```
# Python 버전 확인
!python --version

# 설치된 패키지 목록을 requirements.txt로 저장
!pip list --format=freeze > requirements.txt

# requirements.txt 내용 확인
!cat requirements.txt
```

```
Python 3.10.12
absl-py==1.4.0
accelerate==0.34.2
aiohappyeyeballs==2.4.3
aiohttp==3.10.10
aiosignal==1.3.1
alabaster==0.7.16
albucore==0.0.19
albumentations==1.4.20
altair==4.2.2
annotated-types==0.7.0
anyio==3.7.1
argon2-cffi==23.1.0
argon2-cffi-bindings==21.2.0
array_record==0.5.1
arviz==0.20.0
astropy==6.1.4
astropy-iers-data==0.2024.10.21.0.33.21
astunparse==1.6.3
async-timeout==4.0.3
atpublic==4.1.0
attrs==24.2.0
audioread==3.0.1
autograd==1.7.0
babel==2.16.0
backcall==0.2.0
beautifulsoup4==4.12.3
bigframes==1.24.0
bigquery-magics==0.4.0
bleach==6.1.0
blinker==1.4
blis==0.7.11
blosc2==2.0.0
bokeh==3.4.3
Bottleneck==1.4.2
bqplot==0.12.43
branca==0.8.0
CacheControl==0.14.0
cachetools==5.5.0
catalogue==2.0.10
certifi==2024.8.30
cffi==1.17.1
chardet==5.2.0
charset-normalizer==3.4.0
chex==0.1.87
clarabel==0.9.0
click==8.1.7
cloudpathlib==0.20.0
cloudpickle==3.1.0
cmake==3.30.5
cmdstanpy==1.2.4
colorcet==3.1.0
colorlover==0.3.0
colour==0.1.5
community==1.0.0b1
confection==0.1.5
cons==0.4.6
contourpy==1.3.0
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
import torch
print(f"GPU 사용 가능 여부: {torch.cuda.is_available()}")
```

```
GPU 사용 가능 여부: True
```

```
#이미지 데이터 수 확인
import os
# 이미지 데이터 폴더 경로 설정
training_image_path = "/content/drive/MyDrive/final/dataset/training_image"
validation_image_path = "/content/drive/MyDrive/final/dataset/validation_image"
```

```
# 폴더 내 파일 목록 확인 + 숫자 확인
training_image_num = len(os.listdir(training_image_path)) #os.listdir : 폴더 내의 모든 파일과 디렉토리를 리스트로 반환
validation_image_num = len(os.listdir(validation_image_path))

print(f"Number of training images: {training_image_num}")
print(f"Number of validation images: {validation_image_num}")
```

```
Number of training images: 4070
Number of validation images: 951
```

```
코딩을 시작하거나 AI로 코드를 생성하세요.
```

1-1

```
import os
import pandas as pd
from collections import defaultdict
```

```
# 파일명을 분석하여 성별 & 스타일 정보를 추출하는 함수
def extract_gender_style_info(folder_path):
    # 성별과 스타일별 이미지 수를 저장할 딕셔너리
    gender_style_count = defaultdict(int)

    # 폴더 내 모든 파일명 가져오기
    for file_name in os.listdir(folder_path):
        if file_name.endswith('.jpg'):  # jpg 파일만 처리
            # 파일명 분해: 예시 - "W_96469_60_minimal_W.jpg"
            parts = file_name.split('_')
            if len(parts) >= 5:
                gender = '여성' if parts[-1][0] == 'W' else '남성'  # 성별 추출
                style = parts[-2]  # 스타일 추출
                # 성별 & 스타일 기준으로 이미지 수 카운트
                gender_style_count[(gender, style)] += 1

    return gender_style_count

# 각 데이터 폴더에서 성별 & 스타일별 통계 정보 추출
training_stats = extract_gender_style_info(training_image_path)
validation_stats = extract_gender_style_info(validation_image_path)

# 통계 정보를 데이터프레임으로 변환
def convert_to_dataframe(stats_dict):
    # 딕셔너리를 데이터프레임으로 변환 후, '성별', '스타일', '이미지 수'로 분리
    df = pd.DataFrame(list(stats_dict.items()), columns=['성별 & 스타일', '이미지 수'])
    df[['성별', '스타일']] = pd.DataFrame(df['성별 & 스타일'].tolist(), index=df.index)
    df.drop(columns='성별 & 스타일', inplace=True)

    # 컬럼 순서를 성별, 스타일, 이미지 수 순으로 변경
    df = df[['성별', '스타일', '이미지 수']]

    # 성별, 스타일 기준으로 정렬 (여성 -> 남성, a -> z)
    df = df.sort_values(by=['성별', '스타일'], ascending=[False, True])

    # 인덱스 초기화하여 순서 번호 재정렬
    df.reset_index(drop=True, inplace=True)
    return df
```

```
# Training 데이터 통계표
training_df = convert_to_dataframe(training_stats)
print("Training 데이터 통계표:")
print(training_df)

# Validation 데이터 통계표
validation_df = convert_to_dataframe(validation_stats)
print("WnValidation 데이터 통계표:")
print(validation_df)
```

```
Training 데이터 통계표:
    성별          스타일  이미지 수
0   여성     athleisure     67
1   여성   bodyconscious     95
2   여성       cityglam     67
3   여성        classic     77
4   여성          disco     37
5   여성        ecology     64
6   여성       feminine    154
7   여성     genderless     77
8   여성          grunge     31
```

9	여성	hiphop	48
10	여성	hippie	91
11	여성	kitsch	91
12	여성	lingerie	55
13	여성	lounge	45
14	여성	military	33
15	여성	minimal	139
16	여성	normcore	153
17	여성	oriental	78
18	여성	popart	41
19	여성	powersuit	120
20	여성	punk	65
21	여성	space	37
22	여성	sportivecasual	157
23	남성	bold	268
24	남성	hiphop	274
25	남성	hippie	260
26	남성	ivy	237
27	남성	metrosexual	278
28	남성	mods	269
29	남성	normcore	364
30	남성	sportivecasual	298

Validation 데이터 통계표:

	성별	스타일	이미지 수
0	여성	athleisure	14
1	여성	bodyconscious	23
2	여성	cityglam	18
3	여성	classic	22
4	여성	disco	10
5	여성	ecology	17
6	여성	feminine	44
7	여성	genderless	12
8	여성	grunge	10
9	여성	hiphop	8
10	여성	hippie	14
11	여성	kitsch	22
12	여성	lingerie	5
13	여성	lounge	8
14	여성	military	9
15	여성	minimal	35
16	여성	normcore	20
17	여성	oriental	18
18	여성	popart	8
19	여성	powersuit	34
20	여성	punk	12

1-2

```
# GPU 사용 여부 확인
print(f"GPU 사용 여부: {torch.cuda.is_available()}")

# 현재 할당된 메모리 (bytes 단위)
allocated_memory = torch.cuda.memory_allocated()
print(f"현재 GPU에 할당된 메모리: {allocated_memory / (1024 ** 2)} MB")

# 현재 캐시된 메모리 (PyTorch는 메모리를 캐시로 잡아두기 때문에 실제 사용량과 차이가 있을 수 있음)
cached_memory = torch.cuda.memory_reserved()
print(f"현재 GPU에 캐시된 메모리: {cached_memory / (1024 ** 2)} MB")
```

```
GPU 사용 여부: True
현재 GPU에 할당된 메모리: 42.74755859375 MB
현재 GPU에 캐시된 메모리: 64.0 MB
```

```
import os
import re
import torch
from torchvision import transforms
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.models as models
import matplotlib.pyplot as plt
import matplotlib.font_manager as fm
from PIL import Image
import numpy as np
```

```
# 파일명에서 성별 & 스타일 정보를 추출하는 함수
def get_label_from_filename(filename):
    """
    파일명에서 성별과 스타일 정보를 정규식을 통해 추출
```

```
    파일명 예시: W_65122_10_sportivecasual_W.jpg
    """
    # 정규식 패턴: (W|T)_숫자_숫자_스타일_(W|M).jpg
    pattern = r"^(W|T)_(\d+)_(\d+)_(\w+)_(W|M)\W.jpg$"

    # 정규식으로 파일명 분석
    match = re.match(pattern, filename)

    if match:
        style = match.group(4)  # 네 번째 그룹이 스타일
        gender_code = match.group(5)  # 다섯 번째 그룹이 성별 ('W' 또는 'M')
        gender = '여성' if gender_code == 'W' else '남성'
        label = f"{gender}_{style}"  # 성별과 스타일을 결합한 라벨 생성
        return label
    else:
        return None  # 매칭되지 않으면 None 반환

# 학습 및 검증 데이터를 위한 클래스 정의 함수
def define_classes_from_images(image_folder, label_to_idx=None, current_idx=0):
    """
    이미지 폴더에서 성별 & 스타일 정보를 추출하여 클래스 이름을 정의
    고유한 성별 & 스타일 조합을 인덱스로 매핑
    중복되는 클래스는 동일한 인덱스를 사용

    label_to_idx: 기존 클래스 사전 (None이면 새로 생성)
    current_idx: 기존 클래스 번호에서 이어서 번호를 부여
    """
    if label_to_idx is None:
        label_to_idx = {}

    # scandir() 사용: 폴더에서 파일을 순차적으로 처리
    with os.scandir(image_folder) as entries:
        for entry in entries:
            if entry.is_file() and entry.name.endswith(".jpg"):  # 이미지 파일만 처리
                label_key = get_label_from_filename(entry.name)
                print(f"Processing file: {entry.name}, Label Key: {label_key}")

                # 이미 정의된 클래스는 동일한 인덱스를 사용
                if label_key not in label_to_idx:
                    label_to_idx[label_key] = current_idx
                    current_idx += 1  # 새로운 라벨에 대해 인덱스를 1 증가

    return label_to_idx, current_idx
```

```
# 학습 데이터에서 성별 & 스타일로 클래스 이름 정의
training_image_path = "/content/drive/MyDrive/final/dataset/training_image"
training_label_to_idx, current_idx = define_classes_from_images(training_image_path)
```

숨겨진 출력 표시

```
# 검증 데이터에서 성별 & 스타일로 클래스 이름 정의 (중복 클래스는 동일 인덱스 사용)
validation_image_path = "/content/drive/MyDrive/final/dataset/validation_image"
validation_label_to_idx, _ = define_classes_from_images(validation_image_path,
                                                        label_to_idx=training_label_to_idx,
                                                        current_idx=current_idx)
```

숨겨진 출력 표시

```
# 사용자 정의 데이터셋 클래스
class CustomDataset(Dataset):
    def __init__(self, image_folder, label_to_idx, transform=None):
        self.image_folder = image_folder
        self.transform = transform
        self.image_filenames = [f for f in os.listdir(image_folder) if f.endswith('.jpg')]
        self.labels = [get_label_from_filename(f) for f in self.image_filenames]
        self.label_to_idx = label_to_idx

    def __len__(self):
        return len(self.image_filenames)

    def __getitem__(self, idx):
        img_path = os.path.join(self.image_folder, self.image_filenames[idx])
        image = Image.open(img_path).convert('RGB')
        label = self.labels[idx]
        label_idx = self.label_to_idx[label]

        if self.transform:
            image = self.transform(image)

        return image, label_idx
```

1. 이미지 전처리 및 증강

```
# 이미지 전처리(파이프라인 포함X) : object detcetion

"""
!pip install rembg
from rembg import remove
import os
import cv2
import torchvision.transforms.functional as F


training_image_path = "/content/drive/MyDrive/dataset/training_image"
output_folder = "/content/drive/MyDrive/dataset/no_bg_training_image"

# 전체 이미지 수 계산
total_images = len(os.listdir(training_image_path))

# 배경 제거 후 이미지 저장
for idx, file in enumerate(os.listdir(training_image_path), start=1):
    file_path = os.path.join(training_image_path, file)

    img = Image.open(file_path)  # 이미지 열기

    # 이미지를 바이너리 데이터로 변환
    img_byte_arr = io.BytesIO()
    img.save(img_byte_arr, format='PNG')  # 이미지를 PNG 형식으로 저장
    img_byte_arr = img_byte_arr.getvalue()  # 바이너리 데이터로 변환

    # 배경 제거
    img_no_bg = remove(img_byte_arr)  # rembg 모듈을 사용해 배경 제거
    img_no_bg = Image.open(io.BytesIO(img_no_bg))  # PIL 이미지로 변환

    # 만약 이미지가 RGBA 모드이면 RGB 모드로 변환
    if img_no_bg.mode == 'RGBA':
        background = Image.new('RGB', img_no_bg.size, (255, 255, 255))  # 흰색 배경 생성
        img_no_bg = Image.alpha_composite(background.convert('RGBA'), img_no_bg).convert('RGB')  # 흰색 배경과 합성

    # 배경 제거된 이미지를 저장할 경로 설정
    save_path = os.path.join(output_folder, file)  # 동일한 파일명으로 저장
    img_no_bg.save(save_path)

    # 진행 중인 이미지 번호 및 총 이미지 수 출력
    print(f"Processed {idx}/{total_images}: {save_path}")


import io

training_image_path = "/content/drive/MyDrive/dcc2024/data/validation_image"
output_folder = "/content/drive/MyDrive/dcc2024/no_bg_validation_image"

# 전체 이미지 수 계산
total_images = len(os.listdir(training_image_path))

# 배경 제거 후 이미지 저장
for idx, file in enumerate(os.listdir(training_image_path), start=1):
    file_path = os.path.join(training_image_path, file)

    img = Image.open(file_path)  # 이미지 열기

    # 이미지를 바이너리 데이터로 변환
    img_byte_arr = io.BytesIO()
    img.save(img_byte_arr, format='PNG')  # 이미지를 PNG 형식으로 저장
    img_byte_arr = img_byte_arr.getvalue()  # 바이너리 데이터로 변환

    # 배경 제거
    img_no_bg = remove(img_byte_arr)  # rembg 모듈을 사용해 배경 제거
    img_no_bg = Image.open(io.BytesIO(img_no_bg))  # PIL 이미지로 변환

    # 만약 이미지가 RGBA 모드이면 RGB 모드로 변환
    if img_no_bg.mode == 'RGBA':
        background = Image.new('RGB', img_no_bg.size, (255, 255, 255))  # 흰색 배경 생성
        img_no_bg = Image.alpha_composite(background.convert('RGBA'), img_no_bg).convert('RGB')  # 흰색 배경과 합성

    # 배경 제거된 이미지를 저장할 경로 설정
    save_path = os.path.join(output_folder, file)  # 동일한 파일명으로 저장
    img_no_bg.save(save_path)

    # 진행 중인 이미지 번호 및 총 이미지 수 출력
```

```
    print(f"Processed {idx}/{total_images}: {save_path}")

"""
```

```
'₩n!pip install rembg₩nfrom rembg import remove₩nimport os₩nimport cv2₩nimport torchvision.transforms.functional as F₩n₩n₩ntraining_image_path = "/content/drive/MyDrive/dataset/training_image"₩noutput_folder = "/content/drive/MyDrive/dataset/no_bg_training_image"₩n₩n# 전체 이미지 수 계산₩ntotal_images = len(os.listdir(training_image_path))₩n₩n# 배경 제거 후 이미지 저장₩nfor idx, file in enumerate(os.listdir(training_image_path), start=1):₩n    file_path = os.path.join(training_image_path, file)₩n₩n    img = Image.open(file_path)  # 이미지 열기₩n₩n    # 이미지를 바이너리 데이터로 변환₩n    img_byte_arr = io.BytesIO()₩n    img.save(img_byte_arr, format='PNG')  # 이미지를 PNG 형식으로 저장₩n    img_byte_arr = img
```

```
no_bg_training_image_path = "/content/drive/MyDrive/final/dataset/no_bg_training_image"
no_bg_validation_image_path = "/content/drive/MyDrive/final/dataset/no_bg_validation_image"
```

```
# 이미지 정규화(Normalization) 및 증강(Data Augmentation)을 위해 Transform 정의(파이프라인 구축)

import torchvision.transforms as transforms
from PIL import Image
import io
import matplotlib.pyplot as plt
import numpy as np


# ResNet 모델용 전처리 파이프라인
train_transform = transforms.Compose([
    transforms.Resize(200),  # 이미지 크기 조정
    transforms.RandomHorizontalFlip(p=0.5), #이미지를 왼쪽에서 오른쪽으로 뒤집는 변형을 무작위로 적용
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1), #이미지의 밝기, 대비, 채도, 색조를 무작위로 조절하여 색상 변형
    transforms.ToTensor(),  # 이미지를 텐서로 변환
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # 정규화
])#이미지의 각 픽셀 값을 일정 범위로 정규화
#ResNet 모델은 일반적으로 ImageNet 데이터셋으로 사전 학습되었으므로, ImageNet의 평균 및 표준편차 값을 사용해 정규화


val_transform = transforms.Compose([
    transforms.Resize(200),            # 이미지를 크기 변경
    transforms.ToTensor(),             # 텐서로 변환
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # 이미지 정규화
])
```

2. 데이터로더 설정

```
# 2. 데이터 로더 설정
# 훈련 데이터 로더 설정
train_dataset = CustomDataset(image_folder=no_bg_training_image_path,
                              label_to_idx=training_label_to_idx,
                              transform=train_transform)
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True, num_workers=0)

# 학습 데이터 로더 설정
val_dataset = CustomDataset(image_folder=no_bg_validation_image_path,
                            label_to_idx=validation_label_to_idx,
                            transform=val_transform)
validation_loader = DataLoader(val_dataset, batch_size=128, shuffle=False, num_workers=0)  # validation_loader로 정의

# 배치 크기는 128 : 한번에 이미지를 모델에 공급
#suffle=Ture : 학습 데이터는 매 epoch마다 섞는다
#num_workers : 데이터를 미리 준비하는 병렬 작업자 수 설정 -> 빠른 업로드 가능
```

3. 모델 정의

3.1 기본 학습 설정

```
#기본 학습 설정
EPOCHS = 80
# GPU 사용 가능 여부 확인 후, 사용 가능하면 GPU 장치 할당
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

3.2 resnet 모델 생성

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
```

```
from torch.optim import Adam
from torch.utils.tensorboard import SummaryWriter

# ResNet 모델의 ResidualBlock 정의
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super(ResidualBlock, self).__init__()

        # 첫 번째 Convolution-BatchNorm-ReLU
        # 입력 채널 수와 출력 채널 수, stride를 사용하여 Convolution Layer를 생성
        # kernel_size=3, padding=1로 설정하여 원본 이미지와 동일한 크기의 출력이 생성
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)  # Batch Normalization 적용
        self.relu = nn.ReLU(inplace=True)        # ReLU 활성화 함수 사용

        # 두 번째 Convolution-BatchNorm
        # 첫 번째 Convolution Layer의 출력 크기와 동일한 채널 수를 사용하여 또 다른 Convolution Layer를 생성
        # kernel_size=3, padding=1로 설정하여 입력과 동일한 크기의 출력이 생성
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)  # Batch Normalization 적용

        #*relu 추가 가능성? 근데 이걸하려면 층을 하나 더 만들어야 함

        # Shortcut connection 설정
        # 만약 stride가 1이 아니거나, 입력과 출력의 채널 수가 다를 경우 차원을 맞춰주는 Conv-BatchNorm 계층을 사용해 입력텐서 변환
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride, bias=False),  # 1x1 Conv Layer로 채널 수 조정
                nn.BatchNorm2d(out_channels)  # Batch Normalization 적용
            )

    def forward(self, x):
        # 입력을 첫 번째 Convolution-BatchNorm-ReLU로 변환
        out = self.relu(self.bn1(self.conv1(x)))
        # 두 번째 Convolution-BatchNorm으로 변환
        out = self.bn2(self.conv2(out))
        # 원본 입력과 변환된 결과를 더함 (Residual Connection)
        out += self.shortcut(x)
        # 최종 출력에 ReLU 활성화 함수 적용
        return self.relu(out)  # ReLU 활성화 추가
```

```
# ResNet18 모델 정의
class ResNet18(nn.Module):
    def __init__(self, num_classes=31):
        super(ResNet18, self).__init__()

        # 입력 채널 수 설정 (ResidualBlock을 쌓아갈 때 사용)
        self.in_channels = 64

        # 초기 Convolution 레이어
        # RGB 이미지(3 채널)를 입력으로 받아 64개의 채널을 출력하는 Convolution Layer
        # kernel_size=7 : 7x7필터 사용, stride=2, padding=3으로 설정
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False)
        self.bn1 = nn.BatchNorm2d(64)  # Batch Normalization 적용
        self.relu = nn.ReLU(inplace=True)  # ReLU 활성화 함수 적용 #inplace=True : 입력텐서에서 직접연산을 수행함으로써 메모리 절약
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)  # 3x3 Max Pooling으로 다운샘플링

        # Residual 블록을 포함한 4개의 레이어
        # 각 레이어는 두 개의 Residual Block을 포함하며, 채널 수와 크기를 점진적으로 증가시킴
        self.layer1 = self.make_layer(64, 2, stride=1)   # 첫 번째 레이어, 채널 수 64, stride=1
        self.layer2 = self.make_layer(128, 2, stride=2)  # 두 번째 레이어, 채널 수 128, stride=2 (크기 절반으로 감소)
        self.layer3 = self.make_layer(256, 2, stride=2)  # 세 번째 레이어, 채널 수 256, stride=2 (크기 절반으로 감소)
        self.layer4 = self.make_layer(512, 2, stride=2)  # 네 번째 레이어, 채널 수 512, stride=2 (크기 절반으로 감소)


        # Adaptive Average Pooling을 통해 고정된 1x1 크기로 축소
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))

        # 최종 Fully Connected Layer
        # ResNet의 출력 채널 수 512에서 num_classes로 매핑
        self.fc = nn.Sequential(
            nn.Linear(512, num_classes)
        )

    # make_layer 함수 (Residual Block 생성 함수)
    # out_channels: 출력 채널 수
    # blocks: 블록의 개수
    # stride: 첫 번째 블록의 stride 설정 (그 외는 stride=1)
    def make_layer(self, out_channels, blocks, stride):
```

```
        layers = []

        # 첫 번째 Residual Block은 주어진 stride를 적용
        for _ in range(blocks):
            layers.append(ResidualBlock(self.in_channels, out_channels, stride))
            self.in_channels = out_channels  # 다음 블록의 입력 채널 수를 현재 출력 채널 수로 업데이트
            stride = 1  # 이후 블록은 stride=1로 고정

        # Sequential로 묶어 nn.Module로 반환
        return nn.Sequential(*layers)

    # Forward 함수 (입력 데이터가 네트워크를 통과하는 경로 정의)
    def forward(self, x):
        # 초기 Convolution-BatchNorm-ReLU-MaxPool 처리
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.maxpool(out)

        # Residual Layer 통과 (네 개의 레이어)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)

        # Adaptive Average Pooling 적용 (고정된 1x1 출력으로 축소)
        out = self.avgpool(out)

        # Flattening (1x1 크기로 축소된 텐서를 1차원으로 펼침)
        out = torch.flatten(out, 1)

        # Fully Connected Layer 통과하여 최종 출력 생성
        return self.fc(out)
```

3.3 모델, 손실 함수, 최적화 함수 초기화 및 earlystopping

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
# 모델, 손실 함수, 최적화 함수 초기화
model = ResNet18(num_classes=31).to(device)  # 모델을 GPU로 이동
criterion = nn.CrossEntropyLoss()  # 다중 클래스 분류를 위한 손실 함수
optimizer = Adam(model.parameters(), lr=0.001, weight_decay=1e-4)  # Adam 최적화 알고리즘 #l2규제 설정

writer = SummaryWriter()  # TensorBoard 로깅을 위한 SummaryWriter 생성
```

```
class EarlyStopping:
    def __init__(self, patience=5, min_delta=0):
        """
        초기화 함수
        :param patience: 성능이 개선되지 않은 에포크를 얼마나 기다릴지 결정
        :param min_delta: 성능 개선이 미미한 경우에도 개선된 것으로 보지 않기 위한 최소 개선값
        """
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
        elif val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.counter = 0  # 성능이 개선되었으므로 카운터 초기화
        else:
            self.counter += 1
            if self.counter >= self.patience:
                print("조기 종료(Early Stopping) 조건 만족. 학습을 중단합니다.")
                self.early_stop = True
```

```
# EarlyStopping 인스턴스 생성
early_stopping = EarlyStopping(patience=15, min_delta=0)
```

3.4 모델 학습 및 검증

```
from tqdm import tqdm
import time
```

```
# 학습 함수
def train(model, train_loader, optimizer, epoch, log_interval=200):
    model.train()
    train_loss = 0
    correct = 0
    total = 0

    # 에폭 시작 시간 기록
    start_time = time.time()

    # tqdm을 사용하여 학습 진행 표시
    with tqdm(total=len(train_loader), desc=f"Epoch [{epoch}/{EPOCHS}]", unit="batch", leave=True) as pbar:
        for batch_idx, (inputs, targets) in enumerate(train_loader):
            inputs, targets = inputs.to(device), targets.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()

            # 손실 및 정확도 계산
            train_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()
            accuracy = 100. * correct / total

            pbar.set_postfix({
                "batch_size": inputs.size(0),
                "loss": f"{train_loss / (batch_idx + 1):.2f}",
                "accuracy": f"{accuracy:.2f}%"
            })
            pbar.update(1)

    # 에폭 시간 계산
    epoch_time = time.time() - start_time

    # 최종 손실과 정확도 계산 및 출력
    train_loss /= len(train_loader)
    train_accuracy = 100. * correct / len(train_loader.dataset)
    print(f"Epoch [{epoch}/{EPOCHS}], Training Loss: {train_loss:.4f}, Training Accuracy: {train_accuracy:.2f}%, Time: {epoch_time:.2f} seconds", flu
    writer.add_scalar("Loss/Train", train_loss, epoch)
    writer.add_scalar("Accuracy/Train", train_accuracy, epoch)

# 검증 함수
def validation(model, validation_loader, epoch):
    model.eval()
    validation_loss = 0
    correct = 0
    with torch.no_grad():
        for inputs, targets in validation_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            validation_loss += loss.item()
            pred = outputs.argmax(dim=1, keepdim=True)
            correct += pred.eq(targets.view_as(pred)).sum().item()

    # 평균 손실과 정확도 계산
    validation_loss /= len(validation_loader)
    accuracy = 100. * correct / len(validation_loader.dataset)
    print(f"Epoch [{epoch}/{EPOCHS}], Validation Loss: {validation_loss:.4f}, Validation Accuracy: {accuracy:.2f}%", flush=True)
    writer.add_scalar("Loss/Validation", validation_loss, epoch)
    writer.add_scalar("Accuracy/Validation", accuracy, epoch)

# 훈련 루프
for epoch in range(1, EPOCHS + 1):
    train(model, train_loader, optimizer, epoch)
    validation(model, validation_loader, epoch)

# SummaryWriter 종료
writer.close()
```

```
Epoch [42/80]: 100%|██████████| 32/32 [08:56<00:00, 16.76s/batch, batch_size=102, loss=0.09, accuracy=98.45%]Epoch [42/80], Training Loss: 0.094

Epoch [42/80], Validation Loss: 3.3805, Validation Accuracy: 33.33%
Epoch [43/80]: 100%|██████████| 32/32 [08:47<00:00, 16.49s/batch, batch_size=102, loss=0.09, accuracy=98.85%]Epoch [43/80], Training Loss: 0.086

Epoch [43/80], Validation Loss: 2.0891, Validation Accuracy: 61.62%
Epoch [44/80]: 100%|██████████| 32/32 [08:44<00:00, 16.40s/batch, batch_size=102, loss=0.07, accuracy=99.16%]Epoch [44/80], Training Loss: 0.069

Epoch [44/80], Validation Loss: 2.0703, Validation Accuracy: 62.46%
Epoch [45/80]: 100%|██████████| 32/32 [08:52<00:00, 16.64s/batch, batch_size=102, loss=0.06, accuracy=99.19%]Epoch [45/80], Training Loss: 0.058

Epoch [45/80], Validation Loss: 2.1842, Validation Accuracy: 61.93%
Epoch [46/80]: 100%|██████████| 32/32 [08:54<00:00, 16.69s/batch, batch_size=102, loss=0.06, accuracy=99.26%]Epoch [46/80], Training Loss: 0.055

Epoch [46/80], Validation Loss: 2.3188, Validation Accuracy: 61.51%
Epoch [47/80]: 100%|██████████| 32/32 [08:49<00:00, 16.54s/batch, batch_size=102, loss=0.04, accuracy=99.53%]Epoch [47/80], Training Loss: 0.039

Epoch [47/80], Validation Loss: 2.2118, Validation Accuracy: 62.15%
Epoch [48/80]: 100%|██████████| 32/32 [08:50<00:00, 16.57s/batch, batch_size=102, loss=0.03, accuracy=99.66%]Epoch [48/80], Training Loss: 0.028

Epoch [48/80], Validation Loss: 2.3055, Validation Accuracy: 62.46%
Epoch [49/80]: 100%|██████████| 32/32 [08:51<00:00, 16.61s/batch, batch_size=102, loss=0.03, accuracy=99.63%]Epoch [49/80], Training Loss: 0.028

Epoch [49/80], Validation Loss: 2.3035, Validation Accuracy: 61.30%
Epoch [50/80]: 100%|██████████| 32/32 [08:50<00:00, 16.59s/batch, batch_size=102, loss=0.03, accuracy=99.58%]Epoch [50/80], Training Loss: 0.027

Epoch [50/80], Validation Loss: 2.4897, Validation Accuracy: 60.78%
Epoch [51/80]: 100%|██████████| 32/32 [08:49<00:00, 16.53s/batch, batch_size=102, loss=0.03, accuracy=99.61%]Epoch [51/80], Training Loss: 0.032

Epoch [51/80], Validation Loss: 2.3899, Validation Accuracy: 61.09%
Epoch [52/80]: 100%|██████████| 32/32 [08:45<00:00, 16.42s/batch, batch_size=102, loss=0.02, accuracy=99.75%]Epoch [52/80], Training Loss: 0.023

Epoch [52/80], Validation Loss: 2.2003, Validation Accuracy: 62.57%
Epoch [53/80]: 100%|██████████| 32/32 [08:53<00:00, 16.67s/batch, batch_size=102, loss=0.02, accuracy=99.73%]Epoch [53/80], Training Loss: 0.019

Epoch [53/80], Validation Loss: 2.5723, Validation Accuracy: 59.41%
Epoch [54/80]: 100%|██████████| 32/32 [08:55<00:00, 16.73s/batch, batch_size=102, loss=0.02, accuracy=99.75%]Epoch [54/80], Training Loss: 0.016

Epoch [54/80], Validation Loss: 2.2155, Validation Accuracy: 62.88%
Epoch [55/80]: 100%|██████████| 32/32 [08:55<00:00, 16.73s/batch, batch_size=102, loss=0.01, accuracy=99.78%]Epoch [55/80], Training Loss: 0.011

Epoch [55/80], Validation Loss: 2.4512, Validation Accuracy: 60.67%
Epoch [56/80]: 100%|██████████| 32/32 [08:46<00:00, 16.46s/batch, batch_size=102, loss=0.03, accuracy=99.66%]Epoch [56/80], Training Loss: 0.025

Epoch [56/80], Validation Loss: 2.0635, Validation Accuracy: 62.88%
Epoch [57/80]: 100%|██████████| 32/32 [08:54<00:00, 16.70s/batch, batch_size=102, loss=0.02, accuracy=99.66%]Epoch [57/80], Training Loss: 0.015

Epoch [57/80], Validation Loss: 2.0440, Validation Accuracy: 63.30%
Epoch [58/80]:  38%|███▊      | 12/32 [03:21<05:36, 16.81s/batch, batch_size=128, loss=0.01, accuracy=99.80%]
```

```
# 학습 성능 그래프 그리기
def plot_training_performance(train_losses, val_losses, train_accuracies, val_accuracies):
    epochs = range(1, num_epochs + 1)

    plt.figure(figsize=(12, 6))

    # 손실 그래프 (학습 손실과 검증 손실을 함께)
    plt.subplot(1, 2, 1)
    plt.plot(epochs, train_losses, label='Training Loss', color='blue')
    plt.plot(epochs, val_losses, label='Validation Loss', color='orange')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss over Epochs')
    plt.legend()

    # 정확도 그래프 (학습 정확도와 검증 정확도를 함께)
    plt.subplot(1, 2, 2)
    plt.plot(epochs, train_accuracies, label='Training Accuracy', color='blue')
    plt.plot(epochs, val_accuracies, label='Validation Accuracy', color='green')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.title('Training and Validation Accuracy over Epochs')
    plt.legend()

    plt.tight_layout()
    plt.show()
```

```
plot_training_performance(train_losses, val_losses, train_accuracies, val_accuracies)
```