

Project 3 - Report

2016112083 김연웅

Environment

os : window

versions : 구글 콜라보 플랫폼을 사용.

Task 1

Analyze and Compare the difference between FCN Model and LeNet-5 Model

- FCN Model

```
## Fully Connected Layer
class FCN(nn.Module):

    def __init__(self):
        super(FCN, self).__init__()
        self.layer1 = nn.Sequential(

            nn.Linear(28*28, 512, bias = True),
            torch.nn.BatchNorm1d(512),
            torch.nn.ReLU()

        )
        self.layer2 = nn.Sequential(
            nn.Linear(512, 10, bias=True),
        )

    def forward(self, x):
        x = x.view(x.size(0), -1) # flatten the input image
        x = self.layer1(x)
        x_out = self.layer2(x)
        return x_out
```

```
FCN(
  (layer1): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
  )
  (layer2): Sequential(
    (0): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

내가 구현한 Fully connected model이다. Fashion Mnist Dataset의 이미지가 28*28의 사이즈이므로, input dimension 은 28 x 28 로 설정했다. 이후 input dimension이 512이고, output dimension이 10 인 하나의 hidden layer을 배치했다. hidden layer을 통과한 후에는 Batch Normalization 과정을 실시 했고, activation function으로 ReLU를 사용했다. 이 모델의 최종 output dimension은 10개이고, 사용하는 loss function이 torch.nn.CrossEntropyLoss()이므로 마지막 output을 거친 후 softmax()와 같은 activation function을 배치하지는 않았다. (nn.CrossEntropyLoss는 내부적으로 softmax()를 적용시켜 주기 때문이다.)

- LeNet-5

```
# define LeNet-5 model
class LeNet_5(nn.Module):
    def __init__(self):
        super(LeNet_5, self).__init__()
        self.layer1 = nn.Sequential(

            torch.nn.Conv2d(in_channels=1,
out_channels=6, kernel_size=5, stride=1, padding=2),
            torch.nn.Tanh(),
            torch.nn.AvgPool2d(2, stride=2),
        )
        self.layer2= nn.Sequential(
            torch.nn.Conv2d(in_channels=6, out_channels=16,
kernel_size=5, stride=1),
            torch.nn.Tanh(),
            torch.nn.AvgPool2d(2, stride=2),
        )
        self.layer3= nn.Sequential(
            torch.nn.Conv2d(in_channels=16, out_channels=120,
kernel_size=5, stride=1),
            torch.nn.Tanh(),

        )
        self.fclayer1 = nn.Sequential(
            torch.nn.Linear(120, 84),
            torch.nn.Tanh(),
        )
        self.fclayer2 = nn.Sequential(
            nn.Linear(84, 10)
        )
    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = x.view(x.size(0), -1)
        x = self.fclayer1(x)
        x_out = self.fclayer2(x)
        return x_out
```

```

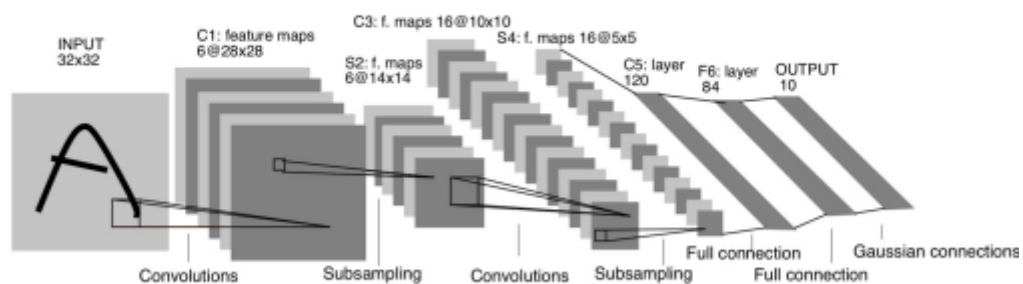
Lenet_5(
  (layer1): Sequential(
    (0): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): Tanh()
    (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (layer2): Sequential(
    (0): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (1): Tanh()
    (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (layer3): Sequential(
    (0): Conv2d(16, 120, kernel_size=(5, 5), stride=(1, 1))
    (1): Tanh()
  )
  (fc1): Sequential(
    (0): Linear(in_features=120, out_features=84, bias=True)
    (1): Tanh()
  )
  (fc2): Sequential(
    (0): Linear(in_features=84, out_features=10, bias=True)
  )
)

```

LeNet-5 모델이다. 모델의 세부적인 내용은 과제 스펙과, 구글링을 통해서 설정했다. 총 3개의 convolution layer와 1개의 Fully Connected Layer로 이루어져 있다. Input layer의 경우, Fashion MNIST dataset의 이미지가 28 x 28 인 반면, LeNet-5는 32x32이기에 padding을 적용시켜 dimension을 통일 시켜주었다.

Activation function은 torch.nn.Tanh()를 사용했다. 앞서 내가 구현한 FCN model과 마찬가지로, 같은 이유로 마지막 output 에는 따로 softmax activation 함수를 적용시키지 않았다.

2. LeNet-5



(출처: 과제 스펙)

- Weight initialization

```

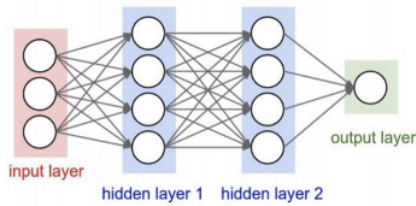
# method for initialize model`s weights using xavier normalization
def initweights(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_normal_(m.weight)

```

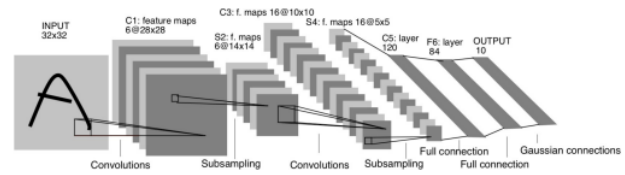
앞서 구현한 두 모델의 weight들을 xavier initialization 방법으로 초기화해주는 메소드이다. 모델을 만들고 동시에 apply 메소드를 통해 적용시켜주었다.

Difference between Fully Connected Model and Convolutional Model(LeNet-5)

1. Fully Connected Network



2. LeNet-5



Fully connected model에서 각 뉴런은 이전 layer의 모든 뉴런들과 연결되어있으며 이 모든 뉴런들은 각각 weight parameter을 지닌다. 이 모든 뉴런들의 연결은 데이터의 특징이나 유의미한 feature을 뽑아내는 것이 아니라, 특정한 목적이나 가정이 없는 general한 연결이다. 모든 뉴런들이 weight를 지니고 모든 층위의 모든 뉴런들이 서로 연결되어있기에 memory측면이나 performance측면에서 비용이 과중하다.

그 반면에 Convolutional model은 각 층위의 뉴런들이 모두 서로 연결되어있지 않고 일부만 연결되어 있다. 또한 각 층위에서는 convolution연산과 pooling연산 등을 사용하여서, same set of weights parameter을 사용한다. 이러한 연결 패턴은 FCN과 달리 인풋 데이터에 대한 유의미한 feature을 뽑아내는데 유리하다. 또한 memory나 performance 측면에서도 일부 연결과 parameter만을 사용하기 때문에 비용도 적고 효율적이다

(출처: https://www.reddit.com/r/MachineLearning/comments/3yy7ko/what_is_the_difference_between_a_fullyconnected/
<https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5>)

Create DataLoaders and Configure Hyperparameters

```
# hyper parameter setting
batch_size = 100
num_epochs = 10
learning_rate = 0.001

# Prepare Dataset
path = './'

# Normalize dataset
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize(mean=(0.5,), std=(0.5,))])

# Download dataset
train_data = dset.FashionMNIST(path, train=True, transform=transform,
                                download=True)
test_data = dset.FashionMNIST(path, train=False, transform=transform,
                                download=True)

# Make DataLoader
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_data, batch_size=batch_size)
```

train_data와 test_data를 각각 다운로드받고, 이후 학습시 mini-batch loading을 하기 위해 pytorch의 DataLoader 클래스를 통해 각각 DataLoader를 만들어 주었다. 학습을 위한 데이터로더는 shuffle을 했으며 test를 위한 데이터로더는 shuffle을 하지 않았다. mini-batch size는 100으로 설정했다.

총 epoch 수는 10, learning_rate 는 0.001로 설정하고 학습을 진행했다.

Loss function and Optimizer

```
# Make Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model_fcn.parameters(), lr = learning_rate)
```

loss function으로는 nn.CrossEntropyLoss()를, optimizer는 torch.optim.Adam() 을 사용했다.

Cross Entropy Loss 함수란?

Entropy는 불확실성의 척도로, 엔트로피가 높다는 것은 정보가 많고, 확률이 낮음을 의미하며 엔트로피가 낮다는 것은 정보가 적고 확률이 높음을 의미한다. 크로스 엔트로피란, 실제 데이터의 분포에 대해 알지 못하는 상태에서, 모델링을 통해 구한 계산된 분포로 실제 데이터의 분포를 예측하는 개념이다. 실제 분포와 모델링을 통해 구한 분포 두가지 개념이 사용되어 크로스 엔트로피라고 칭한다고 한다. 크로스 엔트로피는 실제 값과 예측 값이 맞는 경우 0으로 수렴하고, 서로 맞지 않는 경우 값이 커지므로, 실제 데이터의 y 값과 모델에 의한 아웃풋 예측 값의 차이를 줄이는 용도로 cross entropy를 손실함수로 이용한 것이다.

$$H(p, q) = - \sum_{x \in \text{classes}} p(x) \log q(x)$$

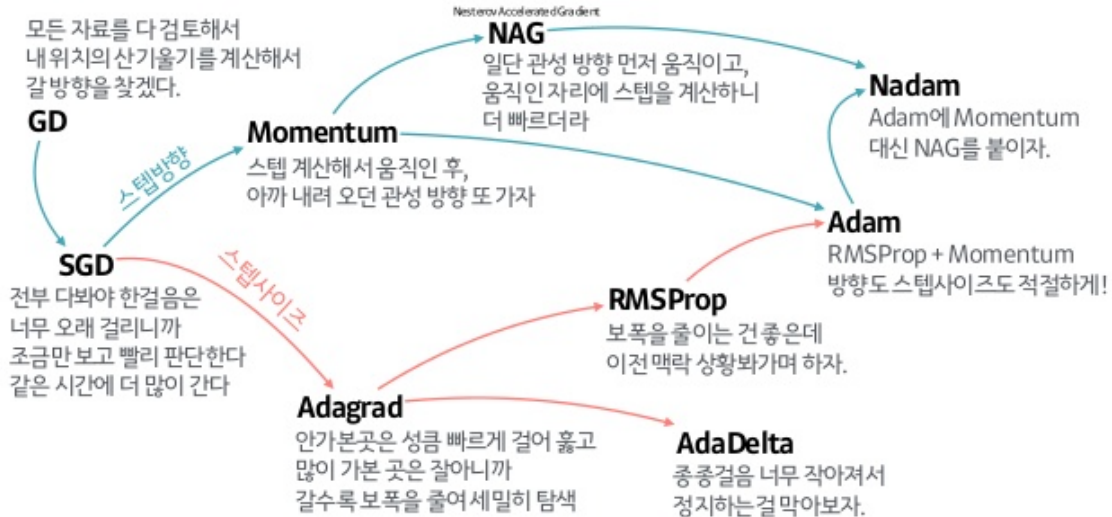
True probability distribution (one-hot)

Your model's predicted probability distribution

(<https://stackoverflow.com/questions/41990250/what-is-cross-entropy/41990932>)

Adam Optimizer란?

산 내려오는 작은 오솔길 잘찾기(Optimizer)의 발달 계보



출처: 하용호, 자습해도 모르겠던 딥러닝, 머리속에 인스톨 시켜드립니다

먼저, optimizer는 신경망 모델의 학습과 그 결과에 따른 손실함수의 값을 최소화하는 방향으로 모델의 parameter 값을 update하는 최적화 도구이다. Adam optimizer는 경사 하강법에서 출발한 알고리즘으로 Momentum과 RMSProp 기법을 합친 기법이다.

- **Momentum** : 미분을 통해 기울기를 구하고, 가중치 값을 갱신하는 경사하강법은 한가지 단점이 존재하는데 local minimum에 빠지게 되는 경우이다. 이 단점을 보완하기 위해 고안된 것이 바로 momentum이다. 가중치 값을 갱신할 때 일종의 관성을 부여해 매 갱신시 이전 단계의 갱신 방향을 반영하게끔 하는 기법이다.

• AdaGrad와 RMSProp

learning rate는 너무 커서도 너무 작지도 않은 적절한 값일때 최적의 학습이 일어나게 된다. 너무 크다면 oscilation(진동)이, 너무 작다면 학습 효율이 떨어지기 때문이다. 또한, 학습이 진행됨에 따라, 달리 말해 우리가 원하는 최적의 지점에 도달할수록 정답에 도달하기 위해서는 좀더 미세한 탐색이 요구된다. 이를 위해 학습이 진행됨에 따라 학습률을 낮춰준것이 AdaGrad이다. 과거의 기울기 값을 제공해서 연속적으로 더하는 식으로 학습률을 낮춘다고 한다.

이때, AdaGrad 기법으로 학습을 충분히 진행하게 되면 제공의 값 정도로 학습률이 반복적으로 줄어들기 때문에 학습률이 점점 0에 가까워져 오히려 학습의 진행이 안되는 문제가 발생한다. 이를 해결하는 방법으로 RMS Prop이 등장한다. RMS Prop 기법은 AdaGrad에서처럼 과거의 모든 기울기를 균일하게 더하는 것이 아니라 새로운 기울기의 정보만 반영하도록 해서 학습률이 0에 가까워지는 것을 방지하는 기법이다.

Adam optimizer는 Momentum과 RMS Prop 기법을 합친 optimizer로 경사 하강법의 단점들을 보완해 오늘날 가장 널리 쓰이는 optimizer로 알려져 있다

(출처: <https://sacko.tistory.com/42>)

Model Train

```
# Train
model_fcn.train() # LeNet-5의 경우 model_fcn 대신 model_lenet을 사용
loss_list_fcn = [] # LeNet-5의 경우 model_fcn 대신 model_lenet을 사용
for epoch in range(num_epochs):
```

```

avg_loss=0
# loop until every batch dataset is learned by model
for i, (imgs, labels) in enumerate(train_loader):
    # send it to gpu
    imgs, labels = imgs.to(device), labels.to(device)

    # predict
    prediction = model_fcn(imgs) # LeNet-5의 경우 model_fcn 대신 model_lenet
을 사용

    # set optimizer
    optimizer.zero_grad()

    # calculate loss
    loss = criterion(prediction, labels)

    # learn
    loss.backward()
    optimizer.step()

    # record avg loss
    avg_loss += loss.item()

    # print learning loss
    if (i+1) % 100 == 0:
        print('[{}], {}] loss: {}'.format(epoch+1, i+1, loss.item()))

# record avg loss for given batch data for later visulization
loss_list_fcn.append(avg_loss / len(train_loader))

```

학습과정을 진행하는 구현 부분이다. 총 epoch만큼 loop를 반복한다. 각 epoch에서는 앞서 만들어준 DataLoader에 의해 training dataset을 batch size만큼 반복해서 불러온다. 이후 구현한 모델에 데이터를 집어넣어 모델의 결과와 데이터셋의 y값을 loss function을 이용해 비교 후 loss 값을 저장한다. 이후 loss.backward()를 통해 backpropagation을 진행한 후, optimizer.step()을 통해 모델의 parameter들을 업데이트해준다.

출력은 매 mini-batch마다 해당 loss 값을 출력하게 하였으며, 추가적으로 매 epoch의 avg_loss를 저장해 이후 pyplot graph로 출력했다.

위의 코드는 FCN model을 구현하는 부분이지만, Lenet-5 모델의 경우에도 사용하는 모델 이외에 모두 동일하다.(loss function과 optimizer는 재정의하여 사용.)

Model Evaluate

```

# evaluate model after training

model_fcn.eval() # LeNet-5의 경우 model_fcn 대신 model_lenet을 사용
with torch.no_grad():
    correct = 0
    total = 0
    for i, (imgs, labels) in enumerate(test_loader):
        imgs, labels = imgs.to(device), labels.to(device)
        prediction = model_fcn(imgs) # LeNet-5의 경우 model_fcn 대신 model_lenet을
사용
        _, argmax = torch.max(prediction, 1)

```

```

total += imgs.size(0)
correct += (labels == argmax).sum().item()

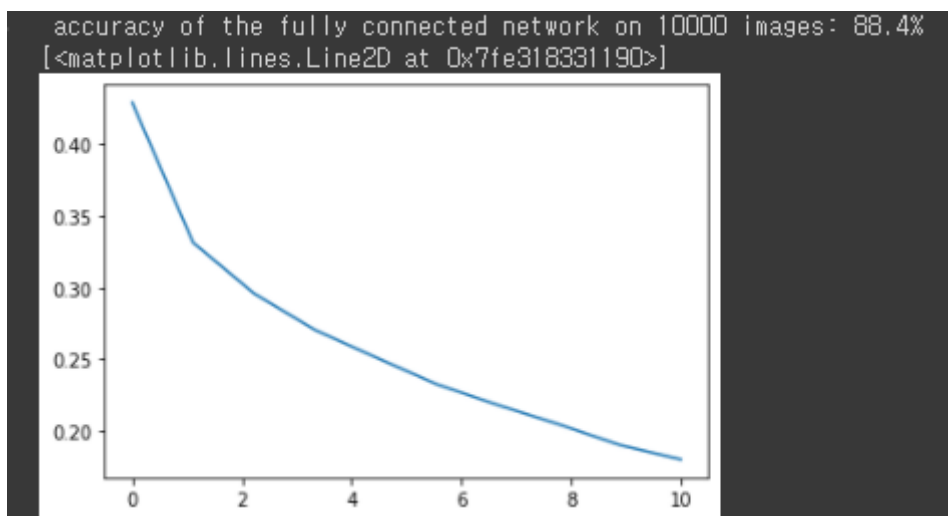
# print accuracy score
print(' accuracy of the fully connected network on {} images:
{}%'.format(total, correct / total * 100))

# visualize training result
step = np.linspace(0,num_epochs,num_epochs)
plt.plot(step,np.array(loss_list_fcn)) # LeNet-5의 경우 model_fcn 대신 model_lenet
을 사용

```

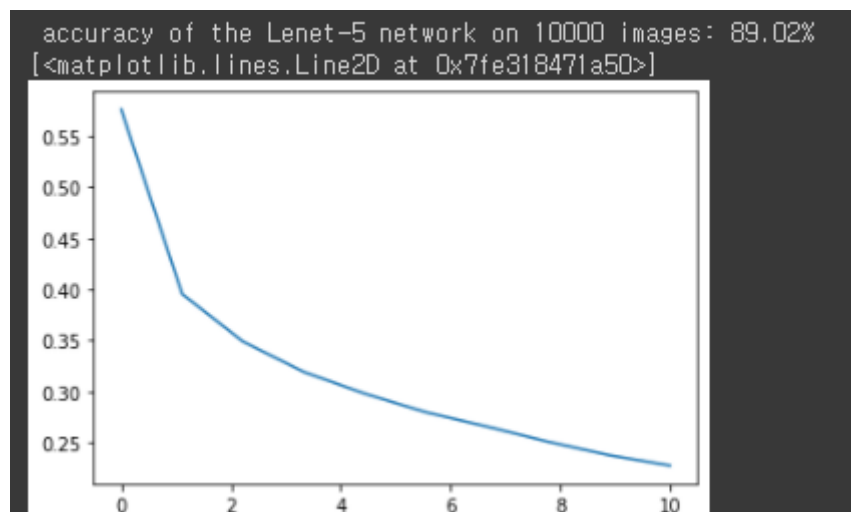
이전에 train한 모델에 test dataset을 넣어 결과를 받아온다. 결과를 테스트 데이터셋의 y값과 비교해 총 맞춘 개수를 총 데이터셋의 수로 나누어 accuracy를 계산한다. 마찬가지로 위의 코드는 FCN 모델의 평가 부분이며 LeNet-5의 경우 모델과 loss_list 변수만 달리 한 것과 같다

- FCN Model 결과



FCN model의 accuracy 점수는 88.4%를 기록했다. 그래프의 가로축은 epoch를, 세로축은 avg_loss를 의미한다. loss 값이 epoch이 진행됨에 따라 잘 수렴하고 있으므로 학습이 잘 진행되었다고 할 수 있다.

- LeNet-5 Model 결과



LeNet-5 모델의 accuracy는 89.02%를 기록했다. 마찬가지로 가로축은 epoch를, 세로축은 avg_loss를 의미한다. loss값이 epoch가 진행됨에 따라 잘 수렴하고 있으므로 학습이 잘 진행되었다고 할 수 있다.

예상과 달리, LeNet-5의 정확도 수치가 FCN Model과 비교했을때 크게 뛰어나지는 않았다. 초창기의 CNN 모델이라 그런 듯 하다. 오히려, 평균 손실 수치는 FCN이 더 낮은 값을 기록했다. task2 에서 해당 모델의 세부적인 내용들을 조작해 정확성을 올리려 시도했다.

Task 2

Analyze the elements of training process using 'LeNet-5' network

Original setting

	learning rate	weight decay	loss function	batch size	Accuracy
	0.001	none	nn.CrossEntropyLoss()	100	89.02%

원래 구현한 LeNet-5 모델의 설정 값들이다. 이 값들을 기준으로 삼아 앞으로의 분석을 진행했다.

*각 테스트 케이스를 수행할 때 마다 모델을 새로 재정의 하여 파라미터가 초기화된 상태로 진행했다.

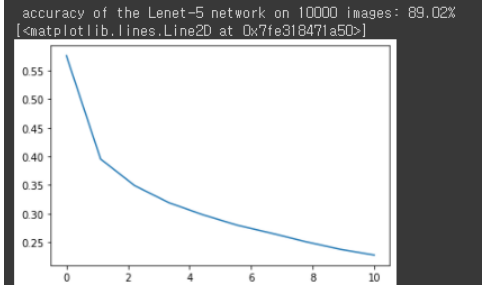
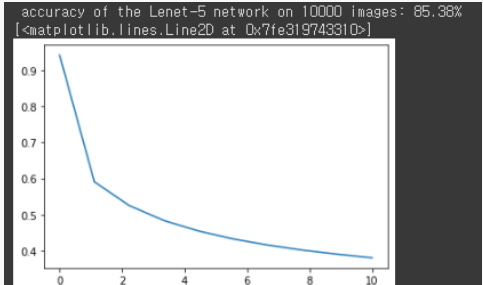
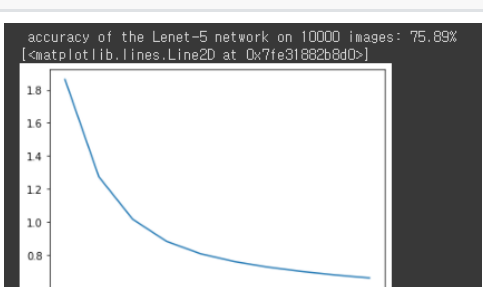
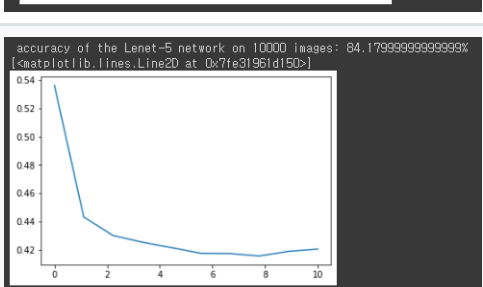
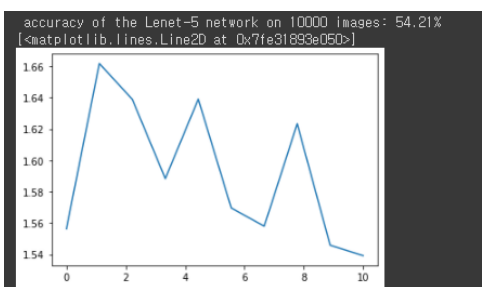
1. Optimizer parameter

Adjust parameter related with optimizer and compare the results before and after doing those

총 두가지의 파라미터에 다양한 값을 적용시켜 보았다.

- learning-rate : 학습률, gradient descent할 때 스텝의 크기
- weight_decay: L2 penalty (regularization), dataloss 에 추가하는 페널티 텀의 계수, 트레이닝 데이터셋에 대한 과적합을 방지한다.

Weight decay를 없음으로 고정하고, learning rate 값을 다양하게 넣어 테스트 했다.

	learning rate	weight decay	accuracy score	average loss graph
original	0.001	default	89.02%	
test1	0.0001	default	85.38%	
test2	0.00001	default	75.89%	
test3	0.01	default	84.18%	
test4	0.1	default	54.21%	

original 세팅인 learning rate=0.001의 세팅은 avg loss가 약 0.6에서 시작하여 0.1으로 수렴하는 것을 그래프로 확인 할 수 있다. 적절한 step으로 학습이 매 epoch마다 지속적으로 이루어졌다고 생각되며, 실제로 accuracy score도 테스트 세팅들 중 가장 높은 수치를 기록했다.

test1과 test2는 각각 0.0001, 0.00001의 learning rate를 세팅했고 학습률이 낮아지는 것이 모델에 어떤 영향을 끼치는지를 살펴보기 위한 의도로 시험했다. 각 테스트의 avg loss 그래프를 살펴보면, avg loss가 점점 낮아지며 동시에 수렴하고 있는 것으로 보아 학습이 진행이 된다고 생각된다. 하지만, learning rate이 작아짐에 따라, 수렴하는 loss의 수치가 높아지고 있다. (test1의 경우 약 0.3, test2의 경우 약 0.6) 이는 learning rate이 너무 작아, local optima에 빠져버려 헤어 나오지 못하는 상황, 혹은 학습률이 너무 낮아 학습의 진행 속도가 심각하게 더디기에 발생한 결과라고 생각된다. learning rate가 작아

질 수록 최적의 learning rate(original setting)에 비해 수렴하는 loss의 값이 점점 커졌으며, accuracy도 점점 낮아졌다.

test 3과 test 4는 각각 0.01, 0.1의 learning rate을 세팅했고 학습률이 높아지는 것이 모델에 어떤 영향을 끼치는지를 살펴보기 위한 의도로 시험했다. test 3의 그래프에서, avg loss 수치가 학습 초기에 0.55에서 0.42로 살짝 떨어지고 이후로는 거의 떨어지지 않다가 오히려 후반부에는 살짝 상승하는 수치를 기록했다. test 4에서는 1.5~1.66대의 다소 높은 수치의 avg loss를 기록했으며 점점 낮아지는 것이 아니라 불규칙하게 들쭉날쭉한 그래프를 그리는 것으로 보아 학습이 전혀 이루어지지 않았음을 알 수 있다. 너무 큰 learning step으로 인해 optima에 도달하기보다는 주변부를 맴도는 상황이라고 생각된다.

learning rate을 original setting 값인 0.001로 고정하고, weight decay를 부여하여 실험했다.

	learning rate	weight decay	accuracy score	avg loss graph
original	0.001	default(0)	89.02%	
test1	0.001	0.0001	89.38%	
test2	0.001	0.001	86.71%	
test3	0.001	0.01	83 %	
test4	0.001	0.1	63.94%	

weight decay는 트레이닝 데이터셋의 오버피팅(과적합)을 막기 위해 weight의 제곱합을 제약을 걸어 loss를 최소화하는 것이다. L2 Regularization 혹은 L2 penalty라고도 불리운다.

실험 전, 나는 weight decay를 적용하면 트레이닝 데이터셋에 대한 과적합을 막아주므로 training 과정에서의 avg loss 그래프 곡선이 좀더 완만하게 하락할 것이며 수렴하는 avg loss 수치가 더 올라갈 것이라고 예상했다. 더불어, 만일 적정량의 weight decay를 적용시켰을 때, 만일 정확성이 상승한다면 구형한 LeNet-5 모델이 training dataset에 대한 과적합이 진행되었음을 알 수 있으리라 생각했다.

test1~test4까지 learning rate은 0.001로 고정하고 weight decay를 각각 0.0001, 0.001, 0.01, 0.1을 주어 실험했다. 실험 결과, 0.0001의 weight decay를 준 test 1에서 original 보다 약 0.26% 높은 정확성을 기록했다. 이후 test2~test4로 진행됨에 따라 accuracy는 점점 낮아졌다. 실험들의 accuracy 점수로 보아 이번 프로젝트의 LeNet-5와 Fashion Mnist의 training dataset 간에는 과적합이 거의 혹은 전혀 없다

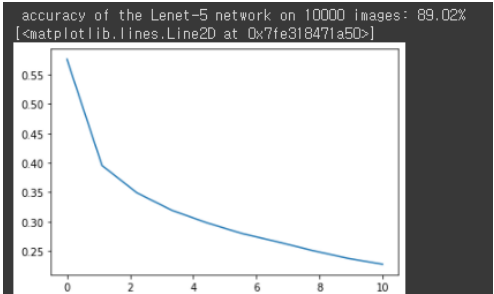
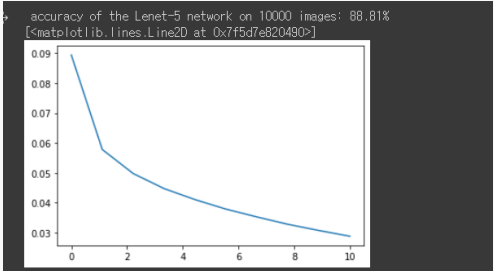
고 생각된다.

또한 weight decay를 점점 늘려갈수록 수렴하는 avg loss가 예상대로 점점 높아짐을 확인할 수 있었다. 심지어, 0.1의 weight decay를 준 test4의 결과 수치를 살펴보면 전혀 학습이 진행되지 않았다고 생각된다. L2 penaty가 너무 높아 모델이 학습을 전혀 못했다고 생각된다. 정확성도 기존의 감소 추세(약 -2~-3%)를 훨씬 뛰어넘은 63.94%를 기록했다.(-20%) L2 penalty가 가해질 수록 training dataset에 대한 학습이 penalize되어 avg loss 감소가 더뎠다고 생각된다. 더불어 original, test1, test 2 의 그래프를 살펴보면 weight decay가 늘어날 수록 평균손실 곡선이 미약하지만, 점점 완만해 짐을 확인할 수 있었다.

2. Loss Function

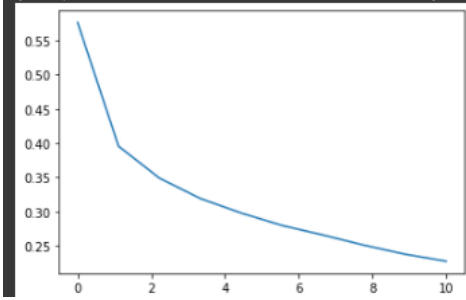
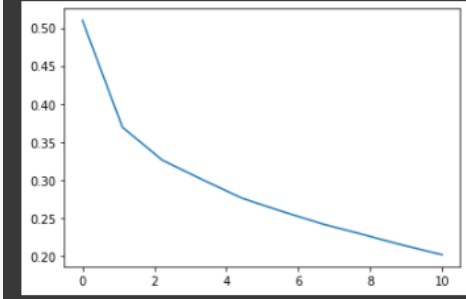
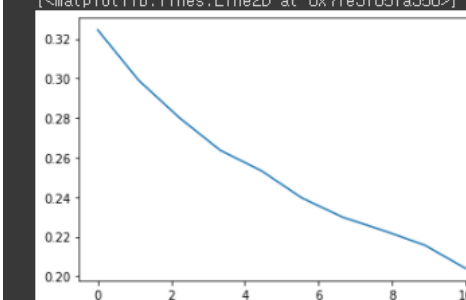
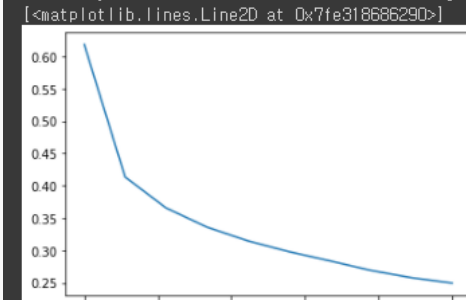
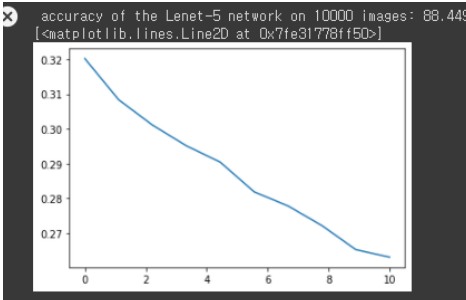
loss function을 바꾸는 테스트케이스를 실험해 보았다.

loss function은 nn.MultiMarginLoss()을 사용했다. 이 함수는 multi classification 문제에서, 특정 loss 수치 이하면 loss를 아예 0으로 취급하는 함수이다.

	loss function	accuracy score	avg loss graph
original	nn.CrossEntropyLoss()	89.02%	
test1	nn.MultiMarginLoss()	88.81%	

테스트 결과 CrossEntropyLoss()를 사용했을 때 보다 약간 떨어진 수치를 기록했다.

3. Batch Size

	batch size	accuracy score	avg loss graph
original	100	89.02%	<p>accuracy of the Lenet-5 network on 10000 images: 89.02% [<matplotlib.lines.Line2D at 0x7fe318471a50>]</p> 
test1	50	89.24%	<p>accuracy of the Lenet-5 network on 10000 images: 89.24% [<matplotlib.lines.Line2D at 0x7fe318566250>]</p> 
test2	10	89.09%	<p>accuracy of the Lenet-5 network on 10000 images: 89.09% [<matplotlib.lines.Line2D at 0x7fe3185fa350>]</p> 
test3	200	88.55%	<p>accuracy of the Lenet-5 network on 10000 images: 88.55% [<matplotlib.lines.Line2D at 0x7fe318686290>]</p> 
test4	1000	88.49%	<p>accuracy of the Lenet-5 network on 10000 images: 88.44999999999999% [<matplotlib.lines.Line2D at 0x7fe31778f150>]</p> 

mini batch size를 조정하여 실험을 진행해 보았다. 테스트 결과, original인 batch size=100을 기준으로 더 큰 batch size 테스트들은 정확도가 감소했고 더 작은 batch size 테스트들은 정확도가 증가했음을 알 수 있었다. 정확도의 차이가 그리 큰 수치는 아니라 유의미한 결과로 보기 힘들지만, 수치들의 방향성이 확실히 존재한다는 것을 미루어보아, 너무 큰 batch size는 모델의 정확도를 낮출 것이라 분석했다.

또한 측정은 하지 않았지만, 큰 batch size의 테스트가 실행 속도 측면에서 훨씬 빠른 성능을 보였고, batch size가 작아질 수록 실행 속도가 느려졌다. 이는 묶음으로 연산하는 선형대수 관련 library들 때문이라 생각되는데, matrix 연산을 할 때 메모리를 많이 사용하는 대신 연산 효율을 높이는 식으로 구현되었기 때문에 큰 mini batch일수록 빠른 것이라 이해했다.

테스트 결과가 잘 납득이 가지 않아 추가적인 구글링을 통해 mini batch size와 모델 학습에 관한 관련 내용을 조사했다. 조사 결과 너무 큰 batch size는 모델 학습도를 낮추는 결과를 초래함이 맞다고 한다. 그 이유는 다음과 같다.

큰 batch size일수록 모델은 데이터에 대한 전체적인 학습을 진행하고 작은 batch size일수록 모델은 좀 더 해당 mini batch에 대하여 편향된 학습을 하게 된다. 이 때문에 작은 batch size로 학습하는 모델은 training 단계에서 더 많은 loss noise을 발생시키게 된다. (loss가 oscillation 한다는 의미). 이 loss의 noise는 모델이 local minimum에서 탈출할 기회를 끊임없이 만들어주게 되고 결국 이것이 작은 batch size가 모델 학습에 긍정적인 방향으로 기여하는 이유라고 한다.

(출처:<https://nyanye.com/paper/2020/05/30/good-batchsize/>)

조사 이후, test2의 train 단계에서의 loss 출력을 살펴보니, 실제로 많은 loss noise가 있음을 확인할 수 있었다. 반면 test4의 train loss는 비교적 stable한 수치들을 출력했음을 확인했다.

4. Be creative and analytical! (Change other hyper parameters or model elements)

Fashion MNIST 데이터셋에 대한 LeNet-5 모델의 정확성을 올리기 위해, 기존의 LeNet-5모델의 element들과, 다른 hyper parameter들을 수정하였다.

LeNet-5 Modified

```
# define Lenet-5 model
class Lenet_5_modified(nn.Module):
    def __init__(self):
        super(Lenet_5,self).__init__()
        self.layer1 = nn.Sequential(

            torch.nn.Conv2d(in_channels=1, out_channels=6,
kernel_size=5, stride=1, padding=2),
            torch.nn.BatchNorm2d(6),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(2, stride=2),

        )
        self.layer2= nn.Sequential(
            torch.nn.Conv2d(in_channels=6, out_channels=16,
kernel_size=5, stride=1),
            torch.nn.BatchNorm2d(16),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(2, stride=2),
```

```

    )
    self.layer3= nn.Sequential(
        torch.nn.Conv2d(in_channels=16,out_channels=120,
kernel_size=5,stride=1),
        torch.nn.BatchNorm2d(120),
        torch.nn.ReLU()

    )
    self.fclayer1 = nn.Sequential(
        torch.nn.Linear(120,84),
        torch.nn.ReLU()
    )
    self.fclayer2 = nn.Sequential(
        nn.Linear(84,10)
    )

    self.dropout = torch.nn.Dropout(0.2)
def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = x.view(x.size(0), -1)
    x = self.fclayer1(x)
    x = self.dropout(x)
    x_out = self.fclayer2(x)

    return x_out

# Optimizer and Step Scheduler

optimizer = torch.optim.Adam(model_lenet.parameters(), lr = 0.01)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1000,
gamma=0.9)

```

변경사항

- Model Elements

3개의 convolutional 를 통과한 이후 각각 torch.nn.BatchNorm2d()를 활용해 batch normalization을 적용시켰다.

모든 Activation Function을 torch.nn.tanh()에서 torch.nn.ReLU()로 교체했다.

Convolutional layer 이후 pooling 함수를 torch.nn.AvgPool2d()에서 torch.nn.MaxPool2d로 교체했다.

Fully connected Layer 을 통과시킨 후 과적합을 막기 위해 torch.nn.Dropout(0.2)를 적용시켜주었다.

- Scheduler와 Optimizer

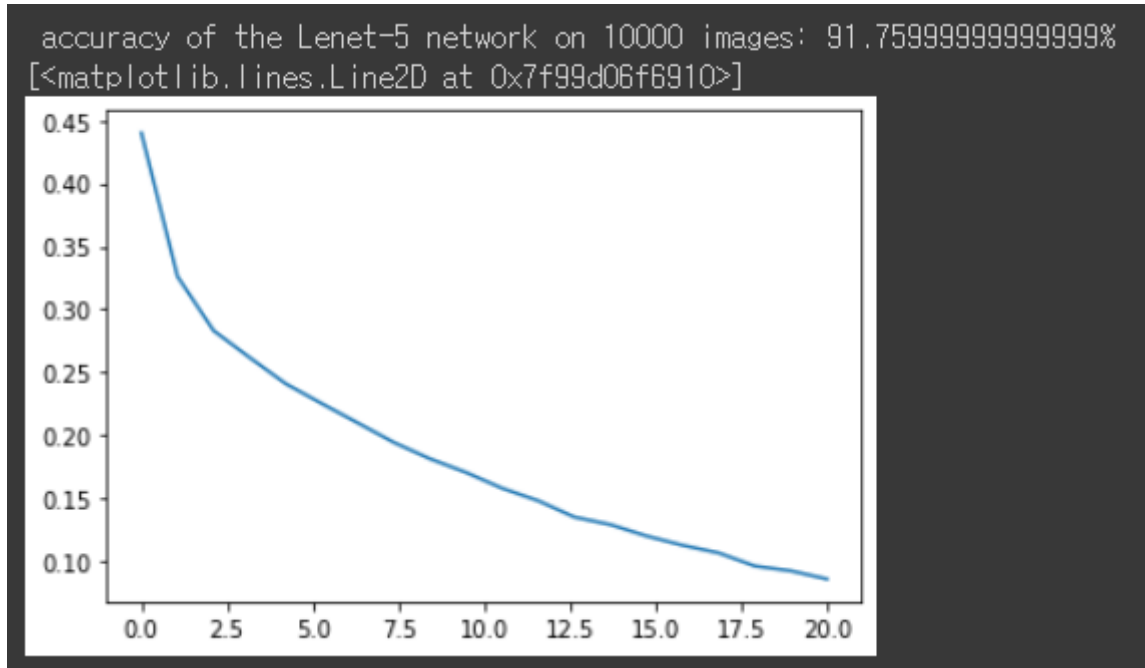
Optimizer는 그대로 Adam optimizer를 사용했고 learning_rate decay scheduler를 사용하기 때문에 기존의 learning_rate인 0.001에서 0.01로 변경해주었다.

scheduler의 경우 torch.optim.lr_scheduler.StepLR()을 사용했으며 batch size가 100이므로 10번 batch시마다 learning rate을 적용시키기 위해 step_size를 1000으로 설정했다. gamma값으로는 0.9를 주어서 지정한 간격마다 learning rate에 0.9를 곱하게 하였다.

- Epoch

기존의 10 epoch에서 좀더 정확한 학습을 위해 20 epoch으로 변경했다.

결과



LeNet-5_modified의 결과는 기존모델의 89.02% 정확도 점수에서 약 2.8% 향상한 91.76%을 기록했다. 최종 epoch의 avg_loss도 0.10 보다 낮은 수치로 기존 모델의 avg_loss에 비해 훨씬 낮은 수치를 기록했다.