COSE474-2021F: Final Project Proposal
optimizer **가능성 탐구**

2019320138 **김규진**

1. Introduction

딥러닝 과정에서 설정하고 연결해줘야 할 모듈들로는 activation function, normalize, regularize 등 기본적인 부분에서조차 다양한 방식이 사용되고 있으며, 학습하는 모델에 따라 어떤 것을 활용해야 할지 설정해야 최적의 결과를 얻을 수 있다. 이중 적절한 optimizer을 정하는 것도 결과에 영향을 주는 주요한 부분으로 알려져 있다. Optimizer를 잘 선택한다면 연산량을 획기적으로 줄일 수 있기 때문에 결과가 나오기까지 기다려야 하는 시간을 최소화할 수 있고, 결과값 또한 더 최적값을 이끌어낼 수 있다. Momentum은 이름에서 볼 수 있듯 관성을 활용한 방식인데, 이처럼 optimizer에는 각각 현실에서 어떤 개념을 가지고 optimizer을 구현하고 있다. 그래서 이번 연구에서 또한 직접 철학적인 개념을 활용해 여러 optimizer을 고안해 적용해보고자 했다. Base가 되는 모델로는 momentum을 이용했다. 관성을 이용하는 가장 기본적인 모델이기도 하며, 다양한 optimizer가 momentum에서 파생해 만들어졌기 때문에 선택하게 되었다. 또, learning rate를 연산과정에 넣어 따로 scheduling을 진행하는 기본적인 의례와는 달리, learning rate를 optimizer에서 고려하도록 설계해 차별점을 주었다.
새로운 개념의 optimizer들을 활용한다면 나머지 같은 모듈들을 활용한다 하더라도 작은 learning rate를 이용하더라도 더 적은 epoch를 이용해도 결과를 더 빠르게 도출해낼 수 있을 것이라고 예상하고 진행했다.
하지만 두 가지 개념을 이용해 만든 두 가지 optimizer은 모두 아쉽게도 유의미한 연산 속도 차이를 보여주지 못했다. 하지만, 이를 통해 왜 learning rate를 learning scheduler를 이용해 작업을 하는 것이 나은지 알게 되었고, 각 기능마다 의존성을 떨어뜨리는 것이 작업 능률이 좋다는 것을 알 수 있었다.

2. Methods

여기서는 optimizer을 만드는 데 있어 새로운 개념을 도입한 두 가지 새로운 optimizer을 이용한다. 이 두 가지는 learning rate를 결과값을 향한 중력 방식의 가속도의 강도라는 개념을 도입한 gravity optimizer와, momentum이 가리키는 방향과 진행방향이 다르다면 급격한 가중치를 주어 방향을 전환시키는 cosine optimizer을 소개한다.

1. gravity optimizer

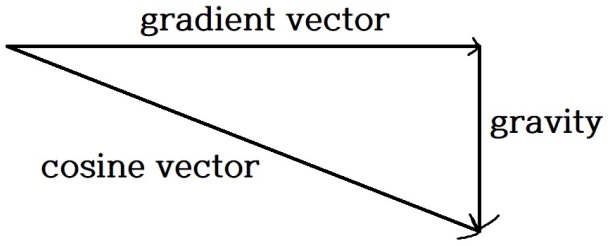


Figure 1. Gravity Momentum

첫 번째 소개할 방식은 gravity optimizer이다. 이는 기본적인 weight의 방향 말고, 결과값을 향한 가속도를 lr의 크기로 설정한다. 이를 통해 벡터가 클수록 더 강한 속도로 학습되고 있음을 보정한다. Algorithm 1 이 그 의사코드 형태를 보여준다.
우선 Figure 1을 보면, 유도 과정을 쉽게 이해할 수 있다. 결과부터 이야기를 하자면, 원래의 gradient 값을 가지는 gradient vector에서 gravity만큼의 방향벡터를 추가해 cosine vector와 같은 크기로 조정을 해주는 것이다. 먼저 gradient vector는 gradient로 이루어진 전체 vector의 크기이다. gravity가 사실상 핵심 부분이다. gravity는 $lr \cdot \sqrt{N}$으로, learning rate를 vector dimension의 루트로 나눈 것으로 한다. 그냥 lr로 할 수 있지만 $\sqrt{N}$으로 곱해준 것은 demension이 너무 커질 경우, 아무리 lr을 값을 조정해도 그 사이즈 때문에 의미가 없는 값처럼 작아지기 때문이다. 그렇기 때문에 $\sqrt{N}$를 곱해 의미가 있도록 해준다. 이후 계산이 중요하다.

$$\vec{cosine} = \sqrt{(\vec{gradient})^2 + gravity^2}$$

$$cos\theta = \frac{|\vec{gradient}|}{|\vec{cosine}|}$$

$$new\vec{gradient} = \vec{gradient} * cos\theta$$

이렇게 gradient를 새로 조정하게 되면 learning rate를 gravity로 적용할 수 있다. 이를 momentum 알고리즘에 적용해 만든 gravity momentum optimizer가 Algorithm 1에 작성되어있다.

2. cosine optimizer

두 번째 소개할 방식은 cosine optimizer이다. 이는 현재 진행방향이 이번에 받은 가속도와 방향이 많이 다를수록, 더 확실하게 이번 가속도를 많이 반영하는 방식이다. 말 그대로 두 벡터의 회전각을 구해 각이 평각에 가까워질수록 더 작은 값을 적용하도록 한다.

Algorithm 1 Gravity Optimizer

Input: parameter matrix array $params$, gradient matrix array $grad$, saver matrix array m, learning rate lr, parameter dimension N, momentum rate u
repeat
  for param in params
  for $param$ in $params$ do
    $S = \sum \frac{(grad)^2}{N}$
    $k = \sqrt{1 + \frac{lr^2}{S}}$
    $m = um - k \cdot lr \cdot grad$
    $param+ = m$
  end for
until backpropagations are finished

Algorithm 2 Cosine Optimizer

Input: parameter matrix array $params$, gradient matrix array $grad$, saver matrix array m, cosine value matrix array $cosine$, cosine constant C, learning rate lr
repeat
  for $param$ in $params$ do
    for $\vec{m}, \vec{g}, \vec{cos}$ in $m, gradient, cosine$ do
      $cos\theta = \frac{\vec{m} \cdot \vec{g}}{||m|| \cdot ||g||}$
      $\vec{cos} = cos\theta \cdot \vec{1}$
    end for
    $m = C\frac{cosine+1}{2}m + lr \cdot gradient$
    $param- = m$
  end for
until backpropagations are finished

$cos\theta = \frac{\vec{m} \cdot \vec{g}}{||m|| \cdot ||g||}$ 를 통해 각도를 구한 후, 저장되어있는 과거의 momentum에 추가해준다. 각이 벌어질수록 0에 가깝고 각이 작을수록 그대로 해줘야하니 값을 변형해 적용한다. Algorithm 2에서 의사코드 형태를 보인다.

두가지 방식이 모두 효과적인 향상을 불러일으키지 못하더라도 optimizer을 실제로 구현해보며 optmizer에 대한 개인적 이해를 높일 수 있고, optimizer가 요구하는 사항이 어떤 것이 있는지 더 자세하게 알아볼 수 있을 것이다. plateau 현상과 zigzag가 가장 optimizer의 과제이다. 이를 위해 Momentum 방식을 이용했지만, 이는 zigzag 문제를 해결해주지 못했기에 adaptive방식과 혼합된 adam이 주로 사용되는 추세이다. nesterov momentum 방식은 비용에 비해 성능 향상이 적었다고 한다. 이에 많은 optimizer들 중 momentum optimizer을 이용한 결과와 비교하여 성능향상이 실제로 일어났는지 비교하고자 한다.

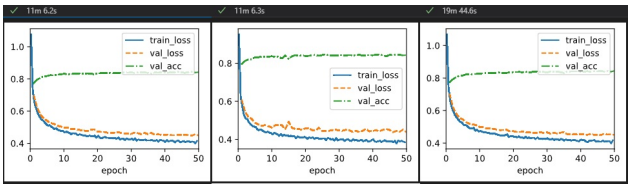
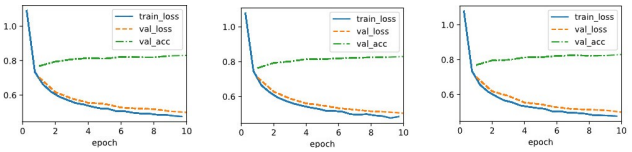


Figure 2. lr=0.05



(a) sgd (b) gravity (c) cosine

Figure 3. lr=0.1

3. Experiments

데이터셋은 튜토리얼에서 진행한 옷을 이용한 FashionMNIST dataset을 이용했다. computing resource는 dataset의 규모가 크지 않기에 CPU를 이용한다. windows 10 환경에서 실행했으며, pytorch를 활용해 코드를 작성한다. setup으로 python 3.10.4를 이용했다. ipynb로 작성되었다. 기본적으로 input 784개, output 10개, learning rate 0.05로 하고 momentum rate는 0.03으로 정한다. epoch 50개로 늘려 진행한다. SOTA는 momentum으로 실행했을 때의 결과를 Default result로 잡았다. Figure 2에서, gravity optimizer에서는 약 1.8배 빠르게 training loss와 validation loss가 낮아진 모습을 볼 수 있다. 하지만, cosine optimizer에서는 큰 변화가 나타나지 않은 모습을 볼 수 있다. 결과 예시 사진 loss and accuracy. 여기서 img파일 안에 있는 사진 파일들을 활용한다.

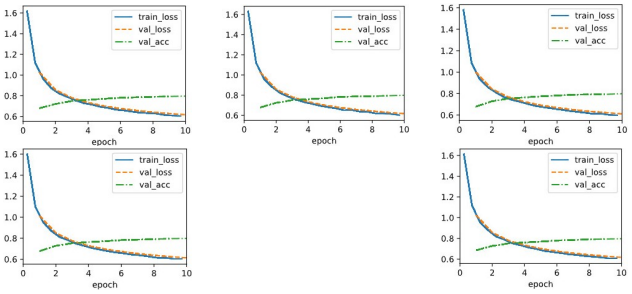


Figure 4. lr=0.01

이게 왜 실패했는가? 너무 복잡하고, learning rate를 넣어서 계산한다고 해서 gradient가 드라마틱하게 달라지는 것이 아닌 것 같음. 그럼에도 미세한 차이가 보이는 것으로 보아 dataset이 너무 작아서 차이를 비교하기 어려운 수준이 아니었나 라는 예상이 됨.

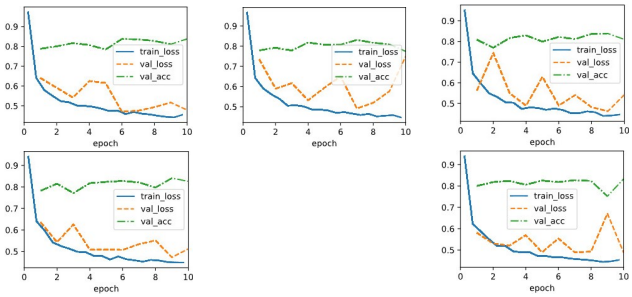


Figure 5. lr=0.2

4. Future direction

다음에는 다른 데이터셋을 활용해 훨씬 대규모의 데이터를 처리해 봐야할 것이다. epoch가 100은 되는 것을 해봐야하지 않을까? classification을 활용하는 것은 그대로 가는 것이 좋을 것 같다.