Gradient Descent
Optimization Algorithms

TEAM Kai.Lib

발표자 : 원철황

2020.02.17 (MON)

Gradient Descent

- **In Neural network**
  - Neural Network의 weight을 조절하는 과정에는 보통 'Gradient Descent' 라는 방법을 사용함.
  - 네트워크의 Parameter들을 θ
  - 네트워크에서 내놓는 결과값과 실제 결과값 사이의 차이를 정의하는 함수 Loss function J(θ)
  - J(θ)를 최소화하기 위해 기울기 (Gradient) ∇_θ J(θ) 를 이용하는 방법을 Gradient Descent라 함.
  - ∇_θ J(θ) 는 J(θ) 가 증가하는 방향을 뜻하므로 반대 방향으로 일정 크기만큼 이동해내 Loss Function 값을 최소화하는θ 의 값을 찾는다.
  - 한 iteration 에서의 변화 식은 다음가 같음

$$\theta = \theta - \eta \nabla_\theta \mathrm{J}(\theta)$$

Gradient Descent

- **In Neural network**

$$\theta = \theta - \eta \nabla_{\theta} \mathrm{J}(\theta)$$

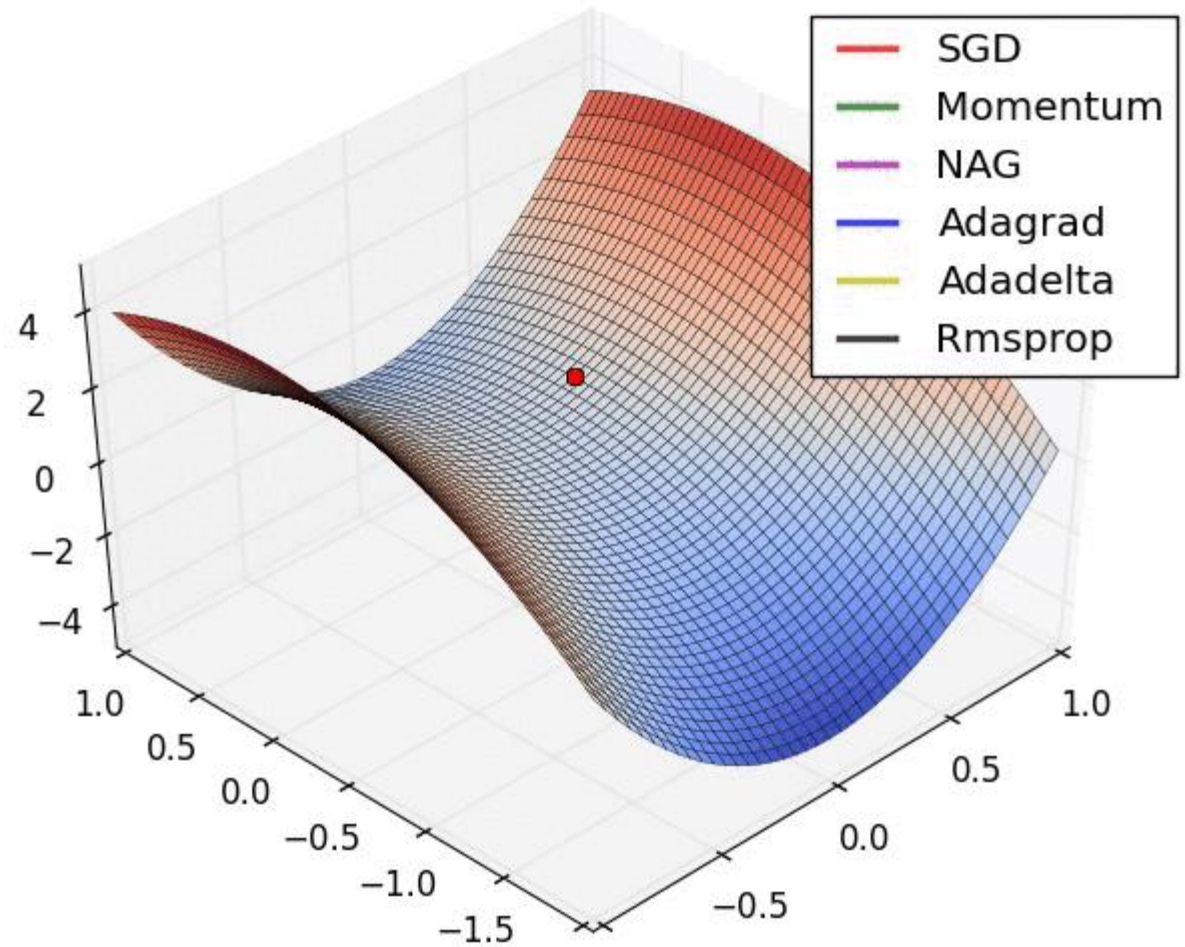
에타(η) 는 미리 정해진 step size로 learning rate 로 불린다.
보통 0.01~ 0.001 정도의 적당한 크기를 사용한다.

Loss Function을 계산할 때 전체 train set을 사용하는 것을 Batch Gradient Descent 라고 한다.
이렇게 계산할 경우 한 step마다 전체 데이터에 대해 Loss Function을 계산하므로 너무 많은 계산량이 필요하다.
이를 방지하기 위해 Stochastic Gradient Descent (SGD) 방법을 사용한다.

SGD는 loss function을 계산할 때 전체 데이터 (batch) 대신 일부 조그마한 데이터모음 (mini-batch)에 대해서만 loss function을 계산하여 이 값을 이용해 Parameter Update를 진행한다.

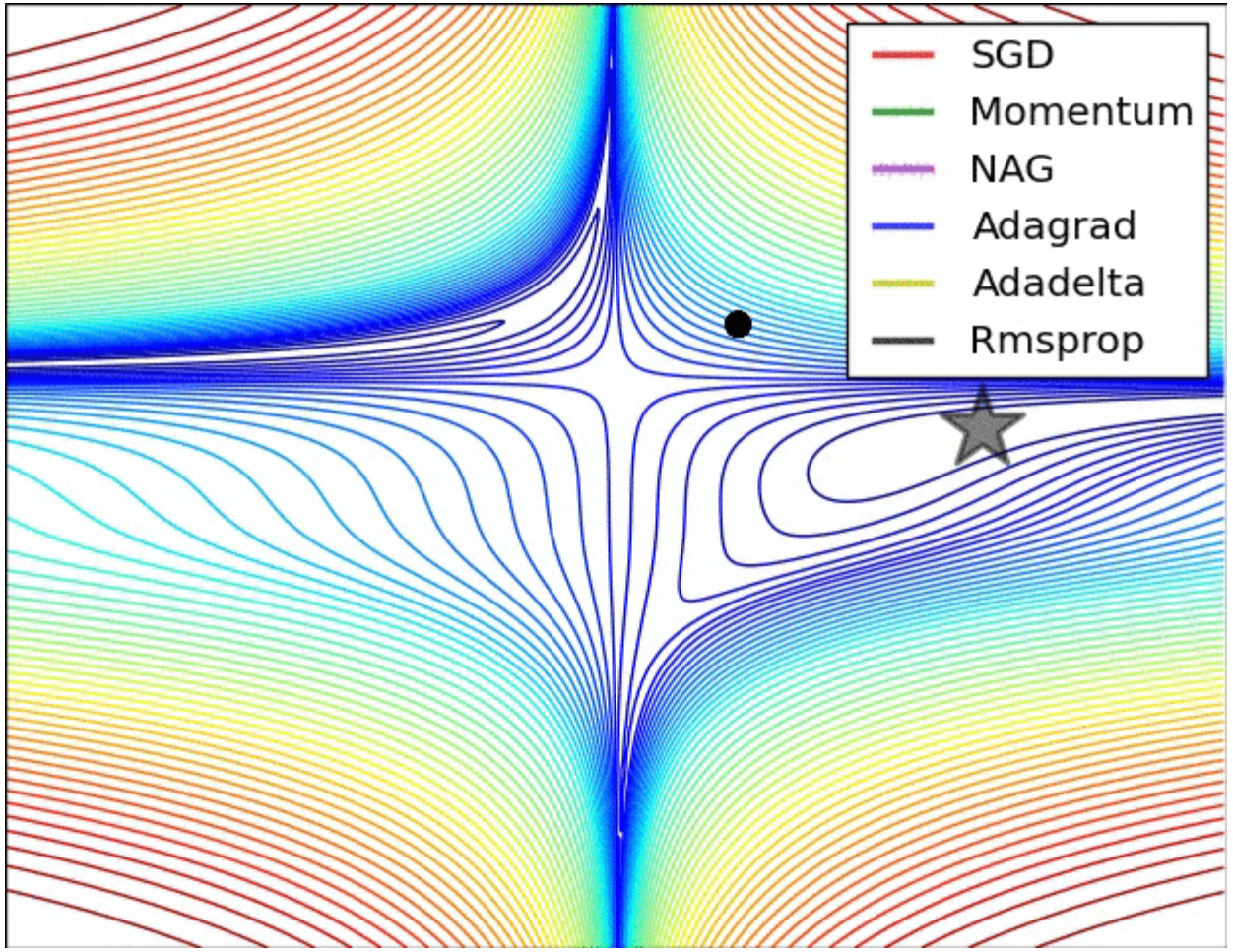
장점1) 계산속도 측면에서 훨씬 나은 성능을 기대할 수 있다.
장점2) 여러 번 반복 수행할 경우 보통 batch의 결과와 유사한 결과로 수렴
장점3) Batch Gradient Descent에서 빠질 local minima에 빠지지 않을 가능성 있음.

Gradient Descent



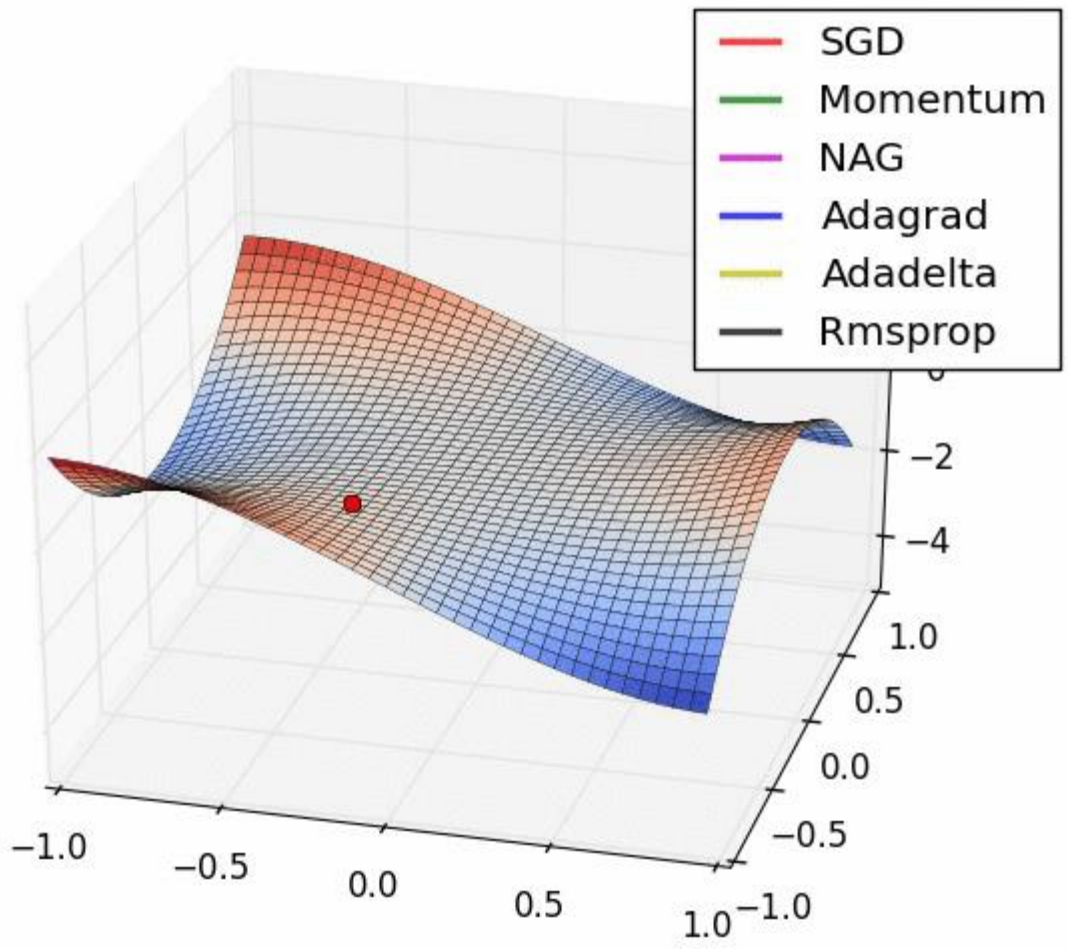
Gradient Descent Optimization Algorithms at Long Valley

Gradient Descent



Gradient Descent Optimization Algorithms at Beale's Function

Gradient Descent



Gradient Descent Optimization Algorithms at Saddle Point

Gradient Descent

위 그림들은 SGD 및 SGD의 변형 알고리즘들이 최적값을 찾는 과정을 시각화한 것이며 아래는 특징을 정리한 것.

- SGD는 다른 알고리즘들 보다 이동속도가 현저하게 느리다.
- 단순한 SGD를 이용해 네트워크를 학습시킬 경우 네트워크가 상대적으로 좋은 결과를 얻지 못할 것이라 예측할 수 있다. (같은 epoch만큼 돌았을 경우 최적값 까지 도달하는 epoch의 차이 발생 가능성)
- 네트워크와 Loss Function 의 특징에 맞춰 적당한 알고리즘을 채택하는 것이 학습 성능 개선에 영향을 줄 것.

Momentum

Momentum 방식은 말 그대로 Gradient Descent를 통해 이동하는 과정에 일종의 '관성'을 주는 것. 현재 Gradient를 통해 이동하는 방향과는 별개로, 과거에 이동했던 방식을 기억하면서 그 방향으로 추가적으로 이동하는 방식.

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

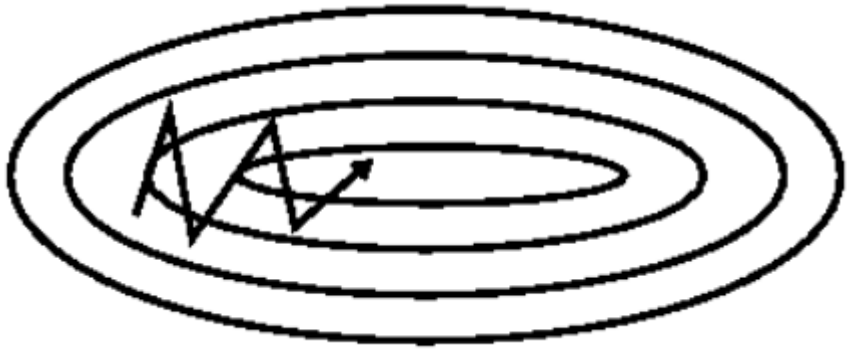
$$\theta = \theta - v_t$$

- γ 는 얼마나 momentum을 줄 것인지에 대한 momentum term 으로 보통 0.9 값 사용.
- 과거에 얼마나 이동했는지에 대한 정보를 항 v_{t-1} 에 기억
- 새로운 이동을 진행할 경우 이를 고려해 Parameter θ 를 업데이트.

Momentum

SGD가 Oscilation 현상을 겪을 때 적용 (0.9 ^ 22 = 0.0985. 따라서 22_step 부터 영향이 1/10로)

〈 단순 SGD 방식〉

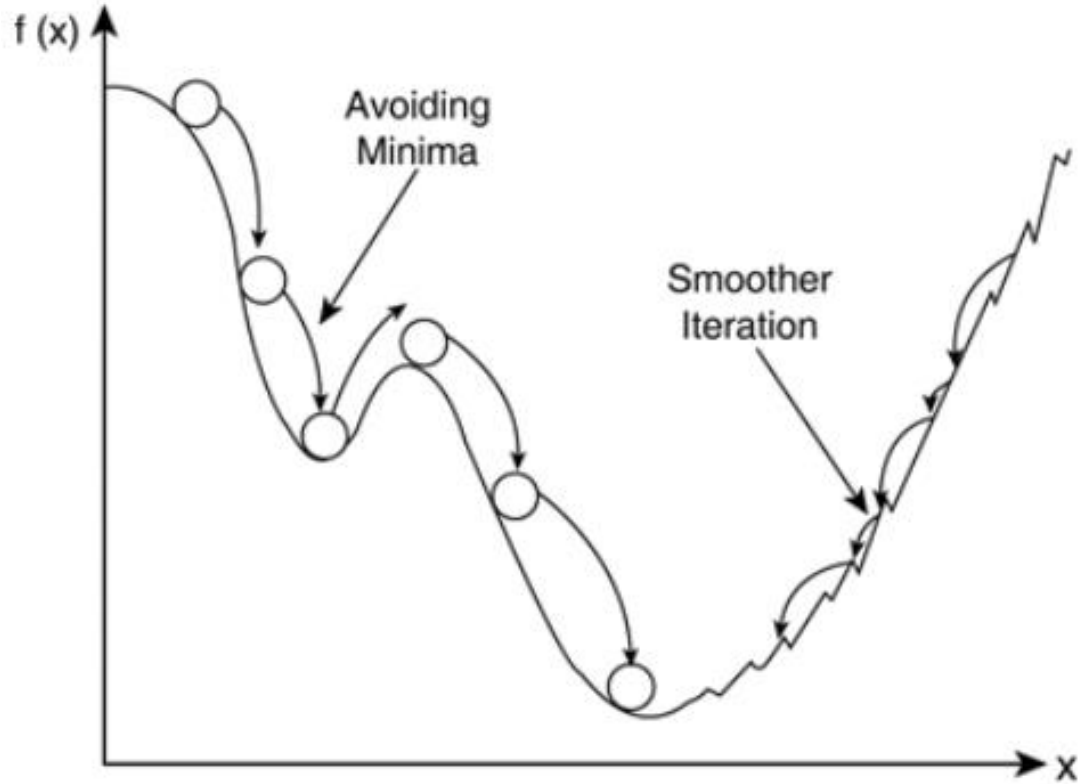


〈 Momentum 방식 〉

중앙의 최적점으로 이동해야 하는 상황에서 한번의 step으로 움직일 수 있는 step size(learning rate)는 각 Parameter θ 들의 방향으로 한계가 생김.
그러나 momentum 방식은 우측 그림에서 처럼 step이 반복될 수록 이동하는 방향으로 힘을 얻기 때문에 상대적으로 빠르게 이동할 수 있다. (그 효과는 γ 값의 곱만큼 계속해서 줄어든다)

$$v_t = \eta\nabla_\theta \mathrm{J}(\theta)_t + \gamma\eta\nabla_\theta \mathrm{J}(\theta)_{t-1} + \gamma^2\eta\nabla_\theta \mathrm{J}(\theta)_{t-2} + \ \ldots$$

Momentum



Avoiding Local Minima. Picture from

장점1) local minima를 빠져나오는 효과

단점1) 기존 변수 θ 외에도 과거에 이동했던 양을 변수별로 (Parameter별로 저장해야 하므로 메모리가 기존의 두 배 필요

Nesterov Accelerated Gradient (NAG)

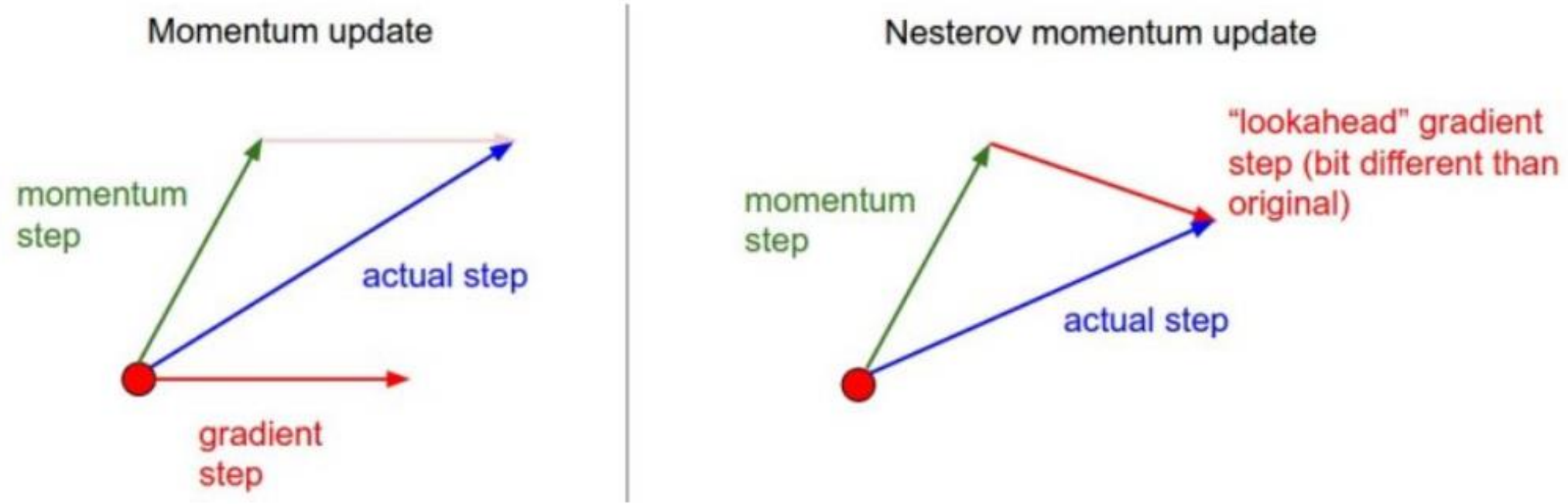
- Momentum 방식에서는 이동 벡터 v_t 를 계산할 때 현재 위치에서의 gradient와 momentum step을 독립적으로 계산하고 합친다.
- 반면 NAG에서는 momentum step을 먼저 고려하여, 먼저 이동했다고 생각한 후 그 자리에서의 gradient를 구해서 gradient step을 이동함.

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

$$\theta = \theta - v_t$$

- 멈춰야 할 시점에도 관성에 의해 훨씬 멀리 갈수도 있다는 단점을 보완.
- Momentum 방식의 빠른 이동에 대한 이점을 누리면서 멈춰야 할 적절한 시점에서 제동을 거는 데 훨씬 용이함.
- 단점) 연산량이 기존 Momentum 방식보다 많음.

Nesterov Accelerated Gradient (NAG)



Adagrad (Adaptive Gradient)

- Adagrad (Adaptive Gradient)는 변수들을 update 할 때 각각의 변수마다 step size를 다르게 설정해서 이동하는 방식
- 알고리즘의 기본적인 아이디어: 지금까지 많이 변화했던 변수들은 step size를 작게 하자
- 자주 등장하거나 변화를 많이 한 변수들의 경우: optimum에 가까이 있을 확률이 높기 때문에 작은 크기로 이동하면서 세밀한 값을 조정
- 적게 변화한 변수들의 경우: optimum 값에 도달하기 위해서 많이 이동해야 할 확률이 높기 때문에 먼저 빠르게 loss 값을 줄이는 방향으로 이동하려는 방식
- Word2vec 이나 GloVe 같이 word representation을 학습시킬 경우 단어의 등장 확률에 따라 variable의 사용 비율이 확연하게 차이 나기 때문에 Adagrad 학습 방식을 사용하면 더 좋은 성능 거둘 수 있음.

Adagrad (Adaptive Gradient)

- Neural Network의 parameter가 k개라고 할 때, G_t 는 k차원 벡터로서 time step t까지 각 변수가 이동한 gradient 의 sum of squares 저장
- θ 를 업데이트하는 상황에서는 기존 step size η 에 G_t 의 루트 값에 반비례한 크기로 이동을 진행
- 지금까지 많이 변화한 변수일 수록 적게 이동
- 적게 변화한 변수일 수록 많이 이동
- ϵ은 $10^{-4} \sim 10^{-8}$ 정도의 작은 값으로 0으로 나눠지는 것을 방지
- G_t 업데이트하는 식에서 제곱은 element-wise 제곱을 의미하며 · 연산 역시 element-wise 한 연산을 의미

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

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

Adagrad (Adaptive Gradient)

- 장점) 고의적 step size decay 를 신경 쓰지 않아도 된다.
- 단점) 학습이 계속적으로 진행될 때 step size가 너무 줄어든다는 문제점.

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

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

- 학습이 오래될 경우 결국 거의 움직이지 않게 되는데, 이를 보완하여 고친 알고리즘 존재
- RMSProp 과 AdaDelta

RMSProp

- 딥러닝의 대가 제프리 힌톤이 제안한 방법
- Adagrad의 단점 해결
- Gradient의 제곱값을 더해 나가면서 구한 G_t 부분을 합이 아니라 지수 평균으로 바꾸어 대체.
- 장점) G_t가 무한정 커지지 않는 효과
- 장점) 변화량의 변수간 상대적 크기 차이는 유지

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

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

AdaDelta (Adaptive Delta)

- 동일하게 G_t 를 구하지만 Parameter 업데이트를 위한 합을 구하는 과정에서 지수 평균을 사용

$$G = \gamma G + (1-\gamma)(\nabla_\theta J(\theta_t))^2$$

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

$$\theta = \theta - \Delta_\theta$$

$$s = \gamma s + (1-\gamma)\Delta_\theta^2$$

- Gradient Descent와 같은 First-order optimization 대신 Second-order optimization을 approximate 하기 위한 방법.
- SGD, Momentum, Adagrad 와 같은 식들의 경우 $\Delta\theta$의 unit(단위)를 구해보면 θ의 unit이 아니라 θ의 unit의 역수를 따름.
- 이를 u(θ) 라고 하고 J는 unit이 없다고 가정하면 first-order optimzation에서는 $\Delta\theta \propto \frac{\partial \bar{J}}{\partial \theta} \propto \frac{1}{u(\theta)}$
- 반면 Newton metho를 이용한 근사로 second-order optimizatio은 다음과 같음.

$$\Delta\theta \propto \frac{\frac{\partial \bar{J}}{\partial \theta}}{\frac{\partial^2 \bar{J}}{\partial \theta^2}} \propto u(\theta)$$

https://arxiv.org/pdf/1212.5701.pdf

Adam (Adaptive Moment Estimation)

- RMSProp과 Momentum 방식을 합친 것 같은 알고리즘.
- Momentum 방식과 유사하게 지금까지 계산해온 지수평균을 저장
- RMSProp과 유사하게 기울기의 지수 평균을 저장

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta))^2$$

- Adam에서는 m과 v가 처음에 0으로 초기화. 때문에 초반부에서는 0에 가깝게 bias 되어있을 것이라 판단하여 이를 unbiased 하게 만들어주는 작업 가짐.
- 보정된 expectation들을 가지고 gradient가 들어갈 자리에 m_hat, G_t가 들어갈 자리에 v_hat 을 넣어 계산 진행.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta = \theta - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t \quad (\beta 1 = 0.9, \beta 2 = 0.999, \epsilon = 10^{-8})$$

Conclusion

- Momentum, NAG, AdaGrad, AdaDelta, RMSProp, Adam 등 다양한 Optimization 알고리즘 존재.
- 어떤 문제를 풀고, 어떤 데이터 셋을 사용하는지, 어떤 네트워크에 적용하는지에 따라 성능은 판이하게 달라짐.
- 실제 네트워크 학습 시 다양한 시도를 해보며 성능 비교를 수행하는 것이 적합.
- Tensorflow 의 Machine Learning librar에는 optimizer 로 정의된 함수들 존재 (PyTorch는 모르겠어요)
- 여기서 설명한 알고리즘들은 모두 Stochastic Gradient Descent 방식으로 단순한 first-order optimization의 변형.
- 단순한 second-order optimization 사용하기 위해서는 Hessian Matrix란 2차 편미분 행렬을 계산한 후 역행렬을 구해야 함. 이 과정이 비싼 작업이라 보통 사용X
- 이러한 계산량을 줄이기 위한 hessian matrix를 근사하거나 추정해 나가며 계산을 진행하는 Algorithms 이 있으나, Python Library 로 구현됐는지는 미지수. (분명 있겠지만 Optimizer에 적용되었는지 여부는 모르겠습니다. 안 됐으면 Go 해보는 것도?)