MNIST의 CNN 모델 기반 TensorFlow Optimizer 성능 비교 분석

Comparison Analysis of TensorFlow's Optimizer Based on MNIST's CNN Model

임 현 교(Hyun-Kyo Lim), 김 주 봉(Ju-Bong Kim), 권 도 형(Do-Hyung Kwon),
한 연 희(Youn-Hee Han)

Advanced Technology Research Center
Korea University of Technology and Education
{glenn89, yhhan, rlawnqhd, dohk}@koreatech.ac.kr

요 약

최근 머신 러닝에 대한 관심이 높아짐에 따라 딥 러닝의 Neural Network에 대한 관심이 높아지고 있다. 특히, 다양한 Neural Network가 등장 하면서 네트워크에서 사용되는 네트워크 파라미터에 대한 최적화 알고리즘의 관심이 증가하고 있다. 또한, 다양한 Neural Network 가 등장함에 따라 네트워크 파라미터를 최적화 시키는 알고리즘도 나타나고 있다. 하지만 새로운 Optimizer들이 모든 Neural Network에 최적화 되어 있지 않기 때문에 학습 데이터의 입력 값과 네트워크 파라미터 값에 따라 학습 결과 값이 달라지는 문제가 있다. 따라서, 본 논문에서는 loss 함수의 값을 최소화 시키기 위한 네트워크 파라미터의 최적화 알고리즘들을 소개하며, 이를 위하여 현재 자주 사용되고 있는 Gradient Descent, Momentum, Adagrad, Adadelta, Adam Optimizer들에 따라 MNIST의 CNN의 학습 결과의 정확도를 비교 평가 하였다.

키워드: Machine-Learning, Deep Learning, Convolution Neural Network, Neural Network, Optimizer

Abstract

As interest in Machine-Learning has increased in recent years, interest in Deep-Learning Neural Networks is growing. Especially, the appearance of various Neural Networks has increased the interest of optimization algorithms on network parameters used in networks. Algorithms for optimizing network parameters are also emerging as various Neural Networks are introduced. However, since the new optimizers are not optimized for all Neural Networks, there is a problem that the learning result value varies depending on the input values of the learning data and the network parameter values. In this paper, we introduce network parameter optimization algorithms to minimize the value of loss function. For this, we compare the accuracy of CNN learning results of MNIST according to Gradient Descent, Momentum, Adagrad, Adadelta, and Adam Optimizers Respectively.

Key words: Machine-Learning, Deep Learning, Convolution Neural Network, Neural Network, Optimizer

I. 서 론

최근 머신러닝의 딥러닝에 대한 관심이 증가하고 있다. 이에 따라 더 나은 학습결과를 도출해 낼 수 있는 다양한 Neural Network가 등장하고 있다. 현재 딥 러닝에서 가장 많이 쓰이는 Neural Network는 Convolution Neural Network (CNN), Recurrent Neural Network (RNN) [1]이 대표적으로 쓰이고 있다. 다양한 Neural Network가 등장함에 따라, 학습 결과의 정확도를 높이기 위하여 각 학습 모델의 loss 함수를 최소화 시키기 위한 네트워크의 파라미터를 최적화 시킬 수 있는 다양한 Optimizer들이 등장하고 있다.

하지만 Optimizer들은 Neural Network의 입력 데이터 값, 초기 네트워크 파라미터 값에 따라 학습 결과값이 달라지게 된다. 따라서 데이터를 학습을 시켜야 하는 사용자의 경우 각 Optimizer들이 각기 다른 학습 결과 값을 나타내기 때문에 해당 학습을 위한 학습 모델에서는 어떠한 Optimizer가 loss 함수를 최소화 시키기 위한 네트워크 파라미터의 최적화 알고리즘인지 찾기 어렵다 [2].

따라서, 본 논문에서는 딥 러닝을 위하여 가장 많이 사용되고 있는 TensorFlow 오픈소스 라이브러리 [3]의 Optimizer들인 Gradient Descent, Momentum, Adagrad, Adadelta, RMSProp, Adam들을 비교 평가하려 한다. 비교를 위하여 MNIST 데이터[4]를 학습을 위한 입력 데이터로 사용 하였으며, 딥 러닝에서 이미지 데이터 학습에 가장 많이 이용되고 있는 CNN 학습 모델을 사용하였다. 그 결과로 CNN 학습 결과의 정확도를 통해 각 Optimizer의 성능을 비교하고, 100개의 Neural과 5개의 Layer로 구성된 학습 모델을 통해 각 Optimizer의 학습 속도를 측정된 loss 값으로 비교 평가한다.

본 논문의 구성은 다음과 같다. 1장의 서론에 이어 2장에서는 본 논문에서 비교 평가를 하기 위하여 TensorFlow에서 가장 많이 사용되고 있는 6개의 Optimizer에 대하여 설명하고, 3장에서는 Optimizer의 비교를 위한 MNIST 데이터와 딥 러닝의 CNN 학습 모델을 설명한다. 4장에서는 각 Optimizer들의 학습 결과의 정확도를 비교 평가하며, 마지막으로 5장에서 결론을 도출한다.

II. 관련 연구

현재 TensorFlow에서 제공하는 Optimizer는 Gradient Descent [5], Adadelta [6], Adagrad [7], Momentum [8,9], Adam [10], Ftrl [1], Proximal Gradient Descent [12], Proximal Adagrad [12], RMSProp [13] Optimizer들을 제공하고 있다. 최근 많이 사용되고 있는 Optimizer는 Gradient Descent Optimizer를 변형한 Adagrad의 단점을 극복한 RMSProp Optimizer와 RMSProp 와 Momentum Optimizer의 장점들을 이용한 Adam Optimizer가 많이 사용되고 있다. 본 논문에서는 딥 러닝의 Neural Network를 통한 데이터 학습이 많이 이용되면서 과거부터 지금까지 많이 이용이 되고 있는 Gradient Descent, Adadelta, Adagrad, Momentum, Adam, RMSProp Optimizer들의 성능을 비교함과 동시에 본 장에서는 6개 Optimizer의 자세한 설명을 한다.

2.1. Gradient Descent Optimizer

Gradient Descent Optimizer는 Neural Network의 가장 기본적인 학습 방법으로, Neural Network에서 내놓은 결과값과 실제 결과값의 차이를 정의하는 Loss 함수를 최소화 하기 위하여 기울기를 이용하는 방법이다. Gradient Descent 에서는 네트워크 파라미터들에 대해 기울기의 반대 방향으로 일정 크기만큼 이동해내는 것을 반복하여 loss 함수의 값을 최소화하는 네트워크 파라미터들의 값을 찾는다. 이를 위하여 한 iteration에서의 변화 식은 다음과 같다.

$$\theta = \theta - \eta \nabla_{\theta} J(\theta) \quad (1)$$

위 수식의 θ는 Neural Network의 파라미터를 나타내며, $\nabla_\theta J(\theta)$는 loss 함수의 기울기를 나타낸다. η는 얼마만큼의 기울기를 사용할지 결정하는 값인 learning rate를 나타낸다. Gradient Descent의 경우 learning rate값에 따라 local minima에 빠지거나, 발산하는 경우가 있다. Local minima문제는 loss 함수 값을 최소화 시키는 최적의 네트워크 파라미터를 찾는 문제에 있어서 지역적으로 홀들이 존재하여 local minima에 빠질 경우 전역적인 최적의 파라미터를 찾기 힘들게 되는 문제를 말한다. 따라서, learning rate는 학습을 위한 입력 데이터의 크기에 따라 적절한 learning rate 값을 결정해야 하며, 이를 통하여 loss함수의 값을 최소화 시킬 수 있는 네트워크 파리미터 값을 찾는 것이 중요하다.

Gradient Descent는 loss 함수를 계산할 때 전체 train 데이터 셋을 사용하게 된다. 따라서, 전체 train 데이터 셋에 대한 계산을 한 뒤에 네트워크 파라미터를 업데이트 하기 때문에 계산 량이 너무 커 최적화된 네트워크 파라미터를 찾아가는 속도가 느려 현재는 거의 쓰이지 않고 있다.

2.2. Momentum Optimizer

Momentum은 기존의 Gradient Descent를 통해 이동하는 과정에 일종의 관성의 원리를 추가하는 것이다. Momentum은 구현과 원리가 매우 간단하면서 Gradient Descent 보다 좋은 성능을 내는 것으로 알려져 있다. 관성의 원리를 추가한다는 것은 이전 계산에 의해 나온 기울기를 일정한 크기만큼 반영하여 새로운 기울기와 합하여 사용하는 것이다. 즉, 기존의 기울기를 일정 부분 유지하면서 새로운 기울기를 적용하여, 관성과 같은 효과를 주는 방법이다. Momentum을 수식으로 표현하면 다음과 같다.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \quad (2)$$

$$\theta = \theta - v_t \quad (3)$$

위 수식의 γ는 얼마 만큼의 관성을 줄 것인지에 대한 momentum 상수로서, 일반적으로 0.9 정도의 값을 사용한다. v_t는 iteration에서의 타임 스텝에 따른 이동 벡터를 의미한다. 위 수식에서 보이는 바와 같이 새로운 이동 벡터를 구할 때, 과거 이동했던 정도에 관성을 나타내는 상수로 미리 정해 놓은 γ만큼을 반영하여 이동 벡터를 구하고, 이동 벡터를 통하여 loss 함수를 최소화 할 수 있는 네트워크 파라미터를 구하게 된다.

Momentum 방법은 과거의 모든 기울기가 계산을 진행 할 때마다 일정하게 누적된다. 하지만, 위 수식을 통하여 계산을 하는 경우 learning rate가 너무 크게 되면 weight가 발산하는 경우가 발생하기 때문에 작게 설정해야 한다. Momentum 방법은 기존의 Gradient Descent 방식을 어느정도 이용하고 있지만, 이를 관성의 원리를 이용하여 보완하는 방법이다. 따라서 기존에 존재하던 local minima에 빠지는 문제를 해결 할 수 있을 것으로 기대하고 있다. 반면, Momentum은 기존의 네트워크 파라미터 이외에도 과거의 이동 벡터에 대한 정보를 따로 저장해야 하므로 최적화 과정에서 기존의 메모리 사용량에 두배를 필요로 하게 된다.

2.3. Adagrad Optimizer

Adagrad는 네트워크 파라미터를 업데이트 할 때 iteration이 반복 될때, 각 변수의 step size를 다르게 설정하여 이동하는 방법이다. Adagrad는 현재까지 각 변수의 변화 값이 작은 경우 변수들의 step size를 크게 설정하고, 이와 반대로 각 변수의 변화 값이 큰 경우 변수의 step size를 줄이는 방법이다. 자주 등장하고 변화 값이 큰 변수의 경우 최적화 값에 가까이 있을 확률이 높기 때문에 작은 크기로 이동하면서 세밀한 값을 조정해야 하며, 변화 값이 작은 변수의 경우 최적화 값에 도달하기 위해서는 더 많은 이동이 필요할 확률이 높기 때문에 빠르게 loss 함수 값을 줄이는 방향으로 이동하려는 방법이다. Adagrad의 한 iteration에서의 변화 식은 다음과 같다.

$$G_t = G_{t-1} + (\nabla_\theta J(\theta_t))^2 \quad (4)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla_\theta J(\theta_t) \quad (5)$$

Neural Network의 파라미터가 k개일 때, G_t는 k 차원의 벡터로서 iteration의 time step t 까지 각 변수가 이동한 기울기의 제곱 합을 저장한다. θ를 업데이트하는 상황에서 learning rate 값을 나타내는 η에 G_t의 루트 값에 반비례한 크기로 이동을 진행하도록 하여 지금까지 많이 변화한 변수일 수록 적게 이동하게 하며, 적게 변화한 변수일 수록 많이 이동하도록 한다. 이때, ϵ은 G_t값이 작아지면서 0으로 나누어지는 것을 방지하기 위하여 $10^{-4} \sim 10^{-8}$ 정도의 작은 값을 나타낸다.

Adagrad를 사용하게 되면 학습을 진행하는 동안, learning rate decay에 대한 고려를 하지 않아도 된다는 장점이 있다. 그러나 Adagrad에서는 학습을 진행 할 수록 learning rate 값이 너무 줄어드는 문제가 존재한다. G_t를 계산할 때, 제곱한 값을 반복적으로 합하기 때문에 G_t의 값이 계속 증가하게 된다. 따라서, learning rate의 값이 너무 작아져 학습의 iteration이 커지게 되면 거의 움직이지 않게 되는 단점이 있다.

2.4. Adadelta Optimizer

Adadelta는 Adagrad의 단점을 극복하기 위하여 개발된 최적화 알고리즘이다. 이를 위하여, G값을 구할 때 합을 구하는 대신 지수 평균을 이용하여 구하는 방법이다. Adadelta는 Adagrad와 다르게 learning rate 값을 단순히 η으로 사용하는 대신 learning rate의 변화 값에 제곱을 가지고 지수 평균 값을 이용하여 loss함수를 최적화 시키는 네트워크 파라미터의 값을 구하게 된다. Adadelta의 수식은 다음과 같다.

$$G = \gamma G + (1-\gamma)(\nabla_\theta J(\theta_t))^2 \quad (6)$$

$$\Delta_\theta = \frac{\sqrt{s+\epsilon}}{\sqrt{G+\epsilon}} \cdot \nabla_\theta J(\theta_t) \quad (7)$$

$$\theta = \theta - \Delta_\theta \quad (8)$$

$$s = \gamma s + (1-\gamma)\Delta_\theta^2 \quad (9)$$

위 수식에서 보이는 바와 같이 Adadelta는 G값을 구하는데 있어서 지수 평균을 이용한다. 지수평균으로 계산된 G 값을 이용하기 때문에 learning rate 값이 학습이 진행됨에 따라 너무 작아지는 단점을 극복하였다. 또한 단순히 learning rate 값 대신에 learning rate 값의 제곱 합에 지수평균 값을 이용하여 learning rate의 변화 량을 나타내는 s를 구하게 된다.

하지만, Adadelta의 경우 Adagrad의 단점을 보완하기 위하여 등장하였으나, Neural Network를 통한 학습 시 Adadelta를 최적화 알고리즘으로 사용하게 될 경우 실제 학습 결과는 Adagrad에 비해 떨어지는 경우가 있다. 그 이유는 Adagrad와 다르게 일반적인 learning rate를 사용하는 것이 아니라 learning rate의 변화 값의 지수 평균을 사용하기 때문에 학습이 진행 되면서 learning rate decay가 Adagrad에 비해 떨어지게 된다. 또한 전체적인 학습의 error값의 변화가 천천히 진행 되기 때문에 Adagrad에 비해 최적화 성능이 낮아지게 된다.

2.5. RMSProp Optimizer

RMSProp 는 Adagrad의 단점을 보완하기 위하여 Adadelta와 함께 제안된 방법이다. Adagrad의 수식에서는 기울기의 제곱 값을 더해 나가면서 G_t를 구하기 때문에 learning rate의 값이 작아지기 때문에 전체적으로 거의 움직이지 않는 단점이 존재한다. RMSProp는 G_t를 구하는 과정에서 제곱 합을 이용하는 것이 아니라, 지수평균으로 바꾸어 구하는 방법이다. 지수평균으로 대체할 경우 G_t가 무한정 커지지 않으며, 최근 변화 량의 변수간 상대적인 크기 차이를 유지할 수 있다. RMSProp를 수식으로 나타내면 다음과 같다.

$$\mathrm{G} = \gamma \mathrm{G} + (1-\gamma)(\nabla_\theta J(\theta_t))^2 \quad (10)$$

$$\theta = \theta - \frac{\eta}{\sqrt{G+\epsilon}} \cdot \nabla_\theta J(\theta_t) \quad (11)$$

위 수식과 같이 γ값을 이용하여 이전 G값을 반영하여 기울기 제곱의 지수 평균 값을 구하게 된다. 따라서 Adagrad와 마찬가지로 learning rate decay

에 대한 고려가 필요 없으며, learning rate의 값이 너무 작아지는 단점 또한 지수 평균을 이용하여 보완하였다.

RMSProp는 Adagrad 보다 좋은 학습결과를 나타내며, Adagrad의 단점을 보완한 Adadelta 보다도 더 나은 학습 결과를 나타내고 있다. 또한 RMSProp는 momentum을 적용할 수 있다. momentum 상수를 적용함으로써 이전 기울기에 대한 이동 벡터를 적용할 수 있어 현재 Adam Optimizer와 함께 많이 사용되고 있다.

2.6. Adam Optimizer

Adam은 RMSProp와 Momentum의 장점을 합친 최적화 알고리즘이다. Adam은 Momentum과 유사하게 지금까지 학습이 진행 되며 계산된 기울기의 지수 평균을 저장하며, RMSProp와 유사하게 기울기의 제곱 값의 지수 평균을 저장한다. 기울기의 지수 평균과 기울기의 제곱 값의 지수 평균은 다음과 같이 나타낸다.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla_\theta J(\theta) \quad (12)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta J(\theta))^2 \quad (13)$$

위 수식과 같이 m과 v가 계산되지만, m과 v는 학습 초기에 0으로 초기화 되어 있기 때문에 학습의 초반부에 m_t, v_t가 0에 가깝게 bias가 설정되어 있기 때문에 이를 unbiased 하게 만들어주는 작업을 거친다. 다음과 같은 보정을 통해 unbiased된 값을 통하여 기울기가 들어갈 자리에 $\widehat{m_t}$,G_t가 들어갈 자리에 $\widehat{v_t}$를 넣어 계산을 진행한다.

$$\widehat{m_t} = \frac{m_t}{1-\beta_1^t} \quad (14)$$

$$\widehat{v_t} = \frac{v_t}{1-\beta_2^t} \quad (15)$$

$$\theta = \theta - \frac{\eta}{\sqrt{\widehat{v_t}+\epsilon}}\widehat{m_t} \quad (16)$$

m과 v를 보정하기 위하여 사용되는 β_1는 일반적으로 0.9, β_2는 0.999로 사용한다. 또한 ϵ는 보정된 $\widehat{v_t}$ 값이 0으로 수렴하는 것을 막기 위하여 10^{-8} 정도의 아주 작은 값을 사용한다.

Adam은 RMSProp와 Momentum의 방법을 동시에 이용하기 때문에 현재 Neural Network 학습에서 많이 사용되는 Optimizer 이며, Adam은 Adagrad와 마찬가지로 learning rate decay에 대한 고려가 필요 없다. 그 이유는 learning rate를 1/2씩 exponential 하게 줄여주는 기능을 수행하고 있기 때문이다. 실제 학습 결과에서도 자동으로 learning rate decay에 대한 고려가 들어간 학습의 최적화 알고리즘이 더 나은 결과를 보여주기 때문이다.

III. MNIST를 이용한 OPTIMIZER 비교 환경

본 장에서는 2장의 관련연구에서 소개한 Gradient Descent, Momentum, Adagrad, Adadelta, RMSProp, Adam Optimizer 들을 비교 분석한다. 학습을 위한 입력 데이터로 기본적으로 많이 이용하는 MNIST 데이터를 사용한다. MNIST 데이터는 아래 그림과 같이 손으로 쓰여진 숫자이다. 해당 손으로 쓰여진 숫자들은 이미지로 구성이 되어 있으며, 각 데이터의 숫자를 원-핫 벡터로 라벨을 만들어 제공하고 있다.

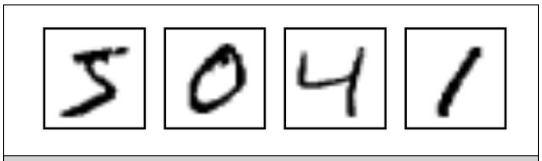

그림 1. MNIST 데이터 예시

해당 Optimizer들의 비교를 위하여 MNIST 데이터의 학습을 위한 모델로 Neural Network 에서 주로 이용되는 Convolution Neural Network (CNN)을 이용한다. CNN은 지역 수용 영역(Local Receptive Fields)과 컨볼루션 레이어(Convolution Layer), 풀링 레이어(Pooling Layer)와 같이 세 단계로 구성되어 있는 Neural Network 로서, 일반적으로 이미지 분석에 대표적으로 쓰이는 딥 러닝 기

법이다. MNIST 데이터의 손 글씨 이미지는 N×N의 형태로써 지역 수용 영역에 (−1, N, N, 1)의 모양으로 입력이 된다. 이 후 입력 데이터는 컨볼루션 레이어를 거치고 샘플링(Pooling) 과정도 같이 거치게 되며 하나의 히든 레이어를 거치게 된다. 이를 통해 생성된 feature들은 다음 히든 레이어의 입력으로 사용되게 된다. 최종적으로 Fully Connected를 통과하여 분류의 과정을 거치게 된다. 이를 통해 학습된 결과값과 실제 손 글씨의 결과값의 차이를 계산하는 loss 함수를 거침으로써 모델을 학습 하게 된다.

CNN을 통해 나온 학습의 결과값과 실제 결과값의 차이를 계산하는 loss 함수의 값을 최소화 시킬 수 있도록 하는 네트워크 파라미터는 Optimizer를 반복적으로 거치면서 결정하게 된다. 이를 위해 2장의 관련연구에서 소개한 6개의 Optimizer들이 많이 사용되며, 그 중 RMSProp와 Adam Optimizer가 최근 기본으로 사용되고 있다. 본 논문에서는 각 6개의 Optimizer들의 비교를 위하여 Optimizer 에 따른 MNIST의 CNN 학습 결과를 비교한다. 또한 각 Optimizer의 learning rate에 따른 MNIST의 CNN 학습 결과를 비교한다.

각 Optimizer에 따른 MNIST의 CNN 학습 결과를 비교하기 위하여 CNN학습 결과의 정확도를 비교한다. 이를 위하여 소규모와 중규모의 CNN네트워크를 구성하여 MNIST를 학습한다. 소규모의 경우 히든 레이어는 2개로 구성하며, 컨볼루션 레이어에서 샘플링을 하기 위한 필터의 사이즈는 3×3 으로 구성 한다. 총 필터의 개수는 32개이며, padding은 입력 값과 출력 값의 크기를 유지하기 위하여 SAME을 사용한다. Stride는 필터의 이동 크기를 나타내며 해당 비교를 위하여 (1, 2, 2, 1)을 사용하였다. 중규모의 CNN 모델의 경우에는 히든 레이어의 개수는 4개로 구성하며, 필터의 사이즈는 7×7로 구성하며, 총 필터의 개수는 64개이다. Padding과 Strides는 소규모와 동일하게 유지한다.

IV. OPTIMIZER 비교 분석

본 장에서는 위에서 소개한 6개의 Optimizer를 비교하기 위하여 MNIST 손 글씨 데이터를 CNN모델을 통하여 학습한 정확도를 비교한다. 이를 위하여, 각 Optimizer에 따른 MNIST의 CNN 학습 결과의 정확도 비교와 각Optimizer의 learning rate에 따른 MNIST의 CNN학습 결과의 정확도를 비교한다.

4.1. Optimizer에 따른 MNIST의 CNN 학습 결과 비교

비교를 위하여 각 Optimizer의 learning rate를 동일하게 0.001로 유지하고 CNN 학습 모델은 소규모와 중규모의 네트워크를 구성하여 MNIST를 학습한다. 소규모의 경우 히든 레이어는 2개로 구성하며, 컨볼루션 레이어에서 샘플링을 하기 위한 필터의 사이즈는 3×3 으로 구성 한다. 총 필터의 개수는 32개이며, padding은 입력 값과 출력 값의 크기를 유지하기 위하여 SAME을 사용한다. Stride는 필터의 이동 크기를 나타내며 해당 비교를 위하여 (1, 2, 2, 1)을 사용하였다. 중규모의 CNN 모델의 경우에는 히든 레이어의 개수는 4개로 구성하며, 필터의 사이즈는 7×7로 구성하며, 총 필터의 개수는 64개이다. Padding과 Strides는 소규모와 동일하게 유지한다. CNN 모델의 학습의 트레이닝 Epoch는 100으로 설정하였다. 이를 바탕으로 Gradient Descent, Momentum, Adagrad, Adadelta, RMSProp, Adam Optimizer들을 비교한다.

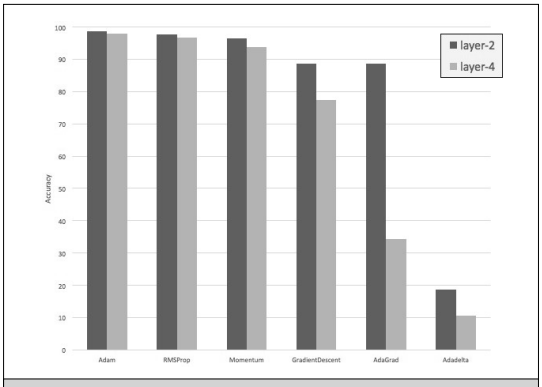

그림 2. Optimizer에 따른 MNIST의 CNN 학습 결과 비교

위 그림 2는 각 Optimizer에 따른 MNIST의 CNN 학습 결과를 보여준다. Layer-2는 히든 레이어의 개수를 나타내며, Layer-2는 소규모의 CNN모델을 의미하며, Layer-4는 히든 레이어 개수가 4개이며, 중규모의 CNN모델을 의미한다. 그림 2의 비교 결과에서 보이듯이 일반적으로 많이 쓰이고 있는 RMSProp와 Adam Optimizer가 가장 높은 정확도를 보이고 있다. RMSProp, Momentum 와 Adam이 높은 정확도를 보이고 있다. 3 개의 Optimizer는 정확도가 낮은 3개의 Optimizer들을 보완한 최적화 알고리즘이기 때문이다. 또한 소규모 네트워크가 중규모 네트워크 보다 정확도가 높은 경향을 보이고 있다. 그 이유는 중규모의CNN 네트워크의 경우, 히든 레이어의 개수가 많아지면서 현재 학습되고 있는 MNIST 데이터에 오버피팅이 되면서 오히려 정확도가 떨어지고 있다.

4.2. Optimizer의 learning rate에 따른 MNIST의 CNN 학습 결과 비교

본 절에서는 Optimizer의 learning rate에 따라 MNIST의 CNN 학습 결과인 정확도 비교를 수행한다. 비교 평가를 위하여 learning rate의 값을 0.0001, 0.0005, 0.001, 0.005, 0.01로 설정한 후 소규모의 CNN을 이용하여 MNIST 데이터에 대한 학습을 진행 하였다. 소규모의 CNN은 2개로 구성하며, 컨볼루션 레이어에서 샘플링을 하기 위한 필터의 사이즈는 3×3 으로 구성 한다. 총 필터의 개수는 32개이며, padding은 입력 값과 출력 값의 크기를 유지하기 위하여 SAME을 사용한다. Stride는 필터의 이동 크기를 나타내며 해당 비교를 위하여 (1, 2, 2, 1)을 사용하였다. 학습 트레이닝의 Epoch는 100으로 설정하였으며 MNIST 데이터의 학습결과에 대한 정확도를 비교의 기준으로 놓고 비교 평가를 수행하였다.

위 그림 3은 learning rate에 따른 소규모 CNN 에서 학습 결과의 정확도를 나타낸 그래프이다. 전체적으로 보았을 때, Adam 과 RMSProp 가 가장 정확도가 높으며, Adadelta의 경우 가장 정확도가 낮은 것을 볼 수 있다. 하지만, Adam과 RMSProp의 경우 learning rate의 값이 증가함에 따라 정확도가 내려가는 것을 알 수 있다. 그 이유는 그림 4와 5를 통해 알 수 있다. learning rate의 경우 네트워크 파라미터의 최적화 값을 찾기 위한 Optimizer의 step size로 볼 수 있다. 따라서 learning rate가 커질 경우 그림 4와 같이 빠르게 최적의 네트워크 파라미터를 찾은 이후에 오버피팅이 발생하기 때문에 정확도

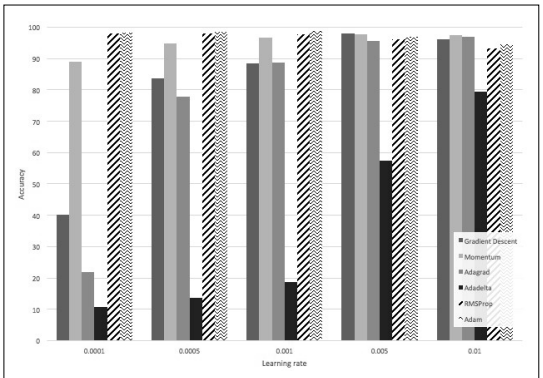

그림 3. Optimizer의 learning rate에 따른 MNIST의 CNN 학습 결과 비교

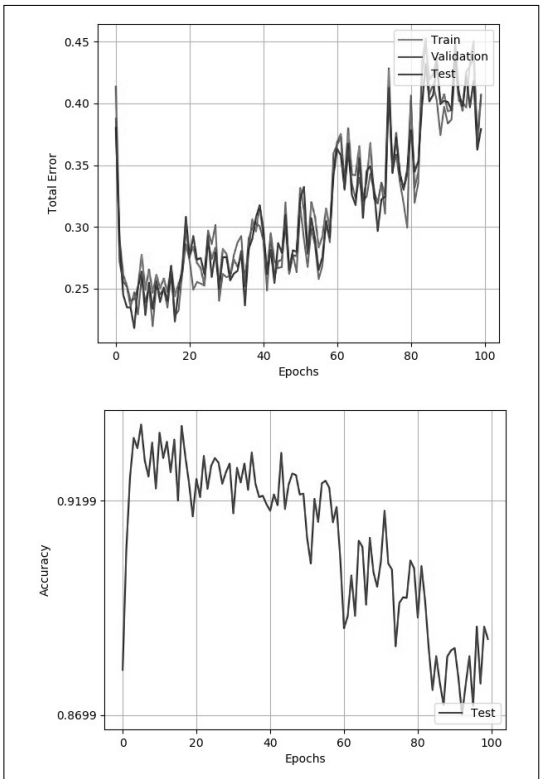

그림 4. Adam Optimizer의 Learning rate 0.01에 대한 실험 결과

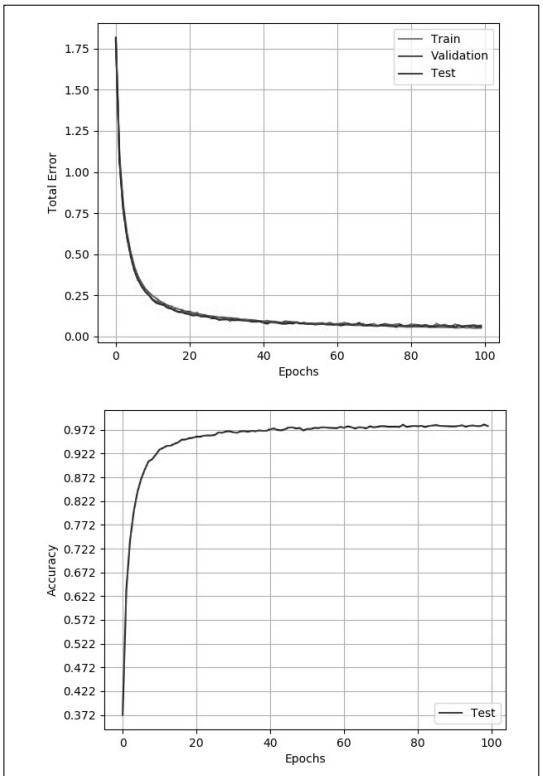

그림 5. Adam Optimizer의 Learning rate 0.0001에 대한 실험 결과

가 떨어지는 경우가 발생한다. 이와 반대로 learning rate가 작아지면, step size가 작아지면서 큰 경우와 다르게 천천히 최적화된 네트워크 파라미터를 찾기 때문에, 계속적으로 정확도가 증가하는 것을 볼 수 있으며, 오히려 오버피팅이 발생하지 않게 된다. 이와 마찬가지로 RMSProp의 경우에도 learning rate 값의 크기에 따라 낮을 수록 오버피팅이 발생하게 될 확률이 높고, 클 수록 오버피팅이 발생하지 않아 정확도가 계속 증가하게 된다.

VI. 결론

본 논문을 통하여 주로 사용되고 있는 Optimizer의 성능과 효율을 비교할 수 있었다. 이를 위하여 MNIST데이터를 활용하여 CNN를 구성하고 각 Optimizer 별로 정확도를 비교하였으며, 단순한 Neural Network를 구성하여 각 Optimizer의 학습 속도를 비교 하였다. 위 비교 평가에서는 CNN의 학습 결과는 Adam 을 사용하는 경우가 가장 결과가 좋았으며, 단순한 Neural Network의 경우에는 Adagrad가 더 좋은 성능을 보였다. 이를 통하여 입력 데이터 값과 가중치, bias의 초기값과 같은 하이퍼파라미터의 초기값에 따라 Optimizer의 성능이 다른 것을 확인 할 수 있다. 따라서 데이터의 학습을 진행할 경우 다양한 Optimizer의 비교를 통하여 해당 데이터와 해당 Neural Network에 맞는 Optimizer를 사용하여야 한다.

참고 문헌

[1] Jürgen Schmidhuber, “Deep learning in neural networks: An overview,” Neural Networks, vol. 61, pp. 85–117, January 2015

[2] Sebastian Ruber, “An overview of gradient descent optimization algorithms,” arXiv, September, 2016

[3] https://www.TensorFlow.org/

[4] http://yann.lecun.com/exdb/mnist/

[5] https://www.TensorFlow.org/api_docs/python/tf/train/GradientDescentOptimizer

[6] Matthew D. Zeiler, “ADADELTA: AN ADAPTIVE LEARNING RATE METHOD,” arXiv.org, December, 2012

[7] John Duchi, Elad Hazan, Yoram Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,” Journal of Machine Learning Research, vol. 12, pp. 2121–2159, 2011

[8] Ilya Sutskever, James Martens, George Dahl, Geoffrey Hinton, “On the importance of initialization and momentum in deep learning,” JMLR:W&CP, vol. 28, 2013

[9] Qian, N, “On the momentum term in

gradient descent learning algorithms," Neural Networks : The Official Journal of the International Neural Network Society, vol. 12, pp. 145 - 151, Jan, 1999

[10] Diederik P. Kingma, Jimmy Ba, "Adam: A Method for Stochastic Optimization," arXiv.org, December, 2014

[11] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, Jeremy Kubica, "Ad Click Prediction: a View from the Trenches," ACM, August, 2013

[12] John Duchi, Yoram Singer, "Efficient Learning using Forward-Backward Splitting," nips, 2009

[13] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf