[Week10]_이원주

최적화 알고리즘

- 신경망을 더 빠르게 학습하도록 하는 알고리즘
- 딥러닝에서는 학습을 여러 번 시도해야 함. 즉, 학습을 빠르게 한다 = 한 번 시도하는 데 걸리는 시간이 줄어든다 = 매우 중요!

미니 배치 경사 하강법

(Mini Batch Gradient Descent)

- **필요성**
  - 벡터화 → 명시적 반복문 없이 m개의 데이터를 계산. 그것도 병렬적으로 빠르게!
  - 그런데 만약 m이 매우 크다면 → 벡터화해도 여전히 느림. m개 데이터를 모두 처리하기 전까지 다음 경사하강법을 시작할 수 없기 때문.
  - 그렇다면 m을 작게 만들면 되겠다. Train set을 한 덩어리로 쓰지 말고,작게 쪼개서 쓰면 어때?
  - 이렇게 작게 쪼갠 Train set : 미니 배치(mini batch)

  ▼ ex) + 표기법

  m = 5,000,000인 Train set이 있을 때

  mini batch size = 1,000라면

  - t `= mini batch 개수` = 5,000
    - X → $X^{\{1\}}$ ~ $X^{\{5000\}}$ 쪼갬.

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

- $Y \rightarrow Y^{\{1\}} \sim Y^{\{5000\}}$ 쪼갬.

▼ 코드 *Aha!*

//모든 미니배치에 대해서

for t = 1 ~ 5000 *//(이거 나중에 벡터화해야지)*

//정전파

$Z^{[1]} = w^{[1]}X^{\{t\}} + b^{[1]}$

$A^{[1]} = \sigma(Z^{[1]})$

…

$Z^{[l]} = \sim$

$A^{[l]} = \sim$

//역전파

(공식 똑같음)

//가중치 업뎃

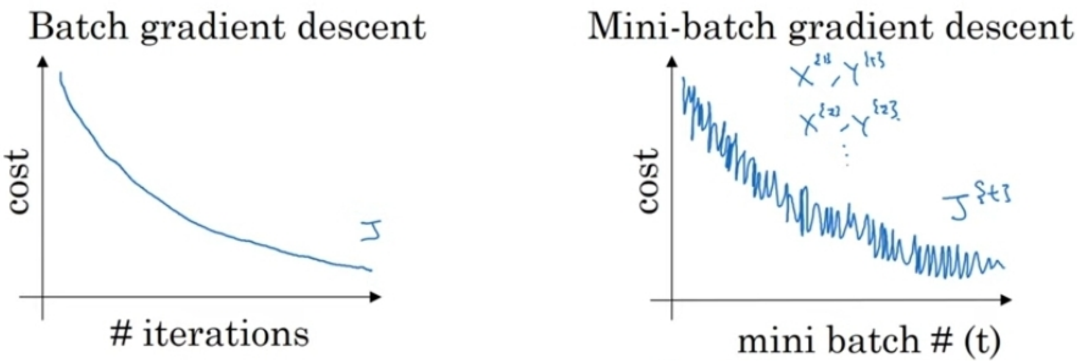
(공식 똑같음)

- **기존 방식과의 차이**
  - 정의

배치 경사하강법	Train set을 한 덩어리로 씀. (기존 방법)
미니 배치 경사하강법	Train set을 미니 배치로 작게 쪼개서 씀.

  - 비용함수 J의 추이

배치 경사하강법	항상 감소. 만약 하나라도 증가한다면 문제가 있는 것.
미니 배치 경사하강법	전체적으로 보면 감소하지만, 노이즈가 있음.



   - 노이즈가 있는 이유 : 우연히 $X^{\{1\}}$ 는 쉬웠는데, $X^{\{2\}}$는 어려우면 J가 증가할 수도 있잖아.

- **미니 배치 사이즈**
  - 얘도 하이퍼 파라미터. 어떻게 정하지?
  - ▼ 직관적 이해
    - size = 1인 경우 → **확률적 경사하강법**
      - 각 미니 배치 = 훈련 데이터 하나씩
      - 수렴하지 않음. J가 전체적으로 감소하는 추세이긴 하지만 노이즈가 너무 큼. → 근데 이건 학습률 작게 해서 해결 가능.
      - 진짜 단점은 벡터화함으로서 얻는 이점이 사라진다는 것.
    - size = m인 경우 → (=전체 데이터 개수)
      - 미니 배치 (1개) = 훈련 데이터 한 덩어리
      - 즉, 그냥 배치 경사하강법과 동일.
      - 한 반복당 훈련 시간이 오래 걸림.
  - 방법
    1. Train set 크기가 애초에 작다면 (m≤2,000) → 그냥 미니 배치 사용 X
    2. 크다면 → 일반적으로 64~512인 2^n 중에서 선택 + 모든 $X^{\{t\}}$, $Y^{\{t\}}$가 내 CPU/GPU에 맞는지 확인.

지수 가중 이동 평균

(Exponentially Weighted Average)

- 배우는 이유
    - 경사하강법보다 빠른 최적화 알고리즘이 몇 개 있음. 그걸 이해하려면 이 배경지식이 필요함.

- 개념
    - 최근의 데이터에 더 많은 영향을 받는 데이터들의 평균 흐름을 계산하기 위해 구함. (ex 기온)
    - 현재 데이터보다 이전 데이터에 더 높은 가중치를 줌.
    - $v_t = \beta \times v_{t-1} + (1-\beta) \times \theta_t$
        - θ_t : t번째 날의 기온
        - v_t : 지수 가중 이동 평균
        - 이때 β는 하이퍼 파라미터. 보통 0.9를 사용.
            - 커지면 → 꽤 이전 데이터까지 반영. 현재 데이터가 확 변해도 v_t의 변동폭은 작고, 곡선이 완만함. but 그만큼 실제 데이터와 오차도 크고, 실제 데이터의 변화를 더 늦게 반영함.
- 구현 코드

```
v = 0
for t in ~
    v = β * v + (1-β)* θ_t
```

    - 평균을 구하는 다른 방법에 비해서 메모리 절약 가능. (왜냐면 v 값은 하나만 저장하면서 계속 덮어쓰기 때문)

      → 이래서 머신러닝에서 이 방법을 사용하는 것.

편향 보정 (Bias Correction)

- 해결할 문제

- $v = 0$으로 초기화했으므로 $v_1 = 0 + (1-\beta) \times \theta_1$로 계산됨.
- 이렇게 추정의 초기 단계일수록 (t가 작을수록) 오차 발생!

- 해결방법

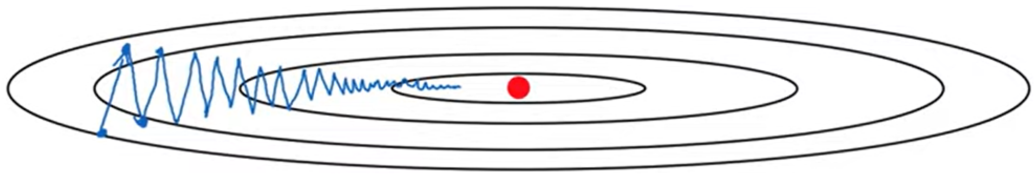
$$v_t^{corrected} = \frac{v_t}{1-\beta^t}$$

  - v_t 구한 다음 $1-\beta^t$로 나눠줌.
    - 그러면 t가 작을 때는,
      예컨데
      $t = 1$일 때는 분모 = 0.1
      그래서
      v_t를 0.1로 나눈, 더 큰 값이 됨.
    - t가 커질수록 β^t는 작아짐 → 분모 점점 1에 가까워짐.
  - 따라서 이렇게 초기 단계의 편향을 보정해서 → 평균을 더 정확하게 계산할 수 있음.
- 근데 사람들 잘 안 쓰긴 함.
  그냥 편향 있는 부분 좀 지나고 난 값부터 (=10번 반복 후부터) 사용한다고.

Momentum 최적화 알고리즘

- 개념
  - **모멘텀 알고리즘** = **모멘텀이 있는 경사하강법**
  - 가중치 업뎃 시 → 그냥 경사(dw, db) 대신 경사에 대한 지수가중평균(V_{dw}, V_{db})을 이용
    - 이때 V_{dw}, V_{db}의 차원 = dw, db의 차원

- 효과
  - 더 빠르게 학습하려면 여기서



수직 방향	진동이 줄어야 함.
수평 방향 (↔)	더 빠르게 나아가야 함.

- 지수가중평균을 이용하면 → 현재 데이터뿐 아니라 이전 데이터도 같이 고려하므로

수직 방향	위아래로 와리가리 → 플마 0 되어서 진동이 줄어듦.
수평 방향 (↔)	한 방향으로만 → 2배로 더 빠르게 나아감.

▼ 참고

논문 보다 보면 종종

$v_t = \beta \times v_{t-1} + (1-\beta) \times \theta_t$ 에서 $(1-\beta)$만 지우고

$v_t = \beta \times v_{t-1} +$ θ_t 이렇게 쓰는 경우가 있음.

딱히 뭐가 달라지진 않는데, 딱 하나. 학습률만 달라짐.

왜냐면 $(1-\beta)$만 지웠다는 건, v_t를 $(1-\beta)$로 나눈 거랑 비슷하잖아.

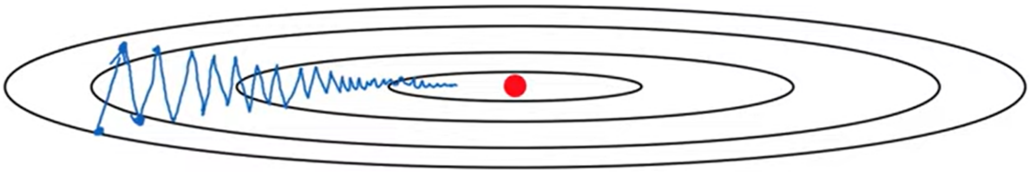
w -= $\alpha * V_{dw}$ 니까 v_t가 $\div(1-\beta)$됐으면 α는 $\times(1-\beta)$되어야지. 이것만 달라짐.

RMSProp 최적화 알고리즘

- 개념
  - 모멘텀 알고리즘이랑 비슷한데,
  - 지수가중평균(V_{dw}, V_{db})를 계산할 때 → 현재 데이터 θ_t를 그대로 쓰는 대신 $(\theta_t)^2$를 사용.
    - 그리고 표기법을 (S_{dw}, S_{db})로 변경.
  - 가중치 업뎃할 때 →
    - w -= $\alpha * V_{dw}$ 대신

        - w -= $\alpha * \frac{dw}{\sqrt{S_{dw}}}$ 로 업뎃.

- 효과
    - 더 빠르게 학습하려면 여기서



수직 방향	진동이 줄어야 함.
수평 방향 (↔)	더 빠르게 나아가야 함.

    - w -= $\alpha * \frac{dw}{\sqrt{S_{dw}}}$ 여기서 분모 $\sqrt{S_{dw}}$를 얘기해보자.

수직 방향	S_{dw} 큼. → $\frac{dw}{\sqrt{S_{dw}}}$ 작아짐. → 진동이 줄어듦.
수평 방향 (↔)	S_{dw} 작음. → $\frac{dw}{\sqrt{S_{dw}}}$ 커짐. → 더 빠르게 나아감.

▼ 참고

RMSProp 의 장점은 미분값이 큰 곳에서는 업데이트 시 큰 값으로 나눠주기 때문에 기존 학습률 보다 작은 값으로 업데이트 됩니다. 따라서 진동을 줄이는데 도움이 됩니다. 반면 미분값이 작은 곳에서는 업데이트시 작은 값으로 나눠 주기 때문에 기존 학습률 보다 큰 값으로 업데이트 됩니다. 이는 더 빠르게 수렴하는 효과를 불러옵니다.

- 주의점
    - w -= $\alpha * \frac{dw}{\sqrt{S_{dw}}}$ 여기서 분모 $\sqrt{S_{dw}} \neq 0$ 이어야 함.

Adam 최적화 알고리즘

- 개념
    - **Adam = Adaptive moment estimation**

- **Momentum + RMSProp 합친 것**
- 여기서는 편향 보정을 함.

$$w = w - \alpha * \frac{V_{dw}^{corrected}}{\sqrt{S_{dw}^{corrected}} + \varepsilon}$$

- 하이퍼 파라미터 추천 값
  - 학습률 α ← 잘 정하기!
  - (Momentum) β_1 = 0.9
  - (RMSProp) β_2 = 0.999
  - ε = 10^{-8} ← 근데 얘 그렇게 상관X

학습률 감쇠

(Learning Rate Decay)

> ⭐ **[요약]**
>
> - 학습률 감쇠 → 우선순위 낮은 방법

- 개념
  - 시간이 지남에 따라 학습률 줄임.
- 방법
  - 방법1

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

  - 방법2 : exponential decay

$$\alpha = 0.95^{epoch\_num}\alpha_0$$

- 방법3

$$\alpha = \frac{k}{\sqrt{epoch\_num}}\alpha 0$$

- 방법4

$$\alpha = \frac{k}{\sqrt{batch\_num}}\alpha 0$$

- 방법5

  → step 별로 α를 직접 다르게 설정