

Course Title (과목명)	인공지능(딥러닝)개론 (01)
HW Number (HW 번호)	HW3
Submit Date (제출일)	2020-12-11
Grade (학년)	3 rd Grade
ID (학번)	20161482
Name (이름)	박준용

Project

1. 과제 목표

• EMNIST dataset (by merge) Classifier

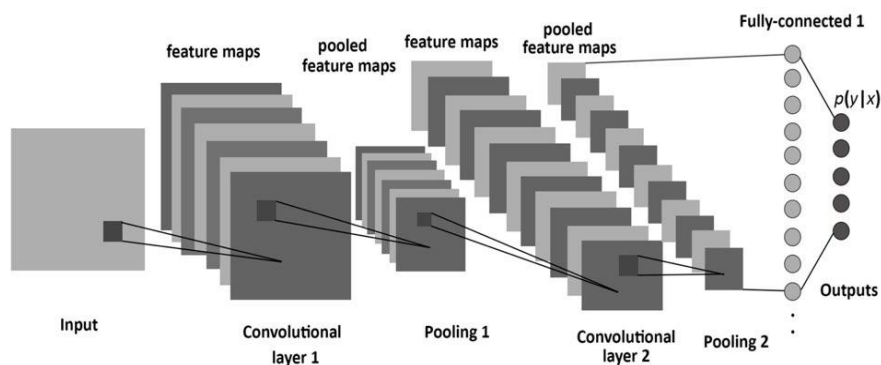
- 영문 및 숫자 손글씨 데이터 분류하기 (digit/letter 구분, EMNIST by merge의 47개 class 구분)
- 자유롭게 모델 구성: Accuracy를 높게 하는 것 목표
- Google Colab GPU 환경 (15분 이내에 Train 완료)

이를 통해서 MNIST보다 더욱 복잡한 EMNIST를 분류하는 과정을 통하여, image classification에 있어 neural network의 효과적인 설계 방법을 파악하며, 나아가 성능과 cost를 절충한 설계 방법을 통해 더욱 복잡한 이미지 분류 문제 또한 효과적으로 수행하는 것을 목표로 한다.

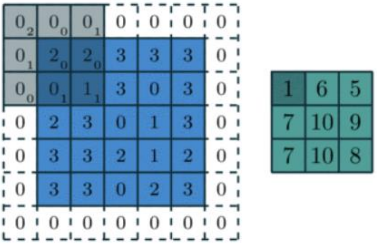
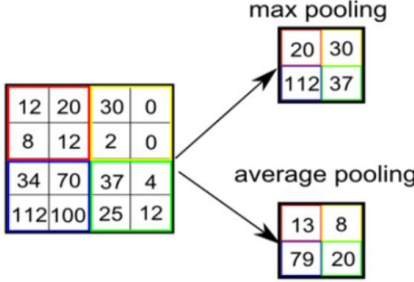
2. 배경 이론

1) CNN (Convolutional Neural Network)

CNN (합성곱 신경망)은 이미지나 행렬 등의 시각적 정보를 분석하는 데에 사용되는 Multilayer Perceptron의 한 종류이며, 합성곱의 개념을 도입하여 입력 데이터로부터 특징을 추출하여 더욱 빠르고 정확한 분류를 가능하게 하는 Neural Network이다.



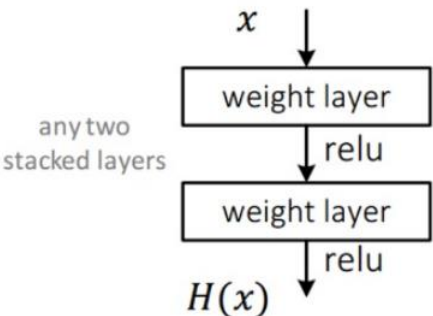
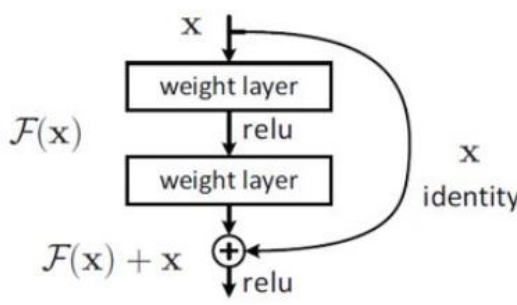
CNN은 위의 이미지와 같이 input layer와 output layer, 그리고 여러 hidden layer의 조합으로 이루어져 있다. 우선, Convolutional layer는 입력 데이터로부터 특징을 추출하고 feature map을 제작하여 분류하기 용이한 데이터를 만들어내는 역할을 한다. 이 때 특징을 도출하는 함수는 필터(kernel)이며, 데이터에 특성이 존재할 경우 큰 결과값을 보이며 그렇지 않을 경우 낮은 값을 보이는 형식이다. 이를 Pooling layer를 통하여 특정하기 쉬운 데이터를 선별하여 데이터의 dimension을 줄일 수 있으며, 이들 layer로부터 계산된 결정함수의 값을 비선형으로 변환하는 activation function을 통하여 가공할 수 있다. 이러한 형식으로 진행된 feature map들을 여타 많은 neural network와 같이 fully-connected layers를 통하여 가공한 뒤 output을 도출하게 된다.

	
Filter and Zero padding	Pooling: max pooling and average pooling

CNN은 모든 데이터에 대한 뉴런을 fully connect하지 않고 일부분에 해당하는 뉴런만 상호 연결 관계를 취한다. 위의 이미지에서 필터는 전체 데이터의 일부분을 convolution 하여 kernel의 데이터에 해당하는 만큼 convolution하여 feature map을 생성한다. 이 때, 필터가 우하단으로 이동하는 간격을 stride라고 하며, 데이터의 크기를 조절하고 오차를 줄이기 위하여 가장자리에 0의 값을 덧붙이는 방법을 zero padding이라고 한다.

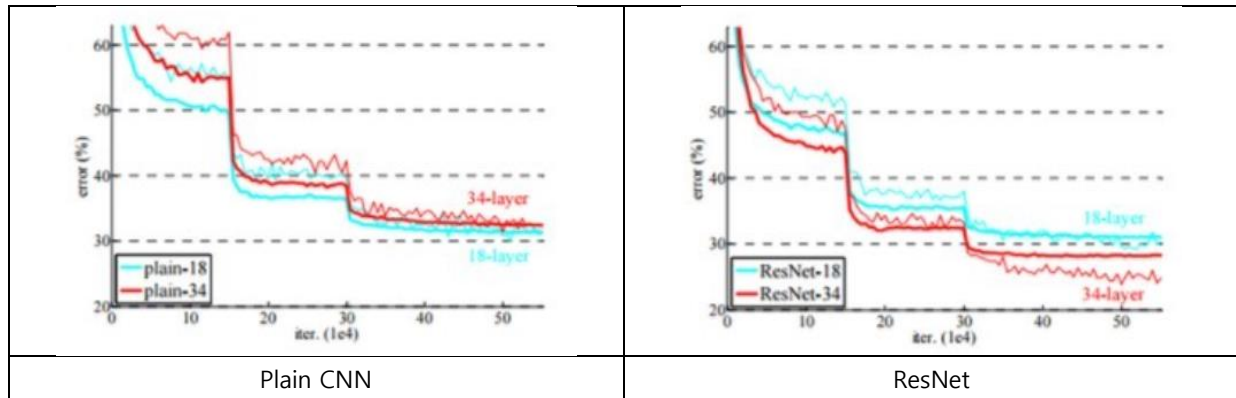
또한, 이러한 형식으로 생성된 feature map을 pooling할 때 여러 방법을 사용할 수 있는데, 대표적인 방법으로 max pooling이 있다. Max pooling은 주어진 filter 안에서 depth가 가장 큰 값을 선정하는 방식으로 현재 가장 널리 쓰이고 있으며, 이외의 방법으로는 depth의 평균을 취하는 average pooling 등이 존재한다.

2) ResNet (Residual Neural Network)

	
Plain CNN	ResNet

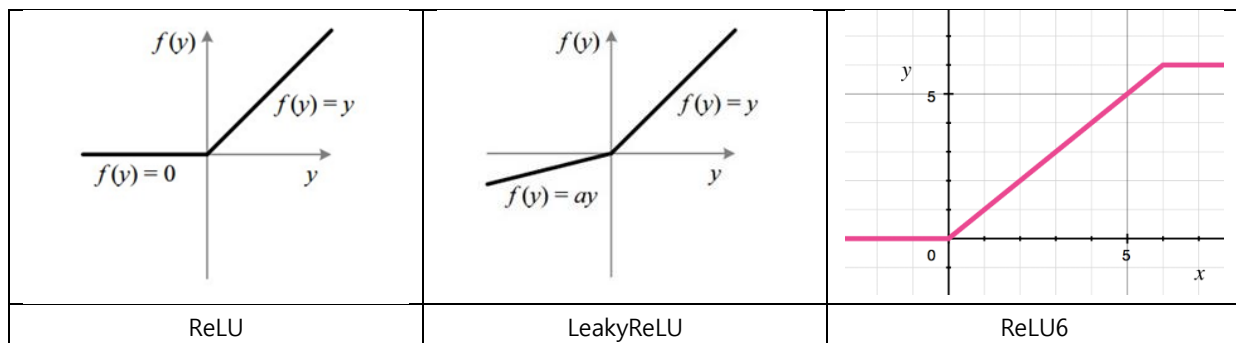
ResNet은 이미지 인식 분류 경진대회인 ILSVRC '15에서 가장 높은 성능을 보인 인공 신경망이다. 일반적인 CNN model과 ResNet model의 가장 큰 차이는 layer의 입력을 layer의 출력에 바로 연결시키는 'skip connection' 방법의 사용이며, 이에 따라 network depth가 깊어짐에 의해 생기는 gradient vanishing을 피하여 degradation 없이 더욱 깊은 network를 설계할 수 있다.

위의 그림과 같이 Plain CNN에서는 forward propagation 시 입력 x 를 출력 y 로 mapping하는 함수 $H(x)$ 를 얻는 것을 목적으로 하며, 이 때 신경망은 $H(x) - y$ 를 최소화하는 방향으로 학습을 진행한다. 그러나 ResNet에서는 이전 layer의 출력값인 $F(x)$ 에 이전 layer의 입력인 x 값을 더해주어 출력 함수의 구조가 $H(x) = F(x) + x$ 의 형태가 되며, 이 경우 신경망은 $F(x) = H(x) - x$ 를 최소화하는 방향으로 학습이 진행된다. 이 경우, 각 layer의 입력이 그 전 layer의 출력에 의존하지 않으며 각 layer가 독립적으로 작동하므로 layer의 path가 짧아지며, 결론적으로는 error를 줄일 수 있다.



위의 plain CNN과 ResNet의 layer 별 error graph를 살펴보았을 때, Plain CNN에서는 gradient vanishing 문제로 인하여 network depth가 큰 model이 더 높은 error를 나타내는 것을 볼 수 있지만, ResNet에서는 network depth가 큰 model이 더 낮은 error를 나타내는 것을 확인할 수 있다.

3) ReLU (Rectified Linear Unit)



ReLU 함수는 $x > 0$ 일 경우 input 함수의 기울기, $x < 0$ 일 경우 기울기가 0의 구조를 가진 비선형 activation function이다. 위와 같이 매우 간단한 구조를 가져 다른 activation function인 tanh, sigmoid 함수 등과 비교할 때 학습이 빠르고, cost가 작으며, 구현이 간단하다는 장점을 가지고 있다.

그러나, 기존 ReLU 함수는 $x < 0$ 일 경우 기울기가 0이기 때문에 dying ReLU 문제가 일어날 수 있다. LeakyReLU 함수는 $x < 0$ 일 경우에도 기울기를 0이 아닌 작은 값으로 구성하여 이 문제를 해결할 수 있다. 또한, ReLU의 상한값을 6으로 제한하여 precision을 향상시킨 ReLU6 등의 함수가 있다.

4) Softmax, log_softmax

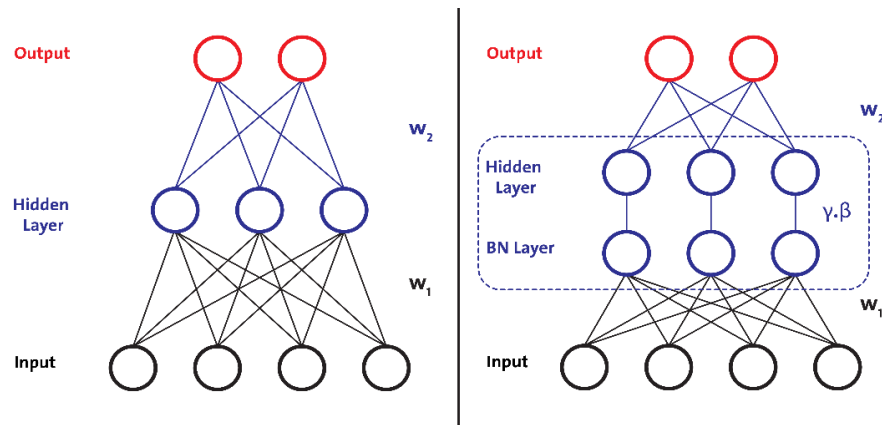
$$y_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Softmax 함수는 비선형 activation function의 일종이며, 입력받은 값을 0~1사이의 값으로 모두 normalize하여 출력하며 값들의 총합은 항상 1이 되는 특성을 가진 확률 분포 함수이다. 일반적으로는 model의 마지막 출력단에 사용되며 확률적 해석을 가능하게 하는 장점이 있다.

$$\log(y_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right) = x_i - \log(\sum_j \exp(x_j))$$

그러나 softmax를 사용하면 신경망은 vanishing gradient 문제에 취약해지며, 지수함수로 되어있기 때문에 overflow 문제 역시 일어날 수 있다. 이를 방지하기 위하여 log_softmax를 도입하여 결과값이 포화되지 않고, vanishing gradient 문제를 피할 수 있다.

5) Batch Normalization



Batch Normalization(배치 정규화)은 activation function의 활성화값 또는 출력값의 평균은 0, 표준 편차는 1로 normalize하는 과정을 의미한다. BN layer를 통하여 입력 x 와 가중치 w 의 convolution 값을 정규화하여 다음 batch에 입력으로 들어갈 때 값이 일정 범위를 벗어나지 않도록 한다. 이에 따라 learning rate를 높게 설정할 수 있어 학습 속도가 빨라지고, 정규화를 거칠 때마다 가중치에 대한 의존성이 줄어들기 때문에 gradient vanishing 문제도 어느 정도 해결할 수 있으며, overfitting을 효과적으로 감소시킬 수 있다는 장점을 가진다.

6) EMNIST



MNIST dataset이 숫자만 포함한 dataset인 반면, EMNIST(Extended MNIST) dataset은 위와 같이 숫자와 영문 대소문자 손글씨로 이루어진 dataset이며 MNIST와 마찬가지로 각각의 이미지마다 $1 \times 28 \times 28$ 의 크기를 가지고 있다. 이번 프로젝트에서 사용하는 dataset은 대/소문자 구별이 어려운 set을 구분하지 않도록 split한 by merge dataset으로, 697932개의 training set과 116323개의 test set, 47 unbalanced classes가 있다.

3. 과제 수행 방법

(a) Neural Network 선정

우선 지금까지 실습하였던 model 중에서 MNIST에 대하여 성능이 좋았던 model은 CNN, LSTM, GRU가 있으므로, 이 세 가지 neural network에 대해서 어느 정도 파라미터를 수정해본 이후 가장 효과가 있

는 model을 선정할 수 있었다.

수행 결과, GRU와 LSTM의 경우 EMNIST dataset이 recurrent한 정보를 요구하지 않으며, hidden layer와 layer 수가 늘어났을 때 시간 대비 상승하는 accuracy가 적었다. 그러므로, image classification에 적합하고 여러 layer를 쌓아 model을 자유도 높게 customization할 수 있는 CNN model을 선정하여, sequential model 설계 및 hyperparameter 세분화를 수행하였다.

(b) CNN model 설계

```
class ConvNet(nn.Module):
    def __init__(self, num_class):
        super(ConvNet, self).__init__()
        self.layer1a = nn.Sequential(
            nn.Conv2d(1, 10, 3, stride=1, padding=1),
            nn.BatchNorm2d(10),
            nn.ReLU()
        )
        self.layer1b = nn.Sequential(
            nn.Conv2d(10, 16, 3, stride=1, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU()
        )
        self.layer2a = nn.Sequential(
            nn.LeakyReLU(),
            nn.Conv2d(16, 16, 3, stride=1, padding=1),
            nn.BatchNorm2d(16)
        )
        self.layer2b = nn.Sequential(
            nn.LeakyReLU(),
            nn.Conv2d(16, 16, 3, stride=1, padding=1),
            nn.BatchNorm2d(16)
        )
        self.layer2c = nn.Sequential(
            nn.Conv2d(16, 32, 5, stride=1, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU()
        )
        self.layer3a = nn.Sequential(
            nn.Conv2d(32, 64, 7, stride=1, padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )
        self.layer3b = nn.Sequential(
            nn.Conv2d(64, 128, 7, stride=1, padding=3),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.AdaptiveAvgPool2d(6),
            nn.Flatten()
        )
        self.fc0 = nn.Sequential(
            nn.Linear(6*6*128, 360),
            nn.BatchNorm1d(360),
            nn.ReLU()
        )
        self.fc1 = nn.Sequential(
            nn.Linear(360, 120),
            nn.BatchNorm1d(120),
            nn.ReLU()
        )
        self.fc2 = nn.Sequential(
            nn.Linear(120, 84),
            nn.BatchNorm1d(84),
            nn.ReLU()
        )
        self.fc3 = nn.Sequential(
            nn.Linear(84, num_class),
            nn.BatchNorm1d(num_class)
        )
        self.maxpool = nn.MaxPool2d(max_pool_kernel)

    def forward(self, x):
        x = self.layer1a(x)
        x = self.layer1b(x)
        res = x

        x = self.layer2a(x)
        x = self.layer2b(x)
        x = self.maxpool(x+res)
        x = self.layer2c(x)

        x = self.layer3a(x)
        x = self.layer3b(x)
        x = x.reshape(x.size(0),-1)
        x = self.fc0(x)
        x = self.fc1(x)
        x = self.fc2(x)
        x = F.log_softmax(self.fc3(x))
        return x
```

처음에는 Convolution layer, Dense layer, ReLU, Max Pooling layer 등의 layer 요소들을 이용한 일반적인 CNN 구조로 모델을 구성하였다. 우선, 유사한 글자들을 더욱 효과적인 형태로 분류하기 위하여 Conv2d layer를 이용하여 channel size를 1에서 128로 증가시켜 정확도를 꾀하였다. 또한, 진행 도중 2x2의 Max Pooling을 2번 수행하여, Dense Layer에 도달하기 전까지 총 $6*6*128 = 4608$ 의 channel을 생성한다. 이 channel을 fully connect하여 다시 channel의 수를 num_class (model 1의 경우 2, model 2의 경우 47)의 수까지 줄일 수 있다. 이 때 각 sequential layer마다 BatchNorm을 적용하고 ReLU 함수로 활성화하였으며, model의 마지막 출력단에 softmax 함수를 사용하여 normalization 하였다. 해당 모델을 사용한 결과, model 1의 경우 대략 92%대 후반, model 2의 경우 90%대 중반 정도의 Test Accuracy를 보였다.

이후, 더 좋은 accuracy를 구현하기 위하여 여러 함수를 도입하였으며, 그 중 Dense Layer에 입력하기 전 Conv2d의 출력을 Max Pooling 대신 적응형 함수인 AdaptiveAvgPool2d를 사용하였다. 또한, vanishing gradient 문제를 방지하기 위하여 softmax 함수를 log_softmax 함수로 교체하였다.

Model의 구조 측면에서도, ResNet의 구조를 참고하여 image channel의 크기가 16일 때 같은

channel size를 반복하는 2개의 Conv2d layer를 추가한 뒤, 이전의 입력을 res로 저장한 후 이후의 출력값과 더하여 max pooling을 수행하는 skip connection을 적용하였다. 이는 유사한 글자들을 Residual Training 방법을 통하여 더욱 세세하게 학습할 수 있게 하며, vanishing gradient 오류를 감소시켜 network depth를 늘릴 수 있었다. Convolutional kernel size가 고정되었을 때의 activation function은 LeakyReLU를 사용하였다.

위와 같은 구조를 종합하여 설계된 model의 구조와 image tensor의 dimension을 정리한다면 아래와 같다. 코드에서 model 명령어를 입력하여 더욱 자세하게 살펴볼 수 있다.

Layer		[C, W, H]	비고
Input		[1, 28, 28]	
layer1a	Conv2d (kernel: 3, padding:1)	[10, 28, 28]	
layer1b	Conv2d (kernel: 3, padding:1)	[16, 28, 28]	res
layer2a	Conv2d (kernel: 3, padding:1)	[16, 28, 28]	
layer2b	Conv2d (kernel: 3, padding:1)	[16, 28, 28]	
	MaxPool2d (2)	[16, 14, 14]	x+res
layer2c	Conv2d (kernel: 5, padding:2)	[32, 14, 14]	
layer3a	Conv2d (kernel: 7, padding:3)	[64, 14, 14]	
layer3b	Conv2d (kernel: 7, padding:3)	[128, 14, 14]	
	AdaptiveAvgPool2d (6)	[128, 6, 6]	
	Flatten	[4608]	reshape
fc0	Dense	[4608, 320]	
fc1	Dense	[320, 120]	
fc2	Dense	[120, 84]	
fc3	Dense	[84, num_class]	log_softmax

(c) Hyper-Parameters 지정

설계한 model을 기반으로, output의 channel 개수가 다른 model 1과 2의 특성이 다르기 때문에 Hyperparameter를 model별로 따로 정의해 주었다.

<pre>#HyperParameters num_class = 2 batch_size = 128 max_pool_kernel = 2 learning_rate = 0.0025 num_epochs = 6</pre>	<pre>#HyperParameters num_class = 47 batch_size = 128 max_pool_kernel = 2 learning_rate = 0.001 num_epochs = 5</pre>
Model 1	Model 2

model 1의 경우, epoch가 늘어날수록 성능 향상이 일어나 제한 시간인 15분까지 학습할 수 있는 최대 epoch인 6으로 설정하였으며, model 2의 경우 epoch가 5에서 더 늘어날 경우 overfitting이 생겨 num_epochs 값을 5로 고정하였다. batch_size는 학습 시간과 효율을 절충하여 128로 설정하였으며, 이후 learning rate를 조절하며 accuracy를 미세 조정할 수 있었으며, 최종 결정된 learning rate의 값은 위와 같다.

4. 결과 및 토의

(a) Training 결과 및 Test Accuracy

```

Thu Dec 10 16:41:06 2020
+-----+
| NVIDIA-SMI 455.45.01    Driver Version: 418.67    CUDA Version: 10.1    |
+-----+-----+-----+-----+-----+
| GPU   Name               Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|     Memory-Usage | GPU-Util  Compute M. |
|-----+-----+-----+-----+-----+
|  0 Tesla P100-PCIE...    Off          | 00000000:00:04:0 Off |                    0 |
| N/A   35C    P0      31W / 250W | 1469MiB / 16280MiB |           0%      Default |
+-----+-----+-----+-----+-----+

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

```

본격적인 결과 분석에 들어가기 앞서, model을 train할 때에는 Google Colab에서 제공하는 Tesla P100-PCIE GPU를 사용하는 환경에서 실행되었다. Tesla P100-PCIE는 Colab에서 제공하는 GPU 중에서 가장 빠른 성능을 보이는 GPU 중 하나로, 이 GPU를 사용하였을 때만 Training Time이 15분을 넘지 않는다. 여타 Colab에서 제공하는 다른 GPU (Nvidia K80, T4, P4 등)를 사용하였을 때는 15분이 넘는 결과가 나올 수 있다.

<pre> Epoch [1/6], Step[1000/5453], Loss:0.1918 Epoch [1/6], Step[2000/5453], Loss:0.1495 Epoch [1/6], Step[3000/5453], Loss:0.1560 Epoch [1/6], Step[4000/5453], Loss:0.1695 Epoch [1/6], Step[5000/5453], Loss:0.2502 Epoch [2/6], Step[1000/5453], Loss:0.1771 Epoch [2/6], Step[2000/5453], Loss:0.1873 Epoch [2/6], Step[3000/5453], Loss:0.1736 Epoch [2/6], Step[4000/5453], Loss:0.1688 Epoch [2/6], Step[5000/5453], Loss:0.1773 Epoch [3/6], Step[1000/5453], Loss:0.1630 Epoch [3/6], Step[2000/5453], Loss:0.1843 Epoch [3/6], Step[3000/5453], Loss:0.0939 Epoch [3/6], Step[4000/5453], Loss:0.1081 Epoch [3/6], Step[5000/5453], Loss:0.1790 Epoch [4/6], Step[1000/5453], Loss:0.1574 Epoch [4/6], Step[2000/5453], Loss:0.2061 Epoch [4/6], Step[3000/5453], Loss:0.0832 Epoch [4/6], Step[4000/5453], Loss:0.1284 Epoch [4/6], Step[5000/5453], Loss:0.1683 Epoch [5/6], Step[1000/5453], Loss:0.1504 Epoch [5/6], Step[2000/5453], Loss:0.2314 Epoch [5/6], Step[3000/5453], Loss:0.1747 Epoch [5/6], Step[4000/5453], Loss:0.1942 Epoch [5/6], Step[5000/5453], Loss:0.1725 Epoch [6/6], Step[1000/5453], Loss:0.0979 Epoch [6/6], Step[2000/5453], Loss:0.1598 Epoch [6/6], Step[3000/5453], Loss:0.1067 Epoch [6/6], Step[4000/5453], Loss:0.1889 Epoch [6/6], Step[5000/5453], Loss:0.1547 Train takes 13.60minutes </pre>	<pre> Epoch [1/5], Step[1000/5453], Loss:0.3783 Epoch [1/5], Step[2000/5453], Loss:0.4479 Epoch [1/5], Step[3000/5453], Loss:0.3601 Epoch [1/5], Step[4000/5453], Loss:0.4808 Epoch [1/5], Step[5000/5453], Loss:0.3303 Epoch [2/5], Step[1000/5453], Loss:0.4093 Epoch [2/5], Step[2000/5453], Loss:0.3497 Epoch [2/5], Step[3000/5453], Loss:0.1983 Epoch [2/5], Step[4000/5453], Loss:0.1933 Epoch [2/5], Step[5000/5453], Loss:0.3784 Epoch [3/5], Step[1000/5453], Loss:0.2061 Epoch [3/5], Step[2000/5453], Loss:0.3674 Epoch [3/5], Step[3000/5453], Loss:0.2951 Epoch [3/5], Step[4000/5453], Loss:0.1892 Epoch [3/5], Step[5000/5453], Loss:0.2623 Epoch [4/5], Step[1000/5453], Loss:0.2458 Epoch [4/5], Step[2000/5453], Loss:0.2683 Epoch [4/5], Step[3000/5453], Loss:0.2466 Epoch [4/5], Step[4000/5453], Loss:0.2226 Epoch [4/5], Step[5000/5453], Loss:0.3725 Epoch [5/5], Step[1000/5453], Loss:0.3022 Epoch [5/5], Step[2000/5453], Loss:0.1832 Epoch [5/5], Step[3000/5453], Loss:0.1357 Epoch [5/5], Step[4000/5453], Loss:0.2417 Epoch [5/5], Step[5000/5453], Loss:0.2852 Train takes 11.53minutes </pre>
Model 1	Model 2

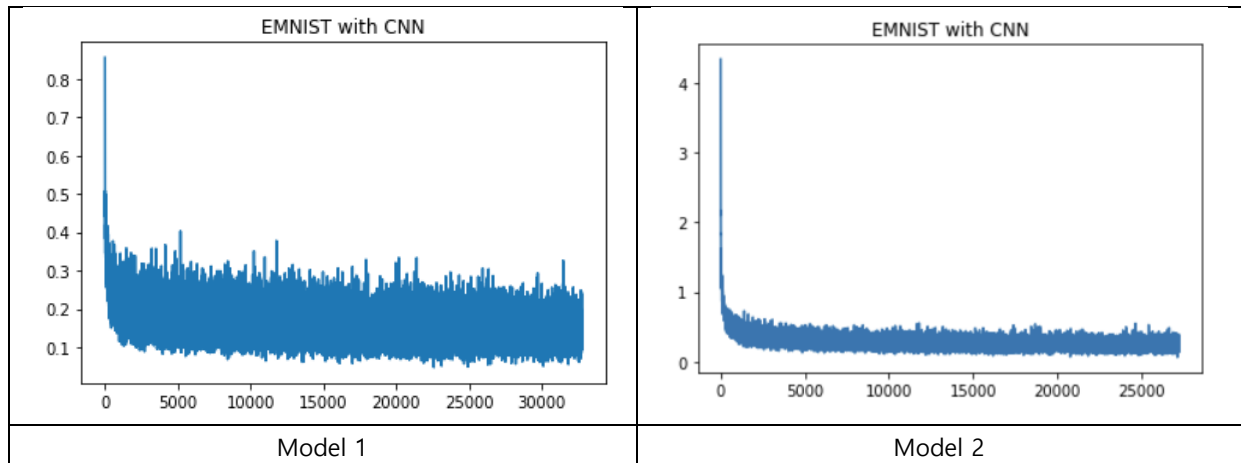
이전의 과제와 같은 형태로 Data를 Train하였다. Criterion은 Cross Entropy Loss를, Optimizer로는 Adam을 사용하였으며 loss를 plot에 사용하기 위하여 저장하여 append 하였다. 위와 같은 형태로 일정 간격의 step당 loss를 확인할 수 있으며, time 함수를 통하여 train에 소요된 시간을 확인할 수 있다.

```
Accuracy of the network on the 116352 test images: 93.26784240924093%
```

(Model 1)

```
Accuracy of the network on the 116352 test images: 91.09598459845985%
```

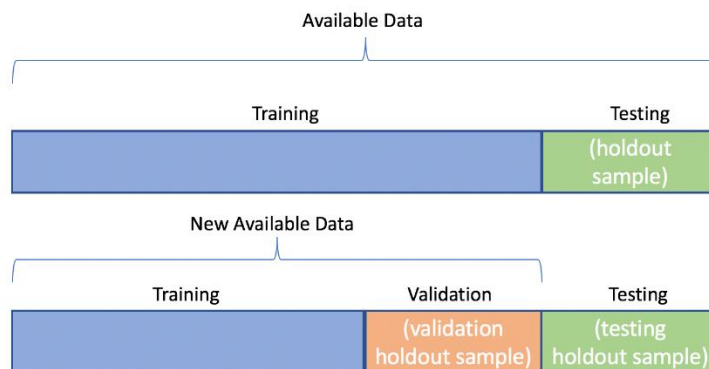
(Model 2)



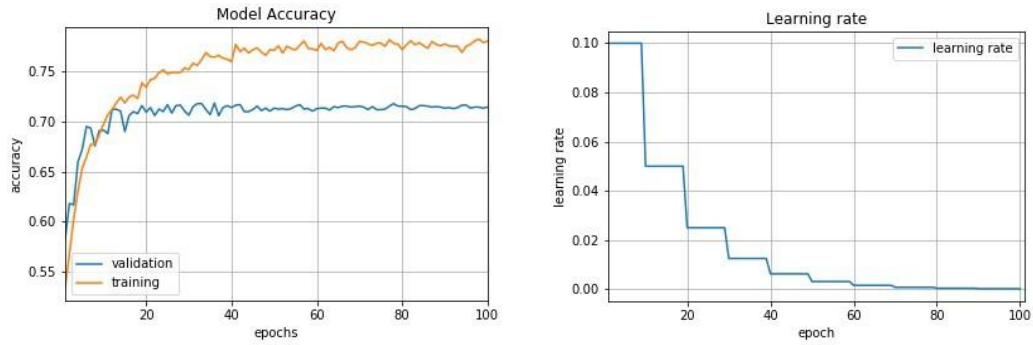
성능을 확인하였을 때 model 1의 정확도가 약 **93.27%**, model 2의 정확도가 약 **91.10%**를 달성하였다. matplotlib의 plt 함수를 사용하여 EMNIST에 대한 각 model의 loss를 plot하여 나타냈다.

(b) 결론 및 토의

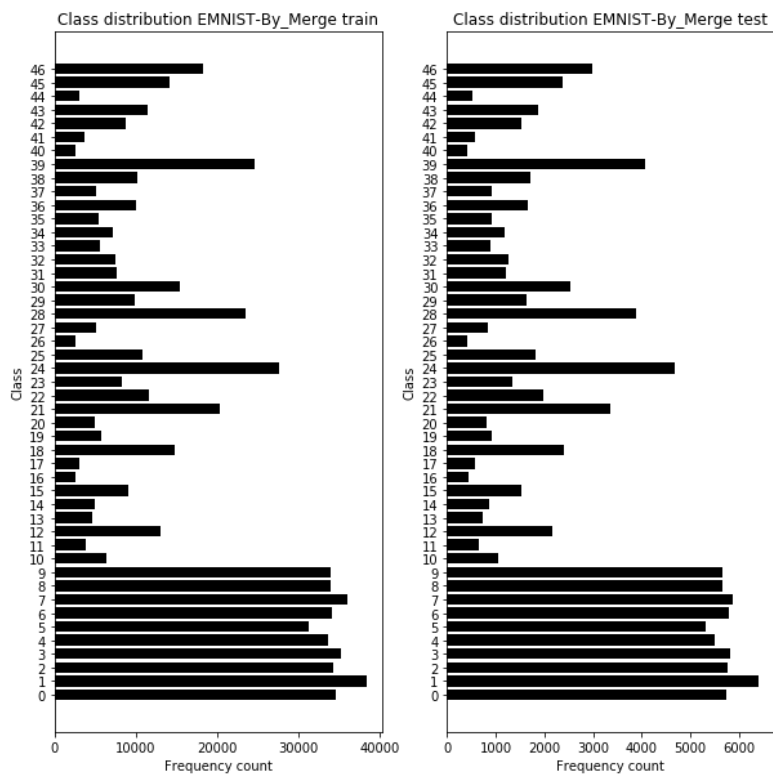
결론적으로 CNN, ResNet 및 기타 activation function과 pooling 등의 활용과 적절한 hyperparameter 조절을 통하여 각 모델에 대해 어느 정도의 정확도를 얻을 수 있었다. 대체로 model 1의 경우 93%, model 2의 경우 91% 수준에서 성능이 크게 좋아지지 않았으므로 이를 넘기는 것을 1차 목표로 하여 달성할 수 있었다. 더욱 높은 수준의 정확도를 가진 model을 제작하기 위하여 다음과 같은 사항을 고려할 수 있을 것이다.



- 우선 training set의 dataset을 일정 부분 split 하여 validation set으로 사용하여, training을 마치고 test set에 성능 평가를 하기 전 미리 검증하는 단계를 추가할 수 있다. 이를 통해서 각 epoch별로 validation loss를 확인하여 가장 작은 model을 test 단계에 사용한다면 혹시나 일어날 수 있는 overfitting 상황에 더욱 유연하게 대처할 수 있다.



- 또한, model을 training할 때 고정된 learning rate를 설정하여 optimizer에 input 하였지만, 학습이 진행되며 learning rate를 조정하는 방법 역시 사용할 수 있다. Learning rate decay는 Adam optimizer에서 제공하는 기능이지만, 위와 같은 형태의 learning rate의 step decay 또는 validation dataset의 accuracy와 비교하여 validation accuracy가 일정 이상일 때 learning rate를 줄이는 형태의 decay 역시 적응형으로 learning rate를 조정하여 test accuracy의 값을 높이는 데에 도움을 줄 수 있다.



- 또한, EMNIST by Merge dataset은 각 class별 데이터 분포가 일정하지 않다. 비교적 균등한 수로 다수의 data가 존재하는 숫자 dataset과는 달리, 영문 dataset은 각 class별로 data의 수가 큰 차이를 보이며 절대적인 수량도 숫자 dataset보다 적다. 이는 특히 model 2에서 문제가 될 수 있는데, 숫자/영문자로 분류하여 학습할 때에는 data가 비교적 고른 분포를 보이는 반면 47 개의 개별적인 class를 분류하여 학습할 때에는 특정 data의 편향성이 높기 때문이다. 있기 때문이다. 이를 해결하기 위하여 data augmentation이나 skip connection 방법을 사용할 수 있다. Data augmentation은 모자란 class의 data를 증식할 수 있지만, dataset의 크기가 매우 커져 학습 시간이 커지는 단점이 존재한다.

- Pooling의 경우 2번까지는 유의미한 성능 향상을 불러일으켰지만 3번 이상의 pooling이 이루어지면 Dense Layer에 이르기까지 image size (W, H) 가 매우 작아져 정보의 생략이 필요 이상으로 많아졌다. 이로 인해 Accuracy의 향상이 되지 않거나 떨어지는 경향을 보였다.
- Batch Normalization을 해 주지 않을 경우 normalization이 되지 않아 loss의 값이 overflow하는 경우가 생길 수 있으며, 초기 loss값이 커진 상태에서 fitting을 하게 되므로 train loss의 감쇠가 느려지는 경향을 보였다.
- Layer의 사이에 Dropout을 적용하여 rate를 증가시킬수록 training accuracy의 증가량이 감소하며 loss도 잘 떨어지지 않는 경향을 보였다. Dropout은 overfit을 예방하는 효과가 있지만 cost가 증가하여 학습에 더욱 많은 시간이 소요되므로 사용하지 않았다.

5. References

Deep Learning in Python, F. Chollet, 1st Edition, Manning

Exploring EMNIST - another MNIST-like dataset, Simon Wenkel

(<https://www.simonwenkel.com/2019/07/16/exploring-EMNIST.html>)

ResNet (34, 50, 101): Residual CNNs for Image Classification Tasks, Neurohive

(<https://neurohive.io/en/popular-networks/resnet/>)

Convolutional neural network, Wikimedia Foundation.

(https://en.wikipedia.org/wiki/Convolutional_neural_network)

Activation Functions: Sigmoid, ReLU, Leaky ReLU and Softmax basics for Neural Networks and Deep Learning, Himanshu S

(<https://medium.com/@himanshuxd/activation-functions-sigmoid-relu-leaky-relu-and-softmax-basics-for-neural-networks-and-deep-8d9c70eed91e>)

EMNIST using Keras CNN, ashwani07 (<https://www.kaggle.com/ashwani07/emnist-using-keras-cnn>)

Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning, Suki Lau

(<https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1>)