Week 2: Optimization Algorithms

Mini Batch Gradient Descent

Here you divide your dataset into batches so gradient descent occurs faster. Lets say you have $m = 5{,}000{,}000$ (5 million) ~~training examples~~

You can make 5000 minibatches of 1000 each

$$X = [\underbrace{x^{(1)}\ x^{(2)}\ x^{(3)} \dots x^{(1000)}}_{X^{\{1\}}}\ \underbrace{x^{(1001)} \dots x^{(2000)}}_{X^{\{2\}}} \dots \underbrace{\dots x^{(m)}}_{X^{\{5000\}}}]$$

(n_x, m) $\quad$ $X^{\{1\}}$: $(n_x, 1000)$ $\quad$ $X^{\{2\}}$: $(n_x, 1000)$

$X^{\{5000\}}$ ↑ for $m = 5{,}000{,}000$ & batchsize is 1000

$$Y = [\underbrace{y^{(1)}\ y^{(2)}\ y^{(3)} \dots y^{(1000)}}_{Y^{\{1\}}}\ \underbrace{y^{(1001)} \dots y^{(2000)}}_{Y^{\{2\}}} \dots \underbrace{\dots y^{(m)}}_{X^{\{5000\}}}]$$

$(1, m)$ $\quad$ $Y^{\{1\}}$: $(1, 1000)$ $\quad$ $Y^{\{2\}}$: $(1, 1000)$

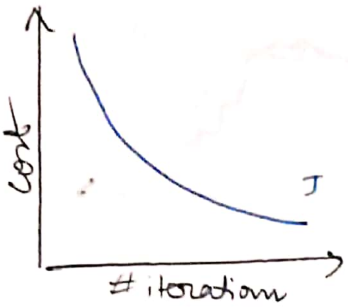
```
repeat {
    for t = 1, ..., 5000 {        <- looping for all 5000 minibatches containing 1000 examples each
        Forward prop on $X^{\{t\}}$:
            $Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$
            $A^{[1]} = g^{[1]}(Z^{[1]})$
            :
            $A^{[L]} = g^{[L]}(Z^{[L]})$        for $X^{\{t\}}, Y^{\{t\}}$
        compute cost $J^{\{t\}} = \frac{1}{1000}\sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000}\sum_{l}\|w^{[l]}\|_F^2$
        backprop to compute gradients wrt $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)
        $W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}$, $b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$
    }
}
```

1 epoch → single pass through training set

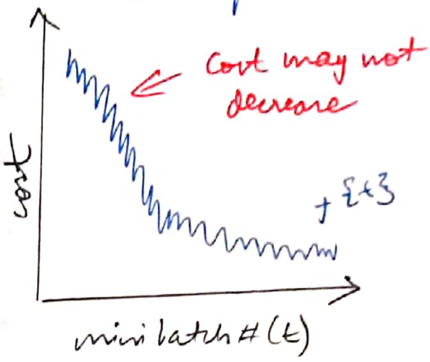
Understanding Mini-batch gradient descent

Training with mini batch gradient descent:

Batch Gradient Descent



Mini-batch gradient descent



Choosing your mini-batch size:

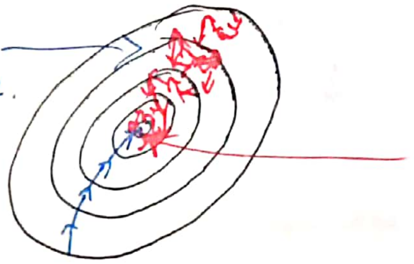
Lets look at 2 extremes,

Takes too long for an iteration

→ If mini-batch size = m, Batch Gradient descent

→ If mini-batch size = 1, stochastic GD ← every example is its own mini-batch

↖ Doesn't use the speed of vectorisation



chaotic, it can increase or decrease. A smaller α will help it be less chaotic

stochastic GD never converges, but keeps oscillating in a particular area ☹

∴ ~~we~~ we need something in between

← Vectorisation

← Makes progress without passing through entire training set

How to choose?

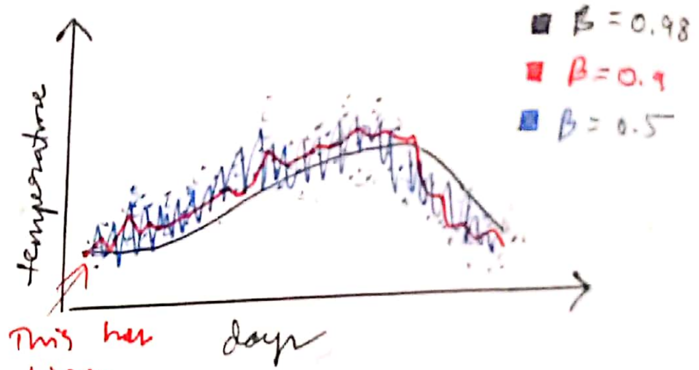
1. If small training set ($m \leq 2000$)
   Use Batch GD

2. If large,
   - Typical mini-batch sizes are powers of 2
     64, 128, 256, 512 1024
     normally used
   - Make sure mini-batch fits in CPU/GPU memory
     $(X^{\{t\}}, Y^{\{t\}})$

Exponentially weighted averages

Consider temperature in London,

day 1. $\theta_1 = 40°F$
day 2. $\theta_2 = 49°F$
day 3. $\theta_3 = 45°F$
⋮
day 180. $\theta_{180} = 60°F$
day 181. $\theta_{181} = 56°F$
⋮



This has bias correction

Exponentially weighted averages:

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

V_t as approximated average $\approx \frac{1}{1-\beta}$ days' temp.

In the example $\beta = 0.9$

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

Therefore β determined how smooth the graph is. Notice $\beta = 0.98$ is smoother, i.e $\frac{1}{1-0.98} = 50$

∴ It considers last 50 days data so a single day's data changes it by a little (0.02)

$\beta = 0.9$ ≈ 10 days' temperature
$\beta = 0.98$ ≈ 50 days' temperature
$\beta = 0.5$ ≈ 2 days' temperature

understanding exponentially weighted averages:

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

Take

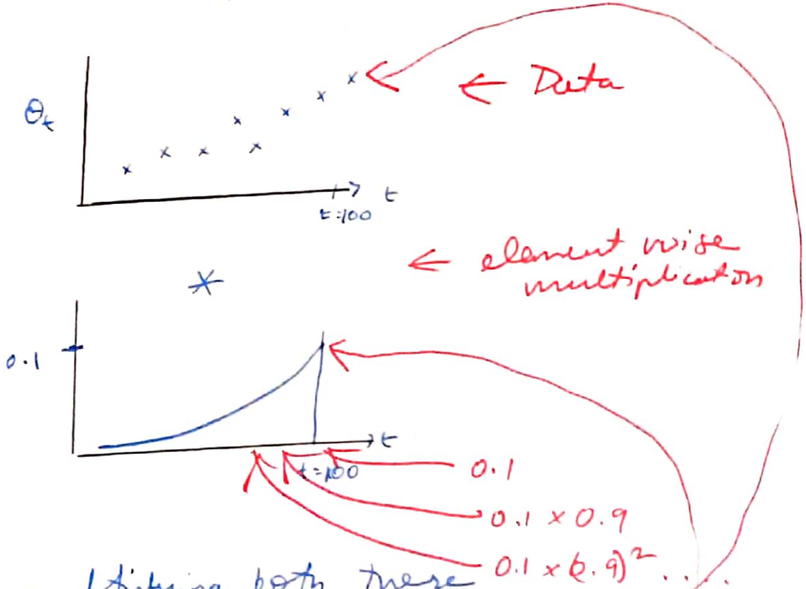
$$V_{100} = 0.9V_{99} + 0.1\,\theta_{100}$$

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

$$V_{98} = 0.9V_{97} + 0.1\,\theta_{98}$$

$$V_{100} = 0.1\,\theta_{100} + 0.9\,(0.1\,\theta_{99} + 0.9\,(0.1\,\theta_{98} + 0.9V_{97}\ldots$$

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



Here by multiplying both these element wise, we get V ie V_{100} = ● × ●

How do we know how many days?

$\beta^x \approx 0.35$ or $\frac{1}{e}$

Exa

$0.9^{10} \approx \frac{1}{e}$ ∴ 10 days

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

Algorithm:

```
$V_\theta = 0$
repeat {
    Get next $\theta_t$
    $V_\theta = \beta V_\theta + (1-\beta)\theta_t$
}
```

↑ new value of V

↑ old value of V in memory from previous iteration

Why better than Gradient Descent?

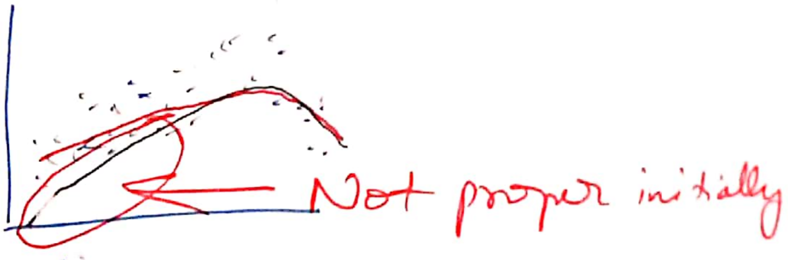
- → Just 1 line of code
- → Fast and takes up very little space in memory

Bias correction in exponentially weighted averages

Because $V_0 = 0$ and $V_1 = 0.98\,\underset{\overset{\uparrow}{0}}{V_0} + 0.02\,\theta_1$

$= 0.02\,\theta_1$

∴ we can see that the initial data will not be proper



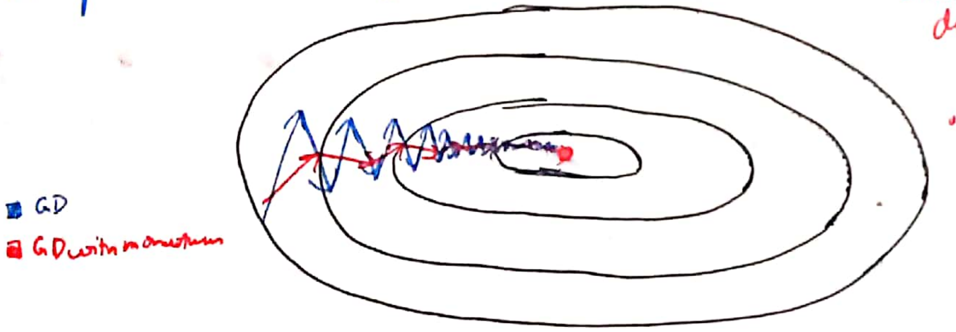
In order to ~~do~~ fix this we can do $\frac{V_t}{1-\beta^t}$

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

$$\frac{V_2}{0.0396} = \frac{0.\overset{0.98\times0.02 \text{ from } 0.98V_1}{\overset{\downarrow}{0}}196\,\theta_1 + 0.02\,\theta_2}{0.0396}$$

Gradient Descent with Momentum

It is gradient descent with exponentially weighted averages.



↕ slow
- we need to slow down learning in vertical direction
- ↔ fast
we need to faster learning in horizontal direction

Usually gradient descent takes zigzag motion steps ~~[illegible]~~ in the direction of the minima. However learning will be faster if it goes straight. This is done by using gradient descent with momentum.

Algorithm:

On iteration t:

Compute dw, db on the current mini-batch

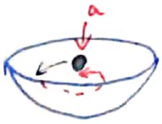
$V_{dw} = \beta V_{dw} + (1-\beta) dw$

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

$W = W - \alpha V_{dw}$, $b = b - \alpha V_{db}$.

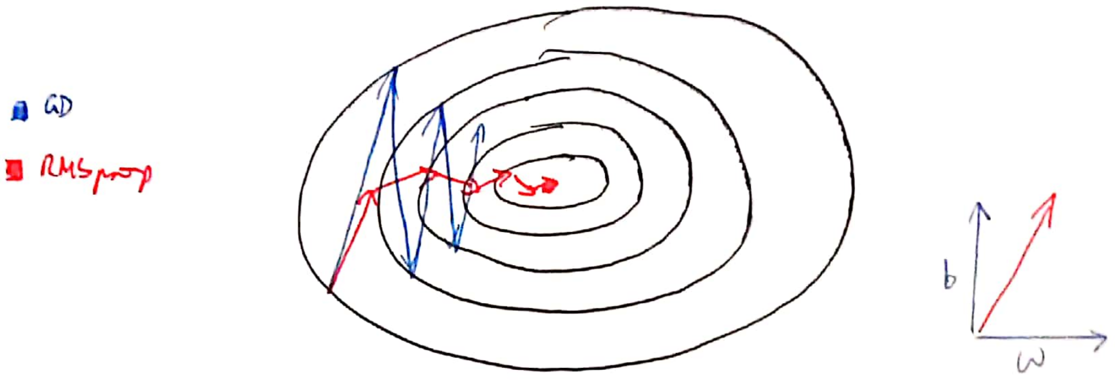
- Bias correction $\left(\frac{V_{dw}}{1-\beta^t}\right)$ isn't required since the initial graph doesn't matter since it train fast
- $\beta = 0.9$ is a standard hyperparameter value
- This algo is similar to providing acceleration (momentum) to a ball spinning in a ~~bowle~~ bowl so it gets pushed closer towards the centre. (Intuition)

$$\underbrace{\overset{}{V_{dw}} = \beta}_{\text{Friction}} \underbrace{V_{dw}}_{\text{Velocity}} + \underbrace{(1-\beta) dw}_{\text{acceleration}}$$



RMSprop

This is another algo that tries to fix the vertical motion of gradient descent



On iteration t:

Compute dw, db on current mini-batch

$S_{dw} = \beta_2 S_{dw} + (1-\beta) dw^2$

$S_{db} = \beta_2 S_{db} + (1-\beta) db^2$

$w := w - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}$, $b := b - \alpha \frac{db}{\sqrt{S_{dB}} + \varepsilon}$

- We set $\varepsilon = 10^{-8}$. This is used so we don't divide by zero.

- Intuition:
  - b is large ↑ and w is small →

  $\therefore$ dw^2 is small, db^2 is large

  $\therefore$ when we divide $w := w - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}$,

  since S_{dw} is small, it won't affect dw

  $\therefore$ when we divide $b := b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$

  since S_{db} is large, it reduces db

  $\therefore$ The vertical movement is damped

Adam (Adaptive moment estimation) algorithm

- This is GD with momentum + RMS prop
- Algorithm,

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

On iteration t:

Compute dw, db using current mini-batch

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

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

Bias correction

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

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

update

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

- Hyperparameter
  - → α - need to choose
  - → β_1 (for GDwithM) : 0.9
  - → β_2 (for RMSprop) : 0.999
  - → ε : 10^{-8}

Learning Rate Decay

Gradient Descent never converges but oscillates around the minima because the step size is large. One way to tackle this while keep training speed high is to decay (reduce) the learning rate α, so that as it approaches the minima, the steps grow smaller and it oscillates around a smaller region

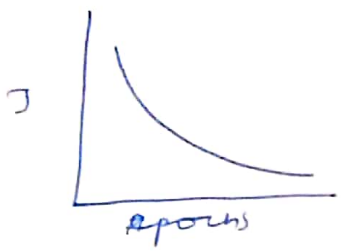
- An epoch is one traversal through the entire training set (one iteration of GD).
- Learing rate decay,

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

← epoch number

For $\alpha_0 = 0.2$ and decay-rate $= 1$,

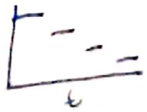
Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04
⋮	



- other learning rate decay formulas

→ $\alpha = 0.95^{\text{epoch-num}} \alpha_0$ — exponential decay

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

→ α  - discrete staircase

- Manual decay - if it is taking days to decrease, you can manually reduce α

The problem of local optima

- Unlikely to get stuck in a local optima:
  Although when we visualise in 2D, we may come across local optimas, in higher dimensions the chances of them occuring is very less ;

- Plateaus are a problem - they slow down learning:
  Plateaus can be encountered ~~that~~.
  They are areas where there derivative is close to 0. The algorithm travels along the plateau ~~and~~ for a long distance before travelling down.