# Week2 - Optimization Algotithms

Samstag, 18. Juli 2020 19:52

## Mini-batch

$$X = \Big[\underbrace{x^{(1)}\ x^{(2)}\ x^{(3)}\ x^{(4)} \dots x^{(400)}}_{\text{one mini batch } x^{\{1\}} \text{ size of } 400} \;\Big|\; x^{(401)} \dots x^{(200)}\Big] \quad \# \text{ batches} = 2$$

Mini-batch gradient descent:

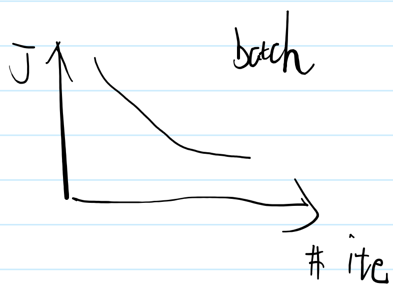
Passing input in the form of mini-batches. -> one mini-batch go through forward propagation, backward progagation, then the next mini-batch.

E.g. 500 mini-batches

For t=1, ……, 500
    Forward Propagation on x{i}
    Computer lost
    Backward Propagation to computer $w^{[l]}, b^{[l]} \dots$

} 1 epoch

Batch gradient descent: the whole dataset as one batch





Because the complexity of each batch is not the same

Stochastic gradient descent: when the mini-batch size == 1

How to choose batch size?

- When m < 2000, batch gradient descent
- Else, typical mini batch size = 64, 128, 256, 512…. $2^x$ (most fit CPU/GPU hardware structures)

## Exponentially weighted averages

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

} $\beta = 0.98$ smooth curve

$$v_t = \underbrace{\beta v_{t-1}}_{\text{The weight of previous values}} + \underbrace{(1-\beta)\theta_t}_{\text{The weight of current value}}$$

$\beta = 0.98$ smooth curve

$\beta = 0.5$ noisy and rapid changed

Especially when beta == 0.9 $\Rightarrow 0.9^{10} \approx 0.35 \approx \frac{1}{e} \quad \left( (1-\varepsilon)^{1/\varepsilon} = \frac{1}{e} \right)$

means only the last 10 values have influence on current value

So if beta == 0.98 $\quad 0.98^{50} \approx \frac{1}{e} \Rightarrow$ last 50 values have influence.

Implementation:

$v_0 = 0$

$v_1 = \beta v_0 + (1-\beta)\theta_1$

$v_2 = \beta v_1 + (1-\beta)\theta_2$

$v_\theta = 0$

$v_\theta = \beta v_\theta + (1-\beta)\theta_1$

$v_\theta = \beta v_\theta + (1-\beta)\theta_2$

$v_\theta = 0$

Repeat:

Get $\theta_t$

$v_\theta = \beta v_\theta + (1-\beta)\theta_t$

Bias correction for Exponentially weighted averages:

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

## Gradient descent with momentum

Momentum takes past gradients into account to smooth out the steps of gradient descent.


Goal: decrease delta_y
increase delta_x
Method: exponentially weighted averages

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

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

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

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

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

Because by using exponentially weighted averages we take average on both dimension. Y will be decrease close to 0 but X will not be affect, because X is all in the same direction.

$\alpha$ → learn $\quad \beta = 0.9$

## RMSprop

Elementwise

$b \uparrow$

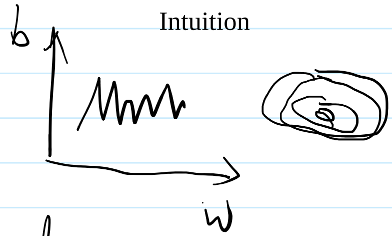
Intuition

## RMSprop

Elementwise

$$S_{dw} = \beta S_{dw} + (1-\beta) dw^2$$
$$S_{db} = \beta S_{db} + (1-\beta) db^2$$
$$w = w - \alpha \frac{dw}{\sqrt{S_{dw}+\varepsilon}} \qquad \varepsilon = 10^{-8}$$
$$b = \text{- - -}$$

Intuition




$dw \rightarrow$ small
$db \rightarrow$ large
$\frac{1}{\sqrt{S_{dw}}} \rightarrow$ large
$\frac{1}{\sqrt{S_{db}}} \rightarrow$ small

## Adam Optimization

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

On iteration $t$:

compute $dw$, $db$ on current mini-batch

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

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

$$\left.\begin{aligned} V_{dw}^{corr} &= V_{dw} / (1-\beta_1^t) \qquad V_{db}^{corr} = V_{db} / (1-\beta_1^t) \\ S_{dw}^{c} &= \text{- - -} \qquad\qquad\qquad \text{- - -} \end{aligned}\right\} \text{bias correction}$$

$$w = w - \alpha \frac{V_{dw}^{c}}{\sqrt{S_{dw}^{c}} + \varepsilon} \qquad b = \text{- -}$$

$\alpha \rightarrow$ Learned $\quad \beta_1 = 0.9 \quad \beta_2 = 0.999 \quad \varepsilon = 10^{-8}$

## Learning rate decay

$$\alpha = \frac{1}{1 + \text{decay\_rate} * \text{epoch\_num}} \alpha_0$$

Mini batch

Exponentially weighted averages

Bias correction in exponentially weighted averages
Gradient descent with momentum
RMSprop
Adam optimization algorithm
Learning rate decay

## Take away from programming assignment:

implementing SGD Stochastic Gradient Descent: requires 3 for-loops in total:

```
Over the number of iterations
Over the  m  training examples
Over the layers (to update all parameters, from (W[1],b[1]) to  (W[L],b[L])
```

Mini-batch GD:

Shuffle: Create a shuffled version of the training set (X, Y) ,  Each column of X and Y represents a training example. the  ith  column of X is the example corresponding to the  ith  label in Y (synchronously shuffle)
Partition: Partition the shuffled (X, Y) into mini-batches of size. The last mini batch might be smaller. Note, take care of the last mini-batch, if m % mini_batch_size != 0; Powers of two are often chosen to be the mini-batch size, e.g., 16, 32, 64, 128.

Momentum:

Momentum takes into account the past gradients to smooth out the update. Store the 'direction' of the previous gradients in the variable v . Formally, this will be the exponentially weighted average of the gradient on previous steps. You can also think of  v  as the "velocity" of a ball rolling downhill, building up speed (and momentum) according to the direction of the gradient/slope of the hill.
The velocity is initialized with zeros. So the algorithm will take a few iterations to "build up" velocity and start to take bigger steps.
The larger the momentum  β  is, the smoother the update because the more we take the past gradients (v/velocity) into account. But if  β  is too big, it could also smooth out the updates too much.
Common values for  ββ  range from 0.8 to 0.999, suggested 0.9

$$\begin{cases} v_{dW^{[l]}} = \beta v_{dW^{[l]}} + (1-\beta) dW^{[l]} \\ W^{[l]} = W^{[l]} - \alpha v_{dW^{[l]}} \end{cases}$$

$$\begin{cases} v_{db^{[l]}} = \beta v_{db^{[l]}} + (1-\beta) db^{[l]} \\ b^{[l]} = b^{[l]} - \alpha v_{db^{[l]}} \end{cases}$$

Adma:

Combines RMSProp and Momentum.
General idea:

calculates an exponentially weighted average of past gradients, stores it in variables  v  (before bias correction) and  v_corrected  (with bias correction).
calculates an exponentially weighted average of the squares of the past gradients/velocity, stores it in variables (before bias correction) and  s_corrected  (with bias correction).
updates parameters in a direction based on combining information from "1" and "2".

$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1-\beta_1)\frac{\partial \mathcal{J}}{\partial W^{[l]}} \\ v^{corrected}_{dW^{[l]}} = \frac{v_{dW^{[l]}}}{1-(\beta_1)^t} \\ s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1-\beta_2)(\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2 \\ s^{corrected}_{dW^{[l]}} = \frac{s_{dW^{[l]}}}{1-(\beta_2)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v^{corrected}_{dW^{[l]}}}{\sqrt{s^{corrected}_{dW^{[l]}}}+\varepsilon} \end{cases}$$

adam has a higher accuracy than mini-batch and mini-batch with only momentum. 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  α)