Computer Vision Assignment 3

Fashion MNIST Classification

2017147505 컴퓨터과학과 김호진

<목차>

1. 개발 환경
2. Network Architecture Description
3. Concept and the function of loss function and optimizer
4. 결과 및 분석
5. Fully Connected Network 와 LeNet-5의 차이점
6. 하이퍼 파라미터의 조정에 따른 영향 분석

1. 개발환경

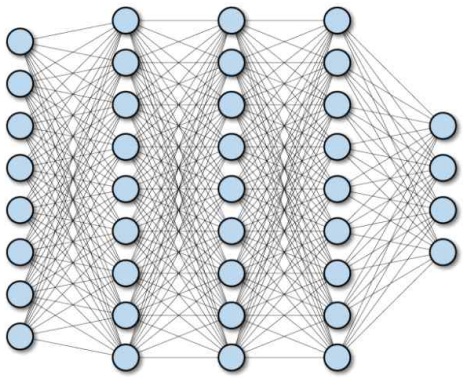
OS - MacOS 11.2.2
Google Colaboratory 사용
Python version : 3.7.10
Torch version : 1.8.1+cu101
cuda version : 10.1

2. Network Architecture Description

해당 과제에서는 Fully Connected Network와 LeNet-5를 구현하고 분석했다. 각각의 네트워크에 대한 아키텍처 설명은 아래와 같다.

가. Fully Connected Network

Fully Connected Network 는 한층의 모든 노드들이 다음층의 모든 노드(뉴런)들과 각각 완전하게 연결된 네트워크 모형이다. 1차원 배열의 형태로 평탄화된 행렬을 이용해 이미지를 분류하는 기능을 수행한다.



이미지 출처 : https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html

Input Layer와 Hidden Layer, Output Layer로 구성된다. 이 때 Network의 깊이와

Hidden layer의 노드의 수는 임의로 지정가능하다.
각 layer사이에서 일어나는 연산은 아래와 같다.
1) 우선 이미지데이터를 벡터화를 통해 1차원 배열로 변환하여 입력으로 받아들인다.

2) 각각의 노드에 각기 다른 가중치(w)와 공유되는 bias(b)를 적용하여 Input(X)에 대한 output을 계산하고 이에 activation function을 적용하여 뉴런을 활성화 한 뒤 다음 노드에 전달한다.

output = wX + b -> activation function

Activation function은 가중치와 bias를 사용하여 값을 구한뒤 다음 계층으로 전달할 때, 특정 조건을 만족하게 되면 활성화되었다는 신호를 다음노드로 보내서 뉴런을 활성화 하고, 조건을 만족하지 못했으면 해당 뉴런을 비활성화 하는 함수이다. 다양한 activation function이 있지만 일반적으로 relu, sigmoid, tanh를 사용한다. 해당과제에서는 ReLU 함수를 사용했다.

3) output layer에서는 softmax라는 활성화함수를 사용해서 최종 예측 값을 결정한다. softmax는 다중분류에서 특히 사용되는 활성화함수이다. softmax는 출력값으로 다중 분류에 대해서 각각의 확률을 내놓는다. softmax 함수는 K개의 값이 존재할 때 각각의 값의 편차를 확대시켜 큰 값은 상대적으로 더 크게, 작은 값은 상대적으로 더 작게 만든 다음에 normalization 시키는 함수이다. 내부의 연산은 아래와 같고 따라서 이때 확률의 총합은 1이

$$p_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} = \frac{e^{x_j}}{e^{x_1}+e^{x_2}+\cdots+e^{x_K}} \quad for\ j\ =\ 1,\ldots,K$$

된다. 최종적으로 softmax를 거친 K개의 값들은 상대적인 중요도를 나타내는 K개 값으로 바뀌고 총합이 1이기 때문에 각각에 대한 확률로 해석된다. 이러한 특성으로 인해 Network의 마지막 레이어에서 다중분류를 통한 클래스 예측에 활용된다. 이렇게 결과 중 가장 확률이 높게 나온 클래스가 최종 예측 클래스가 된다.

4) 하지만 같은 클래스로 예측하더라도 [0.6,0.4]와 [0.9,0.1]은 다른 정확도를 가진다. 따라서 예측값과 실제값을 이용해서 Loss function으로 Loss를 계산할때 사용하는 예측값은 가장 확률이 높은 클래스에만 True값을 부여한 뒤가 아니라 각각의 확률을 가진 값 그자체이다. 이 예측값과 실제값을 loss function, 해당과제에서는 crossentropyloss, 에 넣어서 loss를 구한다.

5) 이 Loss를 기반으로 역전파를 통해 실제값에 가까운 예측값이 구해지도록 각 가중치와 bias를 업데이트한다. 역전파를 통해서 얻은 값으로 실제 파라미터를 업데이트하는데에는 optimizer가 관여하는데 해당 과제에서는 Adam을 사용했다.

나. LeNet-5

LeNet은 Convolutional Neural Network의 발전 버전이다.
LeNet은 다차원 배열 데이터 처리에 특화된 모델 이다. 데이터의 공간적 정보를 유지하면서

배열 데이터 정보를 다음레이어로 보낼 수 있어서 이미지분야에서 적극 활용되고 있다.
하위층에서 선, 색과 같은 특징을 검출했다면 층이 깊어질수록 물체의 특징을 세부적으로 검출한다.

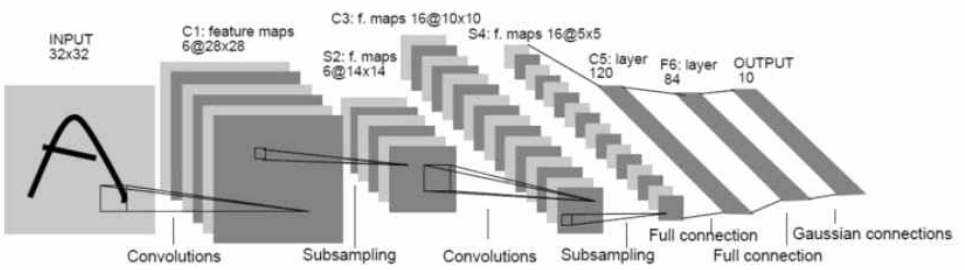


사진 출처 : Gradient-Based Learning Applied to Document Recognition (1998, Yann LeCun)

사진에서 확인할 수 있듯이 LeNet5는 Input, 3개의 Convolutional Layer, 2개의 Subsampling layer, 1층의 Fully connected 레이어 그리고 output layer로 구성되어 있다.
Convolution과 Pooling layer에서는 특징을 추출하고 마지막 Fully connected layer에서 분류를 진행한다.
각 레이어에서 일어나는 연산은 아래와 같다.

1) C1 layer : Padding을 통해 32 x 32로 변환된 Image를 Input으로 받아들인다. 6개의 5x5 kernel을 사용해서 stride = 1으로 각 픽셀들에 대해 convolution연산을 해준다. 따라서 feature map은 각 border가 2만큼 줄어든 6개의 28x28 배열을 갖는다.
이 때 output feature map을 구하는데 input을 전부 사용하지 않고 fileter 범위내의 픽셀들끼리만 계산하기 때문에 local한 특성을 추출해낸다는 특징이 있다.
이 필터를 통과한 이미지 픽셀값은 연산에 의해 변환된다. 이 과정에서 변환된 이미지들은 색상, 선, 형태, 경계 등 특징이 뚜렷해진다. Convolution연산을 진행할수록 이미지 크기는 작아지고 채널 수는 필터의 수에 따라 증가한다.

이후 activation function으로 Tanh(hyperbolic tangent)를 적용하고 출력한다.
Tanh 또는 hyperbolic tangent함수는 쌍곡선 함수이고 시그모이드의 변형된 형태이고, 출력의 중심이 0이아닌 시그모이드의 단점을 개선한 함수이다. 출력의 중심이 0이 아니면 다음 레이어가 받는 입력 데이터가 zero-centered가 아니게되고 이는 LeNet5의 목적과도 맞지 않다.

2) S2 layer : 2x2 kernel을 stride=2로 설정하여 6장의 feature map에 Sub sampling을 진행한다.
사용가능한 Pooling 방법은 Kernel범위내의 가장 큰값을 선택하는 Max pooling과 Average Pooling이 있는데 LeNet-5에서는 Average Pooling을 사용한다. Subsampling을 진행한 뒤 채널의 크기는 고정되고 이미지의 크기는 14x14로 줄어든다. Convolution layer를 통해서 어느정도 feature extraction이 진행되었으면 이 모든 특징을 분류에 활용할 필요는 없다. 따라서 Subsampling을 진행해주고 이는 topological information을 그대로 유지하면서 데이터의 크기를 축소시켜 이후 연산을 진행하는데 필요한 파라미터의 수를 줄이기 때문에

overfitting이 발생하는 것을 방지해준다. 그러므로 네트워크의 전반적인 성능과 정확도를 증가시켜준다.
 다만 이로인해 발생하는 손실은 feature map size가 작아진 만큼 더 많은 filter연산이 가능해지는 이점으로 보완한다.

3) C3 layer : 5x5 kernel을 stride=1로 설정하여 Convolution을 진행한다. 14x14x6의 Input이 10x10x16의 Output으로 출력된다. 이 때 앞의 Convolutional Layer와 같이 6개의 feature map이 전부 16개의 output을 구하는 연산에 반영되는 것이 아니라 아래 사진과 같이 하나의 10x10 feature map을 구하는데 선택적으로 input을 사용한다.
연속적인 3장을 6번, 연속적인 4장을 6번, 불연속적인 4장을 3번, 전체 입력에 대해서 1번의 연산으로 총 16개의 output feature map을 구하는 것으로 해석할 수 있다. 이렇게 하나의 feature map연산에 모든 입력을 활용하지 않고 선택함으로써 얻을 수 있는 이점을 논문에서는 아래와 같이 설명했다.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I
EACH COLUMN INDICATES WHICH FEATURE MAP IN S2 ARE COMBINED BY THE UNITS IN A PARTICULAR FEATURE MAP OF C3.

첫째로, reasonable한 bound범위 내에서 연결의 수를 줄여서 유지한다.
두 번째로, 첫 번째 이유보다도 더 중요하게, network의 symmetry를 깨뜨린다. 서로 다른 featur map을 input으로 사용함으로써 output feature map들은 서로 각기 다른 feature를 추출해낼 수 있게된다.

4) S4 layer : 2x2 kernel을 stride=2로 설정하여 16장의 feature map에 Sub sampling을 진행한다. 이미지의 크기는 5x5로 줄어든다.

5) C5 layer : 120개의 5x5x16 kernel으로 convolution을 진행한다. 따라서 1x1x120의 output을 출력한다. 이렇게 local feature를 얻은 다음 다시 convolution과 sub sampling을 진행함으로써 global한 feature를 얻을 수 있다.

6) F6 layer : 앞서 살펴보았던 Fully Connected Network와 비슷한 구조를 가진 Layer이다. flatten된 1차원 배열 데이터를 입력으로 받아들이고 가중치와 bias로 연산을 진행한다. 다만 활성화 함수로 tanh(Hyperbolic tangent function)를 사용한다는 점에서 다르다. 출력 유닛은 84개이다.

7) F7 layer : 84개의 유닛을 입력으로 받아 MNIST 데이터의 클래스 개수에 해당하는 10개의 output을 출력한다. 활성화함수로 softmax를 사용해서 분류를 완성한다.

8) loss function으로 Loss를 계산한다. 해당 과제에서는 CrossEntropyLoss를 사용했다.

9) 역전파와 optimizer를 사용해서 파라미터를 업데이트하면서 학습한다. 해당 과제에서는 SGD를 사용했다.

요약하자면 receptive field에 shared weights를 가진 filter로 convolution을 적용하고 sub-sampling을 통해서

3. Concept and the function of loss function and optimizer

해당 과제의 구현을 위해서 Loss function으로는 CrossEntropyLoss function을 사용했다.

Pytorch의 library에서 가져와서 사용한 CrossEntropyLoss의 경우 Softmax함수와 Negative log likelihood를 합쳐놓은 것과 같은 기능을 한다. 따라서 Softmax를 거치지 않은 결과를 바로 Loss function에 넣어주었다.

Negative Log likelihood란 말 그대로 입력값과 파라미터가 주어졌을 때 결과가 우리가 원하는 결과와 같을 확률(likelihood)에 log를 취해주고 negative scale을 적용시키는 것을 말한다. 확률은 softmax함수를 통해서 출력되는 값이며 log를 likelihood에 취해주면 log라는 특성상 학습이 잘 될수록 loss가 증가하게 된다. 이를 음수로 바꿔주어서 학습이 잘 될수록 loss가 감소하는 출력을 얻을 수 있다. log를 사용한 이유는 0에 가까워 질수록 값이 급격하게 커져서 더 큰 parameter의 변화를 유도할 수 있기 때문으로 추측했다. softmax의 경우 위에서 설명한 바와 같다.

Loss function

CrossEntropyLoss : Cross Entropy는 분류(Classification) 문제에서 대표적으로 사용되는 손실함수이다. Cross Entropy Loss를 구하는 식은 아래 그림과 같다.

$$H_p(q) = -\sum_{c=1}^{C} q(y_c)\log(p(y_c))$$

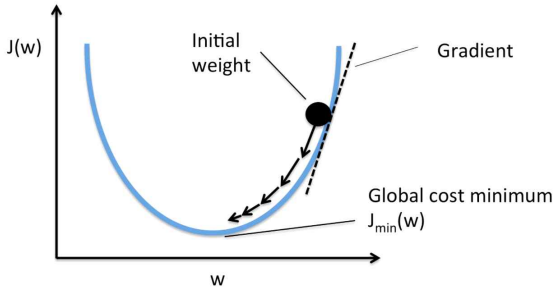
예측값에 대해서 실제 값과 비교해 엔트로피를 계산해서 가까운 경우는 0으로 수렴하고 차이가 클 경우 값이 커진다. 이 때 예측값p 와 실제값q 가 둘 다 연산에 사용되기 때문에 cross라는 명칭이 붙었다. 조건은 classification에서 사용하는 인코딩 방법으로 출력의 형태가 정답 label만 1이고 나머지는 0인 one-hot encoding형태여야 한다는 점이다. 해당 과제에서 사용한 Fashion MNIST dataset의 경우 요구조건에 부합하기 때문에 사용할 수 있었다.

해당 과제의 구현을 위해서 optimizer로는 SGD를 사용했다.

Optimizer

SGD : Stochastic Gradient Descent - 확률적 경사 하강법 으로 현재 구해진 Loss에 대해서 Back propagation을 통해 Gradient를 구하고 그것에 일정한 learning rate를 곱한

뒤 기존 parameter에 적용해주는 방법이다. Loss를 줄이는 것이 가장 큰 목표로 gradient가 점점 감소하도록 값을 이동시켜서 최솟값에 도달하고자 하는 방식이다. 다음 도식으로 설명가능하다.
기존 Gradient Descent 방식과 다른 점은 Batch size씩 나누어진 것에 더하여 데이터마다 바로바로 역전파와 학습을 진행하여서 더 자주 Loss에서 얻어낸 Gradient를 Parameter에 적용시켜줌으로써 빠른 학습이 가능하도록 했다는 점이다.



SGD자체는 현재 시점에서 최적의 optimizer라고 볼 수는 없다. 최소점이 아닌 극소점에서 gradient가 0이 되어버린다면 더 이상 학습을 진행하지 않는 문제도 있고 학습속도 자체도 발전된 optimizer가 많이 고안되었다. 하지만 LeNet-5 Architecture의 정의에서 SGD를 사용하는 것으로 기술되어있기 때문에 SGD로 구현하였다. 아마 1998년 LeNet5를 처음 고안해냈을 시기에는 SGD가 가장 뛰어난 optimizer였을 것으로 예상한다.

4. 결과 및 분석

우선 LeNet-5의 결과이다.

```
Epoch [1/5], Step [500/1500], Loss: 2.2503524
Epoch [1/5], Step [1000/1500], Loss: 1.1639609
Epoch [1/5], Step [1500/1500], Loss: 0.4696307
Epoch [2/5], Step [500/1500], Loss: 0.7160317
Epoch [2/5], Step [1000/1500], Loss: 0.3966402
Epoch [2/5], Step [1500/1500], Loss: 0.7360960
Epoch [3/5], Step [500/1500], Loss: 0.2675364
Epoch [3/5], Step [1000/1500], Loss: 0.2906428
Epoch [3/5], Step [1500/1500], Loss: 0.6325350
Epoch [4/5], Step [500/1500], Loss: 0.3555644
Epoch [4/5], Step [1000/1500], Loss: 0.1398335
Epoch [4/5], Step [1500/1500], Loss: 0.1731055
Epoch [5/5], Step [500/1500], Loss: 0.2663147
Epoch [5/5], Step [1000/1500], Loss: 0.1011809
Epoch [5/5], Step [1500/1500], Loss: 0.0774221
Accuracy of the network on the 10000 test images: 95.04 % when batch_size is 40, learning_rate is 0.01
```

나타나있듯이 batach size로는 40, learning rate로는 0.01을 사용했다.
Loss function으로는 CrossEntropyLoss, optimizer로는 SGD를 사용했다. loss function과 optimizer는 pytorch안에 내장된 library를 활용하여 구현하였다.
train에서는 5번의 epoch를 주었다.
epoch가 진행될수록 Loss값이 점점 작아지면서 0에 수렴하고 있는 것이 보이기 때문에 학습이 정상적으로 진행되었다고 볼 수 있다.

다음은 Fully Connected Network의 결과이다.
LeNet과 동일하게 비교하기 위하여 batach size로는 40, learning rate로는 0.01을

```
Epoch [1/5], Step [500/1500], Loss: 1.1065
Epoch [1/5], Step [1000/1500], Loss: 0.4410
Epoch [1/5], Step [1500/1500], Loss: 0.4183
Epoch [2/5], Step [500/1500], Loss: 0.3405
Epoch [2/5], Step [1000/1500], Loss: 0.2348
Epoch [2/5], Step [1500/1500], Loss: 0.2294
Epoch [3/5], Step [500/1500], Loss: 0.5200
Epoch [3/5], Step [1000/1500], Loss: 0.2596
Epoch [3/5], Step [1500/1500], Loss: 0.2357
Epoch [4/5], Step [500/1500], Loss: 0.2074
Epoch [4/5], Step [1000/1500], Loss: 0.4019
Epoch [4/5], Step [1500/1500], Loss: 0.3018
Epoch [5/5], Step [500/1500], Loss: 0.4191
Epoch [5/5], Step [1000/1500], Loss: 0.2980
Epoch [5/5], Step [1500/1500], Loss: 0.2895
Accuracy of the network on the 10000 test images: 92.68 %
```

사용했다. Loss function으로는 CrossEntropyLoss, optimizer로는 SGD를 사용했다. train에서는 5번의 epoch를 주었다.

epoch가 진행될수록 Loss값이 점점 작아지면서 0에 수렴하고 있는 것이 보이기 때문에 학습이 정상적으로 진행되었다고 볼 수 있다.

동일한 조건하에서 Fully Connected Network의 accuracy가 약 2.5%정도 낮게 나온 것을 확인할 수 있다. 현재의 batch size와 learning rate, Loss function. optimizer등은 최적화시킨 값들이 아니라 임의로 설정해둔 값이며 최적화된 값은 6번에서 찾아서 적용할 예정이다.

5. Fully connected Network 와 LeNet-5의 차이점

Lenet-5의 마지막 두 개의 레이어도 Input과 1개의 hidden, 1개의 output layer로 이루어진 fully connected layer이다. 다만 가장 큰 차이점은 해당 레이어에 이미지를 넣어 분류를 진행하기전에 lenet5은 추가적인 과정을 거친다는 점이다.

이미지를 벡터화 시킨 입력을 받는 FCN과 달리 LeNet-5는 우선 2차원의 이미지 그상태로 입력으로 받는다. 1차원 배열을 Input으로 받는다는 점에서 Fully Connected Network는 Input의 구조에 구애받지 않는다는 장점을 가진다. 입력의 구조에 크게 구애받지 않기 때문에 좀 더 광범위한 문제에 적용가능하다. 이미지는 특성상 인접한 pixel간의 상관관계가 중요한 정보를 가질 확률이 높다. FCN은 데이터의 topological information을 무시해버린다. 하지만 LeNet5는 Convolution과 Subsampling 과정에서도 topology를 유지하기 때문에 2차원 이미지의 연산에 관해서는 더 좋은 네트워크라고 할 수 있다.

LeNet5는 Fully Connected Layer로 분류를 진행하기 전에 3번의 Convolution과 2번의 Pooling layer를 거치면서 Featrue Extraction이 이루어진다. 이 과정에서 Convolution으로 특징을 추출하고 Pooling으로 필요한 정보만을 엄선하고 feature extractor에서 자체적인 학습 또한 이루어진다. 따라서 Image를 바로 분류에 이용하는 Fully Connected Network보다 더 좋은 결과를 출력해낼 가능성이 높아진다.

LeNet5는 Fully Connected Network가 모든 노드 간의 연결에 다른 가중치를 사용해야하는 것과 달리 Convolutional Layer에서 하나의 convolution필터로 같은 피처맵의 유닛들은 5x5개의 가중치와 1개의 bias를 공유하여 연산에 이용한다. 즉, 많은 파라미터를 사용하지 않고 효율적으로 연산을 진행한다. 따라서 LeNet5를 사용하게 되면 Fully Connected Network를 사용할 때 보다 구해야하는 파라미터의 수가 줄어든다. 이렇게 되면 요구되는 연산을 줄여주고 학습해야하는 파라미터 수가 줄어듬으로써 오버피팅을 방지하여 training data과 test data간의 gap을 줄일 수 있다.

실제로 LeNet5의 Architecture는 논문에서 제시된 대로 고정되어있기 때문에 parameter의 수를 계산해보면 60,000개라는 결과를 얻을 수 있다. 이 때, Average Pooling은 커널 범위안의 평균값을 전달하는 역할만을 하지만 실제로는 이후에 1개의 가중치가 곱해지고 1개의 bias가 더해져서 전달된다. 이 또한 parameter이다.
28*28이미지를 입력으로 사용하는 Fully Connected Network의 경우 hidden layer와 유닛의 수에 따라 parameter수가 급격히 늘어난다.

6. 하이퍼 파라미터의 조정에 따른 영향 분석

Network의 Architecture를 그대로 유지하면서 분석을 위해 조정한 Hyper parameter는 아래와 같다.
1. Optimizer parameter - Learning rate
2. Optimizer - SGD 이외의 다른 optimizer들
3. batch size
4. loss function
5. activation function
6. Pooling 방법

1) 우선 가장 빠르게 파악이 가능하고 영향력이 큰 Hyper parameter는 batch size와 learning rate라고 판단했다. 이외의 Hyper parameter들은 위의 LeNet-5 Network Architecture에서 언급한대로 유지하면서(SGD, CrossEntropyLoss, Average Pooling, Activation function으로 Tanh 사용)
batch size와 learning rate를 변경해가며 결과를 확인해보았다.

batch size는 전체 학습 데이터셋을 여러개의 작은 그룹으로 나누었을 때 하나의 그룹에 속하는 데이터 수를 의미한다. SGD를 optimizer로 사용했기 때문에 매 iteration마다 mini-batch에 대한 gradient를 구해서 parameter 업데이트에 사용한다. batch size가 클수록 한번의 gradient계산을 위해서 더 많은 크기의 데이터를 사용한다는 뜻이기 때문에 좀 더 정교하게 gradient를 계산할 수 있지만, 그만큼 computation시간이 오래걸릴 것이다. 이는 병렬화의 적용으로 극복이 가능하다. 따라서 실제로는 GPU메모리가 허락하는 내에서 batch size가 커지면 학습이 오히려 빨라지는 효과가 있다.
반면 batch size가 작을수록 한번의 parameter업데이트에 상대적으로 덜 정교한 gradient를

사용하지만, 여러번의 업데이트를 수행하고, 오히려 부정확한 gradient가 여기저기로 튀면서 local minima 혹은 saddle point에서 벗어나게 해줄 수 있다고 추측했다.

learning rate는 optimizer의 parameter로 SGD를 사용함에 있어 너무 큰 경우 무질서하게 발산하며 최저점에 수렴하지 못할 가능성이 높고, 너무 작을 경우 올바른 방향으로 나아가지만 학습시간이 매우 오래 걸리며, local minima에서 빠져나오지 못한다. 따라서 적절한 수치의 learning rate를 주는 것이 중요하다. 이는 경험적으로 얻고자 했다.
batch size를 40, 400, learning rate를 0.1, 0.01, 0.001 로 설정해서 각각 5번의 epoch를 수행하고 test 데이터로 정확도를 구한 결과는 아래와 같다.

```
Accuracy of the network on the 10000 test images: 98.55 % when batch_size is 40, learning_rate is 0.1
Accuracy of the network on the 10000 test images: 94.89 % when batch_size is 400, learning_rate is 0.1

Accuracy of the network on the 10000 test images: 95.04 % when batch_size is 40, learning_rate is 0.01
Accuracy of the network on the 10000 test images: 71.95 % when batch_size is 400, learning_rate is 0.01

Accuracy of the network on the 10000 test images: 68.33 % when batch_size is 40, learning_rate is 0.001
Accuracy of the network on the 10000 test images: 10.31 % when batch_size is 400, learning_rate is 0.001
```

실험에 따르면 batch size가 40, learning rate가 0.1일 때 가장 높은 정확도를 보여주었다. batch size가 작은 경우과 learning rate가 큰 경우 둘 다 위의 사전지식에 따르면 parameter가 다소 큰 변화폭으로 무작위하게 수정되는 역할을 한다고 볼 수 있다. 따라서 이러한 결과는 해당과제에서 사용한 Fashion MNIST Dataset에 local minima 혹은 saddle point가 많이 분포하고 있기 때문이라고 분석했다.

위의 사진에서 batch size가 10배 커질 때와 learning_rate가 10배 커질 때의 정확도가 다소 유사한 점을 확인할 수 있다. 논문[1]에 따르면 scale of random fluctuation, 즉 얼마나 랜덤하게 이리저리 움직이는지는 계산한 수식이

$$g = \frac{\epsilon}{1-m}\left(\frac{N}{B} - 1\right)$$

과 같이 표현되고 (ε은 learning rate, m은 momentum, B는 batch size, N은 train date size) 이 경우 대부분 momentum m은 과거의 가던 방향대로 가는 가중치이기 때문에 0에 가깝고 학습 데이터 크기인 N이 batch size은 B보다 매우 크기 때문에 단순화하여 표현하면

$$g \approx \epsilon N/B.$$

로 볼 수 있다. 따라서 이는 batch size를 키우는 것과 learning rate를 줄이는 것이 동일한 비율의 효과를 낸다는 계산에 다소 부합하는 결과를 얻었기 때문에 적절한 학습이 이루어졌다고 보았다.

batch size가 크고 learning rate가 매우 작을때는 정확도 또한 매우 작게 얻어진 결과를 보아 학습에 따른 parameter의 변화량이 너무 작아서 학습시 Loss의 변화를 거의 일으키지 못한 것으로 보았다.

---

1) Don't Decay the Learning Rate, Increase the Batch Size, Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, Quoc V. Le

```
Accuracy of the network on the 10000 test images: 98.74 % when batch_size is 40, learning_rate is 0.1
Epoch [1/5], Step [500/1500], Loss: 0.3633468
Epoch [1/5], Step [1000/1500], Loss: 0.0841164
Epoch [1/5], Step [1500/1500], Loss: 0.1259023
Epoch [2/5], Step [500/1500], Loss: 0.0105882
Epoch [2/5], Step [1000/1500], Loss: 0.0925758
Epoch [2/5], Step [1500/1500], Loss: 0.1362740
Epoch [3/5], Step [500/1500], Loss: 0.0036459
Epoch [3/5], Step [1000/1500], Loss: 0.0193947
Epoch [3/5], Step [1500/1500], Loss: 0.0034269
Epoch [4/5], Step [500/1500], Loss: 0.0177712
Epoch [4/5], Step [1000/1500], Loss: 0.0216227
Epoch [4/5], Step [1500/1500], Loss: 0.0772986
Epoch [5/5], Step [500/1500], Loss: 0.0761441
Epoch [5/5], Step [1000/1500], Loss: 0.0605781
Epoch [5/5], Step [1500/1500], Loss: 0.0044291
Accuracy of the network on the 10000 test images: 98.55 % when batch_size is 40, learning_rate is 0.2
Epoch [1/5], Step [500/1500], Loss: 0.3795435
Epoch [1/5], Step [1000/1500], Loss: 0.1145087
Epoch [1/5], Step [1500/1500], Loss: 0.2325993
Epoch [2/5], Step [500/1500], Loss: 0.0768273
Epoch [2/5], Step [1000/1500], Loss: 0.1737412
Epoch [2/5], Step [1500/1500], Loss: 0.2205231
Epoch [3/5], Step [500/1500], Loss: 0.2036956
Epoch [3/5], Step [1000/1500], Loss: 0.2181654
Epoch [3/5], Step [1500/1500], Loss: 0.0939499
Epoch [4/5], Step [500/1500], Loss: 0.0827736
Epoch [4/5], Step [1000/1500], Loss: 0.0611237
Epoch [4/5], Step [1500/1500], Loss: 0.0147507
Epoch [5/5], Step [500/1500], Loss: 0.0102683
Epoch [5/5], Step [1000/1500], Loss: 0.0562312
Epoch [5/5], Step [1500/1500], Loss: 0.1086572
Accuracy of the network on the 10000 test images: 98.08 % when batch_size is 40, learning_rate is 0.05
```

learning rate를 더 다양하게 조정해보았지만 0.1일 때의 정확도가 가장 높았다.

이후 batch size와 learning rate를 변화시키면서 activation function은 Tanh, ReLU, Pooling방법은 Max Pooling, Average Pooling을 적용시켰다. Sigmoid는 Tanh와 비슷한 결과가 나올것이라고 추측해 구현하지 않았다. 결과의 정리표는 아래와 같다.

leatning rate	0.1	0.1	0.01	0.01	0.001	0.001	0.1	0.1	0.01	0.01	0.001	0.001
batch size	40	400	40	400	40	400	40	400	40	400	40	400
pool	AVG	AVG	AVG	AVG	AVG	AVG	MAX	MAX	MAX	MAX	MAX	MAX
activation function	Tanh	Tanh	Tanh	Tanh	Tanh	Tanh	Tanh	Tanh	Tanh	Tanh	Tanh	Tanh
optimizer	SGD	SGD	SGD	SGD	SGD	SGD	SGD	SGD	SGD	SGD	SGD	SGD
loss function	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl
정확도	98.62	94.41	94.87	78.97	72.46	10.85	98.81	97.07	97.25	79.77	77.74	10.09

0.1	0.1	0.01	0.01	0.001	0.001	0.1	0.1	0.01	0.01	0.001	0.001
40	400	40	400	40	400	40	400	40	400	40	400
AVG	AVG	AVG	AVG	AVG	AVG	MAX	MAX	MAX	MAX	MAX	MAX
ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU	ReLU
SGD	SGD	SGD	SGD	SGD	SGD	SGD	SGD	SGD	SGD	SGD	SGD
CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl	CrossEntropyl
98.86	95.94	95.47	57.57	27.86	8.75	98.86	97.45	97.29	80.97	19.13	13.09

하지만 [2]activation function과 pooling방법의 조정으로는 최고 정확도의 관점에서 유의미한 변화를 감지하지는 못하였다.
ReLU와 같은 기능을 하면서 훨씬 성능적으로 뛰어난 것으로 고안된 SiLU(Sigmoid Linear Unit)을 사용해 보았지만 유의미한 차이는 없었다.

이미 정확도가 98%이상으로 매우 높은 상태이기 때문에 [3]Data Quality로 인해 야기되는 최대 정확도에 한계선이 존재할 것이라고 예측했고 그에 매우 근접한 상태라고 추측했다. 따라서 activation function, pooling과 같은 방법의 변화로는 최대 정확도에서 변화를 감지하기는 힘들었다. 따라서 batch size가 40, learning rate가 0.1아 아닐때의 정확도들을 보면 Average와 Max Pooling은 큰 차이가 없지만 ReLU와 Tanh의 사이에서 Tanh의 정확도가 훨씬 높게 측정되는 것을 관찰할 수 있었다.

---

2) https://datascience.stackexchange.com/questions/14349/difference-of-activation-functions-in-neural-networks-in-general

3) https://www.aaai.org/Papers/KDD/1995/KDD95-007.pdf

optimizer 는 Pytorch내에 내장된 library에서 가져와서 사용했다. 앞서 돌렸던 결과들은 SGD를 적용하여 얻은 결과이다. SGD이외에 Momentum, NAG, AdaGrad, AdaDelta, RMSProp, Adam

Momentum : Momentum의 가장 큰 장점은 운동량이라는 개념을 적용해 최적점에서 멀리 있을 때는 learning rate를 크게 적용하고, 최소점에 다가갈수록 learning rate를 작게 만들어서 정확도를 빠르게 높인다는 것이다. pytorch에 이를 구현하기 위해서 기존 SGD의 파라미터로 momentum 0.9를 부여했다. default 값은 0이기 때문에 앞서는 기본 SGD를 진행했다고 볼 수 있다.

```
Epoch [1/5], Step [500/1500], Loss: 1.0049490
Epoch [1/5], Step [1000/1500], Loss: 0.5299679
Epoch [1/5], Step [1500/1500], Loss: 0.9313625
Epoch [2/5], Step [500/1500], Loss: 1.6186857
Epoch [2/5], Step [1000/1500], Loss: 1.0708539
Epoch [2/5], Step [1500/1500], Loss: 1.3830293
Epoch [3/5], Step [500/1500], Loss: 2.6784632
Epoch [3/5], Step [1000/1500], Loss: 0.9483143
Epoch [3/5], Step [1500/1500], Loss: 1.6291300
Epoch [4/5], Step [500/1500], Loss: 2.2106771
Epoch [4/5], Step [1000/1500], Loss: 1.9171110
Epoch [4/5], Step [1500/1500], Loss: 3.4876037
Epoch [5/5], Step [500/1500], Loss: 4.1758642
Epoch [5/5], Step [1000/1500], Loss: 2.8702157
Epoch [5/5], Step [1500/1500], Loss: 3.8401630
Accuracy of the network on the 10000 test images: 10.7 % when batch_size is 40, learning_rate is 0.1
Epoch [1/5], Step [500/1500], Loss: 0.2034371
Epoch [1/5], Step [1000/1500], Loss: 0.0378420
Epoch [1/5], Step [1500/1500], Loss: 0.0963054
Epoch [2/5], Step [500/1500], Loss: 0.0527571
Epoch [2/5], Step [1000/1500], Loss: 0.2026849
Epoch [2/5], Step [1500/1500], Loss: 0.2731149
Epoch [3/5], Step [500/1500], Loss: 0.0036548
Epoch [3/5], Step [1000/1500], Loss: 0.0185019
Epoch [3/5], Step [1500/1500], Loss: 0.0901623
Epoch [4/5], Step [500/1500], Loss: 0.0169494
Epoch [4/5], Step [1000/1500], Loss: 0.0518522
Epoch [4/5], Step [1500/1500], Loss: 0.0227424
Epoch [5/5], Step [500/1500], Loss: 0.0218888
Epoch [5/5], Step [1000/1500], Loss: 0.0542537
Epoch [5/5], Step [1500/1500], Loss: 0.0706577
Accuracy of the network on the 10000 test images: 98.52 % when batch_size is 40, learning_rate is 0.01
Epoch [1/5], Step [500/1500], Loss: 2.1719923
Epoch [1/5], Step [1000/1500], Loss: 0.6589602
Epoch [1/5], Step [1500/1500], Loss: 0.4649466
Epoch [2/5], Step [500/1500], Loss: 0.4054695
Epoch [2/5], Step [1000/1500], Loss: 0.5993456
Epoch [2/5], Step [1500/1500], Loss: 0.2564538
Epoch [3/5], Step [500/1500], Loss: 0.1033382
Epoch [3/5], Step [1000/1500], Loss: 0.1076640
Epoch [3/5], Step [1500/1500], Loss: 0.4933292
Epoch [4/5], Step [500/1500], Loss: 0.3879186
Epoch [4/5], Step [1000/1500], Loss: 0.1938955
Epoch [4/5], Step [1500/1500], Loss: 0.2508239
Epoch [5/5], Step [500/1500], Loss: 0.1817718
Epoch [5/5], Step [1000/1500], Loss: 0.2139835
Epoch [5/5], Step [1500/1500], Loss: 0.2264526
Accuracy of the network on the 10000 test images: 94.95 % when batch_size is 40, learning_rate is 0.001
Accuracy of the network on the 10000 test images: 98.66 % when batch_size is 400, learning_rate is 0.1
Accuracy of the network on the 10000 test images: 94.43 % when batch_size is 400, learning_rate is 0.01
Accuracy of the network on the 10000 test images: 69.88 % when batch_size is 400, learning_rate is 0.001
```

RMSprop 적용 -

```
Epoch [1/5], Step [500/1500], Loss: 9.3712206
Epoch [1/5], Step [1000/1500], Loss: 10.2570248
Epoch [1/5], Step [1500/1500], Loss: 12.3213634
Epoch [2/5], Step [500/1500], Loss: 9.8471594
Epoch [2/5], Step [1000/1500], Loss: 10.2857838
Epoch [2/5], Step [1500/1500], Loss: 6.7944002
Epoch [3/5], Step [500/1500], Loss: 10.0178852
Epoch [3/5], Step [1000/1500], Loss: 10.8628283
Epoch [3/5], Step [1500/1500], Loss: 7.8890977
Epoch [4/5], Step [500/1500], Loss: 12.7729969
Epoch [4/5], Step [1000/1500], Loss: 5.9494104
Epoch [4/5], Step [1500/1500], Loss: 17.6412697
Epoch [5/5], Step [500/1500], Loss: 8.6638470
Epoch [5/5], Step [1000/1500], Loss: 6.3438749
Epoch [5/5], Step [1500/1500], Loss: 16.3750038
Accuracy of the network on the 10000 test images: 11.35 % when batch_size is 40, learning_rate is 0.1
Epoch [1/5], Step [500/1500], Loss: 2.3042631
Epoch [1/5], Step [1000/1500], Loss: 2.2731519
Epoch [1/5], Step [1500/1500], Loss: 2.3906856
Epoch [2/5], Step [500/1500], Loss: 2.3243134
Epoch [2/5], Step [1000/1500], Loss: 2.6391950
Epoch [2/5], Step [1500/1500], Loss: 2.4705429
Epoch [3/5], Step [500/1500], Loss: 2.3132474
Epoch [3/5], Step [1000/1500], Loss: 2.3397005
Epoch [3/5], Step [1500/1500], Loss: 2.3867977
Epoch [4/5], Step [500/1500], Loss: 2.3788123
Epoch [4/5], Step [1000/1500], Loss: 2.5723660
Epoch [4/5], Step [1500/1500], Loss: 2.3609550
Epoch [5/5], Step [500/1500], Loss: 2.3365884
Epoch [5/5], Step [1000/1500], Loss: 2.3846555
Epoch [5/5], Step [1500/1500], Loss: 2.3380184
Accuracy of the network on the 10000 test images: 9.74 % when batch_size is 40, learning_rate is 0.01

Epoch [1/5], Step [500/1500], Loss: 0.4868113
Epoch [1/5], Step [1000/1500], Loss: 0.2738885
Epoch [1/5], Step [1500/1500], Loss: 0.2583424
Epoch [2/5], Step [500/1500], Loss: 0.0887739
Epoch [2/5], Step [1000/1500], Loss: 0.0710357
Epoch [2/5], Step [1500/1500], Loss: 0.0851910
Epoch [3/5], Step [500/1500], Loss: 0.0287415
Epoch [3/5], Step [1000/1500], Loss: 0.0224648
Epoch [3/5], Step [1500/1500], Loss: 0.0661969
Epoch [4/5], Step [500/1500], Loss: 0.2431049
Epoch [4/5], Step [1000/1500], Loss: 0.0087673
Epoch [4/5], Step [1500/1500], Loss: 0.0126567
Epoch [5/5], Step [500/1500], Loss: 0.0123925
Epoch [5/5], Step [1000/1500], Loss: 0.0086912
Epoch [5/5], Step [1500/1500], Loss: 0.1827906
Accuracy of the network on the 10000 test images: 98.07 % when batch_size is 40, learning_rate is 0.001
Accuracy of the network on the 10000 test images: 10.32 % when batch_size is 400, learning_rate is 0.1
Accuracy of the network on the 10000 test images: 10.09 % when batch_size is 400, learning_rate is 0.01
Accuracy of the network on the 10000 test images: 97.39 % when batch_size is 400, learning_rate is 0.001
```

Adam - Adaptive Momentum Estimation : Adam은 현재 가장 자주 사용되는 optimizer로 이전 optimizer들의 장점을 취해 만들어졌다. RMSprop과 momentum을 섞어서 쓴것과 같은 효과이다.

```
Epoch [1/5], Step [500/1500], Loss: 2.6339345
Epoch [1/5], Step [1000/1500], Loss: 2.9285314
Epoch [1/5], Step [1500/1500], Loss: 2.7231584
Epoch [2/5], Step [500/1500], Loss: 3.0577667
Epoch [2/5], Step [1000/1500], Loss: 3.4130147
Epoch [2/5], Step [1500/1500], Loss: 3.7597854
Epoch [3/5], Step [500/1500], Loss: 3.9670148
Epoch [3/5], Step [1000/1500], Loss: 2.8437274
Epoch [3/5], Step [1500/1500], Loss: 2.6278670
Epoch [4/5], Step [500/1500], Loss: 2.8244767
Epoch [4/5], Step [1000/1500], Loss: 3.4107285
Epoch [4/5], Step [1500/1500], Loss: 2.5541644
Epoch [5/5], Step [500/1500], Loss: 2.4788337
Epoch [5/5], Step [1000/1500], Loss: 3.1469808
Epoch [5/5], Step [1500/1500], Loss: 2.8888247
Accuracy of the network on the 10000 test images: 10.32 % when batch_size is 40, learning_rate is 0.1
Epoch [1/5], Step [500/1500], Loss: 0.7641429
Epoch [1/5], Step [1000/1500], Loss: 0.4923216
Epoch [1/5], Step [1500/1500], Loss: 0.3278615
Epoch [2/5], Step [500/1500], Loss: 0.3349781
Epoch [2/5], Step [1000/1500], Loss: 0.6840300
Epoch [2/5], Step [1500/1500], Loss: 0.5795134
Epoch [3/5], Step [500/1500], Loss: 0.3788624
Epoch [3/5], Step [1000/1500], Loss: 0.5720171
Epoch [3/5], Step [1500/1500], Loss: 0.3018228
Epoch [4/5], Step [500/1500], Loss: 0.5008322
Epoch [4/5], Step [1000/1500], Loss: 0.2804329
Epoch [4/5], Step [1500/1500], Loss: 0.7315800
Epoch [5/5], Step [500/1500], Loss: 0.4477713
Epoch [5/5], Step [1000/1500], Loss: 0.4600166
Epoch [5/5], Step [1500/1500], Loss: 0.3901701
Accuracy of the network on the 10000 test images: 88.05 % when batch_size is 40, learning_rate is 0.01
```

```
Epoch [1/5], Step [500/1500], Loss: 0.3006656
Epoch [1/5], Step [1000/1500], Loss: 0.0953600
Epoch [1/5], Step [1500/1500], Loss: 0.2045998
Epoch [2/5], Step [500/1500], Loss: 0.0438014
Epoch [2/5], Step [1000/1500], Loss: 0.0565102
Epoch [2/5], Step [1500/1500], Loss: 0.0650919
Epoch [3/5], Step [500/1500], Loss: 0.0385007
Epoch [3/5], Step [1000/1500], Loss: 0.0143479
Epoch [3/5], Step [1500/1500], Loss: 0.1045709
Epoch [4/5], Step [500/1500], Loss: 0.0679158
Epoch [4/5], Step [1000/1500], Loss: 0.0057907
Epoch [4/5], Step [1500/1500], Loss: 0.0535568
Epoch [5/5], Step [500/1500], Loss: 0.1269337
Epoch [5/5], Step [1000/1500], Loss: 0.1315414
Epoch [5/5], Step [1500/1500], Loss: 0.0559651
Accuracy of the network on the 10000 test images: 98.34 % when batch_size is 40, learning_rate is 0.001
Accuracy of the network on the 10000 test images: 10.28 % when batch_size is 400, learning_rate is 0.1
Accuracy of the network on the 10000 test images: 96.52 % when batch_size is 400, learning_rate is 0.01
Accuracy of the network on the 10000 test images: 97.46 % when batch_size is 400, learning_rate is 0.001
```

Adam의 경우 최적화된 parameter가 batch size는 40, learning rate는 0.001일 때라는 것을 알 수 있다. 이는 기존에 사용한 SGD와 정반대되는 결과이다.

loss function

기존에 사용한 Loss function은 CrossEntropyLoss이다. LeNet-5에 적용가능한 MSELoss를 추가적으로 사용해보았다.

7. 참고자료

[1] W.-Y. Lee, K.-E. Ko, Z.-W. Geem, and K.-B. Sim, “Method that determining the Hyperparameter of CNN using HS algorithm,” Journal of Korean Institute of Intelligent Systems, vol. 27, no. 1, pp. 22-28, Feb. 2017.

[2] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

[3] Kabbai, L., Abdellaoui, M. & Douik, A. Image classification by combining local and global features. Vis Comput 35, 679-693 (2019).

[4] https://wiserloner.tistory.com/1032

[5] https://www.oreilly.com/library/view/tensorflow-for-deep/9781491980446/ch04.html

[6] https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5

[7] http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/