Computer Vision Project3

2nd semester of 2022 Yonsei University Computer Vision

Self Introduction

Affiliation – 연세대학교 컴퓨터과학과

Name – 김신현

ID – 2016147588

Email – ksh415622@naver.com

1 INTRODUCTION

본 보고서는 음식 이미지를 분류하는 뉴런 네트워크를 구현하고 데이터 preprocessing 및 네트워크 아키텍처에 대한 설명으로 구성되어 있다.

2 ENVIRONMENT

Language:	Python	**CPU:**	Intel i5 12400
OS:	Window10	**Editor:**	jupyter notebook & VsCode (Local)
GPU:	RTX 3060ti (8GB)	**RAM :**	32 GB

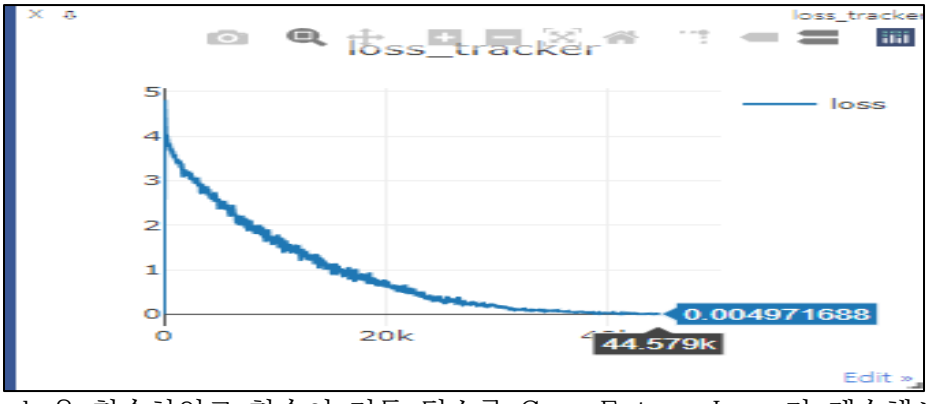
3 RESULT (RMS)

3.1 Leaderboard

23	kimsh415		0.71428	10	4h

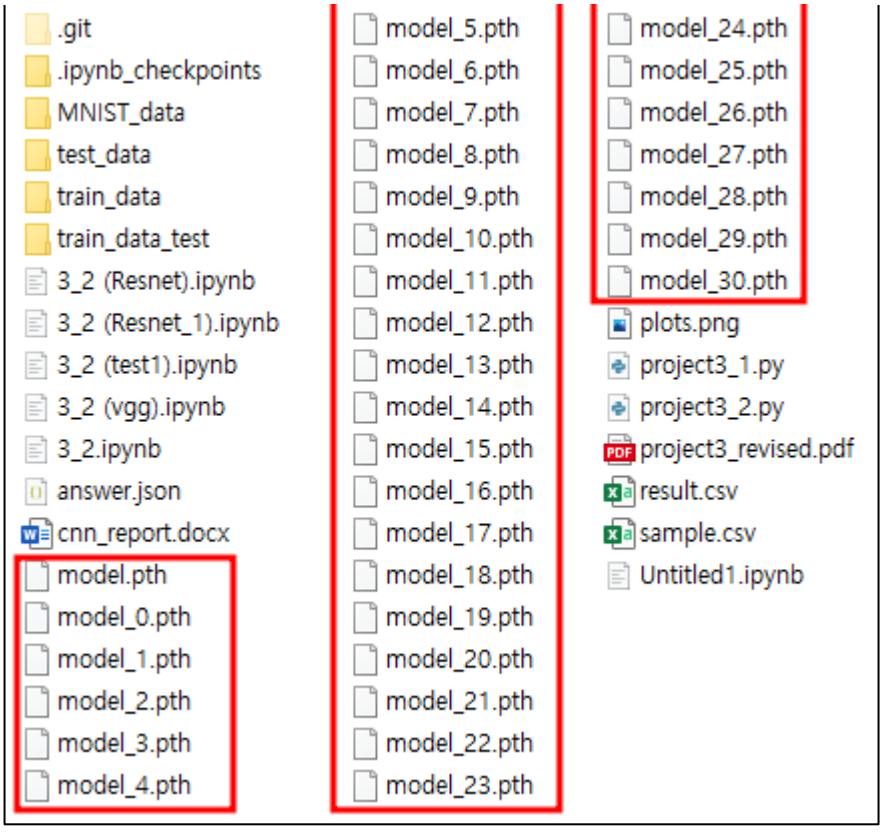
- 주어진 testcase 에는 0.71428 의 결과가 나왔다.

3.2 Train Loss



- 총 30 epoch 을 학습하였고 학습이 거듭 될수록 CrossEntropyLoss 가 계속해서 줄어드는 것을

확인할 수 있었다

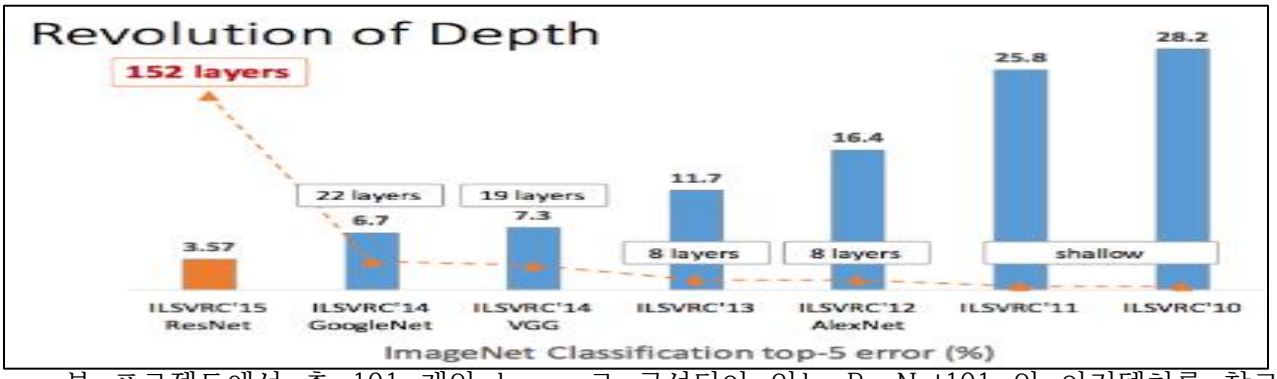


- epoch 이 거듭될수록 Overfitting 이 될 수 있다 판단하여 위 사진과 같이 각 epoch 이 끝날 때마다 파라미터를 저장하였다

- 20~30 epoch 의 총 10 개의 파라미터들을 load 하여 test 데이터를 입력해본 결과 25 번 째 epoch 을 거친 후 학습 된 모델이 약 0.71428 의 accuracy 로 가장 높은 정답율을 보였다.

4 IMPLEMENTATION

4.1 ResNet101:



- 본 프로젝트에선 총 101 개의 layer 로 구성되어 있는 ResNet101 의 아키텍처를 참고하여 구현하였다.

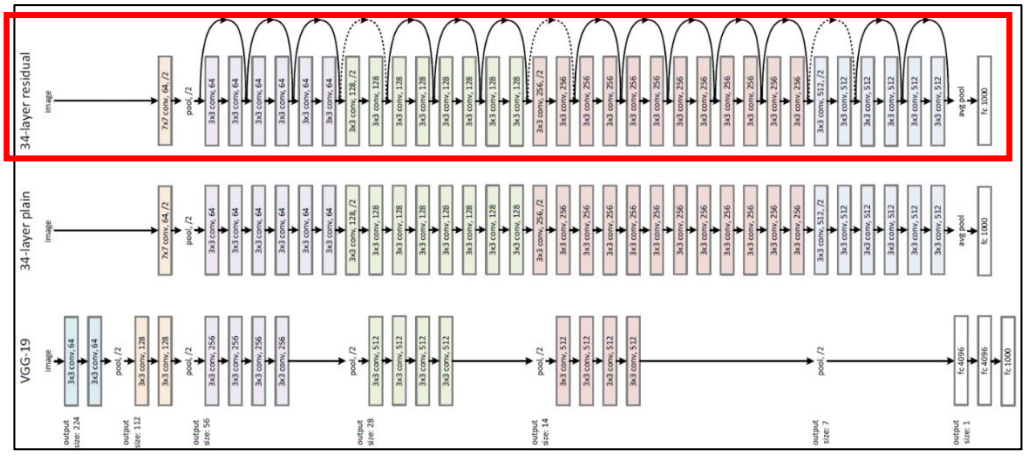


Fig. 1. ResNet Architecture 1

- ResNet 은 Residual Connection(identical connection)기법을 활용하여 short cut 을 만들어 layer 가 매우 깊어 졌을 때도 빠르게 학습하여 파라미터를 수렴시킬 수 있다.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer	
conv1	112×112	7×7, 64, stride 2					-> padding=3
		3×3 max pool, stride 2					-> padding=1
conv2_x	56×56	[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	-> stride=1
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8	-> stride=2
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36	-> stride=2
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	-> stride=2
	1×1	average pool, 1000-d fc, softmax					-> AdaptiveAvePool2d(1,1)
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9	

Fig. 2. ResNet Architecture 2

- 위 사진과 같이 본인이 사용한 ResNet101 의 구조는 빨간 박스 영역과 같이 구현 되어있다.

- ResNet101 은 위 그림과 같이 각각 다른 모양의 블록들이 3,4,23,3 개씩 있다, 자세한 구현 내용은 아래에서 다룬다.

4.2 ResNet Implementation

- block & MyModel 총 두개의 클래스로 모델을 정의하였다

4.2.1 MyModel Class

- block 의 구현은 Fig.2 의 설명에 따라 구현하였다.

```python
class MyModel(nn.Module):
    def __init__(self, block, layers, image_channels, num_classes):
        super(MyModel, self).__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(image_channels, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.layer1 = self.build_layer(
            block, layers[0], out_channel=64, stride=1
        )
        self.layer2 = self.build_layer(
            block, layers[1], out_channel=128, stride=2
        )
        self.layer3 = self.build_layer(
            block, layers[2], out_channel=256, stride=2
        )
        self.layer4 = self.build_layer(
            block, layers[3], out_channel=512, stride=2
        )

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * 4, num_classes)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.reshape(x.shape[0], -1)
        x = self.fc(x)
```

		7×7, 64, stride 2	-> padding=3
		3×3 max pool, stride 2	-> padding=1
	1×1	average pool, 1000-d fc, softmax	-> AdaptiveAvePool2d(1,1)

- **self.conv1:** Fig.2 의 7*7, 64, stride 2, padding 3 의 convolution 커널을 의미.
- **self.maxpool:** Fig.2 의 3*3, stride 2, padding 1 의 maxpooling 커널을 의미
- **self.avgpool & fc:** Fig.2 의 80-dimension avgpool & fully connected 커널을 의미.

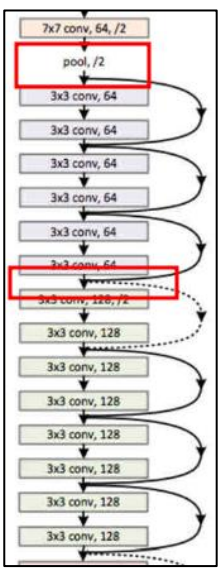
```python
120     def build_layer(self, block, num_residual_blocks, out_channel, stride):
121         identity_downsample = None
122         layers = []
123 
124         if stride != 1 or self.in_channels != out_channel * 4:
125             identity_downsample = nn.Sequential(
126                 nn.Conv2d(
127                     self.in_channels,
128                     out_channel * 4,
129                     kernel_size=1,
130                     stride=stride,
131                     bias=False
132                 ),
133                 nn.BatchNorm2d(out_channel * 4),
134             )
135 
136         layers.append(
137             block(self.in_channels, out_channel, identity_downsample, stride)
138         )
139 
140         self.in_channels = out_channel * 4
141 
142         for i in range(num_residual_blocks - 1):
143             layers.append(block(self.in_channels, out_channel))
144 
145         return nn.Sequential(*layers)
```

- **build_layer**: redisual block 을 만드는 함수

- **line 124**: stride 가 1 이 아니고 ouput channel 이 input 의 4 배가*(아래 block class 에서 self.expansion = 4)* 아닐 때, downsampling 을 한다.

  downsmapling 이란 pooling, convolution layer 를 통해 featuremap 이 축소되어 잔차연결할 때 크기가 맞지 않아 연산을 할 수 없다.

  따라서 1*1 convolution(bottle neck) layer 를 하나 더 연결해준다.



- **line 136**: 첫 블록은 위에서 말한 downsampling 여부를 판단하고 한 블록을 추가해준다,

  다음으로 남아있는 블록 수 만큼 모두 stride 를 1 로, downsampling 없이 쌓는다.

4.2.2 BlockClass

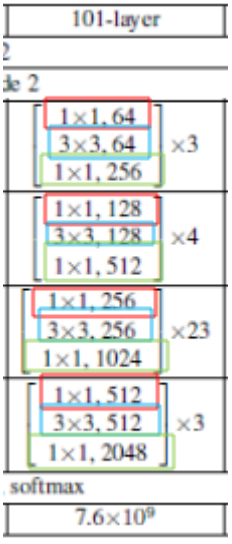
```python
class block(nn.Module):
    def __init__(
        self, in_channels, out_channel, identity_downsample=None, stride=1):
        super(block, self).__init__()
        self.expansion = 4
        self.conv1 = nn.Conv2d(
            in_channels, out_channel, kernel_size=1, stride=1, padding=0, bias=False
        )
        self.bn1 = nn.BatchNorm2d(out_channel)
        self.conv2 = nn.Conv2d(
            out_channel,
            out_channel,
            kernel_size=3,
            stride=stride,
            padding=1,
            bias=False
        )
        self.bn2 = nn.BatchNorm2d(out_channel)
        self.conv3 = nn.Conv2d(
            out_channel,
            out_channel * self.expansion,
            kernel_size=1,
            stride=1,
            padding=0,
            bias=False
        )
        self.bn3 = nn.BatchNorm2d(out_channel * self.expansion)
        self.relu = nn.ReLU()
        self.identity_downsample = identity_downsample
        self.stride = stride

    def forward(self, x):
        identity = x.clone()

        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu(x)
        x = self.conv3(x)
        x = self.bn3(x)

        if self.identity_downsample is not None:
            identity = self.identity_downsample(identity)

        x += identity
        x = self.relu(x)
        return x
```

- self.conv1: residual block 내의 모든 1*1*out_channel 사이즈에서 첫 번째 convolution laye 를 의미 (오른쪽 이미지 빨간색 블록)

- self.conv2: residual block 내의 모든 3*3*out_channel conv layer 의미(오른쪽 이미지 파란색 블록)

- self.conv3: residual block 내의 모든 1*1*out_channel 사이즈에서 세 번째 convolution laye 를 의미 (오른쪽 이미지 녹색 블록)

- line 74: residual connection 부분.



4.3 Data Set (MyDataset):

```
class MyDataset(Dataset) :
    def __init__(self,meta_path,root_dir,transform=None,pre_transform=None, mode="train") :
```

- MyDataset 클래스로 총 5 개의 파라미터를 받는다

  a. **meta_path**: answer.json

  b. **root_dir**: train or test set directory

  c. **pre_transform**: dataset 에 image 데이터를 저장하기 전 전처리용 transform

  d. **transform**: 이미지 학습 시 사용될 전처리용 transform

  e: **mode**: train 용으로 사용될 경우 answer.json 파일을 읽어야 하기 때문에 구분용으로 사용되는 파라미터, 총 “train”,”test” 두가지 값을 받는다.

```
158     if mode == "train":
159         self.mode = 'train'
160         with open(meta_path, 'r') as file:
161             temp_meta_data = json.load(file)
162         meta = pd.json_normalize(temp_meta_data['annotations'])
163         meta['file_name'] = meta['file_name'].apply(parse_file_number)
164         meta = meta.sort_values("file_name").reset_index(drop=True)
165 
166         self.root_dir = root_dir
167 
168         meta['file_name'] = meta['file_name'].map(lambda x :  self.root_dir + '/' + str(x) +'.jpg')
169         self.X = []
170         loop = tqdm(list(meta['file_name']), total=len(meta['file_name']), leave=True)
171 
172         for i, X in enumerate(loop):
173             try:
174                 self.X.append(pre_transform(Image.open(X).convert("RGB")))
175             except:
176                 pass
177         self.y = meta['category']
178         self.transform = transform
```

- 위 코드는 train 의 경우 dataset 을 빌드하는 과정이다.

- line 162: answer.json 을 읽어 dataframe 으로 변환시킨다

- line 172: 학습 속도 향상을 위해 데이터를 학습할 때 이미지를 읽어오는 것이 아닌 dataset 생성 시 모든 학습용 이미지 데이터들을 RAM 에 올린다.

```
180         elif mode == "test":
181             self.mode = 'test'
182             meta = pd.DataFrame()
183             meta['file_name'] = glob.glob(root_dir+"/*.jpg")
184             meta = meta.sort_values("file_name").reset_index(drop=True)
185             self.root_dir = root_dir
186             self.X = []
187             loop = tqdm(list(meta['file_name']), total=len(meta['file_name']), leave=True)
188             for i, X in enumerate(loop):
189     #                 print(i)
190                 try:
191                     self.X.append(pre_transform(Image.open(X).convert("RGB")))
192                 except:
193                     pass
194             self.y = meta['file_name'].apply(parse_file_number_test)
195             self.transform = transform
```

- 위 코드는 test 의 경우 dataset 을 빌드하는 과정이다.

- line 188: train 의 경우와 동일하게 dataset 을 빌드하는 과정에서 미리 이미지 데이터를 모두 RAM 에 올린다.

```
197     def __len__(self) :
198         return len(self.X)
199 
200     def __getitem__(self,idx) :
201         if  self.mode == 'train':
202             X, y = self.transform(self.X[idx]), int(self.y[idx])
203             return X, torch.tensor(y)
204         elif self.mode == 'test':
205             X ,y = self.transform(self.X[idx]), self.y[idx]
206             return X ,y
```

- line 197: 데이터셋 길이는 X 의 길이, 즉 이미지 픽셀 데이터를 기준으로 구현하였다

예) train 의 경우 len(train_dataset) = 40000

- line 203&206: train 와 test 에서 return 하는 X 는 이미지 픽셀 데이터로 동일하지만 y 의 경우 Train 은 카테고리를 반환하고 test 의 경우는 이미지 파일 명을 반환한다.

4.4 train

```python
def train() :
    #======================make dataset=====================
    batch = 20
    mode = 'train'
    train_data_dir = "./train_data"
    meta_path = "./answer.json"

    pre_transformer = transforms.Compose([
        transforms.Resize((400,400)),
        transforms.CenterCrop((224,224)),
    ])

    transformer = transforms.Compose([
            transforms.RandomHorizontalFlip(p=0.5),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
        ])
    train_data = MyDataset(meta_path, train_data_dir, transform=transformer,pre_transform=pre_transformer, mode="train")

    train_loader = DataLoader(
        train_data, batch_size=batch)

    #======================init RestNet101=====================
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model = MyModel(block, [3,4,23,3], 3, 80).to(device)

    #======================make tracker=====================
    # tracker = visdom.Visdom()
    # tracker.close(env="main")
    # loss_plt = tracker.line(Y=torch.Tensor(1).zero_(),opts=dict(title='loss_tracker', legend=['loss'], showlegend=True))

    #======================start training=====================
    criterion = nn.CrossEntropyLoss().to(device)
    optimizer = torch.optim.SGD(model.parameters(), lr = 0.003,momentum=0.9)
    lr_sche = torch.optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.9)
    epochs = 100
    # epochs = 1
    for epoch in range(epochs):  # loop over the dataset multiple times
        running_loss = 0.0
        lr_sche.step()
        loop = tqdm(train_loader, total=len(train_loader), leave=True)
        for i, (inputs, labels) in enumerate(loop):
            inputs = inputs.to(device)
            labels = labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward & backward & optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
    #         lr_sche.step()

            # print statistics
            running_loss += loss.item()
            if i % 30 == 29:    # print every 30 mini-batches
                # loss_tracker(loss_plt, torch.Tensor([running_loss/30]), torch.Tensor([i + epoch*len(train_loader) ]))
                loop.desc = "valid epoch[{}/{}], Loss= {}".format(epoch + 1, epochs, running_loss/30)
                running_loss = 0.0
        torch.save(model.state_dict(), "./model_"+ str(epoch) + ".pth")
    print('Training Finished')
```

- **pre_transformer:** 4.3 에서 설명한 내용으로 dataset 을 구성할 때 먼저 이미지마다 사이즈가 다르기 때문에 400, 400 의 크기로 통일한 후 중앙에서 224*224 만큼의 이미지만 사용한다

- **transformer:** 해당 transformer 는 이미지를 dataset 에서 불러와 학습시킬 때 사용되는 것으로 정확도를 올리기 위해 좌우 반전 및 정규화 두가지를 적용하였다.

- line 242: 효율적인 학습을 위해 경사가 최적화될수록 learning rate 를 줄여나가 미세하게 조정해주는 lr_scheduler 를 사용하였다.

- line 235~237 & line 266:

먼저 다른 콘솔창에서 visdom 을 입력하여 로컬 서버를 실행시킨 뒤 해당 부분들의 주석을 풀고 실행시키면 loss 를 tracking 할 수 있다.

5 APPROACH

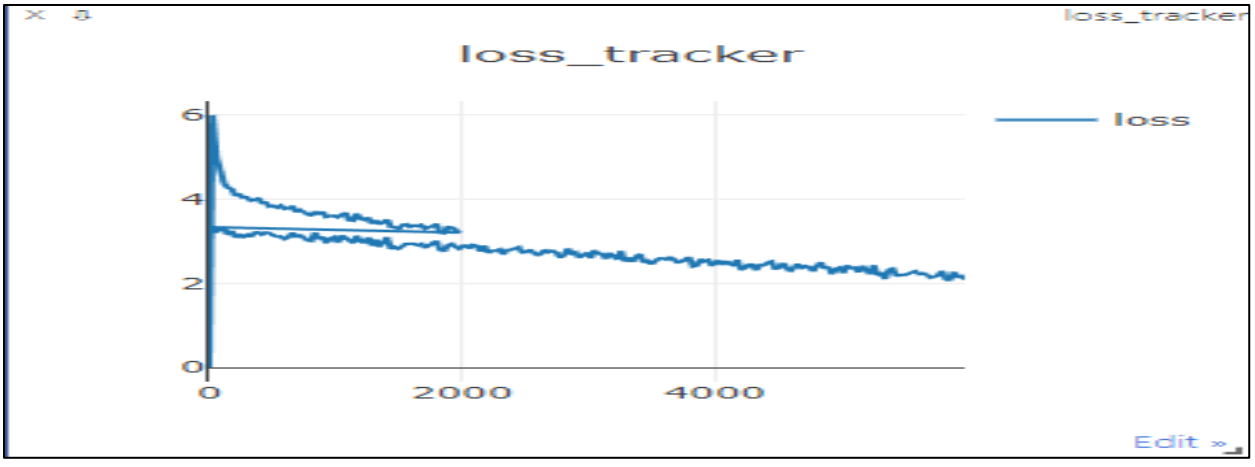
5.1 Why Choose ResNet101

- ResNet 은 18, 34, 50 ,101, 152 총 5 가지의 hidden layer 수로 구성된 버전이 있다

- 본 프로젝트에서는 40000 개의 training data 가 있으므로 152 개 layer 로는 학습할 데이터가 조금 부족하다 생각되었다.

- Memory 및 Computing Power 가 부족하고 또한 총 80 개의 class 밖에 없기 때문에 152 보다는 작은 101 개의 layer 로 구성된 ResNet101 을 사용하였다.

5.2 How to Choose Batch Size



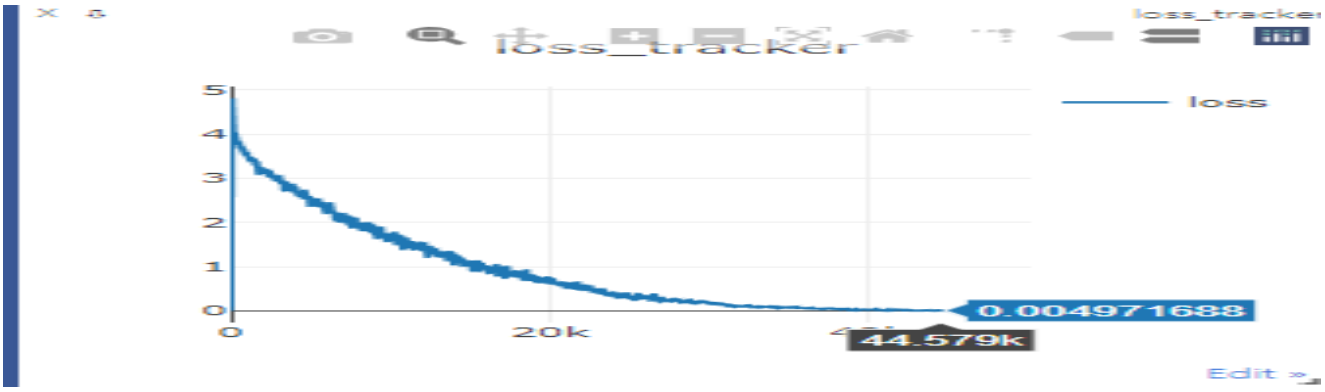
- 위 이미지는 batch size 5 로 하였을 때의 loss tracking 이미지이다
- 각 30 batch 마다 loss 를 기록하도록 설정하였다.
- 그래프를 보면 알 수 있듯이 노이즈가 조금 많다고 판단되어 batch size 를 20 으로 늘렸다
- 또한 batch size 가 20 이상일 경우 **2 Enviroment** 에서 기재한 본인이 사용하는 환경에서는 GPU 메모리가 부족하기 때문에 사용가능한 최대 batch size 인 20 으로 학습하였다.

5.3 How to Choose Learning rate

- 초기 learing rate 를 0.001 보다 조금 더 큰 0.003 으로 설정했다

- 이후 lr_scheduler 를 사용하여 초기에는 학습을 조금 더 빠르게 하고 이후 minimum 에 가까울 질수록 learning rate 가 줄어들어 더욱 세밀한 minimum 을 찾도록 하였다.

5.4 How to Avoid Overfitting

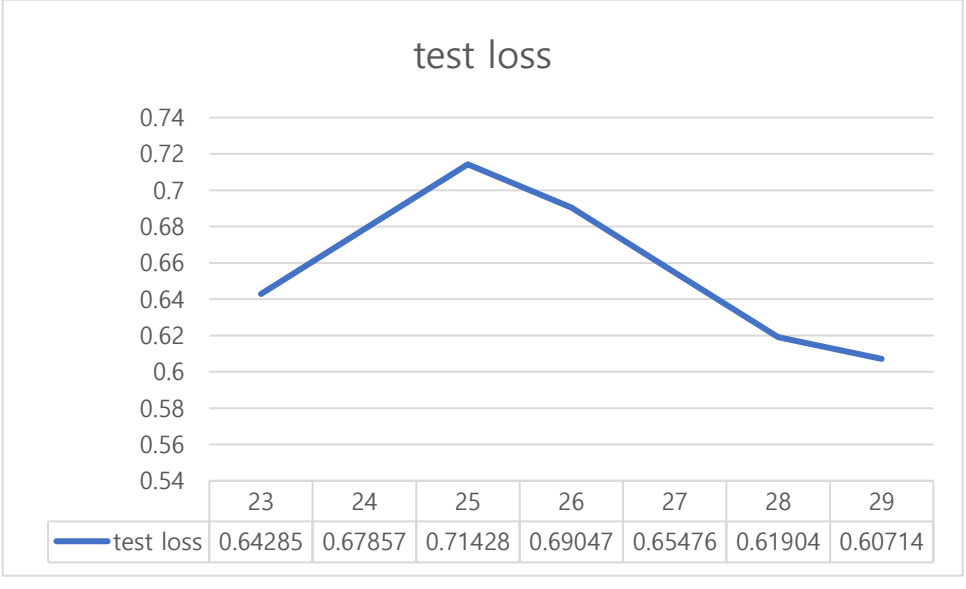


- 위 그림은 총 30epoch 을 학습하였을 때 loss tracking 한 결과이다.

- 학습이 진행될수록 Loss 가 매우 작아지는 것을 확인할 수 있다

- 때문에 더 많은 epoch 을 학습했을 때 Overfitting 이 더욱 심해질 것 같아 30 에서 학습을 멈췄다

- 이후 train()에서 확인할 수 있듯이 각 epoch 마다 모델을 저장하였기 때문에 23~29 epoch 의 학습을 거친 모델을 load 하여 test set 의 class 들을 각각 예측한 후 submit 하여 결과를 비교해봤다.



	23	24	25	26	27	28	29
test loss	0.64285	0.67857	0.71428	0.69047	0.65476	0.61904	0.60714

- 위 그래프는 test loss 를 비교해본 결과로 25 epoch 을 학습했을 때의 모델이 가장 좋은 성능을 보였기 때문에 이를 체택하였다.

- 또한 transformer 를 정의할 때 50%의 확률로 좌우 반전시켜 학습시켜 모델이 더욱 견고하도록 설계하였다.

6 How to Run

6.1 How to Test

```
340  def main() :
341      # train()
342      test()
```

- 코드 수정 없이 python project3_2.py로 실행하면 된다

6.2 How to Train

```
340  def main() :
341      train()
342      # test()
```

- main 함수에서 train() 부분의 주석을 해제한 후 python project3_2.py로 실행하면 된다

References

[1] "ResNet 클래스 정의하기" https://ingu627.github.io/code/ResNet_scratch_pytorch/
[2] "ResNet Architecture" https://bskyvision.com/644
[3] "Pytorch Documentation" https://pytorch.org/docs/stable/index.html
[4] "Pytorch ResNet" https://www.youtube.com/watch?v=DkNIBBBvcPs