- Mini-batch gradient descent
  - Batch VS. mini-batch gradient descent

$X = [x^{[1]}, x^{[2]}, x^{[3]}, \cdots x^{[1000]}, x^{[1001]}, x^{[1002]}, \cdots | \quad | x^{(m)}]$

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

mini-batch:

baby training set

$X = [\underbrace{x^{[1]}, x^{[2]}, x^{[3]}, \cdots x^{[1000]}}_{X^{\{1\}}}, \underbrace{x^{[1001]}, x^{[1002]}, \cdots |}_{X^{\{2\}}} \quad | x^{(m)}]$ $X^{\{3\}}, \cdots$

do the same for Y accordingly

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

- Algorithm

for $t = 1, \cdots 5000$: {

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

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

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

⋮ ⋮ ⋮

} Vectorized implementation (e.g. 1000 examples)

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

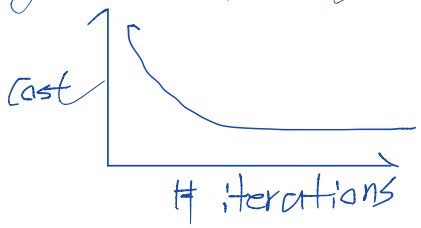
Back prop to compute gradients of $J^{\{t\}}$

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

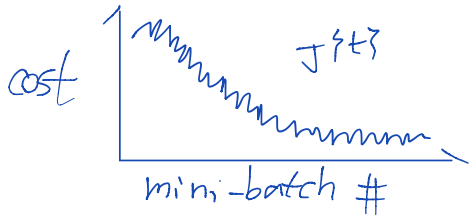
}

"1 epoch": a single pass through training set
Much faster than batch gradient

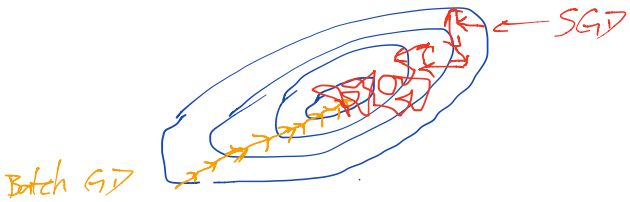
- Understanding mini-batch gradient descent
  - Batch gradient descent



  - Mini-batch gradient descent



  - choosing your mini-batch size
    - If mini-batch size = m
      → batch gradient descend
    - If mini-batch size = 1
      → Stochastic gradient descent
      Won't ever converge, always oscillates around the min cost.
    - In practice: 1< <m

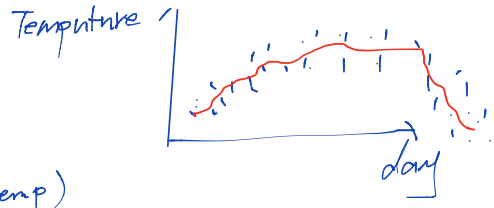


- Compare
  - batch GD: Too long per iteration
  - SGD: lose speedup from vectorization
    Making progress every iteration
    can reduce noise by using a small learning rate
  - mini-batch: Fastest Learning !!!
    - Vectorization
    - make progress w/o wait too long

- Guidlines:
  - If small training set: Use batch GD
  - Typical mini-batch size: 64, 128, 256, 512, ...
  - Make sure each mini-batch fit CPU/GPU memory

- Exponentially weighed averages
  (Faster than gradient descent)

e.g: Temperate in London



$V_0 = 0$ (day 0 temp)

$V_1 = 0.9 V_0 + 0.1 V_1$ (day 1 temp)

$V_2 = 0.9 V_1 + 0.1 V_2$

⋮ ⋮ ⋮

—: exponentially weighted temperature

- Exponentially weighted averages

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

V_t: can be viewed as approx. average over $\approx \frac{1}{1-\beta}$ days temperature

e.g: $\beta = 0.9$ → avg. over last 10 days temp

$\beta = 0.98$ → ------ 50 days ...

- Understanding Exponentially weighted averages

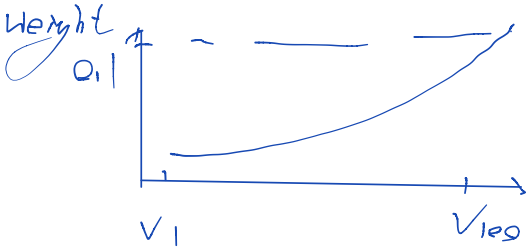
$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

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

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

⋮ ⋮ ⋮

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



$(1-\varepsilon)^{\frac{1}{\varepsilon}} \approx \frac{1}{e}$

$0.9^{10} \approx 0.35 \approx \frac{1}{e}$: takes about 10 days of previouse temp into account

- Implementation

$V_0 = 0$

Repeat {

get next θ_t

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

} (takes little memory)

- Bias Correction In Exponentially Weighted Average
  - Bias Correction

$V_0 = 0$

$V_1 = \underbrace{0.98 V_0}_{=0} + 0.02\theta_1$

$V_2 = 0.98 V_1 + 0.02\theta_2$

⋮ ⋮ ⋮

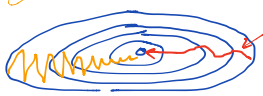
Bias raises from the starting boundary !

- Correction

$$V_t \longrightarrow \frac{V_t}{1-\beta^t}$$

- Gradient Descent Momentum

normal gradient descent → GD with moment



Want: faster learning along horizontal axis
slower learning along vertical axis

- Moment

on iteration t:

Compute dw, db on current batch

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

$$V_{db} = \beta V_{db} + (1-\beta)\, db$$

$$w = w - \alpha V_{dw}$$

$$b = b - \alpha V_{db}$$

- Understanding:

Average out fast oscillating gradient descent steps → more efficient!

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

derivative term
↳ Acceleration !

momentum
↳ velocity
β: friction

JUST LIKE
rolling a ball !!

hyperparameters: α, β

RMSprop

On iteration t:

Compute db, dw on current mini-batch:

$$S_{dw} = \beta S_{dw} + (1-\beta) dw^2 \leftarrow \text{element wise}$$
$$S_{db} = \beta S_{db} + (1-\beta) db^2$$
$$w := w - \alpha \frac{dw}{\sqrt{S_{dw}}}$$
$$b := b - \alpha \frac{db}{\sqrt{S_{db}}}$$

- Understanding

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

Want to be large to learn fast

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

Want to be small to learn slow to reduce oscillation

from the e.g.

$db >> dw$

$\hookrightarrow S_{db} >> S_{dw}$

$\hookrightarrow \frac{db}{\sqrt{S_{db}}}, \frac{dw}{\sqrt{S_{dw}}}$ Damp The Oscillations!!

Sidenote

$S_{db} \to S_{db} + \varepsilon \quad \varepsilon << 1$

$S_{dw} \to S_{dw} + \varepsilon$

To improve stability (divide by 0)

- Adam Optimization ~~Algorithm~~

Adaptive Moment Estimation (Adam)

Momentum + RMS + GD = Adam

↑ 1st momentum ↑ 2nd momentum

- ~~Algorithm~~

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

On iteration t:

Compute dw, db using current batch:

$V_{dw} = \beta_1 V_{dw} + (1 - \beta_1) dw$

$$V_{db} = \beta_1 V_{db} + (1-\beta_1)\, db$$
$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\, dw^2$$
$$S_{db} = \beta_2 S_{db} + (1-\beta_2)\, db^2$$

Bias correction:

$$V_{dw}^c = \frac{V_{dw}}{1-\beta_1^t}$$
$$V_{db}^c = \frac{V_{db}}{1-\beta_1^t}$$
$$S_{dw}^c = \frac{S_{dw}}{1-\beta_2^t}$$
$$S_{db}^c = \frac{S_{db}}{1-\beta_2^t}$$

$$W := W - \alpha \frac{V_{dw}^c}{\sqrt{S_{dw}^c} + \varepsilon}$$
$$b := b - \alpha \frac{V_{db}^c}{\sqrt{S_{db}^c} + \varepsilon}$$

Commonly used algorithm, Very effective!

- Hyperparameter choice
  - α: needs to be tuned
  - β_1: Common choice 0.9 (dw)
  - β_2: recommended by literature 0.999 (dw^2)
  - ε: doesn't matter too much, 10^{-8}

- Learning Rate Decay

  slowly reduce the learning rate over time

mini-batch GD:  use learning rate decay to reduce oscillations!

1 epoc : 1 pass through all data

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

e.g: $\alpha_0 = 0.2$ decay rate $= 1$

Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
⋮	⋮

e.g. $\alpha = 0.95^{\text{epoch\_num}} \cdot \alpha_0$

$\alpha = \frac{k}{\sqrt{\text{epoch\_num}}} \cdot \alpha_0$ or $\frac{k}{\sqrt{t}} \alpha_0$

OR: Manual Decay

- The problem of Local Optima

  There are more saddle points than local Optimal in ML practice, especially for high dim, the chance for local optimal is very low!

- Problem of plateaus (saddle points)
  - slow down learning rate