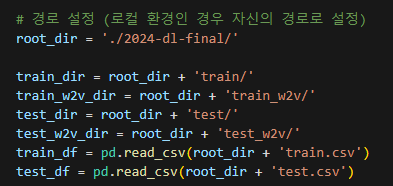
딥러닝이론및응용 기말 Project

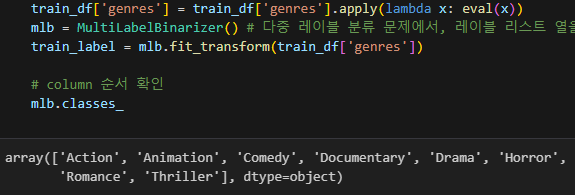
Step 1: Dataset 준비하기

<코드 캡쳐 첨부>

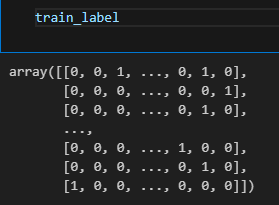
사용한 GPU : NVIDIA RTX A5000

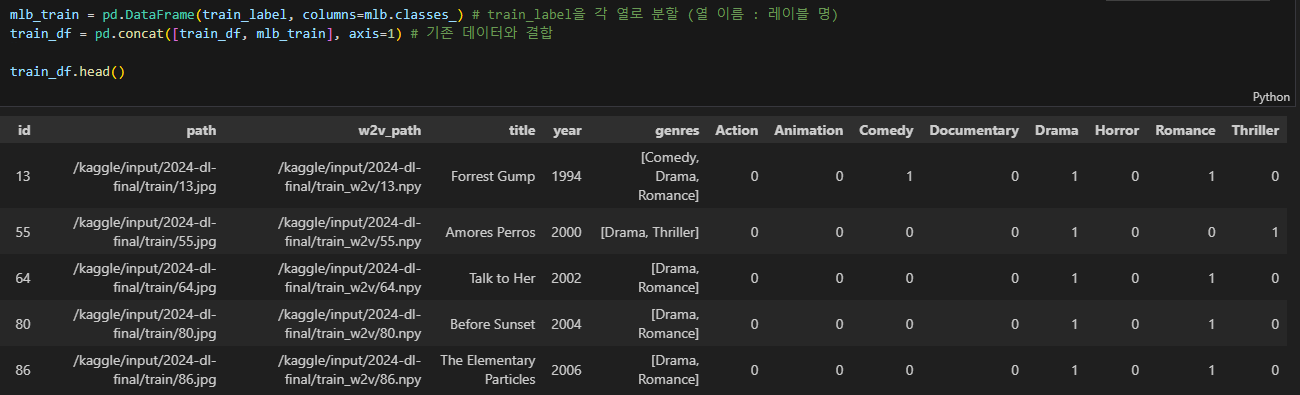
기본적인 데이터셋 준비는 베이스라인과 동일한 코드를 사용했습니다.





경로를 설정하여 csv파일을 불러오고, MultiLabelBinarizer를 사용하여 레이블 인코딩을 수행했습니다. MultiLabelBinarizer는 다중 레이블 분류 문제에서, 레이블 리스트 열을 넣어주면 자동으로 레이블 종류(개수)를 인식하고 레이블 인코딩 수행합니다. MultiLabelBinarizer의 classes\_로 column을 확인한 결과, 8개의 장르가 존재함을 확인했습니다.

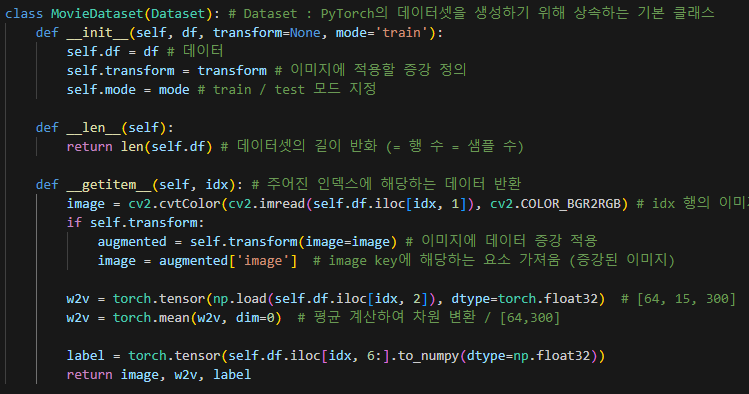
Train\_label을 출력하여 확인해본 결과, 왼쪽 사진과 같이 인코딩 된 것을 볼 수 있었습니다. 예를 들어 [0,0,1,0,0,0,1,0]인 데이터는 3번째 장르인 Comedy와 7번째 장르인 Romance에 속하는 영화를 나타냅니다.



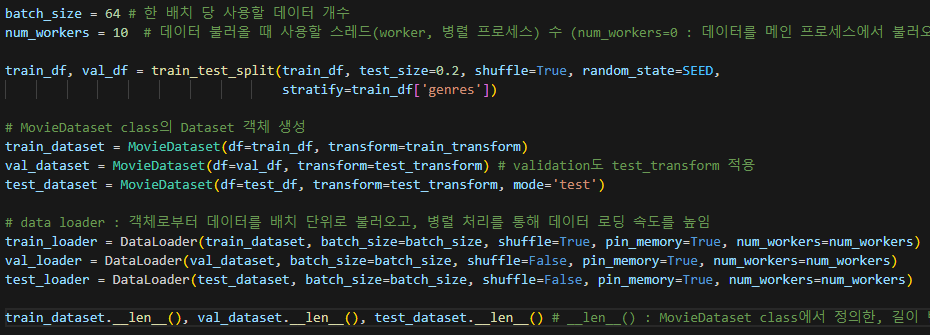
이러한 train\_label을 각 열로 분할하여 총 8개의 열을 생성한 후, 기존의 데이터프레임과 결합하여 위와 같은 하나의 데이터프레임을 구축하였습니다.

Step 2: Dataset에 대한 Data Loaders 구성

<코드 캡쳐 첨부>

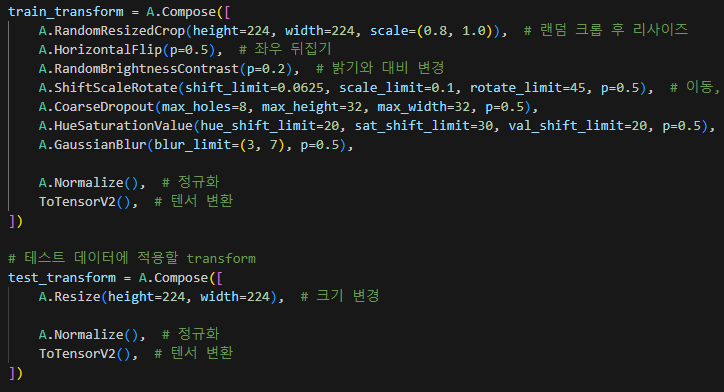


Data Loader와 관련한 코드 역시 base line 코드를 기반으로 모델에 맞게 변형하여 사용했습니다. 기존 w2v 코드에서 불러온 데이터의 shape는 배치사이즈가 64인 경우 [64, 15, 300]이지만, 모델을 2차원 데이터를 입력받도록 구성했기 때문에 tensor를 다루는 함수들에 대한 글(<https://kyull-it.tistory.com/160>)를 참고하여, torch.mean()을 사용하여 [64,300]으로 차원을 축소했습니다. 또한 dtype 인자를 사용하기 위해 torch.Tensor()가 아닌 torch.tensor()를 사용했습니다.

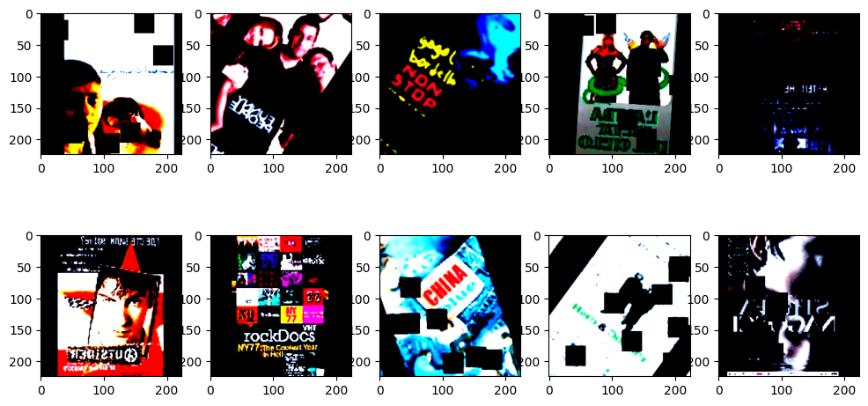
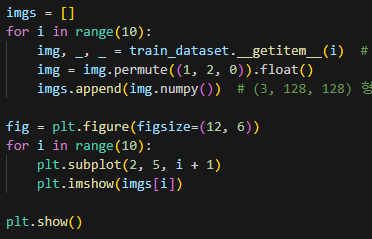


뒤에서 설명드릴 여러 번의 실험을 거쳐 배치사이즈는 64로 설정하였습니다. nproc 명령어로 사용 가능한 cpu 코어가 20개임을 확인했지만, 데이터 수가 매우 많지 않기 때문에 너무 많은 값으로 설정하지 않아도 될 것 같아, 10으로 설정했습니다. 학습 데이터를 나누어 80%는 학습에, 20%는 validation에 사용했습니다. DataLoader에서는 일반화된 학습을 위해 학습 데이터셋에서만 shuffle을 허용하도록 했습니다.

데이터 전처리에 대한 설명



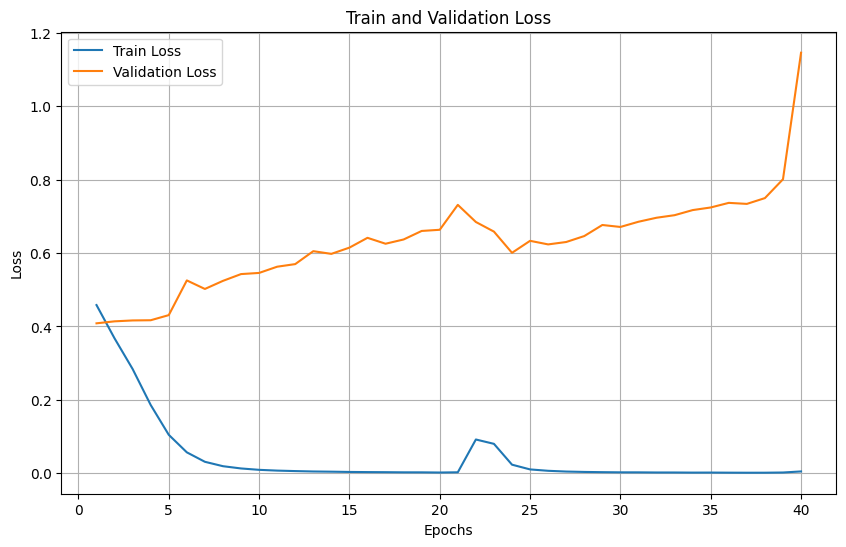
최종적으로 이미지에 적용된 증강 방식입니다. 증강은 Albumentations에서 공식적으로 제공하는 augmentation pipeline 코드(https://albumentations.ai/docs/examples/example/)를 참고하여 작성했습니다. Test set의 경우 데이터를 변형(증강)할 필요가 없기 때문에, train set에 대해서만 증강했습니다. Train data에 대해 증강을 적용한 결과는 아래와 같이 확인하였습니다.



데이터 증강과 관련하여 크게 세 경우의 성능을 비교하였습니다. 아래에 작성한 실험 결과는 증강만으로 성능을 향상시킨 결과가 아니라, 최종 모델에 대해 증강 기법만 달리 하여 실험한 결과입니다.

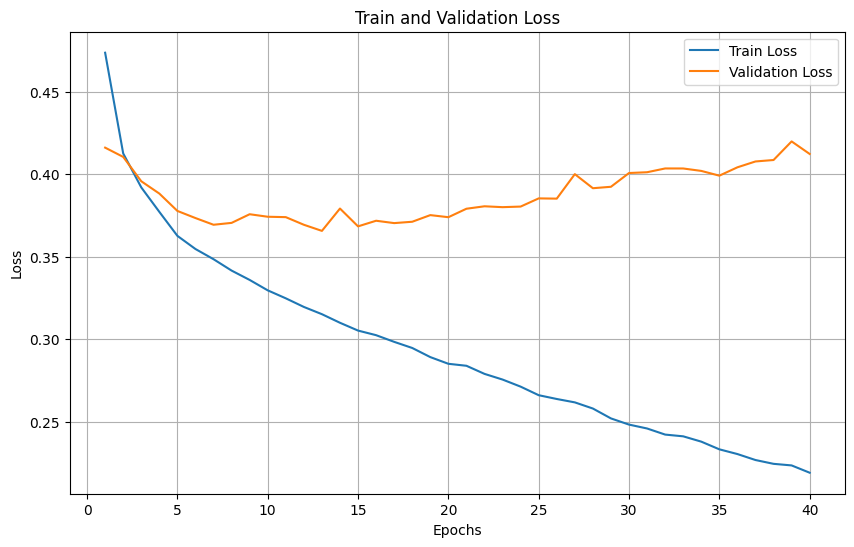
1. 증강을 적용하지 않은 경우

* 증강을 적용하지 않는 경우 모델의 입력 크기와 맞추기 위한 Resize(), 이미지를 평균 0, 표준편차 1로 정규화하는 Normalize()를 적용한 후 텐서로 변환하는 과정만 거쳤습니다.
* 학습 결과, 0.277의 매우 낮은 f1-score를 보였습니다. train loss와 validation loss를 확인한 결과, 10epoch 이상부터는 train loss가 0으로 수렴하고 validation loss는 증가하는 심한 과적합을 보였습니다.



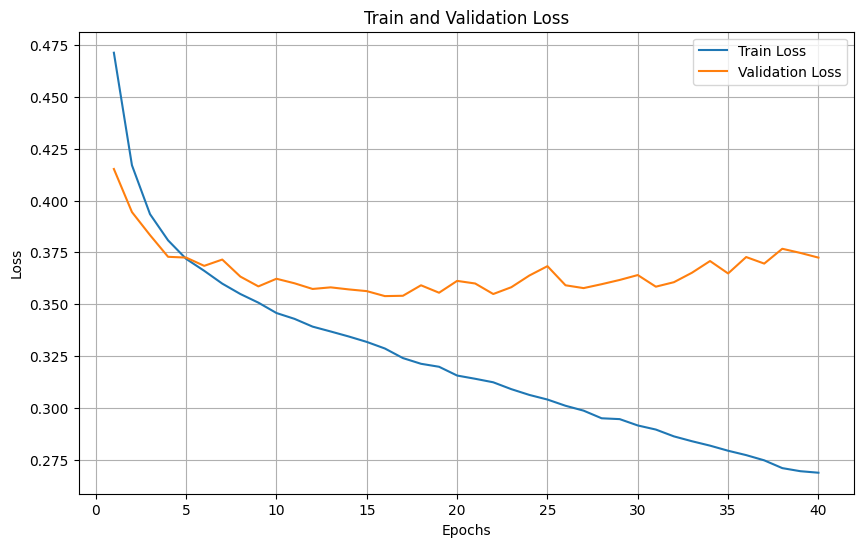
1. 일부 증강만 적용한 경우

* 최종적으로 적용한 증강 기법 중 일부인 네개의 증강 기법을 적용한 후 성능을 확인했습니다. 아래 네개의 증강 기법은 ‘영화 포스터로 장르 분류’라는 태스크를 고려했을 때, 성능 향상이 있을 것이라고 생각한 증강 기법입니다. 각 증강 기법에 대한 설명은 아래와 같습니다.
* RandomResizedCrop() : 이미지 랜덤 크롭 후 지정된 크기(244, 244)로 resize
* HorizontalFlip() : 좌우 반전
* RandomBrightnessContrast() : 밝기, 대비 랜덤 조정
* ShiftScaleRotate() : 이미지 랜덤 이동, 크기 변경, 회전
* 학습 결과, 0.49의 f1-score를 달성했습니다. Loss 그래프를 보면, 증강을 적용하지 않은 경우 1epoch부터 과적합이 발생했던 것과 달리, 과적합이 방지되고 있는 것을 확인할 수 있습니다. 5epoch 이상부터 과적합이 일어나지만 비교적 매우 안정적으로 학습이 진행되는 것을 확인할 수 있습니다.



1. 코드에 기재된 모든 증강 기법을 적용한 경우

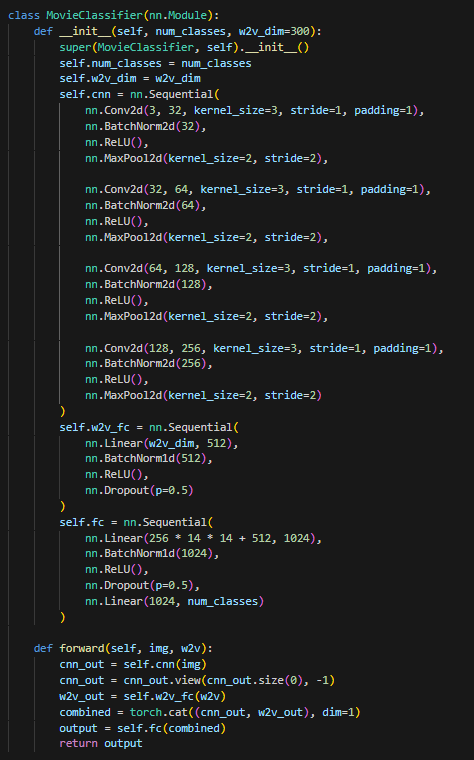
* 2번에서 적용한 4개의 증강 기법에, 아래 세개의 증강기법을 추가적으로 적용하여 성능을 확인했습니다. 추가 적용한 증강 기법에 대한 설명은 아래와 같습니다.
* CoarseDropout() : 이미지의 랜덤한 부분을 사각형으로 가림
* HueSaturationValue : 색조, 채도, 명도 무작위 변경
* GaussianBlur : 가우시안 블러링하여 부드럽게 만듦 - 노이즈 제거
* Train loader를 사용하여 불러온 이미지의 shape는 [224, 224, 3], w2v의 shape는 [64, 300]임을 확인했습니다. 학습 결과, 0.524의 f1-score를 달성했습니다. 위 세 기법은 ‘영화 포스터로 장르 분류’라는 태스크를 고려했을 때, 포스터의 색상이 장르 구분의 요인으로 작용할 수 있고, 블러링 하는 것이 태스크와 무관하다고 생각했기 때문에 성능 향상이 없을 것이라 예상했던 것과 달리, 성능이 향상됨을 확인했습니다.



Step 3: Neural Network 생성

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

<코드 캡쳐 첨부>- 최종 모델 구성



설계한 모델을 출력 후 네트워크를 구성한 방법과 이유를 각 단계별로 설명

Ex) 차원 분석, 채널 수, 커널 크기, linear layer neuron 수

activation function과 선정 이유

최종 layer에서 사용한 activation function이 무엇인지 왜 사용하였는지

CNN 모델이 무엇을 입력 받고 출력하나요? 등…

<서술형>

실습에서 배운 모델 구성 방식을 기반으로 작성하였습니다. 이미지 데이터를 처리하는 cnn, 영화 제목을 벡터로 변환한 데이터를 처리하는 w2v\_fc, 두 결과를 결합하여 최종적으로 처리할 fc로 구성했습니다.

**[1. 이미지 처리하는 self.cnn]**

대체로 좋은 성능을 보인다고 알려진 활성화함수인 ReLU와 풀링 방법 중 Max Pooling을 사용하여 모델을 설계했습니다. 초기에 구성한 모델은 nn.Conv2d(), nn.ReLU(), nn.MaxPool2d()만 사용한 구조였으나, Batch Normalization을 적용한 후 성능이 향상됨을 확인하여 적용했습니다. 이미지를 처리하는 CNN에서도 Drop out을 적용해 보았으나, 오히려 성능이 감소하여 제거했습니다. Batch size가 64인 경우 CNN의 차원 변화는 아래와 같습니다. nn.BatchNorm2d()와 nn.ReLU()는 차원이 변화하지 않기 때문에 변화하는 경우만 기재했습니다.

|  |  |  |
| --- | --- | --- |
| 코드 | 입력 차원 | 출력 차원 |
| nn.Conv2d(3, 32, kernel\_size=3, stride=1, padding=1) | (64, 3, 224, 224) | (64, 32, 224, 224) |
| nn.MaxPool2d(kernel\_size=2, stride=2) | (64, 32, 224, 224) | (64, 32, 112, 112) |
| nn.Conv2d(32, 64, kernel\_size=3, stride=1, padding=1) | (64, 32, 112, 112) | (64, 64, 112, 112) |
| nn.MaxPool2d(kernel\_size=2, stride=2) | (64, 64, 112, 112) | (64, 64, 56, 56) |
| nn.Conv2d(64, 128, kernel\_size=3, stride=1, padding=1) | (64, 64, 56, 56) | (64, 128, 56, 56) |
| nn.MaxPool2d(kernel\_size=2, stride=2) | (64, 128, 56, 56) | (64, 128, 28, 28) |
| nn.Conv2d(128, 256, kernel\_size=3, stride=1, padding=1) | (64, 128, 28, 28) | (64, 256, 28, 28) |
| nn.MaxPool2d(kernel\_size=2, stride=2) | (64, 256, 28, 28) | (64, 256, 14, 14) |

**[2. w2v 데이터 처리하는 self.w2v\_fc]**

w2v 데이터를 처리하는 부분은 매우 간단하게 구현하고, Drop out을 적용했습니다.

|  |  |  |
| --- | --- | --- |
| 코드 | 입력 차원 | 출력 차원 |
| nn.Linear(w2v\_dim, 512) | (64, 300) | (64, 512) |

**[3. Self.fc]**

이미지와 w2v를 각각 처리한 후, 결과를 결합하여 최종 예측을 수행하는 fully connected layer입니다. Drop out을 적용했습니다.

|  |  |  |
| --- | --- | --- |
| 코드 | 입력 차원 | 출력 차원 |
| nn.Linear(256 \* 14 \* 14 + 512, 1024) | (64, 50688) | (64, 1024) |
| nn.Linear(1024, num\_classes) | (64, 1024) | (64, 8) |

**[4. forward()]**

self.cnn()으로 이미지를 처리 후, 결과를 view()로 flatten하여 2D 텐서로 변환합니다. Slef.w2v\_fc()로는 w2v데이터를 처리 후, 두 결과를 torch.cat()으로 결합합니다. 결합한 결과는 최종 fc 레이어의 입력으로 사용하여 최종 예측 결과를 반환합니다.

|  |  |
| --- | --- |
| 코드 | 출력 차원 |
| cnn\_out = cnn\_out.view(cnn\_out.size(0), -1) | (64, 256\*14\*14) |
| combined = torch.cat((cnn\_out, w2v\_out), dim=1) | (64, 256\*14\*14+512) |

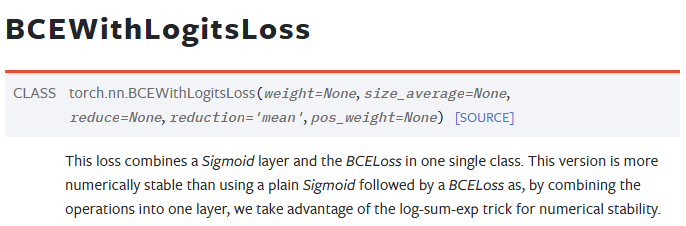
Step 4: Cost (Loss) Function과 Optimizer 선택

<코드 캡쳐 첨부>



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

<서술형>

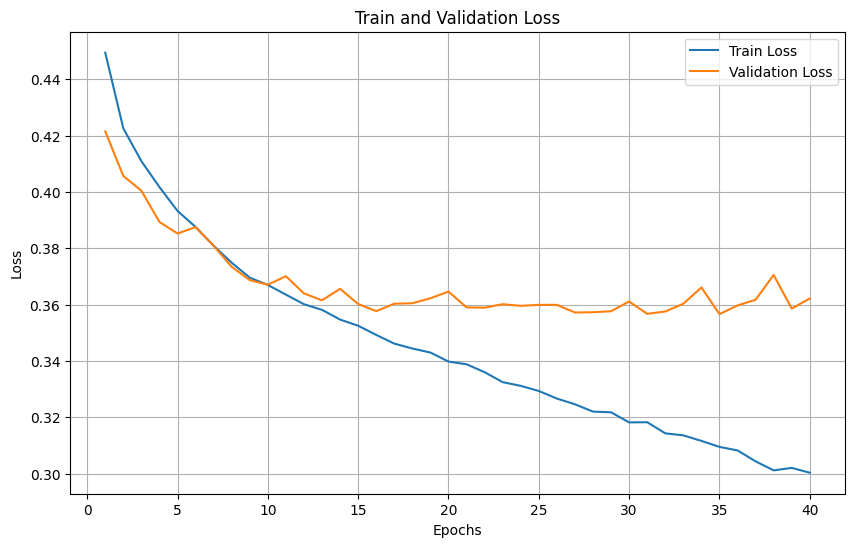


다중 레이블 분류는 Binary Cross Entropy를 많이 사용하고, 그중 BCEWithLogitsLoss는 sigmoid 함수와 Binary Cross-Entropy 손실을 결합하여 안정적인 학습이 가능하다는 장점을 가지기 때문에 선택했습니다. weight\_decay를 사용하여 L2 정규화를 적용했습니다.

Optimizer는 기본이 되는 SGD와, 일반적으로 성능이 좋다고 알려진 Adam을 적용하여 성능을 비교했습니다.

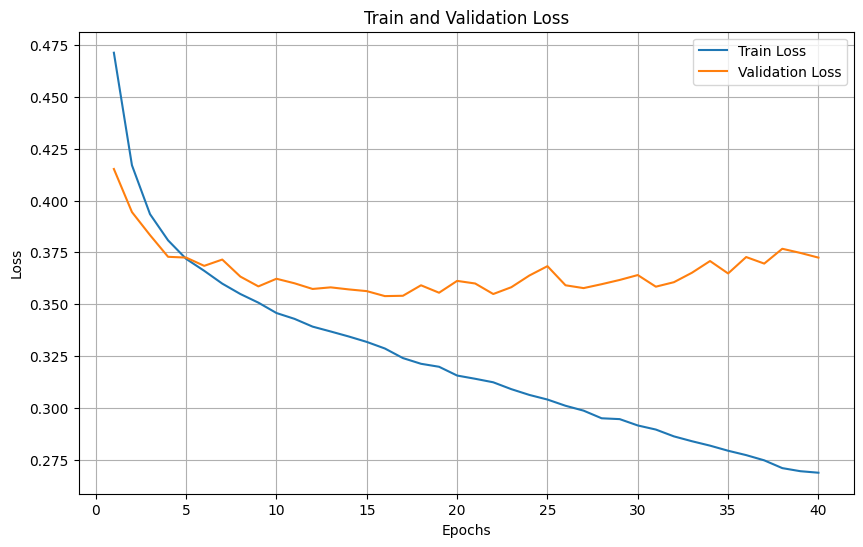
1. SGD

* 코드 : optimizer=optim.SGD(model.parameters(),lr=0.01,momentum=0.9)
* SGD를 Optimizer로 학습한 결과, 0.484의 f1-score를 기록했습니다.



1. Adam

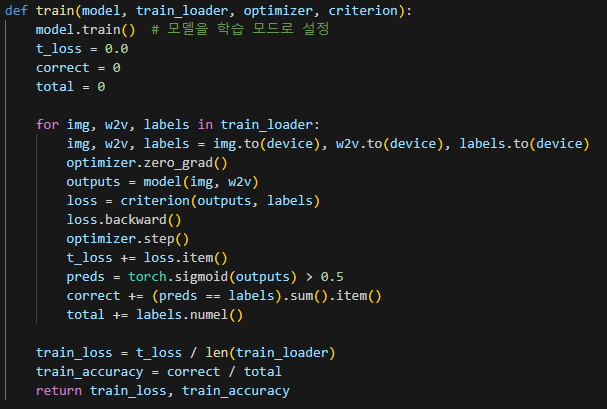
* 코드 : optimizer = torch.optim.Adam(model.parameters(), lr=1e-4, weight\_decay=1e-5)
* Adam을 Optimizer로 학습한 결과, 0.524의 f1-score를 기록했습니다.



Step 5: 구성한 모델에 대한 Train and Validate 진행

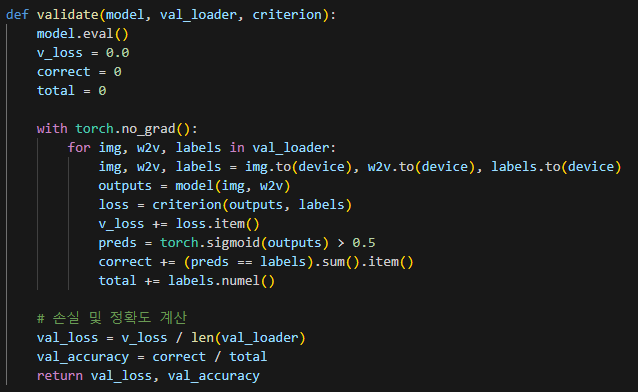
<코드 캡쳐 첨부>

Train 및 validation 코드는 실습 코드를 참고하여 작성했습니다.

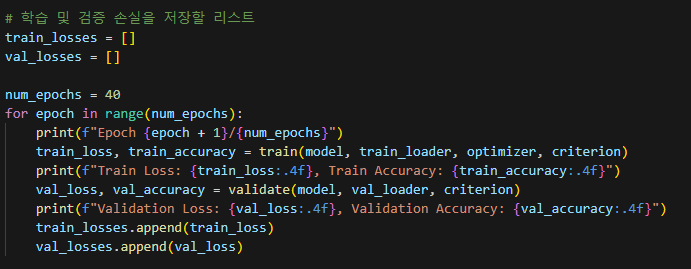


먼저 모델을 학습 모드로 설정합니다. 변수 t\_loss는 train 과정에서의 loss, correct는 올바르게 예측한 샘플의 총 개수, total은 학습 데이터에서 사용된 샘플의 총 개수입니다.

앞서 정의한 train\_loader로 데이터를 불러오고, GPU 사용을 위해 데이터를 이동시킵니다. 학습 과정은 “옵티마이저 초기화 – 순전파 – 손실 계산 – 역전파 – 파라미터 업데이트 – 손실 누적 – 예측 – 정확도 계산 – total 증가”의 순서로 진행됩니다. 예측에서는 시그모이드 함수와 임계값 0.5를 사용하여 0.5 이상이면 1로 예측하도록 합니다. 전체 데이터에 대해 해당 과정을 마치면, epoch 당 평균 손실과 정확도를 반환합니다.



위 코드는 Validation 과정입니다. 모델을 평가모드로 전환하고, val\_loader로 데이터를 로드한 후 모델로 손실, 예측, 정확도를 계산합니다.

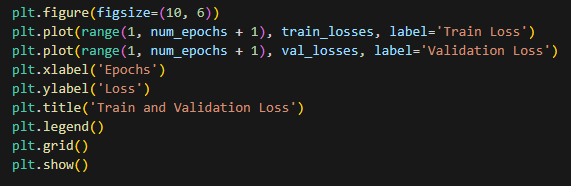


학습시에는 앞서 정의한 두 함수 train()과 validate()를 사용하여, 설정한 epoch수만큼 for문으로 반복합니다. 후에 epoch 별 loss를 그래프로 확인하기 위해, loss를 train\_losses와 val\_losses 리스트에 각각 저장합니다. Validation loss가 5번 연속 증가하면 early stop 되도록 구현하기도 했으나, 확인 결과 loss와 성능이 비례하지 않고, early stop을 적용한 경우보다 임의로 epoch을 설정하는 것이 더 높은 성능을 보여 최종 모델에는 적용하지 않았습니다.

Step 6: CNN model training/validation 분석

<코드 캡쳐 첨부>

Train loss와 validation loss를 시각화하는 데 사용한 코드는 아래와 같습니다.



위에서 수행한 training + validation 과정을 설명하세요

training loss와 validation loss 그래프를 통해서 분석

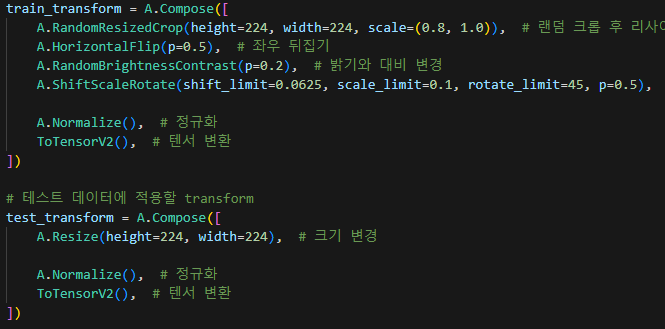
Ex) hyper-parameter, model을 변경하면서 성능 개선한 과정을 최대한 설명하세요

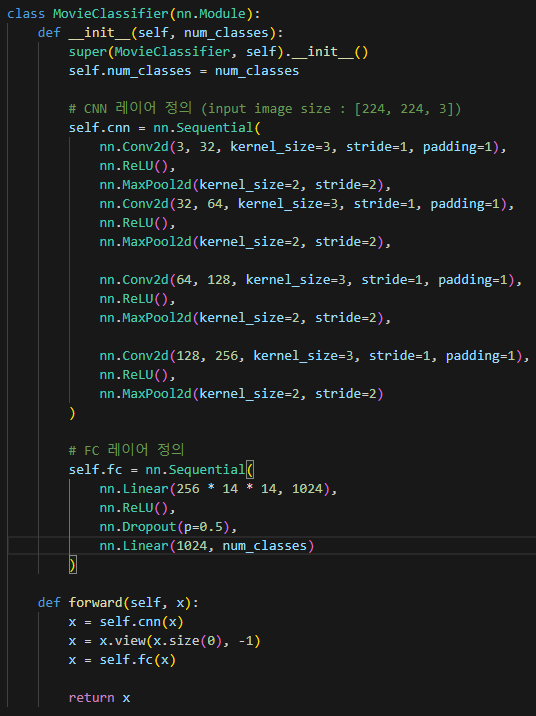
overfitting, underfitting 분석 등..

<서술형>

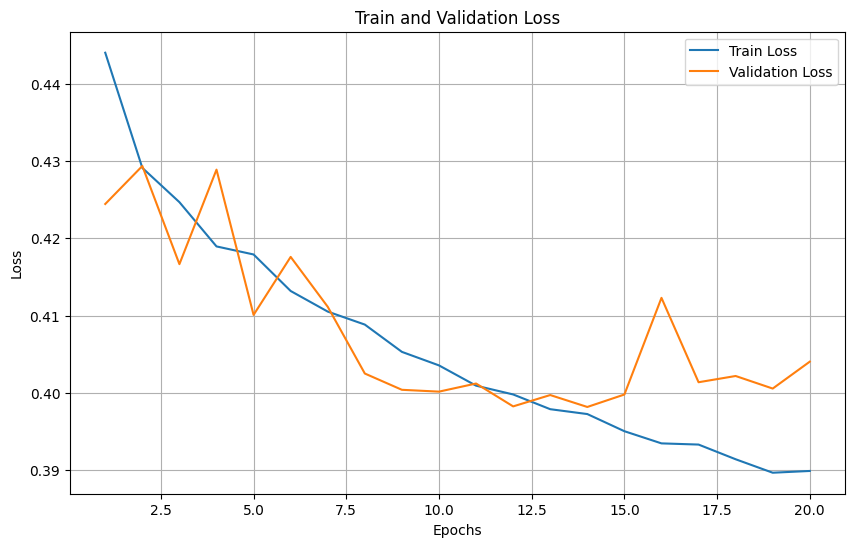
성능 향상에 도움이 된 시도

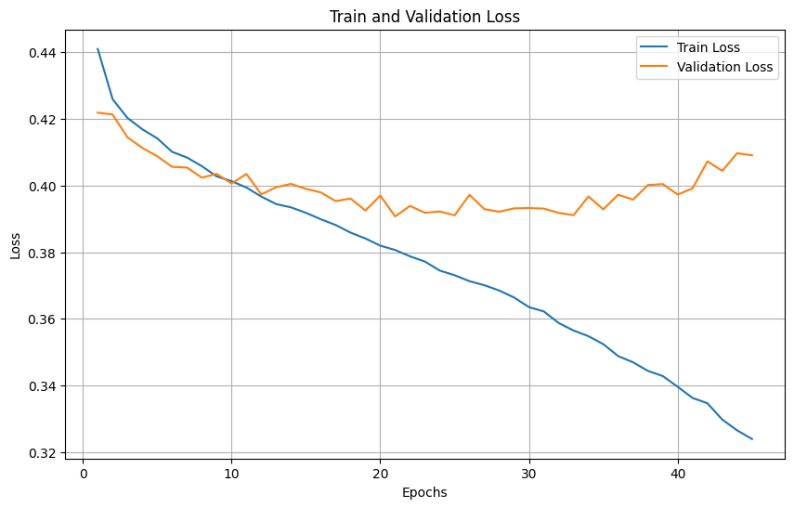
**[시도 1. 초기 모델 구성]**

****



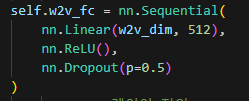
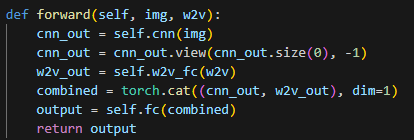
가장 처음으로 구성했던 모델의 코드입니다. 초기 코드에는 w2v 데이터를 사용하지 않고, 영화 포스터 이미지만 사용하여 모델을 학습했습니다. 증강에는 앞서 설명한 1차 증강 (RandomResizedCrop, HorizontalFlip, RandomBrightnessContrast, ShiftScaleRotate)과 정규화, 텐서 변환을 적용했습니다. 배치사이즈 32, learning rate는 1e-3으로 하여 학습한 결과, f1-score는 0.302이고, loss 그래프는 아래와 같이 나타났습니다.



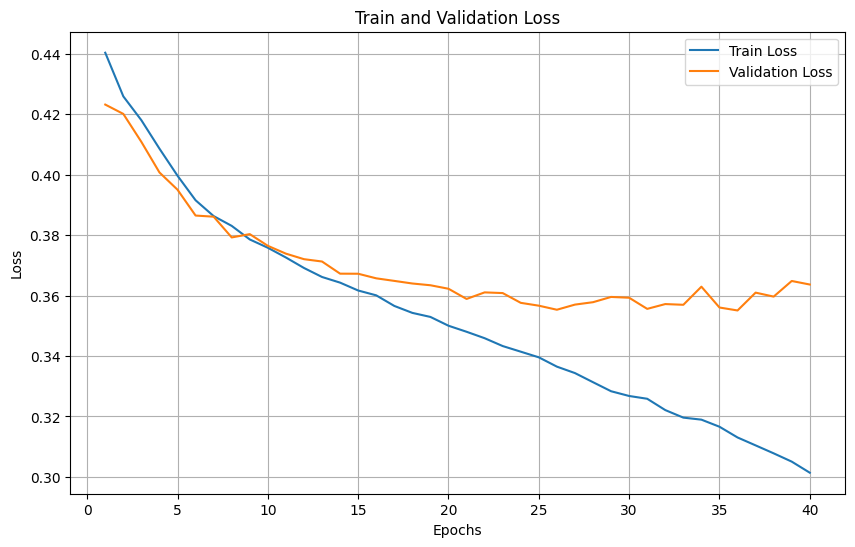
validation loss가 뾰족하게 튀며 학습되는 것을 보고 학습이 안정적이지 않다고 생각했습니다. 그래서 learning rate를 1e-4로 감소시키고, Learning rate를 줄였기 때문에 학습이 더 느리게 진행될 것이라 생각해, epoch도 45로 늘려서 학습시켰습니다. 그리고 batch size가 64인 경우와 256인 경우의 성능을 비교했습니다.

왼쪽이 batch size가 256인 경우이며, f1-score가 0.368로 나타났습니다. 오른쪽은 batch size가 64인 경우이며, f1-score가 0.418로 나타났습니다. 따라서 이후에는 batch size를 64로 사용했습니다.

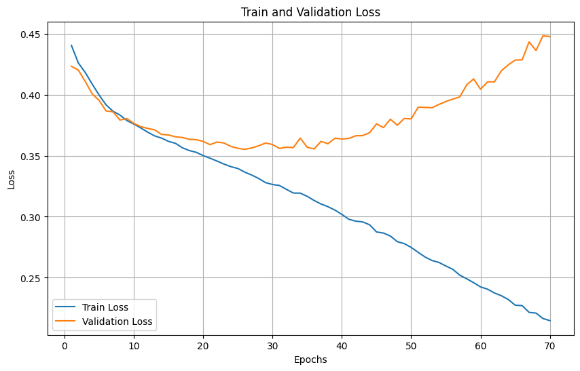
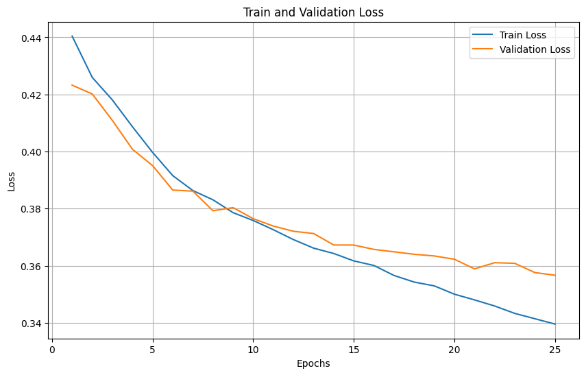
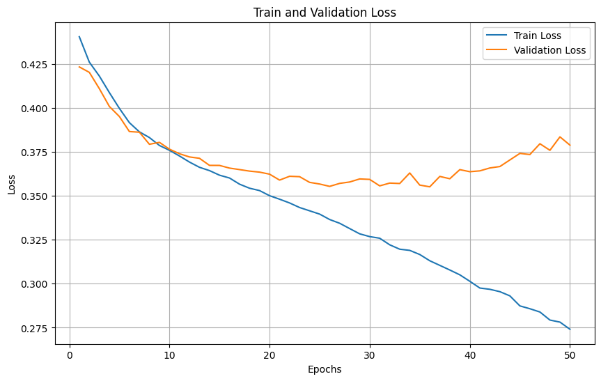
**[시도 2. w2v 데이터 사용]**



W2v 데이터를 사용하기 위해 모델에 w2v 데이터를 처리하는 self.w2v\_fc를 위와 같이 간단하게 구성하고, 이미지를 처리하는 cnn의 출력화 w2v\_fc의 출력을 torch.cat()으로 결합하여 fully connected layer의 입력으로 사용하도록 했습니다. 배치사이즈 64, learning rate는 e-4로 설정하여 40 epoch 학습한 결과 f1-score는 0.483으로 크게 향상되었고, loss 그래프는 아래와 같습니다.

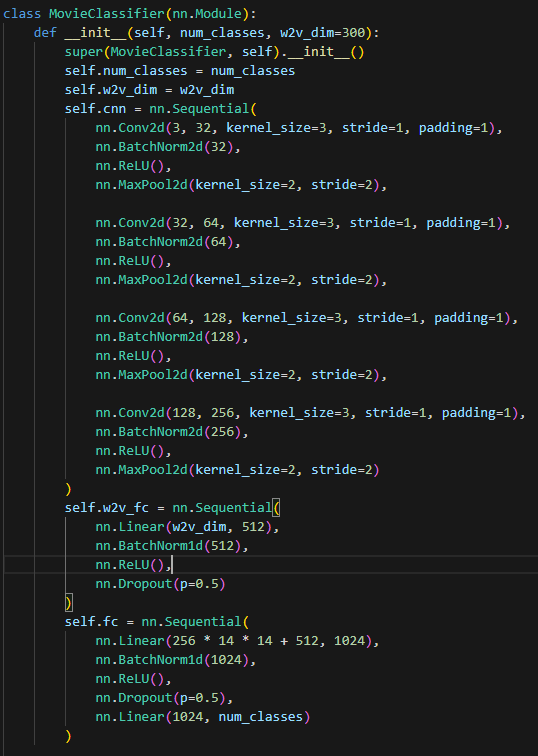


성능은 향상되었지만, 과적합이 발생하기 때문에 validation loss가 더 이상 감소하지 않는 25epoch에서 학습을 중단시켜보았습니다. 하지만 예상과 달리 f1-score가 0.462로 감소하였고, 그래서 epcoh을 25, 70으로 늘려서 학습해 보았습니다.

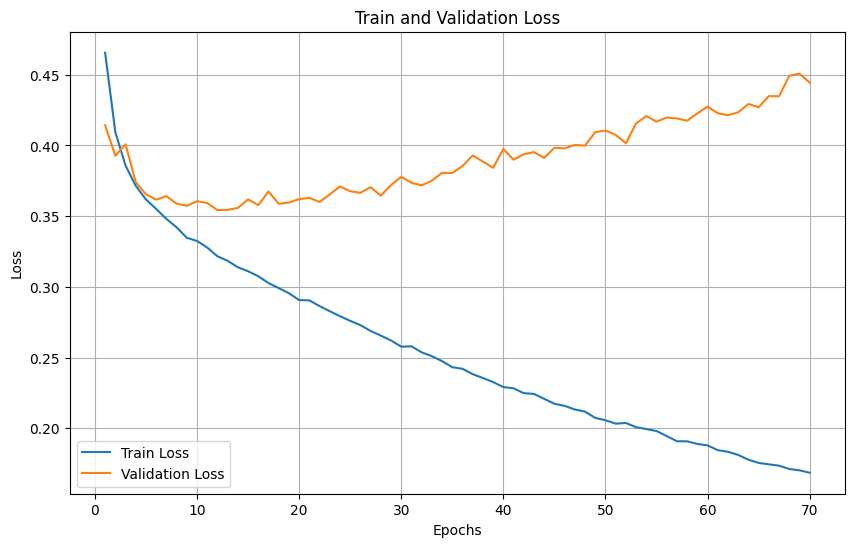


왼쪽부터 순서대로 25, 50, 70 epoch인 경우의 loss 그래프입니다. 과적합이 일어나 성능이 감소할 것이라 예상한 것과 달리, 순서대로 0.462, 0.487, 0.478의 f1-score를 보이며 epoch이 50인 경우 가장 높은 성능을 보였습니다.

**[시도 3. Batch normalization]**

****

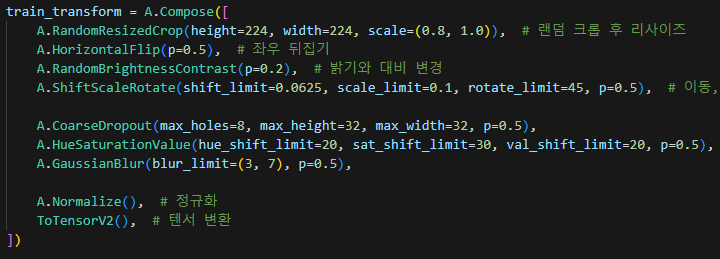
수업에서 배운 내용들을 적용해보자는 생각으로, 위와 같이 모델에 batch norm을 적용했습니다. Batch norm을 적용하여 70 epoch 학습한 결과, f1-score가 0.516으로 향상되었습니다.



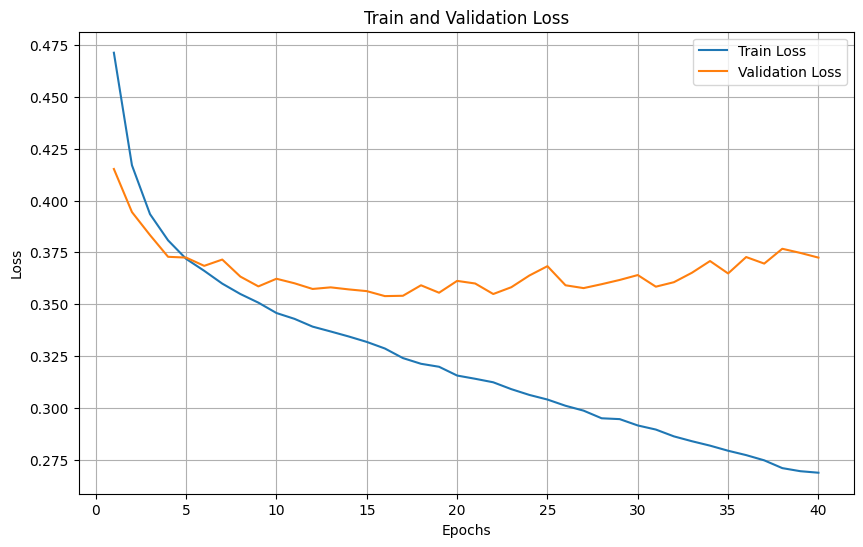
이전에 과적합에도 불구하고 epoch을 늘렸을 때 성능이 향상되었기 때문에 20, 50, 70 epoch으로 학습하여 성능을 비교한 결과 70epoch을 학습한 경우 가장 높은 성능을 보였습니다.

**[시도 4. 추가적인 이미지 transform ]**

처음부터 적용했던 4개의 이미지 transform 방식 외에, CoarseDropout(), HueSaturationValue(), GaussianBlur()를 추가적으로 적용했습니다.



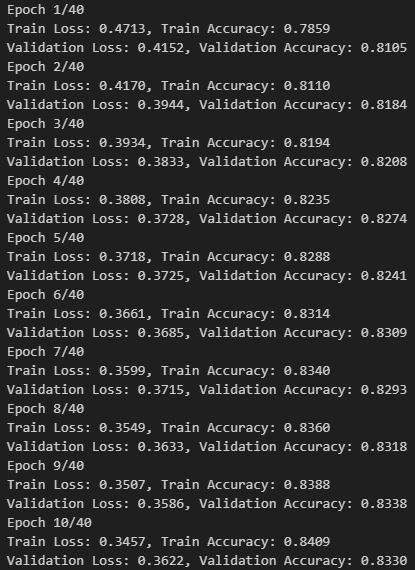
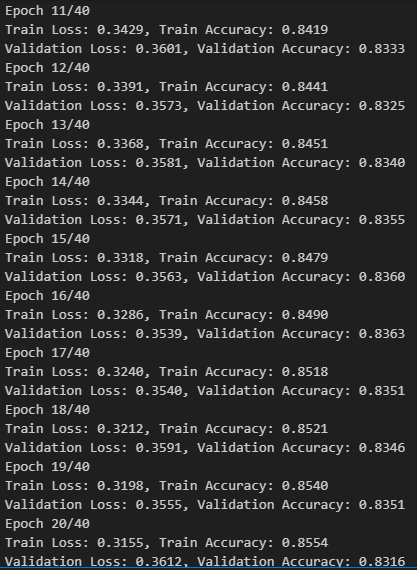
적용한 결과, f1-score 0.524를 달성하였으며, 아래와 같은 loss 그래프를 확인하였습니다.

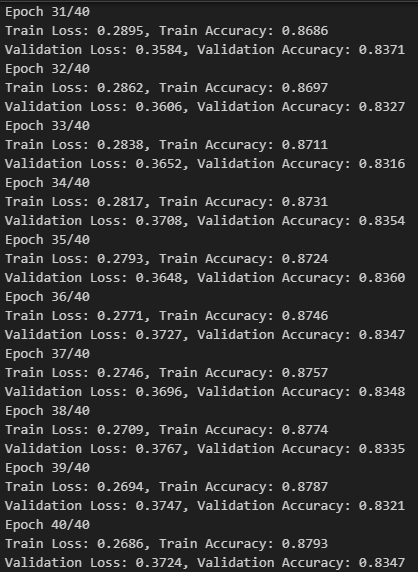
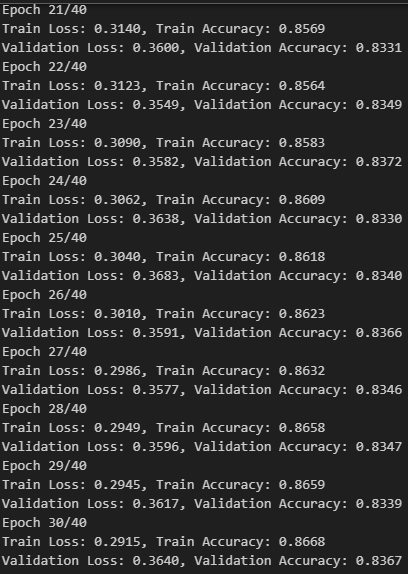


**[시도 5. 최종 제출 : 전체 데이터에 대하여 학습]**

이렇게 완성한 모델에 대하여, 최종 제출에는 데이터를 학습과 검증용으로 나누지 않고, 모두 학습에 사용하여 제출했습니다. 그 결과 private에서 f1-score 0.538를 달성하며 성능이 향상됨을 확인했습니다.

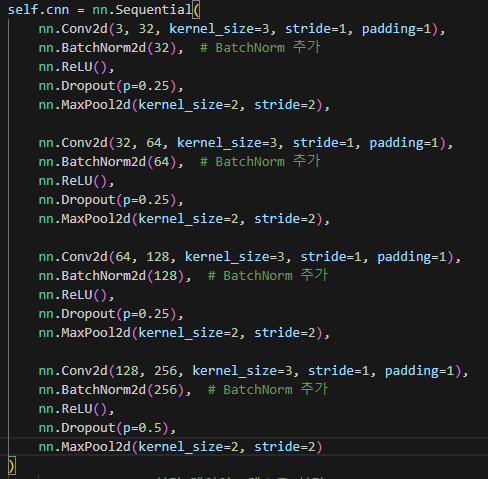


최종 모델의 학습 로그는 아래와 같습니다.



성능 향상에 도움이 되지 않은 시도

**[시도 1. 이미지 처리 CNN에서의 Drop out]**



(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) Drop out과 관련된 논문 리뷰 내용을 참고하여, 한번의 합성곱층 + 풀링레이어가 끝날 때 마다 0.25의 드롭아웃을 적용하고 마지막에는 0.5드롭아웃을 적용했을 때 가장 성능이 높았다는 결과를 보고 동일하게 적용하였습니다. 하지만 예상과 달리 f1-score가 0.524에서 0.486으로 감소하여 이미지를 처리하는 CNN에는 적용하지 않았습니다. 최종 코드에서 Drop out은 fully connected layer에만 적용하였습니다.

**[시도 2. Learning rate scheduler]**

(https://panggu15.github.io/etc/pytorch-learning-rate-Scheduler/#google\_vignette)

최종 모델에 대하여, 위 사이트를 참고하여 LambdaLR, MultiplicativeLR, StepLR, ReduceLROnPlateau를 적용해 보았습니다. 결론적으로 네 방식 중 MultiplicativeLR를 적용한 경우 가장 높은 성능을 보였으나, scheduler를 적용하지 않은 경우와 비교했을 때 성능이 향상되지 않아 사용하지 않았습니다.

1. LambdaLR

* Lambda 표현식으로 작성한 함수를 통해 learning rate를 조절
* F1-score : 0.511
* scheduler = torch.optim.lr\_scheduler.LambdaLR(optimizer=optimizer, lr\_lambda=lambda epoch: 0.95 \*\* epoch)

1. MultiplicativeLR

* 초기 learning rate에 lambda함수에서 나온 값을 누적곱해서 learning rate를 계산
* F1-scroer : 0.517
* scheduler = torch.optim.lr\_scheduler.MultiplicativeLR(optimizer=optimizer, lr\_lambda=lambda epoch: 0.95 \*\* epoch)

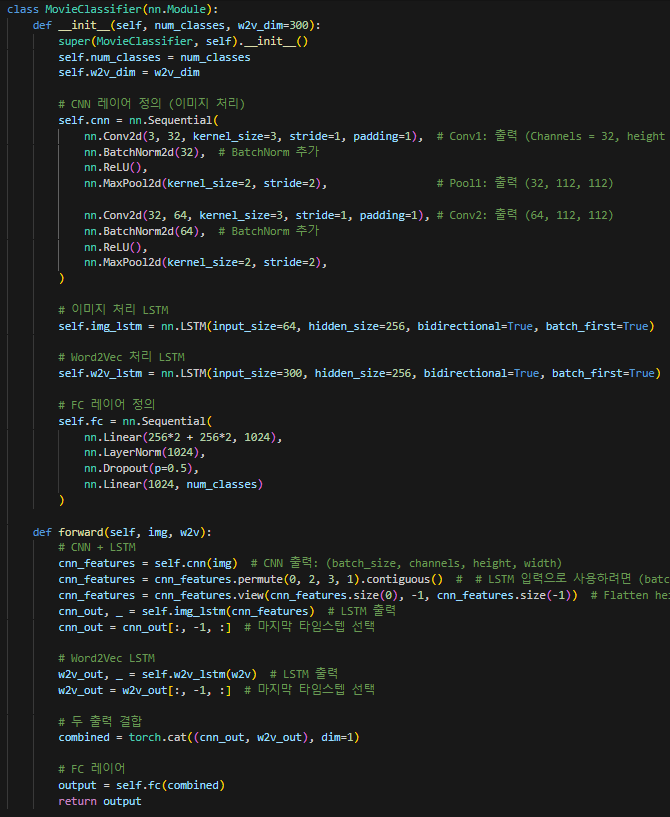
1. StepLR

* 특정 Step에 따라 lr 를 감소
* F1-score : 0.519
* scheduler = torch.optim.lr\_scheduler.StepLR(optimizer, step\_size=20, gamma=0.5)

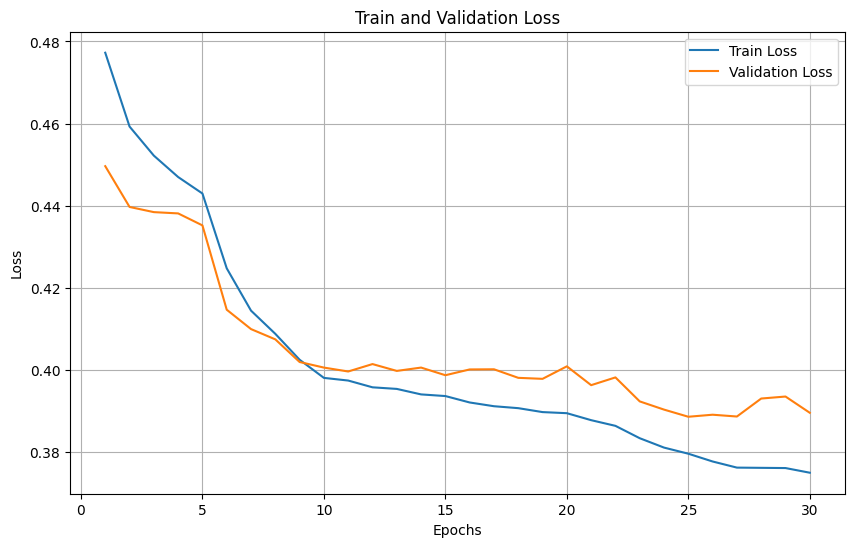
1. ReduceLROnPlateau

* 성능이 향상이 없을 때 learning rate를 감소
* F1-score : 0.504
* scheduler = lr\_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=5, verbose=True)

**[시도 3. CNN + LSTM 모델]**



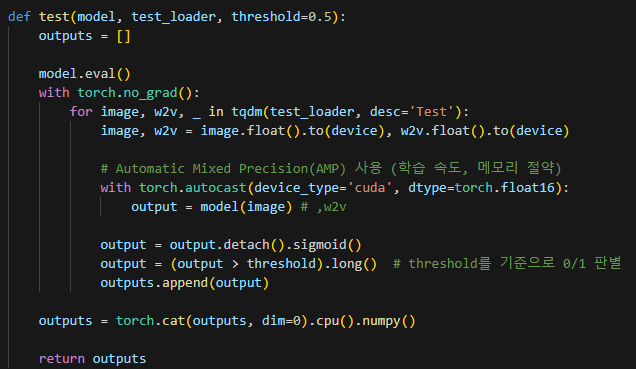
w2v는 텍스트 데이터이니 LSTM으로 분류를 해봐야겠다는 생각과, 이미지 분류에도 LSTM을 활용할 수 있다는 것을 알게 되어 LSTM을 활용한 분류 모델을 구성했습니다. CNN과 LSTM을 결합하여 이미지를 분류하는 코드 (<https://rgbitcode.com/blog/senspond/72>)를 참고하여 CNN으로 이미지를 1차 처리한 후, LSTM으로 학습하는 모델을 구성하였습니다. CNN의 출력인 (64, 64, 56, 56)차원을 LSTM의 입력으로 사용하기 위해 permute()를 사용하여 (64, 56, 56, 64)의 형태로 바꿔줍니다. w2v의 경우 CNN의 입력으로 사용할 때는 MovieDataset()에서 w2v = torch.mean(w2v, dim=0)로 차원을 축소했습니다. 하지만 LSTM의 경우 차원을 축소할 필요가 없어, 해당 코드를 제거했습니다. 즉, w2v의 LSTM 입력 차원은 (64, 15, 300)이 됩니다.



lr=1e-4로 30 epoch 학습한 결과, 안정적인 loss 감소를 보였지만 f1-score가 0.399로 기존의 CNN 모델보다 낮아 사용하지 않았습니다.

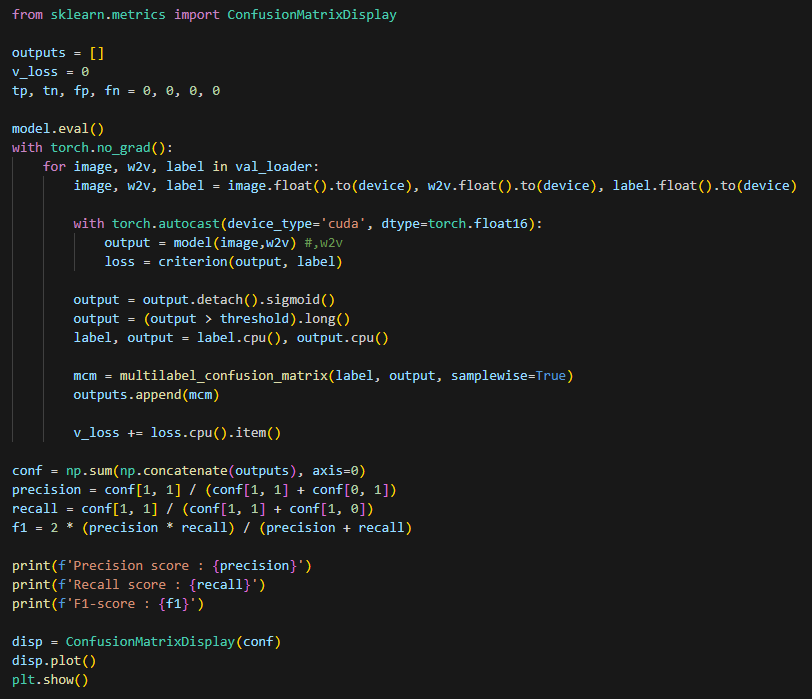
Step 7: Predict with Test Data

<코드 캡쳐 첨부>

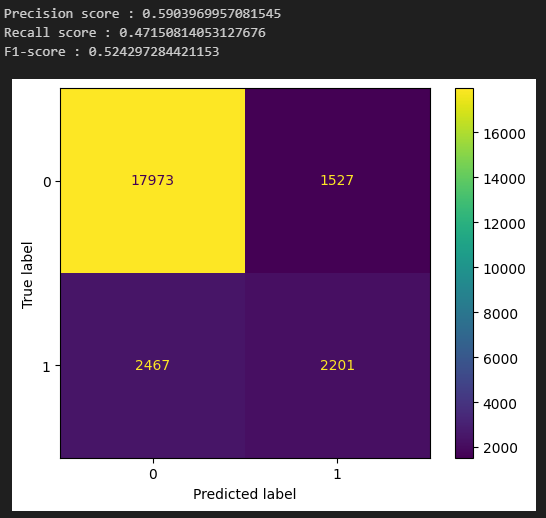




베이스라인에서 제공된 코드와 동일한 함수를 사용하여 test data에 대한 예측 및 평가를 진행했습니다. Confusion matrix를 확인하는 코드는 아래와 같습니다



위 코드로 확인한 최종 모델의 성능 및 confusion matrix는 아래와 같습니다.



Step 8: Training Techniques

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

**[1. Batch size 조절]**

시도 1에서 batch size에 따른 성능을 비교해보며 batch size가 64에 f1-score가 0.418, batch size가 256인 경우에 f1-score가 0.368임을 확인했습니다. 이처럼 같은 데이터를 같은 방식으로 학습했을 뿐인데, batch size를 조절하는 것 만으로도 성능에 큰 변화가 생긴다는 것이 놀라웠습니다. 더불어 learning rate나 epoch과 같은 하이퍼파라미터를 여러 번 시도해보고 적절하게 설정하는 것이 얼마나 모델의 성능 개선에 중요한 과정인지 깨닫게 되었습니다.

**[2. w2v 데이터 적용]**

w2v 데이터를 적용한 경우, f1-score가 0.418에서 0.483로 크게 증가했습니다. 해당 데이터를 추가했을 때 성능이 향상됨은 예상했던 바였지만, w2v 데이터를 처리하는 w2v\_fc를 매우 간단하게 구현했음에도 불구하고 예상보다 더 성능 향상에 효과적이었습니다. 이러한 성능 변화를 통해, 다양한 데이터의 중요성을 깨달았고, 멀티모달 모델을 통한 성능 개선을 경험할 수 있었습니다.

**[3. 여러 번 시도하는 경험]**

본 프로젝트를 진행하며 이론적으로 배운 모델의 성능 향상 방식들을 직접 적용해볼 수 있는 기회가 되었습니다. CNN을 활용한 문제해결 경험이 많지 않다 보니, 해결하고자 하는 태스크와 모델에 적용하는 기법 간의 관계성이 명확히 파악되지 않았고, 기법이 모델의 성능에 미칠 영향에 대한 감이 없었기 때문에 각 방식의 수학적, 이론적 차이를 비교하며 결과를 예상하고 적용하기보다는 이것저것 되는대로 사용해보며 성능을 확인해야 했습니다. 이러한 여러 번의 시도가 본 프로젝트에서는 모델 성능의 향상으로 이어졌습니다. 하지만 실제로 더 많은 데이터와 복잡한 문제를 해결하려고 할 때는 시간 등 여러 자원적 문제로 모든 시도를 해볼 수 없고, 결국 이전의 경험들에 의해 해결하고자 하는 태스크에 맞는 기법들을 선택하여 시도해야 할 것이라고 생각합니다. 때문에 본 프로젝트와 같이 여러 시도를 적용해 보는 것을 다양한 실습 태스크에서 경험하고 고민해보는 것이 중요하다는 것을 깨달았습니다. 더불어 빠른 학습으로 더 많은 시도를 해볼 수 있도록 GPU와 같은 개발 환경 역시 매우 중요하다는 것을 체감하게 되었습니다.