

# COURSE 2 W2

## Vid 1

### Minibatch Gradient Descent

Bu hafta NN'lerimizi daha hızlı eğitmeye olanak sağlayacak optimizasyon algoritmalarını göreceğiz.

Deep learning big data üzerinde çalıştığı için, learning uzun sürebilir bu yüzden iyi çalışan bir optimizasyon algoritması bulmak verimliliğimizi önemli ölçüde artırır.

Bu kısımda minibatch gradient descent'ten bahsedeceğiz.

Önceki kısımlarda vektörizasyon ile işleri hızlandırdığımızdan bahsetmiştik, training example'ları gezmek için loop kullanmıyorduk, X ve Y vektörlerini kullanıyorduk. Ancak eğer m çok büyükse let's say 5000000 o zaman, vektörizasyon bile yavaş kalır.

Tek bir gradient step için 5000000 example üzerinde hesap yapmamız lazım uzun iş.

Bu noktada yapılabilecek bir şey şu: training set'i daha küçük training setlere yani minibatch'lere böleriz, mesela 1000 examplelık gruplara. Sonuçta 5m'lik training set içinde 5000 adet minibatch olacak. Aynı bölme işlemini Y üzerinde yaparız. Burada {} ile yeni bir notasyon söz konusu,  $X\{1\}$  dediğimizde 1. Minibatchten yani ilk 1000 example'ı kaplayan gruptan bahsederiz.

Sonuçta sanki elimizde 5000 farklı training set varmış gibi davranıcaz her minibatch eğitiminden sonra bir gradient step atılacak. Böylece eğitim hızlanacak.

## Batch vs. mini-batch gradient descent

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

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

$(n_x, m)$   $X^{\{1\}}$   $(n_x, 1000)$   $X^{\{2\}}$   $(n_x, 1000)$   $X^{\{5,000\}}$   $(n_x, 1000)$

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

$(1, m)$   $Y^{\{1\}}$   $(1, 1000)$   $Y^{\{2\}}$   $(1, 1000)$   $Y^{\{5,000\}}$   $(1, 1000)$

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

5,000 mini-batches of 1,000 each

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

$x^{(i)}$   
 $z^{[l]}$   
 $X^{\{t\}}, Y^{\{t\}}$

Uygulamasına bakarsak aslında çok yeni bir şey yok, sadece bir for loop oluşturuyoruz ve her minibatch'i sanki bizim X ve Y'miz gibi alıp, loop içinde forward ve backward prop yapıyoruz, her minibatch için bir gradient step atıyoruz. 5000 adımın sonunda, bir epoch tamamlanmış oluyor yani tüm training set'i bir kez dönmüş oluyoruz, dilersek bu işlemi tekrarlayabiliriz yani epoch sayısını artırabiliriz.

## Mini-batch gradient descent

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

Forward prop on  $X^{set}$ .

$$Z^{(t)} = W^{(t)} X^{set} + b^{(t)}$$

$$A^{(t)} = \sigma^{(t)}(Z^{(t)})$$

...

$$A^{(t)} = \sigma^{(t)}(Z^{(t)})$$

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

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

$$W^{(t+1)} = W^{(t)} - \alpha dW^{(t)}, \quad b^{(t+1)} = b^{(t)} - \alpha db^{(t)}$$

}

"1 epoch"  
└ pass through training set.

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

$X, Y$

Vectorized implementation (1000 examples)

for  $X^{set}, Y^{set}$ .

Andrew Ng

Training setimiz büyükse minibatch gradient descent, batch gradient descente göre bariz şekilde daha hızlı çalışır.

Bence yukarıdaki gösterimde loss function'ın sumında 1 olmayacak. Sonuçta orada hesaplanan, bir minibatch için ortalama error. Yani her example için bulunan error'un 1000'e bölümü.

## Vid 2

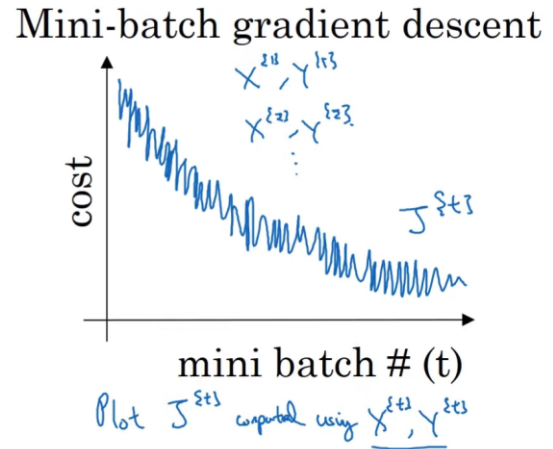
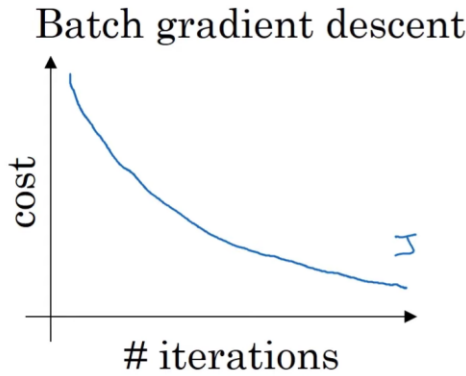
### Understanding Minibatch Gradient Descent

Batch gradient descent için her iterasyonda, cost function düşmeli, eğer düşmüyorsa bir problem var demektir mesela learning rate seçimi yanlış vb.

Ancak minibatch için durum böyle olmak zorunda değil, her minibatch gradient step için düşüş beklemeyiz, çünkü her seferinde bir başka training set üzerinde eğitim yapıyor ve cost'u buna göre hesaplıyoruz, sanki bağımsız setlerin cost'unu hesaplıyoruz gibi düşün, dolayısıyla burada gürültü olması çok normal.

Belki  $X\{1\} Y\{1\}$  için cost azken,  $X\{2\} Y\{2\}$  için durum farklı olabilir, zigzaglar bu yüzden. Ancak yine genel trendin düşüş olmasını bekleriz!

# Training with mini batch gradient descent

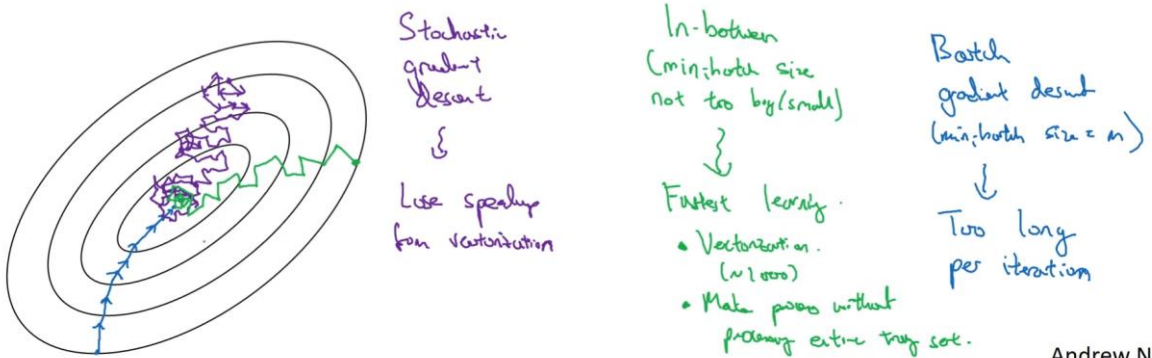


Andrew Ng

Peki minibatch size'ı nasıl seçeceğiz?

## 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  $(X^{t+1}, Y^{t+1}) = (x^{(1)}, y^{(1)}) \dots (x^{(n)}, y^{(n)})$  mini-batch.
- In practice: Somewhere in-between 1 and  $m$



Andrew Ng

Eğer minibatch size= $m$  ise bu bildiğimiz batch gd olur. Benzer şekilde size=1 dersek bu da stochastic gradient descent olur, her example için bir gd step atarız.

Farklı sızelar için gd steplerinin nasıl olacağı yukarıda resmedilmiş, batch gd ile büyük ve gürültüsüz adımlar, minimum'a doğru atılır ancak bu yavaş olur. Stochastic gd ile her example için adım atılır ancak gürültülüdür ve minimum'a converge etmez, minimum'un etrafında dolandır. Minibatch gd ise bu ikisinin arasındadır.

Stochastic gradient descent ile ortaya çıkan problem ise şu, vektörizasyon ile ortaya çıkan hız avantajını kullanamıyoruz. Her example için tek tek forward ve backward prop yapıyoruz, cost hesaplıyoruz uzun iş.

E o zaman, batch gd yavaş çünkü eğer m büyükse tek bir adım için tüm example'ları dönmesi gerek uzun iş, stochastic gd de yavaş çünkü vektörizasyon kullanmıyor her example için forward backward prop ayrı ayrı hesaplanıyor.

En mantıklısı minibatch gradient descent için iyi bir batch size seçip hem, bir adım atmak için tüm örnekleri dönmeyi beklememek hem de vektörizasyondan feragat etmemek ve bir forward backward prop ile birden fazla örnek için hesap yapmak.

Ayrıca minibatch ve stochastic için ortaya çıkan non converge to minimum problemini daha sonra göreceğimiz learning rate'i düşürme yöntemiyle çözebiliriz, küçük adımlar atınca cost giderek minimuma conver edecektir.

Sonuçta minibatch size seçimine gelirsek:

## Choosing your mini-batch size

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

Typical mini-batch sizes:

→ 64, 128, 256, 512  $\frac{1024}{2^{10}}$   
 $2^6 \quad 2^7 \quad 2^8 \quad 2^9$

Make sure minibatch fits in CPU/GPU memory.  
 $X^{(k)}, Y^{(k)}$

Andrew Ng

Eğer m küçük ise mesela  $m < 2000$ , batch gd kullan geç, bariz bir yavaşlama olmayacaktır.

Diğer durumlarda, tipik minibatch sizes: 64, 128, 256, 512 gibi değerler. Genelde 2'nin katı olarak seçilir. Minibatch size 2'nin katı olarak seçilirse internal sebeplerden dolayı bazen kod daha hızlı çalışır.

Son olarak, bir minibatch'in CPU/GPU memory'e uygun olduğuna dikkat etmekte fayda var. Eğer minibatch size CPU/GPU memory'e uymuyorsa performans bariz şekilde düşebilir.

Tabii ki mini-batch size da bir başka hyperparameter, deneme yanılma ile en uygununu bulabiliriz.

Sonuç olarak şunu söylemeliyiz ki, aslında batch gd veya mini-batch gd'den de daha hızlı optimizasyon algoritmaları var!

# Vid 3

## Exponentially Weighted Averages

Gradient descentten daha hızlı optimizasyon algoritmalarından bahsedeceğiz, ancak bu algoritmaları daha iyi anlayabilmek için öncelikle “Exponentially Weighted Averages” konseptini anlayacağız.

Buna bazen “Exponentially Weighted Moving Averages” da denir. Bu kavramı anladıktan sonra, daha karmaşık optimizasyon algoritmalarında kullanacağız.

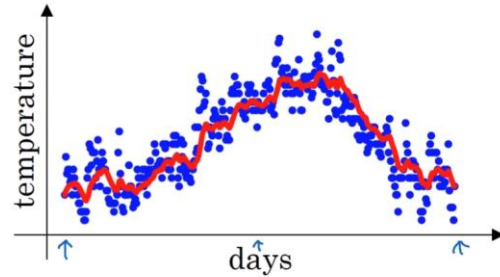
Elimizde yıl boyunca günlük sıcaklık verilerinin olduğunu varsayalım, bu aşağıdaki grafikte gösteriliyor.

Mavi noktalar ile gösteriler scatter plot biraz noisy, eğer bundan kurtulmak istiyorsak ve genel trendi elde edeceğimiz daha smooth bir grafik elde etmek istiyorsak, weighted average kavramından yararlanabiliriz.

Bunun için şöyle bir formülizasyon kullanılıyor: Sanki son 10 günün average’ını almışız gibi yapmak istiyoruz ama tam olarak bunu yapmıyoruz: İlk günden önce 9 gün 0 sıcaklığı varmış gibi kabul ediyoruz,  $V_1 = 0 + 0.1\theta_1$  diyoruz. 2. Gün için yine  $0.9V_1 + 0.1\theta_2$  şeklinde gidiyor, bu formülizasyon ile elde edilen  $V_0, V_1, \dots$  sonunda plot edilince 10 günlük moving average uygulamışız gibi bir effect oluşuyor:

## Temperature in London

$$\begin{aligned}\theta_1 &= 40^\circ\text{F} & 4^\circ\text{C} \leftarrow \\ \theta_2 &= 49^\circ\text{F} & 9^\circ\text{C} \\ \theta_3 &= 45^\circ\text{F} & \vdots \\ &\vdots & \\ \theta_{180} &= 60^\circ\text{F} & 15^\circ\text{C} \\ \theta_{181} &= 56^\circ\text{F} & \vdots \\ &\vdots & \end{aligned}$$



$$\begin{aligned}V_0 &= 0 \\ V_1 &= 0.9V_0 + 0.1\theta_1 \\ V_2 &= 0.9V_1 + 0.1\theta_2 \\ V_3 &= 0.9V_2 + 0.1\theta_3 \\ &\vdots \\ V_t &= 0.9V_{t-1} + 0.1\theta_t\end{aligned}$$

Andrew Ng

Burada genel formüş  $V_t = \text{Beta} \cdot V_{t-1} + (1-\text{Beta}) \cdot \theta_t$  biz yukarıda  $\text{Beta} = 0.9$  olarak seçildi bu yüzden  $1/(1-\text{Beta}) = 10$  günlük moving average etkisi yarattık.

Eğer Beta = 0.98 olarak seçilirse, 50 günlük moving average uygulamış gibi oluruz, bu grafik yeşilli gösteriliyor: Çok daha smooth bir grafik ancak, biraz delay olduğunu görüyoruz bunun sebebi sıcaklıktaki ani değişimlere tepki verememesi çünkü ortalama 50 günlük alındığı için ani bir sıcaklık değişimi etki yapamıyor, etki gecikmeli geliyor

## Exponentially weighted averages

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

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

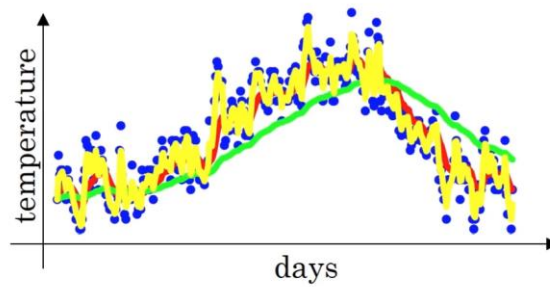
$\beta = 0.98$  :  $\approx 50$  days

$\beta = 0.5$  :  $\approx 2$  days

$V_t$  is approximately  
average over

$\rightarrow \approx \frac{1}{1-\beta}$  days'  
temperature.

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



Andrew Ng

Eğer betayı 0.5 olarak set edersek bu 2 günlük moving average uygulamakla eşdeğer bu durumda, grafik daha noisy oluyor ancak sıcaklık değişimlerini hızlı adapte oluyor, bir gecikme söz konusu değil.

E neden direkt moving average uygulamıyoruz da böyle bir yaklaşım yapıyoruz? Çünkü böylesi computationally more efficient, moving average'da son 10 veriyi tutman gerekiyor vesaire...

## Vid 4

### Understanding Exponentially Weighted Averages

EWA kavramı ilerleyen kısımlarda göreceğimiz yeni optimizasyon algoritmalarında anahtar rol oynayacak bu sebeple bu kavramı biraz daha iyi anlamaya çalışalım.

Aşağıda 100 data point için exponentially weighted average formülünün açılmış halini görüyoruz. Burada Beta=0.9 kabul edersek yani 10 günlük average alıyoruz yaklaşımı yaparsak.  $V_{100} = 0.1 \Theta_{100} + 0.1 * 0.9 \Theta_{99} + 0.1 * 0.9^2 \Theta_{98} + \dots$  şeklinde gidiyor.

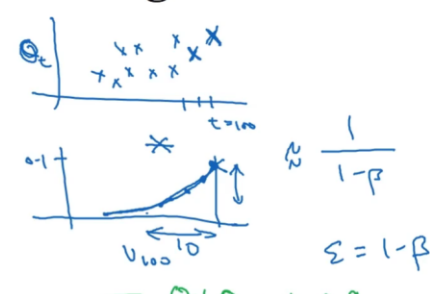
Burada  $\beta^{(1/(1-\beta))}$ 'nin sürekli 1/e ye yani yaklaşık 0.35'e denk geldiğini de ekleyelim. Yani 10 günlük average alıyorsak 0.9'un 10. kuvveti 0.35'e denk geliyor ki bundan sonrasını yok sayıyoruz gibi düşün, böylece 10 günden öncesi sanki etki etmiyormuş gibi oluyor.



# Exponentially weighted averages

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

$v_{100} = 0.9v_{99} + 0.1\theta_{100}$   
 $v_{99} = 0.9v_{98} + 0.1\theta_{99}$   
 $v_{98} = 0.9v_{97} + 0.1\theta_{98}$   
 ...  
 $\rightarrow v_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9v_{98})$   
 $= 0.1\theta_{100} + 0.1 \times 0.9 \cdot \theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^3\theta_{97} + \dots$   
 $0.9^{100} \approx 0.35 \approx \frac{1}{e}$   
 $\frac{(1-\epsilon)^{1/\epsilon}}{0.9} \approx \frac{1}{e}$   
 $\epsilon = 0.02 \rightarrow 0.98^{50} \approx \frac{1}{e}$



Andrew Ng

Implementasyonu da aşağıda görülüyor.

## Implementing exponentially weighted averages

$$v_0 = 0$$

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

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

$$v_3 = \beta v_2 + (1 - \beta) \theta_3$$

...

$v_0 := 0$   
 $v_0 := \beta v + (1 - \beta) \theta_1$   
 $v_0 := \beta v + (1 - \beta) \theta_2$   
 $\vdots$   


---

 $\rightarrow v_0 = 0$   
 Repeat {  
     Get next  $\theta_t$   
      $v_0 := \beta v_0 + (1 - \beta) \theta_t$   
 }

Andrew Ng

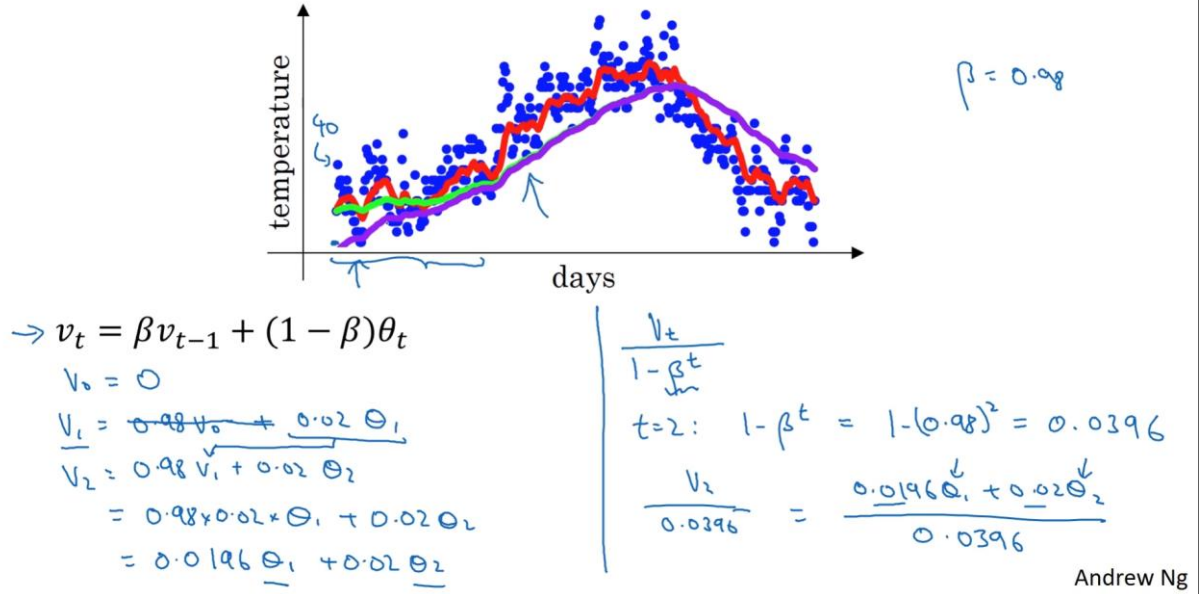
Dediğimiz gibi, direkt moving average uygulamak daha kesin sonuçlar vermesine rağmen bu uygulama da yakın sonuçlar verir ve computationally much more efficienttir.

# Vid 5

## Bias Correction of Exponentially Weighted Averages

EWA kavramını gördük anladık, bu konuda son bir teknik detay kaldı bu da "Bias Correction". Bu sayede computations daha accurate biçimde yapılabilir.

### Bias correction



Daha önce demiştik ki Beta=0.98 için elde edilen grafik yeşil olan smooth grafik, ancak aslında bias correction uygulamazsak elde edilen grafik mor grafik oluyor. Yani ilk başlarda 0'a yakın sonuçlar veren grafik.

Neden böyle? Çünkü  $v_0=0$  kabul ediyoruz yani ilk 9 gün sıcaklık 0 demişiz gibi oluyor, bu yüzden averaged graph 0'dan başlıyor. Yukarıda formülde de görüldüğü gibi  $v_1$ 'in formüşü  $0.02\theta_1$  yani bu da neredeyse 0 demek.

Benzer şekilde  $v_2$ 'nin formülü ise  $0.98v_1 + 0.02\theta_2$  yani  $0.0196\theta_1 + 0.02\theta_2$  oluyor ki bu da  $\theta_1$  ve  $\theta_2$ 'nin iyi bir estimate'ı değil. Bunun sebebi ilk 9 gün'ün sıcaklığının 0 gibi kabul edilmesi.

Bu problemi bias correction ile çözüyoruz, bunun için bir  $v_t$  hesaplanırken sağda görüldüğü gibi  $1 - \beta^t$  ile bölüyoruz yani,  $v_1$  hesaplanıyorsa  $0.02\theta_1 / (1 - 0.98)$  diyoruz ki bu da  $\theta_1$ 'e denk geliyor böylece, ilk 9 günün 0 olma etkisini ortadan kaldırıyoruz.

Şuna dikkat et: daha ileriki örneklerle gittikçe  $\beta^t$  giderk 0'a yaklaşıyor sonuçta eski formüle geri dönmüş oluyoruz, bu correction sadece baştaki 0 kabulünü gidermek için, bu yüzden yeşil ile mor grafik başlarda farklı olsa da daha sonradan oturuyor.



# Vid 6

## Gradient Descent with MOMENTUM

Bu algoritma almost always works faster than the standard gradient descent algorithm.

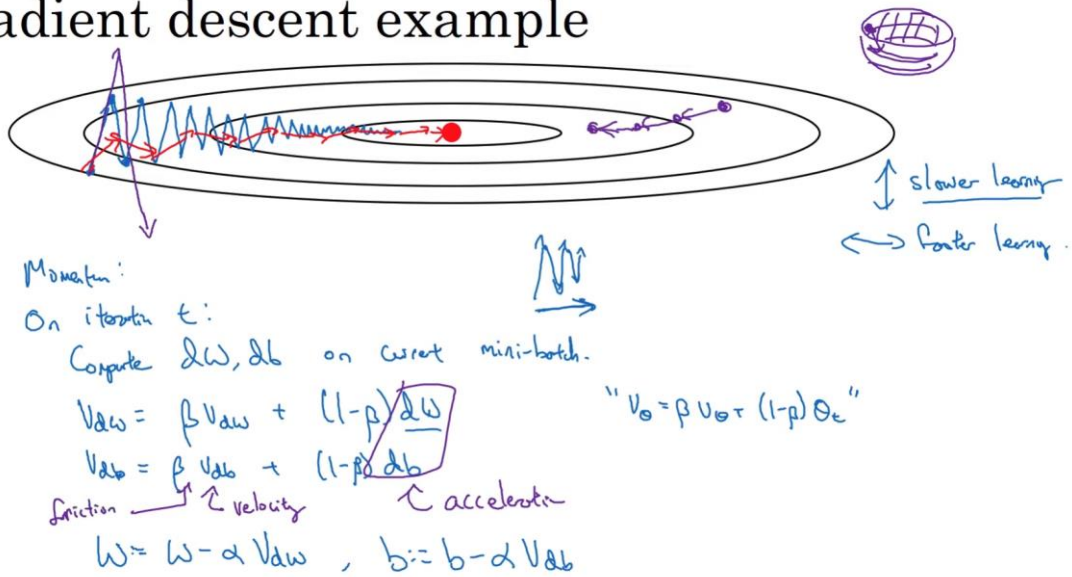
Temel fikir şu: Compute exponentially weighted average of the gradients and use that gradient to update the weights.

Diyelim ki, aşağıdaki gibi bir contour cost function üzerinde optimizasyon yapmaya çalışıyoruz, gradient descent kullanıyoruz ve mavi ile görüldüğü gibi zikzaklı adımlar atılıyor. Görüldüğü gibi gda takes a lot of steps and slowly oscillates towards the minimum.

Bu up and down osilasyonlar gda'yı yavaşlatıyor ve daha büyük bir learning rate kullanmamızı engelliyor, çünkü daha büyük learning rate kullanırsak divergence ile karşılaşmamız muhtemel. Yani osilasyonu engellemeliyiz ki daha büyük learning rateler ile hızlı bir öğrenme sağlayabilelim.

Bu problemi bir başka görme şekli şu: Vertical axiste slower learning isteriz çünkü osilasyondan kurtulmak istiyoruz, horizontal axist'te ise görece faster bir learning isteriz ki hızlıca minimum'a ulaşalım.

## Gradient descent example



Andrew Ng

Bu noktada MOMENTUM'u kullanabiliriz, her iterasyonda eskisi gibi  $dW, db$  değerleri current minibatch (batch size=m yani batch gd de kullanılabilir) ile hesaplanır.

Daha sonra yukarıdaki gibi  $V_{dw}$  ve  $V_{db}$  hesaplanır. Bunları ewa formülü ile hesaplıyoruz, x steplik ortalama gradient'i bulmuş oluyoruz.

Son olarak  $W$  ve  $b$  parametrelerimizi hesaplanan moving gradients ile update ediyoruz.

Momentum mavi steps yerine kırmızı steps ile sonuçlanıyor, az osilasyon hızlı convergence.

Peki nasıl oluyorda MOMENTUM daha smoother gda steps'e yol açıyor? Şöyle düşün diyelimki son 10 stepin ortalamasını alıyorum, steplere baktığımız zaman yukarı aşağı osile olan yukarıdaki mavi steplerin bir kesimini düşün.

Bu steplerin ortalaması alındığında vertical steplerin ortalaması aşağı yukarı 0 çıkar çünkü vertical componentlerin yarısı pozitif yarısı negatif çıkar. Benzer şekilde horizontal steplerin ise hepsi tek yönde sağa doğru olduğu için, ortalama da bu yönde çıkacaktır.

Bu sayede gd with MOMENTUM biraz ısındıktan sonra doğru ortalamaları alır ve küçük osilasyonlarda global minimum'a doğru emin adımlar atar.

Son olarak MOMENTUM'un implementation details'ine bakalım:

## Implementation details

$$v_{dw} = 0, v_{db} = 0$$

On iteration  $t$ :

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

$$\begin{aligned} \rightarrow v_{dw} &= \beta v_{dw} + (1-\beta) dW \\ \rightarrow v_{db} &= \beta v_{db} + (1-\beta) db \end{aligned} \quad \left| \quad v_{dw} = \beta v_{dw} + dW \leftarrow$$

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

$$\frac{v_{dw}}{\beta^t}$$

Hyperparameters:  $\alpha, \beta$

$$\beta = 0.9$$

average over last 10 gradients

Andrew Ng

2 adet hyperparametremiz var learning rate ve Beta, Beta genelde 0.9 olarak seçilir ve gayet robusttur, yani 10 step için average alınır.

Bunun yanında, bias correction pratikte yapılmaz çünkü algoritma 10 stepten sonra ısınır ve iyi sonuçlar vermeye başlar, bu yüzden bias correction'a gerek duyulmaz.

$v_{dw}$  ve  $v_{db}$   $dW$  yani  $W$  ile aynı dimensionda matrixler ve bunları zero matrix olarak initialize ettikten sonra her iterasyonda yukarıdaki adımları uygulayarak ve MOMENTUM'u uygulamış oluruz.

Burada literatürde genelde,  $1-\beta$  term'i kullanmazlar. Bu formda alphası ayarlayarak aynı işlemi yaptırmış olabiliyoruz, ancak Andrew bunu pek önermiyor çünkü  $\beta$ 'yı değiştirirsen  $\alpha$ 'yı da retune etmen gerekir diyor.

# Vid 7

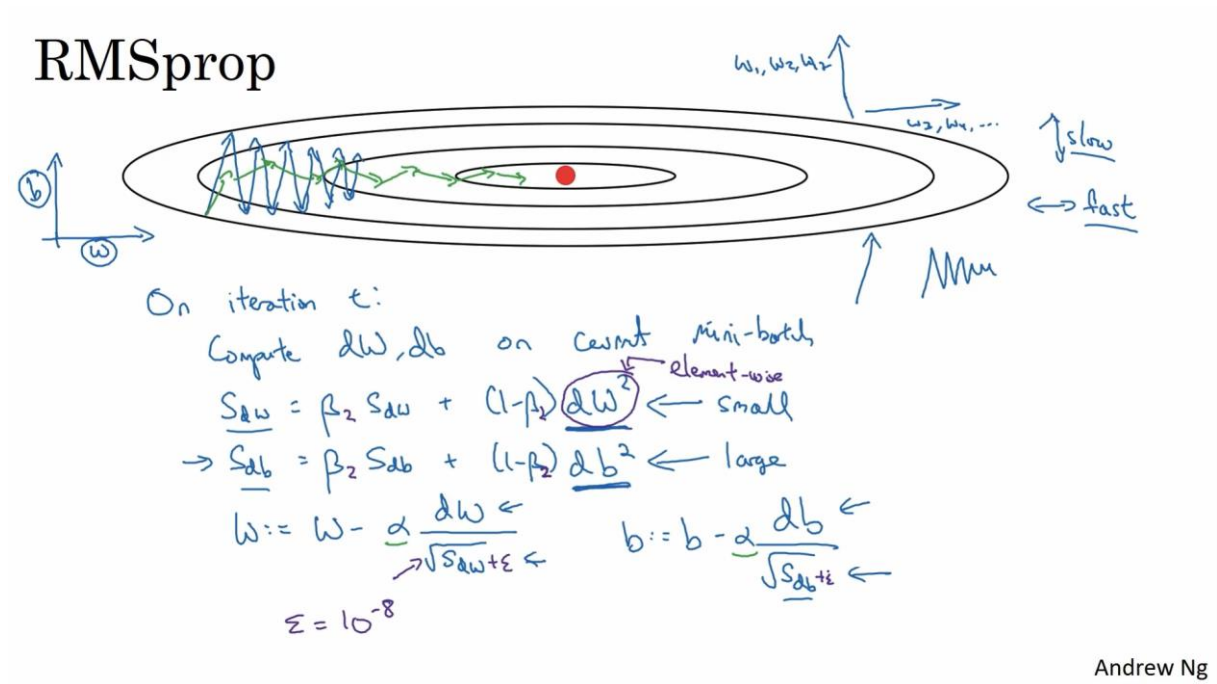
## RMSProp

MOMENTUM'un gd'yi nasıl hızlandırdığını gördük, "RootMeansSquaredProp" ile de gradient descent hızlandırılabilir, bu başlıkta bunu inceleyeceğiz:

Önceki örnekten hatırladığımız gibi diyelim ki gradient descent ile optimizasyon yapıyoruz, ve mavi ile görülen osilasyonlu path izleniyor.

Burada intuition için vertical axis'e b horizontal axis'e w diyelim.

Amacımız, b direction'da learning'i yavaşlatmak, w direction'da ise hızlandırmak veya sabit bırakmak.



Bu amaçla RMSProp algoritması her iterasyonda yukarıdaki adımları uygular.

Önce eskisi gibi  $dW, db$  current mininbatch veya batch ile hesaplanır.

Daha sonra momentum'a benzer şekilde yukarıdaki hesaplamalar yapılır, burada iki notasyonun karışmaması için farklı gösterimler kullanıldı ama,  $S_{dw}$  ve  $S_{db}$  hesaplamasında tek fark  $dW$  ya da  $db$  yerine karelerinin hesaba dahil olması.

Update adımında da yukarıdaki gibi  $S$ 'ler yerine başka bir form kullanılıyor, burada  $S_{dw}$ 'nin 0 olma ihtimalinde bug oluşturmamak için alta tarafa  $10^{-8}$  mertebelerinde bir epsilon eklenmiş.

Ben burada b ve w direction intuition'unu yanlış buluyorum, kafamı karıştırdı. Şöyle düşün, standard gd ile vertical adımlar daha büyük, horizontal adımlar daha küçük oluyor. Burada hem w hem de b içinde hem yatay hem dikey adımı etkileyen componentler olduğunu unutma. Diyelim ki dikey adımdan sorumlu olanlar  $dw_3, dw_7$  ve  $db_4$  olsun bu durumda  $S_{dw_3}, S_{dw_7}$  ve  $S_{db_4}$ 'ün büyük çıkmasını bekleriz bunu sağlayan da kare alma işlemi, küçüğün

karesi daha küçükken büyüğün karesi daha büyük çıkacaktır, bunlar büyük çıkınca update sırasında bunlardan küçük diğer componentlerden yani yatay componentlerden büyük adımlar atılacaktır.

Yani aslında burada yapılan şey şu,  $dW$  ve  $db$  içinde horizontal gradient step componentten sorumlu olan elemanları kare alma yöntemiyle su yüzüne çıkarmak ve bunları ileriki steplerde kısarak, daha az dikey ilerleme ve daha fazla yatay ilerleme sağlamak.

## Vid 8

### Adam Optimization Algorithm (Adaptive Moment Estimation)

ADAM'ın yaptığı şey, MOMENTUM ile RMSProp'u bir araya getirmekten fazlası değil.

#### 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

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

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

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

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

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

Andrew Ng

Burada, tipik olarak bias correction da uygulanıyor ve updates yukarıdaki gibi yapılıyor. Sonuçta hem MOMENTUM'un hem de RMSProp'un etkileri birleştirilmiş oluyor.

Burada bazı hyperparametreleri söz konusu.

#### Hyperparameters choice:

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

Adam : Adaptive moment estimation

Andrew Ng

Burada Beta1, Beta2 ve Epsilon genelde fixtir. Alpha'yı tune etmemiz gerekir.

# Vid 9

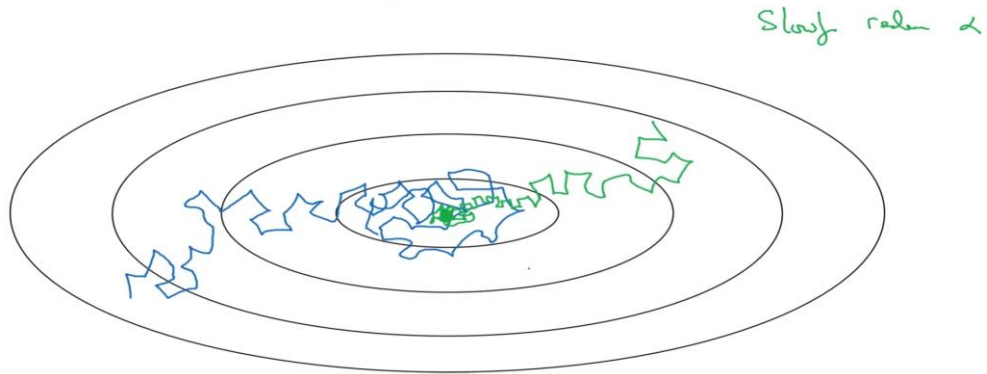
## Learning Rate Decay

Learning rate'ı zamanla azaltarak learning process'i hızlandırabiliriz. Bu kısımda learning rate decay kavramının nasıl implement edileceğinden bahsedeceğiz.

Diyelim ki 64 minibatch size ile bir minibatch gd çalıştırıyoruz, dolayısıyla gradients steps will be little bit noisy. Global minimum'a doğru yaklaşıp tam olarak minimum'a converge edemeyeceğiz, etrafında dolanacağız.

Ancak eğer learning rate'ı zamanla azaltırsam, global minimum'a yaklaştıkça stepler küçülecek ve giderek minimum'a daha yakın bir çevrede küçük adımlarla dolaşmaya başlayacak hale geleceğim.

## Learning rate decay



Andrew Ng

Learning rate decay'i nasıl implemente edeceğimize bakalım:

## Learning rate decay

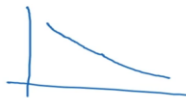
1 epoch = 1 pass through data.

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

Epoch	$\alpha$
1	0.1
2	0.67
3	0.5
4	0.4
$\vdots$	$\vdots$



$\alpha_0 = 0.2$   
 $\text{decay\_rate} = 1$



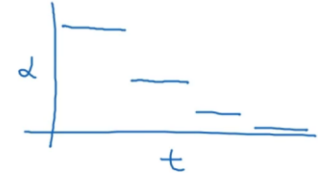
Andrew Ng

Burada yapılan yukarıdaki formülü uygulamak. Alpha0 ile decay\_rate'i tune edilir ve alpha formüle göre set edilirse, epoch'a bağlı olarak sürekli azalacaktır, böylece eğitim ilerledikçe daha küçük adımlar atılacak.

Bu formüle alternatif başka learning rate decay methodları da var:

## Other learning rate decay methods

formül

$$\alpha = 0.95^{\text{epoch-num}} \cdot \alpha_0 \quad - \text{exponentially decay.}$$
$$\alpha = \frac{k}{\sqrt{\text{epoch-num}}} \cdot \alpha_0 \quad \text{or} \quad \frac{k}{\sqrt{t}} \cdot \alpha_0$$


discrete staircase

Manual decay.

Andrew Ng

Formüllere ek olarak, eğitimi durdurup elle learning rate decay yapmak da mümkün.