$x^{(i)} \rightarrow$ ith training example
$z^{[l]} \rightarrow$ value of z for layer l
$X^{\{t\}}, Y^{\{t\}} \rightarrow$ mini-batch t

Week 2 : Optimization Algorithms

Mini-batch Gradient descent

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

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

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

split training sets into smaller parts → (mini-batches)

minibatches each of 1000 examples

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

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

Minibatch t : $X^{\{t\}}, Y^{\{t\}}$

↓ check assignment code for creating mini-batch

Batch vs minibatch gradient descent

Batch ↓ all (X, Y) at once are processed

minibatch ↓ $X^{\{t\}}, Y^{\{t\}}$ are processed at once

// S = no. of samples in the mini-batch

Page No.
Date

Mini-batch gradient descent (algorithm)

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

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

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

this vectorized implementation processes 1000 examples rather than whole training set

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

for $X^{\{t\}}, Y^{\{t\}}$

Backprop to compute gradients cont $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

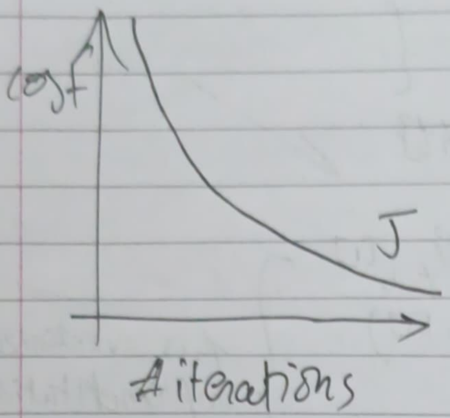
$$W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}, \quad b^{[l]} := b^{[l]} - \alpha\, db^{[l]}$$

}

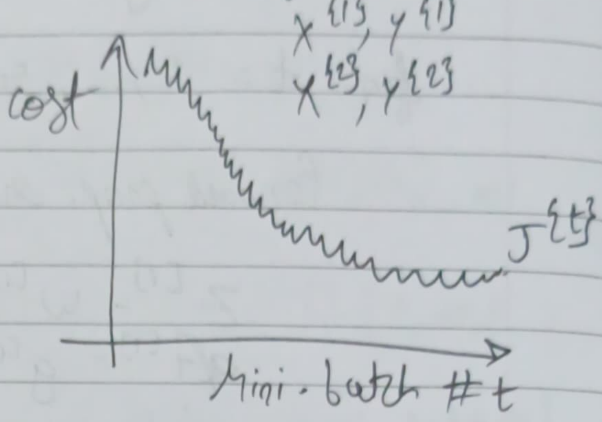
"1 epoch"
↳ single pass through training set

When you have a large training set, a mini-batch algorithm is preffered as it would work faster than the regular algo.

Batch grad. descent



Mini-batch grad. descent



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

The graph in mini-batch gets noisy because we are computing $J^{\{t\}}$ for different different mini-batches i.e. for different $X^{\{t\}}, Y^{\{t\}}$ in each iteration so that is why some $J^{\{t\}}$'s might go up and some might go down, but overall trend should be seen as decreasing only.

Choosing mini-batch size (Hyperparameter)

If mini-batch size = m : Batch grad. descent $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

If mini-batch size = 1 : Stochastic grad. descent
→ every example is its own mini batch.
$(X^{\{1\}}, Y^{\{1\}}) = (x^{(1)}, y^{(1)})$, $(X^{\{2\}}, Y^{\{2\}}) = (x^{(2)}, y^{(2)})$

In practise : mini-batch size is between 1 and m.

Page No.
Date

Stochastic grad descent | In-between (mini-batch size) | Batch grad desent

mini-batch size = 1 → Lose speed up from Vectorization because processing one ex. at a time is ineffecient.

In-between → Fastest learning:
- vectorization
- Make progress w/o training entire set

mini-batch size = m → Too long to process per iteration.

If small training set: Use batch grad. descent
↳ ($m \leq 2000$)

Typical mini-batch sizes:

64, 128, 256, 512
2^6 2^7 2^8 2^9

Also make sure mini-batch size fits in CPU/GPU memory

Other optimization Algos: Exponentially weighted averages
→ Faster than grad. descent

ex: $\theta_1 = 4°C$
$\theta_2 = 9°C$
$\theta_3 = \vdots$
$\theta_{180} = 15°C$
$\vdots$

$V_0 = 0$
$V_1 = 0.9 V_0 + 0.1 \theta_1$
$V_2 = 0.9 V_1 + 0.1 \theta_2$
$V_3 = 0.9 V_2 + 0.1 \theta_3$

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

$v_t = \beta v_{t-1} + (1-\beta)\theta_t$, $\beta = 0.9$

v_t can be inferred as approximately averaging over $\approx \frac{1}{1-\beta}$ days' temperature

$\beta = 0.9$: $\approx$ 10 days' average temp.

$\beta = 0.98$: $\approx$ 50 days' average

↳ when plotted, the curve gets more ~~β =~~ smoother and it shifts more to the right.

$\beta = 0.5$: $\approx$ 2days → more noisy curve

$v_t = \beta v_{t-1} + (1-\beta)\theta_t$, $\beta = 0.9$

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

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

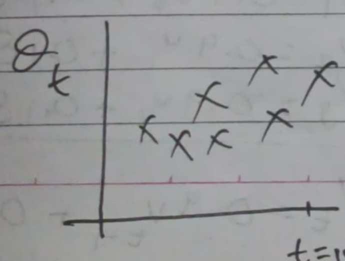
$v_{98} = 0.9\, v_{97} + 0.1\, \theta_{98}$

$\vdots$

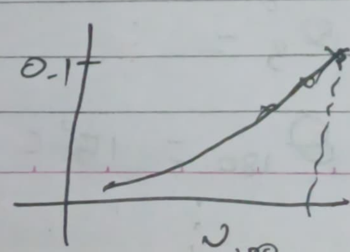
→ $v_{100} = 0.1\, \theta_{100} + 0.9\, \cancel{v_{99}}(0.1\, \theta_{99} + 0.9\, \cancel{v_{98}})$

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

$= 0.1\, \theta_{100} + 0.1 \times 0.9 \times \theta_{99} + 0.1\,(0.9)^2\, \theta_{98} + \dots$



*



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

Implementing exponentially weighted averages

$v_0 = 0$
$v_1 = \beta v_0 + (1-\beta)\theta_1$
$v_2 = \beta v_1 + (1-\beta)\theta_2$
$\vdots$

$v_\theta = 0$
$v_\theta := \beta v + (1-\beta)\theta_1$
$v_\theta := \beta v + (1-\beta)\theta_2$
$\vdots$

$v_\theta = 0$
Repeat {
 Get next θ_t
 $v_\theta := \beta v_\theta + (1-\beta)\theta_t$
}

* Very effecient as it takes only one line of code and less memory (as we are overwritting the same variable), still not the best way ~~to find~~ to compute average.

Bias correction in exponentially weighted averages

$v_t = \beta v_{t-1} + (1-\beta)\theta_t$, $\beta = 0.98$

$v_0 = 0$
$v_1 = \cancel{0.98 v_0} + 0.02\theta_1$, $\beta = 0.98$

If $\theta_1 = 40°C$, $v_1 = 0.02 \times 40 = 0.8$ which is not a good estimate on first day's temperature.

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

$$= 0.98\,(0.02) \times \theta_1 + 0.02\, \theta_2$$

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

→ Even v_2 will not be a very good estimate of first 2 day's temperature.

To solve this:

→ insted of ~~take~~ considering v_t, take $\frac{v_t}{1-\beta^t}$

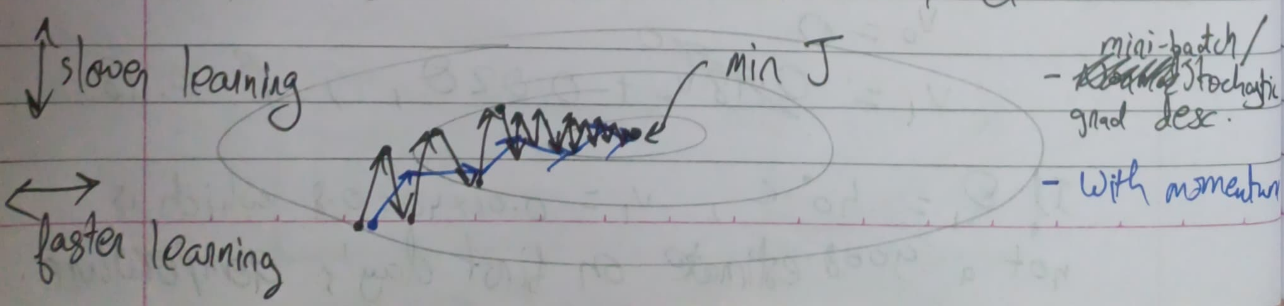
eg: $t = 2$: $1 - \beta^t = 1 - (0.98)^2 = 0.0396$

$$\frac{v_2}{0.0396} = \frac{0.0196\, \theta_1 + 0.02\, \theta_2}{0.0396}$$

so as seen above, this becomes a weighted average of θ_1 & θ_2 $(\because 0.0196 + 0.02 = 0.0396)$ and this removes the bias.

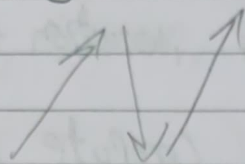
So as t becomes large, ~~$1-\beta^t$ becomes~~ β^t will approach 0, so bias correction will make no difference.

- Gradient descent with Momentum:



v['dw' + str(l)] = np.zeros((parameter['w' + str(l)]).shape)

The up and down oscillations slows down gradient descent and prevents us from using larger learning rates.

If learning rate is large:  it ends up diverging

Momentum: β

On iteration t:

Compute dw, db on current mini-batch

friction (<1)

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

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

velocity   acceleration

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

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

Implementation details:

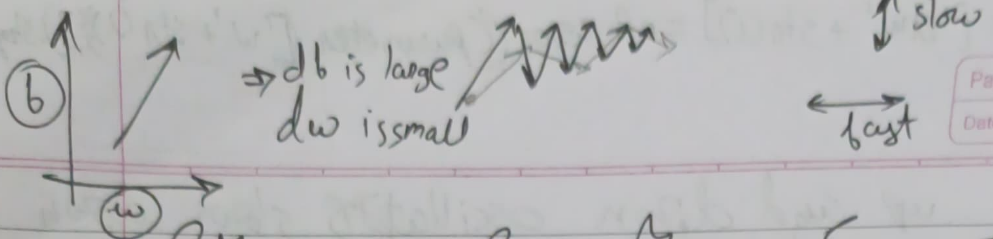
\# $v_{db} = 0$, $v_{dw} = 0$ → shape(dw) = shape(w)

Hyperparameters: α, β

$\beta = 0.9$ (robust value) / common value

i.e. average of temp days $\approx$ 10

→ No need to use bias correction for v_{dw} & v_{db}



RMS prop : Root Mean Square Prop

On iteration t :

Compute dw, db on current mini-batch

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

∵ dw is small #

"dw" ~~this~~ keeps the exponentially weighted average of the squares of the derivatives.

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

∵ db is large #

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

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

Similar effects as seen in momentum as, vertical oscillations damp out and it moves faster horizontally, so we can also increase the learning rate to further make the process and computations faster without any issue.

Page No.
Date

Adam optimization algorithm:

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

On iteration t:

Compute dw, db using current mini-batch

$$\left.\begin{aligned} V_{dw} &= \beta_1 V_{dw} + (1-\beta_1) dw \\ V_{db} &= \beta_1 V_{db} + (1-\beta_1) db \end{aligned}\right\} \text{momentum } \beta_1$$

$$\left.\begin{aligned} S_{dw} &= \beta_2 S_{dw} + (1-\beta_2) dw^2 \\ S_{db} &= \beta_2 S_{db} + (1-\beta_2) db^2 \end{aligned}\right\} \text{RMSprop } \beta_2$$

In adam's optimization, we include correction factors

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

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

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

→ very effective for wide variety of architecting

Hyperparameters choice:

α : needs to be tuned

β_1 : 0.9 $(dw) \rightarrow$ mean of the derivatives

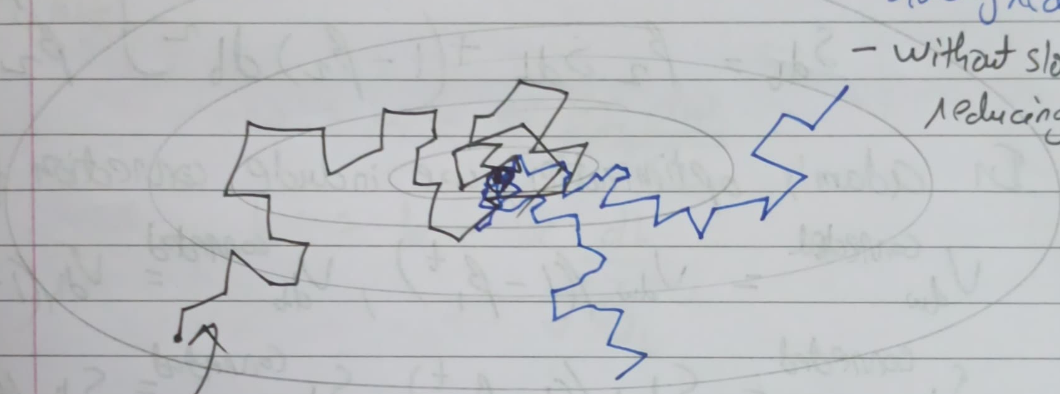
β_2 : 0.999 $(dw^2) \rightarrow$ compute exponentially weighted average of squares

ε : 10^{-8}

Adam: Adaptive moment estimation

Learning rate decay:

- Slowly reduce α
- without slowly reducing α



This does not exactly converge at the minimum hence is wanders around the mininimum J but never really reaches there

By slowly reducing α, we will take smaller and smaller steps and as the min J ~~at~~ comes closer, it will finally converge with smaller steps and not wander around.

Page No.
Date

1 epoch = 1 pass through the data

| $\cancel{x}\,x^{\{1\}}$ | $x^{\{2\}}$ | ... |

→ epoch 1

→ epoch 2 (2nd pass)

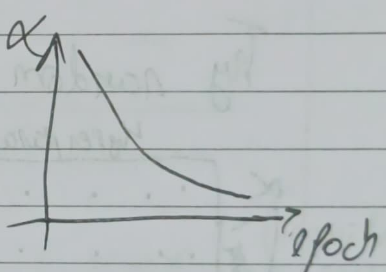
⋮

$$\alpha = \frac{1}{1 + \text{decayrate} * \text{epoch-number}} \alpha_0$$

example:

$\alpha_0 = 0.2$, decay rate $= 0.1$

Epoch	α
1	0.1
2	0.067
3	0.05
4	0.04



other learning rate decay methods:

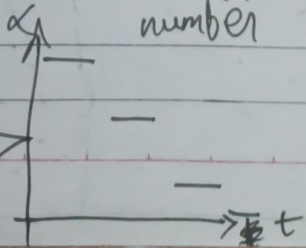
1) $\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0$ → exponentially decay the value of α

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

t → mini-batch number

3) Discrete staircase

Fixed interval scheduling →



or 4) Manual Decay