

## MIR Assignment 2 Report

20191048 김도솔

### Problem 1: Complete Three Dataset Classes (21 pts)

#### 1) OnTheFlyDataset

```
def __getitem__(self, idx):  
  
    audio_sample = None  
    label = None  
  
    # 1) Get the file path of idx-th data sample (use self.labels['mp3_path'])  
    file_path = self.labels['mp3_path'].iloc[idx]  
  
    # 2) Load the audio of that file path  
    audio, sr = torchaudio.load(self.dir / file_path)  
  
    # 3) Resample the audio sample into frequency of self.sr  
    resampler = torchaudio.transforms.Resample(orig_freq=sr, new_freq=self.sr)  
    audio_sample = resampler(audio)  
  
    # 4) Return resampled audio sample and the label (tag data) of the data sample  
    label = self.label_tensor[idx]  
    audio_sample = audio_sample[0] # Number of dimensions of audio tensor has to be 1  
  
    return audio_sample, label
```

**코드 설명:** `__getitem__` 함수는 특정 인덱스(idx)에 해당되는 데이터 샘플을 불러오는 역할을 한다. 먼저 `self.labels['mp3_path']`를 이용해 특정 인덱스의 데이터 샘플에 해당하는 MP3 파일 경로를 구한다. 다음으로 `torchaudio.load` 함수를 이용해 파일을 불러오고, 불러온 오디오 샘플과 해당 오디오의 샘플링 레이트를 `audio`와 `sr` 변수에 저장한다. 그 후 `torchaudio.transforms.Resample`을 이용하여 오디오 샘플을 원하는 샘플링 레이트(`self.sr`)로 변환한다. 마지막으로 리샘플된 오디오 샘플과 해당 샘플의 태그 정보인 `label`을 반환한다. 오디오 샘플의 경우, 텐서 차원이 1차원이 되도록 조정해야 하므로 `audio_sample[0]`을 사용했다.

#### 결과 출력:

```
dummy_set = OnTheFlyDataset(MTAT_DIR, split='train', num_max_data=100)  
audio, label = dummy_set[1]  
assert audio.ndim == 1, "Number of dimensions of audio tensor has to be 1. Use audio[0]  
ipd.display(ipd.Audio(audio, rate=dummy_set.sr))  
print(dummy_set.vocab[torch.where(label)])
```

▶ 0:00 / 0:29 ———— 🔊 ⋮

ambient

## 2) PreProcessDataset

```
def pre_process_and_save_data(self):

    idx = 0

    for mp3_path in self.labels['mp3_path'].values:
        # 1) Check whether pre-processed data already exists
        preprocessed_path = Path(self.dir / mp3_path).with_suffix('.pt')

        # 2) If it doesn't exist, do follow things
        if not preprocessed_path.exists():
            # a) Load audio file
            audio, sr = torchaudio.load(self.dir / mp3_path)

            # b) Resample the audio file with samplerate of self.sr
            resampler = torchaudio.transforms.Resample(orig_freq=sr, new_freq=self.sr)
            audio_sample = resampler(audio)

            # c) Get label of this audio file
            label = self.label_tensor[idx]

            # d) Save {'audio': audio_tensor, 'label':label_tensor} with torch.save
            data = {'audio': audio_sample[0], 'label': label}
            torch.save(data, preprocessed_path)

    idx += 1
```

**코드 설명:** pre\_process\_and\_save\_data 함수는 모든 오디오 파일을 사전 처리하고 .pt 파일로 저장하는 역할을 한다. 먼저 데이터셋 내 각 샘플에 대해 사전 처리된 파일이 존재하는지 확인한다. 사전 처리된 파일의 경로는 원래의 mp3 파일 경로에서 확장자를 .pt로 변경하여 설정한다. 사전 처리된 파일이 존재하지 않을 경우 torchaudio.load를 사용해 오디오 파일을 불러온다. 다음으로 torchaudio.transforms.Resample을 이용해 불러온 오디오를 원하는 샘플링 레이트(self.sr)로 리샘플링한다. idx 변수를 이용해 데이터셋에서 해당 idx의 레이블 정보를 가져온다. 이때 idx 값은 for문을 돌 때마다 1씩 증가하도록 설정했다. 마지막으로 리샘플링된 오디오와 레이블 정보를 preprocessed\_path에 .pt 파일 형태로 저장한다.

```
def __getitem__(self, idx):

    # 1) Get the pt file path of idx-th data sample
    mp3_path = self.labels['mp3_path'].iloc[idx]
    preprocessed_path = Path(self.dir / mp3_path).with_suffix('.pt')

    # 2) Load the pre-procssed data of that file path
    data = torch.load(preprocessed_path)

    # 3) Return the audio sample and the label (tag data) of the data sample
    return data['audio'], data['label']
```

**코드 설명:** `__getitem__` 함수는 특정 인덱스(`idx`)에 해당되는 데이터 샘플을 불러오는 역할을 한다. 먼저 `self.labels['mp3_path']`를 이용해 특정 인덱스의 데이터 샘플에 해당하는 MP3 파일 경로를 구한다. 다음으로 `mp3_path`를 이용해 오디오 데이터의 `.pt` 파일 경로를 구한다. 다음으로 `torch.load`를 사용하여 사전 처리된 데이터 파일을 불러온다. 마지막으로 불러온 데이터에서 오디오 샘플과 레이블 정보를 추출하여 반환한다.

**결과 출력:**

```
dummy_set = PreProcessDataset(MTAT_DIR, split='train', num_max_data=100)
audio, label = dummy_set[15]
assert audio.ndim == 1, "Number of dimensions of audio tensor has to be 1. Use audio[0]"
ipd.display(ipd.Audio(audio, rate=dummy_set.sr))
print(dummy_set.vocab[torch.where(label)])
```

▶ 0:00 / 0:29 ————— 🔊 ⋮

['guitar' 'male']

### 3) OnMemoryDataset

```
def load_audio(self):
    total_audio_datas = []

    ### Write your code from here

    for mp3_path in self.labels['mp3_path'].values:

        # a) Load Audio
        audio, sr = torchaudio.load(self.dir / mp3_path)

        # b) Resample it to self.sr
        resampler = torchaudio.transforms.Resample(orig_freq=sr, new_freq=self.sr)
        audio_sample = resampler(audio)

        # c) Append it to total_audio_datas
        total_audio_datas.append(audio_sample[0])

    return total_audio_datas
```

**코드 설명:** load\_audio 함수는 모든 오디오 데이터를 메모리에 로드한 후 저장하는 역할을 한다. 먼저 self.labels['mp3\_path']를 이용하여 모든 MP3 파일 경로를 반복문을 통해 하나씩 불러온다. 다음으로 torchaudio.load 함수를 이용해 파일을 불러오고, 불러온 오디오 샘플과 해당 오디오 샘플의 샘플링 레이트를 audio와 sr 변수에 저장한다. 다음으로 torchaudio.transforms.Resample를 사용해 원래의 샘플링 레이트(sr)에서 self.sr로 리샘플링한다. 그 후 리샘플링된 오디오 샘플을 total\_audio\_datas 리스트에 추가한다. 반복문이 종료되면 이 리스트는 모든 오디오 데이터를 포함하게 된다. 이런 방식을 사용하면 메모리 사용량이 증가할 수 있지만 데이터 접근 시간은 감소한다는 장점이 있다.

```
def __getitem__(self, idx):

    # 1) Load the pre-procossed audio data from self.loaded_audios
    audio_sample = self.loaded_audios[idx]

    # 2) Return the audio sample and the label (tag data) of the data sample
    label = self.label_tensor[idx]

    return audio_sample, label
```

**코드 설명:** \_\_getitem\_\_ 함수는 특정 인덱스(idx)에 해당되는 데이터 샘플을 불러오는 역할을 한다. 먼저 idx에 해당하는 사전 로드된 오디오 데이터를 self.loaded\_audios에서 가져온다. 다음으로 해당 인덱스에 대응하는 레이블 정보(self.label\_tensor[idx])도 함께 불러온다. 마지막으로 불러온 오디오 샘플과 레이블을 반환한다.

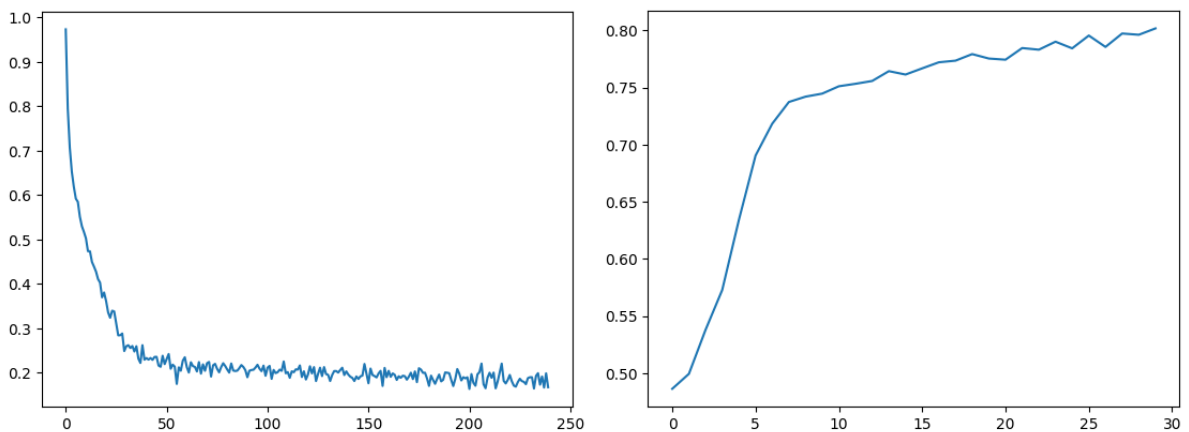
결과 출력:

```
dummy_set = OnMemoryDataset(MTAT_DIR, split='train', num_max_data=50)
audio, label = dummy_set[10]
assert audio.ndim == 1, "Number of dimensions of audio tensor has to be 1. Use audio[0]"
ipd.display(ipd.Audio(audio, rate=dummy_set.sr))
print(dummy_set.vocab[torch.where(label)])
```

▶ 0:00 / 0:29 — 🔊 ⋮

['classical' 'quiet' 'ambient' 'string' 'harp' 'slow']

#### 4) Train the default model



> 모델 트레이닝이 정상적으로 이루어지는 것을 확인할 수 있다.

## Problem 2: Practice with nn.Sequential() (5 pts)

```
class StackManualLayer(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Conv1d(16, 4, kernel_size=2)
        self.activation = nn.Sigmoid()
        self.layer2 = nn.Conv1d(4, 4, kernel_size=2)
        self.layer3 = nn.Conv1d(4, 1, kernel_size=2)

    def forward(self, x):
        out = self.layer1(x)
        out = self.activation(out)
        out = self.layer2(out)
        out = self.activation(out)
        out = self.layer3(out)
        return out
```

```
class SequentialLayer(nn.Module):
    def __init__(self):
        super().__init__()
        ...
        TODO: Complete this nn.Sequential so that it computes exactly same thing with StackManual
        ...

        self.layers = nn.Sequential(
            nn.Conv1d(16, 4, kernel_size=2),
            nn.Sigmoid(),
            nn.Conv1d(4, 4, kernel_size=2),
            nn.Sigmoid(),
            nn.Conv1d(4, 1, kernel_size=2),
        )
    def forward(self, x):
        out = self.layers(x)
        return out
```

**코드 설명:** StackManualLayer와 동일하게 동작할 수 있도록 StackManualLayer class의 \_\_init\_\_과 forward 함수를 참고해 코드를 구성했다. self.layer1(x)에 해당되는 nn.Conv1d(16, 4, kernel\_size=2)를 쌓고, 다음으로 self.activation(out)에 해당되는 nn.Sigmoid()를 쌓는다. 이러한 과정을 self.layer3(out)까지 반복하여 nn.Sequential 안에 여러 신경망 레이어를 순차적으로 구성할 수 있도록 하였다.

### 결과 출력:

```
Output with Manual Stack Layer: tensor([[[[0.0241, 0.0263, 0.0280, 0.0291, 0.0300]]],
      grad_fn=<ConvolutionBackward0>)
Output with Sequential Layer: tensor([[[[0.0241, 0.0263, 0.0280, 0.0291, 0.0300]]],
      grad_fn=<ConvolutionBackward0>)
```

> StackManualLayer와 SequentialLayer의 출력 값이 동일한 것을 확인할 수 있다.

### Problem 3. Make Your Own Conv Layers (15 pts)

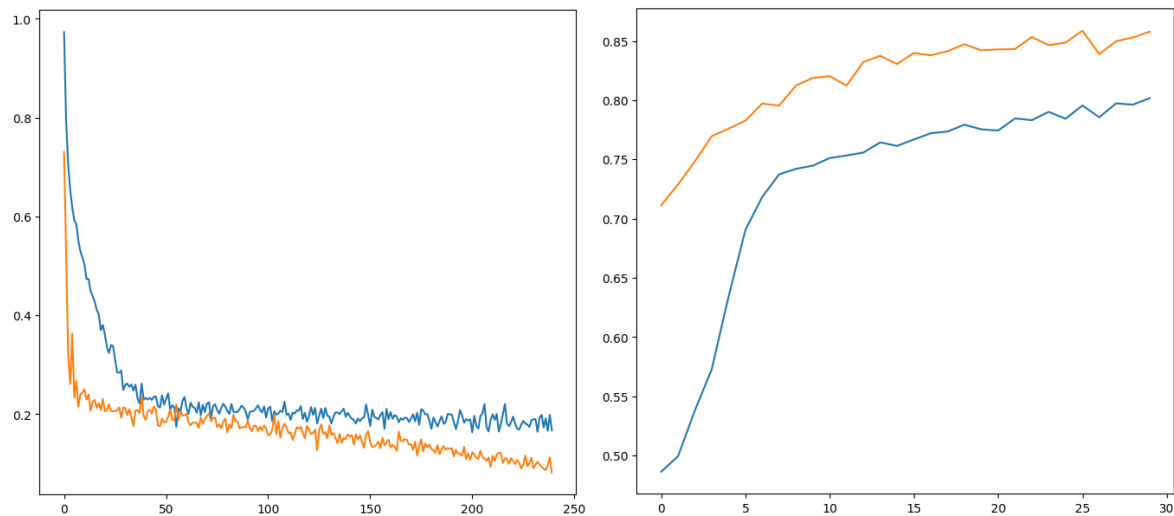
AudioModel conv\_layer:

```
self.conv_layer = nn.Sequential(  
    nn.Conv1d(n_mels, out_channels=hidden_size, kernel_size=3),  
    nn.MaxPool1d(3),  
    nn.ReLU(),  
    nn.Conv1d(hidden_size, out_channels=hidden_size, kernel_size=3),  
    nn.MaxPool1d(3),  
    nn.ReLU(),  
    nn.Conv1d(hidden_size, out_channels=hidden_size, kernel_size=3),  
    nn.MaxPool1d(3),  
    nn.ReLU(),  
)  
self.final_layer = nn.Linear(hidden_size, num_output)
```

my conv\_layer:

```
self.conv_layer = nn.Sequential(  
    nn.Conv1d(n_mels, out_channels=hidden_size, kernel_size=3, padding=1),  
    nn.ReLU(),  
    nn.MaxPool1d(3),  
    nn.Conv1d(hidden_size, out_channels=hidden_size*2, kernel_size=3, padding=1),  
    nn.ReLU(),  
    nn.MaxPool1d(3),  
    nn.Conv1d(hidden_size*2, out_channels=hidden_size*4, kernel_size=3, padding=1),  
    nn.ReLU(),  
    nn.MaxPool1d(3),  
    nn.Conv1d(hidden_size*4, out_channels=hidden_size*8, kernel_size=3, padding=1),  
    nn.ReLU(),  
    nn.MaxPool1d(3),  
    nn.Conv1d(hidden_size*8, out_channels=hidden_size*16, kernel_size=3, padding=1),  
    nn.ReLU(),  
    nn.MaxPool1d(3),  
)  
self.final_layer = nn.Linear(hidden_size*16, num_output)
```

## training result:



## How did you change the structure of CNN?

`nn.Conv1d()`, `nn.ReLU()`, `nn.MaxPool1d()`를 3번에서 5번 반복하도록 설계했다. 또한 `nn.Conv1d()` 함수에서 `padding`을 1로 설정하는 옵션을 추가했고, 각 `layer`를 통과할 때마다 `hidden_size`를 2배씩 증가시켰다.

## What is the difference?

`layer`를 더 많이 쌓아 모델의 특징 추출 능력이 강해지도록 하였고, 패딩을 통해 입력 데이터의 양쪽 끝에 데이터를 추가하여 컨볼루션 연산시 입력과 출력의 크기가 동일하게 유지되도록 하였다. 또한 깊은 층으로 갈수록 채널 수를 늘려 각 층에서 더 많은 특성을 학습할 수 있도록 하였다.

## 시도해본 방법들:

1) `kernel_size`를 5로 늘리고, `padding`을 2로 설정해 더 넓은 범위의 특성을 잡아낼 수 있도록 해보았다. 여러 번 테스트 해본 결과 `kernel_size`가 3이고, `padding`을 1로 설정할 경우와 비슷하지만 정확도가 좀 더 불안정하게 값이 큰 폭으로 움직이는 경향이 있어 최종적으로 `kernel_size`가 3, `padding`을 1로 설정하게 되었다.

2) `hidden_size`를 유지하는 방법과 깊은 층으로 갈수록 채널 수를 늘려 각 층에서 더 많은 특성을 학습할 수 있도록 `hidden_size`에 순차적으로 2, 4, 6, 8을 공급하는 방법을 비교해보았다. 그 결과 채널 수를 늘리는 방법이 정확도를 2-3점 정도 더 증가시켜 후자를 택했다.

3) 수업 시간에 배운 `nn.BatchNorm1d()` 함수를 이용해 각 컨볼루션 층 뒤에 배치 정규화를 추가하여 학습 과정을 안정화하고 빠르게 수렴하도록 해보았다. 그 결과 정확도와 `loss` 개선에 별 효과가 없는 것 같고 오히려 정확도가 낮아지는 경향을 보여 코드에 추가하지 않았다.



4) nn.Conv1d( ), nn.ReLU( ), nn.MaxPool1d( )를 5번 보다 더 많이 6번, 7번 실행해보았는데 5번이 최대의 정확도를 보이고 그 이상으로 실행할 경우 오히려 정확도가 낮아지는 경향을 보여 5번으로 설정하였다.

5) MaxPool의 커널 크기를 2로 줄여 특징맵이 좀 더 줄어들도록 해봤는데 정확도 향상에 큰 도움이 되지 않았고 MaxPool 커널 크기를 4로 늘리고 5번 반복할 경우 input 값이 너무 크게 줄어들어 오류가 발생해 최종적으로 3으로 설정했다. 또한 stride 값을 조정해 커널의 이동 간격을 조정해보았는데 정확도에 별 차이가 안 생겨서 default 값으로 커널 사이즈와 똑같이 설정되도록 두었다.

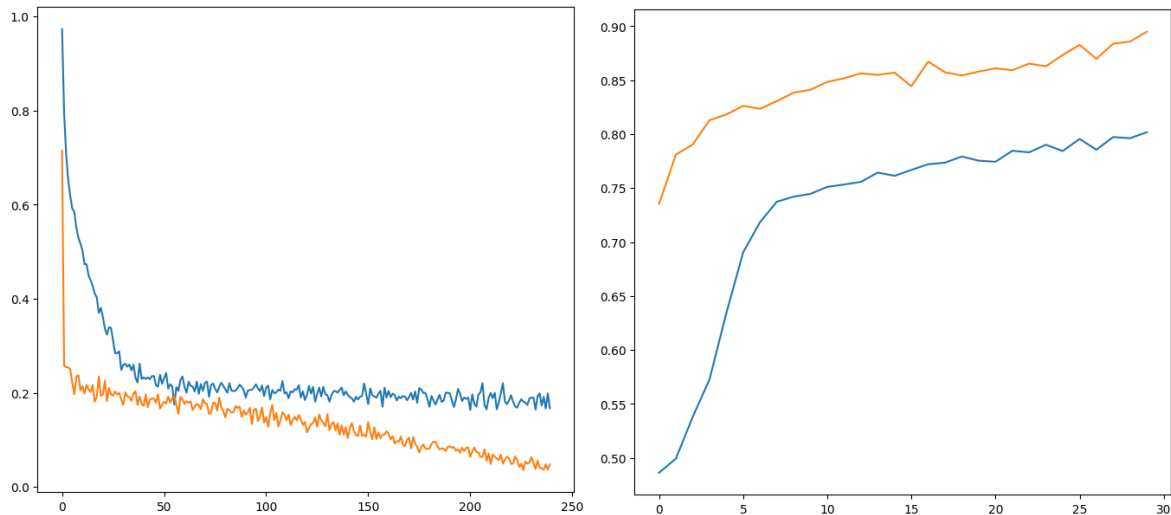
#### Problem 4. Try Various Settings and Report (20 pts)

my code:

```
your_model = YourModel(sr=16000, n_fft=1024, hop_length=512, n_mels=120, num_output=50, hidden_size=80)
```

> n\_mels를 120으로 늘리고, hidden\_size를 80으로 늘렸다.

result:



```
...  
Get the test result  
...  
test_acc = validate_model(your_model, test_loader, DEV)  
print(f"Calculated ROC_AUC value for Test Set is : {test_acc:.4f}")  
  
Calculated ROC_AUC value for Test Set is : 0.8934
```

#### Why you tried those changes

- 1) n\_fft는 다른 값(2048)으로 변경해보았지만 loss 감소와 정확도 향상에 효과가 없고 오히려 감소하는 결과를 보여 초기 값인 1024로 유지했다.
- 2) n\_mels의 경우 기존 48에서 점점 증가시킬 때마다 모델의 정확도가 향상되었고, 100~120보다 큰 값을 주면 더 이상 정확도가 향상되지 않고 오히려 떨어지는 양상을 보였다. 100~120 범위에서 여러 번 돌려본 결과 120이 좀더 안정적으로 높은 정확도를 보여 n\_mels의 값을 120으로 변경했다.
- 3) hidden\_size의 경우 기존 32에서 점점 증가시킬 때마다 모델의 정확도가 향상되었고, 70~80보다 큰 값을 주면 더 이상 정확도가 향상되지 않고 오히려 떨어지는 양상을 보였다. 70~80 범위에서 여러 번 돌려본 결과 80이 좀더 안정적으로 높은 정확도를 보여 hidden\_size의 값을 120으로 변경했다.

4) 변경한 `n_mels`와 `hidden_size`에 맞추어 여러 번 다른 방향으로 `conv_layer` 설계를 시도해보았지만 문제 3에서 완성한 `conv_layer`가 가장 높은 정확도를 보여 변경하지 않았다.

#### **What you have expected from the result with those changes**

- 1) `n_fft`: 값을 늘려 주파수 해상도를 향상시키면 모델이 주파수 관련 특징을 더 잘 학습할 수 있을 것이라 예상했다.
- 2) `n_mels`: 마찬가지로 주파수 대역에 대한 해상도가 향상되어 더 세밀한 주파수 정보를 추출할 수 있을 것이라 예상했다.
- 3) `hidden_size`: hidden layer의 뉴런 수를 늘리면 모델이 더 복잡한 패턴과 관계를 학습할 수 있을 것이라 예상했다.

#### **What you got from the changes**

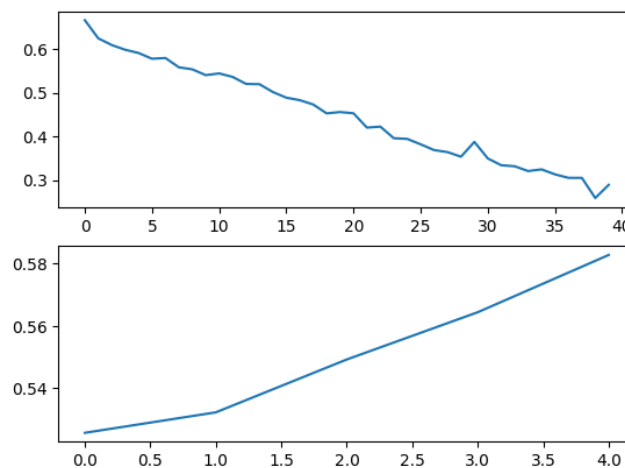
- 1) `n_fft`: `n_fft`를 2048로 늘려 보았지만 정확도가 오히려 감소했다. 아마도 STFT 트레이드 오프 때문에 그런 것이 아닐까 생각한다.
- 2) `n_mels`: 실제로 값을 증가시킬 경우 모델이 음성이나 음악의 세부적인 특성을 더 잘 포착해 성능이 향상된 것을 느꼈다. 그러나 일정 값 이상으로 커지면 정확도가 오히려 감소하는 경향을 보였다.
- 3) `hidden_size`: 실제로 값을 증가시킬 경우 모델의 정확도가 향상되는 것을 확인했다. 마찬가지로 일정 값 이상으로 커지면 정확도가 오히려 감소하는 경향을 보였다.

### Problem 5: Complete Binary Cross Entropy Function (4 pts)

```
def get_binary_cross_entropy(pred:torch.Tensor, target:torch.Tensor):  
    '''  
    pred (torch.Tensor): predicted value of a neural network model for a given input (assume  
    target (torch.Tensor): ground-truth label for a given input, given in multi-hot encoding  
  
    output (torch.Tensor): Mean Binary Cross Entropy Loss value of every sample  
    '''  
    # TODO: Complete this function  
  
    eps = 1e-6 # log 계산용 아주 작은 값  
    loss = -(target * torch.log(pred + eps) + (1-target) * torch.log(1-pred + eps)).mean()  
  
    return loss
```

**코드 설명:** get\_binary\_cross\_entropy는 두 텐서, 예측값 pred와 실제 타겟 target을 입력으로 받아 Binary Cross-Entropy Loss를 계산하는 함수이다. 먼저 eps 변수를 이용해 log 계산시 0이 들어가는 것을 방지하기 위해 매우 작은 값인 epsilon을 설정했다. 다음으로 BCE 식에 따라서 코드를 작성했다. 이때  $\text{target} * \text{torch.log}(\text{pred} + \text{eps})$ 는 target이 1인 경우의 손실을 계산하고,  $(1 - \text{target}) * \text{torch.log}(1 - \text{pred} + \text{eps})$ 는 target이 0인 경우의 손실을 계산한다. 이 두 값을 더하고 부호를 반전시킨 후 mean() 메서드를 사용하여 계산된 손실 값들의 평균을 구해 최종 손실을 얻을 수 있다.

**결과 출력:**



### Problem 6: Complete Precision-Recall Area Under Curve Function (20 pts)

```
def get_precision_and_recall(pred:torch.Tensor, target:torch.Tensor, threshold:float):

    # Write your code here

    assert pred.shape == target.shape
    thresholded_pred = pred > threshold

    precision = None
    recall = None

    # true라고 예측한 것 중에 실제로 true인 경우(TP) -> 둘다 true여야 하니까 곱하기
    num_true_positive = (thresholded_pred * target).sum()
    # true라고 예측한 것의 개수(TP + FP)
    num_positive_preds = thresholded_pred.sum()
    # 실제로 true인 것의 개수(P)
    num_true_target = target.sum()

    # precision과 recall 계산
    if num_positive_preds > 0:
        precision = num_true_positive / num_positive_preds
    else:
        precision = torch.tensor(0.)
    if num_true_target > 0:
        recall = num_true_positive / num_true_target
    else:
        recall = torch.tensor(0.)

    ...

    Be careful for not returning nan because of division by zero
    ...

    assert not (torch.isnan(precision) or torch.isnan(recall))
    return precision, recall
```

**코드 설명:** get\_precision\_and\_recall 함수는 모델의 예측값과 실제 레이블을 비교하여 정밀도 (precision)와 재현율(recall)을 계산하는 함수이다. 먼저 pred의 shape과 target의 shape이 같은지 확인한다. 다음으로 변수에 pred를 주어진 threshold 값으로 이진화하여 thresholded\_pred 변수에 저장한다.

num\_true\_positive는 true라고 예측한 것 중에 실제로 true인 경우(TP)의 개수를 계산한다. 이때 thresholded\_pred와 target이 둘다 true여야 하기 때문에 \*(곱하기 연산)를 이용했고, 이들을 모두 더해 num\_true\_positive를 구했다. num\_positive\_preds는 true라고 예측한 것의 개수(TP + FP)이다. 따라서 thresholded\_pred.sum()을 해주었다. num\_true\_target은 실제로 true인 것의 개수(P)이다. 따라서 target.sum()을 해주었다.

다음으로 위에서 계산한 값들을 가지고 precision과 recall 값을 계산하였다. precision은  $(TP / TP + FP)$ 이므로 num\_true\_positive / num\_positive\_preds를 통해 구하고, recall은  $(TP / P)$ 이므로 num\_true\_positive / num\_true\_target을 통해 구했다. 이때 if-else문을 통해 division by zero를 방지

하고 nan을 반환하지 않도록 하였다.

```
def get_precision_recall_auc(pred:torch.Tensor, target:torch.Tensor, num_grid=500):

    auc = 0
    prev_recall = 0 # recall 초기값

    for thresh in reversed(torch.linspace(0, 1, num_grid)):
        precision, recall = get_precision_and_recall(pred, target, threshold=thresh)
        # 면적 계산
        auc += precision * (recall - prev_recall)
        prev_recall = recall

    return auc
```

**코드 설명:** `get_precision_and_recall_auc` 함수는 주어진 예측값과 실제 레이블에 대해 다양한 임계값을 적용하여 PR-AUC를 계산하는 함수이다. 이때 PR-AUC는 precision-recall curve 아래 면적을 의미한다. `get_roc_auc` 함수를 참고해 코드를 완성하였다. `auc`는 누적 auc값을 저장할 변수이고, `prev_recall`은 x축을 이동할 때 이용할 `recall`의 초기값이다.

for문에서 `reversed(torch.linspace(0, 1, num_grid))`를 이용해 1에서 0까지 `num_grid`만큼의 점을 생성하여 각 점을 임계값으로 사용한다. 각 임계값에 대해 `get_precision_and_recall`을 호출하여 `precision`과 `recall`을 계산한다. 다음으로 `precision * recall - prev_recall` 값을 면적으로 계산하여 `auc`에 누적시킨다. `prev_recall`을 `recall`로 설정해 다음 구간의 면적을 계산할 수 있도록 한다. for문을 통해 반복하여 누적된 `auc` 값이 PR-AUC에 해당되며 이 값을 반환한다.

### 결과 출력:

```
'''  
Test the get_precision_recall_auc  
'''  
  
dummy_pred = torch.Tensor([0.0285,    0.0004,    0.0483,    0.0003,    0.0074,    0.0141,  
                             0.0007,    0.0735,    0.0534,    0.0153,    0.0024,    0.0053,  
                             0.0004,    0.1033,    0.1007,    0.4314,    0.1744,    0.0119,  
                             0.0189,    0.0075,    0.0001,    0.1354,    0.0014,    0.0004,  
                             0.4431,    0.0236,    0.0005,    0.1276,    0.0173,    0.0000,  
                             0.0010,    0.1237,    0.0616,    0.1674,    0.0000,    0.0053,  
                             0.0984,    0.0608,    0.1783,    0.0689,    0.0509,    0.0011,  
                             0.0749,    0.0001,    0.6105,    0.0136,    0.2644,    0.0204,  
                             0.0005,    0.0001])  
  
dummy_target = torch.Tensor([1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
                              0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0,  
                              0, 0])  
  
'''  
  
Printed result of code below has to be tensor(0.1753)  
'''  
  
get_precision_recall_auc(dummy_pred, dummy_target)
```

> 결과 값이 tensor(0.1753)으로 출력되어 코드가 정확하게 동작하는 것을 알 수 있다.

## Problem 7: Load audio and make prediction (15 pts)

Try several audio files and report the result by comparing your expectation and model's output

1)

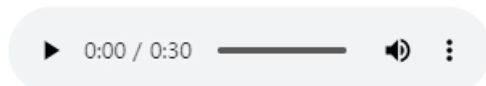
```
your_audio_path = 'Lobo Loco - Water Sun and Love.mp3' #TODO
# your_audio_path = 'Denys Kyshchuk - Drums Energy.mp3' #TODO
# your_audio_path = 'Beat Mekanik - Just Wanna Know.mp3' #TODO
# your_audio_path = 'Gregor Quendel - Chopin - Prelude No. 9, Opus 28.mp3' #TODO
# your_audio_path = 'John Lopker _ Popular USA Majority - Evangelical MAGA Blues.mp3' #TODO
# selected_model = model # Change it if you want to select model with different name
selected_model = your_model
```

```
THRESHOLD = 0.5
y = get_resampled_mono_audio_from_file(your_audio_path, selected_model.sr)

'''
You can slice your desired position
'''

sliced_y = slice_audio(y, selected_model.sr, 0, 30)

with torch.no_grad():
    pred = selected_model(sliced_y.unsqueeze(0).to(DEV)).to('cpu')
pred = pred[0]
ipd.display(ipd.Audio(sliced_y, rate=selected_model.sr))
print(f"Predicted tags are: {model.vocab[torch.where(pred>THRESHOLD)]}")
```



Predicted tags are: ['sitar' 'guitar']

**my expectation:** sitar, guitar, jazz

**model's output:** sitar, guitar

2)

```
# your_audio_path = 'Lobo Loco - Water Sun and Love.mp3' #TODO
your_audio_path = 'Denys Kyshchuk - Drums Energy.mp3' #TODO
# your_audio_path = 'Beat Mekanik - Just Wanna Know.mp3' #TODO
# your_audio_path = 'Gregor Quendel - Chopin - Prelude No. 9, Opus 28.mp3' #TODO
# your_audio_path = 'John Lopker _ Popular USA Majority - Evangelical MAGA Blues.mp3' #TODO
# selected_model = model # Change it if you want to select model with different name
selected_model = your_model
```

```
THRESHOLD = 0.5
y = get_resampled_mono_audio_from_file(your_audio_path, selected_model.sr)

...

You can slice your desired position
...

sliced_y = slice_audio(y, selected_model.sr, 0, 30)

with torch.no_grad():
    pred = selected_model(sliced_y.unsqueeze(0).to(DEV)).to('cpu')
pred = pred[0]
ipd.display(ipd.Audio(sliced_y, rate=selected_model.sr))
print(f"Predicted tags are: {model.vocab[torch.where(pred>THRESHOLD)]}")
```



Predicted tags are: ['india' 'beat']

**my expectation:** drum, beat, no singer

**model's output:** india, beat



3)

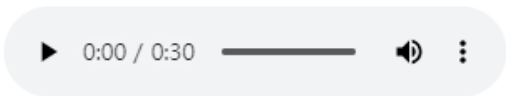
```
# your_audio_path = 'Lobo Loco - Water Sun and Love.mp3' #TODO
# your_audio_path = 'Denys Kyshchuk - Drums Energy.mp3' #TODO
your_audio_path = 'Beat Mekanik - Just Wanna Know.mp3' #TODO
# your_audio_path = 'Gregor Quendel - Chopin - Prelude No. 9, Opus 28.mp3' #TODO
# your_audio_path = 'John Lopker _ Popular USA Majority - Evangelical MAGA Blues.mp3' #TODO
# selected_model = model # Change it if you want to select model with different name
selected_model = your_model
```

```
THRESHOLD = 0.5
y = get_resampled_mono_audio_from_file(your_audio_path, selected_model.sr)

...
You can slice your desired position
...

sliced_y = slice_audio(y, selected_model.sr, 0, 30)

with torch.no_grad():
    pred = selected_model(sliced_y.unsqueeze(0).to(DEV)).to('cpu')
pred = pred[0]
ipd.display(ipd.Audio(sliced_y, rate=selected_model.sr))
print(f"Predicted tags are: {model.vocab[torch.where(pred>THRESHOLD)]}")
```



Predicted tags are: guitar

**my expectation:** singer, guitar, beat

**model's output:** guitar

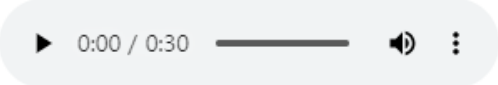
4)

```
# your_audio_path = 'Lobo Loco - Water Sun and Love.mp3' #TODO
# your_audio_path = 'Denys Kyshchuk - Drums Energy.mp3' #TODO
# your_audio_path = 'Beat Mekanik - Just Wanna Know.mp3' #TODO
your_audio_path = 'Gregor Quendel - Chopin - Prelude No. 9, Opus 28.mp3' #TODO
# your_audio_path = 'John Lopker _ Popular USA Majority - Evangelical MAGA Blues.mp3' #TODO
# selected_model = model # Change it if you want to select model with different name
selected_model = your_model
```

```
THRESHOLD = 0.5
y = get_resampled_mono_audio_from_file(your_audio_path, selected_model.sr)

...
You can slice your desired position
...
sliced_y = slice_audio(y, selected_model.sr, 0, 30)

with torch.no_grad():
    pred = selected_model(sliced_y.unsqueeze(0).to(DEV)).to('cpu')
    pred = pred[0]
    ipd.display(ipd.Audio(sliced_y, rate=selected_model.sr))
    print(f"Predicted tags are: {model.vocab[torch.where(pred>THRESHOLD)]}")
```



Predicted tags are: ['guitar' 'string' 'no singer']

**my expectation:** piano, slow, no singer

**model's output:** guitar, string, no singer

5)

```
# your_audio_path = 'Lobo Loco - Water Sun and Love.mp3' #TODO
# your_audio_path = 'Denys Kyshchuk - Drums Energy.mp3' #TODO
# your_audio_path = 'Beat Mekanik - Just Wanna Know.mp3' #TODO
# your_audio_path = 'Gregor Quendel - Chopin - Prelude No. 9, Opus 28.mp3' #TODO
your_audio_path = 'John Lopker _ Popular USA Majority - Evangelical MAGA Blues.mp3' #TODO
# selected_model = model # Change it if you want to select model with different name
selected_model = your_model
```

```
THRESHOLD = 0.5
y = get_resampled_mono_audio_from_file(your_audio_path, selected_model.sr)

...
You can slice your desired position
...

sliced_y = slice_audio(y, selected_model.sr, 0, 30)

with torch.no_grad():
    pred = selected_model(sliced_y.unsqueeze(0).to(DEV)).to('cpu')
    pred = pred[0]
    ipd.display(ipd.Audio(sliced_y, rate=selected_model.sr))
    print(f"Predicted tags are: {model.vocab[torch.where(pred>THRESHOLD)]}")
```



Predicted tags are: ['guitar' 'drum' 'rock']

**my expectation:** guitar, rock, singer

**model's output:** guitar, drum, rock

### Complete slice\_audio function (5 pts)

```
def slice_audio(audio_sample:torch.Tensor, sr:int, start_sec:float, end_sec:float):

    # 초 -> sr로 단위 변경
    slice_start = start_sec * sr
    slice_end = end_sec * sr

    # audio 자르기
    sliced_audio = audio_sample[slice_start:slice_end]

    return sliced_audio
```

**코드 설명:** slice\_start 변수에 start\_sec와 sr을 곱한 값을 넣고, slice\_end 변수에 end\_sec와 sr을 곱한 값을 넣어 초에서 sr로 단위를 변경해주었다. 그 후 slicing을 사용해 오디오를 잘라 sliced\_audio 변수에 저장한 후 sliced\_audio를 반환했다.