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 traing 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)

    $X^{\{1\}} = [\, x^{(1)} \dots x^{(1000)} \,]$ $\quad X^{\{2\}} = [\, x^{(1001)} \dots x^{(2000)} \,]$ $\quad \dots \quad X^{\{5000\}} = [\, x^{(1000\times(5000-1)+1)} \dots x^{(5000\times1000)} \,]$

    similar for Y.

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

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

    Forward prop on $X^{\{t\}}$: $Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$ $\quad \dots \quad Z^{[L]} = W^{[L]} A^{[L-1]} + b^{[L]}$

    $A^{[1]} = g^{[1]}(Z^{[1]})$ $\quad \dots \quad A^{[L]} = g^{[L]}(Z^{[L]})$

    Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2\cdot1000} \sum_{l} \|W^{[L]}\|_F^2$ for $y^{(i)}$ in $Y^{\{t\}}$

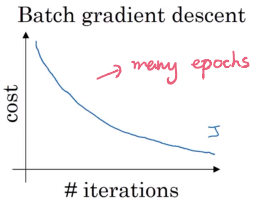
    Back prop to compute gradients wrt. $J^{\{t\}}$

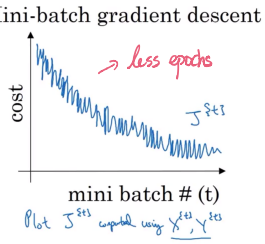
    Update $W^{[L]} := W^{[L]} - \alpha dW^{[L]}$, $b^{[L]} := b^{[L]} - \alpha db^{[L]}$

  } 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 minimise cost.

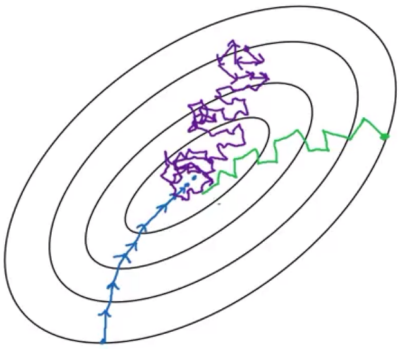




with mini-batch gradient descent, a epoch allows us to take 5000 gradient descent steps. So it's faster to minimise cost.

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 (1 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 scent (→ the power of 2 since it fits in CPU/GPU memory)
- 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:

  $V_t = \beta V_{t-1} + (1-\beta)\theta_t$ (β: hyperparameter; V_{t-1}: previous info; θ_t: current info)

  - V_t as approximately average over $\frac{1}{1-\beta}$ days { $\beta \uparrow$ → average over longer time period → smoother; $\beta \downarrow$ → average over shorter time periods → more sensitive to outliners.

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

- How to implement:

  $V_\theta = 0$

  Repeat {

  Get next θ_t

  $V_\theta := \beta V_\theta + (1-\beta)\theta_t$ → uses very little computer memory since it only stores one value.

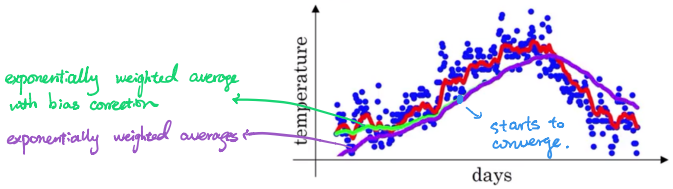
  }

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

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

($1-\beta^t$: bias correction)

- helps to improve initial estimation. Since $V_0 = 0$, $V_1 = \beta V_0 + (1-\beta)\theta_1 = (1-\beta)\theta_1$, $V_2 = \beta V_1 + (1-\beta)\theta_2 = (1-\beta)\beta\theta_1 + (1-\beta)\theta_2 \dots$
  => initial estimations tend to be very small and will only start to pick up when t is large enough, because when t is large $1-\beta^t \approx 1$, which means the bias correction term is close to 1 and $V_{t\_correct} \approx V_t$.



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 V_{dw} and V_{db} to vectors of zeros with the same dimension as w and b

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

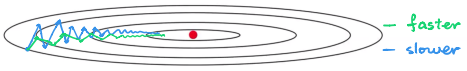
$$V_{dw} = \beta V_{dw} + (1-\beta)dw \quad \Rightarrow \quad w := w - \alpha V_{dw}$$
$$V_{db} = \beta V_{db} + (1-\beta)db \qquad b := b - \alpha V_{db}$$

(V_{dw}, V_{db}: exp. weighted average of gradients)

2 hyperparameters:
α needs to be tuned
$\beta = 0.9$ (usually works very well ≈ 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:

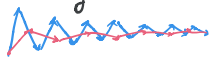


Naturally we want the gradient descent to move faster horizontally and slower vertically.

↕ slower learning
↔ faster learning

Andrew Ng

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: ⩘⩘⩘ ..., 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.

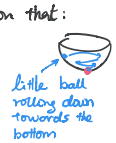


momentum: vertically averaged out towards 0 ⇒ less gradient descent steps ⇒ straight to the point
horizontally no cancelled-out effect

The name momentum comes from the intuition that:

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

(V_{dw}, V_{db}: velocity; β: friction; dw, db: acceleration)



bowl-like cost function
little ball rolling down towards the bottom
→ as the ball rolls down, it gains momentum and speeds up

- **RMSprop (root mean square prop)**

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

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2 \quad \text{(element-wise square)}$$
$$S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$$

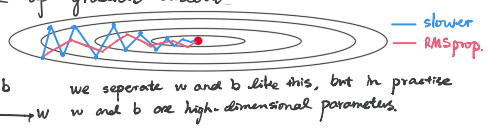
↳ exp. weighted average of squared gradients

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

add in ε to ensure that the denominator is not too small

↳ gradients decided by the square root of exp. weighted ave. of squared gradients

How does RMSprop speed up gradient descent:



we separate w and b like this, but in practise w and b are high-dimensional parameters.

As before, we want ⟷ faster and ↕ slower. Since slope is very large in the b direction and very small in the w direction, db is relatively large and dw is relatively small.

Andrew Ng

dw small $\Rightarrow dw^2$ small $\Rightarrow S_{dw}$ small $\Rightarrow \alpha \frac{dw}{\sqrt{S_{dw}}}$ large $\Rightarrow$ large horizontal gradient descent step.

db large $\Rightarrow db^2$ large $\Rightarrow S_{db}$ large $\Rightarrow \alpha \frac{db}{\sqrt{S_{db}}}$ small $\Rightarrow$ 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 diving the square root of those.

- **Adam Optimization Algorithm.** (Adaptive Moment Estimation)

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

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

On iteration t: Compute dw, db using current mini-batch/batch

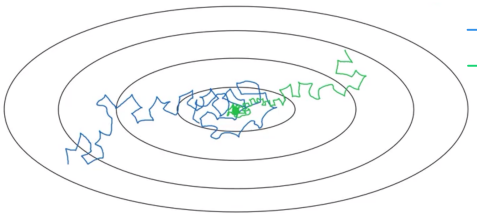
$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw, \quad V_{db} = \beta_1 V_{db} + (1-\beta_1) db \quad \text{← momentum}$$
$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2, \quad S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2 \quad \text{← RMS prop}$$
$$V_{dw}^{corrected} = V_{dw} / (1-\beta_1^t), \quad V_{db}^{corrected} = V_{db} / (1-\beta_1^t)$$
$$S_{dw}^{corrected} = S_{dw} / (1-\beta_2^t), \quad S_{db}^{corrected} = S_{db} / (1-\beta_2^t) \quad \text{← bias correction}$$

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

hyper-parameters:
- α: needs to be tuned
- β_1: 0.9
- β_2: 0.999
- ε: 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

Andrew Ng

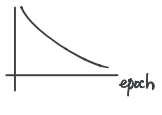
how to implement learning rate decay:

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

↳ decay-rate is another hyper-parameter that needs to be tuned.

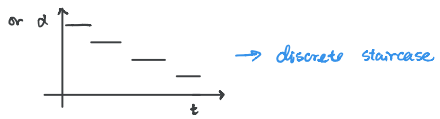
e.g. $\alpha_0 = 0.2$, decay-rate = 1

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



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

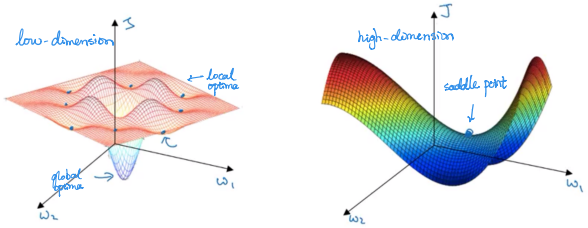
or $= \frac{k}{\sqrt{\text{epoch-num}}} \cdot \alpha_0$ or $\frac{k}{\sqrt{t}} \cdot \alpha_0$ ↳ epoch-num



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

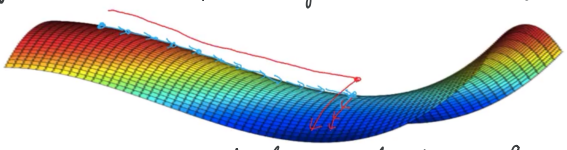
- The problem of plateaus

  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 NN. The real challenge is training very slowly due to plateaus.



A plateaus 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