[2-1. Optimization Algorithms]

<Mini-batch Gradient Descent>

배치 경사 하강법 : 전체 훈련 샘플에 대해 훈련 후 경사 하강 진행

미니 배치 경사 하강법 : 전체 훈련 샘플을 작은 훈련 세트인 미니 배치로 나눈 후, 미니배치 훈련 후 경사하강 진행

- 배치 경사 하강법의 경우, 큰 데이터세트에서는 훈련하는데 시간이 많이 들고 결과적으로 경사 하강을 진행하는데 오래걸리기 때문에 작은 훈련세트인 미니 배치로 나누어 훈련 후 경사하강을 적용

ex) m = 5,000,000 개의 훈련세트 일때 사이즈=1000개인 미니배치 5,000개로 나눠진행

$$X_{(n_x,m)} = [\underbrace{x^{(1)}\ x^{(2)}\ x^{(3)} \cdots x^{(1000)}}_{X^{\{1\}}\ (n_x,1000)} \mid \underbrace{x^{(1001)} \cdots x^{(2000)}}_{X^{\{2\}}\ (n_x,1000)} \mid \cdots \mid \underbrace{\cdots\ x^{(m)}}_{X^{\{5000\}}\ (n_x,1000)}]$$

$$Y_{(1,m)} = [\underbrace{y^{(1)}\ y^{(2)}\ y^{(3)} \cdots y^{(1000)}}_{Y^{\{1\}}\ (1,1000)} \mid \underbrace{y^{(1001)} \cdots y^{(2000)}}_{Y^{\{2\}}\ (1,1000)} \mid \cdots \mid \underbrace{\cdots\ y^{(m)}}_{Y^{\{5000\}}\ (1,1000)}]$$

* 표기법

- i번째 훈련 세트 : $x^{(i)}$
- l번째 신경망의 z값 : $z^{[l]}$
- t번째 미니배치 : $X^{\{t\}}$, $Y^{\{t\}}$

* Mini-batch gradient descent

for $t = 1, \cdots, 5000$

① Forward prop on $X^{\{t\}}$

$$Z^{[1]} = W^{[1]}X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\cdots$$
$$A^{[L]} = g^{[L]}(Z^{[L]})$$

} Vectorization (1000 examples)

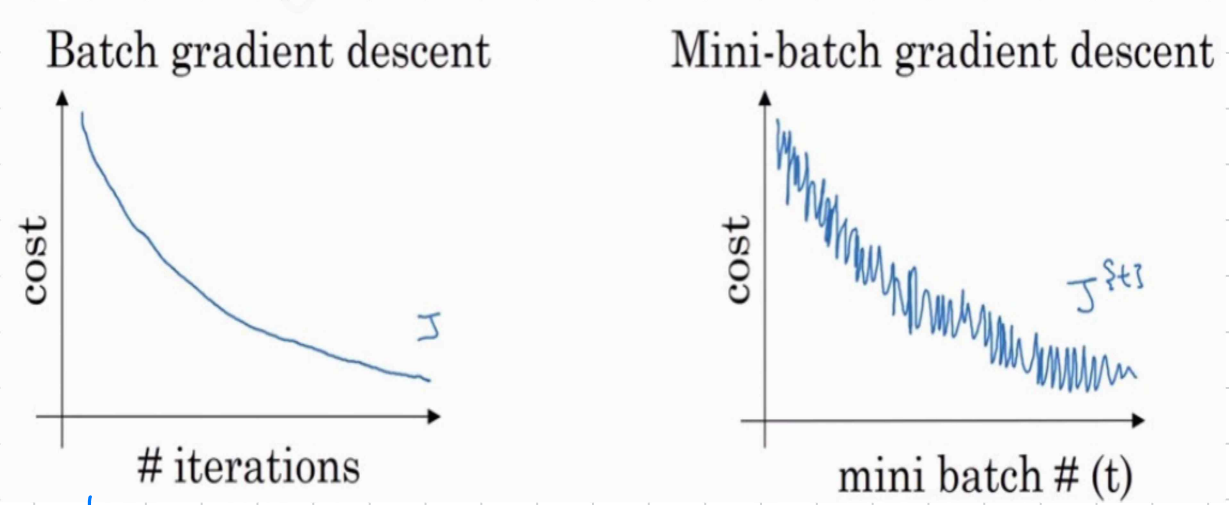
② Compute cost $J = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l}\|w^{[l]}\|_F^2$

③ Backprop to compute gradient $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)

$$w^{[l]} := w^{[l]} - \alpha dw^{[l]}, \quad b^{[l]} := b^{[l]} - \alpha db^{[l]}$$

⇒ "1 epoch" 라고 한다.

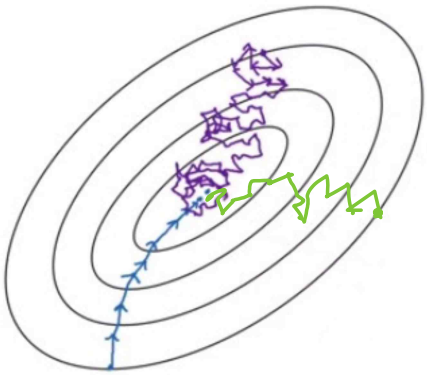
<Understanding Mini-batch Gradient Descent>



↳ 한번 반복을 돌때마다 비용함수↓

↳ 전체적으로 보면 감소하는 형태지만 많은 노이즈를 가짐

* Choosing your mini-batch size



If mini-batch size = m : Batch gradient desent

→ $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$

⇒ Iteration이 너무 오래 걸림

If mini-batch size = 1 : stochastic gradient descent

→ $(X^{\{1\}}, Y^{\{1\}}) = ((X^{(1)}, Y^{(1)}), \cdots (X^{(m)}, Y^{(m)}))$

⇒ 벡터화하는데에서 속도가 느려짐

In practice, somewhere in-between 1 and m

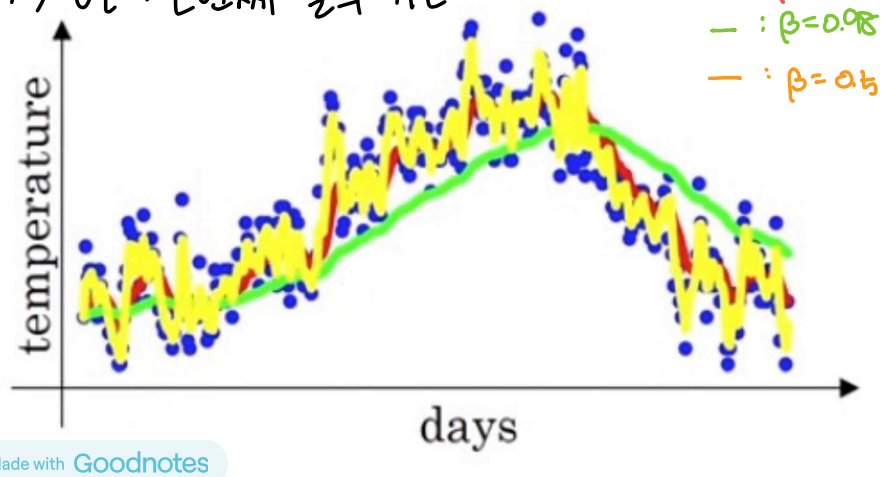
⇒ 가장 빠른 학습 (벡터화, 속도↑)

⇒ small training set ($m \leq 2000$) : batch gradient descent

typical mini-batch size : 64, 128, 256, 512와 같이 2^n으로 정함.

<Exponentially Weighted Average>

ex) θ_t : t번째 날의 기온



— : $\beta = 0.9$

— : $\beta = 0.98$

— : $\beta = 0.5$

V_t (exponentially weighted average)

: $\frac{1}{1-\beta}$ 기간 동안의 기온의 평균

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

✱✱ 많이 사용

i) $\beta = 0.9$ ≈ 10 day's temperature

ii) $\beta = 0.98$ ≈ 50 day's

iii) $\beta = 0.5$ ≈ 2 day's

<Understanding Exponentially Weighted Averages>

ex) $V_t = \beta V_{t-1} + (1-\beta)\theta_t$ 의 지수 가중 함수에서, $\beta = 0.9$라 하면

$V_{100} = 0.9\,V_{99} + 0.1\,\theta_{100}$

$V_{99} = 0.9\,V_{98} + 0.1\,\theta_{99}$

$V_{98} = 0.9\,V_{97} + 0.1\,\theta_{98}$

..

$\Rightarrow V_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9V_{98})$

$= 0.1\theta_{100} + 0.1\times0.9\,\theta_{99} + (0.9)^2 V_{98}$

$= \cdots$

$\Rightarrow V_{100} = 0.1\theta_{100} + 0.1\times0.9\,\theta_{99} + 0.1\times(0.9)^2\theta_{98} + \cdots$

(강조 그래프)

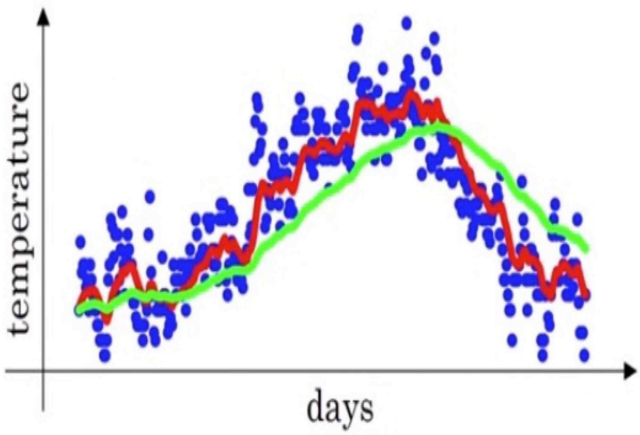
cf) $\beta = 1-\varepsilon$ 이라고 하면, $(1-\varepsilon)^n = \frac{1}{e}$ 을 만족하는 n이 n 기간동안 이동하면서 취하는 평균

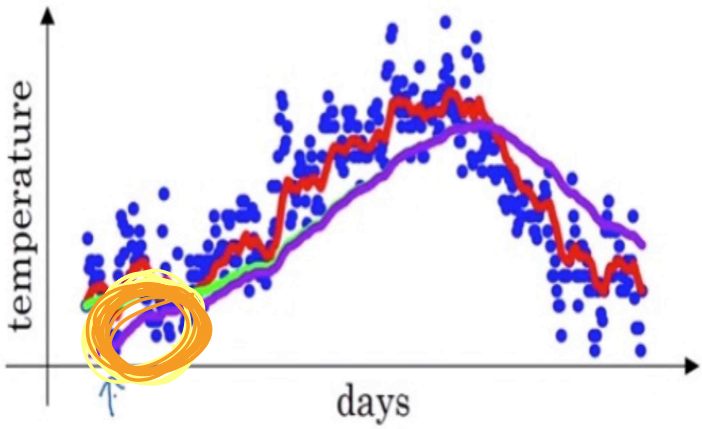
ex) $\beta = 0.9$ 면 $n = 10$, $\beta = 0.98$ 이면 $n = 50$

-exponentially weighted average의 장점 : 아주 적은 메모리를 사용!

(∵ V_θ 하나의 값만 가지고 있으면 됨)

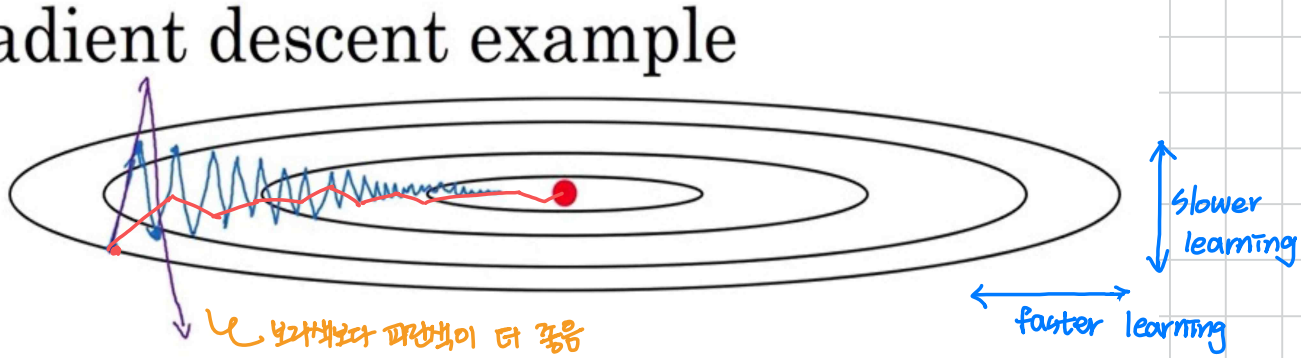
<Bias Correction in Exponentially Weighted Averages>





- 일반적인 지수 가중 평균을 사용하면, 초기값에서 차이가 많이난다.
- 이것을 $\frac{V_t}{1-\beta^t}$ 를 취해서 편향 보정을 사용할 수 있다.
- 그러나 머신러닝에서는 시간이 지나면서 초록색 그래프로 맞춰지기 때문에 많이 사용하지는 않는다.

<Gradient Descent with Momentum>



↳ 보라색보다 파란색이 더 좋음
(일반 경사하강법을 적용하면 세로축으로 진동이 커지는데,
중요한 것은 세로축 이동보다 가로축으로 이동이 빨라야 한다는 것이다)
→ momentum이 이것을 적용하는데에 도움을 준다

*Momentum

On iteration t:

Compute dw, db on current mini-batch

→ $V_{dw} = \beta V_{dw} + (1-\beta) dw$, $V_{db} = \beta V_{db} + (1-\beta) db$ → 논문에서 $(1-\beta)$를 생략한 표기도 자주 사용

$w := w - \alpha V_{dw}$, $b := b - \alpha V_{db}$

Hyperparameter : α, β 이며 $\beta = 0.9$를 주로 사용

- 오멘텀은 오멘텀이 없는 경사하강 알고리즘보다 더 잘 작동한다.
- 매 단계의 경사 하강 방향을 더 부드럽게 만들어준다 (위의 그림의 빨간색 gradient descent)
- 오멘텀은 이동평균을 적용한다고 생각하면 됨. 즉, 세로축(b)의 경우 위아래 변동폭이 작으므로 db가 작고 가로축(w)의 경우 이동폭이 크기 때문에 dw가 커진다.

<RMSprop (Root Mean Square prop)>

*RMSprop

On iteration t:

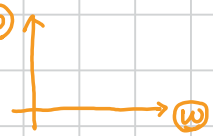
Compute dw, db on current mini-batch

→ $S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$ ↳ element-wise (요소별 제곱) 이며, 지수 평균을 유지

$S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$ > 일반적인 경사하강을 적용한다면 (세로축 ↑, 가로축 ↓ 이동) dw는 작고 db는 큰 값 인것임

$w := w - \alpha \frac{dw}{\sqrt{S_{dw}} + \varepsilon}$, $b := b - \alpha \frac{db}{\sqrt{S_{db}} + \varepsilon}$, $\varepsilon = 10^{-8}$

(ε: 작은값, '0'으로 나누지 않기 위해 추가, 큰값)



- RMSprop는 미분값이 큰 축(db)에서는 큰 값(S_{db})로 나누어주고,
미분값이 작은 축(dw)에서는 작은 값(S_{dw})로 나누어주어서,
각각 기존의 학습률보다 작은값, 큰 값으로 업데이트되기 때문에
진동을 줄이는데에 도움을 주고, 더 빠르게 수렴하는 효과를 기대

<Adam Optimization Algorithm>

* Adam (= Adapted Momentum estmation)

- 광범위한 딥러닝 알고리즘에서 잘 작동하여 'momentum + RMSprop' 방법

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

On iteration t :

Compute dw, db using current mini-batch

→ $V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw$, $V_{db} = \beta_1 V_{db} + (1-\beta_1) db$ ← "momentum"

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2$, $S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$ ← "RMS prop"

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

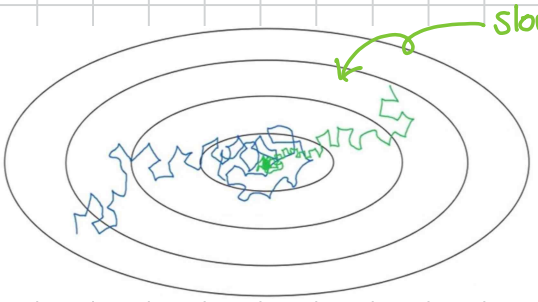
$$S_{dw}^{correct} = S_{dw} / (1-\beta_2^t) \quad , \quad S_{db}^{correct} = S_{db} / (1-\beta_2^t)$$

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

- Hyper parameters choice
  - α : needs to be tune
  - β_1 : 0.9 (= dw)
  - β_2 : 0.999 ($= dw^2$)
  - ε : 10^{-8}

<Learning Rate Decay>

- 학습률 감소: 시간이 지날수록 점점 학습률을 작게 줘서 최적값을 더 빨리 찾도록 함



⇒ 작은 미니배치일수록 noise가 심해지기 때문에, 학습률(α)가 동일하면 최적값에 수렴하기 힘들다.

∴ α를 점점 줄여가면서 최적값을 찾도록

- 1 epoch : 1 pass through all data

$$\alpha = \frac{1}{1 + \text{decay rate} \times \text{epoch num}} \alpha_0$$

ex) $\alpha_0 = 0.2$, decay rate = 1

epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
...	...

→ 점진적으로 감소

*Other learning rate decay methods

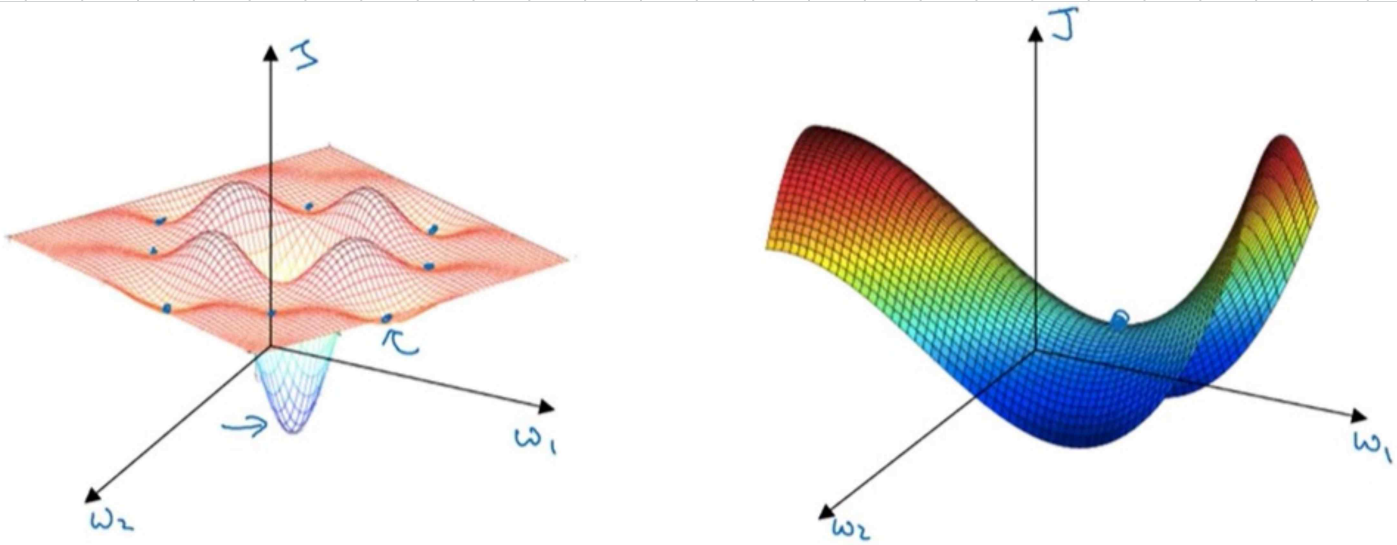
① $\alpha = 0.95^{\text{epoch num}}\,\alpha_0$: exponential decay

② $\alpha = \frac{k}{\sqrt{\text{epoch num}}}\,\alpha_0$

③ $\alpha = \frac{k}{\sqrt{\text{batch num}}}\,\alpha_0$

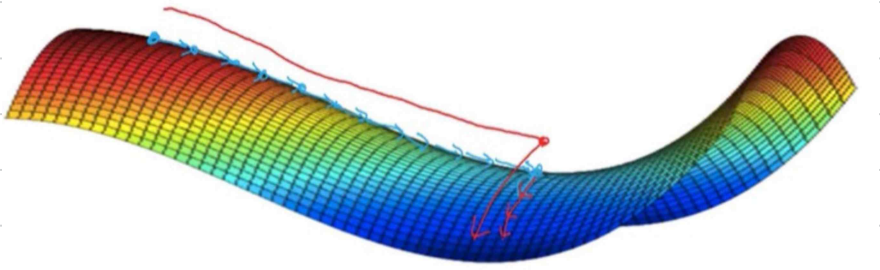
<The Problem of Local Optima>

*



왼쪽 사진에서처럼 local optima problem이 생길 수 있지만,
신경망에서 매우 많은 hyper parameter를 다루고 있으므로 이러한 문제에 빠질일은 거의 X
대부분의 경우에 오른쪽처럼 global optima로 회귀한다.

*Problem of plateaus



plateau : 경사가 완만해지면서 gradient descent를 하는 속도가 굉장히 느려지는 것