

This lesson is about optimization algorithms (Alternative to gradient descent) to help our NN to train much faster.

• Mini-batch gradient descent.

In each iteration, to update the parameters by gradient descent, we have to run through the entire training set. This could take a long time if the training set is big, e.g. 5 millions or even 50 millions. We can then first make some progress in gradient descent before we use the entire training, i.e. mini-batch gradient descent:

- split the training set into mini-batches: (e.g. each batch size is 1000)

$$X^{(t)} = [x^{(1)} \dots x^{(1000)}] \quad X^{(t+1)} = [x^{(1001)} \dots x^{(2000)}] \dots \quad X^{(5000)} = [x^{(1000 \times 5000-t+1)} \dots x^{(5000 \times 100)}]$$

similar for Y.

then mini-batch t : $X^{(t)}, Y^{(t)}$

- for $t = 1, \dots, 5000$

Forward prop on $X^{(t)}$: $\tilde{z}^{(1)} = w^{(1)}x^{(1)} + b^{(1)} \dots \tilde{z}^{(L)} = w^{(L)}A^{(L-1)} + b^{(L)}$

$$A^{(0)} = g^{(0)}(\tilde{z}^{(1)}) \quad \dots \quad A^{(L)} = g^{(L)}(\tilde{z}^{(L)})$$

Compute cost $J^{(t)} = \frac{1}{1000} \sum_{i=1}^{1000} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \|w^{(L)}\|_2^2$ for $y^{(i)}$ in $Y^{(t)}$

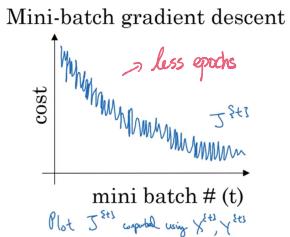
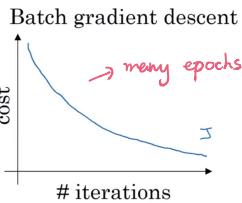
Back prop to compute gradients wrt. $J^{(t)}$

Update $w^{(L)} := w^{(L)} - \text{dow}^{(L)}$, $b^{(L)} = b^{(L)} - \text{doub}^{(L)}$

} also known as "1 epoch"
i.e. a single pass through training set

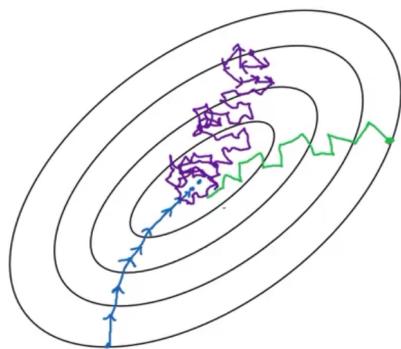
How does mini-batch gradient descent speed up training:

with batch gradient descent, a epoch only allows us to take one gradient descent step in each iteration. So it takes longer to minimize cost.



Andrew Ng

Choosing the batch size:



- if mini-batch size = m (entire training set) : batch gradient descent $(X^{(1)}, Y^{(1)}) = (X, Y)$
 - points to the optimal direction in each gradient descent
 - takes the longest time in each gradient descent
- if mini-batch size = 1 (I example) : stochastic gradient descent $(X^{(1)}, Y^{(1)}) = (X^{(1)}, Y^{(1)}) \dots$
 - points at random direction in each gradient descent but on average converges
 - takes the shortest time in each gradient descent but loses speed-up from vectorization so no speed-up effects
- if mini-batch size is inbetween (e.g. 64, 128, 256, 512) : mini-batch gradient descent
 - points to somewhat correct direction in each gradient descent.
 - takes moderate time in each gradient descent and keeps the advantages of vectorization so to sum up it trains the fastest.

• Gradient Descent with Momentum

To understand the algorithm, we first need to understand exponentially weighted (moving) averages:

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

hyperparameter

↓ previous info ↓ current info

- V_t is approximately average over $\frac{1}{1-\beta}$ days
 - $\beta \uparrow \rightarrow$ average over longer time period \rightarrow smoother
 - $\beta \downarrow \rightarrow$ average over shorter time period \rightarrow more sensitive to outliers.

$$\theta_t = \beta(V_{t-1} + (1-\beta)\theta_{t-1}) + (1-\beta)\theta_t = \dots = (1-\beta)\theta_t + (1-\beta)\beta\theta_{t-1} + (1-\beta)\beta^2\theta_{t-2} + \dots + (1-\beta)\beta^{t-1}\theta_1 + (1-\beta)\beta^t\theta_0$$

* How to implement:

$$V_0 = 0$$

repeat {

Get next θ_t

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

}

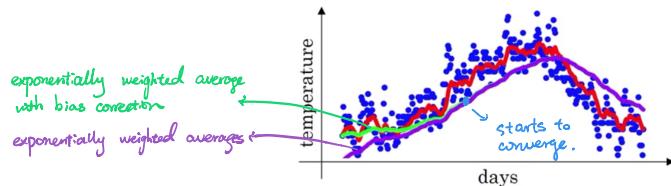
uses very little computer memory

since it only stores one value.

There's one more technical detail called **biased correction** that can make the computation of these averages more accurate.

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

- helps to improve initial estimation. Since $V_0 = 0$, $V_1 = \beta V_0 + (1-\beta) \theta_1 = (1-\beta) \theta_1$, $V_2 = \beta V_1 + (1-\beta) \theta_2 = (1-\beta)^2 \theta_1 + (1-\beta) \theta_2 \dots$
 $1-\beta^t \approx 1$, which means the bias correction term is close to 1 and $V_{t\text{-correct}} \approx V_t$.



A brief description of gradient descent with momentum is to compute an exponentially weighted average of gradients, and then use that gradients to update weights instead.

initialise Vdw and Vdb to vectors of zeros with the same dimension as w and b

On iteration t : compute dW , db on current mini-batch/batch

$$\begin{aligned} Vdw &= \beta Vdw + (1-\beta) dW & \Rightarrow w &:= w - \alpha Vdw \\ Vdb &= \beta Vdb + (1-\beta) db & b &:= b - \alpha Vdb \end{aligned}$$

exp. weighted average
of gradients

2 hyperparameters:
 α needs to be tuned
 $\beta = 0.9$ (usually works very well)
 \approx average of 10 gradients)

How does momentum speed up gradient descent:

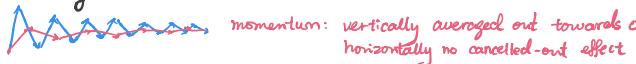
For example, the cost function we want to find the minimum of has a contour like such:



Naturally we want the gradient descent to move faster horizontally and slower vertically.

↑ slower learning
 ← faster learning.

Using the exponentially weighted average of gradients allows us to smooth out the gradients, i.e. take the average of gradients. Since the gradients oscillate a lot like this: $\nearrow \searrow \nearrow \searrow \dots$, if we take the average of these then the values in vertical directions will cancel out each other and tend to be averaged out towards 0. Whereas, on the horizontal direction, all the gradients point to the same direction so the average of horizontal values will still be quite big. So gradient descent with momentum will eventually take larger steps horizontally but oscillate less vertically.

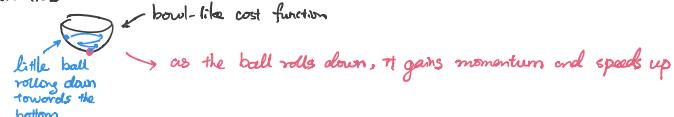

 momentum: vertically averaged out towards 0 \Rightarrow less gradient descent steps \Rightarrow straight to the point horizontally no cancelled-out effect

The name momentum comes from the intuition that:

$$\begin{aligned} Vdw &= \beta Vdw + (1-\beta) dW \\ Vdb &= \beta Vdb + (1-\beta) db \end{aligned}$$

velocity friction

acceleration

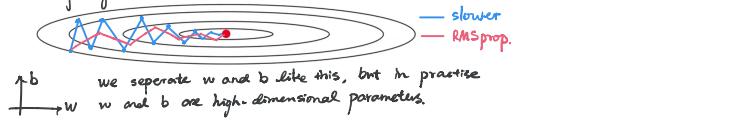


RMSprop (root mean square prop)

On iteration t : compute $d\mathbf{w}$, db on current mini-batch/batch

$$\begin{aligned} S_{d\mathbf{w}} &= \beta_2 S_{d\mathbf{w}} + (1-\beta_2) d\mathbf{w}^2 \xrightarrow{\text{square}} \mathbf{w} := \mathbf{w} - \alpha \frac{d\mathbf{w}}{\sqrt{S_{d\mathbf{w}} + \epsilon}} \\ S_{db} &= \beta_2 S_{db} + (1-\beta_2) db^2 \xrightarrow{\text{exp. weighted average of squared gradients}} b := b - \alpha \frac{db}{\sqrt{S_{db} + \epsilon}} \end{aligned}$$

How does RMSprop speed up gradient descent:



As before, we want \leftarrow faster and \rightarrow slower. Since slope is very large in the b direction and very small in the w direction, db is relatively large and $d\mathbf{w}$ is relatively small.

Andrew Ng

$$d\mathbf{w} \text{ small} \Rightarrow d\mathbf{w}^2 \text{ small} \Rightarrow S_{d\mathbf{w}} \text{ small} \Rightarrow \alpha \frac{d\mathbf{w}}{\sqrt{S_{d\mathbf{w}}}} \text{ large} \Rightarrow \text{large horizontal gradient descent step.}$$

$$db \text{ large} \Rightarrow db^2 \text{ large} \Rightarrow S_{db} \text{ large} \Rightarrow \alpha \frac{db}{\sqrt{S_{db}}} \text{ small} \Rightarrow \text{small vertical gradient descent step.}$$

The name root mean square comes from the fact that we take the exponentially weighted average of squared gradients, and then update the parameters by doing the square root of those.

Adam Optimization Algorithm (Adaptive Moment Estimation)

Adam Optimization algorithm is basically taking momentum and RMSprop and putting them together:

Initialize $V_{d\mathbf{w}} = 0$, $S_{d\mathbf{w}} = 0$, $V_{db} = 0$, $S_{db} = 0$

On iteration t : Compute $d\mathbf{w}$, db using current mini-batch/batch

$$V_{d\mathbf{w}} = \beta_1 V_{d\mathbf{w}} + (1-\beta_1) d\mathbf{w}, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1) db \quad \checkmark \text{momentum}$$

$$S_{d\mathbf{w}} = \beta_2 S_{d\mathbf{w}} + (1-\beta_2) d\mathbf{w}^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2 \quad \checkmark \text{RMS prop} \Rightarrow$$

$$V_{d\mathbf{w}}^{\text{corrected}} = V_{d\mathbf{w}} / (1-\beta_1)^t, \quad V_{db}^{\text{corrected}} = V_{db} / (1-\beta_1)^t \quad \checkmark \text{bias correction}$$

$$S_{d\mathbf{w}}^{\text{corrected}} = S_{d\mathbf{w}} / (1-\beta_2)^t, \quad S_{db}^{\text{corrected}} = S_{db} / (1-\beta_2)^t$$

$$\mathbf{w} := \mathbf{w} - \alpha \frac{V_{d\mathbf{w}}^{\text{corrected}}}{\sqrt{S_{d\mathbf{w}}^{\text{corrected}}} + \epsilon}$$

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

hyperparameters:

α : needs to be tuned

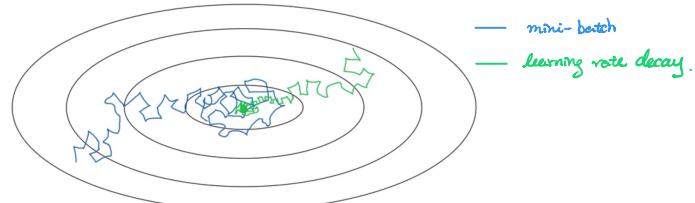
$\beta_1 = 0.9$

$\beta_2 = 0.999$

$\epsilon = 10^{-8}$

Learning rate decay

Another method that can speed up the training algorithm is to slowly reduce learning rate over time, also known as learning rate decay.



mini-batch: large steps and tends to minimum but won't exactly converge

learning rate decay: fast initial learning, but as learning rate gets smaller, steps will be smaller and will end up oscillating in a tighter region around the minimum

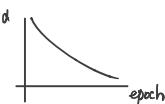
how to implement learning rate decay:

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

decay-rate is another hyperparameter that needs to be tuned.

$$\text{e.g. } \alpha_0 = 0.2 \quad \text{decay-rate} = 1$$

epoch	α
1	0.1
2	0.067
3	0.05
4	0.04
:	:



$$\text{or } = 0.95^{\text{epoch-num}} \cdot \alpha_0 \rightarrow \text{exponentially decay}$$

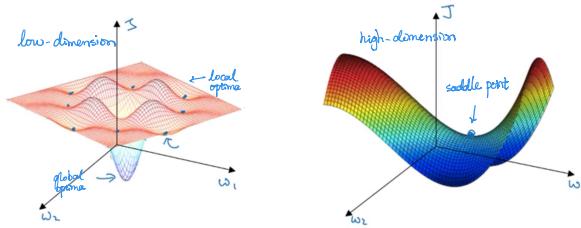
$$\text{or } = \frac{k}{\sqrt{\text{epoch-num}}} \cdot \alpha_0 \quad \text{or } \frac{k}{\sqrt{t}} \cdot \alpha_0$$



or change the decay-rate manually as we monitor the training process.

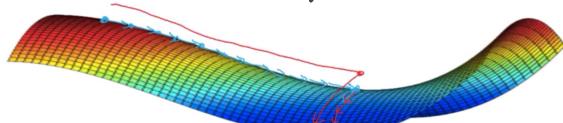
- The problem of plateaus

Unlike low-dimensional space, where there might be lots of different local optima, in high-dimensional space, most points of zero gradients are saddle points, but still it's very hard to gain intuition about what these spaces look like.



Andrew Ng

Therefore, it's very unlikely to get stuck in local optima in large MN. The real challenge is training very slowly due to plateaus.



A plateau is a region where the derivative is close to zero for a long time so the surface is quite flat and it takes a long time to find the optima.

Andrew Ng