딥러닝 기초 기말 Project

학번: 20185124

이름: 나영훈

**# 서약**

아래 보고서는 **본인의 힘만으로 작성**해야 하며, 다른 학생에게 질문과 다른 학생의 코드를 참고 하는 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: 20185124

이름: 나영훈

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

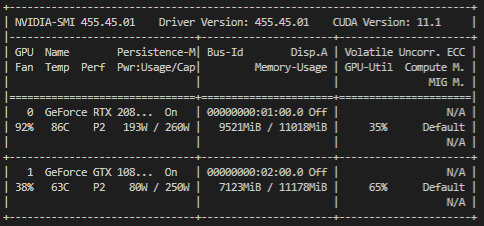
\*모든 코드에는 주석을 작성해 주세요

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.

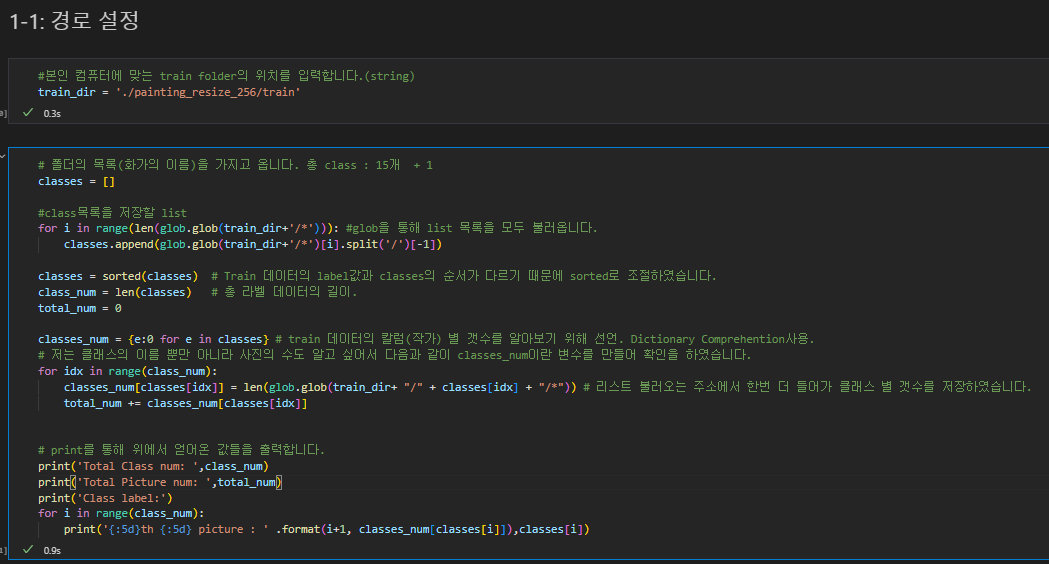
Step 0: 개발 환경 (IDE)

* Python : 3.9.7
* CUDA : 11.3
* GPU : 2080TI, 1080TI (2개의 GPU사용)



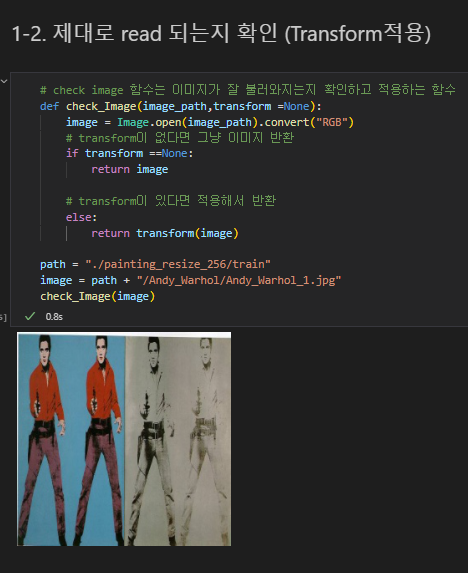
Step 1: Dataset 준비하기

1. 경로 설정

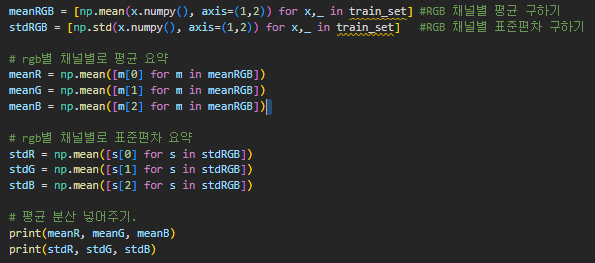


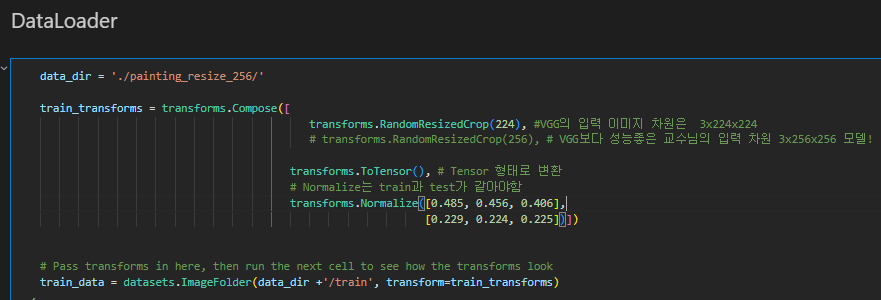
저는 Piet Modrian의 예측 확률이 떨어진다고 생각하여 Piet\_Modrian의 그림을 두 분류로 나눴습니다. 따라서 클래스의 수가 16개가 나옵니다. (성능에 유의미한 영향은 없었습니다).

1-2 제대로 Read 되는지 확인 Transform 적용

 Transform이 잘 적용되는지 확인합니다. 처음에는 기본값만 들어갔으므로 일반 사진을 반환합니다.

원래는 GrayScale이 잘 적용되는지 확인하는 칸이 나오지만 보고서 분량이 많다고 생각하여 생략하였습니다.

Step 2: Dataset 에 대한 Data Loaders 구성

저는 다음과 같이 train\_set의 채널별 평균, 표준편차를 구해 적용하는 방법을 사용하려 하였습니다. 하지만 아래의 방법과 비교하였을 때 시간은 오래 걸리나 성능 차이는 별로 없어서 주석 처리하고 아래 코드를 사용하였습니다.   


transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])

이 코드의 수치는 많은 데이터를 가지고 있으며 모델의 성능평가에 쓰이는 ImageNet 데이터 셋에서 뽑아낸 평균과 표준편차이므로 웬만한 사진 데이터에 적용해도 좋은 성능이 나온다고 생각하여 위의 데이터를 사용하겠습니다.

데이터 전처리 중 DataLoader

저는 Over Sampling을 사용하기 위해 Data Augmentation을 사용하였습니다. 해당 부분에 대한 설명은 Step 8에서 자세히 설명하겠습니다. 따라서 이번 코드에서는 Data Loader에 적은 코드만 설명하도록 하겠습니다.

transforms.RandomResizedCrop(224), #VGG의 입력 이미지 차원은  3x224x224

해당 코드는 VGG11의 코드를 구현하기 위해 다음과 같이 잘라주었습니다.

# transforms.RandomResizedCrop(256), # VGG보다 좋은 실습 모델3x256x256 모델

처음에는 3층짜리 Convolution Layer 를 사용하였습니다. 따라서 실습에서 사용한 그대로 256 x 256 크기의 사진을 사용하였으므로 그대로 256x256을 하였습니다. 나중에는 Rotation을 돌리면서 검은색 영역을 자르기 위해 위처럼 224로 잘라주는 작업을 하였습니다.

transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])

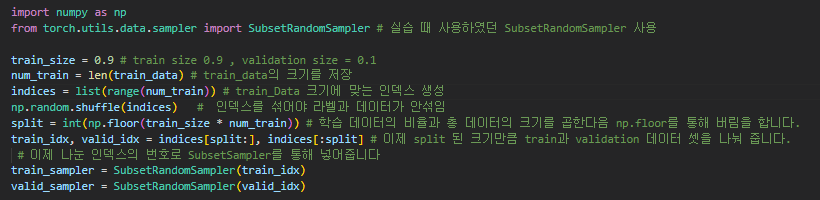
이 코드는 ImageNet 데이터 셋에서 뽑아낸 평균과 표준편차로 많은 이미지에 대한 평균과 표준편차이므로 여기서도 잘 동작하겠다고 생각하였습니다. 데이터 Normalize, 즉 정규화를 하는 이유는 데이터를 정규분포에 가까운 형태로 만들어서 사용하면 더 학습이 잘 되기 때문입니다.

transforms.ToTensor(), # Tensor 형태로 변환

transfors.ToTensor() 를 통해 Tensor 형태로 변환해줍니다.

데이터 전처리 중 Train / Valid 분리

Stratifie\_Kfold를 사용하여 밑에서 만들어도 되는 부분이지만 여기서 설명하고 넘어가겠습니다.



Train set size : 90%

Valid set size : 10%

학습 데이터 셋과 검증 데이터 셋은 위의 비율로 나눠 주었습니다.

실습 시간에 사용한 SubsetRandomsampler를 사용하여 인덱스 번호를 넣어서 나눠주었습니다.

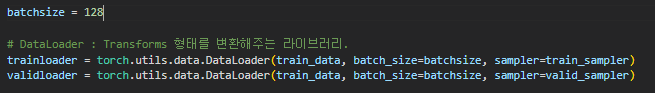
인덱스를 나눠야 하는 까닭은 인덱스가 아닌 학습 데이터와 label을 모두 랜덤으로 섞어서 나누면 섞이기 때문에 index를 만들어 인덱스를 섞은 후 SubsetRandomsampler에 넣어줍니다.

배치 사이즈는 128로 해줍니다. 128과 같이 크게 잡은 이유: 저는 데이터를 병렬 처리하여 GPU에 넣어주었습니다. 이때 하나의 GPU가 가지는 batchsize는 batchsize/gpu\_number입니다. 저는 데이터가 편향된 상황에서 어떻게 학습을하여야 가장 효과적인 성능을 낼지 고민하였습니다. **하나의 배치에는 데이터 수가 많은 클래스가 상대적으로 많을 것이 분명**하며 Loss를 개선할 때도 상대적으로 **많은 클래스의 Loss에 따라 Loss가 개선**되겠다고 생각했습니다. 따라서 배치 사이즈를 크게 하고 상대적으로 적은 수의 샘플을 늘려서 Loss가 모든 클래스에 대해 개선이 잘되도록 해야겠다고 생각했습니다. 클래스 수가 16개(몬드리안 추가)이므로, 예를 들어 배치 사이즈가 16이라면 StratifiedKfold를 사용한다고 하더라도 반 고흐와 같은 데이터는 3, 4개가 들어갈 것이며, 잭슨 폴락, 클라우디오 모네, 에두아르도 마네 등과 같이 샘플 수가 적은 데이터들은 들어간다 하더라도 Loss 개선에 큰 영향을 주지 못할 것으로 생각하였습니다. 결국에는 epochs를 돌며 학습이 되겠지만 전체적으로 Loss 개선은 샘플 수가 많은 곳에 치우칠 것이며 이는 샘플 수가 적은 데이터들의 학습 성능 저하를 초래하겠다고 생각하였습니다. 저는 StratifiedKfold를 통해 데이터가 편향의 문제를 조금이나마 처리해 주었다고 생각했습니다. 하지만 StratifiedKfold를 쓴다고 해결이 되겠다고 생각하지 않았으며 배치 사이즈와 낮은 샘플 수를 동시에 늘려야겠다고 생각하였습니다.

수업 시간에서도 배치 사이즈를 작게 함으로써 Local Minima를 탈출할 수 있다고 하였습니다. 따라서 16보다는 큰 32나 64를 배치 사이즈로 잡고 학습을 진행하였습니다.

GPU당 64의 batchsize로 학습시키고 싶었으며 제가 사용하는 GPU는 2개 이므로 128으로 넣어주었습니다.

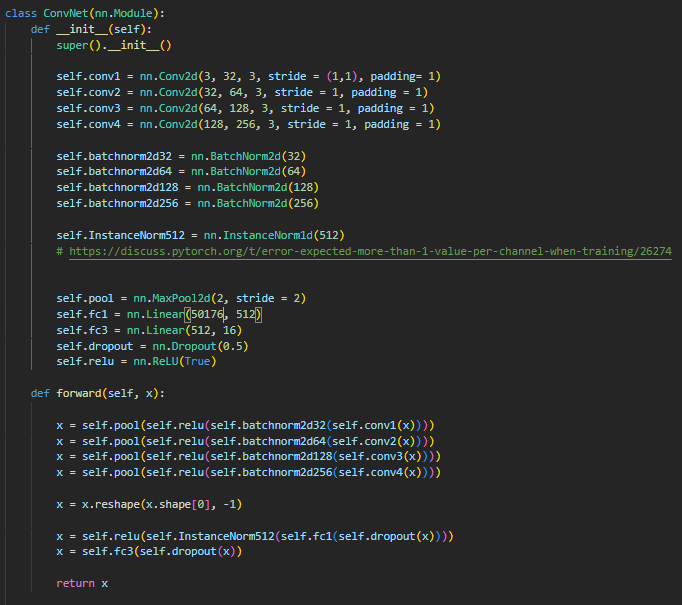
* 처음에는 128로 진행하다가 후반부에 들어서 여러 개의 모델을 사용하다 보니 메모리 초과 오류인 CUDA LAUNCH\_BLOCKING 에러가 떴습니다. 따라서 배치 사이즈를 64로 하여 하나의 GPU가 32개씩 처리하게 하였습니다.



Step 3: Neural Network 생성

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

초기 사용한 4층짜리 Convolution Model



초기에 사용한 4층짜리 Convolution Model은 교수님께서 실습 시간에 사용하신 모델입니다.

복잡한 모델인 VGG16을 먼저 사용하여 모델을 구성하였으나 모델이 너무 복잡하고 커서 학습이 되질 않았습니다. 그 후 단순한 모델인 LeNet, AlexNet 등 여러 모델을 직접 구현해보았습니다. 결과는 교수님께서 실습 시간에 사용하신 모델이 가장 성능이 높게 나왔습니다. 따라서 실습 때 사용한 모델에 한 층씩 쌓아 올리는 방법을 사용하였습니다. 교수님께서 사용한 4층짜리 단순 모델이 성능이 더 좋아서 초기 틀을 4층짜리 Convolution Model을 사용하였습니다.

입력 데이터 차원

[128(batch size) / 2(gpu\_num) , 3, 224, 224] -> [64, 3, 224, 224]

혹은 데이터가 배치 사이즈로 나눠 떨어지지 않을 경우 [-1, 3, 224, 224]

여기서 배치 사이즈는 -1로 표기하겠습니다.

설계한 모델 출력 후 네트워크를 구성한 방법과 이유를 각 단계별로 설명

ConvNet의 차원 분석

|  |  |
| --- | --- |
| 입력 데이터 | [-1, 3, 224, 224] |
| Conv2d\_Layer1 | [-1, 32, 224, 224] |
| BatchNorm2d\_Layer1 | [-1, 32, 224, 224] |
| ReLU\_Layer1 | [-1, 32, 224, 224] |
| MaxPool2d\_Layer1 | [-1, 32, 112, 112] |
| Conv2d\_Layer2 | [-1, 64, 112, 112] |
| BatchNorm2d\_Layer2 | [-1, 64, 112, 112] |
| ReLU\_Layer2 | [-1, 64, 112, 112] |
| MaxPool2d\_Layer2 | [-1, 64, 56, 56] |
| Conv2d\_Layer3 | [-1, 128, 56, 56] |
| BatchNorm2d\_Layer3 | [-1, 128, 56, 56] |
| ReLU\_Layer3 | [-1, 128, 56, 56] |
| MaxPool2d\_Layer3 | [-1, 128, 56, 56] |
| Conv2d\_Layer3 | [-1, 256, 28, 28] |
| BatchNorm2d\_Layer3 | [-1, 256, 28, 28] |
| ReLU\_Layer3 | [-1, 256, 28, 28] |
| MaxPool2d\_Layer4 | [-1, 256, 28, 28] |
| Dropout\_Layer4 | [-1, 50176] |
| Linear\_Layer5 | [-1, 512] |
| InstanceNorm1d\_Layer5 | [-1, 512] |
| ReLU\_Layer5 | [-1, 512] |
| Dropout\_Layer5 | [-1, 512] |
| Linear\_Layer6 | [-1, 16] (출력) |

우선 단순한 모델을 만들기 위해 Conv layer가 4층 + linear Layer 2층짜리 모델을 만들었습니다.

Convolution Layer는 kernel size= (3,3)이며 stride = 1, padding = 1로 주어 크기에 변함이 없게 하였습니다. (image size + 2 x padding – filter\_size) / stride + 1. 즉 ((224 + 2x1 – 3) / 1) + 1 = 224이므로 이미지에 변함이 없습니다.

커널 사이즈를 3으로 한 이유는 커널 사이즈를 작게 하여 최대한 깊게 모델을 쌓고 싶었기 때문입니다. 또한 GoogleNet, AlexNet의 (5,5), (7,7) .. ect 커널과 다르게 (3,3)으로 작게 여러 번 가져가서 큰 필터를 한번 사용한 것과 동일한 효과, 더 좋은 효과를 가져오며 학습해야 할 파라미터 수를 줄이고 싶었습니다.

<https://www.youtube.com/watch?v=Adi0Iasehj8&t=11857s> 를 정독하던 중 Vgg 분석을 보면서 위와 같은 형식으로 모델을 가져가기 위해 노력하였습니다.

Maxpooling에 kernel size = 2, stride = 2를 주어 이미지의 크기가 Maxpooling Layer를 거칠 때마다 크기가 절반이 되게 하였습니다.

BatchNormalization은 뒤에서 설명하겠습니다. (차원의 변화는 없습니다.)

Activation Function은 ReLU를 사용하였습니다. ReLU를 일반적으로 많이 사용하였기 때문에 사용하였습니다. Activation Function의 사용은 차원 변화가 없습니다.

Conv Layer가 끝난 후 Linear Layer를 통과하게 되는데 이때 모델의 출력이 [-1, 256, 28, 28] 이므로 [-1, 50176]으로 차원을 축소합니다.

저는 클래스 중 Piet\_Mondrian을 두 개로 나눠주었으므로 16개의 클래스가 나와야 합니다. 따라서 출력층의 결과는 [-1, 16] 이 나오게 됩니다.

Weight 초기화 방법은 kaiming\_normal을 사용하였습니다. ReLU를 사용할 때는 kaming을 사용한다고 하셨기 때문에 default인 kaiming\_uniform 대신 kaiming\_normal을 사용하였습니다.

* Weight initalization을 하지 않는다면 tanh에서는 Layer를 지날수록 0으로 수렴하는 모습을 보이며 기울기 소실 문제가 생기게 된다. 하지만 Weight initalization을 통해 평균과 표준편차를 유지해 준다면 학습이 더 잘 될 것이라 생각했습니다.
* Weight initalization을 사용하여 큰 성능 향상은 보지 못했습니다. 모델의 높이가 낮아서 큰 효과를 보지 못했다고 생각합니다. 다만 Weight initalization을 사용하였을 때 Loss값이 조금 더 떨어지는 것을 보았으며 제출 시 1% 정도의 성능 향상을 보였습니다.

최종 Layer에서 사용한 Activate Function은 softmax입니다. 분류 문제에서의 마지막 Linear Layer에 들어가는 function은 Cross Entropy를 배웠기 때문에 softmax를 사용하였습니다.

즉 ConvNet은 (3,224,224) 형태의 이미지를 입력받아 (-1, 16) 형태의 텐서를 반환합니다.

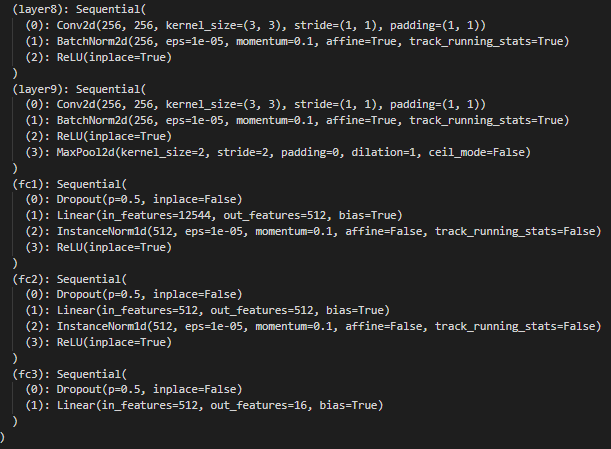
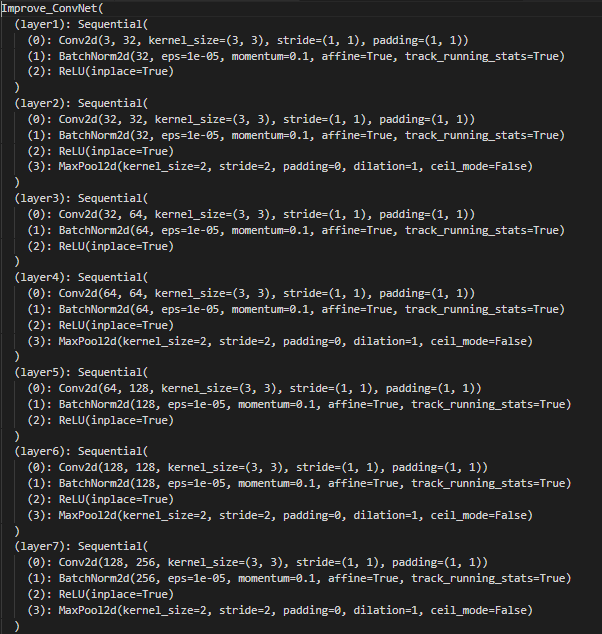
16개의 분류 이미지가 있기 때문에 출력 형태는 (-1, 16)이 됩니다.

InstanceNorm1d 을 사용한 이유 : 마지막 Linear Layer에서 Batch Normaliztion을 사용할 때 배치사이즈를 묶고 남은 이미지들이 BatchNorm1d(batchsize) 에서 batchsize보다 작아서 생기는 문제가 발생하였습니다. DataLoader에서 Droplast를 True로 주는 방법도 있습니다. 하지만 Jackson Pollack과 같이 샘플의 수가 적은 데이터가 배치에 묶이지 못해 학습되지 않는다면 성능에 하락이 올 것이라 생각을 하였으며 IntanceNorm1d를 통해 위와 같은 오류를 해결하였습니다.

참고 링크는 다음과 같습니다.

<https://discuss.pytorch.org/t/error-expected-more-than-1-value-per-channel-when-training/26274>

저는 이 model에서 Loss 값을 보면서 underfitting이 일어날 경우 Model의 Layer를 한 층씩 더 깊게 쌓도록 하였습니다.



기본적인 구조는 VGG16에서 시작 Layer의 filter 수를 64가 아닌 32로 바꿨습니다. 필터의 수가 32부터 256까지 적용한 모델의 성능이 더 우수하였습니다. 64부터 시작하여 512개의 필터를 가지는 모델은 과대 적합의 발생으로 최대 6층까지밖에 쌓을 수 없었습니다. 따라서 이 과대 적합을 없애기 위해 모델의 필터 수를 제한하였고, 과대 적합이 사라지었습니다. 즉 학습이 더 잘되면서 모델 9층까지 쌓을 수 있게 되었습니다.

저는 교수님의 ResNet 설명을 들으면서 모델을 최대한 깊게 짜보려고 노력하였습니다. 필터의 수를 8부터 1024까지 시작해보기도 하였고, Layer를 쌓을 때 필터가 바뀌는 구간마다 하나씩 넣어보며 최적의 성능을 내는 모델을 찾아보았습니다. VGG16(pretrain = True) 를 사용하면 최적의 Weight 값으로 학습된 모델을 사용할 수 있다는 것을 보며 학습시킨 모델을 얼린 후 밑부분에 Layer를 추가하여 Loss를 낮추려는 시도도 해보았습니다.

그 결과 위와 같은 모델이 최적의 성능을 발휘하게 되었습니다.

VGG16, ResNet 와 같은 모델의 성능보다 위에서 설명한 4층짜리 ConvNet의 성능이 더 잘 나오는 것을 보며 모델을 직접 설계하되, **낮고 작은 모델에서 시작하여 점점 깊어지는 모델을** 만들자 하였습니다. VGG, ResNet 모델이 너무 복잡하여 과대 적합이 발생해 성능이 낮다 생각하였습니다. VGG16 모델의 경우 (256, 256), (512,512)와 같이 많은 필터 수를 가지는 Layer가 모델을 복잡하게 하여 성능 저하를 일으킨다고 생각하였기에 저는 **VGG의 구조를 비슷하게 따라가면서 깊게 쌓지만, 학습 파라미터 수는 적은 그런 모델**을 만들고자 하였습니다. 따라서 첫 Conv Layer의 시작 필터 수를 32로 마지막 Conv Layer의 필터 수를 256으로 만들었습니다. 중간중간 추가되는 Layer의 경우에는 32 -> 64 순으로 낮은 곳에서부터 먼저 Layer를 쌓았습니다. 이 이야기는 뒤에 Loss Graph 부분에서 이어서 하도록 하겠습니다.

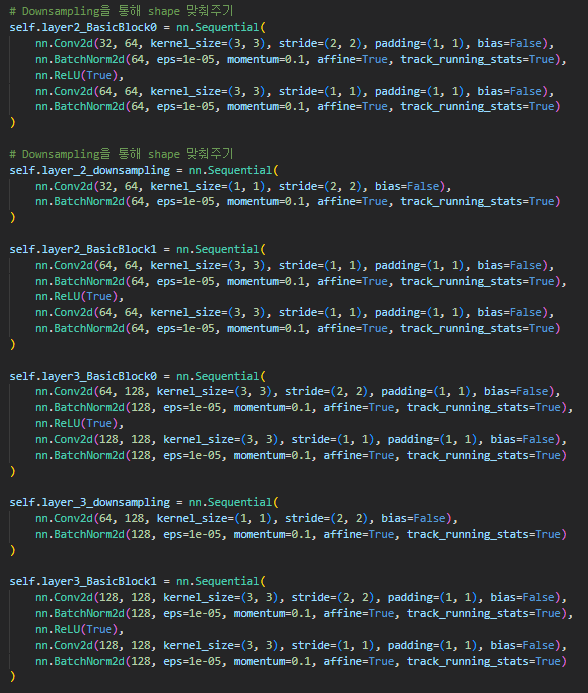
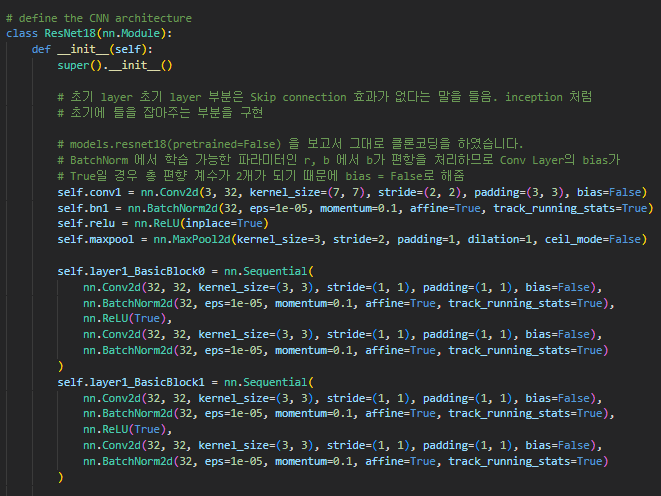
모델의 차원 분석입니다. (Improve\_ConvNet)

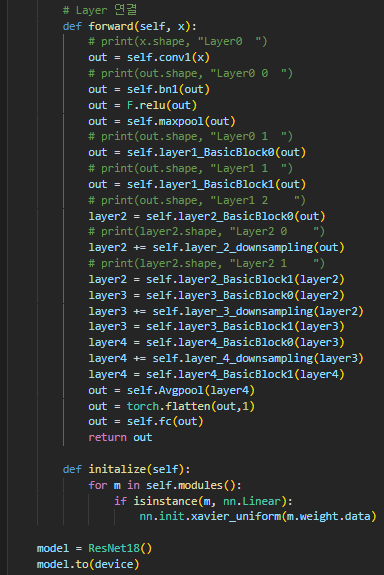
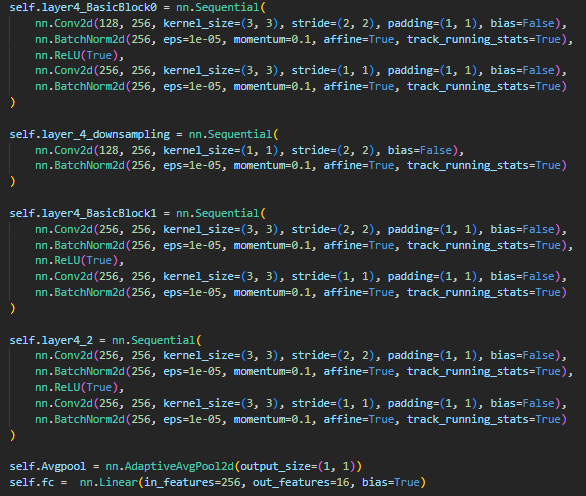
|  |  |
| --- | --- |
| 입력 데이터 | [-1, 3, 224, 224] |
| Conv2d\_Layer1 | [-1, 32, 224, 224] |
| BatchNorm2d\_Layer1 | [-1, 32, 224, 224] |
| ReLU\_Layer1 | [-1, 32, 224, 224] |
| Conv2d\_Layer2 | [-1, 32, 224, 224] |
| BatchNorm2d\_Layer2 | [-1, 32, 224, 224] |
| ReLU\_Layer2 | [-1, 32, 224, 224] |
| MaxPool2d\_Layer2 | [-1, 32, 112, 112] |
| Conv2d\_Layer3 | [-1, 64, 112, 112] |
| BatchNorm2d\_Layer3 | [-1, 64, 112, 112] |
| ReLU\_Layer3 | [-1, 64, 112, 112] |
| Conv2d\_Layer4 | [-1, 64, 112, 112] |
| BatchNorm2d\_Layer4 | [-1, 64, 112, 112] |
| ReLU\_Layer4 | [-1, 64, 112, 112] |
| MaxPool2d\_Layer4 | [-1, 64, 56, 56] |
| Conv2d\_Layer5 | [-1, 128, 56, 56] |
| BatchNorm2d\_Layer5 | [-1, 128, 56, 56] |
| ReLU\_Layer5 | [-1, 128, 56, 56] |
| Conv2d\_Layer6 | [-1, 128, 56, 56] |
| BatchNorm2d\_Layer6 | [-1, 128, 56, 56] |
| ReLU\_Layer6 | [-1, 128, 56, 56] |
| MaxPool2d\_Layer6 | [-1, 128, 28, 28] |
| Conv2d-13\_Layer7 | [-1, 256, 28, 28] |
| BatchNorm2d\_Layer7 | [-1, 256, 28, 28] |
| ReLU\_Layer7 | [-1, 256, 28, 28] |
| MaxPool2d\_Layer7 | [-1, 256, 14, 14] |
| Conv2d\_Layer8 | [-1, 256, 14, 14] |
| BatchNorm2d\_Layer8 | [-1, 256, 14, 14] |
| ReLU\_Layer8 | [-1, 256, 14, 14] |
| Conv2d\_Layer9 | [-1, 256, 14, 14] |
| BatchNorm2d\_Layer9 | [-1, 256, 14, 14] |
| ReLU\_Layer9 | [-1, 256, 14, 14] |
| MaxPool2d\_Layer9 | [-1, 256, 7, 7] |
| Dropout\_Layer9 | [-1, 12544] |
| Linear\_Layer10 | [-1, 512] |
| InstanceNorm1d\_Layer10 | [-1, 512] |
| ReLU\_Layer10 | [-1, 512] |
| Dropout\_Layer11 | [-1, 512] |
| Linear\_Layer11 | [-1, 16] (출력) |

위와 전반적인 틀 (Activation Function, weight initialization) 등은 동일합니다.

저는 개인적으로 Dropout은 0.25 정도를 선호합니다. 하지만 이번 케글 경진대회에서는 0.5를 사용하였을 때 Loss 값을 더 잘 떨어뜨리고 epoch 수도 길게 유지하였습니다. epoch가 20번 정도 저 진행할 수 있게 해줬습니다. 따라서 Dropout은 0.5를 고정해두고 사용하였습니다.

(ResNet18)





ResNet은 모델이 깊어지면 깊어질수록 기울기가 줄어드는 Vanishing Gradient 문제를 해소하기 위해 나온 방법입니다. 모델은 i번째에서 i+1번째만을 연결할 수 있지만 ResNet은 Skip Connection이라는 것을 사용하여 i번째 Layer에서 i+r 번째 Layer로 이동할 수 있습니다.

저는 이 ResNet을 구현하기 위해 많은 자료를 찾아봤지만 클래스를 사용한 ResNet 구현 부분의 코드는 이해하지 못했습니다. 소스 코드를 그대로 배껴서 주석을 다는 것은 가능하겠지만 옳지 않다고 생각하였습니다. 제가 이해한 것은 다음과 같습니다.

1. ResNet은 Skip Connection을 사용하여 모델을 깊게 쌓을 수 있게 하였다.
2. ResNet은 Block 구조를 통해 Layer를 쌓는다.

따라서 저는 아래와 같이 pretrained 모델을 참고하여 모델을 쌓았습니다. 또한 VGG 모델을 64가 아닌 32부터 시작한 것처럼 ResNet 또한 시작 부분의 Conv2d 필터 수를 32로 하였습니다.



skip Connection을 구현하는 부분은 downsampling 부분을 구현한 후 layer2 += self.layer\_2\_downsampling(out) 과 같이 더해서 구현했습니다. 이전 Block의 output이 아닌 그 전전 block의 output에 1x1 필터를 통해 Downsampling을 구현합니다. Downsampling 부분에서는 원래 차원의 2배로 줄어들어 kernel = 3, stride = 2 를 적용하여 크기가 줄어든 Layer에 적용할 수 있습니다.

out = self.layer1\_BasicBlock1(out)

layer2 = self.layer2\_BasicBlock0(out)

print(out.shape, layer2.shape)

layer2 += self.layer\_2\_downsampling(out)

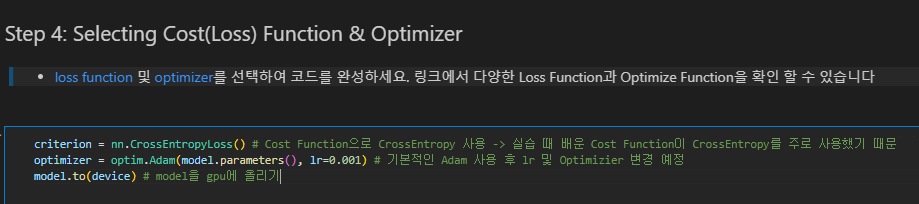
print(out.shape, layer2.shape)

기본적인 차원 분석은 위와 동일하다고 생각이 되어 Skip Connection 부분만 차원 분석을 해보았습니다. 다음 부분의 차원을 분석하자면 Layer1\_BasicBlock1에 Input으로 들어가는 shape은 (-1, 32, 56, 56) 이며 output의 shape 또한 동일합니다. Layer2\_BasicBlock0을 거치면서 Layer2는 (-1, 64, 28, 28) 이 되며 downsampling부분에는 out이 (-1, 32, 56, 56) 들어가기 때문에 layer2\_downsampling 부분은 (-1, 64, 28, 28) 이 됩니다. 따라서 이 부분을 더할 수가 있으며 Skip Connection의 효과를 줄 수 있습니다.

나머지 Block 부분은 그대로 pretrain 모델을 본떠 만들었습니다. ResNet 소스 코드를 짜면서 왜 클래스를 사용하여 모델을 쌓았는지 알게 되었습니다.

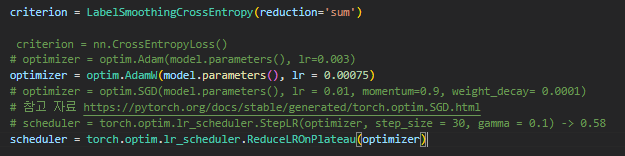
Weight initialization은 Xavier\_uniform을 사용하였습니다. 카이밍 uniform 보다 성능이 더 좋다는 자료를 본 적이 있어서 둘 다 실행하여 비교해 보았습니다. 결과는 Xavier\_uniform의 Loss가 0.05 정도 더 낮았습니다. 따라서 Xavier\_uniform을 넣어 주었습니다.

Step 4: Cost (Loss) Function 과 Optimizer 선택

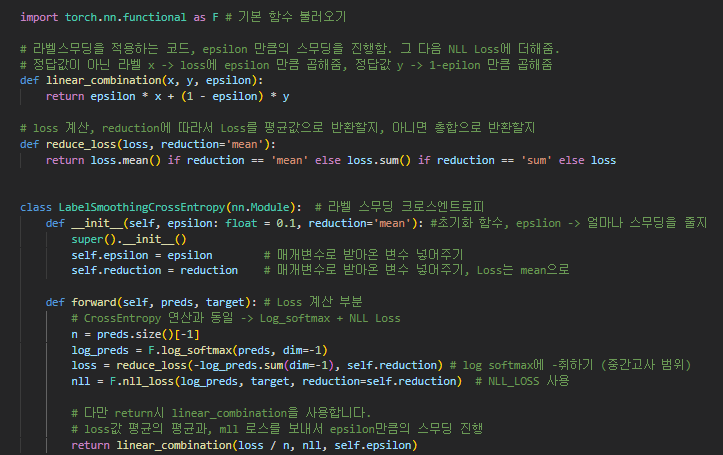


초기에 사용하였던 cost Function과 Optimizer입니다. 시작은 CrossEntropyLoss로 하였으며 옵티마이저는 Adam을 사용하였습니다. Learning\_rate한 실습 때 사용하였던 lr = 0.001을 사용하였습니다. 대회를 진행하며 Momentum/Weight Decay 가 들어간 optimizer를 알게 되었습니다. 따라서 그 후에는 SGD + momentum (0.90) 과 AdamW()를 번갈아 가며 사용하였습니다.

학습이 진행될수록 SGD + momentum보다 AdamW를 사용하는 방법이 수렴도 더 빠르고 성능도 더 좋았습니다. SGD의 경우 언더피팅과 같은 그래프가 나타나다 Early Stopping에 걸리는 경우가 많았습니다. Early Stopping count를 늘리기보단 AdamW를 사용하여 빠르게 학습을 끝내고 개선을 한 뒤 다음 학습을 진행하는 것이 낫다고 판단하였습니다.



최종적으로 사용한 Cost Function과 Optimizer입니다. Cost Function으로는 Label Smoothing Cross Entropy를 사용하였습니다.

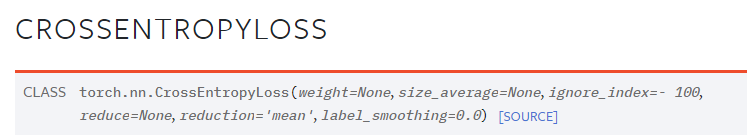


참고 링크 : <https://github.com/wangleiofficial/label-smoothing-pytorch>

LabelsmoothingCrossEntropy에 대해 검색을 하게 되면 CrossEntropyLoss 파라메타로 사용하는 라벨 스무딩에 대한 검색 결과가 적어서 아예 깃허브에서 다른 분의 소스코드를 긁어왔습니다. 또한 Cut-Mix를 사용하기 위해 Cross Entropy를 직접 수정할 수 있어야 한다고 생각하였습니다. 긁어온 부분에 대해서는 주석을 달아두었습니다. Cut-Mix의 경우는 구현하지 못하였습니다.

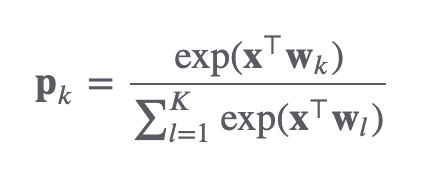
Optimizer와 Cost 함수를 선정한 이유와 선정하는데 중요하다고 생각하는 내용을 모두 작성합니다.

우선 Cost Function로 Label Smoothing Cross Entropy을 사용하였습니다. 저는 3300개에 해당하는 데이터 셋은 너무 양이 적다고 생각하여 데이터 증강을 많이 사용해야 한다고 생각하였습니다. 두 번째로 학습 데이터의 편향이 심하다고 생각하였기 때문에 일반화 기법을 최대한 적용해야 한다고 생각하였습니다. 반 고흐의 데이터는 800개가 넘지만 20개 정도의 잭슨 폴락과의 차이가 크게납니다. 따라서 자연스럽게 모델의 예측 결과는 반 고흐 쪽으로 쏠리겠다고 생각하였으며, 이를 방지하기 위해 Cost function을 찾아보던 도중 CrossEntropy에 label\_smoothing이라는 파라미터가 있는 것을 확인하였습니다.



링크 : <https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html>

따라서 저는 라벨 스무딩이라는 것에 대해서 찾아보기 시작하였습니다. 라벨 스무딩이란 라벨링을 할 때 원 핫 인코딩처럼 정답만 1로 표시하는 것이 아니라 정답이 아닌 라벨에 대해서는 조금 smooth 하게 만들어서 정규화를 시키는 법입니다. 여러 자료를 찾아보니 원 핫 인코딩 방법은 hard target, 라벨 스무딩과 같은 방법은 soft target이라 부르는 것을 확인하였습니다.

다음과 같이 softmax 수식이 있다고 할 때 one hot vector를 맞추는 방식은 CrossEntropyLoss를 미분하여 진행합니다. 위의 상태에서 역전파를 진행하면 x를 정답 class와 가까워지는 데 초점을 두게 되어 x와 w가 벡터 공간상에 가까워져 내적값이 증가하게 된니다. 즉 x를 정답 class에만 가깝게 만드는 것에 관심을 둡니다. 라벨 스무딩을 실시하게 된다면 x를 정답 class와 가깝게 하면서도 x를 오답의 w와는 거리를 두게 만든다는 것입니다. 따라서 오답과는 일정 거리를 유지하며 정답에 근접하게 된다는 것 이였습니다. 이는 logit 값이 지나치게 커지는 것을 방지하여 모델이 정답에 대해 높은 확신을 가지는 것을 막아줍니다. 이는 일반화 성능 향상으로 이어집니다.

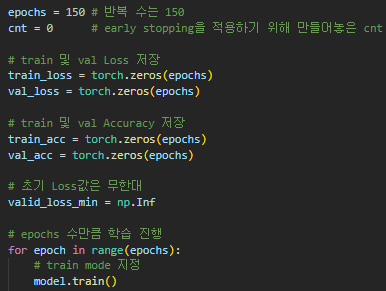
참고 링크 : <https://blog.si-analytics.ai/21> , <https://url.kr/43skgx>

Optimizer 함수로는 AdamW(lr = 0.0075)를 사용하였습니다.

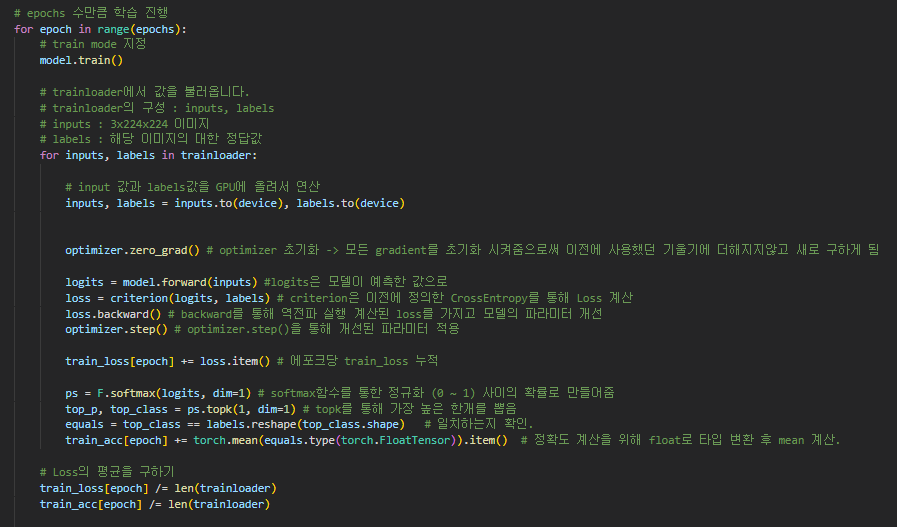
첫 번째 이유는 SGD + momentum은 시간이 오래 걸렸기 때문입니다. 빠른 속도로 Global minima에 수렴하는 AdamW를 사용하였습니다. 실제로 SGD + Momentum 사용 시 1.25배 정도 오래 걸렸습니다. 한번 학습을 돌릴 때 120분 정도 소요가 되었기에 1.25배의 차이는 크다고 생각이 들었습니다. 따라서 AdamW를 주로 사용하였습니다.

두 번째로 lr = 0.0075를 사용한 이유는 learning rate를 점점 줄임에 따라 Loss 값이 낮아지는 것을 볼 수 있었습니다. 0.008과 0.007이 비슷한 성능을 보였습니다. 따라서 중간 값인 0.0075를 사용하였습니다.

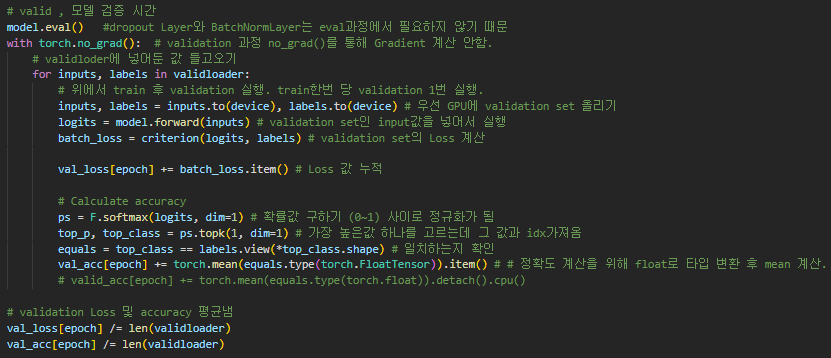
Step 5: 구성한 모델에 대한 Train and Validate 진행

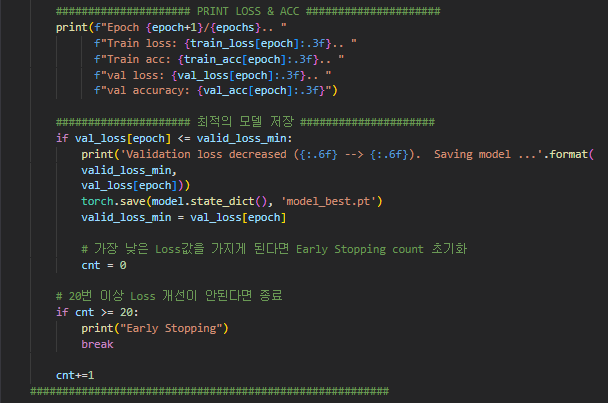
저는 일반 학습 -> KFOLD 사용 -> Stratified KFOLD 순으로 넘어갔습니다.

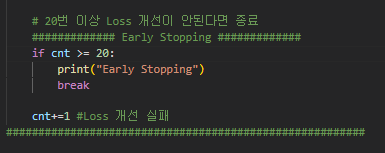
일반 학습 소스코드에 대해서 먼저 설명하겠습니다.



위의 코드는 일반 학습 중 train에 해당하는 부분입니다. 초기 코드이기 때문에 위에서 설명한 LabelsmoothingCrossEntropy 가 적용되지 않았습니다. 또한 소스코드의 기반은 실습 시간에 배운 소스코드와 매우 유사하게 짰습니다.





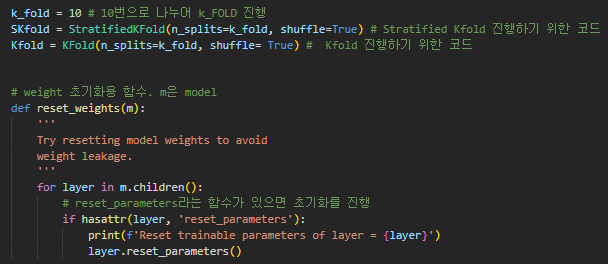
Early Stopping 부분에 주석을 달지 못해 따로 첨부하였습니다.

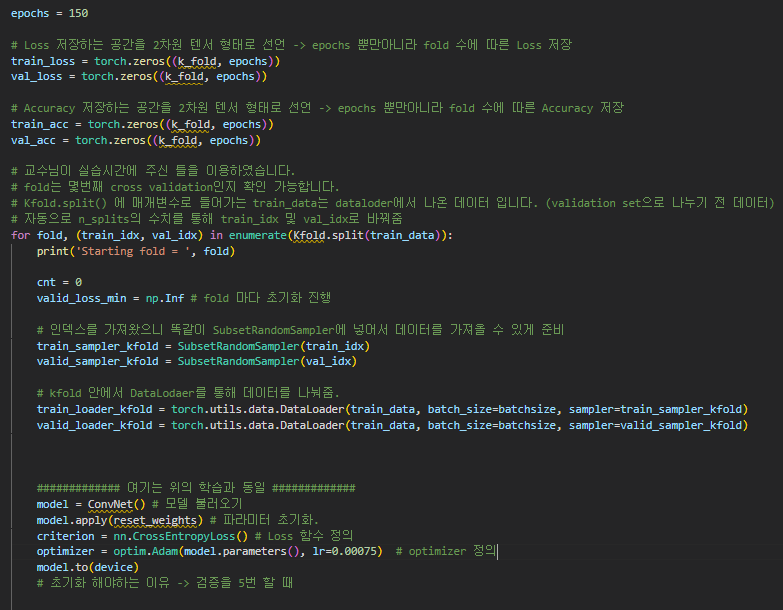
**KFOLD 및 StratifiedKFold**

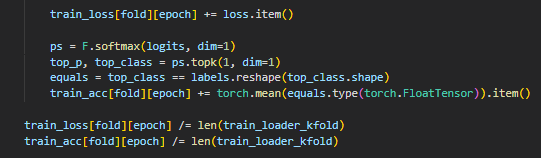
우선 KFOLD에 들어가는 과정까지의 소스코드입니다.

위와 동일한 소스코드는 뺐습니다.

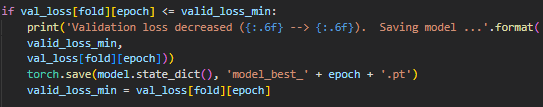








위와 같이 train\_loss 및 train\_acc를 저장을 fold에 따른 epoch에 따라 저장합니다.



모델 저장도 fold 별로 실행합니다.

StratifiedKFold()의 경우 큰 성능 향상을 보여주었습니다. 따라서 밑에서 설명하도록 하겠습니다.

Step 6: CNN model training/validation 분석

위에서 수행한 training + validation 과정을 설명하세요

training loss와 validation loss 그래프를 통해서 분석

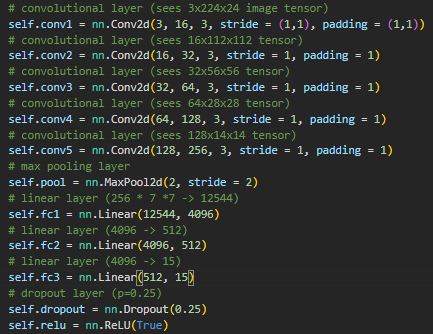
1. Loss 및 Acc 그래프 그리기

Matplotlib.pyplot을 통해 그래프를 간단하게 그렸습니다.

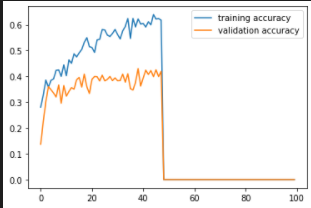
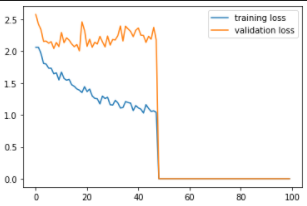


가장 초기 5월 11일

model : 5층 ConvNet + 3층 Linear Layer



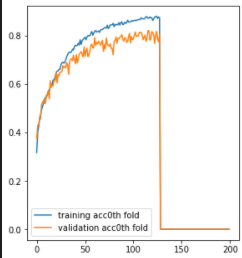
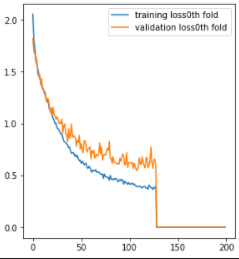




가장 초기에 사용한 그래프입니다. 일반화 기법은 Dropout(0.25) 하나만 사용하였으며 매우 간단한 모델을 사용하였을 때도 overfitting 현상이 발생하였습니다. / public(33.81)

* 저는 이 overfitting 현상을 해결하기 위해 데이터 증가, 모델의 복잡도 감소 다양한 일반화 기법 중 **Dropout비율증가, BatchNormliaztion,** **Data Augmentation**을 적용하였습니다.

사용 모델 : 5층 Conv Layer + 3층 Linear Layer 및 다양한 일반화 기법 / public(71.46)

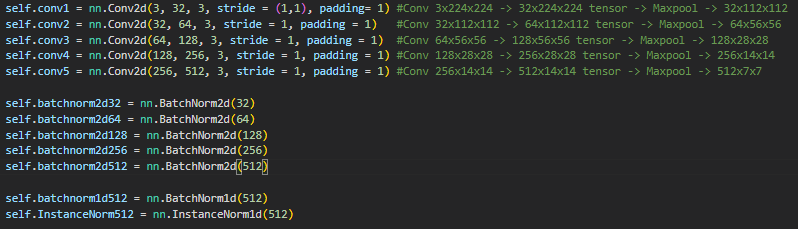


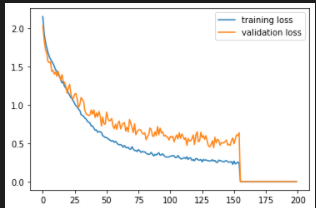
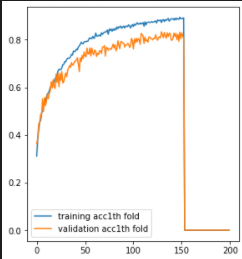


일반화를 적용한 후 학습이 좀 더 잘 되면서 모델의 Loss값이 많이 떨어진 것을 볼 수 있었습니다. 하지만 그래도 Overfitting이 남아있다고 생각하였습니다. 따라서 그 다음으로 **DataAugmentation과 모델의 복잡도도** 늘려보며 실험을 하였습니다.

(처음 시작 필터의 수를 16에서 32로, 마지막 Conv Layer의 필터 수 또한 256,512로 증가)

**사용 모델 : 5층 Conv Layer + 2층 Linear Layer**

****



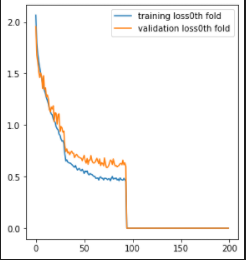
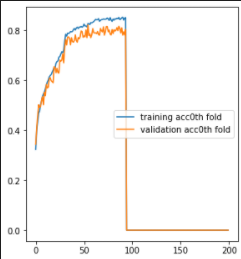
Loss가 줄어들었지만 성능에는 별 차이가 없었습니다. 위와 같은 과대적합 문제를 해결하기 위하여 저는 우선 모델의 **Data Augmentation**을 적용하였습니다**.** 추가적으로 12주차 실습 영상을 보고 **scheduler를 사용하여 test error를 개선**하려 하였습니다. Optimizer또한 **adam에서 AdamW**로 바꾸었습니다. **initialization 또한 kaiming\_normal\_()**로 변경하였습니다.

사용 모델 : **5층 Conv Layer + 2층 Linear Layer 및 Augmentation**

**그 외 : scheduler사용, Optimzer 변경, initialization kaiming\_normal() 사용**





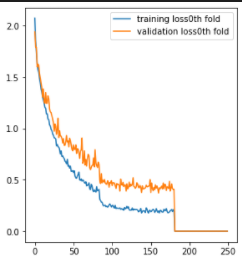
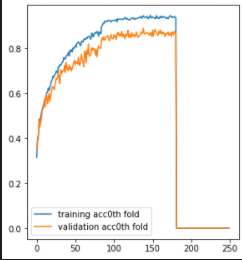
Loss는 비슷하지만 수렴하는 속도가 훨씬 많이 빨라졌습니다. Data Augmentation을 해주어 크기가 늘어났는데도 Loss가 줄어드는 속도가 더 빨라졌습니다. 성능에도 마찬가지로 별 차이가 없었습니다. 위의 방법에서 scheduler를 **ReduceLROnPlateau()** 혹은 **CosineAnnealingLR()**을 사용하면 더 좋은 성능을 낼 수 있을 거라 생각하였습니다.

사용 모델 : **5층 Conv Layer + 2층 Linear Layer 동일 + ReduceLROnPlateau()**

성능 : public 76.019% (5% 상승), Loss : 0.3719

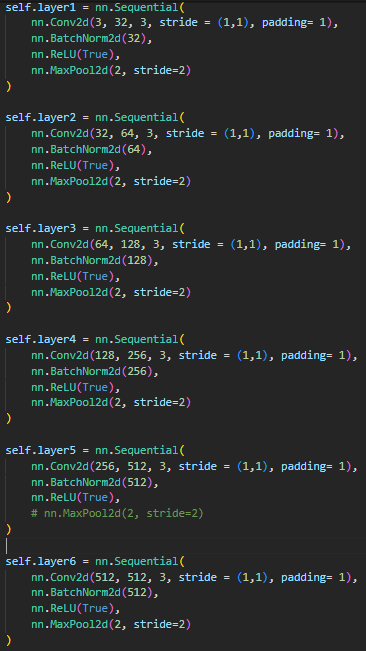




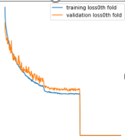
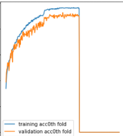
 

ReduceLROnPlateau() 스케쥴러를 사용하여 Loss를 0.37까지 줄였습니다. 또한 모델의 Loss 그래프를 본 후 Underfitting이 발생하였다고 생각하여 **모델의 복잡도를 늘렸습니다**.

* **CosineAnnealingLR()을** 똑 같은 조건에서 사용해 보았습니다. 하지만 Loss가 0.41까지밖에 떨어지지않으며 정확도 또한 public (71.223%) 로 StepLR과 동일하였기에 앞으로 **ReduceLROnPlateau() 스케줄러를 사용**하였습니다.

모델 : 6층짜리 Conv Layer + 2층짜리 Conv Layer

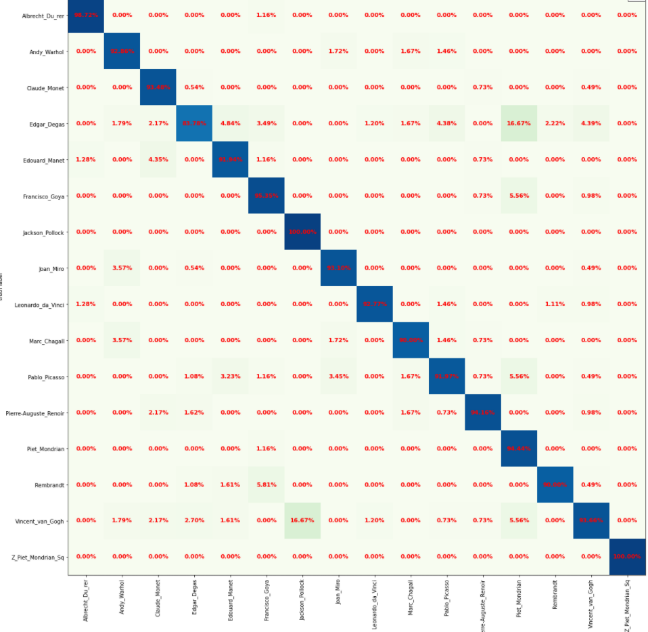
성능 : (77.458)% (1% 상승) , Loss : 0.33716

6+2층짜리 ConvNet의 Loss 및 Accuracy 그래프입니다. 다음과 같이 그림이 잘 나온 것을 볼 수 있습니다. 그림이 학습이 잘 됐다고 생각할 수 있습니다. 하지만 저 간격 마져도 줄일 수 있다고 생각하였습니다. 따라서 **모델의 복잡도를 조금 더 높여서** 저 간격을 좁히고자 하였습니다.

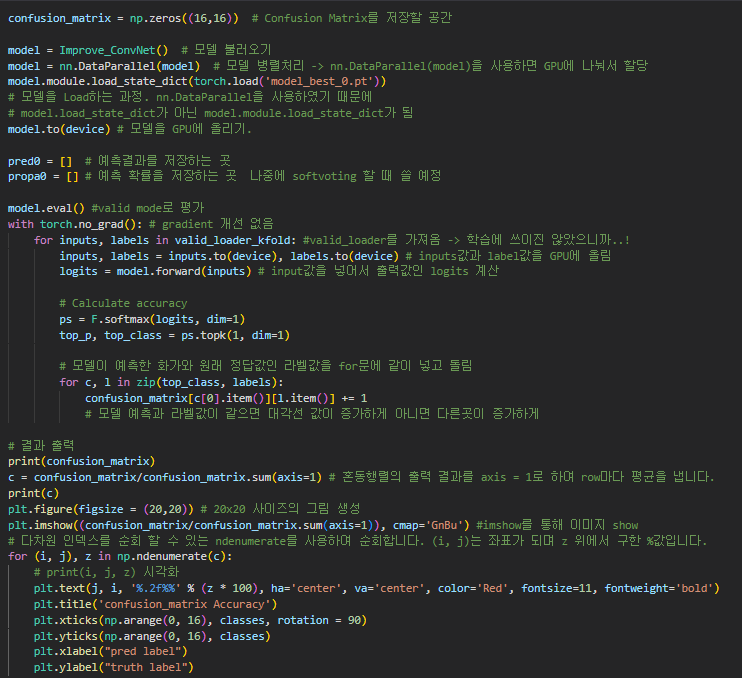


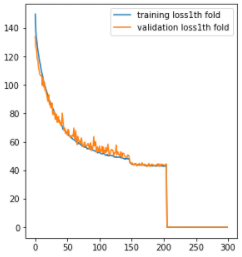
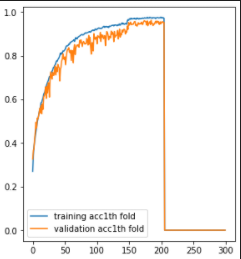
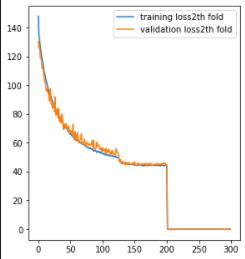
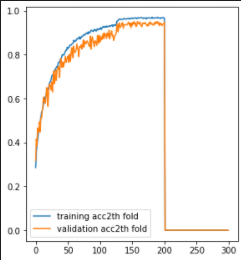
위와 같은 과정을 반복하며 8층짜리 모델을 완성하였으며 후에는 Confusion Matrix를 사용하여 잘못된 예측을 하는 샘플의 수를 늘리거나 줄여주었습니다. 또한 Test data가 각 화가의 Train Set수의 20프로 정도가 되었을 때 모델의 성능이 좋았던 점을 참고하여 Train Data의 수를 조절하였습니다.

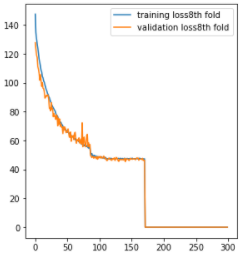
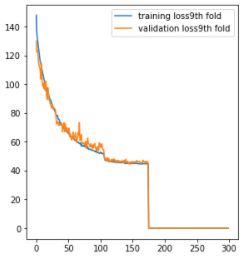
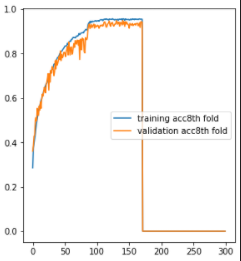
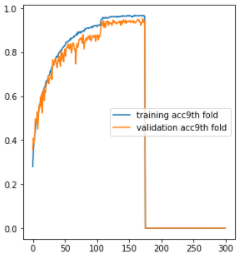
다음은 제가 직접 그린 Confusion Matrix입니다.

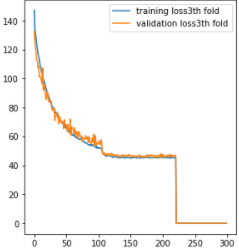
X축이 모델이 예측한 값 Y축이 원래 정답인 label값입니다. 저는 이 그림과 test set을 예측한 결과가 각각 20%에 가까운지 여부를 따지면서 모델의 Augmentation을 조절하였습니다.

만약 Jackson Pollack의 train이미지가 24장이라면 test data의 수는 4~5장 정도 일 것이라 예측하며 이 비중에 맞춰서 조절하였습니다.  
Confusion Matrix 소스코드 matplotlib을 사용하여 가볍게 그려보았습니다.

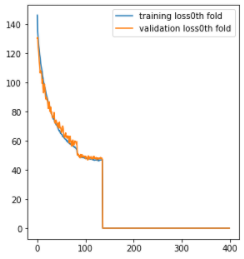
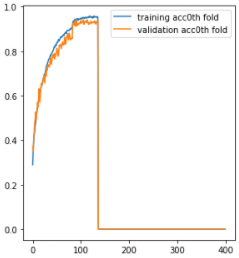


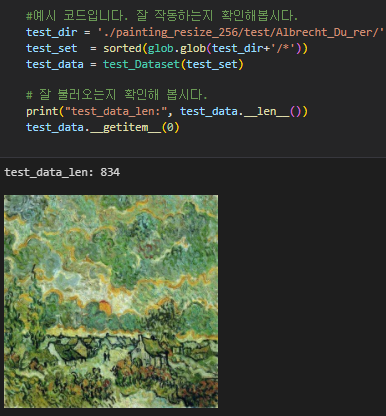
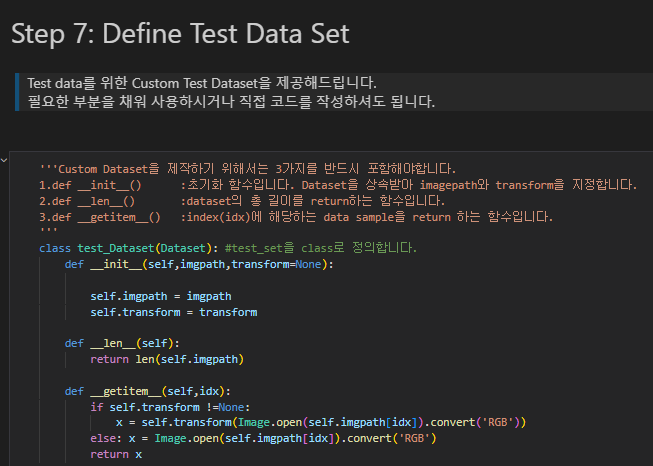
최종적으로 완성된 모델들의 Loss 및 Accuracy를 그래프로 그려보았습니다.

다음 그림과 EarlyStopping이 늦게 적용되어 학습이 끝나지 않고 길어지는 그래프는 피하였습니다. 이유는 모르겠지만 똑같은 데이터, 모델로 학습을 시켰을 때 이런 모양의 Loss 그래프를 가지는 모델의 성능이 앞선 그림의 형태를 가진 모델보다 좋지 않았기 때문입니다.

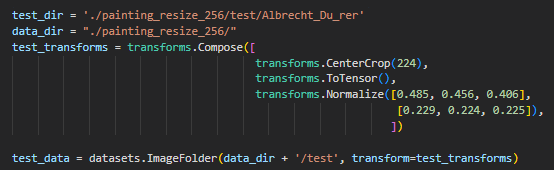
 

ResNet18() 모델을 학습시켜 나온 Loss 및 Accuracy 그래프입니다. 데이콘에서 머신러닝 관련 대회를 진행하였을 때 Hard Voting시에는 성능이 높은 모델 4개, 성능이 앞선 모델보다는 낮은 모델 1개를 사용하였을 때 성능이 좋았으며, Soft Voting 시에는 성능이 높은 5개를 적당한 비율에 맞게 섞어주는 것이 경험적으로 성능이 좋았습니다. (개인적인 경험에 의한 판단입니다. 공짜 점심 없음이라는 말이 있는 것처럼 어떤 모델이 좋은지는 직접 학습 후 확인해봐야 알 수 있지만 ResNet을 사용하기로 마음먹었기 때문에 Hard Voting을 통한 앙상블을 진행하였습니다.) 분류 모델의 경우 경험적으로 soft Voting을 진행하였을 때 성능이 잘 나왔습니다. 하지만 이번 대회에서는 ResNet()을 사용하여 결과를 예측하고 싶었기 때문에 StratifiedKFold를 진행한 Import\_ConvNet 4개의 모델과 ResNet18 모델 1개를 사용하여 Hard Voting을 진행하였습니다.

Step 7: Predict with Test Data



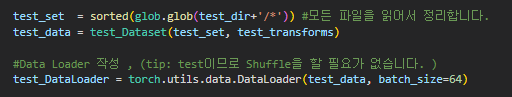
위의 양식은 기본으로 작성되어 있는 코드를 그대로 사용하였습니다.



저는 Test Loader를 작성하였을 때 경우에 따라 오류가 발생하는 경우가 초반에 잦았습니다. (이유는 정확하게 밝혀내지 못했습니다) 따라서 Train 이미지를 ImageFolder로 불러온 것처럼 Test이미지도 임시 폴더를 하나 만들어서 그 안에 넣고 ImageFolder를 사용하여 불러왔습니다.

Train Image shape은 3x224x224 이므로 CenterCrop을 통해 잘라 맞춰줍니다.

위에서 ImageNet의 평균과 표준편차를 사용한 Normalize를 하므로 똑같이 넣어줍니다.



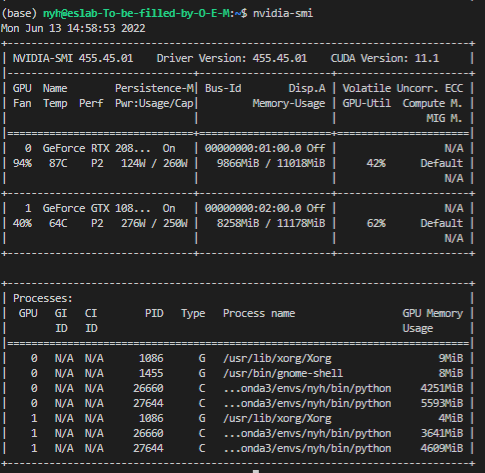
Test 시의 배치 사이즈는 64로 하였습니다.

Step 8: Training Techniques

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

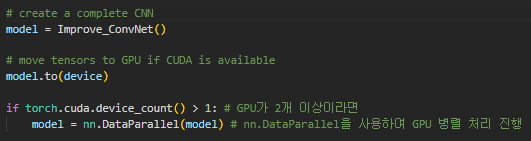
Step 0: GPU 병렬처리를 통한 빠른 학습 진행.

이 부분은 모델의 정확도 성능을 높여주는 것이 아닌 모델의 학습을 여러 대의 GPU에서 진행하므로 속도를 빠르게 해줍니다. 또한 GPU 메모리가 증가하기 때문에 배치사이즈를 늘릴 수 있으며, 메모리를 초과한 후 뜨는 CUDA\_LAUNCH\_BLOCKING ERROR 가 줄어들어 커널 재시작을 해야 하는 횟수가 줄어들었습니다. 즉 GPU 병렬 처리를 통해 개발시간의 단축과 학습 시간의 단축을 하였습니다.

GPU 1: RTX 2080TI

GPU 2: RTX 1080TI

총 GPU Memory : 24G



다음과 같이 모델을 만든 후 nn.DataParallel을 사용하면 메모리에 GPU를 분할하여 올려줍니다. 학습을 진행할 때도 배치 사이즈가 64라면 32, 32로 나누어서 처리를 진행합니다.

반복마다 Batch를 GPU 개수만큼 나누며 모델을 각 GPU에 복사하여 할당한 후 각 GPU에서 학습 진행 후 학습이 끝나면 하나의 GPU로 모아서 Loss를 계산합니다. Loss를 통해 Gradient 를 계산 후 다시 각각의 GPU로 이 Gradient를 분할한 후 각 GPU에서 Backpropagation을 진행합니다.



병렬처리를 사용할 때 모델을 저장하고 싶은 경우 model.state\_dict()가 아닌 model.module,state\_dict()로 접근해야 하여 위와 같이 저장을 하였습니다.



Model load도 마찬가지로 model.module.load\_state\_dict()를 통해 module에서 불러옵니다.

참고자료 : https://tutorials.pytorch.kr/beginner/blitz/data\_parallel\_tutorial.html

Step 2: Dataset 에 대한 Data Loaders 구성 전 Data Augmetation 적용

시작하기에 앞서 저는 데이터 수의 편향이 이번 데이터 셋의 가장 큰 문제라는 생각이 들었습니다. 모델 구조보다 전처리를 더 꼼꼼하게 진행을 해야 겠다는 생각을 하였으며 실제로 전처리에 신경을 더 많이 썼습니다

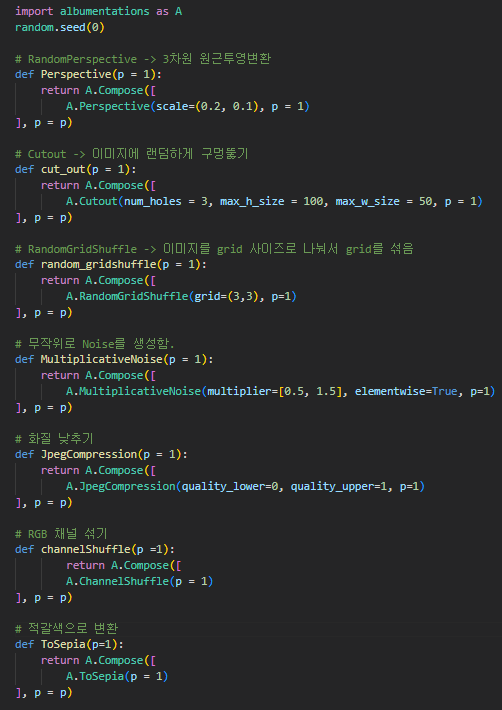
Data Augmentation 이란 데이터 증량 시켜 Test Error를 개선하는 방법 중 하나입니다.

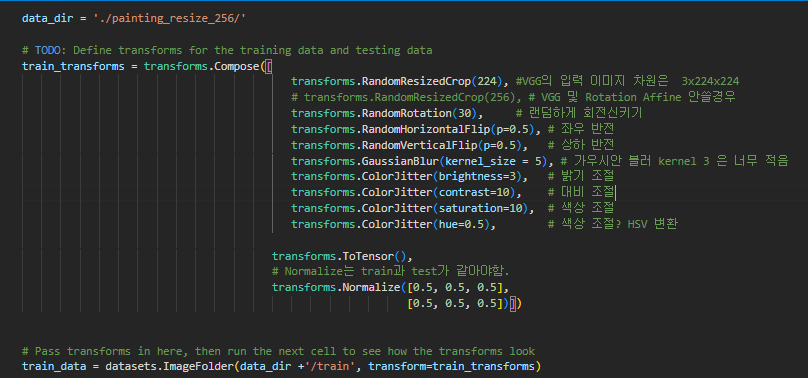
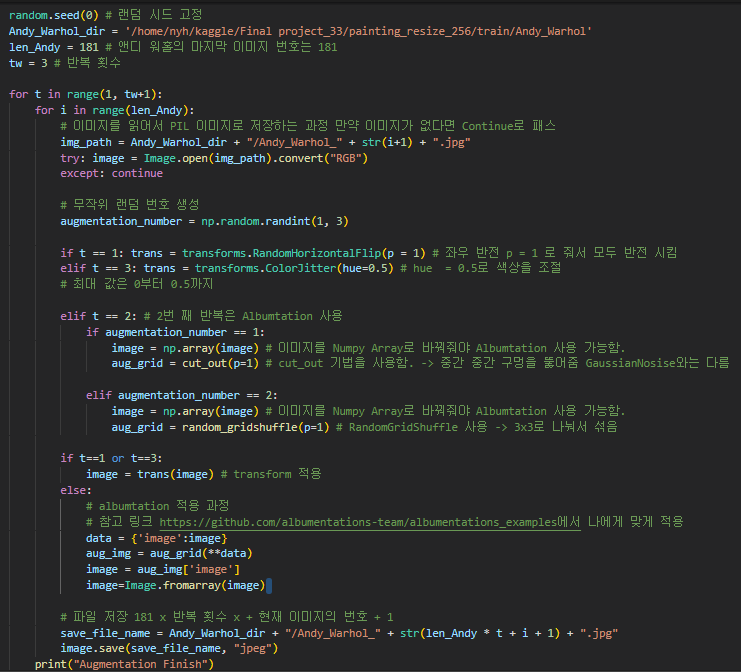
처음에는 실습 시간에 배운 Transform을 적용하였으며 Data Augmetation 방법이 너무 적다 생각이 들었습니다. 따라서 Image Augmentation에 관한 방법을 찾아보았고 albumtation이라는 모듈을 발견하였습니다.

이 중에서 Cut\_out과 Randomgridshuffle이 제일 효과가 좋았습니다. 처음 모델을 학습할 때는 ColorJitter가 학습을 방해하는 요인이 되었지만 데이터를 늘려준 후에는 좋은 성능을 보였습니다.

Data Augmentation을 진행할 때 transform을 통해 학습 중 특정한 확률을 통해 이미지가 변경되어 적용이 되게 해야 한다고 들었습니다. 초반에는 transform을 적용하여 학습 중에 이미지가 자동으로 변경되어 학습이 되도록 하였습니다. 하지만 본 데이터 셋은 클래스에 따른 샘플 수의 차이가 많이 나며, 이에 따른 문제를 해결하기 위해 클래스 별로 이미지를 변환시켜 적용시키는 아래와 같은 방법을 사용하였습니다.

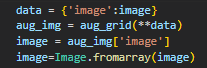
중간에 WeightedRandomSampler 혹은 SMOTE를 사용하기 위해 해당 코드를 없애고 학습 중에 변환이 되어 적용될 수 있게도 해보았지만 이미지의 수를 조절하여 각각의 학습 데이터가 20프로 정도에 맞춰지게 하는 방법이 효과적이라 생각하여 밑의 앤디워홀과 같이 클래스에 따라 다르게 이미지를 증강시켰습니다.

사용한 Data Augmentation입니다.

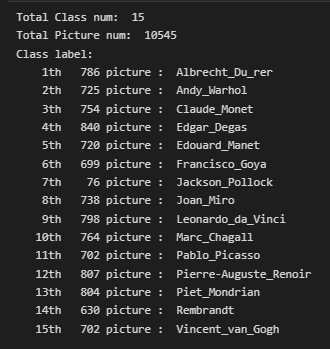
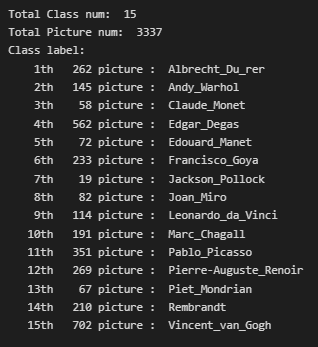
Albumtations 참고 링크 : <https://github.com/albumentations-team/albumentations_examples>

위의 링크에서 Albumtations 적용 방법을 저의 코드에 맞게 참고하였습니다.

 우선 Albumtation 변환을 위해서는 numpy의 ndarray 형태로 바꾸어야 하기 때문에 PIL Image를 ndarray 수정 후 data를 dictionary 형태로 바꾸어 Albumtation transform에 넣습니다. 그 후 변환된 transform에 image 이름으로 저장된 데이터를 꺼내 다시 PIL 이미지 형태로 변환하여 저장을 합니다. (위에서 권순이가 제공한 틀이 PIL 이미지로 저장하는 것을 응용했습니다.)

다른 클래스까지 모두 첨부하기엔 사진이 너무 많아서 대표적인 하나만 넣었습니다.

Step 2: Dataset 에 대한 Data Loaders 구성 전 Data Augmetation 적용시켜 Over Sampling 적용



* 왼쪽 기본 Train Image , 오른쪽 폴락 제외한 이미지 700~800으로 맞춰줌.

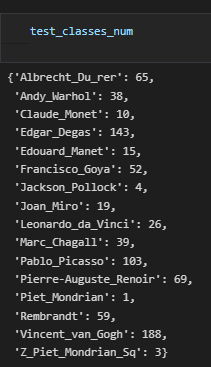
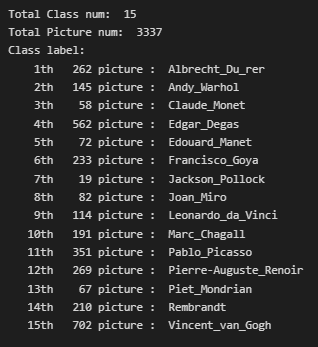
위의 방법을 사용하여 성능 향샹을 도전해 보던 중 저는 모델의 예측이 잘할 때의 특징 중 하나가 모든 train data의 20% 정도의 숫자가 test data의 분포에 수렴할 때였습니다. 또한 모델의 성능이 나쁠 경우에는 샘플의 수가 적은 Claude\_Monet과 Edouard\_Manet, Piet\_Mondrian의 예측 숫자가 매우 적었으며, 샘플 수가 많은 Vincent\_Van\_Gogh의 숫자가 매우 많은 것을 확인하였습니다. 저는 이 문제의 원인을 Train\_data가 편향되었기 때문에 나타났다고 생각하였습니다. 데이터 편향 시 사용하는 StratifiedKfold를 사용하여 성능향상을 이끌어냈지만 그래도 많은 수의 차이가 존재한다고 생각하였습니다.

따라서 Stratified\_K\_fold 외의 어떤 방법이 있을지 고민하게 되었고 ML에서 배운 Over Sampling 방법을 사용하여 Data Augmentation에 적용하기로 하였습니다.

Over Sampling이란 클래스 불균형을 조정해주기 위해 기존에 존재하는 소수의 클래스를 복제하여 비율을 맞춰주는 것을 의미합니다. 하지만 이 Over Sampling 은 똑같은 데이터를 복사하여 붙여넣기 때문에 과대 적합의 위험이 존재합니다.

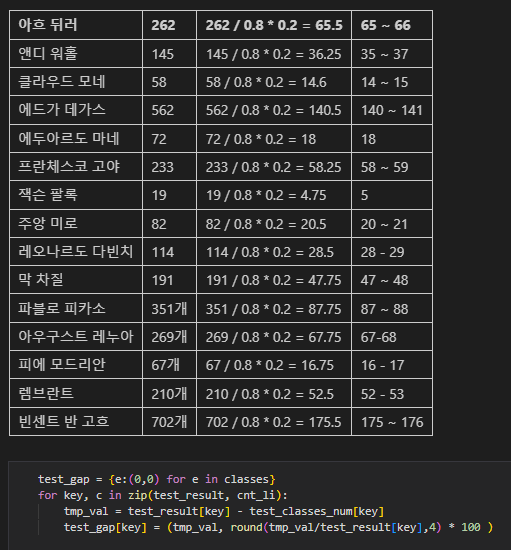
저는 이 Over Sampling 방법을 Data Augmentation 방법에 적용하여 DataLoader로 부르기 전 데이터 별로 700에서 800개 사이에 데이터를 맞춰 놓고 모델을 학습을 진행하였습니다. OverSampling 과정에서 Albumtation 모듈을 사용하였으며 임의로 데이터를 삽입하거나 삭제하는 과정을 거치지는 않았습니다. WeightedRandomSampler 혹은 SMOTE을 통해 오버 샘플링을 해주는 방법도 찾아보았습니다. 하지만 뒤에 이어지는 이미지의 수를 조절하여 통해 Test Data의 수를 20%에 맞추는 방법을 사용하지 못하게 된다 생각하여 위와 같은 방법을 사용하게 되었습니다.

Step 2: 모델이 예측한 Class 별 샘플 수를 각각 Train Data의 20% 정도에 가깝에 Train Data 수 조정 -> 가장 효과적인 방법!!



저는 학습을 진행하던 중 점수가 올라갈 때의 특징을 보면 Train 데이터의 수가 20%에 모여 있다는 것을 확인하였습니다. 위의 이미지를 보면 Joan\_Miro의 Train 이미지 수는 82정도이며 이에 20%정도 되는 값은 16-17 정도의 값입니다. 모델이 예측한 Joan\_Miro의 수는 19였으며 20프로에 가까운 것을 확인할 수 있습니다. 또한 Jackson\_Pollack, Marc\_Chagall 등 이 20프로에 가까운 것을 확인할 수 있습니다. 이러한 예측은 Confusion Matrix를 그린 이후에 더욱 확신을 하게 되었습니다. 테스트 데이터의 예측 결과가 20프로에 가까운 클래스들은 Confusion Matrix에서 거의 95프로, 그 이상의 정확도를 보여주고 있었으며 반대로 20%와 멀어지는 그림들은 정확도가 떨어지는 모습을 보였습니다.

저는 OverSampling 후 이런 부분을 의도적으로 학습 이미지의 수를 통해 조절하였습니다. 특정 화가의 그림의 수가 조금 더 많다면 예측 결과도 그쪽으로 더 쏠릴 것이라 생각하였습니다. 반대도 마찬가지로 이미지가 적을수록 모델의 예측이 더 떨어질 것이라 생각했습니다. 따라서 20%보다 낮은 수를 나타낸 그림의 이미지는 위에서 맞춰준 700~800 개보다 더 많은 Augmentation을, 20%보다 많은 이미지 수가 나왔다면 해당 클래스의 Augmentation을 줄이는 방향을 실행하였습니다.

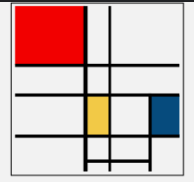


다음과 같이 20프로를 계산하여 test\_gap을 만들어 둔 다음 이 갭 차이를 좁히기 위해 노력하였습니다. 추가적으로 이후에 만들어진 Confusion Matrix를 사용하여 갭 차이 좁히기 및 오 탐지를 줄이려고 하였습니다.

Step 2: 예측률 낮은 샘플 제거 혹은 분할 혹은 클래스를 2개로 만들어서 비중을 높여줌 -> 클래스 제거는 효과 x, 분할은 효과 O

Piet\_Mondrian의 경우에는 Image Augmentation을 해주지 않을 경우 아예 예측을 하지 못하는 모습을 보여주었습니다 67개의 데이터를 가지고 있으므로 20%정도에 해당하는 13~14개의 예측 결과가 나올 것으로 기대하였으나 실제로는 1~2개의 예측밖에 하지 못했습니다. 심지어 1~2개를 예측한 것 중 정답이라고 100% 확신할 수도 없었기 때문에 어떻게 처리를 해야 할지 고민하였습니다. Mondrian이라 예측한 데이터의 Softmax 값이 50-60 근처였기 때문에 더욱 확신을 못했습니다.

우선 위에서 잠깐 몬드리안을 두 분류로 나눴다고 설명을 드렸습니다. 몬드리안의 경우

다음 그림을 그린 화가입니다. 하지만 이번 경진대회를 진행하면서 몬드리안이란 화가가 사각형이 많은 그림뿐 아니라 다른 그림도 그렸다는 것을 알게 되었습니다.

따라서 해당 그림과 비슷하게 생긴 데이터 셋을 모아서 Z\_Piet\_Mondrian 이라는 클래스로 만들어 주었으며 그 외의 데이터 셋은 원래 이름인 Piet\_Mondrian으로 만들어 주었습니다.



Z\_Piet\_Mondrian의 경우 Image Augmentation을 따로 해주지 않았습니다. 이미지를 더 생성할 경우 피카소의 그림과 섞이지 않을까 라는 우려를 하였기 때문입니다. 분할된 Z\_Piet\_Mondrian의 경우에는 Piet\_Mondrian과 동일한 Image Augmentation을 해주었습니다. 해당 작업을 통해 Z\_Piet\_Mondrian의 경우 3~5개를 꾸준하게 검출하였으며 Piet\_Mondrian의 경우에는 여전히 낮은 예측율을 보여주었습니다. 해당 작업을 통해 Public 83.5까지의 스코어를 올릴 수 있었습니다.

두번째 작업은 모네를 제거하였습니다. 모네의 경우 Confusion Matrix를 그렸을 때 정확도 몬드리안과 비슷하게 낮게 나왔습니다. 주로 반 고흐 및 다비치와 겹치는 모습을 보여주었습니다. 58개의 클래스를 가지고 있다면 12개 정도의 데이터가 존재할 것이라 생각하였습니다. 따라서 다음과 같은 가정을 하였습니다.

첫번째 12개 중에 50%를 만족시키지 못한다면 오히려 학습에 방해가 된다.

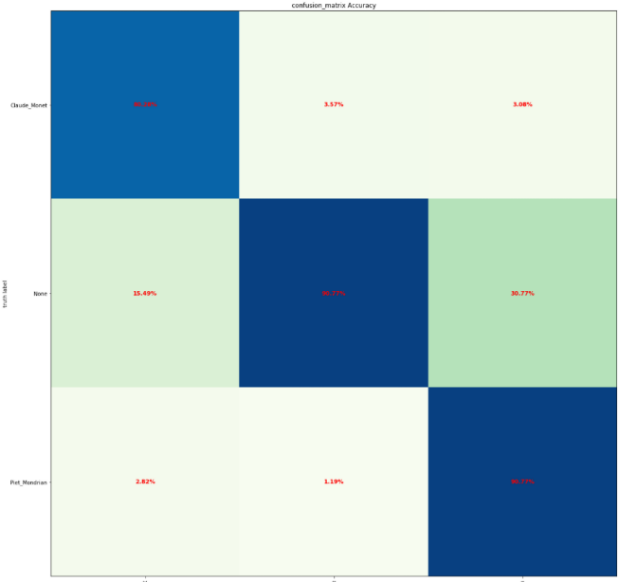
두번째 80%를 정확도를 기준으로 한다면 833개의 테스트 데이터 중 160개의 오차가 존재하며 이 중 10개 정도는 버려도 되지 않을까? 라는 생각을 하였습니다.

결과적으로 몬드리안 클래스까지 제거 후 학습을 진행하며 앙상블을 하였을 때 (public 83.9)라는 최고점을 찍었지만 보고서를 쓰면서 생각해 보면 12개 중 2개만 더 맞았더라면 0.002% 정도는 올랐을 것이라는 생각이 들어 아쉬움이 남는 결정 이였던 것 같습니다.

위의 같은 실수를 한 후 다시 모네와 몬드리안을 합친 후 StratifiedKFold 및 앙상블을 시도해 보았습니다. 성능이 전과 비교하여 뚜렷하게 높아진다고 보장하기도 힘드며, 학습 시간만 무려 30시간이 걸려 기간 내에 학습을 할 수 있을까 걱정하였으나 다행이 완료하였습니다. 몬드리안 클래스를 2개로 나눠서 진행해야 할지 고민하였습니다. 몬드리안 클래스를 하나로 사용하는 것 보다 두 개로 나누어 적용하는 것이 효과가 훨씬 좋았습니다. (3%정도 차이가 남)



모네 와 몬드리안을 따로 학습시킨 모델을 만들어서 위의 모델과 Hard Voting을 통해 덮어 씌우려는 시도도 해보았으나 모네 와 몬드리안을 따로 학습시킨 모델의 F1 score를 구해 보았을 때 만족 할 만한 모델을 만들지 못했습니다.



Precision of Model 구하기

Precision = TP / (TP + FP)

Recall = TP / (TP + FN)

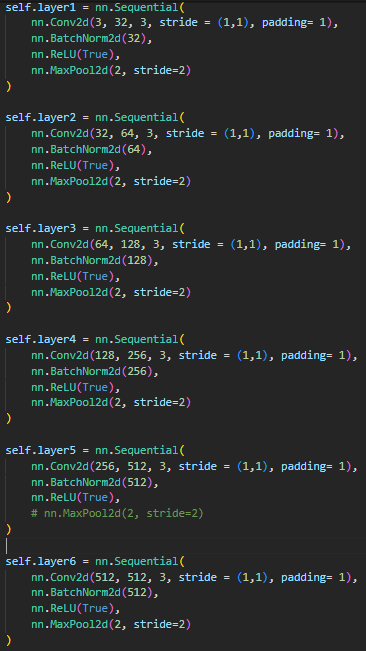
F1score = 2 x Precision x recall / (precision + Recall)



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | TP | FP | FN | Precision | Recall | F1 score |
| Monet | 57 | 13 | 14 | 57/(57+13) = 0.81 | 57/(57+14) = 0.80 | 0.80 |
| None | 305 | 16 | 31 | 305/(305+16) = 0.95 | 305/(305+31) =0.90 | 0.92 |
| Mondrian | 59 | 22 | 6 | 59/(59+22) = 0.72 | 59/(59+6) = 0.90 | 0.80 |

13개의 클래스의 데이터로 학습시킨 모델의 성능은 아쉬움이 남았습니다. 모네 및 몬드리안만 분리한 모델을 통해 83.5를 찍었으며, 앙상블을 통해 최고 점(83.9)를 찍었다는데 의의가 있다 생각하여 적었습니다.

83.9라는 점수를 기록했지만 학습을 진행한 과정이 아쉬움이 많이 남아 다시 도전하게 되었으며 이전의 실수를 바로잡아 87.29라는 점수를 기록하게 되었습니다. 결과적으로는 **하나의 클래스를 분할하였던 것이 큰 효과를 가져다주었습니다.**

Step 3: 모델 구성시 BatchNormaliztion 사용

이 사진은 제가 초기에 사용한 6층짜리 모델입니다.

Convolution 연산을 통과한 이후 BatchNorm2d를 적용하여 가장 큰 성능 개선을 이끌어 냈습니다.

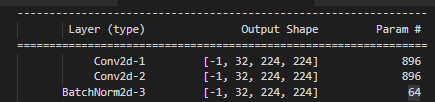
Data Normalize를 하는 이유 -> 평균이 0이고 분산이 1인 데이터를 만든다면 학습에 좋은 효과 -> Gradient가 균일하게 움직일 수 있게 함.

-> 학습에 좋은 효과

BatchNormlize를 하는 이유 위의 Data Normalize를 하는 이유와 비슷합니다. Layer를 거친 출력 데이터를 다시 평균 0 분산 1형태로 만들어주어 Gradient가 균일 하게 움직일 수 있게 합니다. 또한 pytorch에서는 를 학습시켜 Layer 마다 가장 적당한 값을 저장합니다.

저는 모델을 학습하기 전 차원이 맞는지 확인을 해보기 위해 

torchsummary라는 것을 활용하였습니다. 이론 동영상에서 사용되는 값이 궁금하여 찾아보다가 이 torchsummary를 활용하면 모델 학습에 사용되는 학습 파라미터 수를 알 수 있다는 것을 알게 되었습니다. 해당 값을 직접 출력하지는 않았지만 다음과 같이 값이 저장된 다는 것을 알 수 있습니다



Step 4: Label Smoothing Cross Entropy

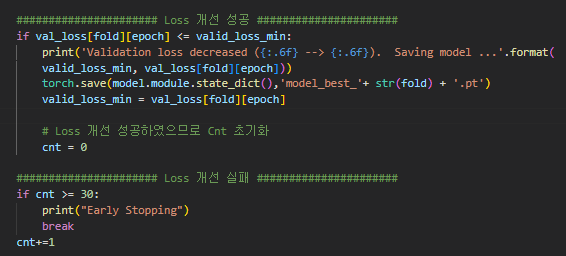
Cost Funciton으로 Cross Entropy를 사용하여 79점에서 80점으로 넘어 갈 수 있었습니다. 설명은 Step 4에서 하였습니다. 위에서 설명하였으므로 넘어가겠습니다.

Step 4: CallBack 함수(scheduler 및 EarlyStopping), Optimizer는 AdamW 사용

scheduler = torch.optim.lr\_scheduler.ReduceLROnPlateau(optimizer)

Schelduer로 ReduceOnPlateau를 사용하였습니다. Step5에서 그림과 같이 설명하였습니다.

학습이 시작된 후 사람이 할 수 있는 것이 없으므로 기다려야 하는데 이를 해결하고자 하는 것이 콜 백 함수입니다. 학습 과정 사이의 학습률을 변화시키는 Schelduer, 혹은 val Loss가 개선되지 않으면 멈추게하는 EarlyStopping이 있습니다. Schelduer는 모델의 성능 개선이 없을 경우 Learning Rate를 조절하여 모델의 개선을 유도하는 콜 백 함수입니다. EarlyStopping의 경우 함수를 구현하지 않고 가볍게 count를 세는 변수를 만들어 사용하였습니다.



Step 5: StratifiedKfold 사용

StratifiedKFold는 KFold와 원리는 동일하지만 편향된 데이터를 처리할 때 더욱 도움이 됩니다. KFold는 Cross Validation 방법 중 하나로, 훈련 데이터를 일정한 비율로 나눈 후 번갈아 가며 Train/validation set이 되어 학습을 진행합니다. 따라서 데이터가 적은 경우 매우 효과적인 방법이 될 수 있습니다. StratifiedKFold와 KFold의 차이가 존재한다면 데이터를 일정한 비율로 나눌 때 무작위로 섞어서 나누는 것이 아닌, 클래스에 따른 샘플 수의 비율을 유지하며 데이터를 분할합니다. 이번 경진대회의 데이터 셋은 데이터의 편차가 심하기 때문에 StratifiedKFold를 사용하는 것은 필수라고 생각하였습니다. 따라서 다음과 같이 사용을 하였으며, 기존 KFOLD의 선언 방식과는 다르게 아래와 같이 target값을 넣어주어 선언을 한 후 사용을 하였습니다.



StratifiedKFold를 사용하여 학습을 마친 후 전체 Fold의 Loss의 평균을 내서 이 모델의 성능을 평가를 하는 것으로 알고 있습니다. 저는 성능평가를 할 때 모델의 Loss의 평균을 낸 것이 아닌 fold마다 최적의 Loss값을 가진 모델을 저장한 후 이 최적의 모델로 앙상블을 진행하는 StratifiedKFold Ensemble을 진행하였습니다.

Step 8: Hard Voting 혹은 Soft Voting 적용

tmp = []; tmp\_prop = []; tmp\_prop\_all = []

# tmp -> 예측값 하나만 가지고 진행 -> Hard Voting

# tmp\_prop\_all -> test 데이터에 대한 예측 확률 모두 조사 softamx() 거친 후 바로 저장

model.to(device)

for t in range(5):

    tmp = []; tmp\_prop = []; tmp\_prop\_all = []

    for data in test\_DataLoader:

        if t == 0:

            model = Improve\_ConvNet()

            print(model)

            model = nn.DataParallel(model)

            model.module.load\_state\_dict(torch.load('model\_best\_ConvNet\_01.pt'))

            model.eval()

            pred0 = []; propa0 = []; prop0\_all = []

            model.to(device)

        elif t == 1:   …

        elif t == 2:   …

        elif t == 3: …

        elif t == 4:

            model = ResNet18()

            model = nn.DataParallel(model)

            model.module.load\_state\_dict(torch.load('model\_best\_ResNet18.pt'))

…

        data = data.to(device)

        logits = model(data)

        ps = F.softmax(logits, dim=1).detach().clone()

        top\_p, top\_class = ps.topk(1, dim=1)

        tmp.append(top\_class)

        tmp\_prop.append(top\_p)

        tmp\_prop\_all.append(ps)

        if t == 0:

            pred0 = tmp

            propa0 = tmp\_prop

            prop0\_all = tmp\_prop\_all

        elif t == 1: …

        elif t == 2: …

        elif t == 3: …

        elif t == 4:

            pred4 = tmp

            propa4 = tmp\_prop

            prop4\_all = tmp\_prop\_all

저는 StratifiedKFold를 사용하여 나온 Improve\_ConvNet 모델 4개와 ResNet18 모델 1개를 사용하여 앙상블을 진행하였습니다. 앙상블이란 여러 모델을 결합하여 정확한 모델을 만드는 방법입니다.

분류 모델에서의 앙상블 방법은 Voting 방법이 있습니다. Voting이란 말 그대로 투표를 실시하는 것 입니다. Hard Voting은 모델이 예측한 클래스를 모아두고 과반 수 이상 넘기는 클래스를 모델의 output으로 지정하는 방법입니다. Soft Voting이란 하나의 test data에 대해 모델이 예측한 softmax 값을 data별로 합하여 가장 높은 확률을 가지는 클래스를 정답으로 처리하는 방법입니다.

이론적인 내용이 매우 간단하다고 생각하여 참고자료 없이 제가 직접 코드로 구현하였습니다.

np.set\_printoptions(threshold=np.inf, linewidth=np.inf) # 출력 제한 없음.

# 여기는 SoftVoting 소스코드

result\_propa = []

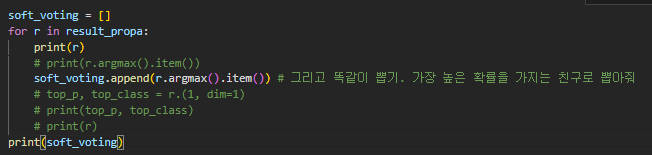
propa\_count = 0

for one, two, three,four, five in zip(prop0\_all, prop1\_all, prop2\_all, prop3\_all, prop4\_all):

    for o, t2, t3, f4, f5 in zip(one, two ,three, four, five):

        result\_propa.append(o + t2 + t3 + f4 + f5) #확률 모두 더하기

result\_propa

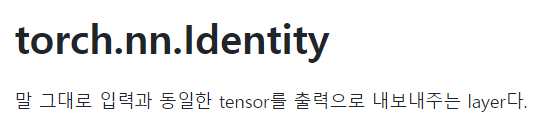
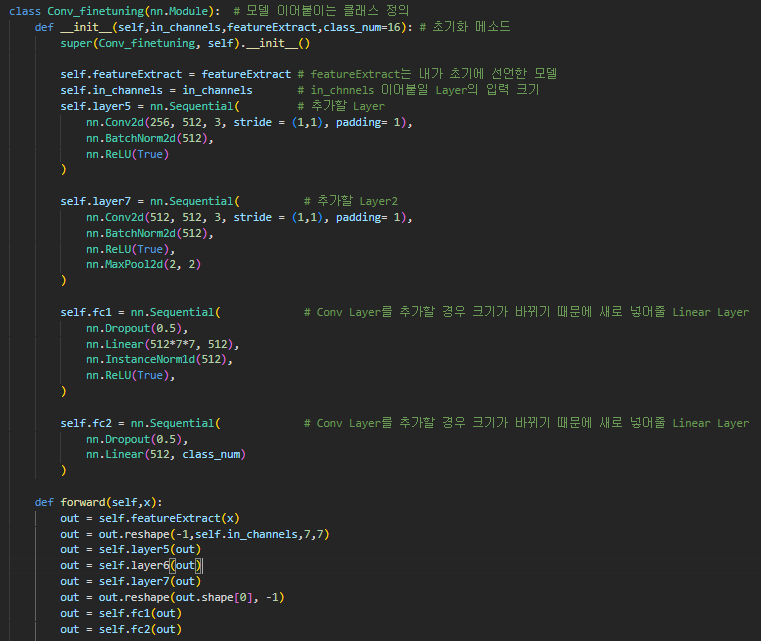
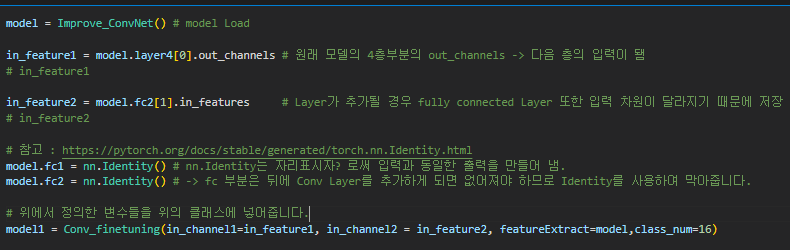
저는 상대적으로 낮은 정확도를 가지고 있는 ResNet18 모델을 사용하기 위해 Hard Voting을 사용하였습니다.

Step 9: 그 외, 성능 개선을 위해 사용한 기법 중 효과가 없었던 행동

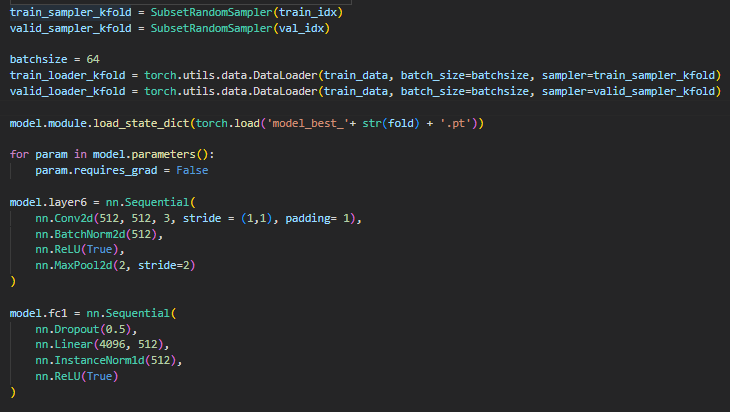
9-1. 학습된 모델을 얼린 후 층을 더 쌓아서 학습하기.

저는 Pretrain 모델을 불러와서 마지막 Classifier 부분만 대체가 가능한 것을 보며 그렇다면 나도 복잡도가 낮은 모델을 만들고 학습을 시킨 후, 모델을 얼린 후 Layer를 이어 붙여 학습을 시킬 수 있지 않을까? 라는 생각을 하였습니다.

모델이 너무 복잡해서 학습 진행이 되지 않는 것이라면, 모델이 어느정도 Layer를 지난 이후에 Layer를 쌓아본다면 효과가 있지 않을까 생각하였습니다. 따라서 오른쪽 같이 클래스를 정의한 후 아래와 같은 명령어를 통해 Class에 넣었습니다.

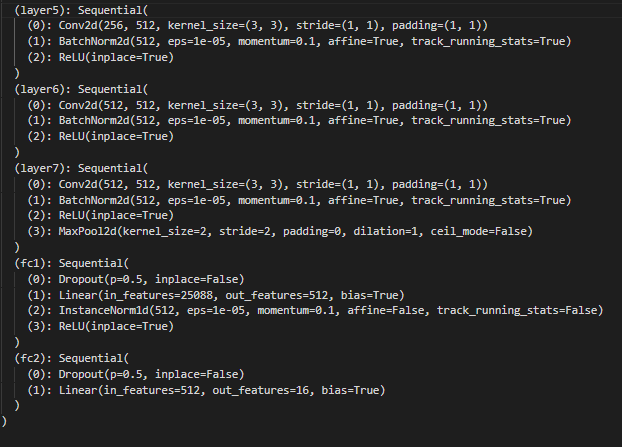
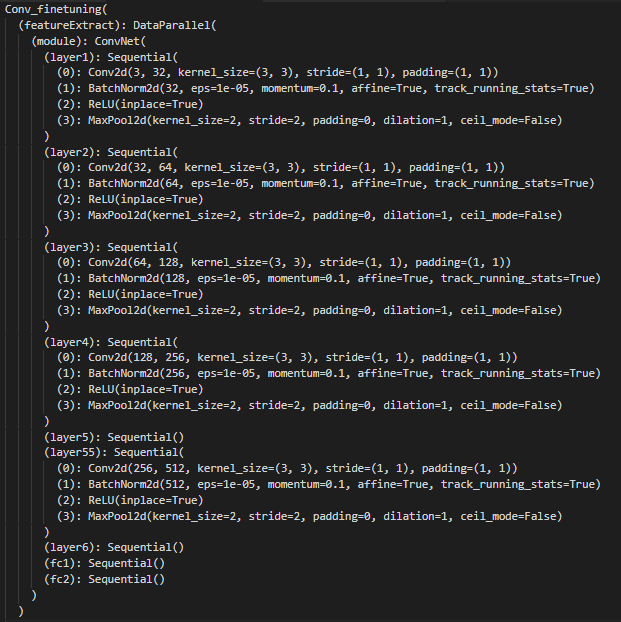
 

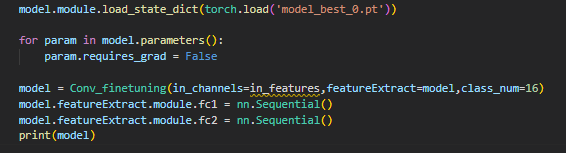
처음에는 아래와 같이 Early Stopping을 통해 학습이 조기 종료된 경우 Load를 통해 모델을 불러온 후 parm.requires\_grad = False를 통해 기존 모델의 Weight를 고정하는 방법을 사용하였습니다. 그 다음 밑에서 Layer를 선언하여 이어 붙인 후 새로 추가되는 Layer의 기울기만 학습시키는 방법을 사용하였으나 모델을 이어 붙이는 과정에서 오류가 나게 되어 모델을 선언할 때 Class를 사용했던 것처럼 구현하려고 하였습니다.



참고 자료로는 ResNet을 구현하기 위해 자료를 찾아보던 중 Block 구조를 만들기 위해 클래스에 여러가지 변수를 \_\_init\_\_() 함수에 매개변수로 넣어서 처리하는 것을 보고 영감을 받았습니다. 클래스의 \_\_init\_\_(): 을 활용하여 클래스로 처리하면 모델을 이을 수 있겠다 라는 생각을 하였으며 위의 클래스 선언부와 매개변수로 넣는 부분을 코드로 구현하였습니다

위의 명령을 실행시킨 후 모델을 print하여 찍은 구조입니다. 중간에 layer5가 비어있는 이유는 저 부분에 이어 붙이려는 시도를 하기 위해 저 부분은 nn.Sequential() 로 초기에 선언 후 비워두었으나, 저 부분에 붙이는 과정은 실패하고 밑을 identity() 혹은 Sequential() 로 막아서 붙이게 되었습니다.



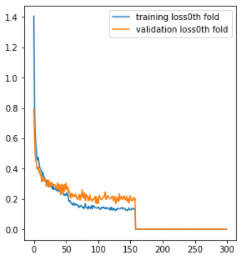
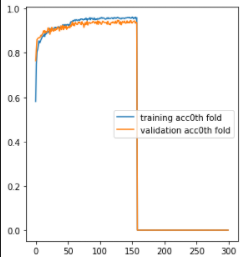


다음과 클래스를 활용할 경우 위와 같이 소스코드가 깔끔해졌습니다. nn.Identity() 대신 와 nn.Sequential()을 사용하여도 동일하게 실행이 되었습니다.





다음과 같이 Loss를 0.50 정도에서 0.16까지 줄이는데 성공하여 성능향상을 가져다줄 수 있을 것으로 기대하였으나 모델을 제출하였을 때 성능은 그렇게 좋지 않았습니다.

Loss 및 Accuracy 그래프입니다. 학습이 끝난 모델을 load 후 재 학습시켰기 때문에 기존의 Loss값과 비교를 하면 떨어지는 조금 더 떨어지는 모습을 보였습니다. 정확도의 경우 조금 더 상승하는 모습을 보였습니다