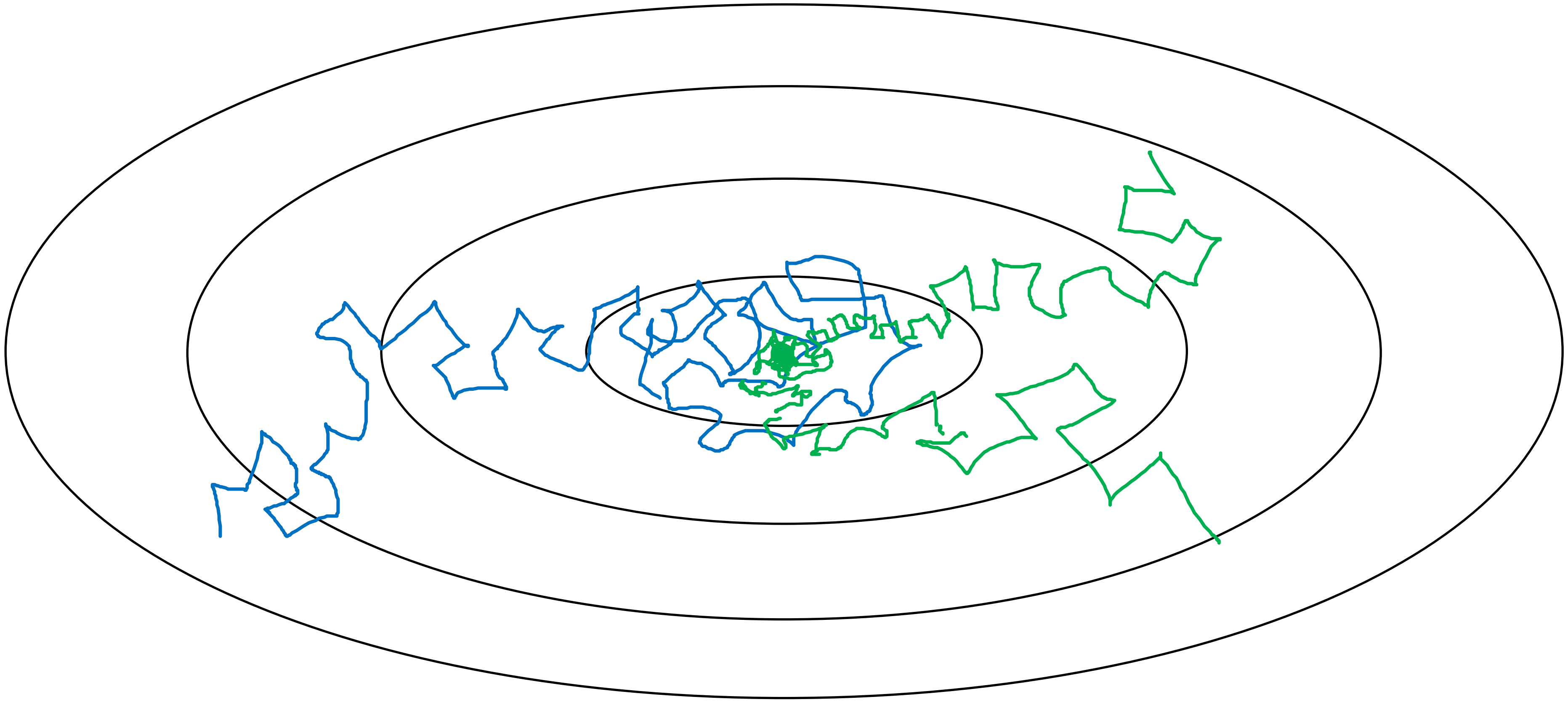
Optimization Algorithms

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



Andrew Ng

Learning rate decay

Idea: reduce the learning rate at every epoch.

$X^{\{1\}}$ | $X^{\{2\}}$ | ....

→ epoch 1

→ epoch 2

⋮

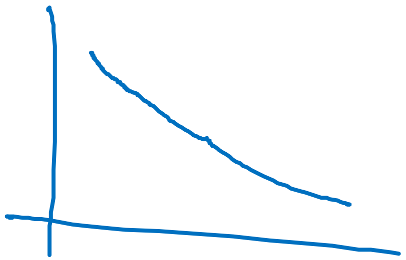
1 epoch = 1 pass through data.

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

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
⋮	⋮

$\alpha_0 = 0.2$

decay.rate = 1

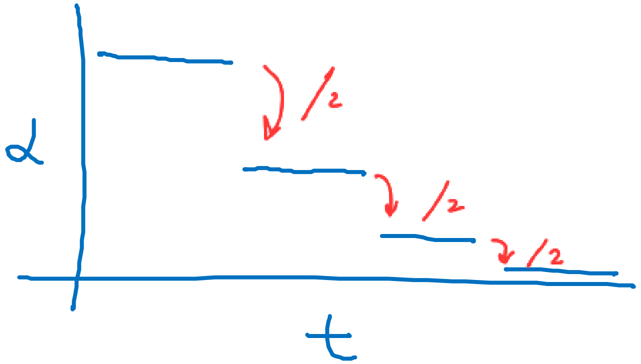


Other learning rate decay methods

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.

Andrew Ng

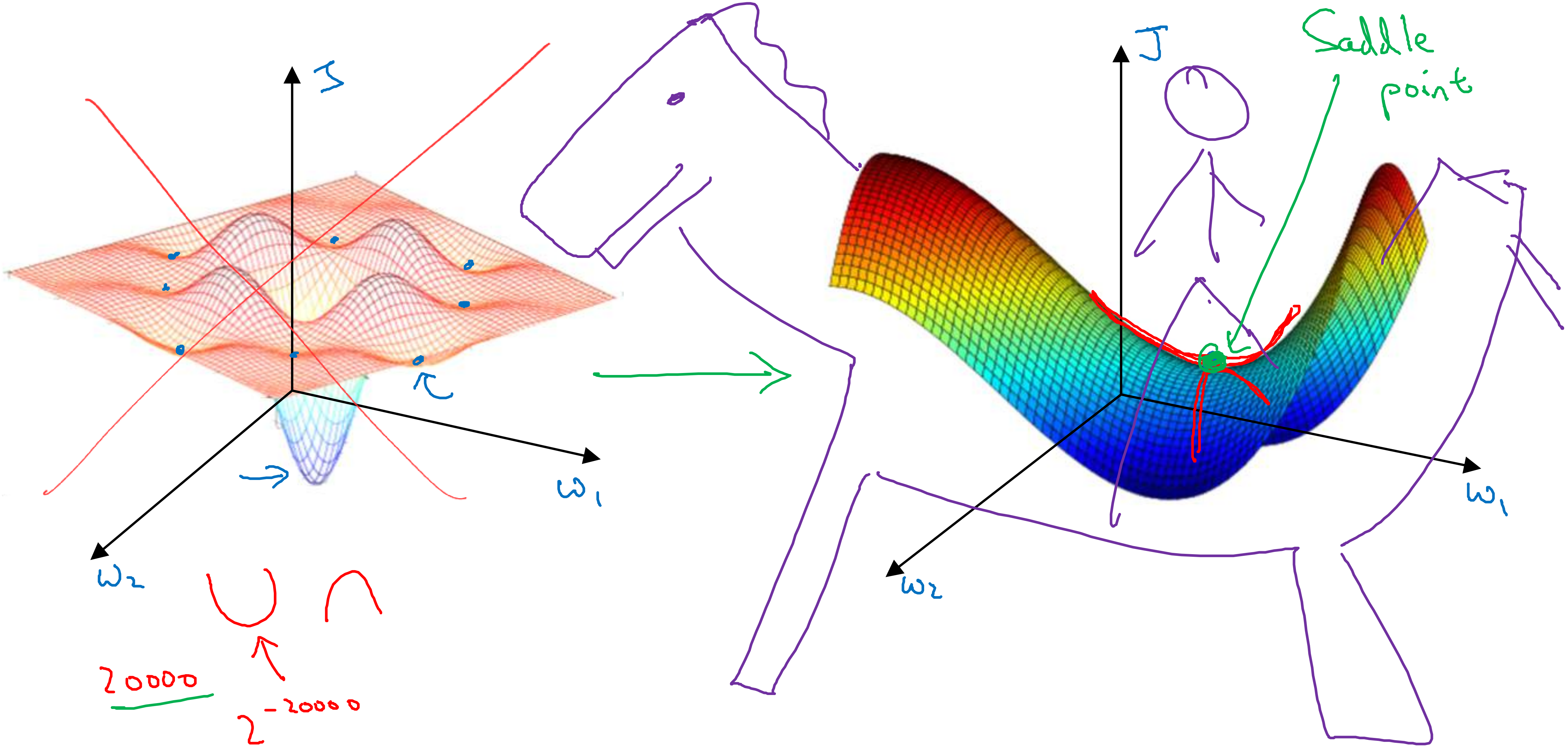
Optimization Algorithms

deeplearning.ai

The problem of local optima

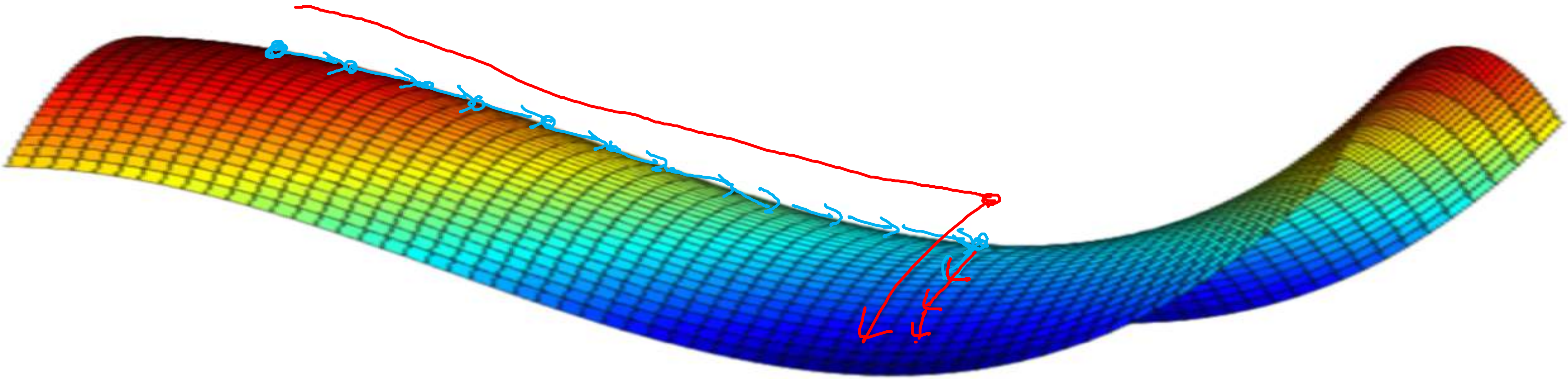
Local optima in neural networks

- In many dimensions, most local minima are saddle points
(because over many dimensions there would be some with grad>0 and others with 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