딥러닝 기초 기말 Project

학번: M23048

이름: 이가현

서약

아래 보고서는 **본인의 힘만으로 작성**해야 하며, 다른 학생에게 질문과 다른 학생의 코드를 참고하는 행위는 모두 금지합니다

* 수업에서 제공한 코드, 노트북은 모두 재활용 가능하며, 카피로 규정하지 않습니다

* 수업 자료 이외에 참고자료가 있다면, 출처와 사용 부분에 모두 표시하는 경우는 모두 합당한 자료로 인정하겠습니다

* 위에 대해서 모두 이해하고 동의했다면, 아래 `서약글`에 다음을 작성해주세요:

"본인은 위 서약글을 이해하고 동의하며, 프로젝트를 수행하는데 있어서 반칙을 할 경우 (제공자 포함) 본 프로젝트에 대한 점수가 반영되지 않는다는 것에 동의합니다."

학번: M23048

이름: 이가현

서약글: 본인은 위 서약글을 이해하고 동의하며, 프로젝트를 수행하는데 있어서 부정행위를 할 경우(제공자 포함)본 프로젝트에 대한 점수가 반영되지 않는다는 것에 동의합니다.

*모든 코드에는 주석을 작성해 주세요

최종 제출시, 본 보고서와 .ipynb 노트북파일, test에 사용한 모델(.pt)파일을 압축해 제출해 주세요.

중요: 사용한 기법은 자신이 이해한 것 만을 사용하세요. 설명하지 않은 기법을 사용하면 그 부분을 제외하고 채점하겠습니다. 예를 들어서 자신의 힘으로 찾은 코드를 이용하려하는 경우 내용을 이해하고 보고서에 이해한 내용이 충분히 설명이 되어야만 사용을 허용합니다.

Step 1: Dataset 준비하기

<코드 캡쳐 첨부>

실험환경 : NVIDIA GeForce RTX 4090

```
# './data.csv' 파일에서 훈련 데이터를 읽어와 train_dataframe 변수에 저장
train_dataframe = pd.read_csv('./data.csv')
# './testdata.csv' 파일에서 테스트 데이터를 읽어와 test_df 변수에 저장
test_df = pd.read_csv('./testdata.csv')
# train_test_split 함수를 사용하여 훈련 데이터를 훈련 세트와 검증 세트로 분리
train_df, valid_df = train_test_split(train_dataframe, shuffle=True, test_size=0.3, stratify=train_dataframe['Label'])
```

scikit-learn의 'train_test_split' 함수를 사용하여 훈련 데이터를 훈련세트와 검증세트로 나누었습니다. 'shuffle=True'로 지정하여 데이터를 섞고, 'test_size=0.3'으로 지정하여 **훈련데이터의 30%를 검증 데이터로 사용**하였습니다. 또한 'stratify=train_dataframe['Label']로 지정하여 클래스 레이블의 분포가 유지되도록 하였습니다.

```
# './data.csv' 파일에서 훈련 데이터를 읽어와 train_dataframe 변수에 저장
train_dataframe = pd.read_csv('./data.csv')
# './testdata.csv' 파일에서 테스트 데이터를 읽어와 test_df 변수에 저장
test_df = pd.read_csv('./testdata.csv')
# train_test_split 함수를 사용하여 훈련 데이터를 훈련 세트와 검증 세트로 분리
train_df, valid_df = train_test_split(train_dataframe, shuffle=True, test_size=0.01, stratify=train_dataframe['Label'])
```

최적의 모델을 찾은 후에는 제가 만든 모델을 믿고 최대한 많은 학습 데이터셋을 모델에 학습시켜야 성능 개선에 효과가 있을 것이라 생각하였습니다. 이러한 이유로 리더보드에 제출하기 전에는 'test_size=0.01'로 지정하여 거의 **모든 훈련 데이터를 학습에 사용**하였습니다.

```
# template-code.ipynb
class CustomDataset(torch.utils.data.Dataset):
    # 초기화 __init__ 함수
    def __init__(self, dataframe, train='train', transform=None):
        # 훈련 데이터셋일 경우
        if train == 'train':
            self.image_list = [] # 이미지 경로
            self.label_list = [] # 레이블
            self.other_list = [] # 나이, 성별, 인종
            path = './dataset/{}/{}'
            for index, row in dataframe.iterrows():
                image_path = row['Image']
                image_label = row['Label']
                image_age = row['Age']
                image_gender = row['Gender']
                image_race = row['Race']
                image = Image.open(path.format(image_label, image_path)).convert('RGB')
                if transform != None:
                    image = transform(image)
                self.image_list.append(image)
                self.label_list.append(image_label)
                self.other_list.append((image_age, image_gender, image_race))
        # 테스트 데이터셋일 경우
        elif train == 'test':
            self.image_list = [] # 이미지 경로
            self.label_list = [] # 레이블
            self.other_list = [] # 성별, 인종
            path = './testset/{}'
            for index, row in dataframe.iterrows():
                image_path = row['Image']
                image_gender = row['Gender']
                image_race = row['Race']
                image = Image.open(path.format(image_path)).convert('RGB')
                if transform != None:
                    image = transform(image)
                self.image_list.append(image)
                self.label_list.append(image_path)
                self.other_list.append((image_gender, image_race))
    # 데이터셋의 총 샘플 수 반환
    def __len__(self):
        return len(self.image_list)
    # 주어진 idx에 해당하는 샘플(이미지 리스트, 레이블 리스트, 다른 정보 리스트) 추출하여 반환
    def __getitem__(self, idx):
        return self.image_list[idx], self.label_list[idx], self.other_list[idx]
```

CustomDataset Class에 대한 코드는 기본 제공된 template-code를 사용하였습니다.

Step 2: Dataset 에 대한 Data Loaders 구성

```
# 참고자료 : https://deep-learning-study.tistory.com/475
# 훈련 데이터셋의 경로
train_path = "C:/Users/user/PycharmProjects/dl/dataset"
# 이미지를 텐서 형태로 변환하고, datasets.ImageFolder를 사용하여 이미지 폴더에 있는 데이터셋을 변수에 저장
train_ds = datasets.ImageFolder(root = train_path, transform=transforms.ToTensor())
# train_ds에서 각 이미지의 RGB 평균 값을 계산하여 meanRGB에 할당
# (x.numpy()는 이미지를 NumPy 배열로 변환, np.mean 함수를 사용하여 픽셀 값을 축에 따라 평균 계산)
meanRGB = [np.mean(x.numpy(), axis=(1,2)) for x,_ in train_ds]
# 표준편차 계산
stdRGB = [np.std(x.numpy(), axis=(1,2)) for x,_ in train_ds]

meanR = np.mean([m[0] for m in meanRGB]) # R채널의 평균 값
meanG = np.mean([m[1] for m in meanRGB]) # G채널의 평균 값
meanB = np.mean([m[2] for m in meanRGB]) # B채널의 평균 값

stdR = np.mean([s[0] for s in stdRGB]) # R채널의 표준편차 값
stdG = np.mean([s[1] for s in stdRGB]) # G채널의 표준편차 값
stdB = np.mean([s[2] for s in stdRGB]) # B채널의 표준편차 값

print(meanR, meanG, meanB)
print(stdR, stdG, stdB)
```

실제로는 모델 성능 향상에 큰 도움이 되지는 않았지만, 저는 제공된 데이터에 맞춤형 데이터 전처리를 하는 것이 미미하지만 영향이 있을 것이라 생각하였기 때문에 자료(https://deep-learning-study.tistory.com/475)를 참고하여 채널 별 평균 값과 표준편차 값을 계산하였습니다. Normalization하기 위해서 사람 얼굴 이미지 데이터셋 픽셀의 평균, 표준편차를 계산한 값을 데이터 전처리에 적용하였습니다.

데이터 전처리에 대한 설명

```
train_transform = transforms.Compose([
    transforms.Resize(224), # 224*224의 크기로 image Resize
    transforms.RandomHorizontalFlip(p=0.5), # 50% 확률로 랜덤 좌우 반전
    transforms.ToTensor(), # tensor로 변환
    transforms.Normalize((0.6029, 0.4615, 0.3949),(0.2195, 0.1960, 0.1866)) # Normalization
])

test_transform = transforms.Compose([
    transforms.Resize(224), # 224*224의 크기로 image Resize
    transforms.ToTensor(), # tensor로 변환
    transforms.Normalize((0.6029, 0.4615, 0.3949),(0.2195, 0.1960, 0.1866)) # Normalization
])
```

위 코드는 데이터 증강(Data Augmentation)을 다양하게 시도한 후, 최종적으로 사용한 데이터 전처리 방법입니다. 저는 최대한 많은 실험을 해보고 싶었기 때문에 학습 속도도 중요했습니다. 연산량이 증가하면 학습 속도가 저하되기 때문에, 데이터 전처리를 시도한 방법 중 검증 정확도(validation accuracy)가 1% 이상 향상된 경우에 사용했던 방식만 적용하였습니다. 그렇지 않은 경우에는 최대한 가장 기본적이고 간단한 데이터 증강만을 사용했습니다.

```
transforms.Resize(224), # 224*224의 크기로 image Resize
```

이미지의 원본 크기는 200x200이며, 이미지 크기를 224로 리사이즈한 이유는 모델 스케일링(Model Scaling) 방법 중 **해상도 스케일링**(Resolution Scaling)을 적용하기 위함입니다.

```
transforms.RandomHorizontalFlip(p=0.5), # 50% 확률로 랜덤 좌우 반전
```

무작위 좌우 반전을 적용했을 때 성능 개선이 눈에 띄었으며, p 값을 조절하여 최적의 반전 비율을 50%로 찾을 수 있었습니다.

```
transforms.ToTensor(), # tensor로 변환
transforms.Normalize((0.6029, 0.4615, 0.3949),(0.2195, 0.1960, 0.1866)) # Normalization
```

이후 이미지를 텐서로 변환하고, 이미지의 RGB 채널별로 평균과 표준 편차를 사용하여 정규화를 진행하여 데이터 증강을 수행했습니다. 테스트 데이터의 경우에는 모델을 더 혼란스럽게 할 필요가 없으므로 기본적인 전처리 외에는 적용하지 않았습니다.

```
train_dataset = CustomDataset(train_df, train='train', transform=train_transform) # 5138
valid_dataset = CustomDataset(valid_df, train='train', transform=test_transform) # 2202
test_dataset = CustomDataset(test_df, train='test', transform=test_transform) # 822

batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)
```

Batch size도 성능 개선에 영향이 있다고 알고 있었기 때문에 32, 64, 128, 256 순으로 진행해보았지만 뚜렷한 차이는 없었습니다. 그래서 최종적으로 Batch 크기는 64로 설정하여 훈련 및 테스트 데이터를 해당 크기의 미니 배치로 나누고 데이터를 shuffle하여 학습을 진행하였습니다.

성능 개선에는 효과가 없었지만 시도한 데이터 전처리

➔ **Torchvision.transforms 변환기**

우리의 얼굴 데이터셋은 대체적으로 Crop된 정면의 얼굴 이미지로 이루어져 있습니다. 이미 데이터 전처리가 잘 되어있는 좋은 데이터라고 생각하였기 때문에, 좌우반전이나 약간의 회전 외에 방법은 성능적인 측면에서 큰 효과를 보지 못할 것이라 생각했었습니다.

실제로 transform에서 RandomRotation, ColorJitter, ToGray, RandAugment를 적용해보았습니다. (참고자료 : https://pytorch.org/vision/stable/transforms.html) 해당 방법을 적용한 이유로는 다음과 같습니다. 원본 이미지를 크게 변환하지 않는 선에서 증강을 시도하기 위해 약간의 RandomRotation을 진행했었고, 대비를 주면 이미지가 선명해져서 모델의 학습에 도움이 될 것이라고 생각한 ColorJitter의 Contrast를 적용하였지만 성능 개선의 효과를 보지 못하였습니다. 데이터셋에 흑백 사진도 있었기 때문에 ToGray를 진행하였으나 눈에 띄는 성과는 없었고 ColorJitter의 brightness와 RandAugment의 경우에는 성능이 떨어졌습니다.

➔ **Albumentation 변환기**

```
import albumentations as A
from albumentations.pytorch import ToTensorV2
```

하지만 일반적으로 딥러닝에서 성능 개선에 있어서 데이터 증강은 필수적이라고 알려져 있기 때문에 우리의 데이터셋에 맞는 증강 방법이 분명히 있을 것이라고 생각하였습니다. 그래서 testset을 직접 눈으로 살펴보면서 다양한 시도를 하였습니다. 잘 정제된 것처럼 보이는 정면 얼굴 사진 외의 사진을 눈여겨 보았습니다. 이 사진들에 대한 설명은 아래에서 그림과 함께 설명하겠습니다. 해당 경우에 제가 적용해보고 싶은 더 다양한 데이터 증강 방법을 사용하기 위해서는 Albumentation 변환기를 사용해야만 가능하였으므로, Custom dataset 코드도 다시 구현하였습니다.

다음은 Albumentation을 위해 다시 작성한 CustomDataset 코드입니다.

```
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, dataframe, train='train', transform=None):
        if train == 'train':
            self.image_list = []
            self.label_list = []
            self.other_list = []
            path = 'dataset/{}/{}'
            for index, row in dataframe.iterrows():
                image_path = row['Image']
                image_label = row['Label']
                image_age = row['Age']
                image_gender = row['Gender']
                image_race = row['Race']
                image = Image.open(path.format(image_label, image_path)).convert('RGB')
                image = np.array(image)
                # if there is transform, apply transform
                if transform != None:
                    image = transform(image=image)["image"]
                self.image_list.append(image)
                self.label_list.append(image_label)
                self.other_list.append((image_age, image_gender, image_race))
        elif train == 'test':
            self.image_list = []
            self.label_list = []  # 이미지의 경로
            self.other_list = []
            path = 'testset/{}'
            for index, row in dataframe.iterrows():
                image_path = row['Image']
                image_gender = row['Gender']
                image_race = row['Race']
                image = Image.open(path.format(image_path)).convert('RGB')
                image = np.array(image)
                if transform != None:
                    image = transform(image=image)["image"]
                self.image_list.append(image)
                self.label_list.append(image_path)
                self.other_list.append((image_gender, image_race))

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

    def __getitem__(self, idx):
        return self.image_list[idx], self.label_list[idx], self.other_list[idx]
```

기존에 작성되어 있는 CustomDataset class를 사용하면 에러가 발생해서 사용할 수 없었습니다. 에러를 고쳐가면서 albumentation을 사용하기 위한 CustomDataset class를 구현하였습니다. 일반적인 transform 같은 경우에는 이미지를 입력으로 받아 변환된 이미지를 반환하는 방식으로 아래와 같이 사용됩니다.

```
image = transform(image)
```

하지만 Albumentations 라이브러리에서는 아래 코드와 같이 딕셔너리 형태로 이미지를 다시 할당해야 Albumentations 라이브러리의 변환 기능을 제대로 활용할 수 있었습니다.

```
image = transform(image=image)["image"]
```

참고자료 : https://geunuk.tistory.com/65

[Case1]

[Case2]

테스트 이미지를 살펴보면 위와 같은 데이터들이 있었고, 이러한 데이터들에 대해 예측을 더 잘 수행했으면 해서 다음과 같은 증강을 진행하였습니다.

```
train_transform = A.Compose(
    [
        A.Resize(224,224),
        A.GridDistortion(always_apply=False, p=1.0, num_steps=5, distort_limit=(-0.3, 0.3), interpolation=0, border_mode=0, value=(0, 0, 0), mask_value=None),
        A.CoarseDropout(always_apply=False, p=1.0, max_holes=8, max_height=8, max_width=8, min_holes=8, min_height=8, min_width=8),
        A.HorizontalFlip(p=0.5),
        A.Normalize([0.6029, 0.4615, 0.3949],[0.2195, 0.1960, 0.1866]),
        A.pytorch.ToTensorV2(),
    ]
)
```

Case1 같은 경우에는 GridDistortion을 적용하면 잘 예측 할 것이라 생각하였고, Case2 경우에는 입력 이미지에 검은색 직사각형을 임의로 넣어주는 CoarseDropout을 적용하면 좋을 것 이라 생각하였습니다.

[GridDistortion 예시 이미지]

[CoarseDropout 예시 이미지]

효과는 없었던 이유는 테스트 데이터에 이러한 데이터가 소수였기 때문이라고 생각하였습니다.

Step 3: Neural Network 생성

- Pretrained model을 허용하지 않습니다. (직접 모델을 설계해 주세요)

```python
class ConvNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 32, 3), nn.BatchNorm2d(32), nn.ReLU(True)) # 3*224*224 -> 32*222*222
        self.cn2 = nn.Sequential(nn.Conv2d(32, 32, 3, padding=1), nn.BatchNorm2d(32), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 32*222*222 -> 32*111*111
        self.cn3 = nn.Sequential(nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 32*111*111 -> 64*55*55
        self.cn4 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 64*55*55 -> 64*27*27
        self.cn5 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 64*27*27 -> 128*13*13
        self.cn6 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 128*13*13 -> 128*6*6
        self.cn7 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 128*6*6 -> 256*3*3
        self.cn8 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 256*3*3 -> 256*1*1

        self.fc1 = nn.Sequential(nn.Linear(256+10, 32),
                                 nn.BatchNorm1d(32),
                                 nn.ReLU(True),
                                 nn.Dropout(0.5),
                                 nn.Linear(32, 5))
```

```python
def forward(self, x, other_gender, other_race): # 매개변수로 이미지 데이터, 성별, 인종 정보 전달
    x = self.cn1(x)
    x = self.cn2(x)
    x = self.cn3(x)
    x = self.cn4(x)
    x = self.cn5(x)
    x = self.cn6(x)
    x = self.cn7(x)
    x = self.cn8(x)

    x = x.view(x.size(0), -1) # flatten

    other_gender = torch.nn.functional.one_hot(other_gender, 5) # 성별에 대해 원핫인코딩
    other_race = torch.nn.functional.one_hot(other_race, 5) # 인종에 대해 원핫인코딩

    other_gender = other_gender.view(-1, 5) # flatten
    other_race = other_race.view(-1, 5) # flatten

    x = torch.cat((x, other_gender), dim=1) # 입력 텐서 x에 성별 정보 추가(concatenate)
    x = torch.cat((x, other_race), dim=1) # 입력 텐서 x에 인종 정보 추가(concatenate)

    x = self.fc1(x)

    return x
```

위 코드는 제가 최종적으로 제출하는데 필요했던 베이스 모델입니다. 위 모델을 기반으로 하이퍼파라미터를 튜닝하는 것처럼 모델을 아주 조금씩 변형하여 새로운 모델을 만들어 나가는 과정을 거쳐서 앙상블을 진행하였습니다. 최종 제출을 위한 앙상블을 할 때 사용한 모델들은 위 모델과 아주 유사하게 설계되었기 때문에 Step6에서 추가적으로 설명하려고 합니다.

설계한 모델을 출력 후 네트워크를 구성한 방법과 이유를 각 단계별로 설명

<서술형>

```
ConvNet(
  (cn1): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (cn2): Sequential(
    (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): ReLU(inplace=True)
  )
  (cn3): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): ReLU(inplace=True)
  )
  (cn4): Sequential(
    (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): ReLU(inplace=True)
  )
  (cn5): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): ReLU(inplace=True)
  )
  (cn6): Sequential(
    (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): ReLU(inplace=True)
  )
```

```
  (cn7): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): ReLU(inplace=True)
  )
  (cn8): Sequential(
    (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): ReLU(inplace=True)
  )
  (fc1): Sequential(
    (0): Linear(in_features=266, out_features=32, bias=True)
    (1): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=32, out_features=5, bias=True)
  )
)
```

아주 초기에는 교수님의 Cifar10 실습 자료와 굉장히 유사하게 모델을 구성하였지만, 다양한 시도 후에 최종 제출을 위한 모델을 만들때는 복잡한 신경망(12층)부터 시작해서 모델을 경량화 시켜 나가는 과정을 거쳤기 때문에 한눈에 보기 쉬운

Pytorch의nn.Sequential(https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html)을 이용하여 구현하였습니다.

- 차원분석

```
class ConvNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 32, 3), nn.BatchNorm2d(32), nn.ReLU(True)) # 3*224*224 -> 32*222*222
        self.cn2 = nn.Sequential(nn.Conv2d(32, 32, 3, padding=1), nn.BatchNorm2d(32), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 32*222*222 -> 32*111*111
        self.cn3 = nn.Sequential(nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 32*111*111 -> 64*55*55
        self.cn4 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 64*55*55 -> 64*27*27
        self.cn5 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 64*27*27 -> 128*13*13
        self.cn6 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 128*13*13 -> 128*6*6
        self.cn7 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 128*6*6 -> 256*3*3
        self.cn8 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True)) # 256*3*3 -> 256*1*1

        self.fc1 = nn.Sequential(nn.Linear(256+10, 32),
                                 nn.BatchNorm1d(32),
                                 nn.ReLU(True),
                                 nn.Dropout(0.5),
                                 nn.Linear(32, 5))
```

	[input]	[output]
Conv2d-1	3x224x224	32x222x222
BatchNorm2d-2	32x222x222	32x222x222
ReLU-3	32x222x222	32x222x222
Conv2d-4	32x222x222	32x222x222
BatchNorm2d-5	32x222x222	32x222x222
MaxPool2d-6	32x222x222	32x111x111
ReLU-7	32x111x111	32x111x111
Conv2d-8	32x111x111	32x111x111
BatchNorm2d-9	32x111x111	32x111x111
MaxPool2d-10	32x111x111	64x55x55
ReLU-11	64x55x55	64x55x55
Conv2d-12	64x55x55	64x55x55
BatchNorm2d-13	64x55x55	64x55x55
MaxPool2d-14	64x55x55	64x27x27
ReLU-15	64x27x27	64x27x27
Conv2d-16	64x27x27	64x27x27
BatchNorm2d-17	64x27x27	64x27x27
MaxPool2d-18	64x27x27	128x13x13
ReLU-19	128x13x13	128x13x13

Conv2d-20	128x13x13	128x13x13
BatchNorm2d-21	128x13x13	128x13x13
MaxPool2d-22	128x13x13	128x6x6
ReLU-23	128x6x6	128x6x6
Conv2d-24	128x6x6	128x6x6
BatchNorm2d-25	128x6x6	128x6x6
MaxPool2d-26	128x6x6	256x3x3
ReLU-27	256x3x3	256x3x3
Conv2d-28	256x3x3	256x3x3
BatchNorm2d-29	256x3x3	256x3x3
MaxPool2d-30	256x1x1	256x1x1
ReLU-31	256x1x1	256x1x1
Linear-32	266x1x1	266x1x1
BatchNorm1d-33	[-1, 266]	[-1, 32]
ReLU-34	[-1, 32]	[-1, 32]
Dropout-35	[-1, 32]	[-1, 32]
Linear-36	[-1, 32]	[-1, 5]

Convolution layer의 차원 계산은 (input dim + 2*padding – filter)/stride+1 수식에 의해서 계산되었습니다. 제 모델에 사용한 Maxpooling의 stride와 padding은 2이므로 input dim의 절반에 해당하는 크기를 얻게 됩니다.

마지막 convolution layer의 차원은 256x1x1입니다. Fully Connected layer의 입력차원은 이미지 데이터만 들어갔다면 256이였을텐데, 저는 성별정보와 인종정보를 원핫인코딩 후 크기 10으로 flatten하였기 때문에 256+10인 266이 됩니다.

모델을 본격적으로 구축하기 전에 다양한 실험을 통해 어떤 모델이 얼굴 데이터셋에 적합한지 감을 익히기 위해서 이미 잘 알려진 모델들로 실험을 진행하였습니다. CNN 실험은 딥러닝기초 교안에 있는 AlexNet과 실습 중 lab_cifar10.ipynb 모델을 통해 진행하였습니다. ResNet 실험은 https://pseudo-lab.github.io/pytorch-guide/docs/ch03-1.html 해당 참고자료를 통해 Resnet34, Resnet50, Wide ResNet, ResNext 모델로 진행하였습니다. EfficientNet 실험은 https://deep-learning-study.tistory.com/563 자료를 참고해서 진행하였습니다.

처음에는 Resnet과 EfficientNet과 같은 깊은 모델이 성능이 우수할 것으로 예상하였으나, 실제 실험 결과 깊은 모델일수록 성능이 좋지 않았습니다. 위 실험결과를 토대로 신경망이 깊다고 무조건 좋은 것은 아니라는 결론을 얻게 되었습니다. 그래서 우리의 데이터셋과 테스크에 맞는 모델을 구축하기 위해 깊이가 비교적 적은 실습시간에 사용한 CNN 모델을 활용하고자 하였습니다.

최종적으로 구성한 모델은 위와 같이 Convolutional Layer 8층과 Linear Layer 2층으로 구성되었습니다. Convolutional Layer의 설계에서는 이미지의 크기를 최대한 유지하기 위해 첫 번째 레이어를 제외한 모든 층에 대해 kernel_size = 3, padding = 1을 적용하였습니다. 이렇게 함으로써 각 층에서의 출력 이미지 크기를 입력 이미지와 동일하게 유지할 수 있었습니다. Kernel_sisze를 3으로 고정시킨 이유는, VGG에서 3x3 filter를 사용한 이유와 같습니다. 같은 feature map을 만드는데 7x7 filter 하나를 사용하는 것 보다 3x3 filter 3개를 사용하는 것이 훨씬 더 적은 parameter를 사용하기에 더 효율적이고 모델을 깊게 구성할 수 있기 때문입니다. 또한, Convolutional Layer를 거친 후에는 학습의 안정성과 성능 향상을 위해 BatchNormalization을 적용하였습니다. Maxpooling을 할 때마다 이미지 크기가 절반이 되도록 하기 위해 kernel과 stride를 2로 설정하였습니다. Activation Function으로는 ReLU를 선택하였는데, 이는 backpropagation 시 gradient vanishing 문제를 완화시키는 효과를 가지고 있습니다.

모델의 입력값으로 이미지만 넣었을 때/ 이미지 + 성별정보만 넣었을 때/ 이미지 + 성별 + 인인종정보 모두 넣었을 때로 나누어 실험을 진행하였습니다. 최종으로 구성한 모델에서는 이미지 + 성별 + 인종정보 모두 사용하였습니다. Fully Connected Layer의 입력값으로 이미지 정보에 성별정보와 인종정보를 추가할 수 있도록 원핫인코딩 후, flatten을 진행하였습니다.

Step 4: Cost (Loss) Function 과 Optimizer 선택

<코드 캡쳐 첨부>

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001,weight_decay=0.01)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10, eta_min=0)
```

Optimizer와 Cost 함수를 선정한 이유와 선정하는데 중요하다고 생각하는 내용을 모두 작성합니다.

<서술형>

최종적으로 선택한 Cost 함수로는 CrossEntropyLoss(), Optimizer는 Adam(), Scheduler는 CosineAnnealingLR입니다.

실험을 통해 확인한 결과, 제 모델에 대해서는 효과가 없었지만, 추가적으로 실험한 Cost 함수로는 label smoothing과 focal loss가 있었습니다. 결과를 뽑아봤을 때, 2번과 3번 클래스를 맞추지 못하는 경향을 관찰하였으며, 해당 Cost 함수를 적용하면 문제가 해결될 것으로 기대하였지만, 데이터셋의 클래스별 데이터 개수가 유사하여 그런것인지 성능 향상에 대한 효과가 전혀 없었습니다.

또한, Optimizer로는 일반적으로 널리 사용되는 Adam을 선택하였으며, loss를 더 낮추기 위해 AdamW와 Adamax를 시도해 보기도 하였지만 큰 효과를 보지 못하여 결국 Adam을 사용하였습니다. Learning Rate는 0.1부터 0.00001까지 조절하며 하이퍼파라미터 튜닝을 진행하였습니다.

Validation loss가 0에 수렴하지 않고 과적합 현상을 보여서 weight_decay를 적용하여 과적합을 해결하였습니다. 교수님께서 제공해주신 수업 자료에 있는 weight_decay의 추천 값을 모두 실험해보았고, 규제를 강화시키기 위해 더 큰 값인 0.01을 사용하였더니 과적합이 어느정도 해결되었습니다.

스케줄러도 다양하게 시도해보았는데 특히 StepLR과 CosineAnnealingLR을 위주로 실험하였으며, StepLR일 때 Validation loss가 0에 수렴하는 것이 더 잘 되었지만, 성능 향상 측면에서는 CosineAnnealingLR이 더 좋은 결과를 보였습니다. Dropout이나 weight_decay를 통해 규제를 충분히 줄 수 있다고 판단하여, 성능에 더 집중하기 위해 CosineAnnealingLR을 선택하였습니다.

Step 5: 구성한 모델에 대한 Train and Validate 진행

<코드 캡쳐 첨부>

```
n_epochs = 30

valid_loss_min = np.Inf # 최소 검증 손실을 무한대로 초기화
train_loss = torch.zeros(n_epochs) # 각 epoch에 대한 훈련 손실 값 초기화
valid_loss = torch.zeros(n_epochs) # 각 epoch에 대한 검증 손실 값 초기화

train_acc = torch.zeros(n_epochs) # 각 epoch에 대한 훈련 정확도 값 초기화
valid_acc = torch.zeros(n_epochs) # 각 epoch에 대한 검증 정확도 값 초기화
```

```
for e in range(0, n_epochs): # n_epoch만큼 반복문 실행
    model.train() # 모델을 훈련모드로 설정
    for image, label, other in train_loader: # 훈련 데이터셋에서 이미지, 레이블, 그 외의 정보를 가져온다.
        data = image.to(device) # 이미지를 GPU에 올리는 작업
        label = label.to(device) # 레이블을 GPU에 올리는 작업
        other_gender = other[1].to(device) # 그 외의 정보(성별)를 GPU에 올리는 작업
        other_race = other[2].to(device) # 그 외의 정보(인종)를 GPU에 올리는 작업
        optimizer.zero_grad() # Optimizer 초기화
        logits = model(data, other_gender, other_race) # 모델에 데이터와 그 외의 데이터(성별, 인종)를 logits를 얻는다.
        loss = criterion(logits, label) # logits과 레이블을 사용하여 loss를 계산한다.
        loss.backward() # 역전파를 통해 gradient를 계산한다.
        optimizer.step() # optimizer를 사용하여 gradient update
        train_loss[e] += loss.item() # 훈련 손실 값에 현재 epoch에서의 손실 값을 더한다.

        ps = F.softmax(logits, dim=1) # softmax를 통해 logits를 확률값으로 변환
        top_p, top_class = ps.topk(1, dim=1) # 가장 높은 확률 값을 가진 클래스를 선택
        equals = top_class == label.reshape(top_class.shape) # 선택한 클래스와 레이블을 비교하여 정확하게 예측한 경우 equals변수에 True를 할당
        # torch.mean 함수로 정확도의 평균을 계산하고 detach() 함수를 통해 gradient 연산을 분리하고 CPU로 이동
        train_acc[e] += torch.mean(equals.type(torch.float)).detach().cpu()

    train_loss[e] /= len(train_loader) # 훈련 데이터셋의 배치 사이즈로 나누어 훈련 데이터셋에 대한 평균 손실 계산
    train_acc[e] /= len(train_loader) # 훈련 데이터셋의 배치 사이즈로 나누어 훈련 데이터셋에 대한 정확도 계산
```

```
with torch.no_grad(): # gradient 연산을 비활성화 하고 메모리 사용량을 줄이는 역할
    model.eval() # 모델을 평가 모드로 설정
    for data, label, other in valid_loader: # 검증 데이터셋에서 이미지와 레이블, 그 외의 데이터(성별, 인종)을 가져온다.
        # 위와 겹치는 주석은 생략하였습니다.
        data = data.to(device)
        label = label.to(device)
        other_gender = other[1].to(device)
        other_race = other[2].to(device)

        logits = model(data, other_gender, other_race) # 모델에 데이터를 넣음으로써 logits 계산
        loss = criterion(logits, label)
        valid_loss[e] += loss.item()

        ps = F.softmax(logits, dim=1)
        top_p, top_class = ps.topk(1, dim=1)
        equals = top_class == label.reshape(top_class.shape)
        valid_acc[e] += torch.mean(equals.type(torch.float)).detach().cpu()

valid_loss[e] /= len(valid_loader)
valid_acc[e] /= len(valid_loader)
scheduler.step() # 한 epoch 당 스케줄러 업데이트

# epoch마다 훈련 및 검증 손실, 훈련 및 검증 정확도를 출력
print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
    e, train_loss[e], valid_loss[e]))
print('Epoch: {} \tTraining accuracy: {:.6f} \tValidation accuracy: {:.6f}'.format(
    e, train_acc[e], valid_acc[e]))
# 검증 손실이 감소할 때마다 모델을 저장
if valid_loss[e] <= valid_loss_min:
    print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss[e]))
    torch.save(model.state_dict(), 'C:/Users/user/PycharmProjects/dl/model/do12.pt') # 모델 저장
    valid_loss_min = valid_loss[e] # 최소 검증 손실(valid_loss_min)을 현재 검증 손실로 업데이트
```

Kfold validation

주로 위의 코드로 모델을 만들었고, public score만 보고 모델의 일반화 성능을 판단 할 수 없었기 때문에 Kfold validation을 진행하여 모델을 검증하였습니다. Kfold validation 코드는 딥러닝기초실습 lab_cifar10.ipynb 템플릿을 기반으로 작성하였습니다.

```
fold_train_losses = []
fold_val_losses = []

valid_loss_min = np.Inf

train_loss = torch.zeros(n_epochs)
valid_loss = torch.zeros(n_epochs)

train_acc = torch.zeros(n_epochs)
valid_acc = torch.zeros(n_epochs)

from sklearn.model_selection import KFold
from torch.utils.data import SubsetRandomSampler
kf = KFold(n_splits=5, shuffle=True)

1 usage
def reset_weights(m): # 가중치 재설정 함수
  for layer in m.children():
   if hasattr(layer, 'reset_parameters'):
    print(f'Reset trainable parameters of layer = {layer}')
    layer.reset_parameters()
```

```python
for fold, (train_ind, valid_ind) in enumerate(kf.split(train_dataset)):
    print('=========================Starting fold = ', fold)

    train_sampler_kfold = SubsetRandomSampler(train_ind) # 해당 폴드에 대한 훈련 데이터 샘플러(Sampler)를 생성
    valid_sampler_kfold = SubsetRandomSampler(valid_ind) # 해당 폴드에 대한 검증 데이터 샘플러(Sampler)를 생성

    # 데이터를 배치 단위로 로드하고, 주어진 배치 크기(batch_size)에 따라 데이터를 분할
    train_loader_kfold = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, sampler=train_sampler_kfold)
    valid_loader_kfold = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, sampler=valid_sampler_kfold)

    # cost 함수, optimizer, scheduler 재정의
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.01)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10, eta_min=0)

    model.apply(reset_weights) # 모델의 가중치 재설정

    # 손실 값, 정확도 값의 초기화
    valid_loss_min = np.Inf

    train_loss = torch.zeros(n_epochs)
    valid_loss = torch.zeros(n_epochs)

    train_acc = torch.zeros(n_epochs)
    valid_acc = torch.zeros(n_epochs)
```

```python
# 훈련을 진행하는 반복문으로 위와 같은 주석은 생략하였습니다.
for e in np.arange(n_epochs):
    model.train()
    for image, label, other in train_loader_kfold:
        data = image.to(device)
        label = label.to(device)
        other_gender = other[1].to(device)
        other_race = other[2].to(device)

        optimizer.zero_grad()
        logits = model(data, other_gender, other_race)

        loss = criterion(logits, label)
        loss.backward()
        optimizer.step()

        train_loss[e] += loss.item()

        ps = F.softmax(logits, dim=1)
        top_p, top_class = ps.topk(1, dim=1)
        equals = top_class == label.reshape(top_class.shape)
        train_acc[e] += torch.mean(equals.type(torch.float)).detach().cpu()

    train_loss[e] /= len(train_loader)
    train_acc[e] /= len(train_loader)
```

```
# 검증을 위한 반복문으로 위와 같은 주석은 생략하였습니다.
with torch.no_grad():
    model.eval()
    for data, label, other in valid_loader_kfold:
        data = data.to(device)
        label = label.to(device)
        other_gender = other[1].to(device)
        other_race = other[2].to(device)

        logits = model(data, other_gender, other_race)
        loss = criterion(logits, label)
        valid_loss[e] += loss.item()

        ps = F.softmax(logits, dim=1)
        top_p, top_class = ps.topk(1, dim=1)
        equals = top_class == label.reshape(top_class.shape)
        valid_acc[e] += torch.mean(equals.type(torch.float)).detach().cpu()

valid_loss[e] /= len(valid_loader_kfold)
valid_acc[e] /= len(valid_loader_kfold)
scheduler.step()

print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
    e, train_loss[e], valid_loss[e]))
print('Epoch: {} \tTraining accuracy: {:.6f} \tValidation accuracy: {:.6f}'.format(
    e, train_acc[e], valid_acc[e]))
if valid_loss[e] <= valid_loss_min:
    print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss[e]))
    torch.save(model.state_dict(), 'C:/Users/user/PycharmProjects/dl/model/bogosetest.pt')
    valid_loss_min = valid_loss[e]
    valid_acc_max = valid_acc[e]
```

Step 6: CNN model training/validation 분석

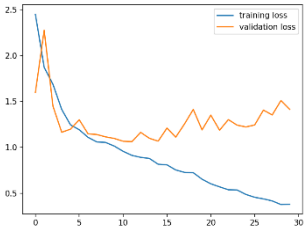
수행한 training + validation 과정을 설명하세요

1차시도 : 강의자료에 제공된 AlexNet 등의 CNN 구조를 기반으로 성능을 내보려고 하였으나 TEST ACC 기준 40후반~50초반 성능이 나왔습니다. 그래서 기본적으로 우리의 테스크에 좋은 성능을 내는 모델에 대한 감을 익히기 위해서 해당 자료(https://pseudo-lab.github.io/pytorch-guide/docs/ch03-1.html)를 참고하여 ResNET 모델로 시도를 하였으나 큰 성능 향상을 이루지 못하였습니다.(48.68%) 그래서 성능 향상이 전혀 없는 상황에서 연산량만 높아지는 크고 깊은 모델은 우리의 테스크에 적합하지 않다는 결론을 내렸습니다.

2차시도 :

```python
class Convnet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 64, 3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(64, 128, 3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(128, 256, 3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(256, 256, 3, stride=1, padding=1)

        self.pool = nn.MaxPool2d(2, stride=2)

        self.bn1 = nn.BatchNorm2d(64)
        self.bn2 = nn.BatchNorm2d(128)
        self.bn3 = nn.BatchNorm2d(256)
        self.bn4 = nn.BatchNorm2d(256)

        self.fc1 = nn.Linear(256 * 4 * 4, 2048)
        self.fc2 = nn.Linear(2048, 5)

        self.fcn1 = nn.BatchNorm1d(2048)

        self.dropout = nn.Dropout(0.5)
        self.relu = nn.ReLU(True)
```

```python
def forward(self, x):
    x = self.pool(self.relu(self.bn1(self.conv1(x))))
    x = self.pool(self.relu(self.bn2(self.conv2(x))))
    x = self.pool(self.relu(self.bn3(self.conv3(x))))
    x = self.pool(self.relu(self.bn4(self.conv4(x))))
    x = x.reshape(-1, 256 * 4 * 4)
    x = self.dropout(x)
    x = self.relu(self.fcn1(self.fc1(x)))
    x = self.dropout(x)
    x = self.fc2(x)
    return x
```



Cifar10 실습시간에 사용하였던 모델이 학습시간이 빠르고 가장 성능이 좋았기 때문에 이를 기반으로 저만의 모델을 구축해 나가려고 노력했습니다. 이 시점에서 저만의 모델을 만들어 나갈 때 다음 세가지를 기준점으로 두었습니다.

1. 성능을 높일 수 있는 기본적인 방법들은 모두 적용한다.
2. 초기 4층짜리 convolution layer와 2층짜리 Linear layer에서 시작하여 모델을 깊게 만든다.
3. Overfitting을 해결한다.

3차시도 :

1. 성능을 높일 수 있는 기본적인 방법들을 모두 적용해보려고 노력하였습니다.

    1-1. 다양한 scheduler를 적용해보는 과정에서 제 모델에 가장 잘 맞았던 것은 CosineAnnealing scheduler입니다. 아래 scheduler를 적용하고 성능이 약 1% 향상됨을 확인하였습니다.

    1-2. 다양한 data augmentation 방법들을 시도하였는데, 성능 향상이 1% 있었던 좌우반전만을 사용하였습니다. 이에 대한 설명은 위의 데이터 전처리 부분에서 설명하였으므로 생략하였습니다.

    1-3. 수업시간에 들었던 교수님의 조언에 따라 인종 정보와 성별 정보도 원핫 인코딩 후 모델에 추가해서 학습시키면 성능이 올라갈 것이라는 생각으로 정보들을 조합해가면서 실험을 진행했습니다. (이미지 데이터만 사용/ 이미지 데이터 + 성별정보/이미지 데이터 + 인종정보/ 이미지데이터 + 성별정보 + 인종정보) 모든 정보를 추가해서 학습시켰을 때, 가장 성능이 좋았기 때문에 모든 정보를 사용하였습니다.

    1-4. 19살과 21살, 29살과 31살처럼 나이의 경계선에 있는 사람들은 구별하기가 어려울 것이라고 생각하였습니다. 그래서 기본적으로 나누어진 5개의 나이대를 예측하는 것이 아닌, 51개의 나이를 예측한 후에 나이대별로 나누면 성능이 올라가지 않을까하여 시도하였으나 성능향상이 없어서 해당 방법을 채택하지는 않았습니다.

    1-5. 학습시간이 많이 걸리지 않았기 때문에, 3차시도때는 grid search나 random search 같은 하이퍼파라미터튜닝을 사용하지는 않고 직접 하나하나 하이퍼파라미터를 튜닝하였습니다.

    1-6. 이때, 앙상블을 진행했으나 아래 사진처럼 단일모델 65%였으나 앙상블을 하면 54%로 하락하였습니다. 이론상으로는 앙상블을 하면 2~5% 정도가 성능이 향상되어야 하는데 10% 이상 성능이 하락한 점(54%)이 이상하다고 생각하였습니다. 이 현상을 해결하는데 시간이 꽤 소요되었으나 4차시도에서 해결하였습니다. 이 부분에 대해서는 4차시도에서 언급하겠습니다.

[앙상블 후, 성능이 10% 이상 하락하는 문제점을 경험]

4차시도

2. 초기 4층짜리 convolution layer와 2층짜리 Linear layer에서 시작하여 모델을 깊게 만들어 가면서 최적의 모델을 찾았습니다. Convolution layer를 4층부터 12층까지 쌓아보았고 반대로 줄여가면서 실험하였습니다. 8층이 가장 적합하였기 때문에 8층으로 고정하였습니다. 이때부터는 앙상블을 위해서 비슷한 모델을 여러 개 만들어서 리더보드에 올려보았고, 이 중에서 가장 높은 정확도를 보이는 모델 위주로 앙상블을 진행해서 최종 제출을 할 수 있었습니다. 또한 public에만 치우친 점수일 수 있다고 판단해서 kfold를 진행해서 검증을 하였습니다. Kfold ensemble을 한 것 보다 약간씩 다른 모델끼리 ensemble을 한 결과가 더 좋다는 것을 실험을 통해서 깨닫고 난 후부터는 kfold를 검증에만 사용하였습니다.

3. Overfitting을 방지하기 위해서 weight_decay와 dropout을 사용해서 규제에 힘을 쓰려고 하였습니다. 강의 자료에 추천 weight_decay (1e-4, 1e-5, 0)을 모두 사용하여 보았습니다. 하지만 제 모델의 경우에는 좀 더 강한 규제가 필요했고 weight_decay를 0.01로 하였을 때 과적합이 어느정도 잡히는 현상을 관찰할 수 있었습니다. 또한 보통은 Fully Connected layer에서 dropout을 사용하지만, 저는 좀 더 규제를 주고 싶었고, 앙상블에서 어려움을 겪고 있었던 상황이였기 때문에 어느정도 앙상블 효과를 볼 수 있다고 생각한 dropout을 convolution layer에도 주었습니다. 이 방법을 사용해서 성능향상 뿐만 아니라 과적합은 처음에 비해서 어느정도 해결할 수 있었다고 생각합니다. 과적합 해결 문제는 5차 시도에서 완전히 잡을 수 있으므로 뒤에서 설명하겠습니다.

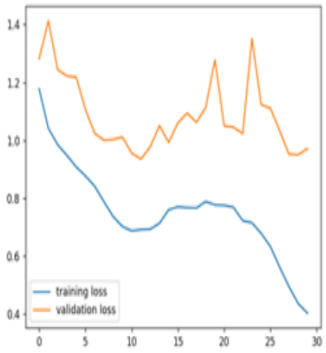
- 모델 구성

8층짜리 convolution layer와 2층짜리 Linear layer를 기본적으로 사용하였습니다. 다음 자료 (https://velog.io/@convin305/%EB%85%BC%EB%AC%B8%EB%A6%AC%EB%B7%B0-dropout%EC%9D%84-%ED%86%B5%ED%95%9C-CNN%EB%AA%A8%EB%8D%B8-%ED%96%A5%EC%83%81%EC%8B%9C%ED%82%A4%EA%B8%B0)를 보고 convolution layer에도 dropout을 써도 된다는 사실을 깨달았고, 직접 적용해 본 후, 효과를 보았기 때문에 다양한 조합으로 dropout을 주어 여러 버전의 모델을 구성하였습니다.

아래는 구성한 모델에 대한 간략한 설명과 kfold validation으로 검증 후의 loss 그래프입니다. 과적합이 대체적으로 1차시도때에 비해서 어느정도 해결되었고, loss도 꾸준히 수렴하는 양상을 보입니다. 대부분 validation loss가 0.9초반~0.8후반까지 떨어졌습니다.

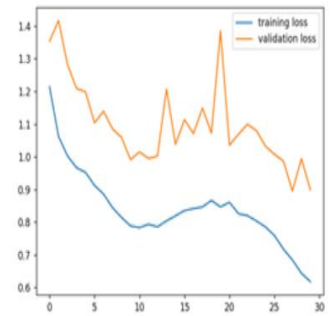
do1 모델 : 7번째 convolution layer에 dropout 0.5 (64.4%)

```
class ConvNet_do1(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 32, 3), nn.BatchNorm2d(32), nn.ReLU(True))
        self.cn2 = nn.Sequential(nn.Conv2d(32, 32, 3, padding=1), nn.BatchNorm2d(32), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn3 = nn.Sequential(nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn4 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn5 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn6 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn7 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.5))
        self.cn8 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.fc1 = nn.Sequential(nn.Linear(266 , 32),
                                 nn.BatchNorm1d(32),
                                 nn.ReLU(True),
                                 nn.Dropout(0.5),

                                 nn.Linear(32, 5))
```



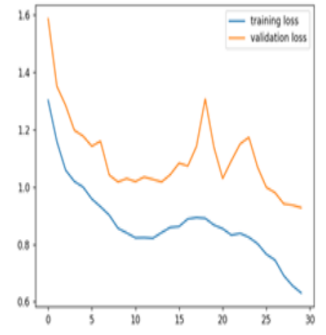
do2 모델 : 6, 7번째 convolution layer에 dropout 0.5 (67.6%)

```
class ConvNet_do2(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 32, 3), nn.BatchNorm2d(32), nn.ReLU(True))
        self.cn2 = nn.Sequential(nn.Conv2d(32, 32, 3, padding=1), nn.BatchNorm2d(32), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn3 = nn.Sequential(nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn4 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn5 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn6 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.5))
        self.cn7 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.5))
        self.cn8 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.fc1 = nn.Sequential(nn.Linear(266_, 32),
                                 nn.BatchNorm1d(32),
                                 nn.ReLU(True),
                                 nn.Dropout(0.5),

                                 nn.Linear(32, 5))
```



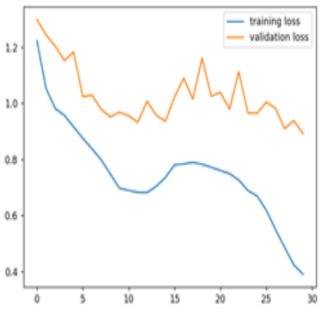
do3 모델 : 6, 7, 8번째 convolution layer에 dropout 0.5 (63.2%)

```
class ConvNet_do3(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 32, 3), nn.BatchNorm2d(32), nn.ReLU(True))
        self.cn2 = nn.Sequential(nn.Conv2d(32, 32, 3, padding=1), nn.BatchNorm2d(32), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn3 = nn.Sequential(nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn4 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn5 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn6 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.5))
        self.cn7 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.5))
        self.cn8 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.5))
        self.fc1 = nn.Sequential(nn.Linear(266, 32),
                                 nn.BatchNorm1d(32),
                                 nn.ReLU(True),
                                 nn.Dropout(0.5),

                                 nn.Linear(32, 5))
```



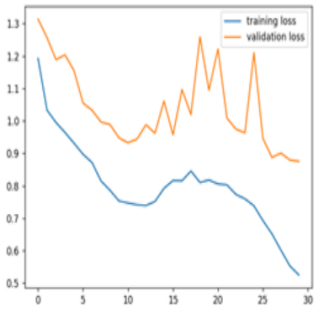
do4 모델 : 7번째 convolution layer에 dropout 0.5 (63.5%)

```
class ConvNet_do4(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 32, 3), nn.BatchNorm2d(32), nn.ReLU(True))
        self.cn2 = nn.Sequential(nn.Conv2d(32, 32, 3, padding=1), nn.BatchNorm2d(32), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn3 = nn.Sequential(nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn4 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn5 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn6 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn7 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.5))
        self.cn8 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.fc1 = nn.Sequential(nn.Linear(266_, 32),
                                 nn.BatchNorm1d(32),
                                 nn.ReLU(True),
                                 nn.Dropout(0.5),

                                 nn.Linear(32, 5))
```



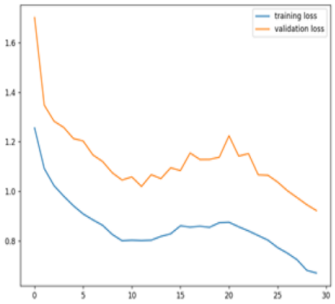
do5 모델 : 5번째 convolution layer에 dropout 0.25, 7번째에 dropout 0.5 (62.7%)

```
class ConvNet_do5(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 32, 3), nn.BatchNorm2d(32), nn.ReLU(True))
        self.cn2 = nn.Sequential(nn.Conv2d(32, 32, 3, padding=1), nn.BatchNorm2d(32), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn3 = nn.Sequential(nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn4 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn5 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.25))
        self.cn6 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn7 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.5))
        self.cn8 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.fc1 = nn.Sequential(nn.Linear(266_, 32),
                                 nn.BatchNorm1d(32),
                                 nn.ReLU(True),
                                 nn.Dropout(0.5),

                                 nn.Linear(32, 5))
```



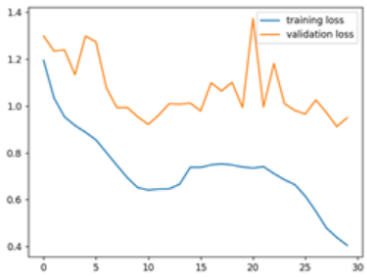
do6 모델 : 모든 convolution layer에 dropout 0.25 (66.1%)

```
class ConvNet_do6(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 32, 3), nn.BatchNorm2d(32), nn.ReLU(True))
        self.cn2 = nn.Sequential(nn.Conv2d(32, 32, 3, padding=1), nn.BatchNorm2d(32), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.25))
        self.cn3 = nn.Sequential(nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.25))
        self.cn4 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.25))
        self.cn5 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.25))
        self.cn6 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.25))
        self.cn7 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.25))
        self.cn8 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.25))
        self.fc1 = nn.Sequential(nn.Linear(266_, 32),
                                 nn.BatchNorm1d(32),
                                 nn.ReLU(True),
                                 nn.Dropout(0.5),

                                 nn.Linear(32, 5))
```



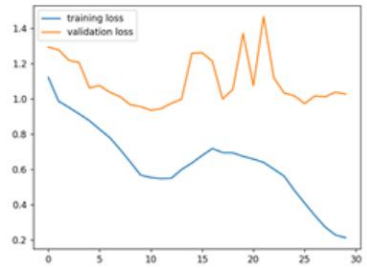
do7 모델 : 모든 convolution layer에 dropout 0.1 (68.6%)

```
class ConvNet_do7(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 32, 3), nn.BatchNorm2d(32), nn.ReLU(True))
        self.cn2 = nn.Sequential(nn.Conv2d(32, 32, 3, padding=1), nn.BatchNorm2d(32), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.1))
        self.cn3 = nn.Sequential(nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.1))
        self.cn4 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.1))
        self.cn5 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.1))
        self.cn6 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.1))
        self.cn7 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.1))
        self.cn8 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.1))
```



do8 모델 : 기본 모델로 convolution layer에 dropout 적용하지 않음. (64.9%)

```
class ConvNet_do8(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 32, 3), nn.BatchNorm2d(32), nn.ReLU(True))
        self.cn2 = nn.Sequential(nn.Conv2d(32, 32, 3, padding=1), nn.BatchNorm2d(32), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn3 = nn.Sequential(nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn4 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn5 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn6 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn7 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn8 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True))
```



- 앙상블을 했을 때, 10% 이상 성능이 하락했던 이유

제가 앙상블을 위한 파일을 하나 더 만들어서 진행했었는데, 이때 test_transform 부분에서 기본적인 데이터 전처리만 진행했어야 하는데 좌우반전 augmentation을 추가한 상태로 앙상블을 진행했기 때문에 성능 하락했다는 점을 발견했습니다. 이 문제를 해결한 후 public score 60점대 후반에서 앙상블 조합에 따라 70점대 초반까지 점수를 올릴 수 있었습니다. Public score의 단일모델 점수와 validation loss를 고려해서 최종적으로 앙상블을 하였을 때 70점의 public score를 낼 수 있었습니다.

5차시도

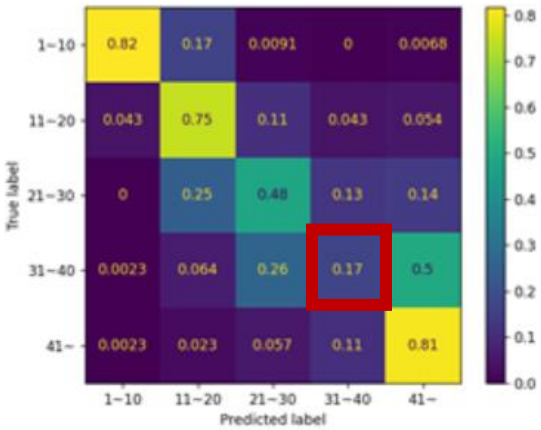
1차시도때부터 4차시도때까지 계속 눈에 걸렸던 부분이 두가지가 있었고 이 부분을 해결하면 성능을 올릴 수 있을 것이라 생각하였습니다.

1. confusion matrix를 뽑아봤을 때, 3번 class를 잘 못맞춘다.
2. validation loss는 0.8이하로 떨어지지 않는다.

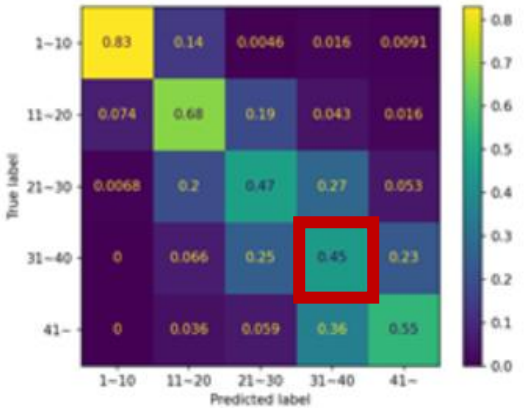
➔ **Focal loss 적용**

우리의 학습데이터에는 클래스별로 거의 비슷한 개수의 데이터가 있어서 클래스 불균형 문제가 있지는 않지만, confusion matrix를 뽑아봤을 때, 3번 클래스가 쉽게 오분류 되는 것을 알 수 있습니다. 4차시도까지 진행하면서 이 방법을 어떻게 해결할 수 있을지에 대해 고민하였고, focal loss를 알게되어 적용하게 되었습니다. Focal loss는 Cross Entropy의 클래스 불균형 문제를 다루기 위해 나왔고 어렵거나 쉽게 오분류 되는 케이스에 대해서 더 큰 가중치를 주는 방법입니다. (개념설명은 https://gaussian37.github.io/dl-concept-focal_loss/를 참고하였고, https://dacon.io/competitions/official/235585/codeshare/1796를 보고 코드구현에 참고하였습니다.)

```python
class FocalLoss(nn.Module):

    def __init__(self, gamma=2.0, eps=1e-7):
        super(FocalLoss, self).__init__()
        self.gamma = gamma # 논문에서 나온대로 2.0으로 설정하였음.
        self.eps = eps
        self.ce = torch.nn.CrossEntropyLoss(reduction="none")

    def forward(self, input, target):
        logp = self.ce(input, target)
        p = torch.exp(-logp)
        loss = (1 - p) ** self.gamma * logp # focal loss 수식
        return loss.mean()

criterion = FocalLoss()
```

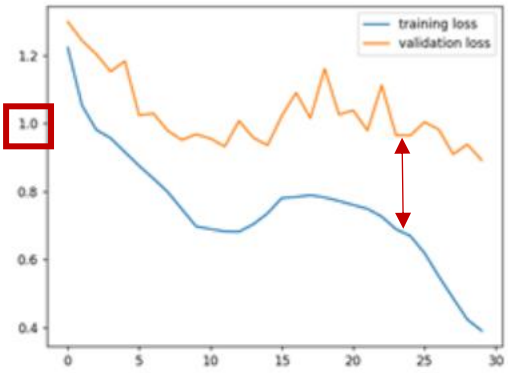


[focal loss 적용 전]

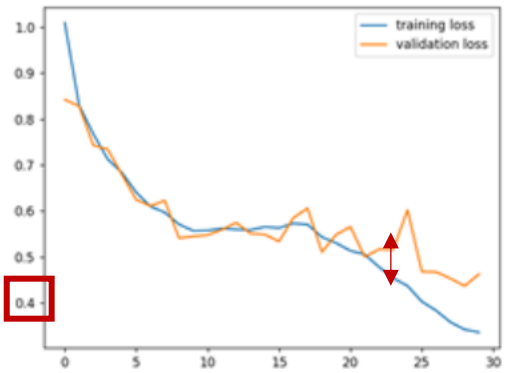


[focal loss 적용 후]

Focal loss 를 적용한 후 confusion matrix를 여러 개 뽑아보면 위 사진과 같이 4번 클래스에 대해서 훨씬 잘 맞추는 양상을 보이는 것을 확인 할 수 있었습니다.



[Focal loss 적용 전]



[Focal loss 적용 후]

또한 loss가 0.8 아래로 떨어지지 않았는데, focal loss를 적용한 후 과적합도 해결되고 0.4까지 떨어지는 양상을 확인할 수 있었습니다. 하지만 CrossEntropyLoss에 비해서 전체적인 정확도 향상에는 효과가 크게 없었기 때문에 최종적으로 사용할 수는 없었습니다.

3. grid search로 하이퍼파라미터 튜닝을 하지 못했다.

➔ **Grid search 적용**

약 3주가량 직접 하나하나 튜닝을 진행하였기 때문에, 사실 대부분 하이퍼파라미터 조합들을 사용해보았다고 생각했습니다. 그래서 grid search를 적용해도 결과는 크게 다르지 않을 것이라 생각하였지만 구현을 해보고 싶어서 직접 구현했습니다. 대회 3일전에 구현을 하게 되어 늦은 감도 있어 random search 까지 구현하지 못한점이 아쉽습니다. 예상대로 직접 튜닝했을때와 같은 결과가 나왔지만 이번에 구현을 했던 경험이 생겼기 때문에 다음번에 사용할 일이 생기면 이번 대회보다 시간 절약을 할 수 있을 것이라 생각이 들고, 다음에는 Random search를 구현해보고 싶습니다.

```
import itertools

# 하이퍼파라미터 조합 정의
learning_rates = [0.001, 0.01, 0.1]
weight_decays = [0.01, 0.001, 0.0001]
gammas = [0.95, 0.9, 0.85]

# 최적의 하이퍼파라미터와 결과 초기화
best_params = None
best_loss = np.Inf

valid_loss_min = np.Inf # 최소 검증 손실을 무한대로 초기화
train_loss = torch.zeros(n_epochs) # 각 epoch에 대한 훈련 손실 값 초기화
valid_loss = torch.zeros(n_epochs) # 각 epoch에 대한 검증 손실 값 초기화

train_acc = torch.zeros(n_epochs) # 각 epoch에 대한 훈련 정확도 값 초기화
valid_acc = torch.zeros(n_epochs) # 각 epoch에 대한 검증 정확도 값 초기화
```

```
# 모든 조합에 대해 반복
for lr, wd, gamma in itertools.product(learning_rates, weight_decays, gammas):
    criterion = FocalLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=gamma)

    n_epochs = 10

    valid_loss_min = np.Inf
    train_loss = torch.zeros(n_epochs)
    valid_loss = torch.zeros(n_epochs)
    train_acc = torch.zeros(n_epochs)
    valid_acc = torch.zeros(n_epochs)

    for e in range(0, n_epochs):
        model.train()
```

```
# 검증 손실이 최소값인 경우 최적의 하이퍼파라미터로 업데이트
if valid_loss[e] < best_loss:
    best_loss = valid_loss[e]
    best_params = (lr, wd, gamma)
```

```
Best Hyperparameters: (0.001, 0.01, 0.9)
```

4. 4차시도에서 가장 성능이 좋은 모델과 유사하게 추가 모델 구축 및 아예 다른 구조의 모델 구축해서 앙상블

   ➔ Kfold 앙상블을 했을때보다 유사한 모델을 생성해서 앙상블했을 때, 최고 성능을 기록할 수 있었습니다. 앙상블을 할때도 다양한 조합으로, 다양한 가중치를 주어 실험하였습니다. 앙상블 코드는 마지막 파트에서 설명하였습니다.

```
class ConvNet_do2_1(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 32, 3), nn.BatchNorm2d(32), nn.ReLU(True))
        self.cn2 = nn.Sequential(nn.Conv2d(32, 32, 3, padding=1), nn.BatchNorm2d(32), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn3 = nn.Sequential(nn.Conv2d(32, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn4 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn5 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn6 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(3, 3), nn.ReLU(True), nn.Dropout(0.55))
        self.cn7 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.55))
        self.cn8 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True))
```

```
class ConvNet_do9(nn.Module):
    def __init__(self):
        super().__init__()
        self.cn1 = nn.Sequential(nn.Conv2d(3, 64, 3), nn.BatchNorm2d(64), nn.ReLU(True))
        self.cn2 = nn.Sequential(nn.Conv2d(64, 64, 3, padding=1), nn.BatchNorm2d(64), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn3 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn4 = nn.Sequential(nn.Conv2d(128, 128, 3, padding=1), nn.BatchNorm2d(128), nn.MaxPool2d(2, 2), nn.ReLU(True))
        self.cn5 = nn.Sequential(nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True), nn.Dropout(0.5))
        self.cn6 = nn.Sequential(nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.MaxPool2d(2, 2), nn.ReLU(True))

        self.fc1 = nn.Sequential(nn.Linear(9226_, 512),
                                 nn.BatchNorm1d(512),
                                 nn.ReLU(True),
                                 nn.Dropout(0.5),

                                 nn.Linear(512, 32),
                                 nn.BatchNorm1d(32),
                                 nn.ReLU(True),
                                 nn.Dropout(0.5),

                                 nn.Linear(32, 5))
```

Step 7: Predict with Test Data

<코드 캡쳐 첨부>

```
classes_cm = [0, 1, 2, 3, 4] # 클래스 레이블을 정의
test_loss = 0 # 테스트 손실 값을 초기화
# 예측된 레이블과 실제 레이블을 저장하기 위한 빈 리스트를 생성
y_pred = []
y_true = []
test_acc = 0 # 테스트 정확도를 초기화
with torch.no_grad(): # 그래디언트 계산을 비활성화
    model.eval() # 모델을 평가 모드로 설정
    for data, labels, other in valid_loader: # validation 데이터로 루프를 반복
        data, labels = data.to(device), labels.to(device)  # 데이터와 레이블을 GPU로 이동

        # 추가 정보(성별 및 인종)도 GPU로 이동
        other_gender = other[1].to(device)
        other_race = other[2].to(device)
        logits = model(data, other_gender, other_race) # 모델에 데이터와 추가 정보를 전달하여 logits 계산
        loss = criterion(logits, labels) # ligits과 실제 레이블 사용해서 loss 계산
        test_loss += loss.item() # test loss 누적

        # 예측된 레이블과 실제 레이블을 비교하여 정확도를 계산하고 누적
        top_p, top_class = logits.topk(1, dim=1)
        y_pred.extend(top_class.data.cpu().numpy())
        y_true.extend(labels.data.cpu().numpy())
        equals = top_class == labels.reshape(top_class.shape)
        test_acc += torch.sum(equals.type(torch.float)).detach().cpu()

    test_acc /= len(valid_loader.dataset) # 정확도를 데이터셋의 크기로 나누어 정규화
    test_acc *= 100 # 정확도를 백분율로 변환
# 예측된 레이블과 실제 레이블을 사용하여 confusion matrix 계산 후 시각화
cm = confusion_matrix(y_true, y_pred, labels=classes_cm, normalize='true')
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes)
disp.plot()
plt.show()
print('Test accuracy : {}'.format(test_acc))
```

해당 코드는 제시된 templete 코드를 그대로 활용하였습니다. 클래스 레이블을 정의한 후, 테스트 손실 값을 초기화 합니다. 예측된 레이블과 실제 레이블을 저장하기 위한 빈 리스트를 생성하고, 테스트 정확도도 초기화합니다. 테스트 데이터에 대해서 예측을 수행하기 위해서 그래디언트 계산을 비활성화 하고 모델을 평가 모드로 설정합니다. 모델에 데이터와 추가 정보를 전달하여 logits값을 얻습니다. Logits과 실제 레이블을 사용하여 손실 값을 계산하고, 테스트 손실 값을 누적합니다. 예측된 레이블과 실제 레이블을 비교하여 정확도를 계산하고 누적합니다. 정확도를 데이터셋의 크기로 나누어 정규화 한 후에 정확도를 백분율로 변환합니다. 예측된 레이블과 실제 레이블을 사용하여 confusion matrix를 계산한 후, 시각화 합니다. 마지막으로 테스트 정확도를 출력합니다.

Step 8: Training Techniques

성능 개선을 위해서 사용한 기법 중에서 특별히 효과적이었던 부분이나 강조하고자 하는 내용을 작성해주세요.

제가 사용한 기법들은 대부분 위에서 상세하게 설명하였기 때문에 기말프로젝트를 진행한 순서에 따른 핵심 위주로 작성하였습니다.

1단계 : 문제 이해 및 환경 설정

가장 기본적인 부분이지만 저는 기말 프로젝트에서 어떤 데이터를 다루는지, 어떤 문제를 다루는지, 어떤 평가지표를 사용하는지에 대해 이해를 완벽히 하고 시작하였습니다.

pretrained model을 사용하면 안되고 직접 만든 모델의 성능을 높여 가는 과정을 거쳐야 했기 때문에 그만큼 다양한 실험을 해보는 것이 중요하다고 생각했습니다. 그래서 모델을 경량화 해야 한다고 느꼈고, 환경 설정 측면에서는 kaggle의 GPU를 활용하는 것 보다 개인 컴퓨터에 RTX3060 이상의 GPU가 있다면 가상환경을 구축해서 개인 GPU를 사용한 것이 좋을 것이라고 생각하였습니다. 그래서 저는 RTX 4090 GPU를 활용해서 가상환경을 구축하여 적은 시간에 다양한 실험을 진행해 볼 수 있었던 점이 도움이 되었던 것 같습니다.

2단계 : 데이터 분석

데이터를 직접 살펴보고 어떤 데이터인지 이해가 선행되어야 한다고 생각했습니다. 데이터의 종류, 데이터의 품질, 데이터의 불균형 여부 등을 살펴보면서 다양한 시도를 해보되, 우리 데이터에 대해 맞춤형 데이터 증강 방식을 찾기 위해 노력했습니다. 전처리가 이미 잘 되어있다고 보여지는 데이터셋이였기 때문에 과한 데이터 증강은 도움이 되지 않을 것이라 예상하였습니다. 실제로 torchvision.transforms 변환기와 Albumentations 변환기를 사용하여 다양한 시도를 해보았지만, 최종적으로 **무작위 좌우반전**의 데이터 증강 방법만이 눈에 띄는 성능 향상을 보였습니다. 클래스별로 데이터의 갯수는 비슷하였기 때문에 nn.CrossEntropy의 매개변수로 들어올 수 있는 label smoothing의 경우 큰 효과를 보지 못할 것 이라고 예상할 수 있었고, 실제로 효과도 없었습니다.

3단계 : 베이스라인 모델 구축

우리 테스크에 맞는 모델이 무엇일지에 대한 감을 익히기 위해 짧은 시간내에 얕은 모델부터 깊은 모델까지 시도를 해본 점이 감을 잡는데 도움이 되었습니다. Step6에서 아주 자세하게 제가 어떠한 시행착오를 거쳐서 베이스라인 모델을 설계하였는지에 대한 내용을 담았기 때문에 이에 대한 내용은 생략하였습니다. 이에 8층짜리 Convolution layer와 2층짜리 Linear layer로 이루어진 가장 기본적인 베이스라인 모델을 설계를 해놓고, 좋은 성능이 나오는 단일 모델을 발견하면 살을 붙혀가면서 앙상블을 위한 모델들을 다양하게 설계하였습니다. 모델을 설계하면서 성능 향상에 도움이 되었던 부분은 다음과 같습니다.

[많은 경우에 공통적으로 효과적일 것이라고 생각하는 방식입니다.]

3-1. Batch Normalization 적용

➔ Batch Normalization은 각 Batch의 입력을 Normalization하여 각 층의 입력 분포가 일정하게 해줌으로써 학습 과정을 안정화 시킵니다. 또한 Backpropagation 과정에서 gradient 소실 문제가 발생할 수 있는데 이 방법을 통해서 gradient의 크기를 안정화 시킴으로써 gradient 소실 문제를 방지해줍니다.

3-2. Activation function으로 ReLU 적용

➔ ReLU는 딥러닝 모델에서 성능을 향상시키기 위해서 일반적으로 사용하는 활성화 함수입니다. ReLU를 사용함으로써 입력이 0보다 작을때는 0을 출력하고, 입력이 0보다 크거나 같을 때는 입력 값을 그대로 출력합니다. 이 비선형적인 특성은 딥러닝 모델이 비선형 관계를 학습할 수 있도록 해줍니다. 또한 Backpropagation 과정에서 gradient update시 gradient가 소실되지 않게 도움을 줍니다.

3-3. Fully Connected layer에 dropout 0.5 적용

➔ Dropout을 통해 모델의 복잡도를 줄여서 과적합을 방지할 수 있고 앙상블 효과를 주면서 일반화 성능을 높일 수 있습니다. 0.5로 설정하였을 때, 가장 좋은 성능을 낼 수 있었습니다.

3-4. 이미 유명한 모델 구조(VGG)를 차용한 후, 이를 기반으로 나만의 모델 구축

➔ 기존 모델의 검증된 성능을 활용함으로써 초기 단계에서의 모델 구축에 대한 불확실성을 줄일 수 있습니다. 나만의 모델을 구축하는 과정에서 이미 유명한 모델 구조를 차용하여 우리의 테스크에 맞게 수정하는 과정이 도움이 되었습니다.

3-5. Adam 계열 optimizer 사용 (Adam, AdamW, Adamax)

➔ Adam 계열 optimizer는 딥러닝 모델에서 성능을 향상시키기 위해서 일반적으로 사용합니다. 최종적으로는 Adam optimizer를 사용하였지만, AdamW와 Adamax도 시도하는 과정에서 안정적인 학습과 높은 성능을 도출하였습니다.

3-6. 모든 데이터셋을 학습에 사용

➔ 7:3으로 data를 split하여 검증한 후에 만들어진 모델을 믿고 전체 학습 데이터를 사용해서 최종 모델을 학습시켰던 것이 성능 향상에 도움이 되었습니다. 직관적으로도 모든 데이터를 사용해서 학습시키는 것이 더 잘 학습시킬 수 있다고 생각해볼 수 있었습니다. 특히 우리의 데이터셋의 양이 많지 않았기 때문에 최대한 많은 데이터로 학습시키는 것이 성능에 도움이 되었을 것입니다.

3-7. kfold validation으로 일반화 성능 향상, kfold ensemble

➔ 5개의 서로 다른 폴드로 나누고, 각 폴드를 검증 데이터로 사용하여 모델을 평가하는 방식을 사용하였습니다. 이러한 방식으로 public dataset에 치우쳐진 점수인지, 아닌지를 대략적으로 알 수 있었습니다. 각 폴드에서 모델을 학습하고, 다른 폴드에서 모델을 검증하는 과정을 k번(5번) 반복하므로 모델의 성능을 보다 신뢰할 수 있었습니다. 제가 최종적으로 사용한 앙상블 방법은 3-8 방법이였지만, kfold ensemble 방법도 성능향상에 도움이 되었습니다.

```
if valid_loss[e] <= valid_loss_min:
    print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss[e]))
    torch.save(model.state_dict(), 'C:/Users/user/PycharmProjects/dl/model/bogosetest.pt')
    valid_loss_min = valid_loss[e]
    if fold == 0:
        torch.save(model.state_dict(), 'C:/Users/user/PycharmProjects/dl/model/redo2_fold0.pt')
    elif fold == 1:
        torch.save(model.state_dict(), 'C:/Users/user/PycharmProjects/dl/model/redo2_fold1.pt')
    elif fold == 2:
        torch.save(model.state_dict(), 'C:/Users/user/PycharmProjects/dl/model/redo2_fold2.pt')
    elif fold == 3:
        torch.save(model.state_dict(), 'C:/Users/user/PycharmProjects/dl/model/redo2_fold3.pt')
    else:
        torch.save(model.state_dict(), 'C:/Users/user/PycharmProjects/dl/model/redo2_fold4.pt')
    valid_acc_max = valid_acc[e]
```

3-8. 정확도가 높은 모델끼리 가중치를 주어 앙상블 진행

➔ Kfold ensemble을 했을 때 보다, 조금씩 다르게 만든 모델끼리 앙상블을 진행했을 때 앙상블 효과가 더 좋았습니다. 또한 성능이 좋은 모델에 더 가중치를 주어 앙상블을 진행했을 때, 더욱 도움이 되었습니다. 제가 다양한 조합으로 실험을 하였기 때문에 코드가 길지만, 최종 제출에 사용한 앙상블 모델은 5가지 였습니다. 다양하게 실험을 하다보니 최고 기록을 달성한 71% 앙상블 조합을 잃어버려서 마지막 제출(70.8%)시 사용한 앙상블코드를 제출하였습니다.

➔ Softmax ensemble을 사용하고 각각의 가중치를 주어서 실험하는 부분은 구현하기 쉬웠기 때문에 아래 코드와 같이 직접 구현해보았습니다. 앙상블 실험에 사용할 모든 모델 구조를 불러온 후, 모델을 GPU에 올립니다. 저장되어있는 모델을 로드하고 그 후부터는 일반적인 test 방법과 동일하게 구현하였습니다. 그 후 logits을 구할 때는 원하는 조합으로 더해주고 모델에 따라 가중치를 다양하게 주면서 실험을 진행하였습니다.

```
# 앙상블 실험에 사용할 모든 모델 구조 불러오기 # 모델을 GPU에 올리기
model_1 = ConvNet_do1()                  model_1.to(device)
model_2 = ConvNet_do2()                  model_2.to(device)
model_3 = ConvNet_do3()                  model_3.to(device)
model_4 = ConvNet_do4()                  model_4.to(device)
model_5 = ConvNet_do5()                  model_5.to(device)
model_6 = ConvNet_do6()                  model_6.to(device)
model_7 = ConvNet_do7()                  model_7.to(device)
model_8 = ConvNet_do8()                  model_8.to(device)
model_9 = ConvNet_do9()                  model_9.to(device)
model_2_1 = ConvNet_do2_1()              model_2_1.to(device)
model_2_2 = ConvNet_do2_2()              model_2_2.to(device)
```

```
# 저장되어 있는 모델(.pt) load
model_1.load_state_dict(torch.load('C:/Users/user/PycharmProjects/dl/model/do1.pt'))
model_2.load_state_dict(torch.load('C:/Users/user/PycharmProjects/dl/69best_model_0609/do_gh/do2.pt'))
model_3.load_state_dict(torch.load('C:/Users/user/PycharmProjects/dl/model/do3.pt'))
model_4.load_state_dict(torch.load('C:/Users/user/PycharmProjects/dl/model/do4.pt'))
model_5.load_state_dict(torch.load('C:/Users/user/PycharmProjects/dl/model/do5.pt'))
model_6.load_state_dict(torch.load('C:/Users/user/PycharmProjects/dl/model/do6.pt'))
model_7.load_state_dict(torch.load('C:/Users/user/PycharmProjects/dl/69best_model_0609/do_gh/do7.pt'))
model_8.load_state_dict(torch.load('C:/Users/user/PycharmProjects/dl/69best_model_0609/do_gh/do8.pt'))
model_9.load_state_dict(torch.load('C:/Users/user/PycharmProjects/dl/69best_model_0609/do_gh/do9.pt'))
model_2_1.load_state_dict(torch.load('C:/Users/user/PycharmProjects/dl/69best_model_0609/do_gh/do2-1.pt'))
model_2_1.load_state_dict(torch.load('C:/Users/user/PycharmProjects/dl/model/do2-2.pt'))
```

```
id_list = []
pred_list = []
with torch.no_grad():
    # 모델을 평가모드로 설정
    model_1.eval()
    model_2.eval()
    model_3.eval()
    model_4.eval()
    model_5.eval()
    model_6.eval()
    model_7.eval()
    model_8.eval()
    model_9.eval()
    model_2_1.eval()
    model_2_2.eval()

    for data, file_name, other in test_loader:
        data = data.to(device)

        other_gender = other[0].to(device)
        other_race = other[1].to(device)

        # 각 모델마다 logits 값 구하기
        logits_model1 = model_1(data, other_gender, other_race)
        logits_model2 = model_2(data, other_gender, other_race)
        logits_model3 = model_3(data, other_gender, other_race)
        logits_model4 = model_4(data, other_gender, other_race)
        logits_model5 = model_5(data, other_gender, other_race)
        logits_model6 = model_6(data, other_gender, other_race)
        logits_model7 = model_7(data, other_gender, other_race)
        logits_model8 = model_8(data, other_gender, other_race)
        logits_model9 = model_9(data, other_gender, other_race)
        logits_model2_1 = model_2_1(data, other_gender, other_race)
        logits_model2_2 = model_2_2(data, other_gender, other_race)
```

```
        # 다양한 모델 조합과 가중치를 주어 실험하기
        logits = logits_model2_1 * (0.16) + logits_model2 * (0.23) + logits_model7 * (0.25) + logits_model8 * (0.14) + logits_model9 * (0.20)

        ps = F.softmax(logits, dim=1)
        top_p, top_class = ps.topk(1, dim=1)

        id_list += list(file_name)
        pred_list += top_class.T.tolist()[0]

handout_result = pd.DataFrame({'Id': id_list, 'Category': pred_list})
handout_result.to_csv('C:/Users/user/PycharmProjects/dl/csv/last_try.csv', index=False)
```

3-9. 하이퍼파라미터 튜닝

➔ 저는 직접 하이퍼파라미터 튜닝을 한 후에, 대회 끝무렵 grid search 하이퍼파라미터 튜닝을 진행하였습니다. 직접 해보는 것 보다 효율적이라고 생각하였고 시간이 더 많이 주어진다면 random search를 통한 튜닝이 효과가 좋을 것이라고 생각하여 명시하였습니다.

[아래는 일반적인 방법인지는 잘 모르겠지만, 제 모델의 경우에 효과를 본 방법입니다.]

3-10. fully connected layer 외에 convolution layer에도 dropout 적용

➔ 실험결과, 과적합 문제로 validation loss이 0에 수렴하지 않는 현상을 관찰하였습니다. 이 문제를 해결하기 위해서 일반화 성능을 향상시키기 위한 방법으로 convolution layer에도 dropout을 적용하면 어떨까라는 생각이 들었습니다. 실제로 Dropout을 다양한 조합으로 convolution layer에 적용해보면서 성능 향상이 뚜렷하게 나타남을 확인할 수 있었습니다. 일반적으로 "convolution with dropout"에 대한 검색결과는 Dropout을 convolution에서 사용하지 말아야 한다는 내용이 많이 나오지만, 우리의 데이터셋과 테스크에는 Dropout을 convolution에 적용하는것이 잘 맞을 수 있다는 결론을 도출하였습니다. 또한 제가 앙상블 기법을 사용하였을 때, 성능이 저하되는 문제를 경험하였는데, Dropout을 사용하면 훈련과정에서 앙상블 효과를 어느정도 대체할 수 있어서 모델 성능 향상에 큰 도움을 주었습니다.

3-11. Fully Connected layer의 input dim을 작게 설정

➔ 실험을 통해 확인된 바에 따르면, Fully connected layer의 입력값이 너무 크면 모델이 성능이 저하될 수 있음을 알게되었습니다. 따라서 입력값을 적절하게 제어하여 안정적인 성능을 보이는 범위를 찾기 위해 다양한 실험을 진행하였습니다. 이를 위해 MaxPooling의 stride와 padding을 2로 설정하여 이미지의 크기를 절반으로 줄여가며, 이를 통해 Fully Connected layer의 입력값을 작게 유지할 수 있도록 구성하였습니다. 이렇게 함으로써 fc layer의 입력값을 200~300 정도로 제한하여 모델의 성능을 최적화하였습니다.

3-12. 마지막 Linear layer의 입력을 32로 설정

➔ 클래스 개수가 10개 미만인 경우에, 마지막 Linear layer의 입력값으로 32가 적절하다는 자료를 참고하였습니다. 따라서 우리의 모델에서도 클래스 개수에 맞게 마지막 Linear layer의 입력값을 32로 고정을 시키고 실험을 진행하였습니다. 이를 통해 클래스 개수에 최적화 된 모델을 구축할 수 있었습니다.

3-13. weight_decay를 강하게 적용

➔ 강의안에 따르면 보통 추천되고 있는 weight_decay의 수는 1e-4, 1e-5, 0입니다. 이를 모두 적용해보았음에도 과적합이 해결되지 않아서 제 모델의 경우에는 더 큰 규제가 필요하다고 생각하여 0.01로 적용하였습니다.

3-14. loss가 높더라도 성능 좋은 모델을 선택

➔ 단일 모델의 성능이 67~68% 이더라도 loss는 0.8 이하로 떨어지지 않는 문제점과 3번 클래스에 대해 예측 확률이 매우 낮은 문제점을 겪었습니다. Focal loss를 최종 제출시 사용하지 않았지만, 위 문제에 대해 효과를 보았기 때문에 명시하였습니다. Focal loss를 적용해서 loss는 0.3까지 떨궈서 매우 안정적으로 수렴하는 loss 그래프를 만들 수 있었습니다. 또한 3번 클래스에 대해 기존의 예측 확률은 잘나와도 30%를 넘지 못하였는데 해당 loss를 사용한 후에 45%까지의 정확도를 보였습니다. 물론 loss도 낮고 accuracy도 높으면 좋겠지만, loss가 0에 수렴하지 않더라도 성능이 높은 모델을 선택한 것이 도움이 되었습니다. 하지만 반대로 말하자면 loss를 줄일 수 있고, 어려운 클래스에 대해 예측률을 높이는데에는 Focal loss가 효과적이였음을 알 수 있었습니다.

3-15. epoch수는 30으로 설정

➔ 여러 번 학습을 시키다 보니 대략적으로 과적합이 나는 포인트를 알게되었습니다. 30번째 epoch을 기준으로 공통적으로 과적합이 발생함을 확인할 수 있었고 이에 적절한 epoch수를 설정하였습니다. 물론 early stopping을 구현해도 되었지만, 적절한 epoch을 설정하는 방법을 선택하였습니다.

마지막으로, 나이 분류기를 2개 만들어서 학습을 진행한 시도를 하였습니다. 하나의 예측기로 분류를 하는 것보다, 남자 데이터셋으로 학습시킨 남성 나이 예측기 1개와 여성 데이터셋으로 학습시킨 여성 나이 예측기 1개를 따로 학습시키는게 성능에 도움이 될 것이라 생각하였습니다. 보고서 분량상 모든 내용을 담지는 못했지만, 여성 데이터셋과 남성 데이터셋을 나눈 상태로 따로 학습을 진행하였습니다. 예시 코드로 남성 나이 예측기에 대한 코드를 첨부하였습니다.(모델 구조는 위의 모델과 같습니다.) 유의미한 성과를 보이지는 못해서 아쉽지만(성능은 하나의 예측기로 분류하였을때와 유사하였습니다.), 또 다른 시각에서 다른 방법으로 진행한 시도였기 때문에 간략하게 보고서에 담았습니다.

```
'''
man 나이 예측기
'''
import ...

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# print(device)

train_dataframe = pd.read_csv('./data.csv')
train_dataframe = train_dataframe[train_dataframe["Gender"] == 0]

test_df = pd.read_csv('./testdata.csv')
test_df = test_df[test_df["Gender"] == 0]

# 결과 출력
print(test_df)

from sklearn.model_selection import train_test_split

train_df, valid_df = train_test_split(train_dataframe, shuffle=True, test_size=0.01, stratify=train_dataframe['Label'])


class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, dataframe, train='train', transform=None):
        if train == 'train':
            self.image_list = []
            self.label_list = []
            self.other_list = []
            path = './man_dataset/{}/{}'
            for index, row in dataframe.iterrows():
```