HW3. Fashion-MNIST CRNN Classifier

20191611 유종선

- 이번 과제는 Fashion-MNIST 데이트를 CRNN을 사용해서 classify한다. 총 10종류의 이미지를 가진 데이터 셋이다. 데이터는 28 * 28 크기이다.
- 먼저 모델을 설명해보면, CRNN은 3개의 convolution layer와 2 fully-connected layer로 구성하였다.

  Layer 1 : in = 1, out = 32

  Layer 2 : in = 32, out = 64

  Layer 3 : in = 64, out = 128

  LSTM : embedding_dim : 64, hidden_dim : 128, num_layers = 3

  Fully-connected : 2304 -> 10(num_classes)
- CRNN

```
class CRNN(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, num_layers):
        super(CRNN, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels = in_channel, out_channels = 32, kernel_size = 3, padding = 1, stride = 2),
            nn.BatchNorm2d(num_features=32), #out_channels이 그대로 들어감
            nn.ReLU(),
            nn.Dropout(0.25),
            nn.MaxPool2d(kernel_size = max_pool_kernel, stride = 2) # 2 * 2
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels = 32, out_channels = 64, kernel_size = 3, padding = 1),
            nn.BatchNorm2d(num_features=64),
            nn.ReLU(),
            nn.Dropout(0.25),
            nn.MaxPool2d(kernel_size = max_pool_kernel)
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(in_channels = 64, out_channels = 128, kernel_size = 2, stride = 1, padding = 2),
            nn.BatchNorm2d(num_features = 128),
            nn.ReLU(),
            nn.Dropout(0.25),
            nn.MaxPool2d(kernel_size = max_pool_kernel)
        )
        #fully - connected
        self.fc1 = nn.Linear(in_features = 2304, out_features = 128)
        self.fc2 = nn.Linear(in_features = 128, out_features = 64)
        self.dropout1 = nn.Dropout(p = 0.25, inplace = False)
        self.fc3 = nn.Linear(in_features = 64, out_features = num_classes)

        self.lstm = nn.LSTM(input_size = embedding_dim, hidden_size = hidden_dim, num_layers=num_layers, batch_first=True)
```

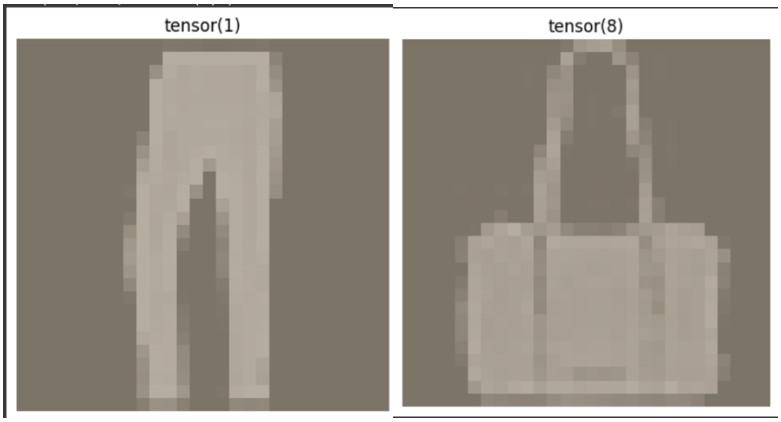
- Forward

```
def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)

    x = x.reshape(x.size(0), -1, embedding_dim)

    x, _ = self.lstm(x)

    x = x.reshape(x.size(0), -1) #fully connected에 넣어주기 위해서 flatten 시켜주기

    x = self.fc1(x)
    #x = self.dropout1(x)
    x = F.relu(x)
    x = self.fc2(x)
    x = F.relu(x)
    x = self.fc3(x)

    # sigmoid 해도 되고 안해도 되고
    return x # 10개의 출력 return
```

- Forward1, forward2 와 같이 forward 함수를 나누어 실행시켜서 단계 별로 이미지가 어떻게 변화하는지 확인해보자.

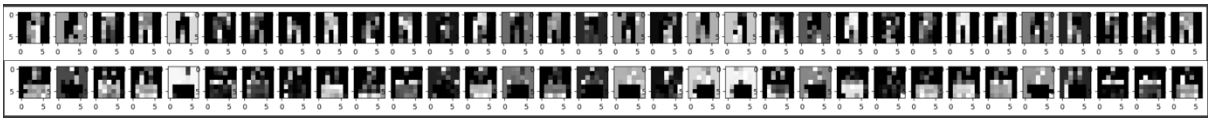
```
def forward1(self, x):
    x = self.layer1(x)
    return x

def forward2(self, x):
    x = self.layer2(x)
    return x

def forward3(self, x):
    x = x.reshape(x.size(0), -1, embedding_dim)
    x, _ = self.lstm(x)
    return x

def forward4(self, x):
    x = x.reshape(x.size(0), -1)
    x = self.fc1(x)
    return x

def forward5(self, x):
    x = self.fc3(self.fc2(x))
    return x
```

1. 먼저 train_loader에 들어간 image를 2개만 뽑아보았다. 이미지는 다음과 같다.



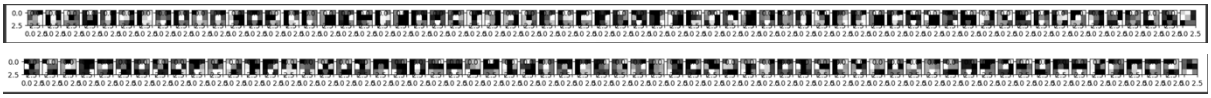
2. Forward1을 실행시키고 난 후에 image 사이즈의 변화이다.

```
torch.Size([64, 1, 28, 28]) --> torch.Size([64, 32, 7, 7])
```



3. Forward2를 실행시키고 난 후이다.

```
torch.Size([64, 32, 7, 7]) --> torch.Size([64, 64, 3, 3])
```



4. Forward3 이후 부터는 이미지로 볼 수 있는 데이터 크기가 아니라는 오류가 뜬다. Image를 print해서 확인해보자.

```
print(image1)

tensor([[-0.0028,  0.0130, -0.0245,  ..., -0.0090,  0.0170, -0.0183],
        [-0.0125,  0.0220, -0.0333,  ..., -0.0096,  0.0252, -0.0273],
        [-0.0211,  0.0297, -0.0330,  ..., -0.0072,  0.0288, -0.0306],
        ...,
        [-0.0574,  0.0319, -0.0447,  ..., -0.0056,  0.0365, -0.0463],
        [-0.0622,  0.0360, -0.0434,  ..., -0.0105,  0.0391, -0.0453],
        [-0.0637,  0.0349, -0.0438,  ..., -0.0140,  0.0414, -0.0442]],
       grad_fn=<SelectBackward0>)
```

5. Forward4와 forward5는 fully-connected이므로 모두 실행시키고, 예측결과와 정답을 출력해보자.

```
torch.Size([64, 9, 128]) --> torch.Size([64, 128])

torch.Size([64, 128]) --> torch.Size([64, 10])
```

```
image1
the final tensor : tensor([-0.0200,  0.1047, -0.0695,  0.0006,  0.0669,  0.0656, -0.0757, -0.1254,
        -0.0943, -0.1322], grad_fn=<SelectBackward0>)
the result prediction: 1
the original label: tensor(4)
```

6. 이러한 식으로 image가 처리되는 것을 확인할 수 있었다. 이제 train을 해보자. Train data set은 6만개이다. Valid test를 위해서 7 : 3의 비율로 데이터를 나누어 준다. Train 42000개, valid data는 18000개로 나누어진 모습을 확인 할 수 있다.

```
train_size = 0.7
val_size = 0.5

train_size = int(0.7 * len(train_data))
valid_size = len(train_data) - train_size
train_dataset, valid_dataset = random_split(train_data, [train_size, valid_size])
print(len(train_dataset), len(valid_dataset))
```

```
42000 18000
```

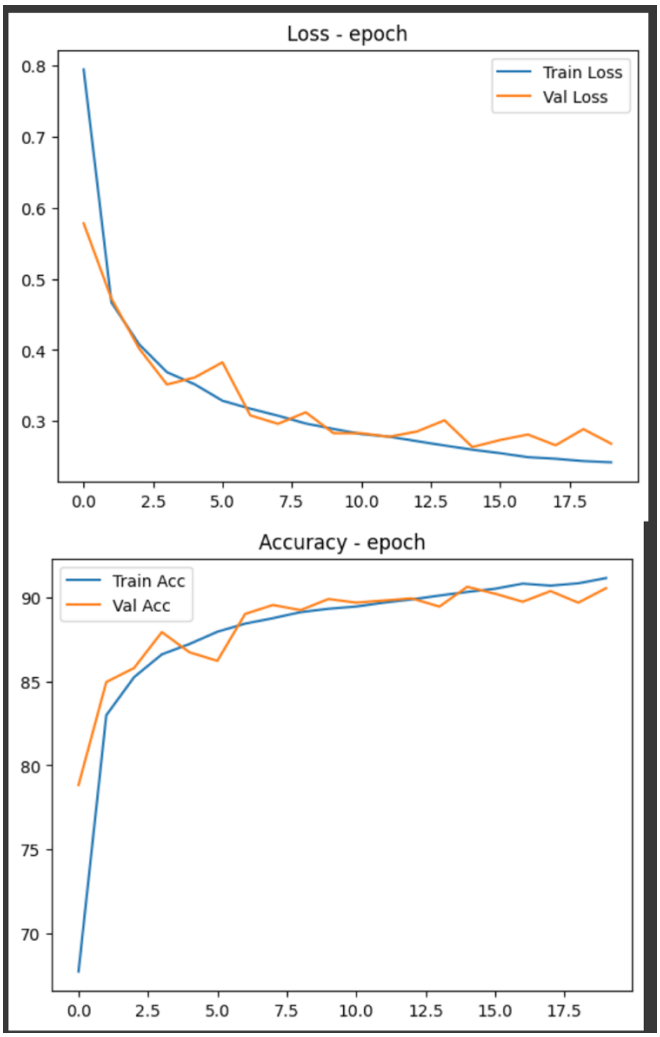
7. 데이터를 나누고, 하이퍼 파라미터는 다음과 같이 두었다.

```
# Model Hyperparamters
out_node = 1
embedding_dim = 64
hidden_dim = 128
num_layers = 3
dropout = 0.25
learning_rate = 0.001
num_epochs = 20
```

8. Train 실행

```
Training:   5%|▌         | 1/20 [00:16<05:02, 15.92s/it, Training batch 11/657]Epoch 1/20 | Train Loss: 0.795 | Train Acc: 0.677 | Val Loss: 0.578 Val Acc: 78.82777777777778%
Training:  10%|▉         | 2/20 [00:31<04:42, 15.71s/it, Training batch 11/657]Epoch 2/20 | Train Loss: 0.467 | Train Acc: 0.830 | Val Loss: 0.472 Val Acc: 84.96111111111111%
Training:  15%|█▍        | 3/20 [00:47<04:26, 15.69s/it, Training batch 11/657]Epoch 3/20 | Train Loss: 0.408 | Train Acc: 0.853 | Val Loss: 0.402 Val Acc: 85.79444444444444%
Training:  20%|█▉        | 4/20 [01:02<04:10, 15.68s/it, Training batch 10/657]Epoch 4/20 | Train Loss: 0.369 | Train Acc: 0.866 | Val Loss: 0.352 Val Acc: 87.93333333333334%
Training:  25%|██▍       | 5/20 [01:18<03:53, 15.57s/it, Training batch 11/657]Epoch 5/20 | Train Loss: 0.352 | Train Acc: 0.872 | Val Loss: 0.361 Val Acc: 86.72777777777779%
Training:  30%|██▉       | 6/20 [01:34<03:39, 15.65s/it, Training batch 8/657]Epoch 6/20 | Train Loss: 0.329 | Train Acc: 0.880 | Val Loss: 0.383 Val Acc: 86.22222222222223%
Training:  35%|███▍      | 7/20 [01:49<03:24, 15.70s/it, Training batch 8/657]Epoch 7/20 | Train Loss: 0.317 | Train Acc: 0.884 | Val Loss: 0.308 Val Acc: 89.01666666666667%
Training:  40%|███▉      | 8/20 [02:05<03:09, 15.79s/it, Training batch 10/657]Epoch 8/20 | Train Loss: 0.308 | Train Acc: 0.888 | Val Loss: 0.296 Val Acc: 89.55%
Training:  45%|████▍     | 9/20 [02:21<02:53, 15.73s/it, Training batch 10/657]Epoch 9/20 | Train Loss: 0.297 | Train Acc: 0.891 | Val Loss: 0.312 Val Acc: 89.24444444444445%
Training:  50%|████▉     | 10/20 [02:37<02:36, 15.70s/it, Training batch 11/657]Epoch 10/20 | Train Loss: 0.289 | Train Acc: 0.893 | Val Loss: 0.283 Val Acc: 89.90555555555557%
Training:  55%|█████▍    | 11/20 [02:52<02:20, 15.66s/it, Training batch 10/657]Epoch 11/20 | Train Loss: 0.282 | Train Acc: 0.895 | Val Loss: 0.283 Val Acc: 89.68888888888888%
Training:  60%|█████▉    | 12/20 [03:08<02:05, 15.72s/it, Training batch 8/657]Epoch 12/20 | Train Loss: 0.278 | Train Acc: 0.897 | Val Loss: 0.278 Val Acc: 89.81666666666666%
Training:  65%|██████▍   | 13/20 [03:25<01:51, 15.93s/it, Training batch 10/657]Epoch 13/20 | Train Loss: 0.272 | Train Acc: 0.899 | Val Loss: 0.285 Val Acc: 89.94444444444444%
Training:  70%|██████▉   | 14/20 [03:40<01:35, 15.93s/it, Training batch 11/657]Epoch 14/20 | Train Loss: 0.266 | Train Acc: 0.901 | Val Loss: 0.301 Val Acc: 89.45555555555555%
Training:  75%|███████▌  | 15/20 [03:56<01:19, 15.95s/it, Training batch 11/657]Epoch 15/20 | Train Loss: 0.260 | Train Acc: 0.903 | Val Loss: 0.263 Val Acc: 90.63333333333333%
Training:  80%|████████  | 16/20 [04:13<01:04, 16.03s/it, Training batch 8/657]Epoch 16/20 | Train Loss: 0.255 | Train Acc: 0.905 | Val Loss: 0.273 Val Acc: 90.22222222222223%
Training:  85%|████████▌ | 17/20 [04:29<00:48, 16.11s/it, Training batch 11/657]Epoch 17/20 | Train Loss: 0.249 | Train Acc: 0.908 | Val Loss: 0.281 Val Acc: 89.74444444444445%
Training:  90%|█████████ | 18/20 [04:45<00:32, 16.01s/it, Training batch 11/657]Epoch 18/20 | Train Loss: 0.247 | Train Acc: 0.907 | Val Loss: 0.266 Val Acc: 90.37777777777778%
Training:  95%|█████████▌| 19/20 [05:01<00:16, 16.16s/it, Training batch 11/657]Epoch 19/20 | Train Loss: 0.244 | Train Acc: 0.908 | Val Loss: 0.289 Val Acc: 89.68888888888888%
Training: 100%|██████████| 20/20 [05:18<00:00, 15.91s/it, Validation batch 281/282]Epoch 20/20 | Train Loss: 0.242 | Train Acc: 0.912 | Val Loss: 0.268 Val Acc: 90.55%
```

9. Train의 accuracy와 loss, valid의 accuracy와 loss의 변화를 그래프를 보고 변화를 확인해보자. Epoch이 기준이다.



10. 마지막으로 test를 해보자. 대략 89.6% 정도의 정확도를 가지게 된다.

```
Inference: 100%|■■■■■■■■■■| 157/157 [00:01<00:00, 94.96it/s]

Accuracy: 89.67% / Loss: 0.2955
```

11. 정확도를 올리기 위한 과정

- Conv2d layer : 처음에는 convolution layer를 2로 두고 실행하였는데, layer를 하나 더 늘려서 모델을 구성하였다.
- Maxpooling : 현재는 3 layers 에서 maxpooling을 3번 진행한다. 그런데 현재 입력 데이터가 28 * 28 이기 때문에, 더 적게 maxpooling을 진행해보았다. 더 많이 Maxpooling을 할 수록 정보를 너무 많이 잃을 수도 있겠다는 예상을 했다.
  - Maxpooling *2

```
Inference: 100%|■■■■■■■■■■| 157/157 [00:02<00:00, 77.47it/s]

Accuracy: 88.73% / Loss: 0.3053
```

  3번을 두번으로 줄인 결과 더 낮은 결과값이 나왔다. Maxpooling은 3번으로 유지하도록 하였다.

- Filter size : 필터 사이즈를 조절하였다. 현재 filter size는 3, 3 ,2 이렇게 진행하고 있다. Filter size가 크게 되면 이미지에서 더 큰 패턴이나 특징을 잡을 수 있는데 반해, 세부적인 정보를 놓칠 염려도 있다. 그래서 필터사이즈를 조절해서 실험을 해보았다.
  - Filter size = 2, 2 ,2

```
Inference: 100%|■■■■■■■■■■| 157/157 [00:02<00:00, 54.69it/s]

Accuracy: 86.1% / Loss: 0.3906
```

  - Filter size = 3, 3, 3

```
Inference: 100%|■■■■■■■■■■| 157/157 [00:01<00:00, 96.36it/s]

Accuracy: 88.13% / Loss: 0.3283
```

  - Filter size = 5, 5, 5

```
Inference: 100%|■■■■■■■■■■| 157/157 [00:01<00:00, 97.19it/s]

Accuracy: 87.68% / Loss: 0.3293
```

  다음의 결과를 통해서, 그대로 filter_size를 유지하였다.

- LSTM에서 hidden dimesion의 설정에 따른 결과 값을 확인해보았다. 현재는 128로 진행하였는데, 이 값을 256으로 높여서 확인해보자.
  - hidden = 256

```
Inference: 100%|■■■■■■■■■■| 157/157 [00:02<00:00, 65.71it/s]

Accuracy: 90.06% / Loss: 0.2808
```

조금 더 높은 성능을 받을 수 있었다.

- Dropout : 오버피팅을 방지하고, 앙상블 메소드 효과를 주기 위해서 dropout 함수를 사용하였다. 여기서 drop out을 크기를 조절해보자
  - ■ Drop out : 0.1

```
Inference: 100%|■■■■■■■■■■| 157/157 [00:02<00:00, 76.17it/s]

Accuracy: 90.28% / Loss: 0.2841
```

  - ■ Drop out : 0.1, 3번의 dropout을 1번으로 줄여서 학습

```
Inference: 100%|■■■■■■■■■■| 157/157 [00:01<00:00, 88.85it/s]

Accuracy: 89.83% / Loss: 0.3319
```

Dropout 실행 결과 Drop out은 0.1로 3번 진행하는 것이 가장 좋은 결과를 얻을 수 있었다.

➔ hidden dimension = 256, Dropout = 0.1 로 하는 것이 성능을 높혀주었다.