In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

Mission1. 패션 스타일 이미지 분류

Mission 1-1.

주어진 이미지 데이터의 파일 명은 아래와 같은 형식이다.
"{W/T}{*이미지ID*}{시대별}{*스타일별*}{성별}.jpg"이에 기반하여 "이미지ID" 수 기준으로
"성별 & 스타일"통계치를아래표형식으로기입한다.

In [ ]:
```
import os
import pandas as pd
```

In [ ]:
```
def count_images_by_gender_and_style(file_names):
    rows = []  # 결과를 저장할 리스트

    # 파일명 분석 및 카운트
    for filename in file_names:
        # 파일 확장자 체크
        if not filename.endswith('.jpg'):
            continue

        # .jpg 제거
        no_jpg_filename = filename.split('.')[0]
        parts = no_jpg_filename.split('_')

        # 성별, 스타일, 이미지 ID 추출
        gender = '여성' if parts[-1] == 'W' else '남성'
        style = parts[3]
        image_id = parts[1]

        # 유효한 스타일과 이미지 ID일 때만 추가
        if style is not None and image_id is not None:
            rows.append({'성별': gender, '스타일': style, '이미지 ID': image_id})

    # DataFrame 생성
    df = pd.DataFrame(rows)

    # 중복된 행 제거
    df = df.drop_duplicates(subset=['성별', '스타일', '이미지 ID'])

    # 성별과 스타일별로 이미지 수 집계
    result = df.groupby(['성별', '스타일']).size().reset_index(name='이미지 수')
    result = result.sort_values(by='성별')

    return result
```

train

In [ ]:
```
# traininig_image 폴더
train_folder_path = '/content/drive/MyDrive/kict/dataset/training_image'

# 폴더 내의 파일 목록 가져오기
train_file_list = os.listdir(train_folder_path)
train_file_names = [filename for filename in train_file_list]
```

In [ ]:
```
# traininig_image 폴더
result_train = count_images_by_gender_and_style(train_file_names)
result_train.to_csv('mission_1-1_train.csv', index=False)
```

In [ ]:
```
result_train
```

Out[ ]:

	성별	스타일	이미지 수
0	남성	bold	268
1	남성	hiphop	274
2	남성	hippie	260
3	남성	ivy	237
4	남성	metrosexual	278
5	남성	mods	269
6	남성	normcore	364
7	남성	sportivecasual	298
21	여성	lounge	45
22	여성	military	33
23	여성	minimal	139
24	여성	normcore	153
28	여성	punk	65
26	여성	popart	41
27	여성	powersuit	120
20	여성	lingerie	55
25	여성	oriental	78
19	여성	kitsch	91
15	여성	genderless	77
17	여성	hiphop	48
16	여성	grunge	31
29	여성	space	37
14	여성	feminine	154
13	여성	ecology	64
12	여성	disco	37
11	여성	classic	77
10	여성	cityglam	67
9	여성	bodyconscious	95
8	여성	athleisure	67
18	여성	hippie	91
30	여성	sportivecasual	157

validation

In [ ]:
```
# validation_image 폴더
val_folder_path = '/content/drive/MyDrive/kict/dataset/validation_image'

# 폴더 내의 파일 목록 가져오기
val_file_list = os.listdir(val_folder_path)
val_file_names = [filename for filename in val_file_list]
```

In [ ]:
```
# validation_image 폴더
result_val = count_images_by_gender_and_style(val_file_names)
result_val.to_csv('mission_1-1_val.csv', index=False)
```

In [ ]:
```
result_val
```

Out[ ]:

	성별	스타일	이미지 수
0	남성	bold	57
1	남성	hiphop	66
2	남성	hippie	82
3	남성	ivy	79
4	남성	metrosexual	58
5	남성	mods	80
6	남성	normcore	51
7	남성	sportivecasual	52
21	여성	lounge	8
22	여성	military	9
23	여성	minimal	35
24	여성	normcore	20
28	여성	punk	12
26	여성	popart	8
27	여성	powersuit	34
20	여성	lingerie	5
25	여성	oriental	18
19	여성	kitsch	22
15	여성	genderless	12
17	여성	hiphop	8
16	여성	grunge	10
29	여성	space	15
14	여성	feminine	44
13	여성	ecology	17
12	여성	disco	10
11	여성	classic	22
10	여성	cityglam	18
9	여성	bodyconscious	23
8	여성	athleisure	14
18	여성	hippie	14
30	여성	sportivecasual	48

Mission 1-2.

ResNet-18를 활용하여 "성별 & 스타일" 단위로 클래스 분류를 수행하고 Validation 데이터에 대한 정확도를 제시한다.

- ResNet-18의 parameters는 무작위로 초기화하여 사용한다. (즉, pretrained weights는 사용할 수 없음)
- 성능을 높이기 위해 object detection, image cropping 등의 다양한 데이터 전처리 기법을 활용해도 무방하다.
  (데이터 전처리 단계에 한해서는 외부 라이브러리 활용 가능)

아래 코드는 모델 학습 속도 향상을 위해 drive에서 이미지 데이터를 직접 가져오는 것이 아니라 미리 colab의 content directory 안에 google drive에 있던 이미지 파일을 복사해서 코랩 세션 안에 포함시키는 코드

In [ ]:

```
from google.colab import drive
import shutil
import os

drive.mount('/content/drive')

source_folder = '/content/drive/MyDrive/kict/dataset/training_image'
destination_folder = '/content/training_image'

if not os.path.exists(destination_folder):
    os.makedirs(destination_folder)

shutil.copytree(source_folder, destination_folder, dirs_exist_ok=True)

print("training_image가 Colab content 폴더로 복사되었습니다.")

source_folder = '/content/drive/MyDrive/kict/dataset/training_label'
destination_folder = '/content/training_label'
```

```
if not os.path.exists(destination_folder):
    os.makedirs(destination_folder)

shutil.copytree(source_folder, destination_folder, dirs_exist_ok=True)

print("training_label가 Colab content 폴더로 복사되었습니다.")

source_folder = '/content/drive/MyDrive/kict/dataset/validation_image'
destination_folder = '/content/validation_image'

if not os.path.exists(destination_folder):
    os.makedirs(destination_folder)

shutil.copytree(source_folder, destination_folder, dirs_exist_ok=True)

print("validation_image가 Colab content 폴더로 복사되었습니다.")

source_folder = '/content/drive/MyDrive/kict/dataset/validation_label'
destination_folder = '/content/validation_label'

if not os.path.exists(destination_folder):
    os.makedirs(destination_folder)

shutil.copytree(source_folder, destination_folder, dirs_exist_ok=True)

print("validation_label가 Colab content 폴더로 복사되었습니다.")
```

```
validation_image가 Colab content 폴더로 복사되었습니다.
```

In [ ]:
```
import torch.nn as nn
import torch

from torch import Tensor
from typing import Type
```

ResNet을 구현할 때에 두 개의 컨볼루션 layer 단위로 residual connection을 이용해 output vector에 input vector를 더해주는 패턴이 반복되므로 class를 만들어주어서 밑의 ResNet class를 만들 때에 일일이 반복 구현하는 것이 아니라 block으로 편하게 가져다 쓰게 하기 위한 코드

In [ ]:
```
class BasicBlock(nn.Module):
    def __init__(
        self,
        in_channels: int,
        out_channels: int,
        stride: int = 1,
        expansion: int = 1,
        downsample: nn.Module = None
    ) -> None:
        super(BasicBlock, self).__init__()
        self.expansion = expansion
        # self.downsample = downsample
        self.conv1 = nn.Conv2d(
            in_channels,
            out_channels,
            kernel_size=3,
            stride=stride,
            padding=1,
            bias=False
        )
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(
            out_channels,
            out_channels*self.expansion,
            kernel_size=3,
            padding=1,
            bias=False
        )
        self.bn2 = nn.BatchNorm2d(out_channels*self.expansion)
        self.downsample = downsample

    def forward(self, x: Tensor) -> Tensor:
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

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

ResNet18 모델 구현 클래스 => 처음에는 7X7 kernel을 쓰는 convolution layer 1개, 64 channel을 가지는 convolution layer 4개(BasicBlock 2개 가져와서 구현함), 128 channel을 가지는 convolution layer 4개(BasicBlock 2개 가져와서 구현함), 256 channel을 가지는 convolution layer 4개(BasicBlock 2개 가져와서

구현함), 512 channel을 가지는 convolution layer 4개(BasicBlock 2개 가져와서 구현함), 마지막으로 classification을 위한 fully connected layer 1개로 이루어져 총 18개의 layer를 가지는 모델 생성

In [ ]:

```
class ResNet(nn.Module):
    def __init__(
        self,
        img_channels: int,
        num_layers: int,
        block: Type[BasicBlock],
        num_classes: int  = 1000
    ) -> None:
        super(ResNet, self).__init__()
        if num_layers == 18:
            layers = [2, 2, 2, 2]
            self.expansion = 1

        self.in_channels = 64
        self.conv1 = nn.Conv2d(
            in_channels=img_channels,
            out_channels=self.in_channels,
            kernel_size=7,
            stride=2,
            padding=3,
            bias=False
        )
        self.bn1 = nn.BatchNorm2d(self.in_channels)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

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

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

    def _make_layer(
        self,
        block: Type[BasicBlock],
        out_channels: int,
        blocks: int,
        stride: int = 1
    ) -> nn.Sequential:
        downsample = None
        if stride != 1:
            downsample = nn.Sequential(
                nn.Conv2d(
                    self.in_channels,
                    out_channels*self.expansion,
                    kernel_size=1,
                    stride=stride,
                    bias=False
                ),
                nn.BatchNorm2d(out_channels * self.expansion),
            )
        layers = []
        layers.append(
            block(
                self.in_channels, out_channels, stride, self.expansion, downsample
            )
        )
        self.in_channels = out_channels * self.expansion

        for i in range(1, blocks):
            layers.append(block(
                self.in_channels,
                out_channels,
                expansion=self.expansion
            ))
        return nn.Sequential(*layers)

    def forward(self, x: Tensor) -> Tensor:
        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)
        # print('Dimensions of the last convolutional feature map: ', x.shape)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x
```

이미지 데이터 정규화를 위한 평균과 표준편차 계산 수행

In [ ]:
```
# # 데이터셋의 평균과 표준편차 계산 함수
# def calculate_mean_std(loader):
#     mean = 0.0
#     std = 0.0
#     total_images_count = 0
#     for images, _ in loader:
#         batch_samples = images.size(0)  # 배치 크기 (이때 마지막 배치는 더 작을 수 있음)
#         images = images.view(batch_samples, images.size(1), -1)
#         mean += images.mean(2).sum(0)
#         std += images.std(2).sum(0)
#         total_images_count += batch_samples

#     mean /= total_images_count
#     std /= total_images_count
#     return mean, std

# # 임시로 ToTensor 변환만 적용하여 데이터 로더 생성
# temp_transform = transforms.Compose([
#     transforms.Resize((224, 224)),
#     transforms.ToTensor(),
# ])

# temp_train_dataset = CustomDataset(train_image_directory, transform=temp_transform)
# temp_train_loader = DataLoader(temp_train_dataset, batch_size=32, shuffle=True)

# # 평균과 표준편차 계산
# mean, std = calculate_mean_std(temp_train_loader)
# print(f"Calculated mean: {mean}")
# print(f"Calculated std: {std}")
```

위의 코드를 통해 이미지 특성 벡터의 평균과 표준편차를 구함

> Calculated mean: tensor([0.5498, 0.5226, 0.5052])
> Calculated std: tensor([0.2600, 0.2582, 0.2620])

실행 코드

In [ ]:
```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms
from PIL import Image
from sklearn.metrics import accuracy_score
from typing import Type
import numpy as np
import random

def seed_everything(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)
    os.environ["PYTHONHASHSEED"] = str(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = True
seed_everything()

# 이미지 파일이 있는 디렉토리 경로
train_image_directory = '/content/training_image'
valid_image_directory = '/content/validation_image'

# CustomDataset 클래스 정의
class CustomDataset(Dataset):
    def __init__(self, image_directory, transform=None):
        self.image_directory = image_directory
        self.image_files = [f for f in os.listdir(image_directory) if f.endswith('.jpg')]
        self.transform = transform

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

    def __getitem__(self, idx):
        file_name = self.image_files[idx]
        image_path = os.path.join(self.image_directory, file_name)
        image = Image.open(image_path).convert('RGB')

        # 이미지 파일명에서 스타일과 성별 정보 추출
        parts = file_name.split('_')
        if len(parts) < 4:
            raise ValueError(f"Invalid file name format: {file_name}")

        style_gender = parts[-2] + '_' + parts[-1].split('.')[0]  # 스타일과 성별 정보 추출

        # 스타일과 성별 정보를 레이블로 변환
```

```
        label = style_gender
        label_idx = label_to_index[label]

        if self.transform:
            image = self.transform(image)
        return image, label_idx

# 레이블을 숫자로 매핑하기 위한 딕셔너리 생성
label_set = set()
for file_name in os.listdir(train_image_directory) + os.listdir(valid_image_directory):
    parts = file_name.split('_')
    if len(parts) >= 4:
        style_gender = parts[-2] + '_' + parts[-1].split('.')[0]
        label_set.add(style_gender)

label_list = sorted(list(label_set))
label_to_index = {label: idx for idx, label in enumerate(label_list)}
index_to_label = {idx: label for label, idx in label_to_index.items()}

# 데이터 전처리 및 로드
# 이미지 크롭 및 회전 변환, 좌우 반전, 밝기, 채도, 색상 변화
# 평균과 표준편차를 기준으로 정규화
transform = transforms.Compose([
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
    transforms.RandomRotation(degrees=15),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ColorJitter(brightness=0.2,
                           contrast=0.2,
                           saturation=0.2,
                           hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5498, 0.5226, 0.5052], std=[0.2600, 0.2582, 0.2620]),
])

train_dataset = CustomDataset(train_image_directory, transform=transform)
val_dataset = CustomDataset(valid_image_directory, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True, num_workers=12, pin_memory=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False, num_workers=12, pin_memory=True)

# ResNet-18 모델 생성 함수
def resnet18(img_channels: int, num_classes: int) -> ResNet:
    return ResNet(img_channels, 18, BasicBlock, num_classes)

# 모델 인스턴스 생성
num_classes = len(label_list)  # 레이블의 총 개수
model = resnet18(img_channels=3, num_classes=num_classes) # RGB 이미지: 3개의 채널

# 손실 함수 및 옵티마이저 정의
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

#GPU 설정
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)

# 모델 학습
num_epochs = 100
patience = 5  # Early stopping patience
best_val_loss = float('inf')  # Initialize best validation loss as infinity
counter = 0  # Counter to track patience

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader)}')

    # Validation phase
    model.eval()
    all_preds = []
    all_labels = []
    val_running_loss = 0.0
    with torch.no_grad():
        for images, labels in val_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_running_loss += loss.item()
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
```

```
    # Calculate average validation loss
    avg_val_loss = val_running_loss / len(val_loader)
    print(f'Epoch [{epoch+1}/{num_epochs}], Validation Loss: {avg_val_loss}')

    # Early stopping logic
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        counter = 0  # Reset counter if validation loss improves
    else:
        counter += 1
        if counter >= patience:
            print(f"Early stopping at epoch {epoch+1}")
            break

# 정확도 계산
accuracy = accuracy_score(all_labels, all_preds)
print(f'Validation Accuracy: {accuracy * 100:.2f}%')

torch.save(model, '/content/drive/MyDrive/kict/model_path.pth')

# 예측 결과를 {W/T}_{스타일별} 형식으로 변환하여 출력
def predict_image(image_path, model, transform):
    image = Image.open(image_path).convert('RGB')
    image = transform(image).unsqueeze(0)  # 배치 차원 추가
    model.eval()
    with torch.no_grad():
        output = model(image)
        _, pred = torch.max(output, 1)
    return index_to_label[pred.item()]
```

```
Epoch [1/100], Loss: 3.357233326882124
Epoch [1/100], Validation Loss: 3.270069980621338
Epoch [2/100], Loss: 3.233607143163681
Epoch [2/100], Validation Loss: 3.208656088511149
Epoch [3/100], Loss: 3.2227494679391384
Epoch [3/100], Validation Loss: 3.25605149269104
Epoch [4/100], Loss: 3.195202112197876
Epoch [4/100], Validation Loss: 3.1865834395090737
Epoch [5/100], Loss: 3.180710546672344
Epoch [5/100], Validation Loss: 3.215518109003703
Epoch [6/100], Loss: 3.167237877845764
Epoch [6/100], Validation Loss: 3.1556209405263265
Epoch [7/100], Loss: 3.143862571567297
Epoch [7/100], Validation Loss: 3.2024326006571453
Epoch [8/100], Loss: 3.1315836757421494
Epoch [8/100], Validation Loss: 3.2884361584981283
Epoch [9/100], Loss: 3.109022404998541
Epoch [9/100], Validation Loss: 3.1331790924072265
Epoch [10/100], Loss: 3.1041545271873474
Epoch [10/100], Validation Loss: 3.111859591801961
Epoch [11/100], Loss: 3.0984501093626022
Epoch [11/100], Validation Loss: 3.2657238165537517
Epoch [12/100], Loss: 3.0663234181702137
Epoch [12/100], Validation Loss: 3.1748205184936524
Epoch [13/100], Loss: 3.073189754039049
Epoch [13/100], Validation Loss: 3.08036584854126
Epoch [14/100], Loss: 3.0528842583298683
Epoch [14/100], Validation Loss: 3.089481560389201
Epoch [15/100], Loss: 3.040457859635353
Epoch [15/100], Validation Loss: 3.1154067357381185
Epoch [16/100], Loss: 3.029873725026846
Epoch [16/100], Validation Loss: 3.029563554128011
Epoch [17/100], Loss: 3.008648782968521
Epoch [17/100], Validation Loss: 3.094943062464396
Epoch [18/100], Loss: 3.0057201385498047
Epoch [18/100], Validation Loss: 3.065688117345174
Epoch [19/100], Loss: 2.983147196471691
Epoch [19/100], Validation Loss: 3.021623150507609
Epoch [20/100], Loss: 2.9728832580149174
Epoch [20/100], Validation Loss: 3.101625108718872
Epoch [21/100], Loss: 2.9537645503878593
Epoch [21/100], Validation Loss: 2.9909815311431887
Epoch [22/100], Loss: 2.926023408770561
Epoch [22/100], Validation Loss: 3.1061229070027667
Epoch [23/100], Loss: 2.91116376593709
Epoch [23/100], Validation Loss: 3.066168435414632
Epoch [24/100], Loss: 2.890888437628746
Epoch [24/100], Validation Loss: 2.985174226760864
Epoch [25/100], Loss: 2.8555603213608265
Epoch [25/100], Validation Loss: 2.9513689041137696
Epoch [26/100], Loss: 2.849388759583235
Epoch [26/100], Validation Loss: 2.9888696511586508
Epoch [27/100], Loss: 2.825270812958479
Epoch [27/100], Validation Loss: 2.8901660124460857
Epoch [28/100], Loss: 2.802323404699564
Epoch [28/100], Validation Loss: 2.8883403460184733
Epoch [29/100], Loss: 2.7727655544877052
Epoch [29/100], Validation Loss: 2.997769260406494
Epoch [30/100], Loss: 2.761992748826742
Epoch [30/100], Validation Loss: 3.0199042638142903
Epoch [31/100], Loss: 2.7172203846275806
Epoch [31/100], Validation Loss: 2.878716739018758
Epoch [32/100], Loss: 2.689149621874094
Epoch [32/100], Validation Loss: 3.068777147928874
Epoch [33/100], Loss: 2.651088874787092
Epoch [33/100], Validation Loss: 3.0269419670104982
Epoch [34/100], Loss: 2.628532864153385
Epoch [34/100], Validation Loss: 2.7997143268585205
Epoch [35/100], Loss: 2.6071041338145733
Epoch [35/100], Validation Loss: 2.9269118944803876
Epoch [36/100], Loss: 2.5348040983080864
Epoch [36/100], Validation Loss: 2.817941093444824
Epoch [37/100], Loss: 2.5064006820321083
Epoch [37/100], Validation Loss: 2.9650727907816568
Epoch [38/100], Loss: 2.4639041535556316
Epoch [38/100], Validation Loss: 2.822582991917928
Epoch [39/100], Loss: 2.416964638978243
Epoch [39/100], Validation Loss: 2.7112606525421143
Epoch [40/100], Loss: 2.347602155059576
Epoch [40/100], Validation Loss: 2.803162209192912
Epoch [41/100], Loss: 2.3088497538119555
Epoch [41/100], Validation Loss: 3.080643622080485
Epoch [42/100], Loss: 2.2824249360710382
Epoch [42/100], Validation Loss: 2.669070529937744
Epoch [43/100], Loss: 2.2145906537771225
Epoch [43/100], Validation Loss: 2.606461842854818
Epoch [44/100], Loss: 2.150831762701273
Epoch [44/100], Validation Loss: 2.630651028951009
Epoch [45/100], Loss: 2.0828488916158676
Epoch [45/100], Validation Loss: 2.58056001663208
Epoch [46/100], Loss: 2.0086297020316124
```

```
Epoch [46/100], Validation Loss: 3.0513306617736817
Epoch [47/100], Loss: 1.9452666565775871
Epoch [47/100], Validation Loss: 2.532557153701782
Epoch [48/100], Loss: 1.8440248724073172
Epoch [48/100], Validation Loss: 2.6259525934855144
Epoch [49/100], Loss: 1.7597901802510023
Epoch [49/100], Validation Loss: 2.3900417486826577
Epoch [50/100], Loss: 1.6668452601879835
Epoch [50/100], Validation Loss: 2.4842599391937257
Epoch [51/100], Loss: 1.5919275227934122
Epoch [51/100], Validation Loss: 2.4924280484517416
Epoch [52/100], Loss: 1.5407171081751585
Epoch [52/100], Validation Loss: 2.4546359062194822
Epoch [53/100], Loss: 1.3797376342117786
Epoch [53/100], Validation Loss: 2.4075982173283896
Epoch [54/100], Loss: 1.3194763213396072
Epoch [54/100], Validation Loss: 2.2892051378885907
Epoch [55/100], Loss: 1.1769518638029695
Epoch [55/100], Validation Loss: 2.223650821050008
Epoch [56/100], Loss: 1.0946923708543181
Epoch [56/100], Validation Loss: 2.2058370033899943
Epoch [57/100], Loss: 1.0348248276859522
Epoch [57/100], Validation Loss: 2.4128665765126547
Epoch [58/100], Loss: 0.9448500042781234
Epoch [58/100], Validation Loss: 2.548377593358358
Epoch [59/100], Loss: 0.8542025191709399
Epoch [59/100], Validation Loss: 2.2537124395370483
Epoch [60/100], Loss: 0.7428056672215462
Epoch [60/100], Validation Loss: 2.449845727284749
Epoch [61/100], Loss: 0.715821304358542
Epoch [61/100], Validation Loss: 2.3449257055918378
Early stopping at epoch 61
Validation Accuracy: 49.63%
```

위의 모델과 정확히 똑같은 모델이지만 위의 코드에서 각 epoch 별로 accuracy를 출력해주는 것을 깜빡했다는 것을 뒤늦게 발견해서 아래 수정된 코드로 학습을 시켰지만 colab 컴퓨팅 리소스 단위를 다 쓰는 바람에 아래 코드는 제대로 학습을 마무리하지 못했습니다. 그래서 밑에 주석 처리한 형태로 첨부해 드렸습니다.

In [ ]:
```
# # ResNet-18 모델 생성 함수
# def resnet18(img_channels: int, num_classes: int) -> ResNet:
#     return ResNet(img_channels, 18, BasicBlock, num_classes) # 18은 ResNet-18을 지정

# # 모델 인스턴스 생성
# num_classes = len(label_list)  # 레이블의 총 개수
# model = resnet18(img_channels=3, num_classes=num_classes) # RGB 이미지: 3개의 채널

# # 손실 함수 및 옵티마이저 정의
# criterion = nn.CrossEntropyLoss()
# optimizer = optim.Adam(model.parameters(), lr=0.001)

# # 모델을 GPU로 이동
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# model.to(device)

# # 모델 학습
# num_epochs = 20
# patience = 5  # Early stopping patience
# best_val_loss = float('inf')  # Initialize best validation loss as infinity
# counter = 0  # Counter to track patience

# for epoch in range(num_epochs):
#     model.train()
#     running_loss = 0.0
#     correct = 0
#     total = 0

#     for images, labels in train_loader:
#         optimizer.zero_grad()
#         images, labels = images.to(device), labels.to(device)

#         outputs = model(images)
#         loss = criterion(outputs, labels)
#         loss.backward()
#         optimizer.step()

#         running_loss += loss.item()

#         # 정확도 계산
#         _, preds = torch.max(outputs, 1)
#         total += labels.size(0)
#         correct += (preds == labels).sum().item()

#     train_loss = running_loss / len(train_loader)
#     train_accuracy = correct / total * 100

#     print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.2f}%')

#     # Validation phase
#     model.eval()
#     all_preds = []
```

```
#     all_labels = []
#     val_running_loss = 0.0
#     val_correct = 0
#     val_total = 0

#     with torch.no_grad():
#         for images, labels in val_loader:
#             images, labels = images.to(device), labels.to(device)
#             outputs = model(images)
#             loss = criterion(outputs, labels)
#             val_running_loss += loss.item()

#             _, preds = torch.max(outputs, 1)
#             val_total += labels.size(0)
#             val_correct += (preds == labels).sum().item()
#             all_preds.extend(preds.cpu().numpy())
#             all_labels.extend(labels.cpu().numpy())

#     # Calculate average validation loss and accuracy
#     avg_val_loss = val_running_loss / len(val_loader)
#     val_accuracy = val_correct / val_total * 100

#     print(f'Epoch [{epoch+1}/{num_epochs}], Validation Loss: {avg_val_loss:.4f}, Validation Accuracy: {val_accuracy:.2f}%')

#     # Early stopping logic
#     if avg_val_loss < best_val_loss:
#         best_val_loss = avg_val_loss
#         counter = 0  # Reset counter if validation loss improves
#     else:
#         counter += 1
#         if counter >= patience:
#             print(f"Early stopping at epoch {epoch+1}")
#             break
```

In [ ]:
```
# # 정확도 계산
# accuracy = accuracy_score(all_labels, all_preds)
# print(f'Validation Accuracy: {accuracy * 100:.2f}%')
```

In [ ]:
```
# # 모델 저장
# torch.save(model, '/content/drive/MyDrive/kict/model_path.pth')
```