

Batch

Vectorization allows you to efficiently compute on m examples.

$$X = \underbrace{[x^{(1)} \ x^{(2)} \ x^{(3)} \ \dots \ x^{(1000)}]}_{\{(n_x, m)\}} \left| \underbrace{x^{(1001)} \ \dots \ x^{(2000)}}_{\{x^{(2)}\} (n_x, 1000)} \right. \dots \left| \underbrace{\dots \ x^{(m)}}_{\{x^{(5000)}\} (n_x, 1000)} \right]$$

$$Y = \underbrace{[y^{(1)} \ y^{(2)} \ y^{(3)} \ \dots \ y^{(1000)}]}_{\{(1, m)\}} \left| \underbrace{y^{(1001)} \ \dots \ y^{(2000)}}_{\{Y^{(2)}\} (1, 1000)} \right. \dots \left| \underbrace{\dots \ y^{(m)}}_{\{Y^{(5000)}\} (1, 1000)} \right]$$

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

5,000 mini-batches of 1,000 each

Mini-batch t : $\underbrace{X^{(t)}, Y^{(t)}}_{\{(i)\}}$

$$\begin{aligned} & X^{(i)} \\ & z^{(i)} \\ & X^{(t)}, Y^{(t)} \end{aligned}$$

Andrew Ng

Mini-Batch

1 epoch - single pass through training set

Mini-batch gradient descent

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

Forward prop on $X^{(t)}$.

$$\begin{aligned} Z^{(t)} &= W^{(t)} X^{(t)} + b^{(t)} \\ A^{(t)} &= g^{(t)}(Z^{(t)}) \end{aligned}$$

Vectorized implementation
(1000 examples)

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

$$\text{Compute cost } J = \frac{1}{1000} \sum_{i=1}^{\sqrt{1000}} l(A^{(t)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{j=1}^n \|W^{(t)}\|_F^2.$$

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

$$W^{(t)} = W^{(t)} - \alpha \nabla J^{(t)}, \quad b^{(t)} = b^{(t)} - \alpha \nabla J^{(t)}$$

3

"1

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

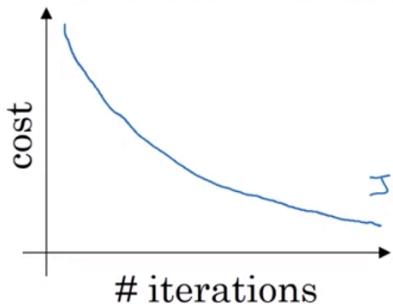
$$X, Y$$

Large dataset - mini batch way faster

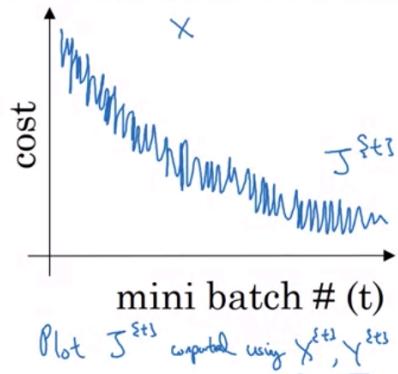
Understanding Mini-batch

- May not decrease on every mini-batch, unlike batch
- Why? Some batch may be harder (higher costs)

Batch gradient descent



Mini-batch gradient descent



Stochastic: loss speedup through vectorization
Batch: too long per iteration

- Small-training set (<2000) use batch
- Typical mini-batch: 64, 128, 256, 512
- Make sure mini batch fit in CPU/GPU mem
- Generally Try a few and pick the most efficient.

Exponentially weighted average

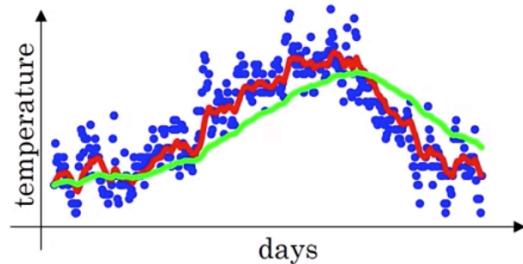
Exponentially weighted averages

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

$\beta = 0.9$: ≈ 10 days' temporal.
 $\beta = 0.98$: ≈ 50 days

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

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



Understanding Exponentially weighted average

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

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

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

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

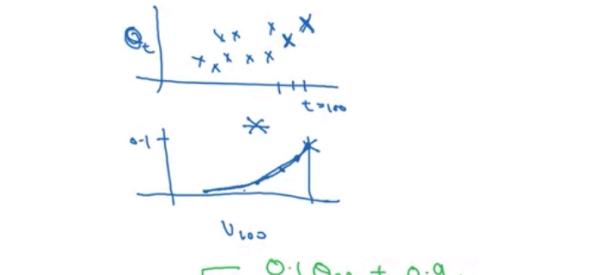
...

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

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

$$\frac{(1-\beta)^{10}}{0.9} = \frac{1}{e}$$

W

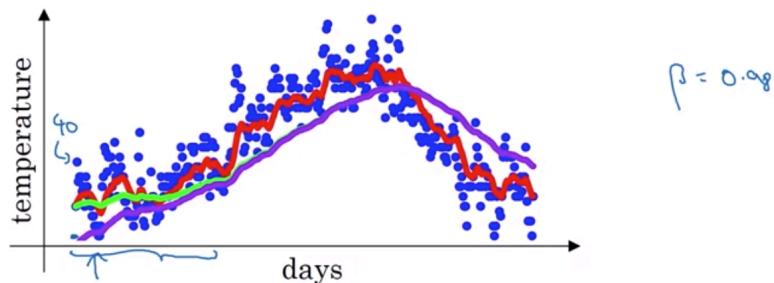


when beta = 0.9, roughly 10 days, when beta = 0.98, 50 days

Bias Correction

The purple line deviate from ideal (green) initially, just set $v_t / (1 - \beta)$

Bias correction



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

$$v_0 = 0$$

$$v_1 = 0.98 v_0 + 0.02 \theta_1$$

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

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

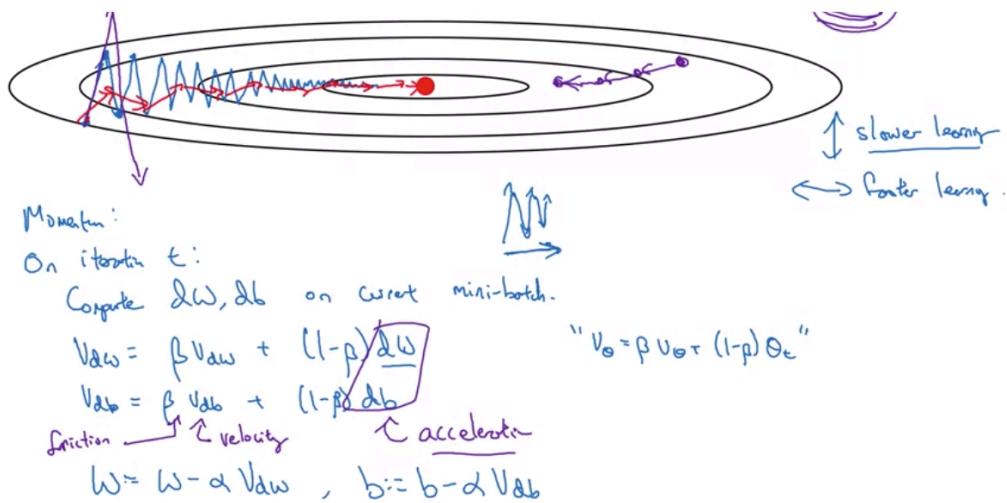
$$= 0.0196 \theta_1 + 0.02 \theta_2$$

$$\begin{aligned} &\frac{v_t}{1 - \beta^t} \\ t=2: \quad 1 - \beta^t &= 1 - (0.98)^2 = 0.0396 \\ \frac{v_2}{0.0396} &= \frac{0.0196 \theta_1 + 0.02 \theta_2}{0.0396} \end{aligned}$$

Andrew Ng

Momentum

v_{dw} is velocity



Implementation details

$$v_{dW} = 0, \quad v_{db} = 0$$

On iteration t :

Compute dW, db on the current mini-batch

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

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

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

$$\frac{v_{dW}}{1 - \beta^t}$$

Hyperparameters: α, β

$$\uparrow \uparrow$$

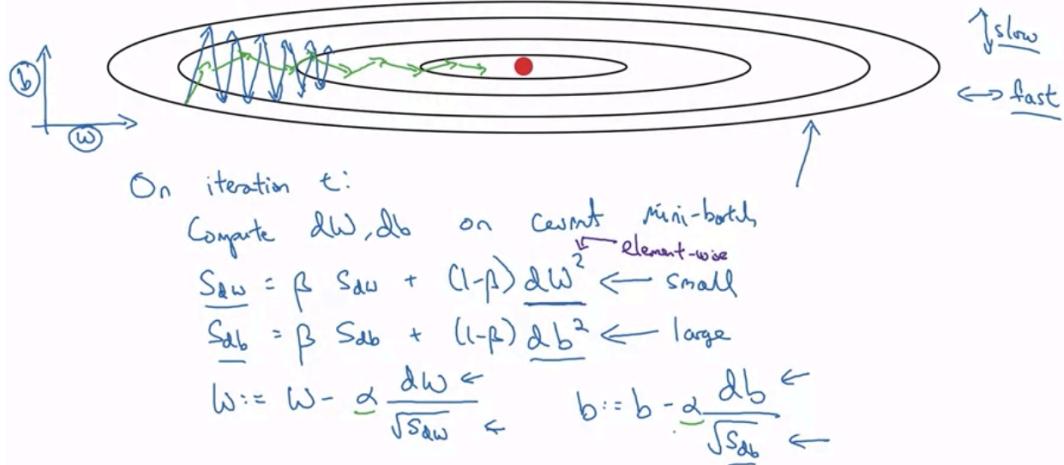
$$\beta = 0.9$$

average over loss ≈ 10 gradients

RMSProp

- Can use larger learning rate without diverging
- Root mean square

RMSprop



Adam

Adam optimization algorithm

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

On iteration t :

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

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

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

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

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

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

Hyperparameters choice:

α : needs to be tune

β_1 : 0.9 ($d\omega$)

β_2 : 0.999 ($d\omega^2$)

ϵ : 10^{-8}

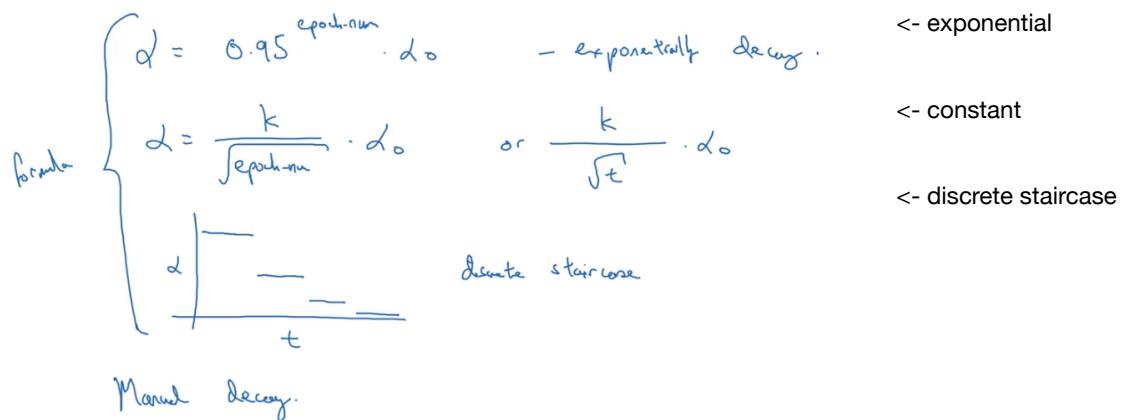
Learning Rate Decay. (Not usually high on the list for optimizing)

Learning rate decay

1 epoch = 1 pass through data.

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

Other learning rate decay methods



Local Optima

- Very less likely to stuck in local optima
- More saddle points.
- **Plateaus** makes learning slow

