Since deep learning is a highly empirical and iterative process, it's really helpfull to train models quickly. Having fast optimization algorithms is important and can really speed up the efficiency of the process.

Mini-batch gradient descent

Instead of processing the entire dataset all at the same time in mini-batch gradient we partition the dataset in N smaller subsets, then each subset $(X^{[t]}, Y^{[t]})$ of the dataset are processed.

When every subset of the dataset is processed we say that we did "1 epoch" or one single pass through the training set. Usually several epochs are made.

Mini-batch gradient descent is considerably faster than normal gradient descent. A comparison between the cost functions of batch and mini-batch gradient descent is given in the following image:

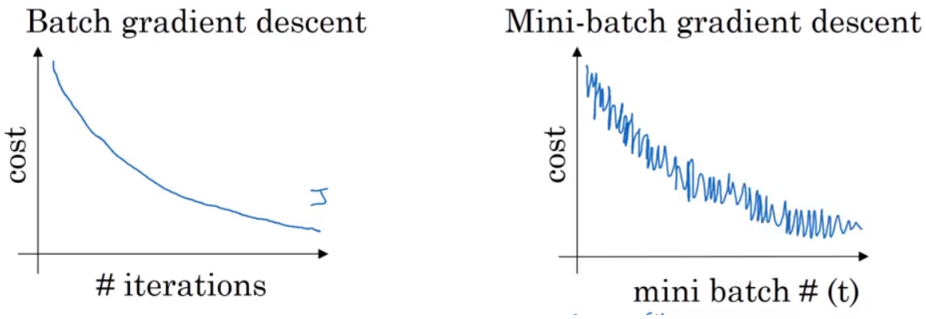

Figure 1: Batch vs Mini-batch gradient descent cost function

Differente mini-batch sizes

- mini-batch size m : Batch gradient descent.
- mini-batch size 1 : Stochastic gradient descent.

In practice the best mini-batch size should be between 1 and m; if you take mini-batch size m then it takes too long to train the neural network, in the other case taking size = 1 is really noisy and losses almost all the speed of doing vectorization.

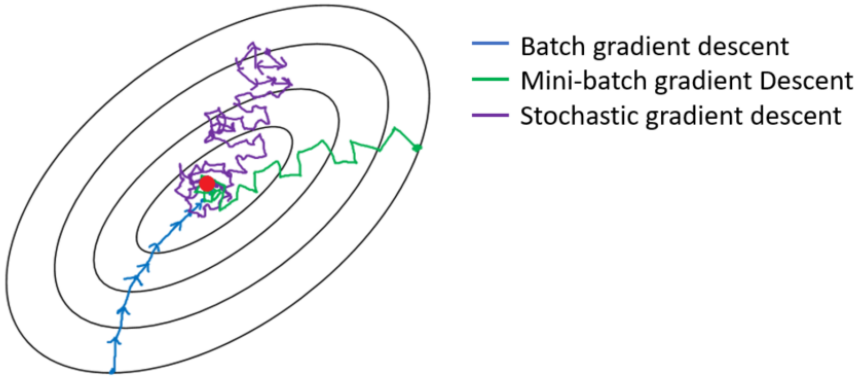

Figure 2: Comparison between mini-batch sizes

Which mini-batch size to choose?

- If the training set is small use batch gradient descent
- In any other case, typical mini-batch sizes are: $64, 128, 256, 512, 1024$. They're all powers of two because of the way computer memory is layed out and accessed, sometimes the code runs faster if the mini-batch size is a power of 2.

Exponentially weighted averages

Exponentially weighted averages is a technique for smoothing time series data using the exponential window function. Let's say we have a sequence $\{x_t\}_{t=0}^n$, the output of the exponentially weighted averages is $\{v_t\}_{t=0}^n$ where:

$$\begin{aligned} v_0 &= x_0 \\ v_i &= \beta v_{i-1} + (1-\beta)x_i \qquad\qquad \forall i > 0 \end{aligned}$$

Where β is the smoothing factor, $0 < \beta < 1$

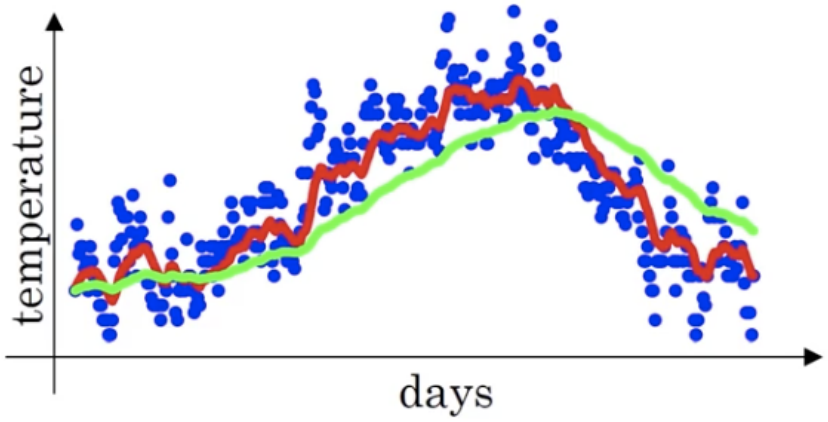

Figure 3: Comparison between different smoothing factors

The green line is for a greater β and the red line for a smaller β, in general the bigger the value of the smoothing factor the longer it takes for the exponentially weighted average to adapt.

Notice that in general v_i can be expressed as follows:

$$\begin{aligned} v_i &= \beta v_{i-1} + (1-\beta)x_i \\ &= (1-\beta)x_i + \beta(1-\beta)x_{i-1} + \beta^2 v_{i-2} \\ &\vdots \\ &= (1-\beta)x_i + \beta(1-\beta)x_{i-1} + \beta^2(1-\beta)x_{i-2} + \cdots + \beta^i(1-\beta)x_0 \\ &= \sum_{k=0}^{i} \beta^{i-k}(1-\beta)x_i \end{aligned}$$

That means that we have an exponentially decaying function.

Bias correction in exponentially weighted averages

While computing the value og the exponentially weighted averages for small values of t there's a bias because we have few observations and for big value

of β the algorithm may take a while to react to the inicialization. A way to correct this is by changing v_t with the following expression:

$$v_t^* := \frac{v_t}{1-\beta^t}$$

Notice that as t grows v_t^* approaches v_t.

Gradient descent with momentum

Usually this algorithm is faster than the standard gradient descent procedure. The idea is to compute an exponentially weighted average of the gradients and then use that to update the weights instead. Formally, Compute dW, db normally with the current mini-batch, then:

$$\begin{aligned} V_{dW} &= \beta V_{dW} + (1-\beta)dW \\ V_{db} &= \beta V_{db} + (1-\beta)db \\ W &= W - \alpha V_{dW} \\ b &= b - \alpha V_{db} \end{aligned}$$

Supose that when using the standard gradient descent you get a lot of oscilations when converging to the optimal value, using momentum can fix this because you are taking into consideration the former results of the gradient therefore the oscilation starts to decrease and the covergence is faster.

Intuition: in the convex combination of V_{dW} and dW the derivative term represents the acceleration in the current point but the momentum term let's you take into consideration the path traveled before.

Notice that now you have another Hyperparameter, β. Usually $\beta = .9$ works pretty well.

RMSprop

RMSprop, which stands for root mean square prop is another algorithm that can speed up gradient descent, on iteration t we would compute dW, db

normally with the current mini-batch and then do the following calculation:

$$S_{dW} = \beta S_{dW} + (1-\beta)dW^2$$
$$S_{db} = \beta S_{db} + (1-\beta)db^2$$
$$W = W - \alpha \frac{dW}{\sqrt{S_{dW}}}$$
$$b = b - \alpha \frac{db}{\sqrt{S_{db}}}$$

Observation: Using RMSprop makes it posible to use a higher learning rate and perform faster learning without diverging

Adam

Adam algorithm has been chown to work wll across a wide range of deep learning architectures, the basic idea is to combine momentum and RMSprop. On iteration t we would compute dW, db normally with the current mini-batch and then do the following:

Compute momentum and RMSprop:

$$V_{dW} = \beta_1 V_{dW} + (1-\beta_1)dW \qquad V_{db} = \beta_1 V_{db} + (1-\beta_1)db$$
$$S_{dW} = \beta_2 S_{dW} + (1-\beta_2)dW^2 \qquad S_{db} = \beta_2 S_{db} + (1-\beta_2)db^2$$

Perform bias correction:

$$V_{dW}^{\text{corr}} = \frac{V_{dW}}{1-\beta_1^t} \qquad V_{db}^{\text{corr}} = \frac{V_{db}}{1-\beta_1^t}$$
$$S_{dW}^{\text{corr}} = \frac{S_{dW}}{1-\beta_2^t} \qquad S_{db}^{\text{corr}} = \frac{S_{db}}{1-\beta_2^t}$$

Perform the update:

$$W = W - \alpha \frac{V_{dW}^{\text{corr}}}{\sqrt{S_{dW}^{\text{corr}}}}$$
$$b = b - \alpha \frac{V_{db}^{\text{corr}}}{\sqrt{S_{db}^{\text{corr}}}}$$

Hyperparameters choice:

- α : needs to be tuned
- β_1 : .9
- β_2 : .999

The betas can be tuned but usually those default values are used.

Learning rate decay

One of the things that might help speed up the learning algorithm, is to slowly reduce the learning rate over time. The idea is that during the initial phases while the learning rate alpha is still large, the algorithm shows a relatively fast learning. But then as alpha gets smaller, the steps it takes will be slower and smaller. And so it ends up oscillating in a tighter region around the minimum. If you never reduce the learning rate you might oscilate a lot around the minimum without converging at all.

The most used formula to compute the learning rate is the following:

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

Other learning rate decay methods:

$$\alpha = .95^{\text{epoch}}\alpha_0 \qquad \text{exponentially decay}$$

$$\alpha = \frac{k}{\sqrt{\text{epoch}}}\alpha_0 \qquad \text{constant decay}$$