* OPTIMISING ALGORITHMS :

* Mini - Batch Gradient Descent :

Vectorizations allows you to efficiently compute m examples.
However, if m is very large ($m = 5{,}000{,}000$), then gradient descent becomes slow as it works by iterating over each training example.

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

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

5000 mini-batches :

mini batch t : $X^{\{t\}}$ $Y^{\{t\}}$ } Notations

$x^{(i)}$ $z^{[l]}$

- use vectorization to process each mini-batch over for-loop.

for $t = 1 \longrightarrow 5000$
  forward prop on $X^{\{t\}}$
  Vectorized Implementation :
  $Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$
  $A^{[1]} = g^{[1]}(Z^{[1]})$
  ⋮
  $A^{[l]} = g^{[l]}(Z^{[l]})$

Compute cost :

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

for $x^{\{t\}}, y^{\{t\}}$

Everything is same, except
$x \rightarrow x^{\{t\}}$
$y \rightarrow y^{\{t\}}$

Backprop to compute $J^{\{t\}}$

$$w^{[l]} = w^{[l]} - \alpha \, dw^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha \, db^{[l]}$$

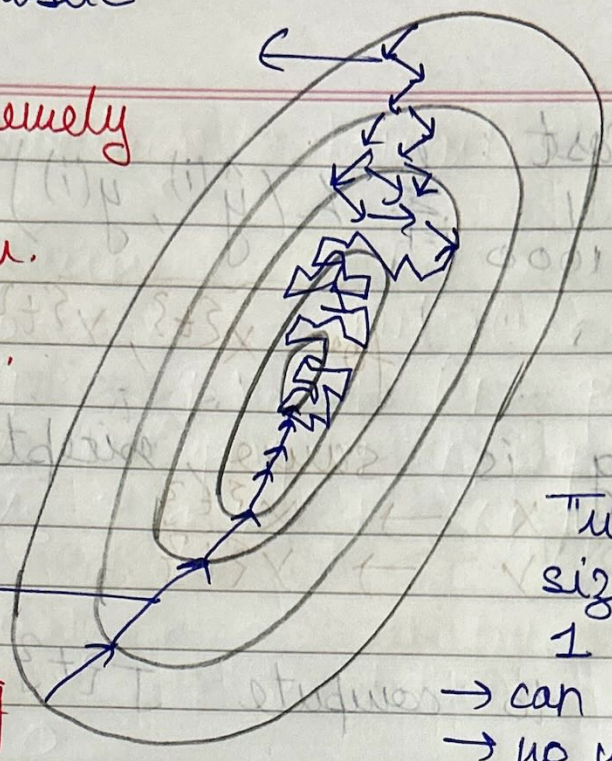
entire process is called 1 epoch (1 single pass)

(Mini Batch > Batch) Gradient Descent.

* Choosing mini-batch size :

If mini-batch-size = m : Batch GD.
If mini-batch-size = 1 : Stochastic GD
→ every example is its own $x^{\{t\}}, y^{\{t\}}$

Stochastic GD
→ extremely noisy
→ no adv. from vec.



Batch GD
→ too long

Thus, mini-batch size ranges from 1 to m
→ can use vectorization
→ no need to wait for the entire set.

Small Training set : Batch GD.
($m \leq 2000$)

Typical mini-batch size : 64, 128, 256, 512
→ better if size is a power of 2.

- Mini-batch size is another hyperparameter

* Exponentially Weighted Averages :

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

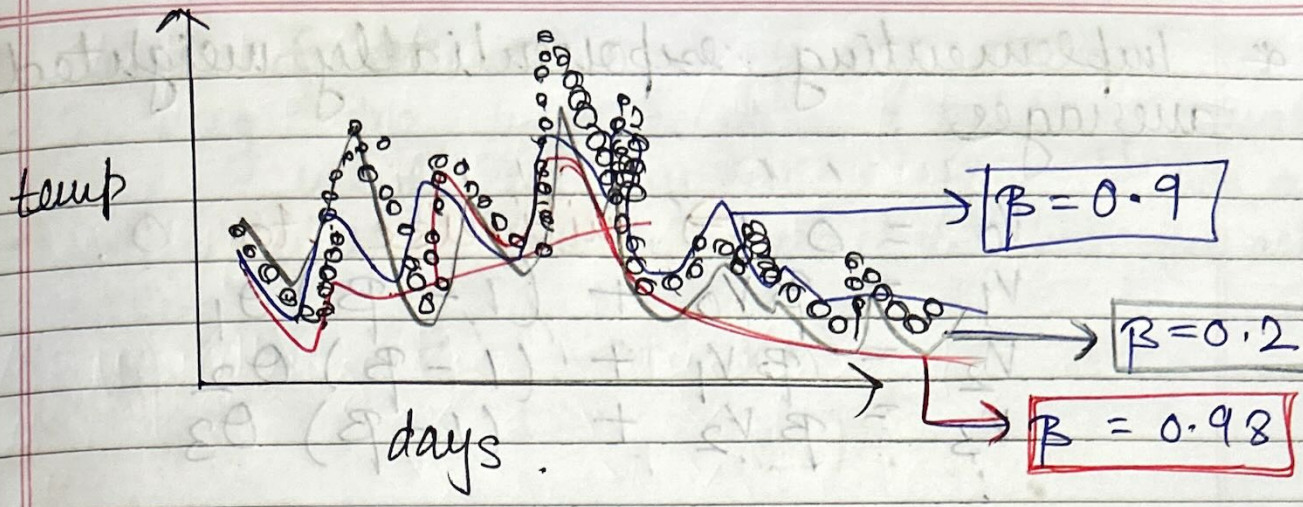
→ Exponentially weighted func.

V_t approximately averages over : $\frac{1}{1-\beta}$ days temp.

$\beta = 0.9 \approx 10$ days
$\beta = 0.98 \approx 50$ days
$\beta = 0.5 \approx 2$ days

$\beta \uparrow$, $\frac{1}{1-\beta} \uparrow$

↑ days are averaged over.



as V_t's is averaged over how many days depends on β, as $\beta \downarrow$ there is $\uparrow$ accurate curve (kinda overfit).

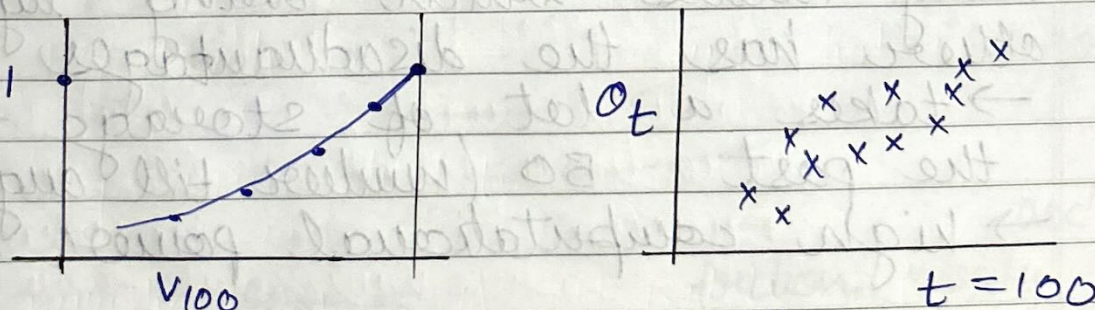
- key component to many optimisation algorithms

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

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

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

exponential decay



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

Similarly $(0.98)^{50} \approx 0.35 \approx 1/e$ $\quad 1-\varepsilon = 0.98$

$(1-\varepsilon)^{1/\varepsilon} = 1/e$

* Implementing exponentially weighted averages :

$V_0 = 0 \Rightarrow$ initialise to 0.

$V_1 = \beta V_0 + (1-\beta)\theta_1$

$V_2 = \beta V_1 + (1-\beta)\theta_2$

$V_3 = \beta V_2 + (1-\beta)\theta_3$

.
.
.

$V_0 = 0$

~~for~~ repeat {

get next θ_t

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

}

Advantages

→ storage & memory required is less (only V_0)

→ owing to this, efficiency is good as there's only one line of code.

- It helps to calculate the average over many values which doing by some other has the disadvantages :

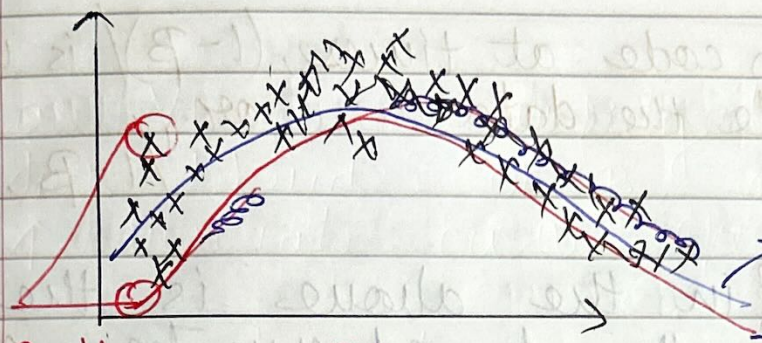
→ takes a lot of storage to save the past 30 (number till avg) values

→ high computational power.

* Bias Correction:

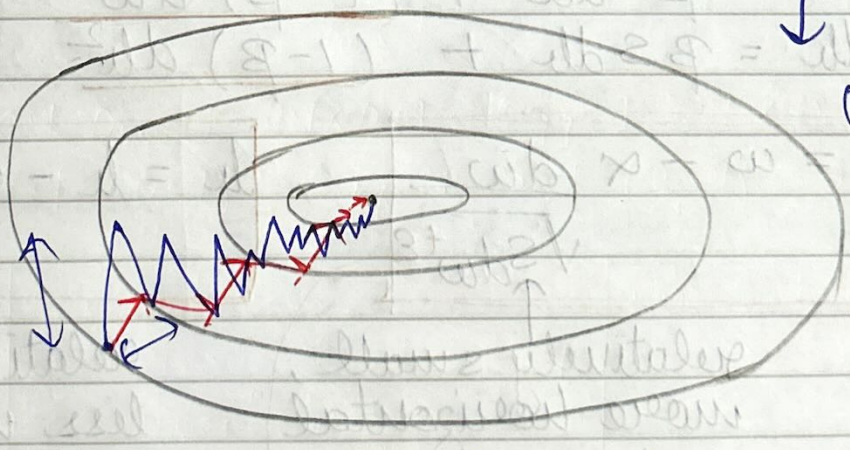
- During the initial phase (t being small), the weighted average func^n has a high bias (much smaller than the actual value).

thus $V_t \rightarrow \frac{V_t}{1-\beta^t}$



value of ⊗ & its value by func^n, thus bias correction req.

* Gradient Descent ~~Example~~ Momentum:



↕ slowing learning (vertical) ⟷ faster running (horizontal)

Momentum: on iteration t

on current mini-batch

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

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

friction, velocity, acc^n

Hyperparameters : α, β.

Most common, $\beta = 0.9$.

In code :

$V_{dw} = \beta V_{dw} + (1-\beta) dW$ ⇒ Preferred latter to avoid rescaling

↳ $V_{dw} = \beta V_{dw} + dW$

used in code at times, $(1-\beta)$ is used to scale the data or $1/1-\beta$.

- difference in the above is the training of the hyperparameter α

* RMS Prop :

On iteration t :

Compute dW, db on current minibatch:

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

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

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

S_{dw}: relatively small, more horizontal variance

S_{db}: relatively large, less vertical variance

ε: to ensure numerical stability

- Advantage: greater α (learning rate) with less divergence

- Has the effect of damping out oscillations in gradient descent as momentum in GD.

* Adam Optimisation Algorithm:

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

On iteration t:
Compute dw, db, using t^{th} mini-batch:

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

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

$$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t)$$
$$V_{db}^{corrected} = V_{db} / (1-\beta_1 t)$$
$$S_{dw}^{corrected} = S_{dw} / (1-\beta_2 t)$$
$$S_{db}^{corrected} = S_{db} / (1-\beta_2 t)$$

$$w = w - \alpha \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon}$$

$$b = b - \alpha \frac{V_{db}^{corrected}}{\sqrt{S_{db}^{corrected}} + \varepsilon}$$

Hyperparameters Choice :

α ⇒ needs tuning

β_1 ⇒ 0.9 (dw) → moment 1

β_2 : 0.999 (dw^2) → moment 2.

ε ⇒ 10^{-8}.

ADAM: Adaptive Moment Estimation

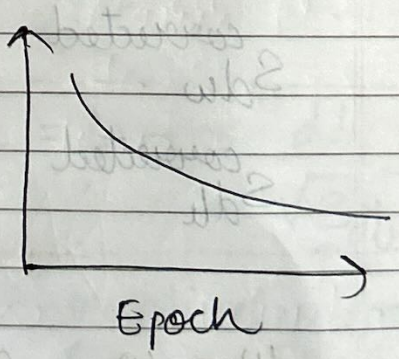
* Learning Rate Decay :

- SLOWLY reducing α.

1 epoch = 1 pass through data.

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

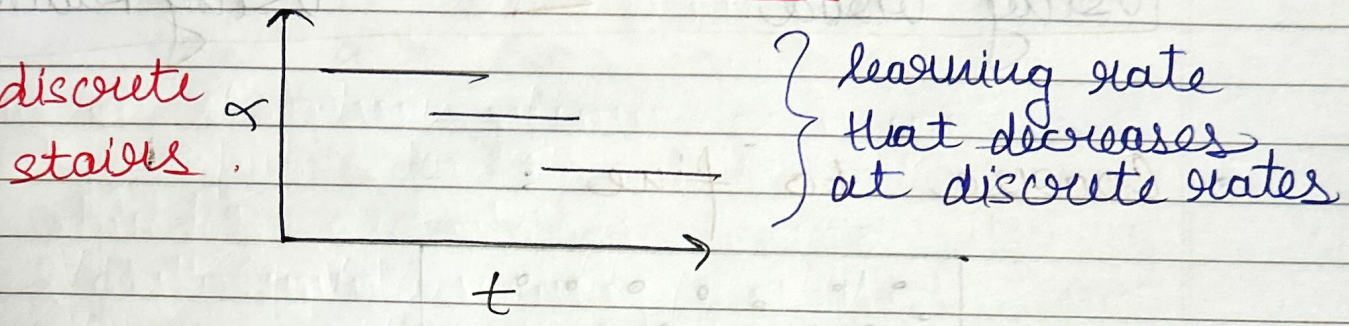
Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
⋮	⋮



Other learning rate decay methods:

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

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



- Manual Decay works only on small sets.
- One of the later steps to work on.

* Hyperparameters (Week 3)