인공지능(딥러닝)개론 HW3 결과보고서

Fashion-MNIST Classifier using CRNN

학번: 20180032 이름: 남기동

1. Fashion-MNIST Dataset

Fashion MNIST 데이터셋은 아래의 그림과 같이 패션 용품들에 대한 이미지 모음이다. 손글씨를 저장하고 있으며 딥러닝 모델 학습에 기본적으로 사용되는 MNIST 데이터셋과 동일하게 총 10개의 클래스를 가지고 있으며 기본적으로 이미지는 흑백에 28 x 28 픽셀의 크기로 총 70,000개로 구성되어 있다. torchvision.dataset을 통해 다운받을 수 있으며 train 데이터 6만개와 test 데이터 1만개로 구분된다.

```
train_data = torchvision.datasets.FashionMNIST(root='./datasets',
                                               train=True,
                                               transform=transforms.ToTensor(),
                                               download=True)

test_data = torchvision.datasets.FashionMNIST(root='./datasets',
                                              train=False,
                                              transform=transforms.ToTensor(),
                                              download=True)
```

```
len(train_data), len(test_data)
```

```
(60000, 10000)
```

```
image, label = next(iter(test_loader))
print(image.size()) # [Batch, Channel, Height, Width]
                    # batch size로 가져온 데이터 개수, 입력 채널(1), 28 x 28(FashionMNIST의 이미지 형식)
                    # FashionMNIST는 흑백이기 때문에 입력 채널이 1이다
```

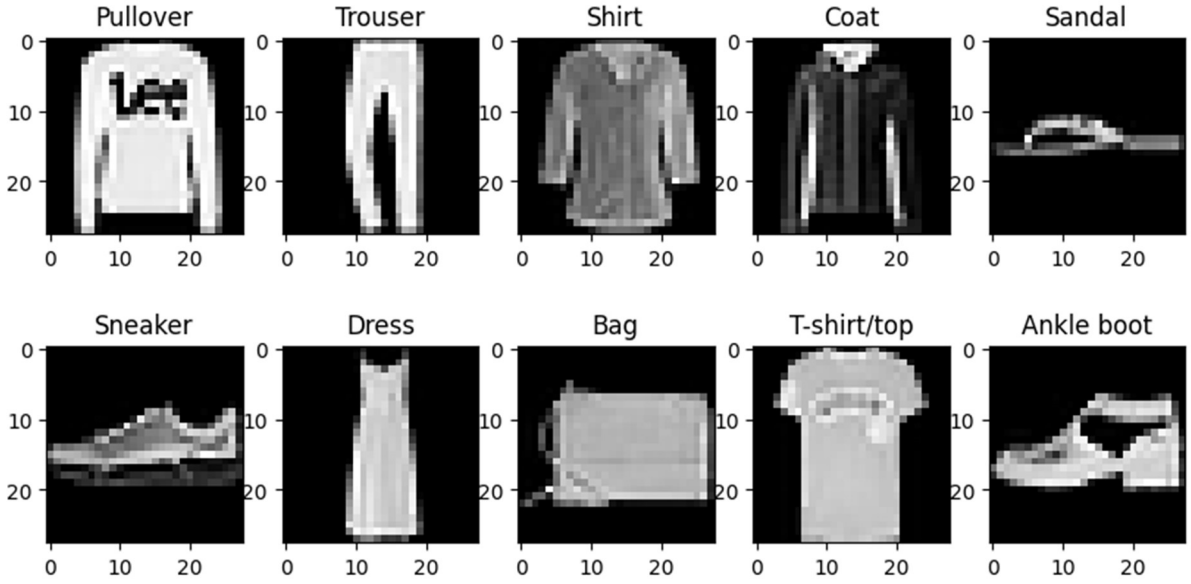
```
torch.Size([50, 1, 28, 28])
```

Test data는 Train 과정에서 불필요하지만 Fashion MNIST의 데이터를 분석하기 위한 목적에서 불러왔다. Test data를 test_loader에 올리고 한 개의 image와 label을 추출하여 size를 확인한 결과 위의 사진과 같이 [50, 1, 28, 28]이 나왔다. 50은 이번 과제에서 사용한 모델에서 사용한 Batch size이며 앞서 이야기했듯 Fashion MNIST는 흑백 이미지이기 때문에 입력 채널로 1, 이어서 이미지의 픽셀 크기로 28 x 28이라는 것을 확인할 수 있다.

```
plt.figure(figsize=(10,5))
cnt = 0
classList = [] # 클래스의 이름을 저장하여 현재까지 나온 클래스의 개수를 세기 위한 리스트

while True:
  if len(classList) == 10: break                                    # 10개의 클래스가 모두 출력되었으면 종료
  cnt += 1
  if test_data.classes[test_data[cnt][1]] not in classList:         # 아직 나오지 않은 클래스라면
    classList.append(test_data.classes[test_data[cnt][1]])          # 리스트에 추가하고
    plt.subplot(2, 5, len(classList))                               # 위치 지정하고
    plt.title(test_data.classes[test_data[cnt][1]])                 # 타이틀로 클래스 적고
    plt.imshow(test_data.data[cnt].reshape(28,28), cmap='gray')     # 이미지 출력
```

이어서 Test data에서 10개의 클래스가 나올 때까지 이미지를 추출하고 각각의 클래스에 해당하는 이미지를 위의 코드를 통해 살펴볼 수 있다. 그 결과는 아래와 같다.



10개의 클래스는 Pullover, Trouser, Shirt, Coat, Sandal, Sneaker, Dress, Bag, T-shirt/top, Ankle boot 라는 것을 위의 예제 데이터 사진과 함께 확인해볼 수 있다.

추가적으로 이번 과제에서는 이전 과제와 달리 Validation data를 사용하여 매 epoch마다 Validation data에 대한 성능을 측정하여 가장 좋은 성능을 가진 모델을 선정해야 한다. 따라서 아래와 같이 9:1의 비율로 train data를 분할하여 validation data를 따로 설정하였다. 일반적으로는 train data에서 20% 정도를 분할하여 validation data로 사용한다. 하지만 그렇게 하면 학습할 데이터의 양이 감소하기 때문에 성능을 높이는 것이 중요한 이번 과제의 특성상 validation data의 크기를 줄임으로써 최대한 학습할 수 있는 train data의 양을 늘리고자 하였다. 따라서 20%보다 작은 10%를 validation data로 분할하기로 결정하였다. 해당 코드와 split 이후 train data, validation data, test data의 개수는 5만 4천개, 6천개, 1만개인 것을 아래의 사진을 통해 확인할 수 있다.

```
## train data의 10%를 validation data로 분할한다
train_size = int(0.9 * len(train_data))
val_size = len(train_data) - train_size

train_data, val_data = torch.utils.data.random_split(train_data, [train_size, val_size])
```

```
len(train_data), len(val_data), len(test_data)
```

```
(54000, 6000, 10000)
```

2. RCNN 모델 설명

이번 과제를 해결하기 위해 설계한 모델은 2개의 Covolution Layer와 RNN이 결합되어 있는 구조를 갖고 있다. 우선 모델에 사용된 하이퍼 파라미터를 살펴보고, 처음에 사용된 2개의 Convolution Layer의 구조에 대해 이해한 후, 데이터를 RNN의 입력으로 넣기 위해 적절히 reshape하는 과정과 함께 RNN의 구조를 살펴볼 것이다. 이를 토대로 output shape이 각 layer에서 어떻게 변화하는지를 예측하고 해당 예측이 summary 함수를 통해 출력된 결과와 일치함을 확인할 것이다.

1) Hyper-Parameter

```
num_classes = 10  # FashionMNIST의 클래스는 10개
in_channel = 1    # 흑백 이미지이기 때문에 입력 채널은 1

# Hyper-parameters
batch_size = 50
max_pool_kernel = 2
learning_rate = 0.0005
num_epochs = 15
```

4개의 하이퍼 파라미터를 설정하였는데 batch_size, max_pool_kernnel, learning_rate, num_epochs이 그것이다. 우선 batch size는 50으로 설정하였으며, Max Pooling layer에서 사용되는 Kernel의 사이즈를 설정하는 하이퍼 파라미터인 max_pool_kernel은 2로 설정하였다. 이는 이미지의 크기를 반으로 줄이는 역할을 하게 된다. 다음으로 learning_rate는 0.0005로 설정하였는데, 처음에는 일반적으로 사용하는 0.001을 사용했다가 모델이 제대로 수렴하지 못하는 것 같아서 learning rate를 반으로 줄인 것이다. 마지막으로 num_epochs는 전체 학습 횟수를 의미하며 15로 설정하였다.

2) Covolution Layer

```
class CRNN(nn.Module):
  def __init__(self, input_size, hidden_size, num_layers, num_classes):
    super(CRNN, self).__init__()

    # 1st Layer - Convolution
    self.layer1 = nn.Sequential(  # 28x28x1 --> padding==2 --> 32x32x1 --> filter 5x5 --> 28x28x32
        nn.Conv2d(in_channels=in_channel, out_channels=32, kernel_size=5, stride=1, padding=2), # 5x5 kernel
        nn.BatchNorm2d(num_features= 32),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=max_pool_kernel) # 28x28x32 --> 2x2 max_pooling --> 14x14x32
    )
    # 2nd Layer - Convolution
    self.layer2 = nn.Sequential( # 14x14x32 --> padding==2 --> 18x18x32 --> filter 5x5 --> 14x14x64
        nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5, stride=1, padding=2),
        nn.BatchNorm2d(num_features=64),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=max_pool_kernel) # 14x14x64 --> 2x2 max_pooling --> 7x7x64
    )
```

Convolution Layer는 총 2개로 구성되어 있다. 첫 번째 Layer에서는 입력 채널을 1로 받아서 5x5 kernel(filter)를 사용하여 합성곱 연산을 수행한다. 두 개의 convolution layer 모두 동일한 구조를 갖고 있다. 기본적으로 5x5 kernel을 사용하며 padding은 2, stride는 1로 고정된다. 이러한 설정은 기존에 입력된 이미지의 크기를 그대로 유지할 수 있다는 장점이 있다. 이어서 Batch Normalization을 수행하고 activation function으로 ReLU를 사용하며 2x2 Maxpooling을 사용하여 이미지의 크기를 반으로 줄인다. 즉, 첫 번째 Convolution Layer를 지나고 이미지는 14x14가 될 것이며 채널은 32가 된다. 이어서 두 번째 Convolution Layer를 지나고 이미지는 7x7가 되며 채널은 총 64개가 된다.

3) Recurrent Layer

```
# 2개의 Covolution Layer를 지나고 RNN에 들어가기 전의 이미지 구조는 7x7x64채널이다
# 이것을 sequence length와 input size로 분할하기 위해 각각 7x8로 설정하였다
self.hidden_size = hidden_size
self.num_layers = num_layers
self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True, dropout=0.4)
# "batch_first=True" 옵션으로 x -> (batch_size, seq, input_size) 로 dimension이 정해진다

self.fc = nn.Linear(in_features=hidden_size, out_features=num_classes)
```

위에서 살펴본 2개의 Convolution Layer를 지나면 이미지는 7x7x64가 되며 이것을 입력으로 하여 Recurrent Layer를 지나게 된다. Recurrent Layer의 하이퍼 파라미터로는 hidden size, sequence length, num_layers가 있다. 해당 하이퍼 파라미터들은 다음과 같이 설정했다.

```
sequence_length = 7*8   # 입력으로 주는 sequence를 몇으로 줄 것이냐
input_size = 7*8        # input data의 차원
hidden_size = 128       # hidden state의 차원
num_layers = 5          # RNN의 은닉층 레이어 개수
```

Sequence_length는 입력으로 주는 sequence의 길이를 의미하며 input data의 차원을 설정하는 input_size와 함께 Recurrent Layer의 입력과 일치해야 한다. 두 하이퍼 파라미터의 값이 7x8인 이유는 이후 추가적으로 설명하도록 한다. 다음으로, Hidden_size는 RNN에서 사용되는 hidden state의 차원을 의미하며 128로 설정하였고 num_layers는 RNN의 은닉층 레이어의 개수를 의미하며 5로 설정하였다. 이러한 하이퍼 파라미터를 통해 다시 위의 RNN의 구조를 살펴보면 하이퍼 파라미터를 입력받아서 저장하고 batch_first=True 옵션과 함께 RNN 모델의 객체가 생성되는 것을 확인할 수 있다. Batch_first 옵션은 batch_size, sequence_length, input_size로 입력 데이터의 dimension을 재설정해주는 옵션이다. 마지막으로 Recurrent Layer의 출력 차원인 hidden size가 Fully Connected Layer의 입력이 되고 최종적으로 10개의 클래스로 출력된다.

```
def forward(self, x): # 설정한 변수들로 순전파
  h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device) # 0으로 이루어진 텐서로 초기화
                                                # (hidden layer개수, batch_size, hidden state의 차원)
  x = self.layer1(x)
  x = self.layer2(x)

  # RNN에 들어갈 때 batch_size, seq_length, input_size로 맞춰줘야 한다
  x = x.reshape(x.size(0), -1, input_size) # x.size(0) = batch_size, input_size를 넣어주고 seq_length를 알아서 계산

  out, _  = self.rnn(x, (h0)) # out -> (batch_size, seq_length, hidden_size)

  # 필요한 것은 마지막 시퀀스의 출력 뿐이므로, out -> (batch_size, -1, hidden_size)로 설정해준다
  out = self.fc(out[:,-1,:]) # 마지막 out만 가져와서 fully connected에 넣는다

  return out
```

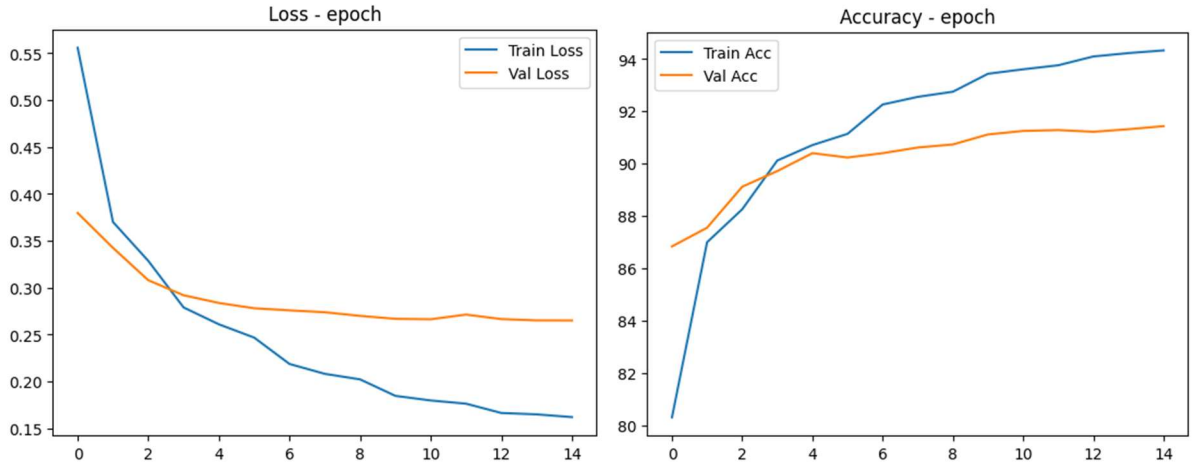
추가적으로 확인할 것은 두 개의 Convolution Layer를 지나고 나온 데이터의 구조가 RNN의 입력에 맞게 reshape 되어야 한다는 것이다. 위의 코드를 보면, x = x.reshape(x.size(0), -1, input_size) 부분이 RNN의 입력에 맞게 데이터의 dimension을 재구성 하고 있다. 첫 번째는 batch size이며 이는 x.size(0)으로 표현했고 두 번째는 sequence length이고 세 번째가 input_size이다. Input_size를 넣어주고 두 번째 인자에 -1을 넣어 sequence_length가 자동적으로 계산되어 reshape 되는 것이다. 이때 Convolution layer를 지난 데이터가 7x7x64라고 언급한 것을 토대로 input_size와 sequence_length가 설정되어야 한다. 7x7x64가 input_size x sequence_length가 되어야 하기 때문에 각각을 7x8로 설정하여야 하며 그렇기 때문에 위에서 하이퍼 파라미터로 input_size, sequence_length를 7x8로 설정한 것이다.

4) Summary

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 28, 28]             832
       BatchNorm2d-2           [-1, 32, 28, 28]              64
              ReLU-3           [-1, 32, 28, 28]               0
         MaxPool2d-4           [-1, 32, 14, 14]               0
            Conv2d-5           [-1, 64, 14, 14]          51,264
       BatchNorm2d-6           [-1, 64, 14, 14]             128
              ReLU-7           [-1, 64, 14, 14]               0
         MaxPool2d-8             [-1, 64, 7, 7]               0
               RNN-9  [[-1, 56, 128], [-1, 2, 128]]               0
           Linear-10                   [-1, 10]           1,290
================================================================
Total params: 53,578
Trainable params: 53,578
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 13.07
Params size (MB): 0.20
Estimated Total Size (MB): 13.27
----------------------------------------------------------------
```

위에서 분석했듯, 첫 번째 Convolution Layer를 지나고 14x14 이미지 크기에 32 채널이 생긴 것을 알 수 있으며 두 번째 Convolution Layer를 지나고 7x7 이미지 크기에 64 채널이 출력된 것을 확인할 수 있다. RNN-9는 [[-1, 56, 128], [-1, 2, 128]]이라 적혀 있는데, 각 리스트의 첫 번째 부분은 batch size로 -1로 설정되어 자동적으로 체크되고 있다. 두 번째 인자인 56과 2는 각각 RNN layer에 입력으로 들어가는 sequence의 길이와 RNN layer에서 출력되는 sequence의 길이이다. 입력으로 들어가는 sequence의 길이는 앞에서 분석했듯 7x8인 56을 나타내고 있는 것이다. 세 번째 인자인 128은 hidden state를 가리킨다.

3. 결과 분석



좌측의 그래프는 Train Loss와 Validation Loss를 나타낸 그래프이다. 그리고 우측의 그래프는 그러한 Loss를 바탕으로 batch size만큼 나눠주어 train과 validation에 대한 정확도를 나타낸 그래프이다. 두 그래프를 보면 알 수 있지만 처음에는 train과 validation에 대해 모두 학습을 잘 하고 그에 따라 loss가 유의미한 수준으로 떨어지는 것을 확인할 수 있다. 하지만 epoch이 3일 때부터 train loss가 감소하는 정도와 validation loss가 감소하는 정도가 현격히 차이나는 것을 볼 수 있다. 이는 과적합을 우려해볼 수 있는 지점이지만, 최종 epoch인 15까지 validation loss가 증가하는 경향성은 관측되지 않았기 때문에 조기 종료(Early Stopping)을 하지는 않았다.

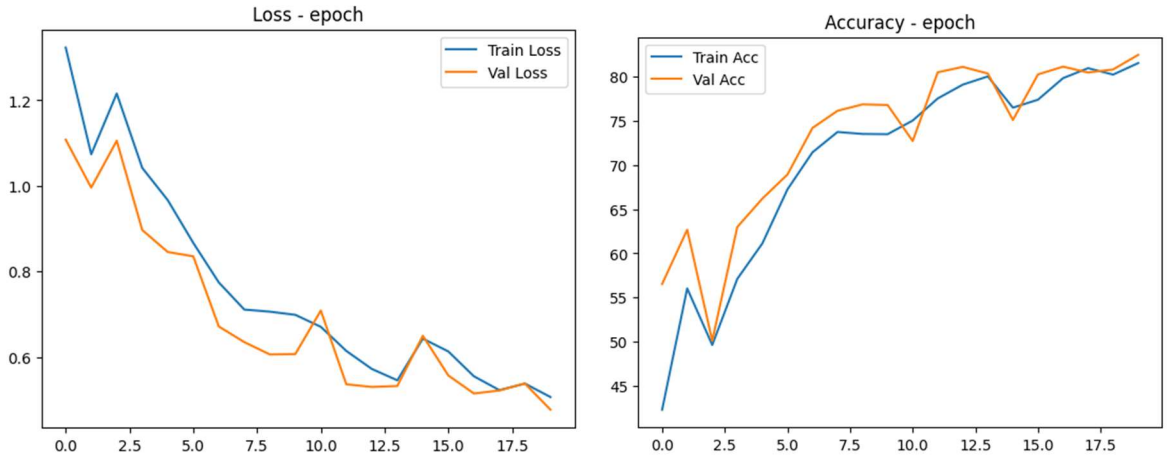
```
Epoch 1/15 | Train Loss: 0.556 Train Acc: 80.296% | Val Loss: 0.380 Val Acc: 86.833%
Epoch 2/15 | Train Loss: 0.370 Train Acc: 86.994% | Val Loss: 0.343 Val Acc: 87.550%
Epoch 3/15 | Train Loss: 0.329 Train Acc: 88.254% | Val Loss: 0.308 Val Acc: 89.117%
Epoch 4/15 | Train Loss: 0.279 Train Acc: 90.120% | Val Loss: 0.292 Val Acc: 89.717%
Epoch 5/15 | Train Loss: 0.261 Train Acc: 90.709% | Val Loss: 0.284 Val Acc: 90.400%
Epoch 6/15 | Train Loss: 0.247 Train Acc: 91.139% | Val Loss: 0.278 Val Acc: 90.233%
Epoch 7/15 | Train Loss: 0.219 Train Acc: 92.261% | Val Loss: 0.276 Val Acc: 90.400%
Epoch 8/15 | Train Loss: 0.208 Train Acc: 92.554% | Val Loss: 0.274 Val Acc: 90.617%
Epoch 9/15 | Train Loss: 0.202 Train Acc: 92.750% | Val Loss: 0.270 Val Acc: 90.733%
Epoch 10/15 | Train Loss: 0.185 Train Acc: 93.437% | Val Loss: 0.267 Val Acc: 91.117%
Epoch 11/15 | Train Loss: 0.180 Train Acc: 93.607% | Val Loss: 0.266 Val Acc: 91.250%
Epoch 12/15 | Train Loss: 0.176 Train Acc: 93.761% | Val Loss: 0.271 Val Acc: 91.283%
Epoch 13/15 | Train Loss: 0.167 Train Acc: 94.098% | Val Loss: 0.267 Val Acc: 91.217%
Epoch 14/15 | Train Loss: 0.165 Train Acc: 94.228% | Val Loss: 0.265 Val Acc: 91.317%
Epoch 15/15 | Train Loss: 0.162 Train Acc: 94.330% | Val Loss: 0.265 Val Acc: 91.433%
```

위의 사진은 그래프를 그리는데 사용되는 train loss, validation loss, train accuracy, validation accuracy 수치인데 수치의 변화를 보면 약 10 epoch부터 train은 지속적으로 loss가 감소하는 반면 validation loss는 거의 감소하지 않는다는 것을 확인할 수 있다. 10 epoch에서 validation loss가 0.267이었는데 15 epoch에서 validation loss가 0.265로 0.002만 감소한 것을 확인 가능하다. 이와 달리 train loss는 계속 감소하는 것을 볼 수 있는데 이는 일종의 train data에 대해 과적합이 일어나는 것이라고 볼 수 있다. 따라서 현재는 epoch을 15로 설정하였는데 10으로 설정하였어도 test 성능에는 큰 차이가 없을 것으로 예상되며 반대로 15보다 epoch을 더 늘리는 것은 test 성능을 높이기는커녕 오히려 과적합 때문에 test 성능을 낮출 수 있다고 생각해볼 수 있다. 결과적으로 해당 CRNN 모델을 통해 학습한 후 test 데이터에 대한 정확도는 아래의 사진과 같이 90.33%로, 과제의 기준이 되었던 88%를 넘겼다는 것을 확인할 수 있다.

```
Test Accuracy of RNN model on the 10000 test images: 90.33%
```

4. Accuracy를 올리기 위한 시도

1) Scheduler 사용



위의 사진은 epoch이 올라감에 따라 train과 validation의 loss와 accuracy를 나타내는 그래프이다. 하지만 결과 분석에서 살펴본 그래프와 달리 그래프가 들쑥날쑥한 현상을 확인할 수 있다. 이는 일종의 노이즈(noise)로 볼 수 있으며 이러한 현상이 나타나는 이유로 Learning rate가 높은 것을 생각해볼 수 있다. 따라서 기존에 0.001로 설정했던 Learning rate를 0.0005로 줄여서 설정하였으며, scheduler를 사용하여 epoch이 올라감에 따라 learning rate를 일정 비율 감소시켜 더 잘 수렴할 수 있도록 하였다.

```
criterion = nn.CrossEntropyLoss()                                    # 분류 문제이기에 CrossEntropy로 loss를 계산
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)   # optimizer는 adam으로 설정
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.5)
```

사용한 scheduler는 StepLR로 일정 step_size마다 gamma의 값을 learning rate에 곱해주는 방식으로 learning rate를 조절하는 역할을 한다. Step_size는 전체 epoch 15의 5분의 1인 3으로 설정하였으며 gamma는 0.5로 설정하여 매 3 epoch마다 learning rate가 절반이 되도록 설정하였다.

2) Dropout을 적용하여 일반화 성능 향상

드롭 아웃은 정해진 p확률만큼 학습 중에 뉴런의 연결을 제거, 혹은 비활성화하여 모델의 과적합을 방지하고 일반화 성능을 향상시키기 위해 사용하는 방법이다. 연산의 양이 증가함에 따라 모델이 학습 데이터에 대해서 과적합(Overfitting)되는 문제가 발생할 수 있다. 이번에 설계한 모델에서는 RNN layer에서 dropout을 0.4로 설정하였다. 이는 RNN layer에서 40%를 임의로 비활성화하여 모델의 과적합을 방지하는 역할을 하게 된다.

```
self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True, dropout=0.4)
```