

## 1. 과제 목표

특정 시간 내에 좋은 성능을 달성할 수 있는 CNN 모델을 설계한다.

빠르게 좋은 정확도에 수렴하는 학습 방법을 탐구한다.

이미지를 분류하는 문제를 풀 수 있는 다양한 방법을 탐구한다.

## 2. 배경 이론

이미지를 분류하는 task는 CNN을 이용한다. 왜냐하면 어떠한 이미지 픽셀 간의 local shape을 점차적으로 잡아내는 것을 목표로 하는 task를 CNN이 잘 풀기 때문이다. 이미지를 Sequential하게 푸는 방법들은 CRNN과 같이 cnn과 혼용하는 방법과, vision transformer와 같이 attention만 사용하는 방법이 효과적이라고 여겨지고 있다. Transformer 관련 현행 연구 중에는 distilled token을 활용한 Deit, 그냥 cls token만 활용한 Beit이 대표적이다. 그러나 이러한 방법들은 기본적으로 방대한 데이터와 메모리가 어느정도 보장되어야 하고, 충분히 시간을 들여서 학습을 시켜야 한다. 특히 transformer 모델은 pretrained를 갖고 와서 fine-tuning을 하는 방법을 사용하는 것이 효과적이기 때문에, 본 task에서 10분이라는 제한 시간을 이용할 수 없다. 마찬가지로 강화학습 계열 알고리즘 또한 시뮬레이션 설계와 reward를 얻을 수 있는 환경/보상/정책 등을 설계해서 학습해야 하기에 10분이라는 시간동안 모든 경우의 수를 고려하기 힘들다. 따라서 이러한 시간적 한계를 인지하고 간단한 CNN 모델을 타협했다.

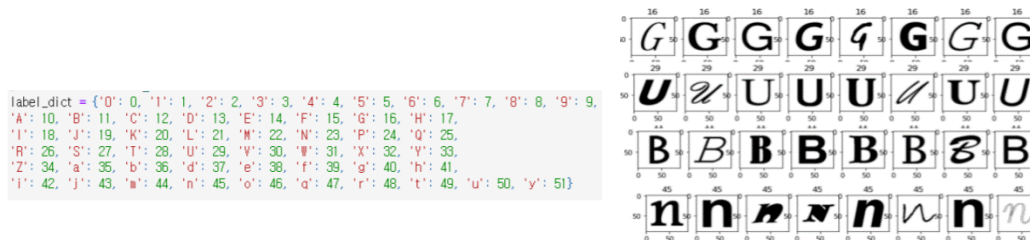
그래서 이번 프로젝트에 사용하기로 한 구조는 CNN을 쓰긴 하지만, CNN에 적용할 수 있는 비교적 최신 기법들을 적용해볼 것이다. CNN의 폭발적인 성공에 기여한 weight init method인 xavier initialization과 학습의 안정성을 위해 data normalization과 model 중간 output normalization(batch norm)을 사용해 볼 것이다. 또한 빠른 학습을 위해 resblock 구조를 차용하여 10분 안에 준수한 성능을 달성할 수 있도록 model을 설계해 볼 것이다. 마지막으로 activation을 relu를 활용해 학습을 진행하도록 할 것이다.

또한 복잡한 resnet 모델이 overfitting 될 경우가 존재하므로, 복잡한 모델에서 점차 경량화를 하면서 데이터셋에 맞는 모델을 찾아볼 것이다. 또한 성능이 빨리 수렴하지 않을 경우 lr\_scheduler를 사용해본다.

또한 training에 비해 valid의 성능이 잘 나오지 않는다면, 데이터 noise에 robust하게 만들기 위해 augmentation을 도입해보고, dropout과 regularizer를 도입해볼 것이다.

### 3. 과제 수행 방법

#### 3.1) 과제 접근 방법



이번 과제의 가장 큰 제약 사항은 시간과 데이터셋 제한이다. 사실 image가 10개에서 52개로 늘었고 gray scale 90\*90로 변했을 뿐, task 자체는 무척 쉬운 편이다. 추가적 데이터를 수집 및 투입을 못하기 때문에 모델 자체로 문제를 풀어야 한다. 안정적인 학습을 위해 가장 먼저 kaiming He의 init기법을 사용해 weight를 초기화할 것이다. 그리고 complexity가 어느정도 있는 모델과 경량화 모델을 둘 다 test 해볼 것이다. Complexity가 있다고 함은 resblock을 여러 개 쌓는다는 의미이다. 또한 빠른 학습이 관건이므로 batch size와 Learning rate를 조절해서 1~2번 epoch만에 제일 좋은 성능을 뽑아내는 모델을 사용해볼 것이다. 그리고 제일 좋은 성능을 뽑아내는 모델이 10분 안에 정확도가 어느 정도까지 올라가는지 체크해볼 것이다.

만약 성능이 좋지 않다면 시간이 허락하는 한 augmentation 기법의 도입을 고려해볼 것이다. 숫자와 문자데이터셋의 특성상 이미지를 flip하거나 너무 회전시키는 것은 오히려 학습을 방해하기 때문에, 약간의 이미지의 회전만 줄 것이다. 만약 시간 안에 원하는 성능이 달성되지 않으면 epoch별로 lr을 조절해본다. 또한 valid의 성능을 극대로 끌어 올리기 위해, dropout과 batch normalization을 도입해서 training 데이터에 너무 overfitting되지 않게 만들 것이다.

마지막으로 dataloader의 pin\_memory와 num\_workers를 사용해서 cpu core를 최대한 이용해보려고 한다. Colab의 경우 cpu가 core가 2개 이므로 이것을 활용해볼 것이다. 그리고 train data의 평균과 분산을 구한 다음에, test데이터에서 그 값으로 normalize해서 똑 같은 분포를 지니고 있다고 가정하고 적용해 볼 것이다.

#### 3.2) 경량화 모델 – 완화되기 전 모델

: 1개의 residual 연결만 사용

: maxpooling, relu, Dense layer 사용

: batch\_normalization 사용

```
class MyNet(nn.Module):
    def __init__(self, block, layers, num_classe):
```

```

super(MyNet, self).__init__()
self._norm_layer = nn.BatchNorm2d

self.conv1=nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3,
                     bias=False)
self.bn1 = nn.BatchNorm2d(self.inplanes)
self.relu = nn.ReLU(inplace=True)
self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
self.conv2=nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1,
                     bias=False)
self.downsample = nn.Sequential(
    nn.Conv2d(64, 128, kernel_size=1, stride=2, bias=False)
    ,
    nn.BatchNorm2d(128),
)
self.bn2 = nn.BatchNorm2d(128)
self.relu2 = nn.ReLU(inplace=True)
self.fc = nn.Linear(18432, num_classe)

for m in self.modules():
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode='fan_out', nonli
nearity='relu')
    elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)

def _forward_impl(self, x):
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)
    ident = self.downsample(x)
    x = self.conv2(x)
    x = self.bn2(x) + ident
    x = self.relu2(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)

    return x

def forward(self, x):
    return self._forward_impl(x)

```

### 3.3) 무거운 모델 - 2개

MyNet의 conv2를 layer 2개로 바꾸고 resnet18의 original resblock을 이용하여 설계

→ Layers [1,1], layers[2,2] 두개 test.

```
self.layer1 = self._make_layer(block, 128, layers[0], stride=1)
self.layer2 = self._make_layer(block, 256, layers[1], stride=1)
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc = nn.Linear(256, num_classe)

for m in self.modules():
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode='fan_out', nonli
nearity='relu')
    elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)

def _make_layer(self, block, planes, blocks, stride):
    norm_layer = self._norm_layer
    layers = []
    layers.append(block(self.inplanes, planes, stride))
    self.inplanes = planes
    for _ in range(1, blocks):
        layers.append(block(self.inplanes, planes, stride))
```

### 3.4) 경량화 모델 2 실험 – 완화된 시간 기준으로 이 모델 선택

: 2개의 residual 연결 사용, conv block 128 추가

: maxpooling, relu, Dense layer 사용

: batch\_normalization, dropout(0.4) 사용

```
class MyNet(nn.Module):

    def __init__(self, block, layers, num_classe):
        super(MyNet, self).__init__()
        self._norm_layer = nn.BatchNorm2d

        self.dilation = 1
        self.conv1 = nn.Conv2d(1, 32, kernel_size=7, stride=2, padding=
3,
                                bias=False)

        self.bn1 = nn.BatchNorm2d(32)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=2, padding
=1,
```

```

        bias=False)
self.bn2 = nn.BatchNorm2d(64)
self.relu2 = nn.ReLU(inplace=True)

self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1,
                        bias=False)
self.bn3 = nn.BatchNorm2d(128)
self.relu3 = nn.ReLU(inplace=True)

self.downsample1 = nn.Sequential(
    nn.Conv2d(32, 64, kernel_size=1, stride=2, bias=False),
    nn.BatchNorm2d(64),
)
self.downsample2 = nn.Sequential(
    nn.Conv2d(64, 128, kernel_size=1, stride=2, bias=False),
    nn.BatchNorm2d(128),
)
self.dropout = nn.Dropout(0.4)
self.fc = nn.Linear(4608, num_classes)

for m in self.modules():
    if isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
    elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
        nn.init.constant_(m.weight, 1)
        nn.init.constant_(m.bias, 0)

def _forward_impl(self, x):
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.maxpool(x)
    ident = self.downsample1(x)
    x = self.conv2(x)
    x = self.bn2(x) + ident
    x = self.relu2(x)
    ident = self.downsample2(x)
    x = self.conv3(x)
    x = self.bn3(x) + ident
    x = self.relu3(x)

    x = torch.flatten(x, 1)
    # print(x.shape)
    x = self.dropout(x)

```

```

        x = self.fc(x)

    return x

def forward(self, x):
    return self._forward_impl(x)

```

## 4. 결과 및 토의

### 1) pin\_memory 및 num\_workers test

pin\_memory, num\_workers test

```

# 1 분 56 초 num worker2 & pin memory
# 2 분 44 초 pin_memory
# 1 분 51 초 num worker2
# 2 분 28 초 num worker1

```

### 2) 모델 batch, lr 학습 성능 test

#### 2.1) 복잡한 resnet 모델[2,2]

: 복잡한 모델은 오래 학습하는 것에 비해 초기 도달이 좋지 않다. 3번 학습하면 10분 된다. 시간 안에 학습이 불가능하다.

Bs:64, lr:0.05

```
Epoch[1/5], Train Loss:1534.8766, Train Acc:0.30, Valid Loss:375.2853, Valid Acc:0.65.
```

128 0.005

```
Epoch[1/3], Train Loss:101.4009, Train Acc:0.92, Valid Loss:100.9463, Valid Acc:0.91
```

256 0.005 안 좋음.

128 0.001

```
Train Loss:294.0975, Train Acc:0.79, Valid Loss:612.8418, Valid Acc:0.48
```

## 2.2) 덜 복잡한 resnet 모델[1,1]

: 마찬가지로 오래 학습에 비해 초기 정확도가 높지 않다. 더군다나 계속 학습시켜도 시간 내에 valid 0.98을 넘지 못했다. 왜냐하면 10분 안에 5~6번밖에 학습할 수 없다.

128 0.001

```
Train Loss:126.6419, Train Acc:0.91, Valid Loss:164.0880, Valid Acc:0.87
```

## 2.3) 경량화 resnet – 기준 변경 전 10분 충족 모델

: 10분 내에 학습할 수 있고, 3가지 후보군 중에 2번 째의 epoch에서 제일 성능이 좋았다.

: batch size는 1024가 제일 빠른 속도로 수렴하고, lr를 0.0005를 사용했을 때 제일 효과적으로 학습했다.

: 위와 같은 파라미터로 학습했을 때, valid는 94%, training은 99%가 나왔다.

- 2048, 0.001 #10번 학습시키면 0.99

```
01:14 Train Loss:12.2450, Train Acc:0.85, Valid Loss:14.2221, sample Acc:0.86
```

- 1536 0.001

```
01:13 Train Loss:12.5847, Train Acc:0.88, Valid Loss:10.4948, sample Acc:0.90
```

- 1280 0.001

```
01:13 Train Loss:13.6400, Train Acc:0.89, Valid Loss:10.8366, sample Acc:0.91
```

- 1024, 0.001 #10번 학습시키면 0.99

```
01:13 Train Loss:14.2669, Train Acc:0.91, Valid Loss:10.0118, sample Acc:0.93
```

- 1024, 0.0005 #8번 학습 0.99

```
01:13 Train Loss:14.5174, Train Acc:0.91, Valid Loss:10.0198, sample Acc:0.94
```

- 1024, 0.005

```
Train Loss:25.6409, Train Acc:0.85, Valid Loss:17.8783, sample Acc:0.89
```

- 1024, 0.0001

```
Train Loss:23.1047, Train Acc:0.86, Valid Loss:19.0885, sample Acc:0.89
```

- 768, 0.001

```
01:14 Train Loss:16.2760, Train Acc:0.92, Valid Loss:11.0632, sample Acc:0.94
```

- 512, 0.001

01:16 Train Loss:20.2774, Train Acc:0.93, Valid Loss:12.9839, sample Acc:0.95

- 256, 0.001

01:14 Train Loss:35.6185, Train Acc:0.94, Valid Loss:21.1563, sample Acc:0.96

- 128, 0.0005 adam

01:14 Train Loss:64.9226, Train Acc:0.94, Valid Loss:41.5624, sample Acc:0.96

- 128, 0.001 adam

01:15 Train Loss:67.0578, Train Acc:0.94, Valid Loss:39.2943, sample Acc:0.97

- 128, 0.002 adam

01:13 Train Loss:69.9064, Train Acc:0.94, Valid Loss:41.8829, sample Acc:0.96

- 128, 0.005 adam

01:16 train loss:0.93, Valid Loss:44.1919, sample Acc:0.96

- 64, 0.001 adam

01:18 Train Acc:0.95, Valid Loss:77.4098, sample Acc:0.97

- 32 0.001 adam

01:20 Train Loss:254.4664, Train Acc:0.95, Valid Loss:152.7366, sample Acc:0.97

## 2.4) 경량화 모델 2 – 완화된 10분 기준 모델

: 기준이 완화되면서, 좀 더 다양한 기법과 모델을 실험할 수 있게 되었다.

: 위에서 학습할 수 없었던 resnet을 이용하여 실험해 보았지만, valid 성능이 오히려 94%보다 하락했다. -> 이는 데이터에 비해 모델이 더 복잡하다는 것을 깨달을 수 있었다.

: 따라서 경량화 모델을 기준으로 다양하게 실험하고 다양한 기법을 적용해서 generalization을 끌어올리도록 실험해보았다.

: base line – 경량화 모델 93.5

--- 모델 히든 채널 파라미터 조절 + augmentation 도입 : valid 한계치까지 학습

- layer수 그대로 and 전체적으로 cnn 차원 1/4 감소 : 0.9281 valid



```
# Epoch[9/10], Train Loss:7.9414, Train Acc:0.95  
# 2.328733094036579 tensor(0.9246, device='cuda:0')  
# Epoch[10/10], Train Loss:7.3846, Train Acc:0.95  
# 2.2121686339378357 tensor(0.9281, device='cuda:0')
```

- layer수 그대로 and 전체적으로 cnn 차원 1/2 감소

```
# Epoch[23/40], Train Loss:1.6554, Train Acc:0.99  
# 1.780292920768261 tensor(0.9538, device='cuda:0')
```

- layer수 그대로 and 전체적으로 cnn 차원 1/2 감소 + 128 block 추가

```
# Epoch[31/40], Train Loss:1.5932, Train Acc:0.99  
# 1.2462433874607086 tensor(0.9590, device='cuda:0')
```

- layer수 그대로 and 전체적으로 cnn 차원 1/2 감소 + 128 block 추가 + 256block 추가

```
# Epoch[28/40], Train Loss:1.7391, Train Acc:0.98  
# 1.2742572948336601 tensor(0.9582, device='cuda:0')
```

→ 기존 layer를 반으로 줄이고 128 block을 추가하는 것이 valid의 성능을 고려한 적합한 모델 구조이다.

→ augmentation을 도입하기 전보다 도입한 후가 valid의 성능을 유의미하게 끌어낼 수 있었다. 대신 학습 수렴시간 측면에서 적용을 안 한 모델이 2epoch만에 달성할 정확도를 적용을 한 모델은 5epoch나 걸릴 정도로 늘어났다.

--- 모델 regularizer & normalization

- cnn 차원 시작 32에서 128block 하나 추가했을 때 + dropout

```
# Epoch[39/40], Train Loss:1.8657, Train Acc:0.98
```

```
# 1.3427405655384064 tensor(0.9606, device='cuda:0')
```

- 32에서 128block 하나 추가했을 때 + dropout + regularizer

```
# Epoch[39/40], Train Loss:1.8657, Train Acc:0.98
```

```
# 1.3427405655384064 tensor(0.9604 , device='cuda:0')
```

→ dropout을 추가했을 때는 valid 성능이 개선되었지만, regularizer를 사용했을 때는 유의미한 성능 개선이 없었다. 따라서 Dropout만 추가하기로 했다.

--- 새로운 모델에 맞는 Learning rate 탐구

: 모델이 바뀌게 되면서, 경량화된 모델에 맞는 learning rate가 꼭 지금 모델에 맞으리라는 보장이 없다. 시간 제한 규정이 완화되면서 도입할 수 없었던 기법을 도입하게 되어서 모델이 바뀌었으니, 새로 테스트를 해보았다.

- 모델 with 0.001

```
Epoch[5/5], Train Loss:8.7186, Train Acc:0.94
```

```
2.086592584848404 tensor(0.9347, device='cuda:0')
```

- 모델 with 0.0005

```
Train Acc:0.92
```

```
valid acc : 0.92
```

- model with 0.005

```
Epoch[5/20], Train Loss:9.8456, Train Acc:0.93
```

```
2.388212338089943 tensor(0.9238, device='cuda:0')
```

- model with 0.002

Epoch[5/20], Train Loss:8.0757, Train Acc:0.94

2.0057855620980263 tensor(0.9368, device='cuda:0')

- model with 0.003

Epoch[5/20], Train Loss:8.6232, Train Acc:0.94

2.1346734762191772 tensor(0.9329, device='cuda:0')

→ 해당 모델에서 가장 빠르게 성능을 수렴하게 하는 learning rate는 0.002이었다.

--- 새로운 모델에 맞는 dropout 탐구

- dropout 0.3 -> 0.2 : 최고 valid가 0.9596이라서 오히려 떨어졌다.


Epoch[5/40], Train Loss:7.7175, Train Acc:0.94

1.9999942034482956 tensor(0.9362, device='cuda:0')


Epoch[34/40], Train Loss:2.4456, Train Acc:0.98

1.2996799945831299 tensor(0.9596, device='cuda:0')

- dropout 0.3 -> 0.1 : 최고 valid가 0.9586이라서 마찬가지로 떨어졌다. 대신 5 epoch 성능 수렴도는 훨씬 높다.(0.3은 0.9368)


12% | Epoch[5/40], Train Loss:6.6304, Train Acc:0.95

1.8345479518175125 tensor(0.9399, device='cuda:0')

98% | Epoch[39/40], Train Loss:2.0602, Train Acc:0.98

1.3829199895262718 tensor(0.9586, device='cuda:0')

- dropout 0.3 -> 0.4 : 최고 valid가 제일 높다. 대신 5 epoch 성능 수렴도가 낮은 편이다.


12% | Epoch[5/40], Train Loss:9.2802, Train Acc:0.93

2.0461990386247635 tensor(0.9318, device='cuda:0')


50% |

2.169578079134226 tensor(0.9651, device='cuda:0')

- dropout 0.3 -> 0.5 :

12% | Epoch[5/40], Train Loss:18.4056, Train Acc:0.93

3.8461123779416084 tensor(0.9365, device='cuda:0')

98% | Epoch[39/40], Train Loss:5.2022, Train Acc:0.98

2.2064896039664745 tensor(0.9642, device='cuda:0')

→ 성능수렴도가 높은 0.1의 dropout을 쓸 것이냐, 성능 수렴도가 낮지만 기대치가 높은 0.4를 쓸 것이냐를 선택해야한다. 0.4가 잠재치가 높아서 0.4를 사용했다.

--- 새로운 모델에 맞는 batch size 탐구

: 95 valid 넘는 성능 수렴도 찾기

- batch size 1024

32% | 13/40

Epoch[13/40], Train Loss:3.5337, Train Acc:0.97

1.391174964606762 tensor(0.9538, device='cuda:0')


- batch size 1024 -> 768

25% | 10/40

Epoch[10/40], Train Loss:7.4144, Train Acc:0.96


2.0357719734311104 tensor(0.9500, device='cuda:0')

- batch size 1024 -> 1536

40% | 16/40 Train Loss:2.2404, Train Acc:0.97

0.9743789583444595 tensor(0.9540, device='cuda:0')

- batch size 1252

32% | Epoch[13/40], Train Loss:4.3316, Train Acc:0.96


1.2529783993959427 tensor(0.9513, device='cuda:0')

- batch size 512

20% | Epoch[8/40], Train Loss:12.1809, Train Acc:0.95

3.0693433713167906 tensor(0.9533, device='cuda:0')

- batch size 384

22% | Epoch[10/40], Train Loss:16.9864, Train Acc:0.95

3.9040552750229836 tensor(0.9513, device='cuda:0')

- batch size 256

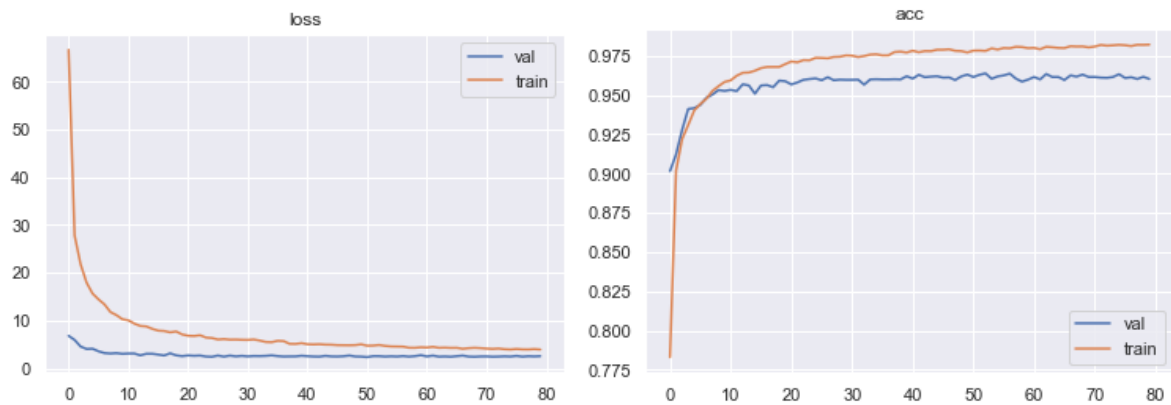
Epoch[8/40], Train Loss:23.8571, Train Acc:0.95

6.172730512917042 tensor(0.9515, device='cuda:0')

→ 이전 모델과 달리 batch size가 512에서 잘 작동한다는 것을 알 수 있다. 8 epoch로 성능 수렴이 제일 빠르고, loss 또한 256으로 했을 때보다 낮아서 이것으로 채택했다. 대신 주의할 점은 augmentation randomness에 따라 성능과 속도가 variance가 있을 수 있다.

: training loss가 5보다 낮아지는 경우 대부분 valid의 성능이 더 올라가지 않고 유지 혹은 떨어지는 경향을 보였다.

## 5 토의 및 결과



- 해당 데이터셋에 가장 성능 수렴이 빠르고 좋은 성능을 학습할 수 있는 batch\_size는 512이다. 그리고 그에 맞는 learning rate를 탐구한 결과 0.002가 적합함을 알 수 있었다. Dropout(0.4)를 활용했다. 모델은 경량화 모델2가 가장 성능이 괜찮았다.

- data\_loader를 test 해보았을 때, pin\_memory를 쓰지 않고 num\_workers=2로 했을 때 가장 빠른 속도가 나오는 것을 확인할 수 있었다.

- 모델이 복잡할수록 학습해야 할 파라미터가 많아져서, 성능 수렴이 빠르게 안된다. 따라서 본 과제에서는 경량화된 resnet을 만들어서 사용했다. Valid accuracy를 올리기 위해 각종 규제 기법과 augmentation을 넣어서 시간을 잡아먹지 않도록 하였고, epoch마다 lr을 조절해주는 것보다 adam의 lr 적응력이 훨씬 효율적이었다.

- 마지막으로 github의 코드를 보니, dataset의 valid, train을 엄격하게 구분하는 것으로 보였다. 그래서 성능 측정 시에 변화량을 보고 마지막엔 train과 valid를 합쳐서 학습시켜주는 것이 일반적인데, 해당 task에서는 합쳐서 학습하지 않았다. **다시 말하자면 train을 한 모델은 valid의 데이터셋을 주입하지 않았다는 의미이다.**

## 5. 참고 문헌

- Xavier initialization

chrome-

extension://efaidnbmnnnibpcajpcglclefindmkaj/viewer.html?pdfurl=https%3A%2F%2Fproceedings.mlr.press%2Fv9%2Fglorot10a%2Fglorot10a.pdf&clen=1647622&chunk=true

- ResNet

<https://arxiv.org/abs/1512.03385>

- BatchNorm

<https://arxiv.org/abs/1502.03167>

- Imagenet CNN

chrome-

extension://efaidnbmnnnibpcajpcgltclfindmkaj/viewer.html?pdfurl=https%3A%2F%2Fproceedings.n  
eurips.cc%2Fpaper%2F2012%2Ffile%2Fc399862d3b9d6b76c8436e924a68c45b-  
Paper.pdf&cflen=1418820

- Torch 공식문서

- 인공지능개론 실습 코드