인공지능(딥러닝)개론 Project 결과보고서

Alphabet & Digit Classifier using CNN

학번: 20180032 이름: 남기동

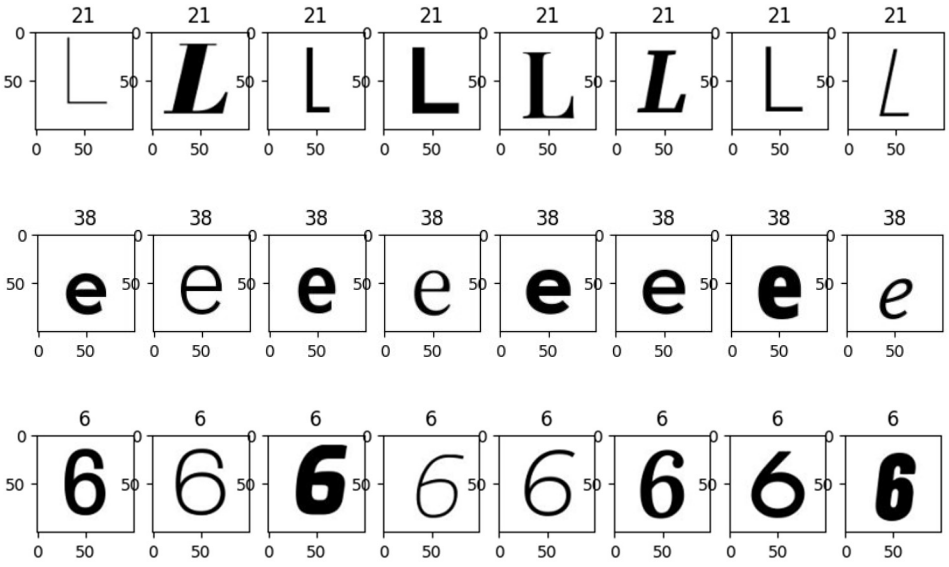
1. 과제 목표 및 데이터셋

인공지능(딥러닝)개론의 마지막 프로젝트의 목표는 다양한 폰트로 구성된 영문자 및 숫자를 분류하는 문제를 딥러닝을 이용하여 해결하는 것이다. 해당 과제에 사용되는 데이터셋은 0~9까지의 숫자와 소문자와 대문자를 합하여 42개가 포함되어 총 클래스가 52개인 데이터셋이다. 알파벳의 경우 소문자와 대문자 각각 26개씩 총 52개가 있지만 ["c", "k", "l", "O", "p", "s", "x", "z"]의 경우 대응되는 대문자 혹은 소문자와 구분이 어렵기 때문에 해당 클래스 10개를 제외하여 알파벳에서 42개의 클래스, 숫자에서 10개의 클래스가 합해져 총 52개의 클래스로 구성되어 있는 것이다. 이미지는 1 x 100 x 100의 픽셀로 흑백 이미지이며 Train 데이터로 41,600개, Validation 데이터로 15,600개가 제공되었으며 채점에 사용되는 Test 데이터는 공개되지 않았다.

```
# check dataloader
image,label = next(iter(valid_loader))
print(image.shape)
print(label.shape)
```

```
torch.Size([50, 1, 100, 100])
torch.Size([50])
```

이미지는 아래의 사진과 같이 같은 클래스에 해당하더라도 다양한 모습으로 저장되어 있다.



2. 과제 해결 아이디어

1) CNN 모델 선정 이유 1 - 이미지 분류

우선, 이번 과제에서 분류해야 하는 숫자 이미지, 알파벳 이미지는 MNIST 데이터셋에서 분류했던 것과 크게 다르지 않다. 클래스가 다양해지고 손글씨인 것이 다양한 형태의 이미지로 나타날 뿐 기본적으로 간단한 이미지를 분류한다는 점에서는 크게 차이점이 없다. 그리고 크게 수업에서 다루었던 딥러닝 모델들 중에서 이미지 분류에 적합한 것은 공간적 특성을 포착할 수 있는 CNN(Convolution Neural Network)이다. 수업 후반부에서 다루고 수업했던 RNN, LSTM, GRU 등의 모델도 있지만 이러한 모델들은 Sequence 데이터에 적합한 모델이기 때문에 이번 과제를 해결하는데 있어서는 기본적으로 CNN 모델을 사용할 것이라 결론을 내렸다.

2) CNN 모델 선정 이유 2 – 학습 시간과 파라미터 수의 제한

다음으로, 이번 과제에서 중요한 제약 조건으로 작용하는 것이 바로 학습 시간이 15분 이내로 제한되었다는 것과 파라미터의 수가 1,200,000 이하로 제한되었다는 것이다. 이처럼 학습 시간과 파라미터 수의 제한이 생긴 만큼 기존의 Fully Connected Network 보다 Convolution Neural Network를 사용해야 할 이유가 추가되었다. CNN은 filter를 사용하여 Fully Connected Network에 비해 적은 파라미터 수와 비교적 짧은 학습 시간을 가능하게 하기 때문이다.

3) BottleNeck - 학습 시간과 파라미터 수를 줄이기 위한 추가적인 조치

하지만 CNN을 사용함에 있어서도 다른 과제에 비해 학습 시간이 짧은 만큼 시간을 단축시킬 수 있기 위한 추가적인 조치가 필요하다고 생각했다. 그리고 시간을 단축하기 위해서는 모델이 train data에 대해 제대로 학습을 하면서도 연산량을 줄일 수 있어야 하기 때문에 BottleNeck 구조를 사용하고자 했다. 1 x 1 Convolution Layer를 활용하여 차원을 축소한 이후 기존의 Convolution layer를 이용하여 공간적인 특징을 추출한 다음 다시 1 x 1 Convolution Layer를 활용하여 차원을 되돌리는 과정을 통해 파라미터의 수를 감소시킬 수 있다는 장점이 있기 때문이다.

4) 데이터 양 부족에 따른 Overfitting 우려

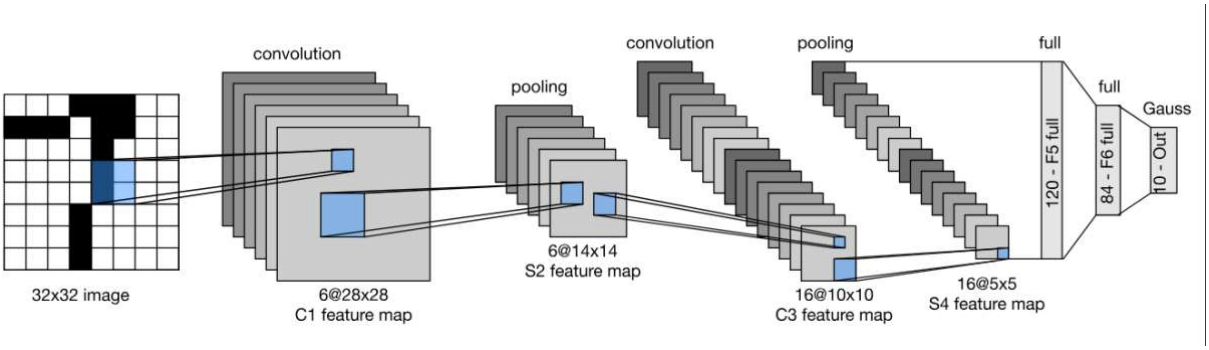
마지막으로 고려해야 할 점은 학습에 사용되는 데이터의 수가 현저히 적다는 것이다. 기존의 MNIST, CIFAR10 데이터셋을 보더라도 기본적으로 6만 개 정도의 데이터를 통해 학습을 진행하였다. 이번 과제에서 사용하는 데이터가 4만 개 정도이니 크게 문제가 될 수준은 아니라고 생각할 수 있지만 클래스의 수도 고려해야 한다. MNIST, CIFAR10의 경우 클래스가 10개이기 때문에 6만 개의 train 데이터에서 실질적으로 각각의 클래스에 대해 6천개 정도의 데이터가 있다고 볼 수 있다. 하지만 이번 과제에

서 사용되는 데이터는 클래스가 52개이기 때문에 각각의 클래스에 대해 학습에 사용할 수 있는 데이터의 개수는 (41,600 / 52)로 800개에 불과하다는 것을 알 수 있다. 즉 데이터의 개수가 적기 때문에 Overfitting 방지하기 위한 조치가 필요하다.

3. 배경 이론

1) Convolution Neural Network

Convolution Nerual Network는 주로 이미지를 분류하거나 패턴을 인식하는데 사용되는 딥러닝 알고리즘 중 하나로 특히나 이미지 처리 작업에서 탁월한 성능을 보인다. 우선, CNN은 Convolution layer, Pooling Layer, Fully Connected Layer로 구성된다. Convolutional Layer(합성곱층)은 CNN의 핵심 구성요소 중 하나로, 입력 데이터에 대해 필터(커널)를 이용해 합성곱 연산을 수행한다. 이를 통해 입력 이미지에서 특징을 추출하며 필터는 이미지의 작은 부분에 대응되는 가중치를 가지고 있다. 다음으로, Pooling Layer(풀링층)은 합성곱층에서 추출된 feature map의 크기를 줄이기 위해서 사용된다. 일반적으로 Convolution layer에서는 stride와 padding을 사용해서 이미지의 크기를 유지하고 pooling layer에서 feature map의 크기를 줄인다. 일반적으로 Max Pooling, Average Pooling이 사용되고 Max Pooling은 각 영역에서 가장 큰 값을, Average Pooling에서는 평균 값을 추출해서 Feature Map을 다운 샘플링한다. 마지막으로 Fully Connected layer(완전 연결층)은 CNN의 최종 부분에서 사용되며 Convolution layer – Pooling Layer를 통해 얻은 특징들을 사용해서 최종 출력을 생성하는데 사용된다.



위의 사진은 수업 자료 11장의 LeNet에서 사용된 Convolution layer, pooling layer, fully connected layer의 구조를 나타낸 사진이다. 32x32 이미지가 들어가는데 Convolution Layer, Pooling Layer의 조합으로 2개의 층을 지나고 2개의 Fully Connected Layer를 통해 출력이 결정되는 것을 확인할 수 있다.

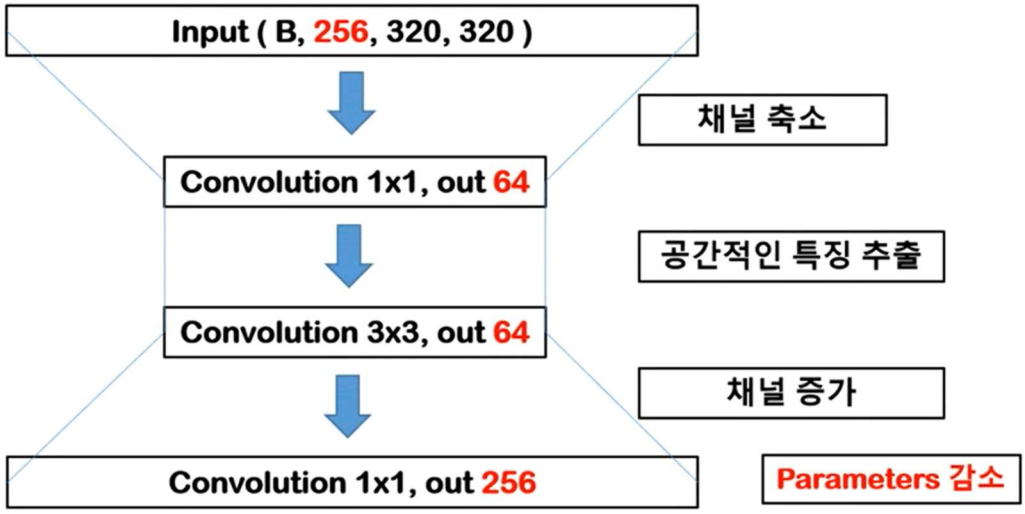
CNN을 사용하는 것의 이점으로는 크게 5가지 정도가 있다. 첫 번째로 공간적인 구조 학습이 가능하다는 것이다. Convolution Layer는 입력 데이터의 지역적인 패턴 및 구조를 학습하는데 효과적이기 때문에 특히나 이미지 분류 문제에서 많이 사용되는 것이다. 이미지에서의 객체나 특징들은 주변 픽셀과의 상관성이 높기 때문에 Convolution Layer에서 filter, kernel을 통해 이러한 구조를 보존하면서 학습할 수 있는 것은 이미지 분류 문제에서 큰 이점으로 작용한다. 두 번째로 Convolution Layer는 동일한

가중치를 여러 위치에서 공유하기 때문에 학습해야 할 파라미터의 수가 크게 감소한다. Fully Connected Layer에서의 파라미터 수는 Input x output으로 계산되는데 Convolution layer의 파라미터는 in_channel x out_channel x kernel_size이기 때문에 파라미터의 수가 크게 감소하는 것을 확인할 수 있다. 세 번째로 Convolution layer는 입력 데이터에 대한 변환 불변성(translation invariance)를 제공한다. 객체나 패턴의 위치가 조금씩 바뀌어도 모델이 이를 인식할 수 있다. 네 번째로 CNN은 계층적 특성 학습이 가능하다. Convolution Layer를 여러 층으로 쌓으면서 모델은 입력의 low-level 특성에서 high-level의 추상적인 특성까지 계층적으로 학습할 수 있다. 마지막으로 이러한 맥락에서 CNN은 일반적으로 사용하던 Fully Connected 연산보다 효율적으로 학습할 수 있다.

2) BottleNeck Layer

이번 과제 해결의 핵심 아이디어로도 선택한 BottleNeck은 딥러닝 모델에서 Networ의 깊이를 증가시킬 때 발생하는 연산 복잡도와 계산 비용을 줄이기 위한 목적에서 도입된 개념이다. 수업 시간에서도 다루었던 ResNet(Residual Network)에서 사용된 개념이기도 하다.

BottleNeck Layer는 세 가지 연속된 합성곱 층(Convolution Layer)로 구성되는데 첫 번째는 1 x 1 Convolution layer로 차원을 축소하는 역할을 한다. 두 번째는 1보다 큰 kernel을 사용하여 기존의 Convolution layer와 동일한 역할로 입력 데이터의 특징을 학습하고 확장하는 역할을 한다. 세 번째는 다시 1 x 1 Convolution Layer로 축소한 차원을 증가시키는데 사용된다. 이러한 구조는 아래의 사진에 잘 드러나 있다.



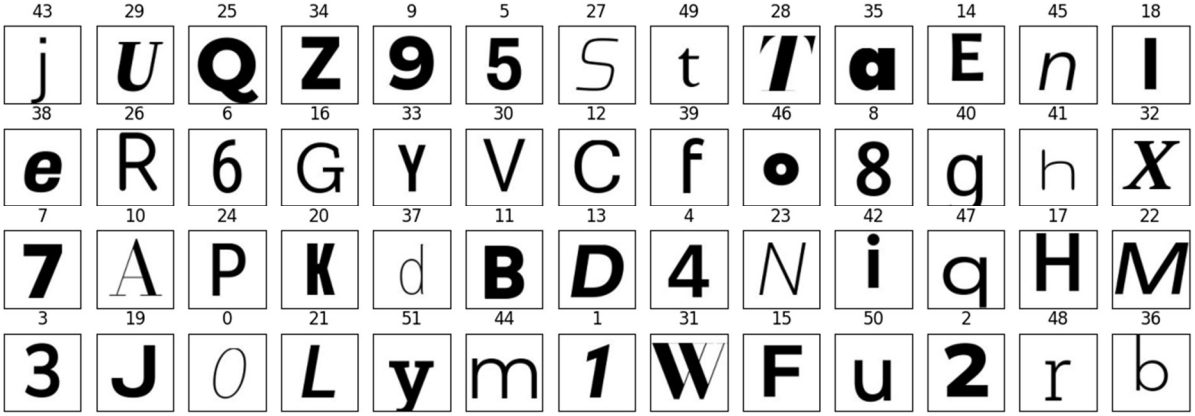
Bottleneck Layer를 사용하는 이점은 바로 연산량을 감소시킬 수 있다는 것이다. Convolution Layer의 파라미터를 구하는 공식은 kernel x input channel x output channel이다. 하지만 Bottleneck layer에서는 kernel의 크기가 1x1이기 때문에 매우 적은 연산량을 가지게 되고 그렇게 1x1 convolution layer로 축소된 차원에서 지나는 Convolution layer에서는 input channel이 줄어든 것이기 때문에 기존보다 파라미터의 개수가 줄어들고, 그에 따라 연산량도 감소하게 되는 것이다.

4. 과제 수행 방법

1) 데이터 구조 확인

52개 클래스의 전반적인 이미지에 대해 살펴보기 위해, valid_data에서 각각의 클래스에 해당하는 이미지를 1개씩 추출하는 다음의 코드를 사용하여 아래와 같이 52개 클래스에 대한 이미지를 개괄적으로 살펴볼 수 있다.

```
fig = plt.figure(figsize=(15, 5))

j = 0
for i in range(52):
    plt.subplot(4, 13, i + 1)
    plt.imshow(valid_data[j][0].squeeze(), cmap='gray')
    plt.title(valid_data[j][1].item())
    plt.xticks([])
    plt.yticks([])
    j += 300
```



앞서 과제 해결 아이디어의 (4)에서도 언급했듯, 각각의 클래스에 해당하는 train 데이터가 800개이기 때문에 학습의 정확도를 높이기 위해서는 데이터를 추가적으로 확보하는 것이 크게 도움이 될 것으로 예상된다. 수업에서 학습했던 'Image Augmentation'이 이에 해당하며 Object Detection의 성능을 향상시키기 위한 기법들에 대한 학습에서 Image Augmentation, Data Augmentation은 기본적으로 사용해야 하는 기법으로 여겨진다. 이는 수업 자료에 나온 다음의 표에서도 확인할 수 있다.

Incremental Tricks	mAP	Δ	Cumu Δ
- data augmentation	64.26	-15.99	-15.99
baseline	80.25	0	0
+ synchronize BN	80.81	+0.56	+0.56
+ random training shapes	81.23	+0.42	+0.98
+ cosine lr schedule	81.69	+0.46	+1.44
+ class label smoothing	82.14	+0.45	+1.89
+ mixup	**83.68**	**+1.54**	**+3.43**

하지만 뒤에서 진행하는 CNN의 모델 구조를 설계한 것을 바탕으로 이미 높은 수준의 정확도를 확인했기 때문에 이번 과제에서는 Image Augmentation을 시도하지는 않았다.

2) BottleNeck 구조를 바탕으로 모델 설계

과제 해결 아이디어의 (1), (2), (3)에서 언급했듯 이미지 분류이기 때문에 CNN 모델을 기본적으로 사용하기로 결정하였으며 그 중에서도 파라미터 수를 줄이고 학습 시간을 단축하기 위해 BottleNeck 구조를 도입하여 연산량을 줄이고자 했다. 다음은 설계한 모델의 각각의 layer에 대한 세부 코드이다.

```
self.layer1 = nn.Sequential(
    nn.Conv2d(1, 32, 5, 1, 2), # 1x100x100 --> 32x100x100
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2) # 32x100x100 --> 32x50x50
)
```

Layer1의 경우 1 x 100 x 100의 이미지를 입력 받아서 5 x 5 kernel에 stride=1, padding=2를 사용하여 이미지의 크기를 유지하면서 출력 채널을 32로 증가시켰다. 그리고 2x2 Max Pooling을 사용해서 100 x 100의 이미지를 50 x 50으로 줄인다. 첫 번째 layer에서 bottleneck을 사용하지 않은 이유는 bottleneck 구조가 1 x 1 convolution layer를 사용하여 차원을 축소한 후 계산하는 것인데 입력 채널이 1이기 때문에 불필요하다고 판단했기 때문이다. 하지만 layer 1의 출력 채널이 32가 되었으니 이후 layer에서는 bottleneck 구조를 추가하도록 한다.

```
self.layer2 = nn.Sequential( # 32x50x50 --> 64x50x50
    nn.Conv2d(32, 16, 1, 1, 0), # Bottleneck layer(차원 축소)
    nn.BatchNorm2d(16),
    nn.ReLU(),

    nn.Conv2d(16, 16, 5, 1, 2),
    nn.BatchNorm2d(16),
    nn.ReLU(),
    nn.Conv2d(16, 64, 1, 1, 0),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2) # 64x25x25
)
```

Layer2에서는 32 x 50 x 50이 입력으로 들어오고 출력은 64 x 25 x 25이다. 우선 입력에 대해 1 x 1 convolution layer를 사용하여 16 채널로 차원을 축소한 다음 5 x 5 kernel에 stride=1, padding=2를 사용하여 이미지의 크기를 유지하면서 5 x 5 kernel로 공간적인 특징을 추출하도록 한다. 이후 1 x 1 convolution layer를 다시 사용하여 16개의 채널을 64개 출력 채널로 차원을 확장한다.

```
self.layer3 = nn.Sequential(
    nn.Conv2d(64, 32, 1, 1, 0),
    nn.BatchNorm2d(32),
    nn.ReLU(),

    nn.Conv2d(32, 32, 5, 1, 2),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.Conv2d(32, 128, 1, 1, 0),
    nn.BatchNorm2d(128),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=5) # 128x5x5
)
self.fc1 = nn.Linear(128*5*5, 100)
self.fc2 = nn.Linear(100, num_classes)

self.dropout = nn.Dropout(0.4)
```

Layer3도 Layer2에서 사용한 bottleneck 구조가 사용되었으며 64의 입력 채널을 32로 차원 축소했다가 5x5 kernel로 공간적인 특징을 추출하고 다시 32채널을 128의 출력 채널로 1 x 1 convolution layer를 사용해서 차원 확대를 진행한다. 이때 마지막으로 사용되는 max pooling의 경우 25의 배수인 5를 사용해서 5x5 kernel로 Max pooling을 진행하였고 그 결과 128 채널의 5 x 5 이미지 크기를 가진 출력을 보내게 된다. 이를 두 개의 Fully Connected Layer에 연결하여 최종적으로 num_classes에 해당하는 52개의 클래스로 분류하도록 한다. 추가적으로 과제 핵심 아이디어 (4)에서 언급했듯, 데이터의 개수가 부족하여 Overfitting에 대한 우려가 있기 때문에 Dropout의 percent를 0.4로 비교적 높게 설정하여 Overfitting을 방지하고자 했다. 이렇게 설계된 모델의 구조는 아래 사진에서 자세하게 확인해볼 수 있다.

```
ConvNet(
  (layer1): Sequential(
    (0): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer2): Sequential(
    (0): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1))
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(16, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1))
    (7): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (layer3): Sequential(
    (0): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(32, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Conv2d(32, 128, kernel_size=(1, 1), stride=(1, 1))
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): MaxPool2d(kernel_size=5, stride=5, padding=0, dilation=1, ceil_mode=False)
  )
  (fc1): Linear(in_features=3200, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=52, bias=True)
  (dropout): Dropout(p=0.4, inplace=False)
)
```

3) HyperParameter & Criterion & Optimizer

```
# Hyper-parameters
batch_size = 50
learning_rate = 0.0001
num_epochs = 12
```

```
model = ConvNet().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

모델의 학습 이전에 설정한 하이퍼 파라미터로는 batch_size와 learning_rate, num_epochs이 있다. Batch_size는 기존에 설정된 50을 유지했고 learning_rate도 무난한 0.0001을 사용하였으며 epoch은 학습 시간이 넘어가지 않는 한에서 최대한인 12로 잡았다.

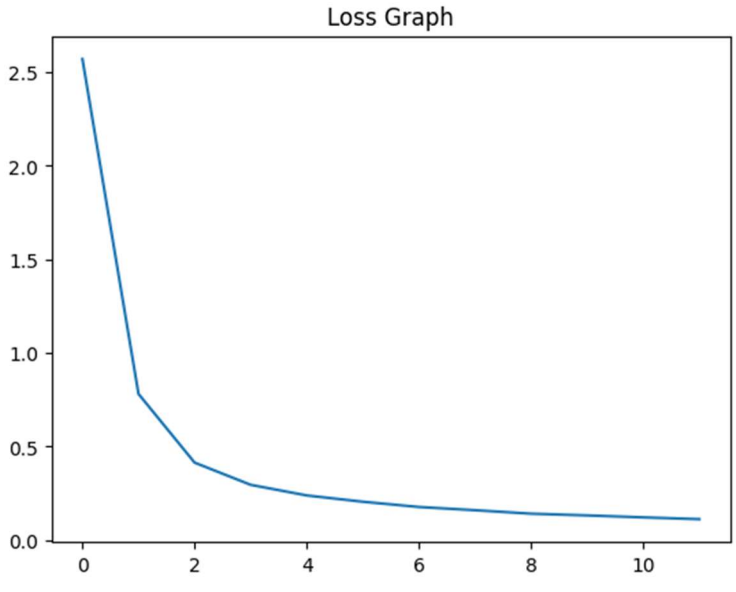
추가적으로 Loss Function으로는 다중 클래스 분류이기 때문에 CrossEntropyLoss()를 사용했으며 Optimizer로는 가장 보편적으로 성능이 우수하다고 알려졌으며, 그렇기에 기존의 과제에서도 사용했던 Adam Optimizer로 설정하였다.

4) Train & Loss Graph

```
total_step = len(train_loader)
total_loss = []
model.train() # train 모드로 지정

start = time.time() # Train 시작 시간 정보 저장
for epoch in range(num_epochs): # 에폭만큼 반복, 각 에폭에 대해서 --> epoch별 반복문
  epoch_loss = []
  for i, (img, label) in enumerate(train_loader): # batch를 하나씩 불러오면서 --> batch별 반복문
    # Assign Tensors to Configures Devices (gpu)
    img = img.to(device)
    label = label.to(device)

    # Forward propagation
    outputs = model(img)

    # Get Loss, Compute Gradient, Update Parameters
    loss = criterion(outputs, label) # loss 계산
    optimizer.zero_grad() # gradient 초기화
    loss.backward() # backpropagation 계산
    optimizer.step() # gradient 업데이트

    epoch_loss.append(loss.detach().cpu().numpy())
    # Print Loss
    if (i+1)==len(train_loader):
      print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, i+1, len(train_loader), loss.item()))
  total_loss.append(np.mean(epoch_loss))
  print(f"epoch{epoch+1} loss: {np.mean(epoch_loss)}") # loss가 점점 줄어드는 것을 확인할 수 있다

end = time.time() # Train 종료 시간 정보 저장
duration = end - start # 종료 시간 - 시작 시간
print("Training takes {:.2f}minutes".format(duration/60)) #초 단위로 저장되므로, 60으로 나누어 분으로 표시
torch.save(model.state_dict(), 'model.pth')
```

학습 코드 자체는 기존의 과제에서 사용하던 것과 다른 것이 거의 없지만 이번 과제에서 학습 시간을 출력하기 위해 train이 시작되는 지점에서 start = time.time(), 끝나는 지점에서 end = time.time()으로 시간을 기록하여 초 단위를 분 단위로 바꾸고 학습에 소요된 시간을 출력하는 부분이 추가된 것을 확인할 수 있다.

```
Epoch [1/12], Step [832/832], Loss: 1.1405
epoch1 loss: 2.5707807540893555
Epoch [2/12], Step [832/832], Loss: 0.4257
epoch2 loss: 0.7808482050895691
Epoch [3/12], Step [832/832], Loss: 0.3893
epoch3 loss: 0.4136706590652466
Epoch [4/12], Step [832/832], Loss: 0.1178
epoch4 loss: 0.2949284315109253
Epoch [5/12], Step [832/832], Loss: 0.1932
epoch5 loss: 0.23810890316963196
Epoch [6/12], Step [832/832], Loss: 0.1552
epoch6 loss: 0.20451395213603973
Epoch [7/12], Step [832/832], Loss: 0.0283
epoch7 loss: 0.17641419172286987
Epoch [8/12], Step [832/832], Loss: 0.0873
epoch8 loss: 0.1592639535665512
Epoch [9/12], Step [832/832], Loss: 0.1352
epoch9 loss: 0.1406630277633667
Epoch [10/12], Step [832/832], Loss: 0.0928
epoch10 loss: 0.13137978315353394
Epoch [11/12], Step [832/832], Loss: 0.0799
epoch11 loss: 0.1208920106291771
Epoch [12/12], Step [832/832], Loss: 0.1653
epoch12 loss: 0.11129775643348694
Training takes 14.46minutes
```



Train을 진행함에 따라 발생한 loss는 좌측 사진을 통해 확인할 수 있으며 total_loss에 저장된, epoch에 따른 Loss의 변화 추이는 우측의 그래프를 통해 확인할 수 있다.

추가적으로 Train 시 할당받은 GPU의 종류를 확인하기 위해 "!nvidia-smi" 명령어를 사용하였으며, 그 결과는 다음과 같다.

```
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   38C    P8              9W /  70W |      3MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

5. 결과 및 토의

1) Test & Accuracy

우선 학습이 완료된 모델을 활용하여 아래의 Test code를 이용해 분류의 Accuracy를 계산한 결과는 98.5%임을 확인할 수 있다.

```
model.eval() # Set model as evaluation mode

with torch.no_grad(): # auto_grad off
  correct = 0
  total = 0
  for images, labels in valid_loader:
    images = images.to(device)
    labels = labels.to(device)
    outputs = model(images)
    _, predicted = torch.max(outputs.data, 1)

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

  print('Accuracy of the last_model network on the {} valid images: {} %'.format(len(valid_data), 100 * correct / len(valid_data)))
```

```
Accuracy of the last_model network on the 15600 valid images: 98.5448717948718 %
```

2) BottleNeck 사용의 이점 – 파라미터 수 감소

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 100, 100]             832
       BatchNorm2d-2         [-1, 32, 100, 100]              64
              ReLU-3         [-1, 32, 100, 100]               0
         MaxPool2d-4           [-1, 32, 50, 50]               0
            Conv2d-5           [-1, 16, 50, 50]             528
       BatchNorm2d-6           [-1, 16, 50, 50]              32
              ReLU-7           [-1, 16, 50, 50]               0
            Conv2d-8           [-1, 16, 50, 50]           6,416
       BatchNorm2d-9           [-1, 16, 50, 50]              32
             ReLU-10           [-1, 16, 50, 50]               0
           Conv2d-11           [-1, 64, 50, 50]           1,088
      BatchNorm2d-12           [-1, 64, 50, 50]             128
             ReLU-13           [-1, 64, 50, 50]               0
        MaxPool2d-14           [-1, 64, 25, 25]               0
           Conv2d-15           [-1, 32, 25, 25]           2,080
      BatchNorm2d-16           [-1, 32, 25, 25]              64
             ReLU-17           [-1, 32, 25, 25]               0
           Conv2d-18           [-1, 32, 25, 25]          25,632
      BatchNorm2d-19           [-1, 32, 25, 25]              64
             ReLU-20           [-1, 32, 25, 25]               0
           Conv2d-21          [-1, 128, 25, 25]           4,224
      BatchNorm2d-22          [-1, 128, 25, 25]             256
             ReLU-23          [-1, 128, 25, 25]               0
        MaxPool2d-24            [-1, 128, 5, 5]               0
          Dropout-25                 [-1, 3200]               0
           Linear-26                  [-1, 100]         320,100
          Dropout-27                  [-1, 100]               0
           Linear-28                   [-1, 52]           5,252
================================================================
Total params: 366,792
Trainable params: 366,792
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.04
Forward/backward pass size (MB): 16.53
Params size (MB): 1.40
Estimated Total Size (MB): 17.97
----------------------------------------------------------------
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 100, 100]             832
       BatchNorm2d-2         [-1, 32, 100, 100]              64
              ReLU-3         [-1, 32, 100, 100]               0
         MaxPool2d-4           [-1, 32, 50, 50]               0
            Conv2d-5           [-1, 64, 50, 50]          51,264
       BatchNorm2d-6           [-1, 64, 50, 50]             128
              ReLU-7           [-1, 64, 50, 50]               0
         MaxPool2d-8           [-1, 64, 25, 25]               0
            Conv2d-9          [-1, 128, 25, 25]         204,928
      BatchNorm2d-10          [-1, 128, 25, 25]             256
             ReLU-11          [-1, 128, 25, 25]               0
        MaxPool2d-12            [-1, 128, 5, 5]               0
          Dropout-13                 [-1, 3200]               0
           Linear-14                  [-1, 100]         320,100
          Dropout-15                  [-1, 100]               0
           Linear-16                   [-1, 52]           5,252
================================================================
Total params: 582,824
Trainable params: 582,824
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.04
Forward/backward pass size (MB): 13.78
Params size (MB): 2.22
Estimated Total Size (MB): 16.05
----------------------------------------------------------------
```

위의 두 사진은 BottleNeck을 사용했는지 여부에 따른 모델의 구조와 파라미터의 개수를 확인할 수 있는, 모델의 summary이다. 좌측의 사진이 BottleNeck을 사용하여 최종적으로 선택한 모델의 구조이며 해당 모델에서 사용한 총 파라미터 수는 366,792임을 확인할 수 있다. 우측의 사진은 동일한 입력과 동일한 출력을 내보내는 구조인데 단지 BottleNeck 구조를 사용하지 않은 모델의 구조이다. 해당 모델의 경우 사용한 파라미터의 개수는 582,824로 분명 BottleNeck 모델의 파라미터 수보다는 많은 것을 확인할 수 있다. 즉, BottleNeck 구조를 사용함에 따라 약 40% 정도 파라미터가 감소한 것이다. 물론 이에 따라 학습의 시간에서도 약간의 단축은 존재했다.

```
Epoch [1/12], Step [832/832], Loss: 1.1405
epoch1 loss: 2.5707807540893555
Epoch [2/12], Step [832/832], Loss: 0.4257
epoch2 loss: 0.7808482050895691
Epoch [3/12], Step [832/832], Loss: 0.3893
epoch3 loss: 0.4136706590652466
Epoch [4/12], Step [832/832], Loss: 0.1178
epoch4 loss: 0.2949284315109253
Epoch [5/12], Step [832/832], Loss: 0.1932
epoch5 loss: 0.23810890316963196
Epoch [6/12], Step [832/832], Loss: 0.1552
epoch6 loss: 0.20451395213603973
Epoch [7/12], Step [832/832], Loss: 0.0283
epoch7 loss: 0.17641419172286987
Epoch [8/12], Step [832/832], Loss: 0.0873
epoch8 loss: 0.1592639535665512
Epoch [9/12], Step [832/832], Loss: 0.1352
epoch9 loss: 0.1406630277633667
Epoch [10/12], Step [832/832], Loss: 0.0928
epoch10 loss: 0.13137978315353394
Epoch [11/12], Step [832/832], Loss: 0.0799
epoch11 loss: 0.1208920106291771
Epoch [12/12], Step [832/832], Loss: 0.1653
epoch12 loss: 0.11129775643348694
Training takes 14.46minutes
```

```
Epoch [1/12], Step [832/832], Loss: 1.1550
epoch1 loss: 2.4951014518737793
Epoch [2/12], Step [832/832], Loss: 0.7144
epoch2 loss: 0.7303168773651123
Epoch [3/12], Step [832/832], Loss: 0.2738
epoch3 loss: 0.44483739137649536
Epoch [4/12], Step [832/832], Loss: 0.3176
epoch4 loss: 0.3027680218219757
Epoch [5/12], Step [832/832], Loss: 0.2941
epoch5 loss: 0.24603864550590515
Epoch [6/12], Step [832/832], Loss: 0.4252
epoch6 loss: 0.2123633176088333
Epoch [7/12], Step [832/832], Loss: 0.1884
epoch7 loss: 0.1851438730955124
Epoch [8/12], Step [832/832], Loss: 0.0851
epoch8 loss: 0.1640375256538391
Epoch [9/12], Step [832/832], Loss: 0.0474
epoch9 loss: 0.14666908979415894
Epoch [10/12], Step [832/832], Loss: 0.1197
epoch10 loss: 0.13283337652683258
Epoch [11/12], Step [832/832], Loss: 0.1704
epoch11 loss: 0.1268453598022461
Epoch [12/12], Step [832/832], Loss: 0.0970
epoch12 loss: 0.11339379101991653
Training takes 13.09minutes
```

좌측의 사진은 앞서 살펴본, BottleNeck 구조가 사용된 모델의 Loss이고 소요 시간은 14.46분이다. 우측의 사진은 동일한 구조인데 BottleNeck을 사용하지 않은 모델의 Loss이고 소요 시간은 13.09분이다. 즉 1.5분 정도가 단축된 것을 확인할 수 있는데, 이는 큰 차이라고 판단되지는 않았다.

사실 이 부분이 처음 BottleNeck 구조를 사용하려고 했던 이유와 예상된 결과와 달라 당황스러운 부분이었다. BottleNeck 구조를 사용하여 얻을 수 있는, 예상되는 이점으로는 차원을 축소한 후 공간적인 특징을 추출하기 때문에 파라미터 수가 감소하고 이 덕분에 연산량이 줄어 학습 시간도 줄어들 것을 기대한 것이다. 하지만 실질적으로 유의미한 수준으로 학습 시간이 줄어들지는 않았다. 아마 이는 파라미터의 수의 감소는 분명 있지만 그 차이가 전체 데이터를 학습하는 과정에서 차지하는 비율이 유의미한 수준이 아니기 때문에 일어난 현상이라 추측된다.

3) BottleNeck 사용의 이점 – 일반화 성능 향상

하지만 분명 BottleNeck을 사용한 것에는 test 성능인 Accuracy 측면에서 큰 이점이 있었다.

```
Accuracy of the last_model network on the 15600 valid images: 98.5448717948718 %
```

```
Accuracy of the last_model network on the 15600 valid images: 92.98076923076923 %
```

위의 98.5%의 성능은 BottleNeck을 사용한 모델의 Test 성능이고 아래의 92.9%의 성능은 BottleNeck을 사용하지 않은 모델의 Test 성능이다. 즉, 5.5% 정도의 정확도 차이가 났다는 것을 확인할 수 있는데 왜 이러한 성능의 향상이 있는 것인가?

이에 대해 고민한 결과, BottleNeck이 Overfitting을 예방하는 맥락에서 도움이 되었다고 추론했다. 앞서 과제 해결 아이디어 (4)에서 언급했던, 데이터의 수가 부족함에 따른 Overfitting 우려에 대해 BottleNeck 구조가 도움이 되었다고 보는 것이다. 그 이유는 DropOut의 percent가 0.4인지, 0.3인지에 따라 발생하는 성능의 차이를 살펴봄으로써 이해해볼 수 있다.

```
(dropout): Dropout(p=0.3, inplace=False)
```

```
Accuracy of the last_model network on the 15600 valid images: 96.42948717948718 %
```

위의 사진은 BottleNeck을 사용하는 동일한 모델에 대해 Dropout의 percent만 0.4에서 0.3으로 낮추었을 때의 성능을 측정한 것이다. 기존의 0.4일 때의 모델은 여러 번 학습하여도 98% 아래로 떨어지지 않는 반면 Dropout의 비율이 0.3으로 감소함에 따라 성능이 감소한 것을 볼 수 있다. 이는 분명 Dropout의 비율을 높이는 것이 train데이터에 대한 overfitting을 완화하고 그에 따라 일반화 성능이 높게 나타나는 것이다. 이처럼 기본적으로 데이터의 수가 부족하며 overfitting의 우려가 있는 만큼 overfitting을 방지할 수 있는 조치가 성능을 향상시킨다고 이해할 수 있다.

이러한 맥락에서 BottleNeck을 사용한 구조와 그렇지 않은 구조를 비교한다면, BottleNeck을 사용했을 때 차원을 축소하고 그 축소된 차원에 대해 kernel을 통해 feature map을 생성하기 때문에 일반화의 측면에서 이점이 있다고 볼 수 있는 것이다. 반대로 생각하면 정보의 손실(loss)가 있다고 볼 수 있지만 현재 데이터의 수가 부족하여 Train 데이터에 대한 overfitting이 우려되는 맥락에서는 이러한 구도가 오히려 overfitting을 예방하고 일반화 성능을 높이는, 긍정적인 기제로 작용한 것이다.

5. 참고 문헌

https://velog.io/@lighthouse97/CNN%EC%9D%98-Bottleneck%EC%97%90-%EB%8C%80%ED%95%9C-%EC%9D%B4%ED%95%B4

Introduction to deep learning_16.pdf (인공지능(딥러닝)개론 수업 자료, Data Augmentation 부분)

Introduction to deep learning_11.pdf (인공지능(딥러닝)개론 수업 자료, CNN 부분)