

# Hyperparameters Tuning

## Week 2

### Batch vs. mini-batch gradient descent

Vectorization allows you to efficiently compute on  $m$  examples.

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(1000)} & | & x^{(1001)} & \dots & x^{(2000)} & | & \dots & | & \dots & x^{(m)} \end{bmatrix}$$

$(n \times m)$        $\underbrace{\hspace{10em}}_{X^{[1:]}}$        $\underbrace{\hspace{10em}}_{X^{[2:]}}$        $\dots$        $\underbrace{\hspace{10em}}_{X^{[5,000:]}}$

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \dots & y^{(1000)} & | & y^{(1001)} & \dots & y^{(2000)} & | & \dots & | & \dots & y^{(m)} \end{bmatrix}$$

$(1 \times m)$        $\underbrace{\hspace{10em}}_{Y^{[1:]}}$        $\underbrace{\hspace{10em}}_{Y^{[2:]}}$        $\dots$        $\underbrace{\hspace{10em}}_{Y^{[5,000:]}}$

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

5,000 mini-batches of 1,000 each

Mini-batch  $t$ :  $X^{[t:]}, Y^{[t:]}$

$x^{(i)}$   
 $z^{[2]}$   
 $x^{[t:]}, y^{[t:]}$

### Mini-batch gradient descent

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

Forward prop on  $X^{[t:]}$ .

$$z^{[2]} = W^{[1]} X^{[t:]} + b^{[2]}$$

$$A^{[2]} = \sigma^{[2]}(z^{[2]})$$

$$\vdots$$

$$A^{[L]} = \sigma^{[L]}(z^{[L]})$$

Vectorized implementation  
(1000 examples)

for  $X^{[t:]}, Y^{[t:]}$ .

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

Backprop to compute gradients w.r.t  $J^{[t]}$  (using  $X^{[t:]}, Y^{[t:]}$ )

$$W^{[1]} := W^{[1]} - \alpha dW^{[1]}, \quad b^{[2]} := b^{[2]} - \alpha db^{[2]}$$

}

"1 epoch"

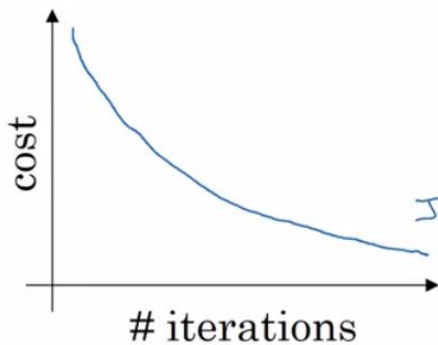
pass through training set.

1 step of gradient descent  
using  $X^{[t:]}, Y^{[t:]}$ .  
(as if  $m=1000$ )

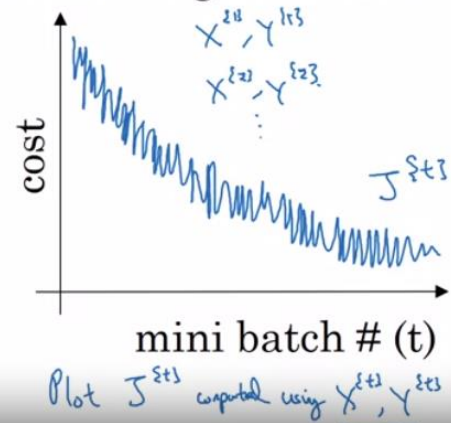
$X, Y$

# Training with mini batch gradient descent

Batch gradient descent

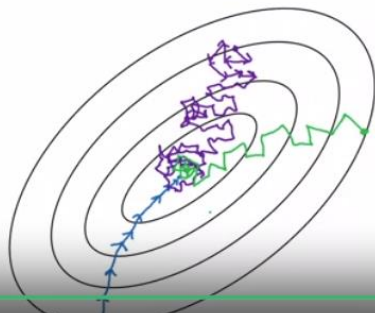


Mini-batch gradient descent



## Choosing your mini-batch size

- If mini-batch size =  $m$  : Batch gradient descent.  $(X^{t+1}, Y^{t+1}) = (X, Y)$
- If mini-batch size = 1 : Stochastic gradient descent. Every example is its own mini-batch.  $(X^{t+1}, Y^{t+1}) = (x^{(1)}, y^{(1)}) \dots (x^{(n)}, y^{(n)})$  mini-batch.
- In practice: Somewhere in-between 1 and  $m$



Stochastic  
gradient  
descent  
↓  
Use speedup  
from vectorization

In-between  
(mini-batch size  
not too big/small)

Fastest learning:

- Vectorization. (1000)
- Make pass without loading entire training set.

Batch  
gradient descent  
(mini-batch size =  $m$ )

↓  
Too long  
per iteration

# Choosing your mini-batch size

If small toy set : Use batch gradient descent.  
( $m \leq 2000$ )

Typical mini-batch sizes:

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

$\frac{1024}{2^{10}}$

Make sure mini-batch fits in CPU/GPU memory.  
 $X^{(i)}, Y^{(i)}$ .

Usually, we choose as size of mini-batch gradient a power of 2; it is more computationally efficient.

# Exponentially weighted averages

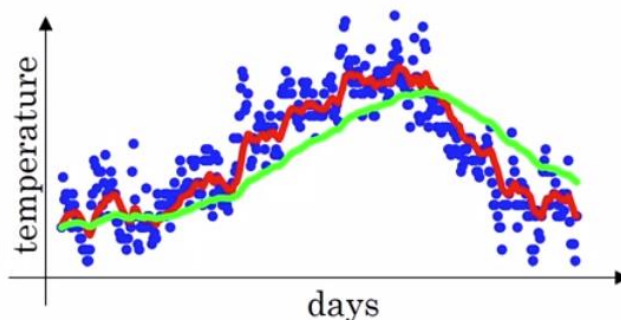
$$V_t = \beta V_{t-1} + (1-\beta) \Theta_t$$

$\beta = 0.9$  :  $\approx 10$  days' temper.

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

$V_t$  is approximately  
average over  
 $\approx \frac{1}{1-\beta}$  days'  
temperature.

$$\frac{1}{1-0.98} = 50$$

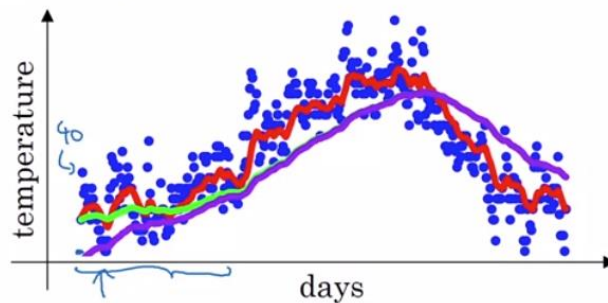


# Exponentially weighted averages

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

$v_{100} = 0.9v_{99} + 0.1\theta_{100}$   
 $v_{99} = 0.9v_{98} + 0.1\theta_{99}$   
 $v_{98} = 0.9v_{97} + 0.1\theta_{98}$   
 ...  
 $\rightarrow v_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9v_{98})$   
 $= 0.1\theta_{100} + 0.1 \times 0.9 \cdot \theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^2\theta_{97} + 0.1(0.9)^4\theta_{96} + \dots$

## Bias correction



$$\beta = 0.98$$

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

$$v_0 = 0$$

$$v_1 = 0.98v_0 + 0.02\theta_1$$

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

$$= 0.98 \times 0.02 \times \theta_1 + 0.02\theta_2$$

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

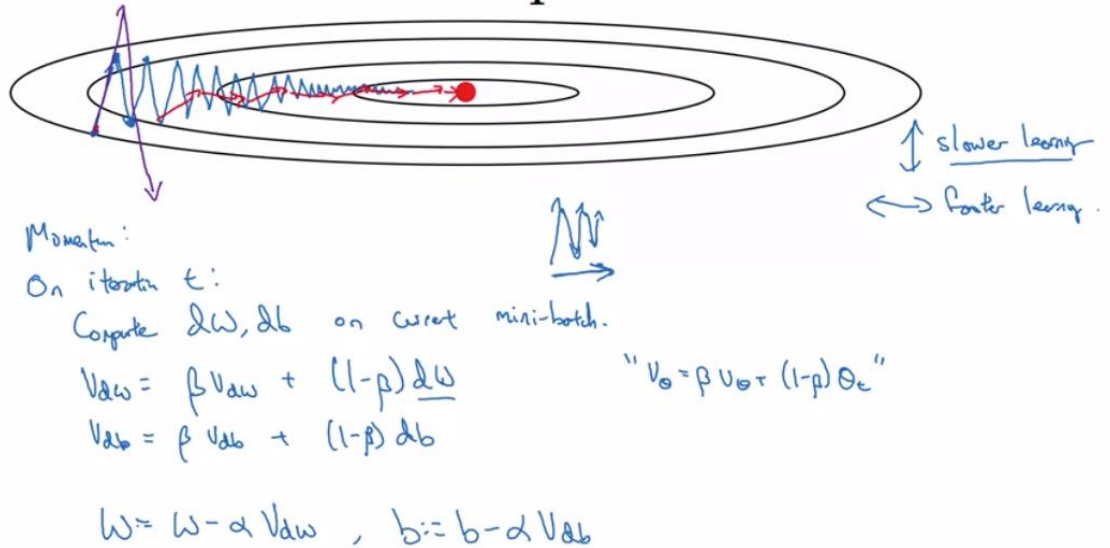
$$\frac{v_t}{1 - \beta^t}$$

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

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

Full Screen

## Gradient descent example



## Implementation details

On iteration  $t$ :

Compute  $dW, db$  on the current mini-batch

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

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

$$W = W - \alpha v_{dW}, \quad b = b - \alpha v_{db}$$

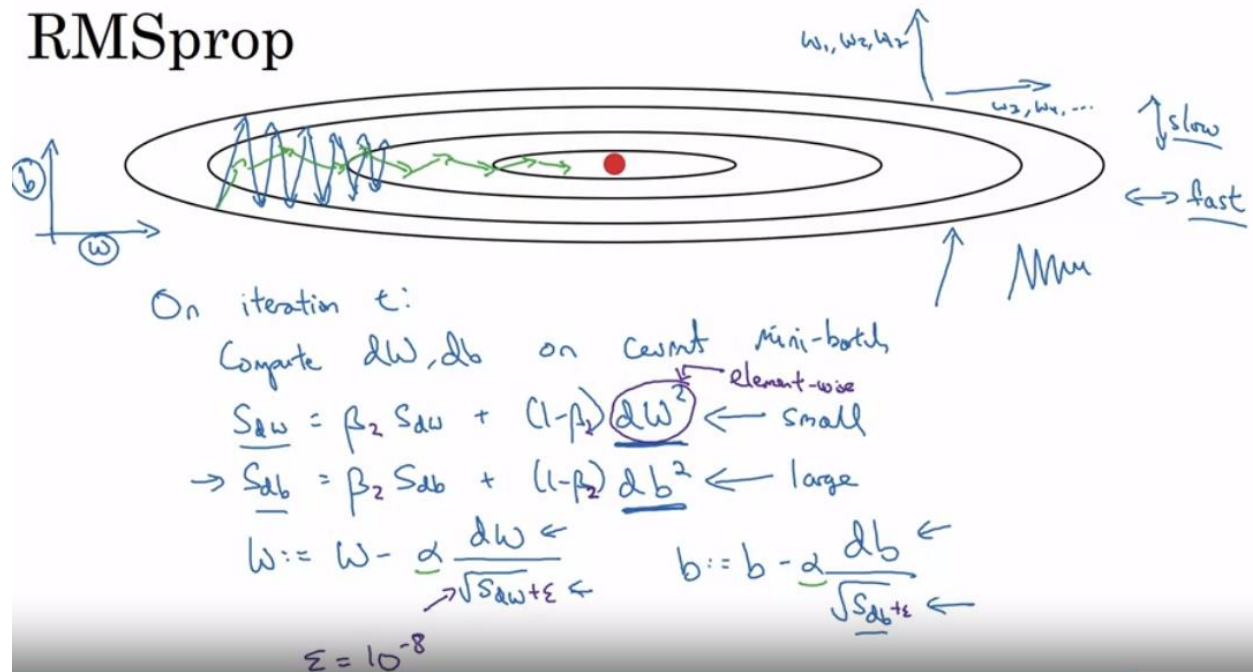
Hyperparameters:  $\alpha, \beta$

$\beta = 0.9$



Usually, there is no need to apply bias correction when implementing momentum on gradient descent.

## RMSprop



## 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

$$V_{dW} = \beta_1 V_{dW} + (1 - \beta_1) dW, V_{db} = \beta_1 V_{db} + (1 - \beta_1) db \leftarrow \text{"momentum"} \beta_1$$

$$S_{dW} = \beta_2 S_{dW} + (1 - \beta_2) dW^2, S_{db} = \beta_2 S_{db} + (1 - \beta_2) db^2 \leftarrow \text{"RMSprop"} \beta_2$$

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

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

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

# Hyperparameters choice:

- $\alpha$  : needs to be tune
- $\beta_1$  : 0.9 → ( $\underline{dw}$ )
- $\beta_2$  : 0.999 → ( $\underline{dw^2}$ )
- $\epsilon$  :  $10^{-8}$

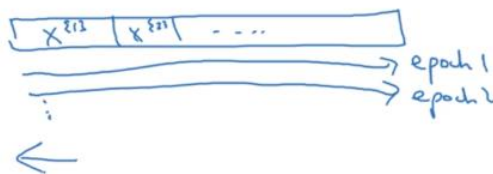
Adam: Adaptive moment estimation

Usually, people use the default values for epsilon and betas, without trying to tune them.

## Learning rate decay

1 epoch = 1 pass through data.

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



# Other learning rate decay methods

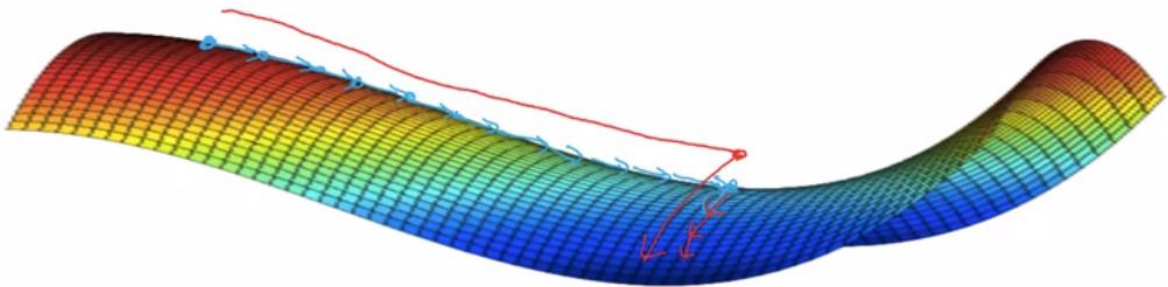
Formula

$$\left\{ \begin{array}{l} \alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0 \quad - \text{exponentially decay} \\ \alpha = \frac{k}{\sqrt{\text{epoch-num}}} \cdot \alpha_0 \quad \text{or} \quad \frac{k}{\sqrt{t}} \cdot \alpha_0 \\ \alpha \text{ vs } t \text{ graph showing discrete staircase decay} \end{array} \right.$$

Manual decay.

In [mathematics](#), a **saddle point** or **minimax point**<sup>[1]</sup> is a [point](#) on the [surface](#) of the [graph](#) of a [function](#) where the [slopes](#) (derivatives) in orthogonal directions are all zero (a [critical point](#)), but which is not a [local extremum](#) of the function.<sup>[2]</sup> An example of a saddle point is when there is a critical point with a relative [minimum](#) along one axial direction (between peaks) and at a relative [maximum](#) along the crossing axis. However, a saddle point need not be in this form. For example, the function  $f(x, y) = x^2 + y^3$  has a critical point at  $(0, 0)$  that is a saddle point since it is neither a relative maximum nor relative minimum, but it does not have a relative maximum or relative minimum in the  $y$ -direction.

## Problem of plateaus



- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow



Adam on the other hand, clearly outperforms mini-batch gradient descent and Momentum. If you run the model for more epochs on this simple dataset, all three methods will lead to very good results. However, you've seen that Adam converges a lot faster.

Some advantages of Adam include:

- Relatively low memory requirements (though higher than gradient descent and gradient descent with momentum)
- Usually works well even with little tuning of hyperparameters (except  $\alpha$ )