Deep Learning-VI

(Optimization Algorithms for Neural Networks)

DR. JASMEET SINGH

ASSISTANT PROFESSOR,

CSED, TIET

Introduction

- Neural Networks are generally trained on a larger set of training examples as compared to machine learning algorithms.
- So, faster optimization algorithms can speed up the training process of neural networks.
- Some of the optimization algorithms commonly used for training neural networks are:
  - Mini-batch Gradient Descent Algorithm
  - Gradient Descent with Momentum
  - RMSprop
  - Adam

Mini-Batch Gradient Descent

- Traditional gradient descent algorithm (also called batch gradient descent) works on the entire training set X and Y of shapes (k, n) and (1,n) respectively ; where k are number of features and n are the number of training examples.
- In mini-batch gradient descent algorithm, we divide n training examples into T mini batches such that each mini-batch contains m examples where $m = \frac{n}{T}$
- Any t^{th} mini-batch training set is represented as X^t and Y^t and is of dimensions (k,m) and (1,m) respectively.

Mini-Batch Gradient Descent Algorithm

For i=1 to epochs

For t =1 to T #where T represents number of mini-batches

Forward Propagation:

For l =1 to L (number of layers)

$$Z_l{}^t = W_l.(A_{l-1})^t + b_L \quad where\ A_0 = X^t$$

$$A_l{}^t = g_l(Z_l{}^t)$$

Compute Cost:

$$J_t = \frac{-1}{m}\sum_{i=1}^{m}\left((y_i)^t \log(a_{Li})^t + (1-y_i{}^t)\log(1-a_{Li}{}^t)\right)\ where\ y_i\ \varepsilon Y^t$$

Backward propagation:

For l =L to 1 (number of layers)

$$dZ_l^t = dA_l^t * g_l'(Z_l^t)$$

$$dW_l = \frac{1}{m} dZ_l^t . A_{l-1}^t{}^T$$

$$dB_l = \frac{1}{m} np.sum(dZ_l^t, axis = 1, keepdims = True)$$

$$dA_{l-1}^T = W_l^T . dZ_l^t$$

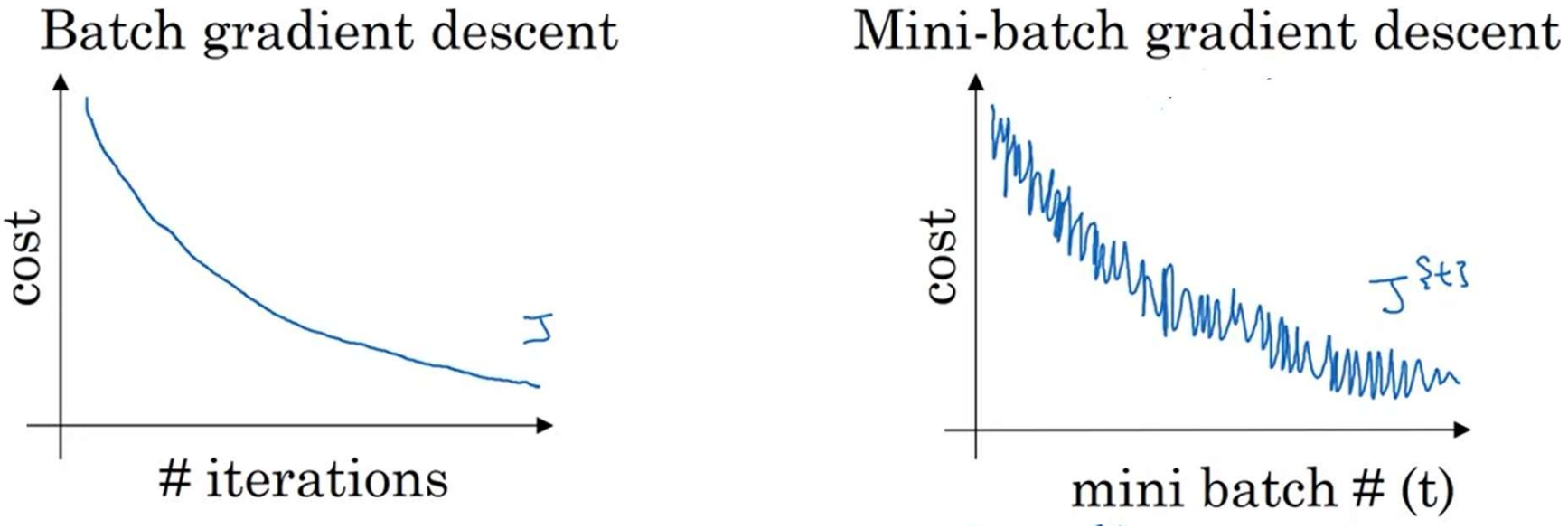
Update Parameters:

$$W_l = W_l - \alpha \times dW_l$$

$$B_l = B_l - \alpha \times dB_l$$

(

Batch vs. Mini-Batch



Stochastic Gradient Descent (SGD)

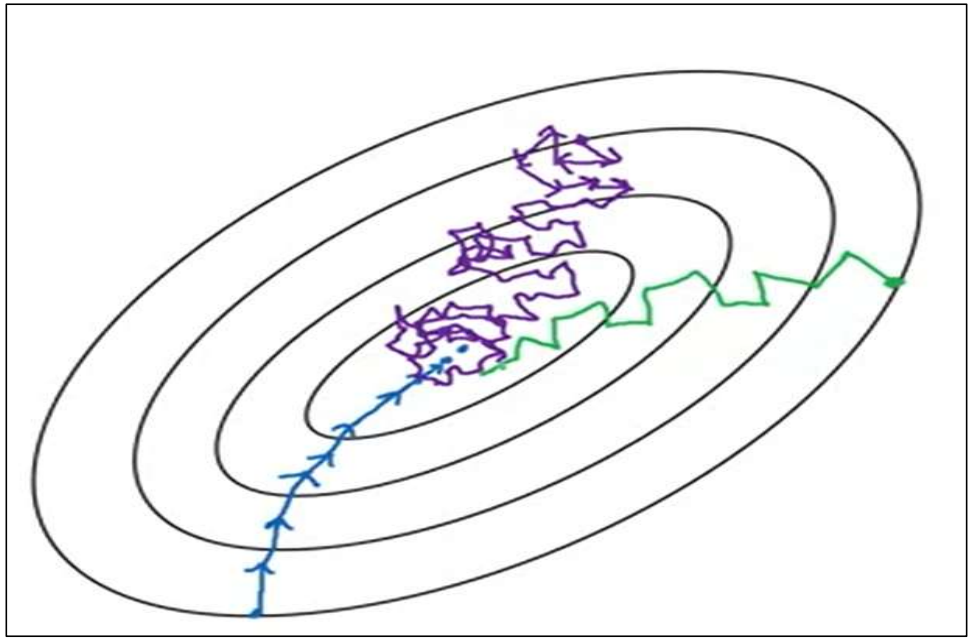
- Stochastic Gradient size is a type of mini-batch gradient descent algorithm where the size of mini-batch is 1 (i.e. m=1).
- So, there are n mini-batches (i..e. T=n) and each t^{th} training set (X^t, Y^t) is of dimension (k,1) and (1,1).

Batch vs. Mini-Batch vs. SGD

	Stochastic Gradient Descent	Mini-Batch Gradient Descent	Batch Gradient Descent
Batch Size	1 training example	m training examples $m = \frac{n}{T}$ 1 < m < n	n training examples
Speed	slowest, no use of vectorization	Fastest, use vectorization where vector size is not to large	Slower per iteration, use vectorization where vector size is very large
Convergence	May not converge to global minimum	May not converge to global minimum; but oscillate close to global minima	Always converge to global minimum

Batch vs. Mini-Batch vs. SGD

Batch Gradient Descent **Stochastic Gradient Descent** **Mini-Batch Gradient Descent**

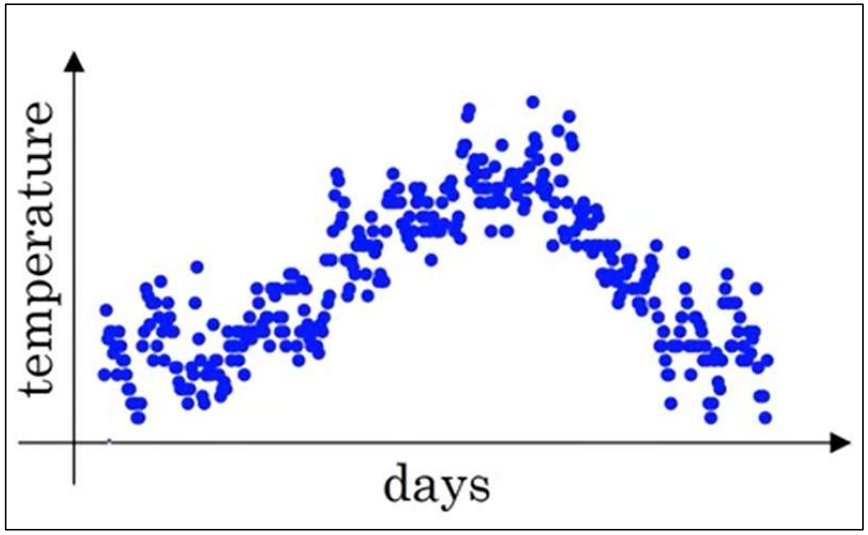


Choosing Mini-batch size

- If the number of training examples are not too large (i.e. n<2000)
  - Use Batch Gradient Descent Algorithm
- If the number of training examples are large (i.e. n>2000)
  - Use Mini-Batch gradient descent algorithm
  - Use mini-batch size from 64 to 1024 i.e. prefer mini-batch size as power of 2
  - Choosing a mini-batch size is another hyperparameter that can be tuned.

Exponentially Weighted Moving Averages

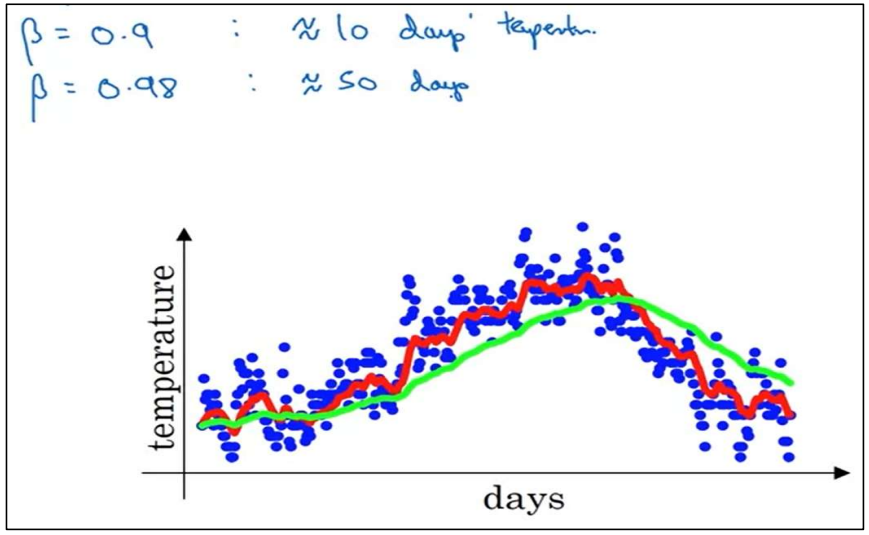
PLOT OF TEMPERATURE W.R.T EACH DAY OF AN YEAR



EFFECT OF WEIGHTED MOVING AVERAGES

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

WHERE V_t AND θ_t ARE WEIGHTED AND ACTUAL TEMPERATURES ON T^{TH} DAY

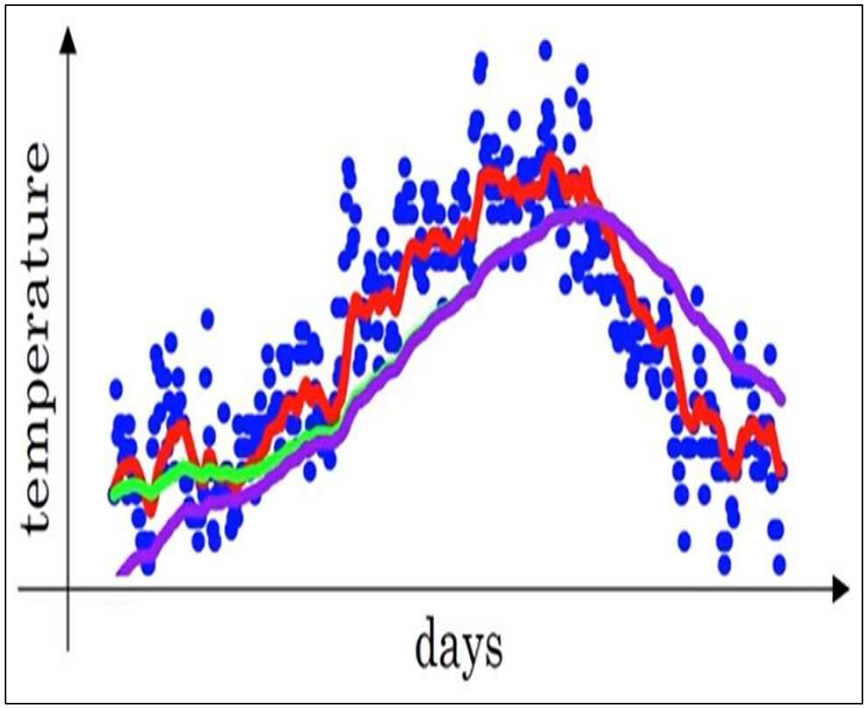


Bias Correction in Weighted Averages

- If β is very small i.e. 0.98, then for the initial values the curve starts very low (as shown by the purple line in the fig)
- So, a bias correction is usually applied to tackle the problem:

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

$$V_t = \frac{V_t}{(1-\beta^t)}$$

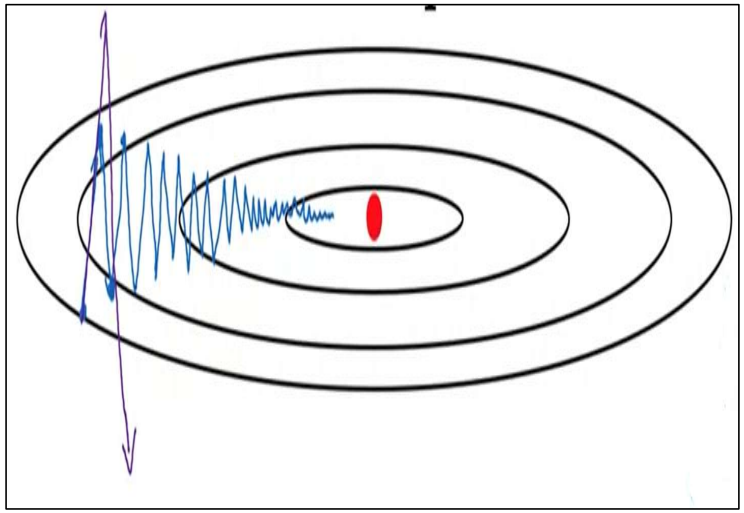


Gradient Descent with Momentum

- Gradient descent with momentum algorithm is generally faster than traditional gradient descent algorithm.
- The basic idea is to compute an exponentially weighted average of the gradients, and then use that gradient to update the weights instead.

Gradient Descent with Momentum (Contd…)

- Let's say that we are trying to optimize a cost function which has contours (as shown in figure). So the red dot denotes the position of the minimum.
- Let say with some learning rate, gradient descent algorithm slowly oscillate toward the minimum.
- It prevents us from using a much larger learning rate. In particular, if we were to use a much larger learning rate we might end up over shooting.
- **Another way of viewing this problem is that on the vertical axis we want our learning to be a bit slower, because we don't want those oscillations. But on the horizontal axis, you want faster learning.**



Gradient Descent with Momentum Algorithm

Initialize $VdW_L = 0\ and\ Vdb_L = 0$; Initialize W_L and B_L

For i=1 to epochs

For t =1 to T #where T represents number of mini-batches

Forward Propagation:

For l =1 to L (number of layers)

$$Z_l{}^t = W_l.(A_{l-1})^t + b_L \quad where\ A_0 = Xt$$

$$A_l{}^t = g_l(Zl^t)$$

Compute Cost:

$$J_t = \frac{-1}{m}\sum_{i=1}^{m}\big((y_i)^t \log(a_{Li})^t + (1-y_i{}^t)\log(1-a_{Li}{}^t)\ \ where\ y_i\ \varepsilon Y^t$$

Backward propagation:

For l =L to 1 (number of layers)

$$dZ_l^t = dA_l^t * g_l'(Z_l^t)$$

$$dW_l = \frac{1}{m}dZ_l^t.A_{l-1}^t{}^T$$

$$dB_l = \frac{1}{m}np.sum(dZ_l^t, axis = 1, keepdims = True)$$

$$dA_{l-1}^T = W_l^T.dZ_l^t$$

- $$VdW_l = \beta \times (VdW_l) + (1-\beta) \times (dW_l)\ ;\ VdW_l = \frac{VdW_l}{1-\beta^t}$$

$$\mathrm{V}dbB_l = \beta \times (VdB_l) + (1-\beta) \times (dB_l)\ ;\ VdB_l = \frac{VdB_l}{1-\beta^t}$$

Update Parameters:

$$W_L = W_L - \alpha \times (VdW_L)$$

$$B = B_L - \alpha \times (VdB_L)$$

Gradient descent with Momentum Intuition

What makes Gradient Descent with Momentum fast?

- If we average out these gradients, we find that the oscillations in the vertical direction will tend to average out to something closer to zero. So, in the vertical direction, where we want to slow things down, this will average out positive and negative numbers, so the average will be close to zero.
- On the horizontal direction, all the derivatives are pointing to the right of the horizontal direction, so the average in the horizontal direction will still be pretty big.

Why it is called momentum?

- Imagine that you have a bowl, and you take a ball and the derivative imparts acceleration to this little ball as the little ball is rolling down this hill.
- So it rolls faster and faster, because of acceleration.
- The parameter β, because this number a little bit less than one, displays a row of friction and it prevents your ball from speeding up without limit.
- So rather than gradient descent, just taking every single step independently of all previous steps. Now, little ball can roll downhill and gain momentum, but it can accelerate down this bowl and therefore gain momentum.

RMSProp

- RMSProp (also called as Root Mean Square Prop) can also speed-up the gradient descent algorithm.
- Like, Gradient Descent with momentum, RMSprop also computes weighted averages of the gradients of each iteration and use them to update the weights.
- But rather than using moving weighted averages, it uses root mean square updation

RMSProp Algorithm

Initialize $SdW_L = 0\ and\ SdB_L = 0$; Initialize W_L and B_L

For i=1 to epochs

For t =1 to T #where T represents number of mini-batches

Forward Propagation:

For l =1 to L (number of layers)

$$Z_l{}^t = W_l.(A_{l-1})^t + b_L \quad where\ A_0 = Xt$$

$$A_l{}^t = g_l(Zlt)$$

Compute Cost:

$$J_t = \frac{-1}{m}\sum_{i=1}^{m}\big((y_i)^t \log(a_{Li})^t + (1-y_i{}^t)\log(1-a_{Li}{}^t)\ \ where\ y_i\ \varepsilon Yt$$

Backward propagation:

For l =L to 1 (number of layers)

$$dZ_l^t = dA_l^t * g_l'(Z_l^t)$$

$$dW_l = \frac{1}{m} dZ_l^t . A_{l-1}^{t\ T}$$

$$dB_l = \frac{1}{m} np.sum(dZ_l^t, axis = 1, keepdims = True)$$

$$dA_{l-1}^T = W_l^T . dZ_l^t$$

$$SdW_l = \beta \times (SdW_l) + (1-\beta) \times (dW_l^2)\ ;\ SdW_l = \frac{SdW_l}{1-\beta^t}$$

$$SdB_l = \beta \times (SdB_l) + (1-\beta) \times (dB_l^2);\ Sdb_l = \frac{Sdb_l}{1-\beta^t}$$

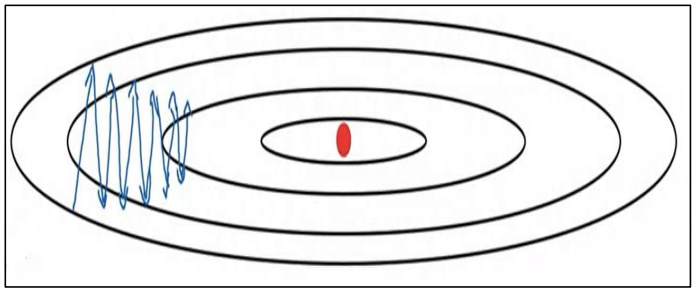
Update Parameters:

Update $W_L = W_L - \alpha \times \frac{dW_L}{\sqrt{SdW_L}+\epsilon}$

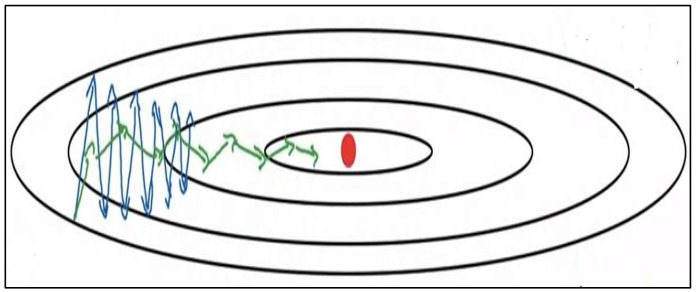
Update $B_L = B_L - \alpha \times \frac{db_L}{\sqrt{Sdb_L}+\epsilon}$ where $\epsilon\ is\ a\ samll\ number\ generally\ taken\ (10^{-8})\ to\ avoid\ divide\ by\ zero\ problem$

RMSProp Intuition

- Consider the behavior of gradient-descent algorithm for cost function, shown in figure (a), Let one axis represents b (Vertical) and other axis represents W (horizontal).
- In each iteration db will be large (since along vertical axis oscillations are large), so will be db^2, hence Sdb will also be large.
- On the other hand, dW will be small (since along horizontal axis oscillations are less), so will be dW^2, hence SdW will also be small.
- So, in each iteration, W will be larger (as a smaller value is deducted from it) and b will be smaller as a larger value is deducted from it. Hence, it will slow down oscillations in vertical direction and fast down oscillations in horizontal direction (as shown by green color in figure (b).



(a)



(b)

ADAM Optimization

- Adam combines the advantages of both gradient descent with momentum and RMSprop.
- It stands for ADAptive Moments estimation.
- In each iteration it computes both Weighted Moving average of gradients (i.e. VdW_L, Vdb_L) and RMS average of gradients (i.e. SdW_L, Sdb_L) and use both of them to updates weights and bias (W_L and b_L).
- Since it estimates from two moments, one moment is Weighted moving average and other is RMS average , therefore it is named as Adaptive Moments Estimation.

ADAM Algorithm

Initialize $VdW_L = 0, Vdb_L = 0, SdW_L = 0, Sdb_L = 0$

For i =1 to epochs:

Compute dW_L and db_L according to mini-batch gradient descent algorithm for the iteration

$VdW_L = \beta_1 \times (VdW_L) + (1 - \beta_1) \times (dW_L)$, $Vdb_L = \beta_1 \times (Vdb_L) + (1 - \beta_1) \times (db_L)$

$SdW_L = \beta_2 \times (SdW_L) + (1 - \beta_2) \times (dW_L^2)$, $Sdb_L = \beta_2 \times (SdB_L) + (1 - \beta_2) \times (db_L^2)$

$VdW_L = \frac{VdW_L}{1-\beta_1^t}$, $Vdb_L = \frac{Vdb_L}{1-\beta_1^t}$

$SdW_L = \frac{SdW_L}{1-\beta_2^t}$, $Sdb_L = \frac{Sdb_L}{1-\beta_2^t}$

Update $W_L = W_L - \alpha \times \frac{(VdW_L)}{\sqrt{SdW_L}+\epsilon}$ Update $b_L = b_L - \alpha \times \frac{(Vdb_L)}{\sqrt{Sdb_L}+\epsilon}$

where default values of $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$