Optimization algorithms

Mini-batch gradient descent

(GD we had before is called "Batch Gradient Descent) → all data computed at once

→ Vectorization allows us to efficiently compute on m examples

$X = [x^{(1)}, x^{(2)}, \dots, x^{(m)}]$ (n_x, m)

$Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$ $(1, m)$

↳ can still be slow if m is large

↳ **What if $m = 5'000'000$?**

→ mini-batch GD will perform much better on large data sets

↳ possible to get faster algorithm → split training set into smaller mini-batches

each mini-batch has ~ 1'000 training examples

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

new notation for mini-batches → $X^{\{1\}}$ $(n_x, 1000)$, $X^{\{2\}}$ $(n_x, 1000)$ → for $m = 5'000'000$ we get 5000 mini-batches of size 1000

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

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

How to run?

for $t = 1 \dots 5000$ (One pass through our training set)

- Forward Prop on $X^{\{t\}}$ (Vectorized implementation)

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\vdots$$
$$A^{[L]} = g^{[L]}(Z^{[L]})$$

- Compute cost of one mini-batch (from $X^{\{t\}}, Y^{\{t\}}$)

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

- Backpropagation to compute gradients w.r.t. $J^{\{t\}}(X^{\{t\}}, Y^{\{t\}})$

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

↳ "1 epoch" of training → 1 pass through training set

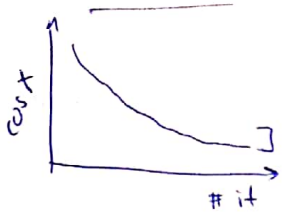
1 step of GD using $X^{\{t\}}, Y^{\{t\}}$ (as if $m = 1000$)

Batch-GD: single pass through training set allows to only take 1 GD step

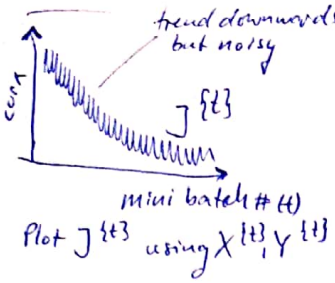
Mini-batch-GD: —" — allows to take 5000 GD steps (in this example)

Understanding mini-batch gradient descent

Batch GD



MBGD



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

oscillations from plot because $X^{\{1\}}, Y^{\{1\}}$ has easy cost value but $X^{\{2\}}, Y^{\{2\}}$ might have higher cost than 1st J

Parameter to chose

→ Choosing your **mini-batch size**!

extreme 1 – If mini-batch size = m: Batch gradient descent (BGD) $X^{\{1\}}, Y^{\{1\}} = (X, Y)$ one mini-batch gives entire set

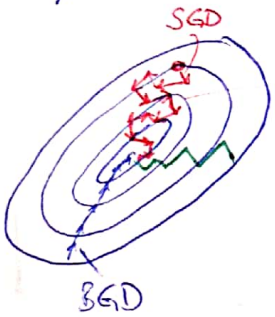
extreme 2 – If mini-batch size = 1: Stochastic Gradient Descent (SGD) : Every example is its own mini-batch

$(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)}) \dots (x^{(2)}, y^{(2)}) \dots$

↳ looking at one training example at a time

In practice mini-batch size will be in between 1 and m

too small — too large



– will never really converge

SGD → lose speed up from vectorization

In practice In-between (mini-batch size not too big/small) → Fastest learning
- can make use of vectorization
- make progress without needing to wait to process the entire training set

BGD (mini-batch size = m) → too long per iteration

Guidelines:

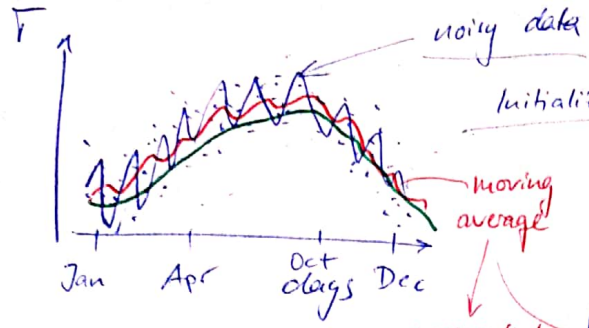
- If small training set: use BGD ($m < 2000$)
- typical mini-batch size: 64, 128, 256, 512 (powers of 2: 2^6 2^7 2^8 2^9 more common)
- Make sure mini-batch fits in CPU/GPU memory $X^{\{t\}}, Y^{\{t\}}$

→ mini-batch size is also a hyperparameter

Exponentially weighted averages

ex. temperature in London

$\theta_1 = 4°C$
$\theta_2 = 9°C$
$\theta_3 = 10°C$
$\vdots$
$\theta_{180} = 15°C$
$\vdots$
$\theta_{365} = 2°C$



Initialize $V_0 = 0$

$V_1 = 0.9 \cdot V_0 + 0.1 \cdot \theta_1$
$V_2 = 0.9 \cdot V_1 + 0.1 \cdot \theta_2$
$V_3 = 0.9 V_2 + 0.1 \cdot \theta_3$
$\vdots$

$\boxed{V_t = 0.9 \cdot V_{t-1} + 0.1 \cdot \theta_t}$

exponentially weighted averages

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

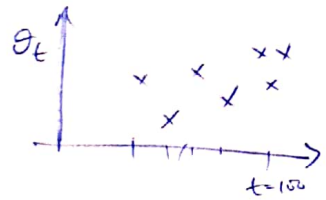
$\beta = 0.9$, V_t as ~~averaging~~ averaging over $\approx \frac{1}{1-\beta}$ days' temperature

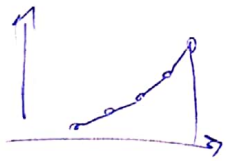
$\frac{1}{1-0.9} = 10$ $\beta = 0.9$: average over 10 days

$\frac{1}{1-0.98} = 50$ $\beta = 0.98$: average over 50 days → plot is smoother with higher β but curve is actually shifted (latency)

$\beta = 0.5$ average over 2 days --- → much more noisy but adapts quicker to temperature changes (less latency)

$\boxed{\rightarrow \beta \text{ is also hyperparameter}}$





$0.9^{10} \approx 0.35 \approx \frac{1}{e}$ $\quad (1-\varepsilon)^{1/\varepsilon} = \frac{1}{e}$

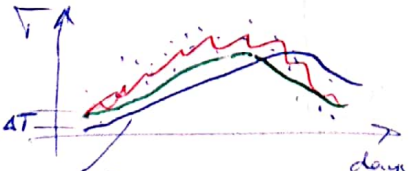
β

$0.98^{50} \approx \frac{1}{e}$

Implementation

$V_\theta = 0$
$V_\theta := \beta \cdot v + (1-\beta)\theta_1$
$V_\theta := \beta \cdot v + (1-\beta)\theta_2$
$\vdots$
repeat {
get next θ_t
$V_\theta := \beta V_\theta + (1-\beta)\theta_t$
}

Bias correction



$V_t = \beta V_{t-1} + (1-\beta)\theta_t$
↳ Starts with lower value at the beginning!

↳ do this!

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

↳ get better estimation for early values

Gradient Descent with momentum → compute exponentially weighted averages of the gradients and use that to update the weights

GD with momentum

oscillations slow down GD and prevent us from taking a large learning rate

↕ want slower learning on vertical axis
↔ want faster learning on horizontal axis

GD with momentum will take smaller steps in vertical direction and converge faster → damping out oscillations

with momentum:

On iteration t: Initialize $V_{dw}=0$, $V_{db}=0$

Compute dW, db on current mini-batch (works also for BGD)

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

(moving average of gradient / derivative)

β: like friction; V_{db}: like a velocity; acceleration

Analogy: ball rolling down a hill

update parameters:

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

→ smoothing out steps of gradient descent

PYTHON: np.zeros_like()

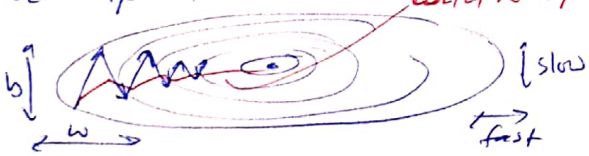
Implementation:

Hyperparameters: α, β — most common 0.9 = averaging over last 10 gradients

In practice: don't really see bias correction when implementing GD with momentum

RMS prop (Root mean square prop)

example from above:



On iteration t:

Compute dW, db on current mini-batch

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) \cdot dW^2$$
$$S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$$

element-wise

→ keeping exponentially weighted average of squares of derivatives

update parameters

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

→ derivatives are larger in "vertical" than in "horizontal" direction

divide by small number

helps damping out the oscillations

divide by large number

add ε to denominator $\approx 10^{-8}$ → for numerical stability

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

(Adaptive moment estimation) DL spec // Course II // week 2

ADAM optimization algorithm → momentum + RMSprop

↳ works well as generalized opt. algos.

β_1 & β_2 ⇒ hyperparams

initialize:

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

On iteration t:

compute dw, db using current mini-batch (MBGD)

momentum-like update with β_1

momentum exponentially averaged

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

RMSprop → $S_{dw} = \beta_2 \cdot S_{dw} + (1-\beta_2) dw^2$, $S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$ — RMSprop-like update with β_2

Implementation:

after bias correction

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

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

Perform update:

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

Hyperparameters choice:

α: needs to be tuned ← try range

β_1: 0,9 (dw)

β_2: 0,999 (dw^2) db^2

ε: 10^{-8}

Quiz

a [3] (7) {2}

$\theta_1 = 10°C$

$\theta_2 = 10°C$

$V_0 = 0$ $\beta = 0,5$

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

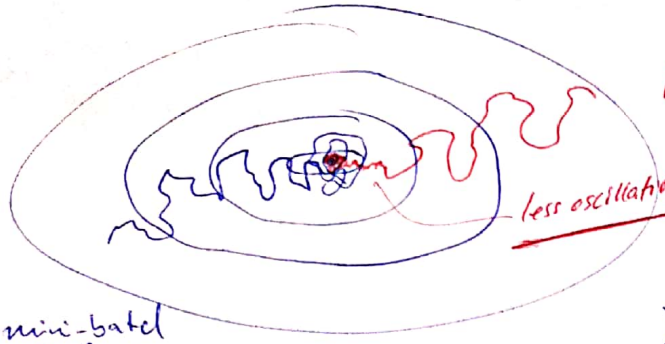
$V_2 = 0,5 \cdot 10°C + (0,5) \cdot 10°C = 10°C$

$5 + 5$

$V_2^{corr} = \frac{10°C}{1-0,5^2} = \frac{0,5 \cdot 10°C + 10°C}{1-0,5^2}$

Learning rate decay

→ way to speed up learning algorithm, is to slowly reduce α over time

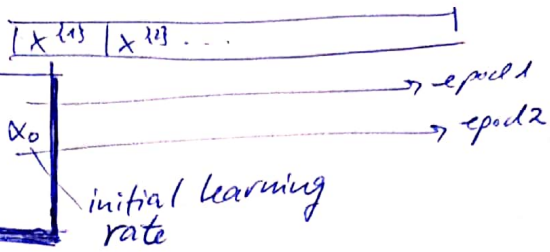


If α is slowly reduced

→ less oscillation more close to minimum

1 epoch = 1 pass through data

mini-batch with fixed α



$$\alpha = \frac{1}{1 + \text{decay\_rate} * \text{epoch\_num}} \cdot \alpha_0$$

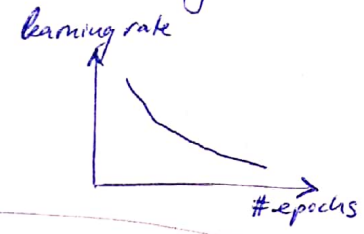
α_0 initial learning rate

Hyperparameter !

Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04
⋮	⋮

$\alpha_0 = 0.2$

decay_rate = 1



exponential decay

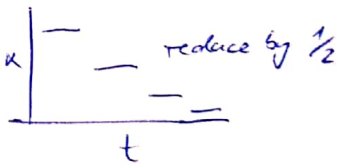
$$\alpha = 0.95^{\text{epoch\_num}} \cdot \alpha_0$$

other

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

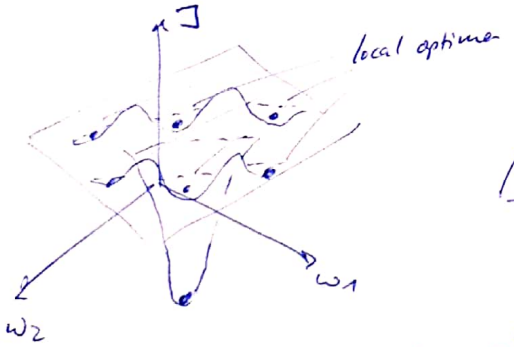
or $\frac{k}{\sqrt{t}} \cdot \alpha_0$

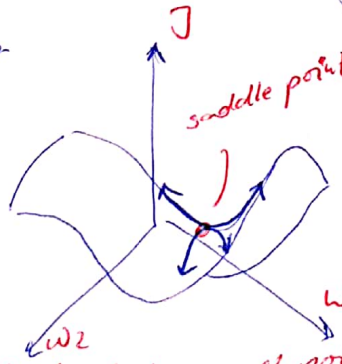
or



or even manual decay

The problem of local optima





In NN training most points of zero-gradients are not local optima, instead are saddle points!

If gradient = 0 in each direction can either be convex or concave

In 20000 dimensional space more likely to be in saddle point than in local optimum

Plateau can really slow down learning

region where gradient too 0 for a long time

derivative

- Unlikely to get stuck in local optima as long as NN is big enough, J defined over high dimensional space → $J \in \mathbb{R}^{\text{large}}$
- Plateau can make learning slow → Adam, RMSprop, momentum

DLspecII Course2 week2

Assignment Optimization Methods

Gradient Descent

→ Difference between batch gradient descent, mini-batch GD and stochastic GD is the number of examples you use to perform one update step

→ You have to tune a learning rate hyperparameter α

→ A well-tuned mini-batch rate usually outperforms GD or SGD, especially when training set is large

Mini-Batch Gradient Descent

→ Shuffling and partitioning are the two steps required to build mini-batches

→ Powers of 2 are commonly chosen → 16, 32, 64, 128...

Momentum

→ takes past gradients into account to smooth out the steps of GD. It can be applied with BGD, MBGD or SGD

→ You have to tune a momentum hyperparameter β and α

Adam