인공지능(딥러닝)개론

Term Project

조주현

20171483

박형민교수님

2021 년 12 월 14 일 화요일

1. 과제 목표

English Character & Number with various fonts 분류

숫자 0~9 와 대소문자가 포함된 영문자. 단, 대소문자 구분이 어려운 "c", "k", "l", "o", "p", "s", "v", "w", "x", “z"는 제외.

- Data set 정보

test set : 37,232
validation set : 7,800

totally balanced set 으로 각 class 별 data 의 개수는 모두 동일

data shape : 90 x 90 x 1

- 제한사항

Google colab gpu 기준 10 분 이내

Pre-trained model 사용 불가

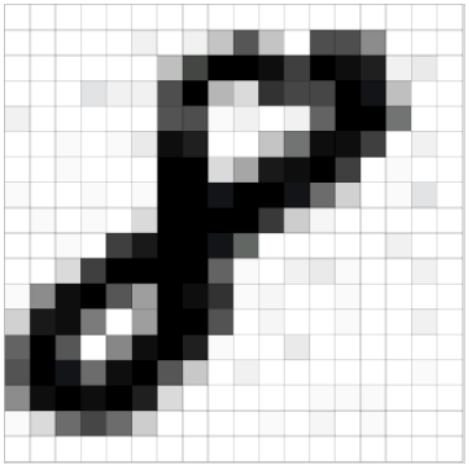
Python 3.5+, Pytorch 1.2+ 이상

2. 배경이론

CNN 은 Convolution Neural Networks 의 약자로 딥러닝에서 영상이나 이미지 데이터를 처리할 때 사용하며, 이름에서 볼 수 있듯이 convolution 이라는 전처리 작업이 들어가있다. Deep Neural Networks(DNN)에서는 데이터를 1 차원으로 줄여 사용을 하는데, 이미지나 영상자료의 경우 1 차원으로 줄이게 되면 공간적, 지역적 정보가 손실된다. 또한 추상화 과정이 없어 연산과정에서 학습 시간과 능률이 저하된다. 따라서 공간, 지역적 정보의 손실 없이 데이터를 처리하기 위해 CNN 이라는 개념이 나오게 되었다.

CNN 에선 2 차원 데이터를 그대로 받아 특성(feature)들을 계층화시켜 모델을 구성한다. CNN 의 중요한 포인트는 이미지 전체를 보는 것이 아니라 부분을 보는 것이다. 이미지의 한 픽셀과 그 근처 픽셀간의 연관성을 살리는 것이다. 예를 들어 사람의 얼굴을 detecting 하는 모델을 CNN 을 이용해 만든다고 하면 사람 얼굴의 특징인 눈, 코, 입, 귀 등의 feature 를 파악하고, test 시에는 해당 feature 가 있는지 없는지를 확인하고 판단하게 된다. 때문에 이미지 전체를 볼 필요 없이 이미지의 작은 부분을 보면서 feature 를 체크하는 것이 중요하다.

이렇게 feature 를 추출할 때 사용하는 전처리 방법이 바로 Convolution 이다.



< figure 1. MNIST data(class ‘8’) >

MNIST 데이터 중 하나이다. 크기는 28x28x1 이고, class 는 ‘8’이다.

Figure 1 은 유명한 손글씨 데이터 중 하나는 MNIST 데이터이다. CNN 을 수행할 때 해당 이미지의 픽셀값을 2 차원의 행렬값으로 받게 된다. 예시 이미지의 경우 크기가 28x28 임으로 입력 받을 행렬의 크기 또한 28x28 가 된다.

입력 이미지에서 적절한 feature 를 뽑아낼 때 kernel(filter)을 이용한다. 입력값 이미지의 모든 영역에 동일한 filter 를 적용해 패턴을 찾아 처리한다. 이 때 적용한다는 것은 행렬과 행렬간 inner product 를 의미한다. Inner product 는 두개의 행렬의 동일한 위치에 있는 숫자를 곱해 모두 더해주는 것을 말한다. 예를 들어 5x5 로 구성된 칸들에 3x3 크기의 kernel 를 적용한다고 하면 너비와 높이에 모두 3 번씩 놓을 수 있다. 이를 수학적으로 표현하면 아래와 같다.

입력값 $:d_1 \times d_2$

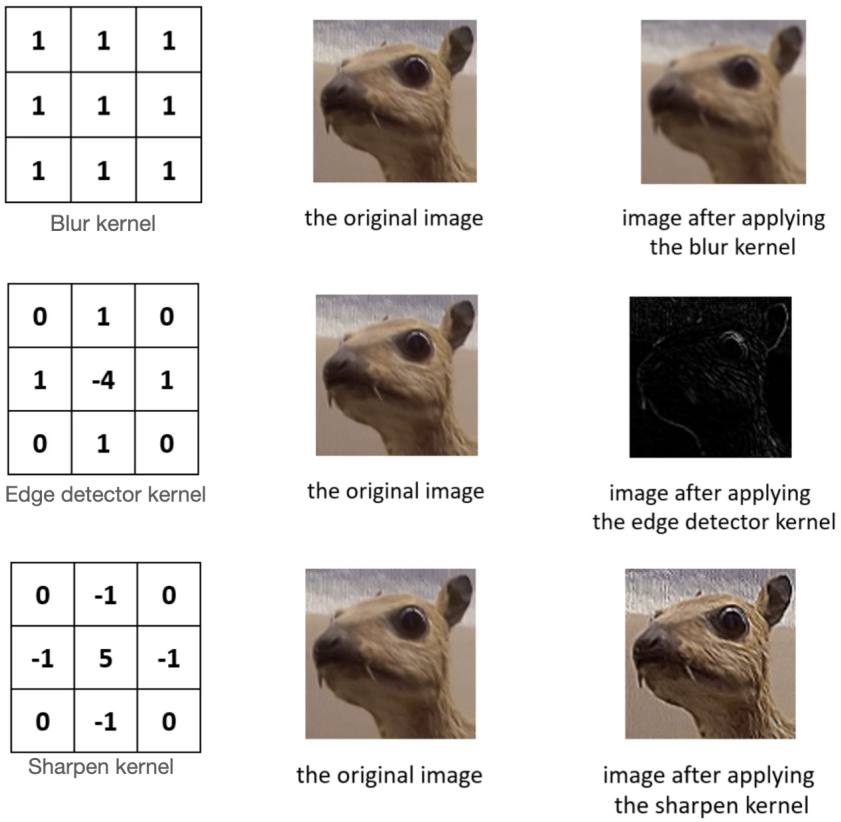
필터 : $k_1 \times k_2$

stride : s

결과값 : $[\{(d_1 - k_1)/s\} + 1] \times [\{(d_2 - k_2)/s\} + 1]$

사용할 kernel 는 사용자가 정하는 것이 아니라 CNN 모델에서 알아서 정해준다. 하지만 kernel 들을 미리 정해놓고 학습을 하는 것이 아니라 여러 kernel 들을 사용해보고, 가장 좋은 결과를 내는 kernel 를 사용한다. Kernel 를 적용하고 난 뒤 결과값들을 모아둔 것을 feature map 이라고 하고, 그 과정을 feature mapping 이라고도 한다.

대표적은 filter 를 살펴보면 아래와 같다.



< figure 2. Common filters >

대표적인 3 개의 filter 이다. 위에서부터 blur, edge detector, sharpen filer 로 입력 이미지를 각 이름에 맞게 변환한 것을 볼 수 있다.

kernel 를 적용할 때 stride 라는 개념이 쓰인다. 이는 쉽게 말해 kernel 를 얼마만큼 움직여 주는가에 대한 것이다. Default 값은 1 이며, stride 는 사용자가 임의로 지정해 줄 수 있다. Stride 를 키운다는 것은 필터가 이미지를 건너뛰는 칸이 커짐을 의미함으로 결과값 이미지의 크기는 작아진다. Stride 값을 적용한 수학적 표현은 아래와 같다.

입력값 : $d_1 \times d_2$

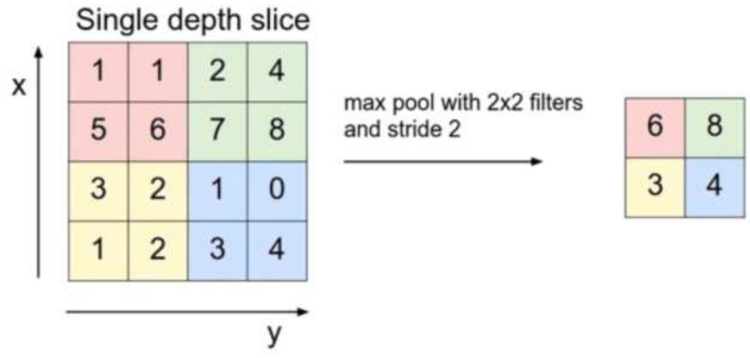
필터: $k_1 \times k_2$

$stride = s$

결과값 : $[\{(d_1 - k_1)/s\} + 1] \times [\{(d_2 - k_2)/s\} + 1]$

앞서 살펴본 convolution 처리를 보면 5x5 이미지에서 3x3 kernel 를 적용했을 때 결과값의 크기가 3x3 으로 줄어든 것을 확인할 수 있다. 해당 과정에서 stride 는 1 인데, 만약 stride 를 키운다고 하면 결과값의 크기는 더욱 줄어들 것이다. 즉, 태두리 부분의 data 의 손실이 발생한다는 것이다. 이를 해결하기 위해 padding 이라는 방법을 사용한다. 0 으로 이미지의 가장자리 부분을 채워주는 방법이다. 그렇다면 5x5 크기의 입력값에 padding = 1 을 적용한다면 입력 이미지의 크기는 7x7 이 될 것이고, 3x3 kernel 를 적용해도 5x5 의 결과값을 얻을 수 있다. 즉 입력값과 결과값의 크기가 동일해 손실이 없음을 확인할 수 있다.

많은 kernel 를 사용해 입력 이미지에 대한 feature map 을 생성했을 때 생기는 문제점은 한개의 이미지에서 kernel 수 만큼의 추가적인 이미지가 생겼다는 것이다. 이렇게 될 경우 계산해야 하는 이미지의 수가 너무 많아져 연산량을 감당할 수 없게 된다. 때문에 고안된 방법이 pooling 이라는 과정이다. Pooling 은 각 feature map 의 차원을 축소해 주는 것을 목적으로 한다. 종류로는 max pooling, average pooling, min pooling 이 있고, 각각 가장 큰 값, 평균값, 가장 작은 값을 추출하여 사용한다. Feature map 을 n x n 크기의 window 로 훑으면서 그 안에서 특정 값을 추출한다. 이 때 역시 stride 라는 개념이 사용되며 앞서 말했듯이 차원을 축소하는 것을 목적으로 하기에 padding 은 잘 사용하지 않는다. CNN 에서는 주로 max pooling 과 average pooling 을 주로 사용한다. 아래는 max pooling 의 예시이다.



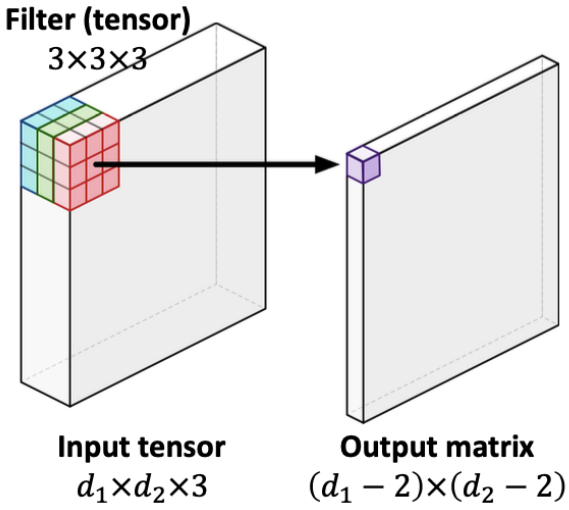
$$OutputRowSize = \frac{InputRowSize}{PoolingSize}$$

$$OutputColumnSize = \frac{InputColumnSize}{PoolingSize}$$

< figure 3. Max pooling example and its equation >

Max pooling 의 예시이다. Window 의 크기는 2x2, stride 는 2 로 4x4 크기의 입력값이 2x2 로 줄어든 것을 확인할 수 있다. 또한 Max pooling 이기에 각 window 에서 가장 큰 값이 추출된다.

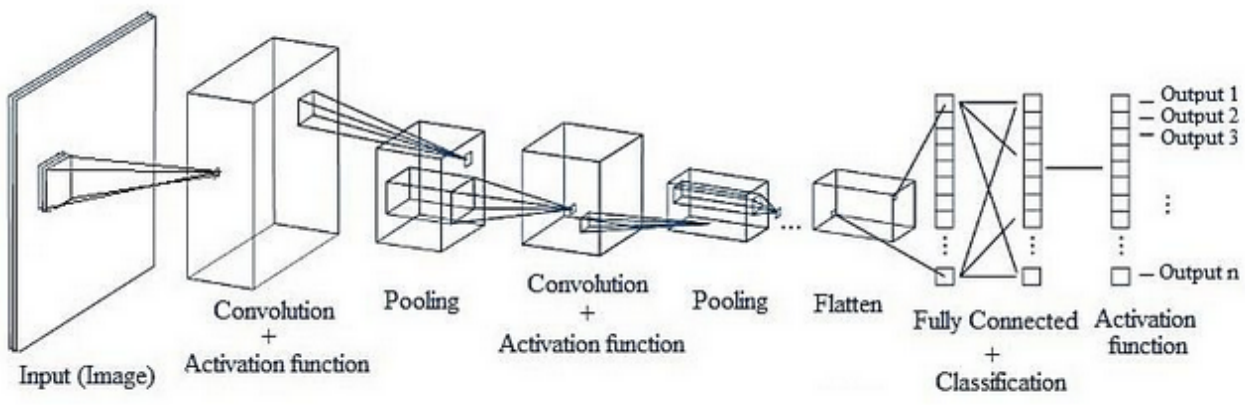
지금까지의 설명은 흑백 이미지의 입력값에 대한 설명이었다. 하지만 만약 이미지가 흑백이 아니라 컬러라면 입력값에 Red, Green, Blue 3 개의 체널이 추가된 3 차원 데이터가 될 것이다. 이러한 모양을 order-3 tensor 이라고 한다. 이러한 이미지의 경우 $k_1 \times k_2 \times 3$ 의 크기를 가진 order-3 tensor 를 kernel 로 사용한다. 그림으로 살펴보면 아래와 같다.



< figure 4. 3 차원 이미지에 3 차원 kernel 의 적용 >

3 차원 이미지 데이터에 kernel 를 적용한 모습이다. Order-3 tensor 를 적용해 2 차원의 행렬로 변환된 것을 확인할 수 있다.

CNN layer 가 끝나면 2 차원 데이터를 다시 1 차원으로 reshape 해준 뒤 fully connected layer 를 통과시켜 마지막에 softmax activation function 을 적용해 최종결과물을 출력한다. 이 때 2 차원 데이터를 1 차원으로 바꿔도 상관이 없는 이유는 pooling layer 를 거치면서 얻어낸 축소된 차원의 데이터는 이미지 자체라기보단 입력된 이미지로부터 얻어낸 특이점 데이터가 된다. 즉, 1 차원의 벡터로 변환시켜주어도 무관한 상태가 된다는 의미이다.



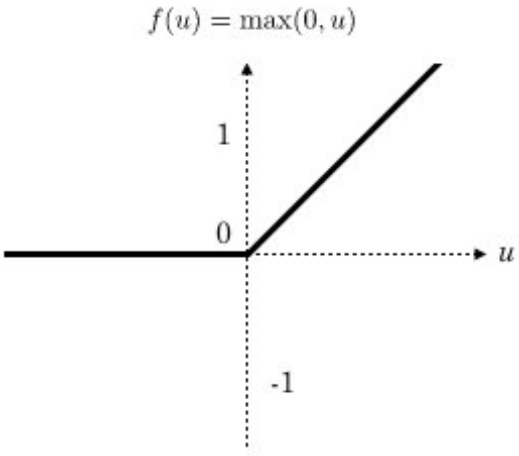
CNN 의 전체적인 네트워크 구조를 살펴보면 아래와 같다.

< figure 5. Overall CNN structure >

input 으로 부터 convolution layer 와 pooling layer 가 반복적으로 나오고, Flatten 을 통해 1 차원 데이터로 바꾸어 준 뒤 Dense layer 를 통해 최종적인 예측값을 구한다.

CNN 을 구성하는 것들 중 convolution layer, pooling layer 외에 추가적으로 activation function, optimizer, dropout, batch normalization 등이 사용된다.

먼저 activation function 은 모델에 비선형성을 추가해주기 위해 사용된다. 일반적인 네트워크의 구조는 “FC(fully connected layer) → batch normalize → activation function → FC → batch normalize → activation function” 와 같고, 주로 사용하는 activation function 은 ‘ReLU’이다.



ReLU function 의 정의와 그래프는 아래와 같다.

< figure 6. Relu activation function >

Relu funtion 의 정의와 그래프를 그렸다. u 가 0 이하일 때 f(u) = 0, 0 이상일 때는 f(u) = u 이다. ReLU function 은 양 극단의 값이 포화되지 않으며 계산이 효율적이고 빠르다는 장점이 있지만, 중심값이 0 이 아니고, 입력값이 음수일 경우 파라미터의 업데이트가 일어나지 않는다는 단점이 존재한다.

Deep learning 은 기본적으로 gradient descent 를 이용해 모델을 학습한다. Loss function 의 기울기(gradient)를 구하고 loss 값을 줄이는 방향으로 조정해가며 모델을 학습하는 것이다. 이 때 줄이는 방향으로 정해진 스텝량(learning rate, lr)을 곱해서 weight 를 업데이트한다. 하지만 모든 데이터에 대해 gradient descent 를 적용하는 것은 시간과 computational force 가 많이 투입되는 방법으로, 이를 개선한 stochastic gradient descent 방법을 고안했다. 이는 batch size 만큼 학습 데이터를 분리해 훑고 지나가는 개념으로 빠르게 전진하지만, 방향설정이 일정하지 않고, learning rate 의 최적화가 부족하다는 단점이 존재한다. 이를 해결하기 위해 momentum 이라는 변수를 두어 방향설정을 돕고, 이전 상황을 보며 learning rate 의 값을 조절해주는 Adam 이라는 optimizer 를 고안해 이를 사용한다. 현재에는 Adam 을 주로 사용하고 있으며, 이번 project 에서 가장 먼저 Adam 을 사용하고, 훈련 결과가 만족스럽지 않으면 다른 optimizer 를 적용하는 방법으로 진행했다.

모델을 훈련하면서 가장 주의해야 할 점은 학습데이터에 대한 과적합(overfitting)이다. 학습 모델에 과하게 맞추어져 있어 새로운 입력값이 들어왔을 때 이를 제대로 예측하지 못하는

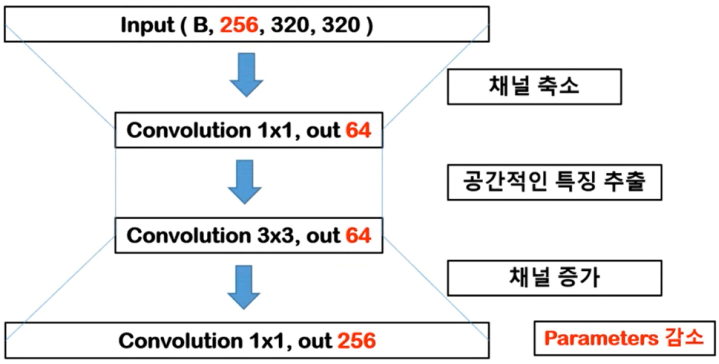
문제이다. 과적합의 해결책 중 하나가 정규화이며, 일반적으로 모델이 복잡해질수록 불이익(penalty)를 주는 방향으로 진행한다. 이 때 정규화는 training error 를 줄이는게 아닌, generalization error 를 감소시키는 것으로 정의한다.

대표적인 정규화는 weight decay, data augmentation, early stopping, ensemble methods 등이 있고, 본 과제에서 사용한 방법은 dropout 이다. Dropout 은 학습시 일부 뉴런을 '0'으로 만드는 방법이다. 일종의 비활성화이며, 이 비율은 조절이 가능하다. 확률은 일반적으로 0.2 ~ 0.5 를 사용하며, 학습할 때만 적용하고, 테스트 시에는 모든 뉴런을 사용한다. 이런 dropout 을 통해 앙상블 학습처럼 마치 여러 모델을 학습시킨 것과 같은 효과를 주어 overfitting 을 해결할 수 있다.

Training 과정 중 출력값이 과도하게 커지는 일이 발생할 수 있다. 이 경우 네트워크의 학습이 제대로 되지 않을 수 있다. 이러한 부분을 해결하기 위해 batch normalize layer 를 추가한다. Batch normalize 는 활성함수의 활성화 값 혹은 그 전의 출력값을 정규화 하는 작업으로 mini batch 데이터의 분포를 정규분포로 바꾸어 주는 작업을 수행한다. 이를 통해 gradient exploding 을 방지할 수 있으며, 학습속도 또한 빨라지는 장점이 있다.

Dropout 과 batch normalize 를 적용할 때 주의할 점은 test 혹은 validation data 에 대해 정확도 혹은 loss 값을 계산할 때는 적용하지 않아야 한다. tensorflow 의 경우 library 에서 자동으로 적용하지 않고, pytorch 의 경우 model.eval()함수를 이용해주어야 한다.

본 과제에서 convolution layer 를 구성할 때 bottleNeck 구조를 사용하였다. 대표적으로 ResNet 에서 BottleNeck 구조를 사용했다. BottleNeck 구조에선 1x1 크기의 kernel 를 사용한 convolution layer 가 있으며, channel 수를 늘릴 때 연산량을 줄이고, 비선형성을 늘릴 수 있는 특징이 있다. BottleNeck 구조를 단순히 표현하면 아래와 같이 나타낼 수 있다.



< figure 7. BottleNeck 구조 >

BottleNeck 구조를 간단하게 그린 것이다. Input shape 는 320x320 픽셀에 256 개의 kernel 이고, 1x1 convolution kernel 를 이용해 bottleNeck 을 구현했다. 최종 kernel 의 수는 512 개이다. Input channel 이 256 개에서 1x1 kernel 를 이용해 channel 수를 64 개로 줄였다. 이렇게 강제로 축소한 이유는 오로지 연산량을 줄이기 위함이다. 이 때 1x1 convolution 은 공간적인 특징이 존재하지 않는다. 공간적인 특징을 추출하기 위한 kernel 의 최소 크기는 2x2 이다. 이후 3x3

kernel 를 이용해 공간적인 특징을 추출한다. 3x3 convolution 의 연산량은 1x1 의 그것보다 9 배 많기 때문에 1x1 convolution 후에 3x3 convolution 을 이용해 특성을 추출한다. 이후 체널을 다시 증가시켜 256 개의 output channel 을 추출한다. 이렇게 하면 연산량을 줄이면서 channel 의 수를 늘릴 수 있다. 특히 마지막 layer 에서 output channel 을 256 이 아닌 512 로 하면 체널의 수를 더욱 늘릴 수 있다. 일반적으로 layer 가 깊어질수록 channel 의 수를 증가시키기 때문에 복잡한 data 를 분류하는 깊은 모델 작성시 사용할 수 있다. 하지만 강제로 channel 을 줄이고 늘리는 것은 정보 손실을 일으킴으로 정확성을 떨어뜨리는 결과를 낳게된다. 따라서 서로의 합의점을 찾는 것이 중요하다.

마지막으로 딥러닝 학습시 초기 가중치 설정은 중요한 역할을 한다. 가중치를 잘못 설정할 경우 vanishing gradient 문제나 표현력의 한계를 갖는 증 여러 문제를 일으킨다. 또한 딥러닝 학습은 비선형적인 문제를 풀기 때문에 초기값을 잘못 설정할 경우 local minimum 에 수렴할 가능성이 높아진다. 예를 들어 weight 의 평균을 0 으로 초기화시키면 모든 뉴런들이 동일한 값을 나타낼 것이고, back propagation 과정에서 weight 의 update 가 동일하게 이뤄질 것이다. 이러한 update 는 아무리 학습을 해도 동일하게 발생할 것이며, 결국 제대로 된 학습이 되지 않을 것이다. 또한 이런 동일한 update 는 여러 deep 하게 layer 를 쌓는 것을 상쇄시킬 것이다.

가중치의 초기화 방법에는 여러가지가 있다. 본 과제에서 사용한 초기화 방법은 He Initialization 이다. ReLU activation function 에 맞춰 초기화하는 방법으로 균등분포(He Uniform Initialization)와 정규분포(He Normal Initialization) 두가지로 나뉜다. 본 과제에서는 균등분포 방법을 사용했으며, 그 수식은 아래와 같다.

$$W \sim U(-\sqrt{\frac{6}{n_{in}}}, +\sqrt{\frac{6}{n_{in}}})$$

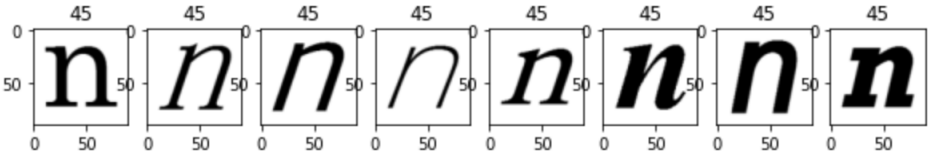
n_{in} : 이전 layer(input)의 노드 수

이 방법은 이전 노드와 다음 노드의 개수에 의존하는 방법으로, 많은 연구를 통해 최적화된 상수값을 찾아낸 결과이다.

3. 과제 수행 방법

모델의 목표는 10 개의 숫자와 42 개의 영문자를 분류한는 multi class classification 이다. 필요한 데이터는 google drive 에 오려 load 했고, 실습 코드의 Project_utils.ipynb 에 있는 custom dataset class 와 pytorch 의 torch.utils.data.DataLoader library 를 이용했다. 데이터 시각화 역시 Project_utils.ipynb 의 image_show() 함수를 사용하였다. 시각화 코드와 결과는 다음과 같다.

```
[5]  1 # visualize data
     2 # character - encoded_label dict , 데이터 확인해보고 싶을 때 참조
     3 label_dict = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9,
     4 'A': 10, 'B': 11, 'C': 12, 'D': 13, 'E': 14, 'F': 15, 'G': 16, 'H': 17,
     5 'I': 18, 'J': 19, 'K': 20, 'L': 21, 'M': 22, 'N': 23, 'P': 24, 'Q': 25,
     6 'R': 26, 'S': 27, 'T': 28, 'U': 29, 'V': 30, 'W': 31, 'X': 32, 'Y': 33,
     7 'Z': 34, 'a': 35, 'b': 36, 'd': 37, 'e': 38, 'f': 39, 'g': 40, 'h': 41,
     8 'i': 42, 'j': 43, 'm': 44, 'n': 45, 'o': 46, 'q': 47, 'r': 48, 't': 49, 'u': 50, 'y': 51}
     9
    10 # image_show function : num 수 만큼 dataset 내의 data를 보여주는 함수
    11 def image_show(dataset, num):
    12   fig = plt.figure(figsize=(10,10))
    13
    14   for i in range(num):
    15     plt.subplot(1, num, i+1)
    16     plt.imshow(dataset[i+1200][0].squeeze(), cmap = "gray")
    17     plt.title(dataset[i+1200][1])
    18
    19 image_show(train_data, 8)
```



< figure 8. Training set visualizing code and data >

label 에 대한 dictionary 를 정의하고, 데이터 8 개를 추출하여 시각화 했다.

데이터 시각화 결과 같은 label 임에도 font 가 다르기에 여러가지 feature 가 섞여있는 것을 확인할 수 있었다. 따라서 실습코드, 혹은 기존 과제에서 사용한 CNN model 들 보다 복잡한 model 을 구성했다. 또한 데이터의 크기가 90 x 90 이고, Fully connected layer 에서의 연산량을 줄이기 위해 pooling layer 을 최소한 4 번 이상(4 번의 pooling 에서 모두 same padding 을 사용할 경우 5 x 5 크기의 output 이 생성된다) 사용하고자 하였다. 또한, 일반적으로 layer 가 깊어질수록 channel 수는 증가하게 구성함으로 4 개의 block 을 구성할 때 convolution layer 의 parameter 수 조절을 위해 BottleNeck 구조를 이용하였다. BottleNeck 을 구성할 때 ResNet 의 convolution block 구성을 참고하였다. Pytorch 의 내장함수를 이용해 model 구조를 출력할 수 있지만, 너무 길어져 따로 표로 정리하였다. Convolution layer 의 구성은 아래 표와 같다.

< table. 1 Convolution Layer 구조 >

layer name	output size	layer	Parameter
conv1	45 x 45	7 x 7 x 16	stride = 1 padding = 3
		3 x 3 max pooling	stride = 2 padding = 1
conv2	22 x 22	1 x 1 x 16 3 x 3 x 16 1 x 1 x 32	x 2
		2 x 2 max pooling	stride = 2
conv3	11 x 11	1 x 1 x 32 3 x 3 x 32 1 x 1 x 128	x 2
		2 x 2 max pooling	stride = 2
conv4	5 x 5	1 x 1 x 128 3 x 3 x 128 1 x 1 x 256	x 2
		2 x 2 max pooling	stride = 2
conv5	2 x 2	1 x 1 x 256 3 x 3 x 256 1 x 1 x 512	x 2
		2 x 2 Avg pooling	stride = 2
Flatten	2048 x 1	dropout(p = 0.5)	
fc1	256 x 1	dropout(p = 0.3)	
fc2	52 x 1		
Activation : ReLU Optimzer : Adam (lr = 0.001, decrease in half after epoch = 3) Total parameter : 2,636,148			

가장 먼저 7 x 7 kernel, 16 channel 로 넓게 feature 를 출력하였다. 이는 ResNet 에서 사용한 방법을 차용하였고, 본 과제에서도 입력값의 크기를 줄이는게 가장 큰 문제였으므로 사용하게 되었다. Stride 는 1, padding 은 3 으로 데이터의 손실은 최소한으로 하였다. BottleNeck 구조에서 kernel 의 수는 layer 가 깊어질수록 증가하는 방향으로 설정하였다. Kernel 수를 맞추는 과정은 가장 복잡한 모델에서 channel 수를 줄여가면서 진행하였다. conv5 layer 에서 마지막에 average pooling 을 사용하였는데, 이 또한 ResNet 모델을 참고하였고, 실제로 Max pooling 를 사용했을 때보다 높은 accuracy 값이 나와서 사용하였다. Dropout 의 경우 Flatten layer 와 fc1 layer 에서 각각 0.5, 0.3 을 적용하였고, 0.5 이하에서 가장 결과값이 좋은 값을 사용하였다. fc2 layer 의 경우 가장 마지막에 정답을 예측하는 구간임으로 dropout 을 적용하지 않았다.

Convolution layer 는 He uniform initialization 을 적용하였고, batch normalize 는 weight = 1, bias = 0 으로 설정한 뒤 진행하였다.

Optimizer 는 Adam optimizer 를 사용하였고 learning rate = 0.001 로 설정하였다. 하지만 epoch 수가 4 일 때부터 학습 결과가 불규칙적으로 나왔고, 이를 해결하고자 learning rate 를 줄여주는 learning rate scheduler 를 적용하였다. StepLR 를 사용해 3 번째 epoch 이후 값을 0.5 로 줄여주는, 즉 0.0005 로 바꾸어 진행하였다. 해당 함수는 아래와 같다.

| scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.5)

이 때 step_size 는 해당 함수의 적용시기이며 gamma 는 감소하는 정도를 나타내는 parameter 이다. gamma 의 default 값은 0.1 이지만, 너무 줄이면 학습이 잘 되지 않을 것이라 판단하였다. 특히 시간 제한이 있어 epoch 수를 많이 가져갈 수 없어서 절반만 줄이는 방향으로 진행하였다. Cosine annealing, ReduceLROnPlateau 등 여러 scheduler 가 있지만 적은 epoch 수를 다루는 만큼 단순하고 명료한 scheduler 를 사용하였다.

총 epoch 수는 6, batch size 는 64 로 설정한 뒤 Training 을 진행했다.

```
train_loss_value, val_loss_value, train_acc_value, val_acc_value = 0, 0, 0, 0

train_loss_list = []
train_acc_list = []
val_acc_list = []

save_best = 0
```

< figure 9. 모델 평가 지표 저장 변수 >

Training 과 validation 평가를 위한 loss 값과 accuracy 값을 저장할 변수명을 설정한 함수이다.

다음은 training 과정에서 loss 와 accuracy 를 저장하기 위한 변수명이다. 또한 save_best 는 가장 높은 validation accuracy 값을 저장하고, 그 때의 model.pth 를 저장하기 위한 변수이다.

```
16 # Train
17 start = time.time()
18 for epoch in range(num_epochs):
19   total = 0
20   correct = 0
21 
22 
23 
24 
25 
26 
27 
28 
29 
30 
31 
32 
33 
34 
```

```
35     # evaluate loss and accuracy of train data set
36     _, predicted = torch.max(outputs, 1)                                  n+1))
37     correct += (predicted == labels).sum().item()
38     train_loss_value = loss.item()
39 
40     if (i+1) %100 == 0 :
41       train_loss_list.append(train_loss_value)
42       print("Epoch [{}/{}], Step [{}/{}], Loss:{:.4f}"\.
43             format(epoch+1, num_epochs, i+1, total_step, loss.item()))
44 
45   train_acc_value = correct*100 / (len(train_loader)*batch_size)
46   train_acc_list.append(train_acc_value)
47 
48   # Print Loss for Tracking Training
49   print('\nCompleted training Epoch', epoch + 1,
50         '\n Training Accuracy: %.2f%%' %(train_acc_value),
51         '\n Training Loss: %.4f' %train_loss_value)
```

< figure 10. Training 진행 함수 >

training 을 진행하기 위한 function 이다. time.time()을 이용해 학습 시간을 측정하였고, pytorch library 를 통해 학습을 진행하였다. total 과 correct 는 accuracy 를 계산하기 위한 변수이다.

Train 에 소요되는 시간을 측정 하기 위해 time 라이브러리에서 time.time() function 을 사용하였다. train 을 할 때에는 optimizer 를 초기화 해준 후 CNN model 에 데이터를 반복해서 입력하고 최적화를 진행함으로써 train 한다. 또한 accuracy 와 loss 값을 출력할 때 구분하기 편하기 위해 line 22 에 구분할 선을 출력해주었다.

```
36     _, predicted = torch.max(outputs, 1)
37     correct += (predicted == labels).sum().item()
38     train_loss_value = loss.item()
39 
40     if (i+1) %100 == 0 :
41       train_loss_list.append(train_loss_value)
42       print("Epoch [{}/{}], Step [{}/{}], Loss:{:.4f}"\.
43             format(epoch+1, num_epochs, i+1, total_step, loss.item()))
44 
45   train_acc_value = correct*100 / (len(train_loader)*batch_size)
46   train_acc_list.append(train_acc_value)
47 
48   # Print Loss for Tracking Training
49   print('\nCompleted training Epoch', epoch + 1,
50         '\n Training Accuracy: %.2f%%' %(train_acc_value),
51         '\n Training Loss: %.4f' %train_loss_value)
```

< figure 11. Loss, accuracy 출력 함수 >

앞서 선언한 train_acc_value, train_loss_value 를 사용해 각 지표를 출력하였다.

Training accuracy 와 loss 값을 출력하고, 후에 loss curve 를 plot 하기 위해 선언한 train_loss_list 에 append()로 저장해주었다. 또한 1 epoch 당 학습시간이 짧지 않아 training epoch 안에서의 학습 경과를 확인하기 위해 line 40~43 을 추가해 loss 값을 확인할 수 있게 하였다.

```
53  model.eval()
54  with torch.no_grad():
55      correct, val_acc, val_loss = 0, 0, 0
56
57      for images, labels in valid_loader:
58          images = images.to(device)
59          labels = labels.to(device)
60          outputs = model(images)
61          val_loss += criterion(outputs, labels)
62          _,predicted = torch.max(outputs.data,1)
63          correct += (predicted==labels).sum()
64
65      val_loss = val_loss/len(valid_loader)
66      val_acc = 100*correct/len(valid_data)
67      val_acc_list.append(val_acc)
68      print('\nAccuracy and Loss for {} valid images\.
69      \n valindation accuracy: {:.2f}%\n validation loss: {:.4f}'\.
70      format(len(valid_data), 100 * correct / len(valid_data), val_loss))
71
72  if val_acc > save_best:
73    print("Your model is improved than the last model and saved as \'model_{:02d}.pth\'\n\n".format(epoch+1))
74    torch.save(model.state_dict(), 'model_epoch_{:02d}.pth'.format(epoch+1))
75    save_best = val_acc
76  else:
77    print("Your model haven't improved from last one\n\n")
78
79 end = time.time()
80 print("Train takes {:.2f}minutes".format((end-start)/60))
```

< figure 12. Validation 평가 함수 >

Validation data 에 대한 loss, accuracy 값을 계산하고 출력하는 함수이다.

Validation 평가는 이전 과제들과 동일한 함수를 사용했다. 다만 accuracy 를 기준으로 best model 을 저장하는 함수를 추가하였다. 저장 했을 시 저장된 모델 이름과 함께 저장되었다는 문장을 출력하였고, accuracy 값이 증가하지 않았을 경우도 따로 출력하였다.

```
============training epoch 7============
Epoch [7/7], Step [100/582], Loss:3.0314
Epoch [7/7], Step [200/582], Loss:3.0144
Epoch [7/7], Step [300/582], Loss:3.1248
Epoch [7/7], Step [400/582], Loss:3.0172
Epoch [7/7], Step [500/582], Loss:3.0175

Completed training Epoch 7
 Training Accuracy: 94.73%
 Training Loss: 3.0667

Accuracy and Loss for 7800 valid images
 valindation accuracy: 95.00%
 validation loss: 3.0346
Your model is improved than the last model and saved as 'model_07.pth'


Train takes 9.98minutes
```

< figure 13. 최종 출력 결과 >

학습 최종 결과이다. Training accuracy 는 94.73%, validation accuracy 는 95.00% 임을 확인할 수 있다.

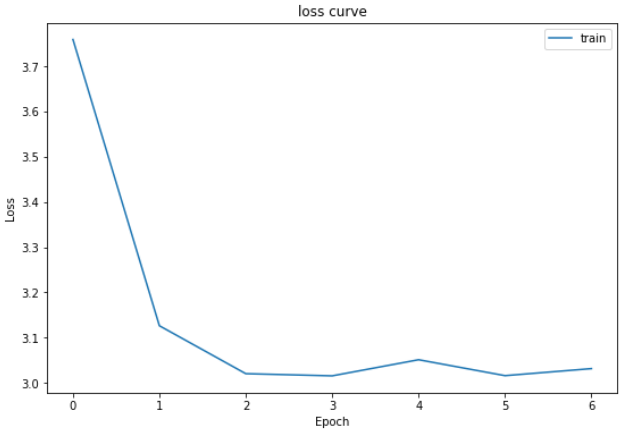
Training 과정의 모든 출력 결과를 레포트에 첨부하긴 너무 길어 최종 결과만 첨부하였다.

4. 결과 및 토의

최종 모델의 training 과 validation accuracy 는 아래와 같다.

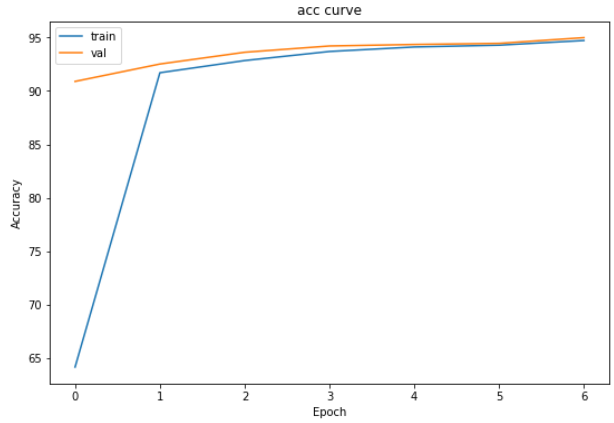
Training accuracy : 94.73%
Validation accuracy : 95.00%

loss curve 와 accuracy curve 를 살펴보면 다음과 같다.



< figure 14. Training Loss curve >

1~2 epoch 에서 가장 크게 줄어들고, 그 뒤로는 큰 변동이 없는 모습을 보인다.



< figure 15. Training, Validation accuracy curve >

Training 과 validation dataset 에 대한 accuracy curve 이다. Training curve 는 65% 가량에서 1 epoch 만에 크게 증가한 모습을 보이고, validation 은 처음부터 90% 가량의 정확도를 보인다. 하지만 2 epoch 부터는 두 dataset 모두 비슷한 정확도를 보인다.

Loss 값 변화의 경우 3.0 언저리에서 크게 벗어나지 못하는 모습을 보였다. 아무래도 50 개 이상의 분류문제이다 보니 일정 수준의 loss 는 기본적으로 발생하는 것으로 보인다. Accuracy 의 경우 validation 은 학습 초기에 training accuracy 보다 큰 값을 보였는데, 이는 아마 dropout 의 영향이라고 생각한다. Training 의 경우 dropout 으로 인해 빼먹는 data 가 있지만 validation 의 경우 없기 때문이다. 하지만 어느정도 학습을 진행하고 나면 dropout 의 영향력은 줄어드는 것으로 판단된다. 학습 후반부를 보면 95% 이상을 넘어가기 힘든 모습을 보인다. 이는 영문자, 숫자에서

비슷한 feature 들이 존재해 비교적 간단한 모델에서는 쉽게 분류가 되지 않는 것으로 판단된다. 학습시간이 제한되어 있고, 그로 인해 간단한 모델을 만들어야 하는 제약조건 하에서 어느정도 포기해야 하는 부분이 생긴 것이다.

좀 더 나은 학습 결과를 위해서 다음과 같은 방법을 사용할 수 있다.

먼저 당연하지만 model 의 complexity, 혹은 epoch 수의 증가이다. Complexity 를 증가시키면 본 과제에서 사용한 모델보다 좀 더 복잡한 feature, 혹은 class 별로 유사한 feature 를 잘 구분할 수 있게되고, 그로써 정확도 또한 증가할 것이다. 또한 epoch 수를 증가시킴으로 인해 feature 의 정확성을 올리고, 동시에 다른 learning rate scheduler 를 이용해 local minimum 을 벗어날 수 있을 것이다. 특히 validation accuracy 가 train accuracy 보다 큰 것을 확인할 수 있는데, 이는 모델의 학습이 완전히 끝나지 않았다고 해석할 수 있다. 즉, epoch 수를 늘려 더 높은 정확도를 얻을 수 있을 것이라 예측할 수 있다. 하지만 이 경우 training set 에 대한 overfitting 을 주의해야 한다.

또 다른 방법은 image augmentation 을 사용하는 것이다. Image 의 회전, 확대, 축소 또는 noise 를 주어 원본과 살짝 다른 입력값을 만들어 학습을 진행해 data 의 수를 증가시키는 방법이다. 이 방법은 data 의 수가 증가하는 것이 아니라 입력할 때 입력값을 수정하는 방식이기 때문에 학습 시간에 큰 영향을 주진 않는다. Data 의 수가 늘어나는 효과를 통해 overfitting 을 줄이고, 더 높은 정확도를 기대할 수 있다. 이 방법은 특히 data 의 수가 많지 않을 때 주로 사용한다.

마지막으로 사용할 수 있는 방법은 이전에 사용했던 convolution layer 의 feature map 을 가져와 사용하는 것이다. 이는 2017 년에 발표된 Densenet 에 적용했던 개념으로, 이전에 사용했던 feature map 을 이후에 있는 layer 에서 사용하는 것이다. 이렇게 했을 때 deep learning 의 고질적인 문제인 vanishing gradient 를 어느정도 해결할 수 있고, 적은 채널 수를 이용해 각 레이어의 feature 연결하여 다음 레이어로 전달하여 적은 parameter 수로 학습이 가능하다. 즉 시간적 측면과 computational force 측면에서 장점이 존재한다.

5. 참고 문헌

1. https://arxiv.org/abs/1608.06993 (DensNet)
2. https://arxiv.org/abs/1512.03385 (ResNet)
3. Francois Chollet, Deep Learning with Python, 1st Edition, Manning Publications
4. Ian Goodfellow, Deep Learning, MIT Press