딥러닝 홀로서기#17

주제 : Advanced Optimizer than SGD

링크 :

[#17.Lec] Advanced Optimizer than SGD - 딥러닝 홀로서기
발표자료 링크 : https://github.com/heartcored98/Standalone-DeepLearning/blob/master/Lec4/Lec4-B.pdf자료 저장소 링크 : https://github.com/heartcored98/Standalone-DeepLearn...
https://youtu.be/a5R4gL1ObP8

1. Review from Last Lecture

- Overfitting : L2 Regularization, Dropout
  - Training Set으로 학습을 시, Model Capacity가 증가하면 Training Set을 외워버리고 Validation set, Test Set에 대해서는 정확도가 현저히 낮아지는 현상
  - Overfitting을 해결하는 Technic
    - L2 Regularization
      - Regression Problem일 경우 Loss = MSE라 정의하는데, 여기에 파라메터들이 너무 커지지 않게하는 새로운 항을 추가하는 방식
    - Dropout
      - MLP Layer의 일부 노드들을 임의적으로 P의 확률로 꺼버려서 Model Capacity를 줄이고, 하나의 복잡한 Model을 Training하기 보다, 여러개의 단순한 모델들을 합쳐서 Overfitting을 줄이는 방식
  - Gradient Vanishing : ReLU Activation
    - Sigmoid함수와 같이 너무 작은 값 또는 너무 큰 값일 때 Gradient가 0이 되어버리는 문제
      - Layer가 깊어질 수록, 이 성질로 인하여 Gradient가 0이 되는 부분을 거치면 그 뒤로는 모두 0으로 수렴하여 Gradient가 더 깊은 곳까지 전달되지 못함

        - 복잡한 문제를 해결하기 위해서는 Gradient가 끝까지 전달이 되어야 하는데, 그렇지 못하고 얕은 곳이 훈련되지 못함
    - ReLU : 0이하에서는 0, 0이상일때는 Y=X의 성질을 가진 함수로, Gradient Vanishing문제를 해결
- Xavier Initialization
    - MLP를 잘 훈련시키고 효율적으로 사용하기 위해서 Input X값을 초기화할 때 너무 크거나 너무 작은값으로 초기화하지 않도록 만드는 방식
- Batch Normalizatioon
    - 데이터가 편향되어 있으면 Activation Function을 적용하여도 똑같은 값으로 반환되기 때문에 Activation Function을 사용하지 않는 것과 같음. 이는 아무리 Layer를 깊게 쌓아도 행렬 곱 연산에 의하여 결국 하나의 행렬만 계속하여 만드는 효과와 같기에, Non-Linear Activation Function이 적용되지 않는 것과 같다
    - 따라서 적절히 Input X값을 Normalization을 하여 Activation Function이 잘 작동하도록 만들어주는 것을 Batch Normalization이라 함

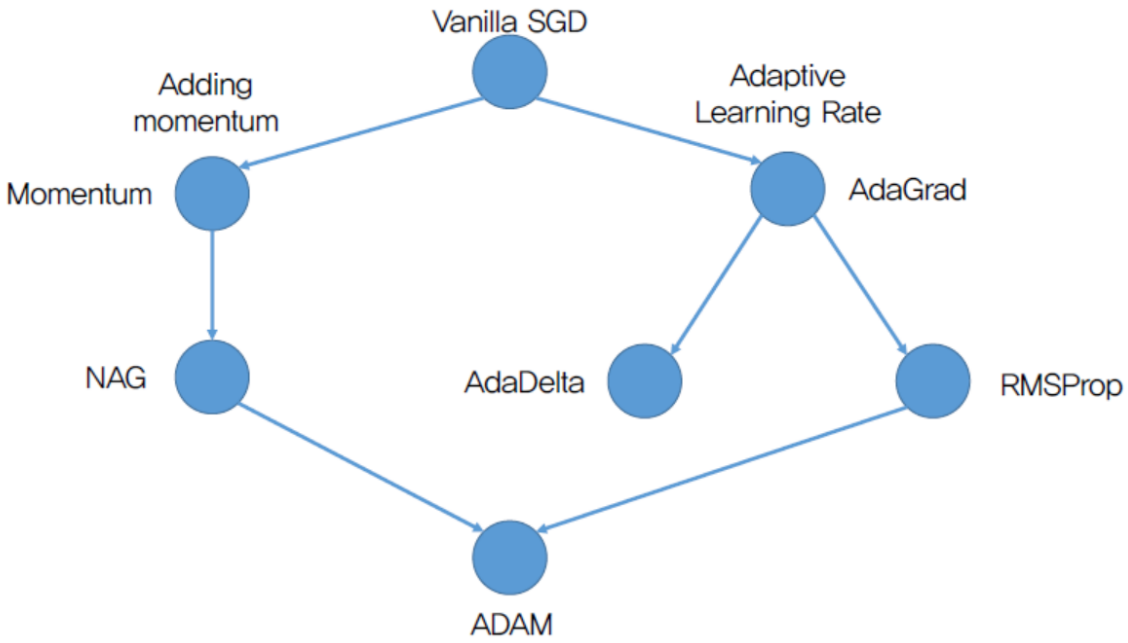
2. Gradient Descent

- 정의 : Cost Function을 $J(\theta) = wx + b$라 했을 시, 우리의 목적은 Minimum Cost를 도출하기 위한 $J(\theta)$를 찾는 것이며, 다시말해 최적의 파라메터 θ(w와 b)를 찾는 과정과 동일함. 이 함수를 찾을 때까지 두개의 파라미터를 업데이트하여 $J(\theta)$를 계속하여 줄여가야 하는데, 이때 사용하는 방식이 Gradient Descent 방식임
    - Parameter θ를 업데이트 하는 방식
        - Parameter θ = Parameter θ - Loss Function의 Gradient (=θ에 대한 Gradient) x 일정한 상수 (Learning Rate)
        - 위의 식을 계속하여 반복하다보면 어떤 Minimum값으로 수렴

$$\theta = \theta - \eta \nabla J(\theta)$$

θ : Parameter set of the model η : Learning rate $J(\theta)$: Loss function

- 종류



① Batch Gradient Descent

- 한번 Step을 밟을 때 전체 Training Dataset에 대해서 Gradient를 계산하고 Step을 밟는 방식
- 문제점
  - Image와 같이 Data수가 너무 많을 경우, Memory의 한계로 인하여 로딩하는데 한계가 있을 수 있음
  - 한번에 모든 Data를 계산해야하기에, 학습에 걸리는 시간이 오래걸림

② Stochastic Gradient Descent (SGD)

- 개념
  - 전체 Training Dataset을 여러개의 Mini Batch로 나누어, 각각의 Mini Batch마다 Gradient를 계산하며 Step을 밟아가는 방식
  - Mini Batch를 어떻게 선택하느냐에 따라(확률적) 그 결과값이 다르게 나오기 때문에 Stochastic이라 부름
  - Mini Batch로 계산하는 것이 더 빠른 이유는 GPU의 병렬 계산을 활용할 수 있기 때문
    - Small Chunk : 작은 데이터 덩어리
    - Deterministic : 결과값이 하나로 결정되는 것
    - Stochastic : 결과값이 확률로 결정되는 것

- Local Minimum : Loss Surface내의(Parameters로 이루어진 각 축과 종속변수 Loss에 대한 영역) Global Minimum이 아닌 다른 Minimum으로 하나의 축에 대해서는 Minimum이지만, 다른 축에 대해서는 Minimum이 아닌 지점을 뜻함
    - Batch를 어떻게 선택하느냐에 따라서 Global Minimum이 아닌 Local Minimum에 빠질 수 있기 때문에 Gradient Descent방식의 선택은 굉장히 중요
- 문제점
    - SGD는 속도가 Batch Gradient보다는 빠르지만, 전체적으로는 느린편에 속함
    - Local Minimum에 빠지면 벗어날 수 없음 (Gradient = 0인 지점)
    - Momentum을 더하거나(Momentum, NAG), 파라메터마다 Learning Rate를 달리해서(AdaGrad, AdaDelta, RMSProp) 해결가능

③ Momentum

- 개념
    - SGD의 Local Minimum에 갇히는 문제를 해결하기 위해 Momentum개념을 도입
        - 언덕에서 공을 굴렸을 때, 관성에 의하여 Local Minimum을 빠져나가는 원리
- 수식 해석
    - θ에서 v_t를 감소시켜 줌
    - v_t : 이전 값과 decay constant를 곱한 후 Gradient값을($\eta\nabla J(\theta)$) 더해줌
    - γ : Decay Constant라 부르며, gradient를 더함으로써 v_t가 계속해서 커지는 것을 방지하는 역할

$$\theta = \theta - v_t$$

$$v_t = \gamma v_{t-1} + \eta\nabla_\theta J(\theta)$$

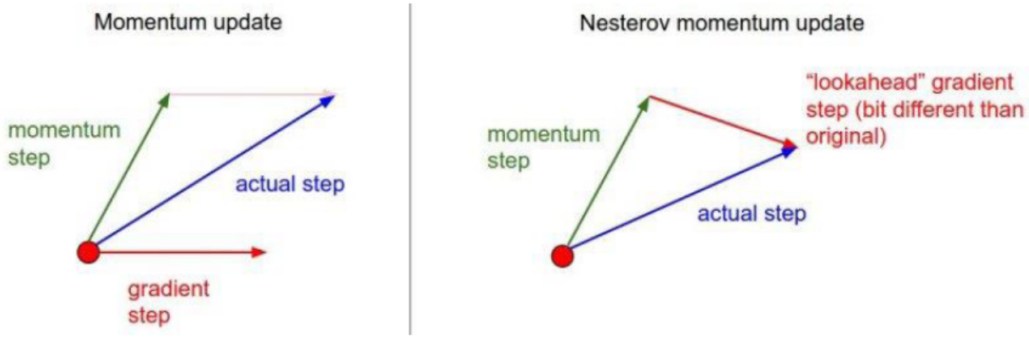
- 알고리즘
    - 특정 시점에서의 미분 값만 고려하던 Gradient Descent 방식과 달리, 직전뿐만 아니라 이전 step의 모든 Gradient 값들을 모두 포함하므로써 '관성

(Momentum)'을 구현

- 장점
  - Oscillation Problem : Gradient Descent방식은 불필요한 방향으로 쉽게 변하여 시간이 오래걸리는 단점이 있는데, Momentum을 활용하면 방향이 잘 변하지 않는 방향으로 가중치의 갱신이 일정하게 가속하게 되며, 이로 인하여 더 빠르게 최적 값을 향해 다가갈 수 있음
- 문제점
  - Global Minimum에 도착하여도, 관성으로 인하여 계속 움직이는 성질이 있음

④ Nesterov Accelerated Gradient (NAG)

- 개념
  - Momentum방식의 단점인 Global Minimum에서의 움직임을 해결하기 위한 알고리즘
  - Momentum에서는 지금까지의 Momentum Step과 현 지점에서 고려한 Gradient Step의 합산 지점을 Actual Step으로 이동
  - NAG방식은 Momentum Step으로 일단 이동 후, 그 지점에서의 Gradient Step을 계산하여 Actual Step으로 이동



- 수식해석
  - $\eta \nabla_\theta J(\theta - \gamma v_{t-1})$: 원래의 자리(θ)에서 Momentum(v_{t-1})을 고려하여 Gradient를 계산
  - θ, γ의 의미와 계산방식은 기존 Momentum과 동일

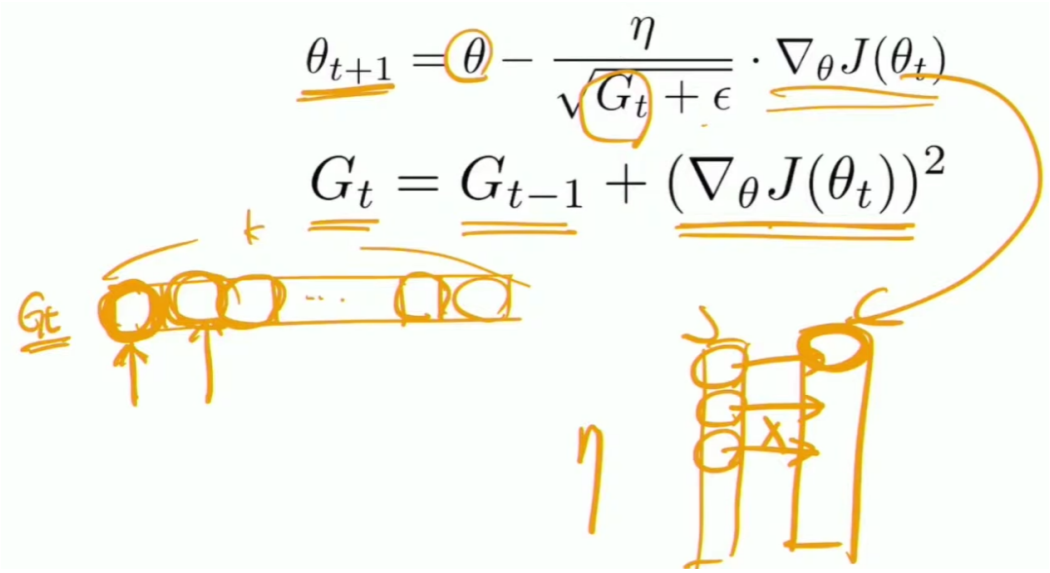
$$\theta = \theta - v_t$$

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

- 장점
  - Global Minimum에 가까워지면, 일단 Momentum으로 이동한 후, 그 자리에서 Gradient를 측정하면 굉장히 작은 값이기에 Global Minimum에 빨리 도착할 수 있음
- 문의 : 이 과정이 Global Minimum을 찾는 과정인가?
  - 그렇지는 않다. 실제 이 지역이 Global Minimum인지를 확인하려면 모든 Loss Surface에 대한 정보가 있어야 하나, 그건 실제로 불가능하므로, 너무 얕은 Local Minimum에 빠지지 않는 정도로만 해결
  - 따라서 다양한 Initial Point로 변경시켜가며 Loss Surface상에서 Grid Search를 하여 Global Minimum을 찾는 과정을 수행

⑤ Adaptive Gradient (Adagrad)

- Step Size가 파라메터마다 모두 동일한 문제를 해결하고자 Adaptive Learning Rate를 적용한 방식
- 수식해석
  - θ_{t+1} : 지금까지 변해온 만큼 반비례하는 Learning Rate값을 각각의 파라메터에 곱해준다는 개념 ((작게, 크게) 변한 파라메터 θ는 Learning Rate가 (크게, 작게) 곱해져 Step을 (크게, 작게) 이동)
  - G_t : K개의 파라메터 Set안에 Gradient의 제곱값들을 더한 행렬
  - ϵ : 0으로 나누지 않기 위해서 더해진 작은 값
  - ● : 행렬곱이 아닌, Element끼리의 곱셈 연산
  - $\frac{\eta}{\sqrt{G_t + \epsilon}}$: 각각의 다른 Learning Rate값을 지금까지 변해온 G_t만큼과 반비례하는 값으로 곱해준 1xK짜리 벡터
  - $\nabla_\theta J(\theta_t)$: 어떤 Loss값을 K개의 파라메터에 대해서 편미분한 값들을 쌓아놓은 1 X K개의 벡터



- 문제점
  - G가 무한대로 발산하게되어, 학습이 길어질 수록, Step Size가($\frac{\eta}{\sqrt{G_t+\epsilon}}$) 작아져서 Training을 오래 해도 학습이 제대로 되지 않은 문제발생

⑥ RMSProp

- Adagrad의 문제를 해결하기 위해서 Decay Constant값을 곱하여 Training을 오래해도 적절한 Step을 밟을 수 있게하는 방식
- 수식해석
  - 기존의 Adagrad에서 Decay Constant값을 G에 곱해준 형태
  - γ : Decay Constant로 G가 무한대로 발산하는 것을 방지하기 위한 상수

$$\theta_{t+1} = \theta - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla_\theta J(\theta_t)$$

$$G_t = \gamma G_{t-1} + (1-\gamma)(\nabla_\theta J(\theta_t))^2$$

⑦ AdaDelta

- θ_{t+1} : θ_t에서 $\Delta\theta$을 빼줌
- $\Delta\theta$: $\frac{1}{\sqrt{G+\epsilon}} * \nabla_\theta J(\theta_t)$부분은 기존과 동일하지만 $\sqrt{s+\epsilon}$부분이 달라짐
- G_{t+1} : 기존과 동일

- S_{t+1} : G랑 비슷하지만 G는 Gradient 제곱값을 더해주고, S는 파라메터의 변화량을(=$\Delta\theta$) 더해준다는 차이가 있음

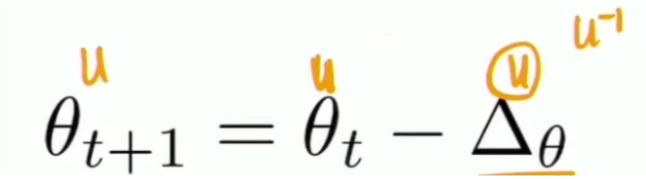
$$\theta_{t+1} = \theta_t - \Delta_\theta$$

$$\Delta_\theta = \frac{\sqrt{s+\epsilon}}{\sqrt{G+\epsilon}} \cdot \nabla_\theta J(\theta_t)$$

$$s_{t+1} = \gamma s_t + (1-\gamma)\Delta_\theta$$

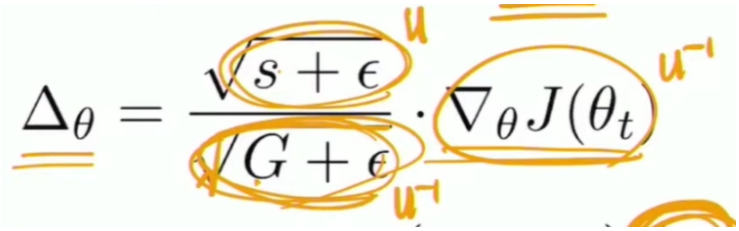
$$G_{t+1} = \gamma G_t + (1-\gamma)(\nabla_\theta J(\theta_t))^2$$

- AdaDelta에서 S값을 고려하는 이유
  - (1st line) θ가 U라는 어떤 단위(Unit)를 가지고 있다고 하면, U의 단위를 가진 두 값의 차($\theta_t - \Delta\theta$)로 표현될 수 있다
  - 그럼 기존의 식대로 $\Delta\theta = \theta - \eta\nabla J(\theta)$으로 계산 한다면, (Loss는 Unit이 없다고 가정) $J = \frac{\partial J}{\partial \theta}$로 표현할 수 있고 이는 U^{-1}의 단위로 표현된다
  - 이는 θ_{t+1}값을 계산시 서로 단위가 맞지 않는 문제를 발생하기 때문에 (θ_t : $U, \Delta\theta : U^{-1}$) 단위를 맞춰서 계산해야 한다



  - (2nd line) $\Delta\theta$ 를 계산시 $\nabla_\theta J(\theta_t)$는$U^{-1}$의 단위이고 $\sqrt{G+\epsilon}$ 부분도 U^{-1}의 단위이며, $\sqrt{S+\epsilon}$은 파라미터의 변화량인 $\Delta\theta$에 Root를 씌워준 것이므로 U의 단위를 갖게 되어 결국 $\theta_{t+1} = \theta_t - \Delta\theta$의 셋 항이 모두 같은 Unit을 갖게 되어 정확한 값이 계산됨
    - $\sqrt{G+\epsilon}$의단위가U^{-1}인이유 : (4th line)
      G_{t+1}이라는값은G_t에$\nabla_\theta J(\theta_t)^2$을 더해준 값인데, $\nabla_\theta J(\theta_t)$는$U^{-1}$

의 단위를 가지고 있으며, 이걸 제곱하여 다시금 Root를 씌워준 것이므로 U^{-1}의 단위를 갖는다고 설명할 수 있음



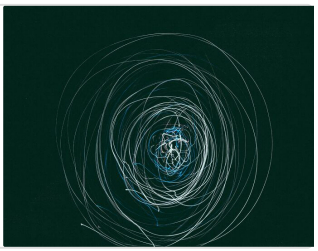
- 특징
  - 기존에는 Learning Rate 값인 임의의 상수 λ를 θ계산시 곱해줬는데, AdaDelta는 Learning Rate를 곱해주지 않아도 잘 돌아감
  - 어떤 Task에서는 Adagrad보다 더 잘될때도 있지만, 그렇지 않을 때도 많음

⑧ Adaptive Moment Estimation (ADAM)

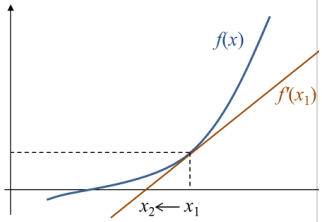
- Adadelta와 RMSProp의 장점을 모은 방식
- 알고리즘 해석
  - g_t : $Gradient(=\nabla_\theta J(\theta_t))$
  - m_t : $Momentum$, Original M에 Decay Constant를 곱한 Gradient를 더해줌
    - 기존의 Momuntum 공식 개념을 활용
  - v_t : $AdaptiveLearningRate$, Original V에 Decay Constant를 곱한 Gradient의 제곱값을 더해줌

    (Decay Constant)

    [Deep Learning from Scratch] 6장. 학습 관련 기술들
    신경망 학습의 목적은 손실 함수(loss function)의 값을 가능한 낮추는 것으로 이는 곧 신경망의 매개변수, 즉 가중치의 최적값을 찾는 문제로 귀결된다. 이러한 문제를 최적화(Optimization)라고 한다.
    https://devkor.tistory.com/entry/%ED%95%99%EC%8A%B5%EA%B4%80%EB%A0%A8%EA%B8%B0%EC%88%A0%EB%93%A4

    
  - 즉, 1st Order Gradient와 2nd Order Gradient를 둘다 고려하는 방식 (=Momentum과 Adaptive Learning Rate를 둘다 고려)
  - 수학적 해석 : Loss Surface를 Newton's Method를 활용하여 2차 함수로 근사하는 방식 → Convex Function에 대해서는 조금 더 빠르게 Minimum으로 향할 수 있음

(Newton Method)



- 중간 과정에 대한 설명 $(\hat{m}_t, \hat{v}_t)$
  - $\beta_1 = 0.9, \beta_2 = 0.9999$라 가정하자
  - Initial State에 m_{t-1}과v_{t-1}은 모두 0이고, $(1-\beta_1) = 0.1, (1-\beta_2) = 0.001$로 엄청 작은 값이되어 θ_{t+1}을 계산하면 분모는 너무 작아지고 분자가 너무도 커져버려서 첫 Step이 너무 크게 이동되는 문제가 발생
  - 따라서 m_t와v_t를각각$1-\beta_1^t$와$1-\beta_2^t$로 나누어서 첫 Step을 너무 크지 않고 적절하게 Step을 밟을 수 있도록 만들어줌
  - Step이 나아가면 갈 수록, m_{t-1}과v_{t-1}의 중요도가 커지게되어, $1-\beta_1^t$와$1-\beta_2^t$의 역할은 점점 축소된다. 즉, 첫 스텝이 너무 크지 않게하는 효과.
  - 관련 논문

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

3. How to use Advanced Optimizers in Pytorch

- 앞의 수식들은 이미 파이토치로 구현되어 있으며, 다음과 같이 함수와 다양한 파라메터들을 변경하며 사용할 수 있음

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9
```