

Course Title (과목명)	인공지능통신 (01)
HW Number (HW 번호)	Project 1
Submit Date (제출일)	2021-05-12
Grade (학년)	4학년
ID (학번)	20151483
Name (이름)	이창헌

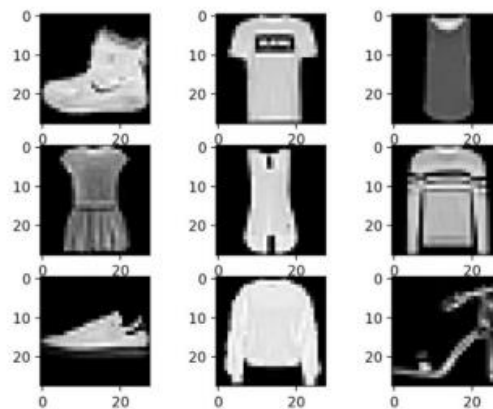
Project 1

1. 과제 목표

- 본 프로젝트는 주어진 데이터셋에 맞춰 신경회로망을 학습하는 것을 목표로 함.
- 사이버 캠퍼스에는 학습 데이터만 업로드 되며, 테스트 데이터는 공개하지 않음
- 조별로 학습이 끝난 모델을 제출하면 해당 모델을 기반으로 테스트를 진행함.

2. 모델 구조

1) Dataset



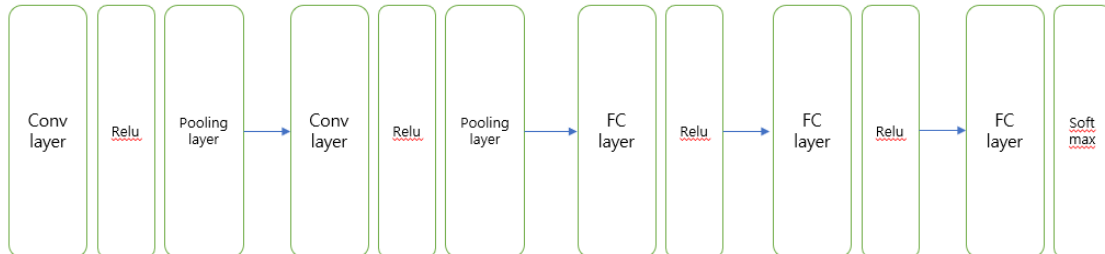
- 0 T-shirt/top
- 1 Trouser
- 2 Pullover
- 3 Dress
- 4 Coat
- 5 Sandal
- 6 Shirt
- 7 Sneaker
- 8 Bag
- 9 Ankle boot

Alcomm_project_1_train.csv 파일은 흑백으로 된 60000개의 28 x 28 크기의 패션 관련 데이터 셋으로 구성되어 있다. 각각 sample은 0~9의 10가지 class로 label 되어 있고 흑백 이미지는 1 channel임으로 1x28x28의 크기를 가지고 있다.

이번 프로젝트에서는 60000개의 data를 4:1의 비율로 training dataset과 validation dataset으로 나눠서 모델을 학습시켰다.

2) CNN (Convolutional Neural Network)

CNN(Convolutional neural network) 구조는 convolutional layer와 pooling layer를 활용하는 network으로 보통 여러 계층의 조합으로 만든다.



위의 CNN으로 만든 Model 1의 예시로 두 개의 CNN layer와 3개의 FC(fully connected) layer로 구성되어 있다.

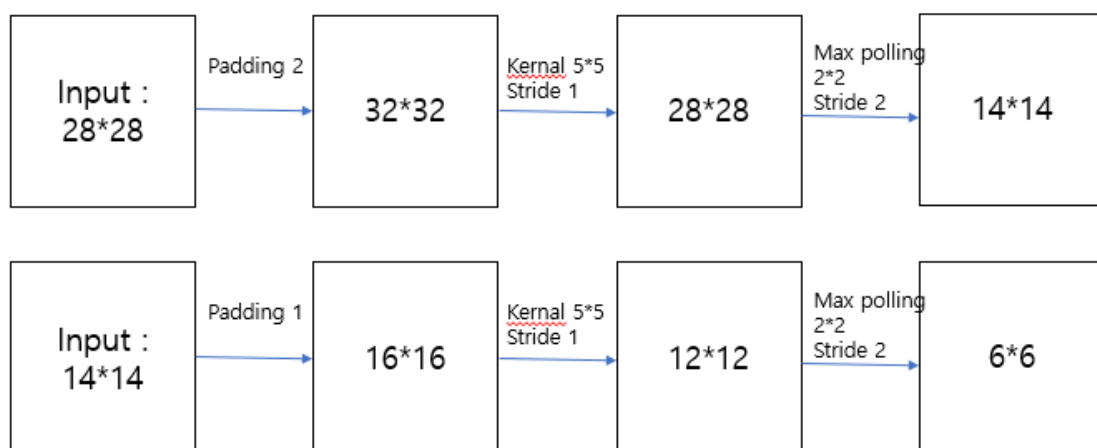
각 CNN layer에서는 패딩과 스트라이드 값에 따라서 convolution 연산으로 이미지를 처리 즉 필터한다.

패딩은 convolution 연산을 수행하기 전에 입력 데이터 주변을 0으로 채운다.

커널은 필터로서 입력을 커널의 크기에 따라서 convolution을 연산하고 출력한다. 스트라이드는 이때 필터를 적용하는 위치 사이의 간격을 말한다.

풀링은 풀링의 영역 크기안에서 특정 값 하나를 저장하는 필터로 max pooling이라고하면 가장 큰 값을 저장한다.

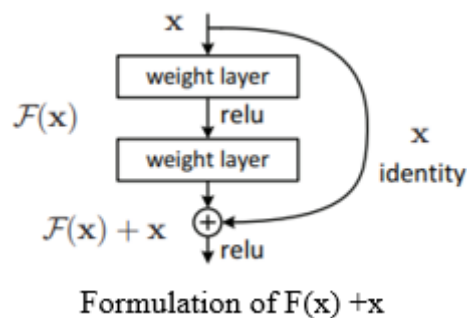
위의 예시에서 첫 번째 CNN layer의 kernel이 5*5 with stride 1, padding은 2, max pooling with stride 2라고하면 EMNIST 입력 28*28은 CNN layer를 거쳐서 14*14로 출력된다.



2) ResNet (Residual Neural Network)

Deep model을 stacking 하는 것을 통해서 network는 더 많은 features를 학습할 수 있다. 그러나 deeper models는 동시에 vanishing/ exploding gradients와 degradation의 문제를 발생시킨다. 이 중에서 vanishing/ exploding gradients는 initial normalization과 batch normalization을 사용해서 intermediate normalization을 수행해서 해결할 수 있으나 degradation은 여전히 문제로 남는다.

ResNet은 network에 shortcut connections을 더해서 이전 단계의 layers의 identity mapping이 stacked layers의 output에 더해지는 $F(x) + x$ formulation으로 degradation을 해결한다.



```
#ResNet20 residual net
class ResNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ResNet, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channel, conv_channel1, 3, stride=1, padding=1),
            nn.BatchNorm2d(conv_channel1)
        )
        self.layer2 = nn.Sequential(
            nn.Conv2d(conv_channel1, conv_channel1, 3, stride=1, padding=1),
            nn.BatchNorm2d(conv_channel1)
        )

        #layer3s2 is only used once for downsampling
        self.layer3s2 = nn.Sequential(
            nn.Conv2d(conv_channel1, conv_channel2, 3, stride=2, padding=1),
            nn.BatchNorm2d(conv_channel2)
        )
        self.layer3 = nn.Sequential(
            nn.Conv2d(conv_channel2, conv_channel2, 3, stride=1, padding=1),
            nn.BatchNorm2d(conv_channel2)
        )

        #layer4s2 is only used once for downsampling
        self.layer4s2 = nn.Sequential(
```

```

        nn.Conv2d(conv_channel2, conv_channel3, 3, stride=2, padding=1),
        nn.BatchNorm2d(conv_channel3)
    )
    self.layer4 = nn.Sequential(
        nn.Conv2d(conv_channel3, conv_channel3, 3, stride=1, padding=1),
        nn.BatchNorm2d(conv_channel3)
    )

    self.fc = nn.Sequential(
        nn.Linear(8*8*conv_channel3, num_classes),
        nn.BatchNorm1d(num_classes)
    )

def forward(self, x):
    #layer1
    x = F.relu(self.layer1(x))

    #layer2
    shortcut = self.layer2(x)
    x = F.relu(self.layer2(x))
    x = self.layer2(x)
    x = F.relu(x+shortcut)
    shortcut = self.layer2(x)
    x = F.relu(self.layer2(x))
    x = self.layer2(x)
    x = F.relu(x+shortcut)
    shortcut = self.layer2(x)
    x = F.relu(self.layer2(x))
    x = self.layer2(x)
    x = F.relu(x+shortcut)

    #layer3
    shortcut = self.layer3s2(x)
    x = F.relu(self.layer3s2(x))
    x = self.layer3(x)
    x = F.relu(x+shortcut)
    shortcut = self.layer3(x)
    x = F.relu(self.layer3(x))
    x = self.layer3(x)
    x = F.relu(x+shortcut)
    shortcut = self.layer3(x)
    x = F.relu(self.layer3(x))
    x = self.layer3(x)
    x = F.relu(x+shortcut)

    #layer4
    shortcut = self.layer4s2(x)
    x = F.relu(self.layer4s2(x))

```

```

x = self.layer4(x)
x = F.relu(x+shortcut)
shortcut = self.layer4(x)
x = F.relu(self.layer4(x))
x = self.layer4(x)
x = F.relu(x+shortcut)
shortcut = self.layer4(x)
x = F.relu(self.layer4(x))
x = self.layer4(x)
x = F.relu(x+shortcut)

#avg_pool with stride 1 and a 10-way fully-connected layer with softmax
x = F.avg_pool2d(x, 1)
x = x.reshape(x.size(0), -1)
x = F.softmax(self.fc(x))
return x

```

프로젝트에서 사용하기 위해서 구현해본 20개의 layers로 구성된 ResNet20의 architecture은 위와 같다.

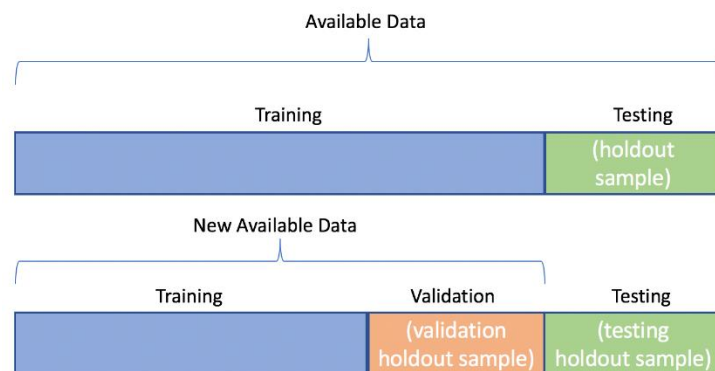
주어진 dataset에서 각 image data의 크기가 28*28으로 매우 작기 때문에 이에 맞춰서 convolutional layer의 kernel와 stride, padding을 작게 설정하였다.

Down sampling을 위해서 pooling을 사용하는 일반적인 CNN과 다르게 ResNet은 stride가 2인 convolution layer를 사용해서 down sampling을 수행한다. 주어진 dataset은 10개의 class로 구성되어 있기 때문에 마지막 fc dense layer의 출력은 10이다.

최종적으로는 torchvision library에서 제공하는 ImageNet으로 pretrain된 ResNet18을 사용하는 것이 validation dataset에 더 높은 performance를 가져서 ResNet18을 채택하였다.

3. 과제 수행 방법

(a) Neural Network 선정



주어진 training dataset에서 60000의 sample을 4:1의 비율로 나눠서 일부를 validation dataset으로 사용하였다.

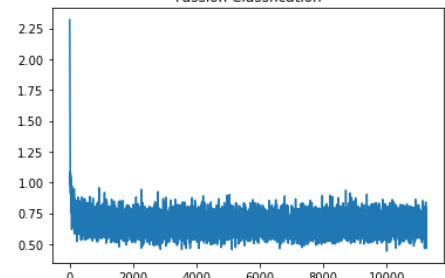
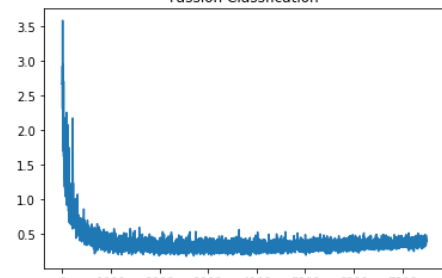
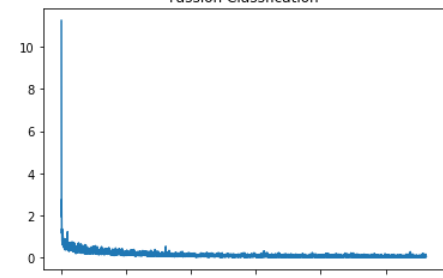
```
1 #data augmentation
2
3 transforms_augmentation = transforms.Compose([transforms.Resize((28,28)),
4         transforms.RandomCrop(28, padding= 4),
5         transforms.RandomHorizontalFlip(),
6         transforms.ToTensor(),
7         transforms.Normalize((0.5,0.5,0.5),(0.5,0.5,0.5))
8 ])
```

[Data augmentation]

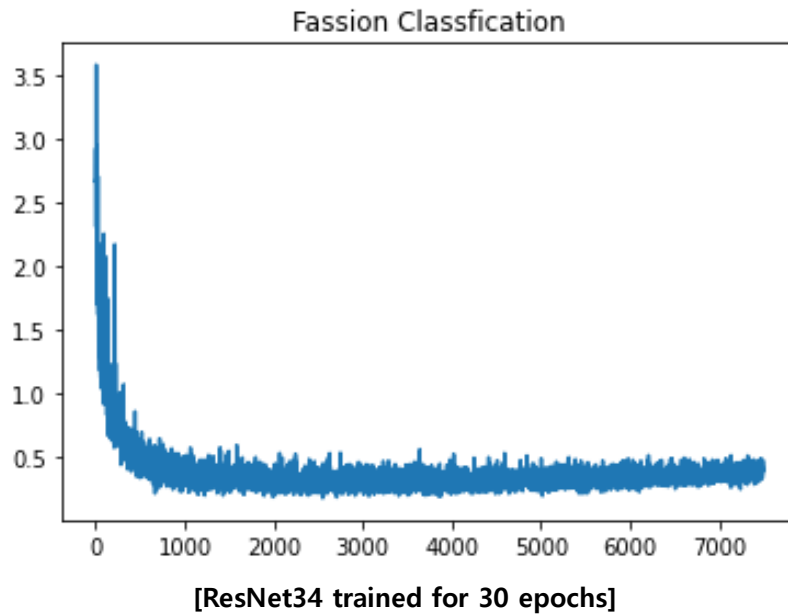
또한 학습과정에서 training dataset에 overfitting되는 것을 완화하고 제한된 개수의 training dataset에서 더 많은 features들을 학습해서 general data에 대한 performance를 향상시키기 위해서 training dataset에 data augmentation을 수행하였다. 사용한 data augmentation method는 아래와 같다.

- Random crop을 사용해서 28*28 size의 data에 4의 padding을 추가하고 그 중에서 28*28 영역을 랜덤하게 crop한다.
- Random Horizontal Flip을 사용해서 확률적으로 랜덤하게 일부의 data들은 수평적으로 flip한다

CNN, ResNet34, DenseNet121을 선정하여서 training dataset을 사용해서 학습하고 학습이 끝난 model의 performance를 검증하는 단계에서 validation dataset을 사용하였다. 또한 unseen data에 대해 가장 performance가 우수한 model을 고르기 위해서 epoch마다 validation accuracy를 확인하여 accuracy가 가장 높은 model을 저장하였다

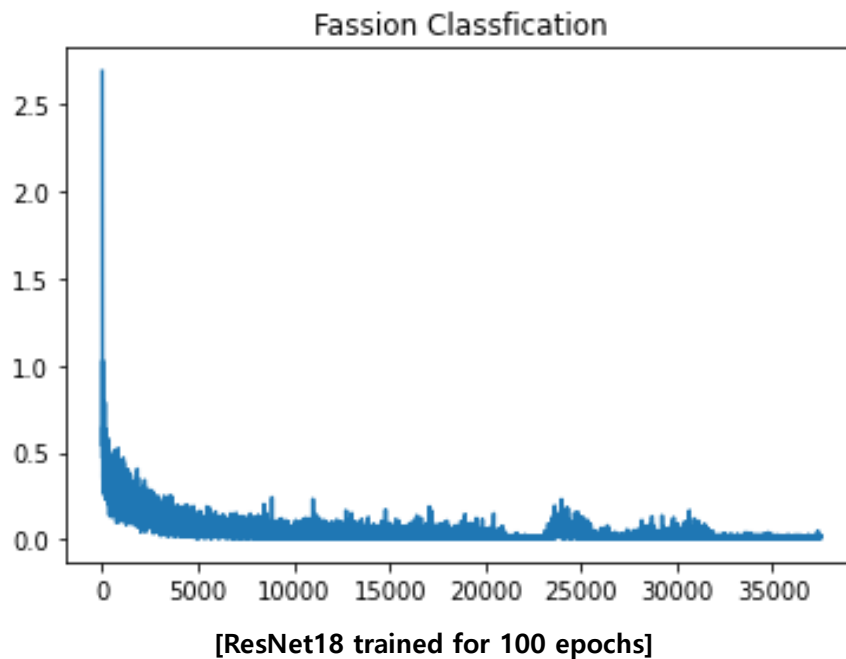
CNN	ResNet34	DenseNet121
<p>Fassion Classification</p>  <p>Validation accuracy : 85.16%</p>	<p>Fassion Classification</p>  <p>Validation accuracy : 91.68%</p>	<p>Fassion Classification</p>  <p>Validation accuracy : 92.43%</p>

각 Network 별로 30 epoch의 학습을 수행해본 결과 모든 epoch 마다 validation accuracy의 평균을 내보면 DenseNet121의 performance가 가장 뛰어났다.



그러나 DenseNet121은 121개의 layer들로 구성된 network로 학습에 지나치게 오랜 시간이 걸린다. ResNet34으로 training하면서 발생한 loss를 표현한 위의 그래프를 자세히 보면 15 epoch 정도가 경과하고 나서는 overfitting으로 인해서 ResNet34에서 validation loss가 오히려 늘어나고 있는 것을 알 수 있다.

따라서 model의 complexity가 더 낮아서 overfitting이 발생하는데 더 많은 epoch가 필요한 ResNet18을 대상으로 100 epoch의 학습을 수행해보았다.



그 결과 위의 그림과 같이 ResNet18은 overfitting으로 인해서 performance on generalization data가 떨어지는 상황이 발생하지 않아서 Dense121만큼 평균적인 validation accuracy가 높으면서도

18개의 layer로 구성되어 있기 때문에 학습 속도가 매우 빨랐다.

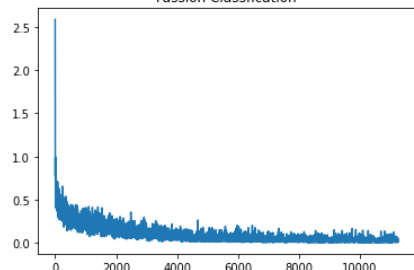
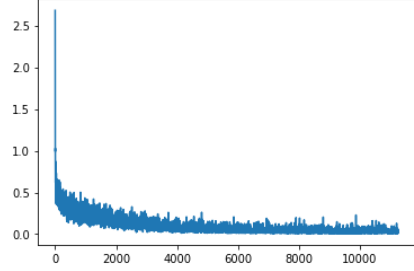
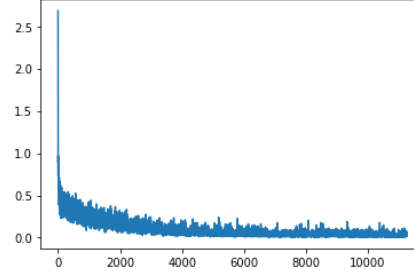
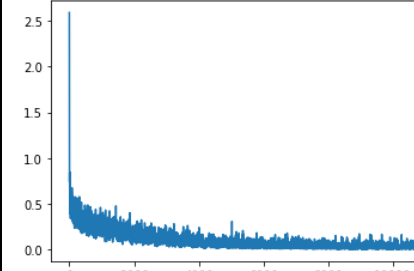
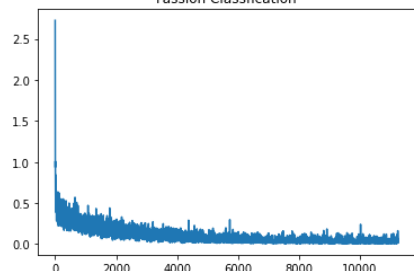
ResNet18을 최종적인 network로 선정하고 hyperparameter들을 바꿔가면서 실험해보았다.
ResNet18을 사용한 model의 구조와 image tensor의 dimension은 아래와 같다.

ResNet18 Architecture			[C, W, H]	비고
Input			[1, 28, 28]	
1	Conv1	Conv2d (kernel: 3, padding:1)	[64, 28, 28]	
2	Layer1 block0	Conv2d (kernel: 3, padding:1)	[64, 28, 28]	
3		Conv2d (kernel: 3, padding:1)	[64, 28, 28]	
4	Layer1 block1	Conv2d (kernel: 3, padding:1)	[64, 28, 28]	
5		Conv2d (kernel: 3, padding:1)	[64, 28, 28]	
6	Layer2 block0	Conv2d (kernel: 3, stride: 2, padding:1)	[128, 14, 14]	
7		Conv2d (kernel: 3, padding:1)	[128, 14, 14]	down sample
8	Layer2 block1	Conv2d (kernel: 3, padding:1)	[128, 7, 7]	
9		Conv2d (kernel: 3, padding:1)	[128, 7, 7]	
10	Layer3 block0	Conv2d (kernel: 3, stride: 2, padding:1)	[256, 4, 4]	
11		Conv2d (kernel: 3, padding:1)	[256, 4, 4]	down sample
12	Layer3 block1	Conv2d (kernel: 3, stride: 2, padding:1)	[256, 2, 2]	
13		Conv2d (kernel: 3, padding:1)	[256, 2, 2]	
14	Layer4 block0	Conv2d (kernel: 3, stride: 2, padding:1)	[512, 1, 1]	
15		Conv2d (kernel: 3, padding:1)	[512, 1, 1]	
16	Layer4 block1	Conv2d (kernel: 3, padding:1)	[512, 1, 1]	
17		Conv2d (kernel: 3, padding:1)	[512, 1, 1]	
18	Conv1	Conv2d (kernel: 1, padding:1)	[512, 1, 1]	average pooling
	fc	Dense	[512, 10]	

(c) Hyper parameters 선정

Learning rate, number of epochs, batch size 등의 hyperparameter들을 바꿔가면서 실험해보았다

Batch	Learning rate = 0.001	Learning rate = 0.005	Learning rate = 0.01
-------	-----------------------	-----------------------	----------------------

size			
64			
	Validation accuracy : 96.08%		
128			
	Validation accuracy : 96.88%	Validation accuracy : 93.23%	Validation accuracy : 94.43%
256			
	Validation accuracy : 95.31%		

먼저 learning rate = 0.001일 때 batch size를 64, 128, 256으로 바꿔가면서 학습해보았다. 각각 30 epoch씩 학습시키고 best epoch의 validation accuracy를 비교해보면 batch size = 128일 때가 가장 높았다.

이제 batch size = 128일 때 learning rate을 0.001, 0.005, 0.01로 바꿔가면서 학습해보았다. 각각 30 epoch씩 학습시키고 best epoch의 validation accuracy를 비교해보면 learning rate = 0.01일 때가 가장 높았다.

위의 표에서 그래프는 학습이 경과하면서 loss의 변화를 그래프로 그린 것이다. Learning rate이 높으면 더 빠르게 loss가 줄어들지만 여러 epoch가 지난 후에도 loss가 안정되지 못하고 크게 증감을 반복하는 것을 알 수 있다. 서로 다른 batch size에 대해서는 loss의 그래프로는 큰 차이를 볼 수 없지만 validation accuracy로 비교해보면 batch size가 128일 때 unseen data에 대한 performance가 가장 뛰어난 것을 알 수 있다.

```
Epoch [31/100], Loss: 0.0228, Train Accuracy: 100.00%, Valid Accuracy: 91.41%
Epoch [32/100], Loss: 0.0378, Train Accuracy: 100.00%, Valid Accuracy: 93.75%
Epoch [33/100], Loss: 0.0169, Train Accuracy: 100.00%, Valid Accuracy: 88.28%
Epoch [34/100], Loss: 0.0427, Train Accuracy: 99.22%, Valid Accuracy: 92.97%
Epoch [35/100], Loss: 0.0307, Train Accuracy: 100.00%, Valid Accuracy: 96.09%
Epoch [36/100], Loss: 0.0139, Train Accuracy: 100.00%, Valid Accuracy: 93.75%
Epoch [37/100], Loss: 0.0651, Train Accuracy: 100.00%, Valid Accuracy: 89.84%
Epoch [38/100], Loss: 0.0139, Train Accuracy: 100.00%, Valid Accuracy: 92.19%
Epoch [39/100], Loss: 0.0384, Train Accuracy: 100.00%, Valid Accuracy: 92.97%
Epoch [40/100], Loss: 0.0361, Train Accuracy: 99.22%, Valid Accuracy: 88.28%
Epoch [41/100], Loss: 0.0501, Train Accuracy: 96.88%, Valid Accuracy: 93.75%
Epoch [42/100], Loss: 0.0778, Train Accuracy: 100.00%, Valid Accuracy: 91.41%
Epoch [43/100], Loss: 0.0214, Train Accuracy: 100.00%, Valid Accuracy: 90.62%
Epoch [44/100], Loss: 0.0566, Train Accuracy: 100.00%, Valid Accuracy: 90.62%
Epoch [45/100], Loss: 0.0581, Train Accuracy: 99.22%, Valid Accuracy: 92.19%
Epoch [46/100], Loss: 0.0211, Train Accuracy: 99.22%, Valid Accuracy: 94.53%
Epoch [47/100], Loss: 0.0160, Train Accuracy: 100.00%, Valid Accuracy: 90.62%
Epoch [48/100], Loss: 0.0689, Train Accuracy: 99.22%, Valid Accuracy: 92.19%
Epoch [49/100], Loss: 0.0303, Train Accuracy: 100.00%, Valid Accuracy: 88.28%
Epoch [50/100], Loss: 0.0118, Train Accuracy: 99.22%, Valid Accuracy: 92.97%
```

[Epoch 30-50]

Number of epoch를 선정하는 것에 대해서는 100 epoch 동안 학습시키면서 살펴본 결과 약 30 epoch정도 진행되고 나면 training dataset에 대한 accuracy가 100%에 도달하고 validation dataset에 대한 accuracy는 오히려 감소하는 것을 알 수 있었다. 따라서 epoch가 30을 넘어가면 training dataset에 대해서 overly optimize되어서 general data에 대한 performance가 떨어진다고 판단해서 30 epoch동안 학습되도록 정했다.

결과적으로 선정된 hyper parameters는 batch size = 128, learning rate = 0.001, number of epochs = 30이다.

3. Test를 위한 코드

모델 저장, 불러오기 기능을 활용해서 학습 데이터와 동일한 형태의 csv 파일로 test dataset을 입력하면 테스트 결과를 낼 수 있도록 코드를 구성하고 test.ipynb 파일로 제출하였다.

이때 colab에 gdrive를 mount해서 사용한다는 가정하에 코드를 작성하였기 때문에 csv 파일의 directory와 저장된 모델 pth 파일의 directory부분을 수정해서 사용해야한다.

```
1 #load test_data from csv
2
3 csv_test = '/content/gdrive/MyDrive/AIC/AIcomm_project_1_test.csv'
```

[test.csv directory example]

```
1 #load trained model and evaluate model on test data
2 test_model = ResNet(1,resnet18,10).to(device)
3 test_model.load_state_dict(torch.load('/content/gdrive/MyDrive/Fassion_Classification.pth'))
```

[model.pth directory example]

4. Reference

- 1) *Learning from Data: A Short Course*, Yaser S. Abu-Mostafa, AMLbook
- 2) *Deep Residual Learning for Image Recognition*, Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun
- 3) *Deep Learning from Scratch*, Saito Goki, (O'REILLY, 2016)
- 4) *Batch normlization: Accelerating deep network training by reducing internal covariate shift*, Ioffe and Szegedy.. 2015