Deep learning for Music & Audio(AATG015-01)

Assignment 2

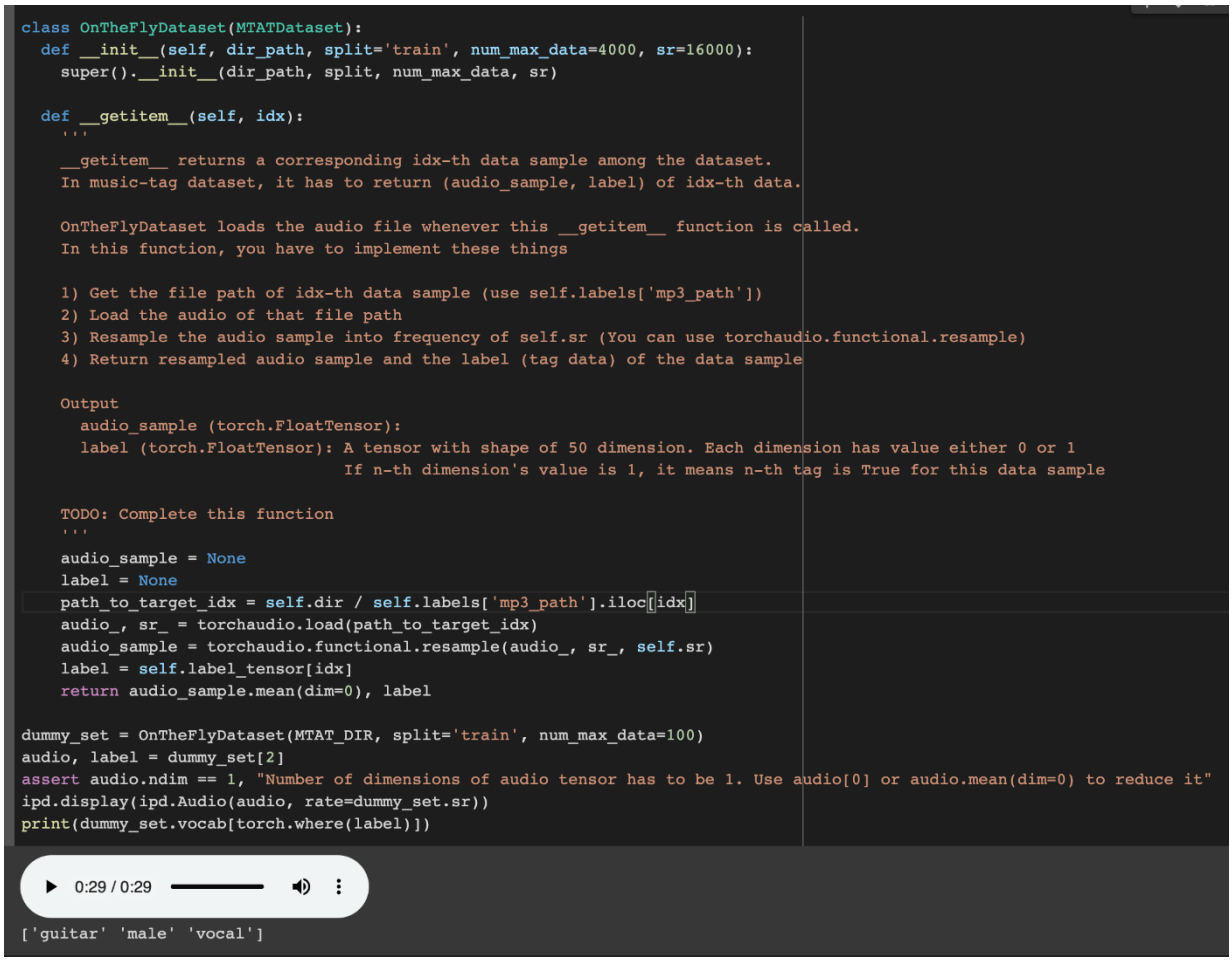
20200118 조창희

Problem 1. Complete Dataset Class

Data set을 다루는 방법에는 크게 세 가지 방법이 있다. 각 방법을 코드로 구현하였다.

(1) class OnTheFlyDataset(MTATDataset):

해당 함수를 호출할 때마다 오디오 파일을 로드하고, 리샘플링하는 방법이다.



```
class OnTheFlyDataset(MTATDataset):
  def __init__(self, dir_path, split='train', num_max_data=4000, sr=16000):
    super().__init__(dir_path, split, num_max_data, sr)

  def __getitem__(self, idx):
    '''
    __getitem__ returns a corresponding idx-th data sample among the dataset.
    In music-tag dataset, it has to return (audio_sample, label) of idx-th data.

    OnTheFlyDataset loads the audio file whenever this __getitem__ function is called.
    In this function, you have to implement these things

    1) Get the file path of idx-th data sample (use self.labels['mp3_path'])
    2) Load the audio of that file path
    3) Resample the audio sample into frequency of self.sr (You can use torchaudio.functional.resample)
    4) Return resampled audio sample and the label (tag data) of the data sample

    Output
      audio_sample (torch.FloatTensor):
      label (torch.FloatTensor): A tensor with shape of 50 dimension. Each dimension has value either 0 or 1
                                 If n-th dimension's value is 1, it means n-th tag is True for this data sample

    TODO: Complete this function
    '''
    audio_sample = None
    label = None
    path_to_target_idx = self.dir / self.labels['mp3_path'].iloc[idx]
    audio_, sr_ = torchaudio.load(path_to_target_idx)
    audio_sample = torchaudio.functional.resample(audio_, sr_, self.sr)
    label = self.label_tensor[idx]
    return audio_sample.mean(dim=0), label

dummy_set = OnTheFlyDataset(MTAT_DIR, split='train', num_max_data=100)
audio, label = dummy_set[2]
assert audio.ndim == 1, "Number of dimensions of audio tensor has to be 1. Use audio[0] or audio.mean(dim=0) to reduce it"
ipd.display(ipd.Audio(audio, rate=dummy_set.sr))
print(dummy_set.vocab[torch.where(label)])
```

getitem() 인자로는 self와 호출하는 인덱스가 들어온다. self.dir은 기존의 정의한 것처럼 Path(dir_path),즉 MTAT_DIR을 의미한다. self.labels를 통해 해당 인자의 mp3 경로를 불러온다. torchaudio.load를 이용하여 해당 경로의 파일을 로드한다. 그러한 다음 self.sr인 16000으로 오디오를 리샘플링한다. 해당 인덱스의 label을 label_tensor을 이용하여 구해주고 리샘플링한 오디오 샘플과 label값을 반환한다.

Audio.ndim을 1로 맞추기 위해 audio_sample.mean(dim=0)으로 바꿔주었다.

(2) class PreProcessDataset(MTATDataset):

decoding과 resample을 처리한 전처리파일을 pt로 저장해두고, pt파일을 불러오는 방법이다. 함수를 호출할 때마다 decoding과 resample을 하지 않아도 되어서 (1)에 비해 시간이 덜 소요된다는 장점이 있다.

```python
class PreProcessDataset(MTATDataset):
  def __init__(self, dir_path, split='train', num_max_data=8000, sr=16000):
    super().__init__(dir_path, split, num_max_data, sr)

    self.pre_process_and_save_data()

  def pre_process_and_save_data(self):
    '''
    self.pre_process_and_save_data loads every audio sample in the dataset, resample it, and save it into pt file.
    In this function, you have to implement these things

    1) For every data sample in the dataset, check whether pre-processed data already exists
      - You can get data sample path by self.labels['mp3_path'].values
      - path of pre-processed data can be in the same directory, but with different suffix.
      - You can make it with Path(mp3_path).with_suffix('.pt')
    2) If it doesn't exist, do follow things
      a) Load audio file
      b) Resample the audio file with samplerate of self.sr
      c) Get label of this audio file
      d) Save {'audio': audio_tensor, 'label':label_tensor} with torch.save

    Output
      None

    TODO: Complete this function
    '''
    sample_path = self.dir / self.labels['mp3_path'].values
    for i, s in enumerate(sample_path):
      dir_path = Path(s).with_suffix('.pt')
      if Path(dir_path).is_file():
        continue
      audio_, sr_ = torchaudio.load(s)
      resample_audio = torchaudio.functional.resample(audio_, sr_, self.sr)
      label = self.label_tensor[i]
      processed_data = {'audio': resample_audio, 'label': label}
      torch.save(processed_data, dir_path)



  def __getitem__(self, idx):
    '''
    __getitem__ returns a corresponding idx-th data sample among the dataset.
    In music-tag dataset, it has to return (audio_sample, label) of idx-th data.

    PreProcessDataset loads the pre-processed pt file whenever this __getitem__ function is called.
    In this function, you have to implement these things

    1) Get the pt file path of idx-th data sample (use self.labels)
    2) Load the pre-procssed data of that file path (use torch.load)
    3) Return the audio sample and the label (tag data) of the data sample

    TODO: Complete this function
    '''
    path_to_target_idx = Path(self.dir / self.labels['mp3_path'].iloc[idx]).with_suffix('.pt')
    audio = torch.load(path_to_target_idx)
    return audio['audio'].mean(dim = 0), audio['label']

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] or audio.mean(dim=0) to reduce it"
ipd.display(ipd.Audio(audio, rate=dummy_set.sr))
print(dummy_set.vocab[torch.where(label)])
```

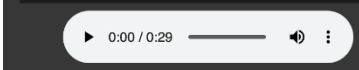
pre_process_and_sasve_data 에서는 with_suffix()를 이용하여 .mp3 경로를 .pt경로로 바꿔준다. For 반복문을 돌면서 해당 경로의 파일이 존재한다면 넘어가고, 존재하지 않는다면 로드하고 리샘플함으로써 .pt형식의 전처리 파일을 만든다.

get_item에서는 torch.load를 통해 딕셔너리 형식으로 불러와서 변수 audio에 저장하고, audio sample과 label을 반환한다. 1번과 달리 getitem에서 리샘플링을 하지 않는다.

(3) class OnMemoryDataset(MTATDataset):

마지막 방법은 모두 메모리에 저장하고 getitem에서 불러오기만 한다. 물리적인 저장 용량을 차지한다는 단점이 있지만, 세 방법 중 가장 속도가 빠르다고 한다.

```
[11] class OnMemoryDataset(MTATDataset):
       def __init__(self, dir_path, split='train', num_max_data=4000, sr=16000):
         super().__init__(dir_path, split, num_max_data, sr)

         self.loaded_audios = self.load_audio()

       def load_audio(self):
         '''
         In this function, you have to load all the audio file in the dataset, and resample them,
         and store the data on the memory as a python variable

         For each data in the dataset,
           a) Load Audio
           b) Resample it to self.sr
           c) Append it to total_audio_datas

         Output:
           total_audio_datas (list): A list of torch.FloatTensor. i-th item of the list corresponds to the audio sample of i-th data
                                     Each item is an audio sample in torch.FloatTensor with sampling rate of self.sr
         '''
         total_audio_datas = []
         ### Write your code from here
         sample_path = self.dir / self.labels['mp3_path'].values
         for s in sample_path:
           audio_, sr_ = torchaudio.load(s)
           resample_audio = torchaudio.functional.resample(audio_, sr_, self.sr)
           total_audio_datas.append(resample_audio)
         return total_audio_datas

       def __getitem__(self, idx):
         '''
         __getitem__ returns a corresponding idx-th data sample among the dataset.
         In music-tag dataset, it has to return (audio_sample, label) of idx-th data.

         OnMemoryDataset returns the pre-loaded audio data that is aved on self.loaded_audios whenever this __getitem__ function is called.
         In this function, you have to implement these things

         1) Load the pre-procssed audio data from self.loaded_audios
         2) Return the audio sample and the label (tag data) of the data sample

         TODO: Complete this function
         '''
         audio = self.loaded_audios
         label = self.label_tensor[idx]
         return audio[idx].mean(dim=0), label

     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] or audio.mean(dim=0) to reduce it"
     ipd.display(ipd.Audio(audio, rate=dummy_set.sr))
     print(dummy_set.vocab[torch.where(label)])
```




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

Load_audio에서 반복문을 돌며 모든 오디오 샘플을 로드하고 리샘플링한 다음, total_audio_datas 리스트에 추가해준다. 그리고 getitem에서는 인덱스와 대응되는 sample값을 찾아 반환한다.

Define Dataset

- You can select one of your implementations

```
[12] your_dataset_class = PreProcessDataset
     '''
     Based on your memory size or storage size, you can change the num_max_data
     '''
     trainset = your_dataset_class(MTAT_DIR, split='train', num_max_data=5000)
     validset = your_dataset_class(MTAT_DIR, split='valid', num_max_data=1000)
     testset = your_dataset_class(MTAT_DIR, split='test', num_max_data=2000)
```

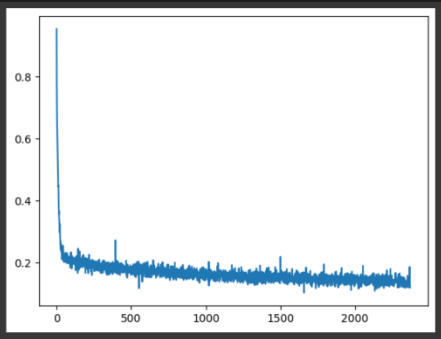
나는 3가지 방법 중 dataset class로 처음에는 OnMemoryDataset 을 사용하고자 했으나, 정의하고 로드하는 중에 colab에서 사용가능한 램을 다 사용하였다며 세션이 다운되었다. 따라서 그 다음 시도로 PreProcessDataset을 선택하였다. 그런데 이후 train을 돌려보니 preprocess 방법도 꽤나 시간이 소요된다는 사실을 알 수 있었다.

```python
class AudioModel(nn.Module):
  def __init__(self, sr, n_fft, hop_length, n_mels, hidden_size, num_output):
    super().__init__()
    self.sr = sr
    self.spec_converter = SpecModel(sr, n_fft, hop_length, n_mels)
    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)
```

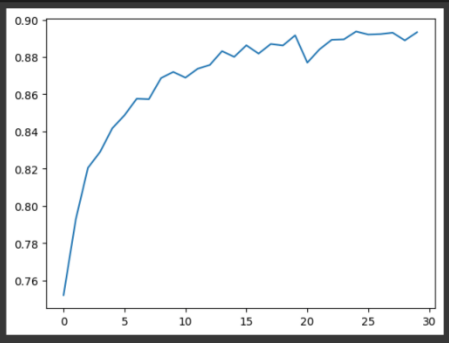
```python
[16] Train the default model
'''

model = AudioModel(sr=16000, n_fft=1024, hop_length=512, n_mels=48, num_output=50, hidden_size=32)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
model = model.to(DEV)
loss_func = torch.nn.BCELoss()
train_record = train_model(model, train_loader, valid_loader, optimizer, num_epochs=30, loss_func=loss_func, device=DEV)
```

```
100%|■■■■■■■■■■| 30/30 [54:11<00:00, 108.39s/it]
```

```python
[17] plt.plot(train_record['loss'])
save_fig_with_date('default_train_loss')
```




```python
[18] plt.plot(train_record['valid_acc'])
save_fig_with_date('default_train_valid_acc')
```



Default model로 한 train 결과이다. loss는 초반에 급진적으로 줄고 이후로 진동하고 있고, valid set에 대한 acc도 우상향하고 있는 모습이다.

Problem 2. Practice with nn.Sequential()

nn.Sequential()은 nn모듈을 인수로 받아 순서대로 전달해주는 역할을 한다. 기존 방법보다 줄 수를 줄여 훨씬 더 간편하게 신경망을 구축할 수 있다.

```
28]
  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__()
      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),
                                   )

    '''
    TODO: Complete this nn.Sequential so that it computes exactly same thing with StackManualLayer
    '''

    def forward(self, x):
      out = self.layers(x)
      return out

  # Do not change the code below
  torch.manual_seed(0)
  manual_layer = StackManualLayer()
  torch.manual_seed(0)
  sequential_layer = SequentialLayer()

  '''
  The printed result has to be same
  '''

  test_dummy = torch.arange(128).view(1,16,8).float()
  manual_out = manual_layer(test_dummy)
  print(f"Output with Manual Stack Layer: {manual_out}")
  sequential_out = sequential_layer(test_dummy)
  print(f"Output with Sequential Layer: {sequential_out}")

  assert torch.allclose(manual_out, sequential_out), "The output of manual layer and sequential layer is different"
```

```
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>)
```

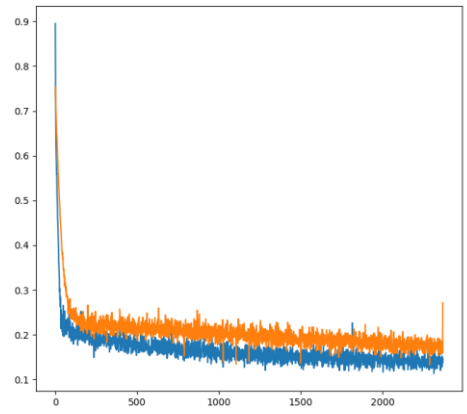
결과를 보면 ManualLayer과 같은 결과를 보임을 알 수 있다.

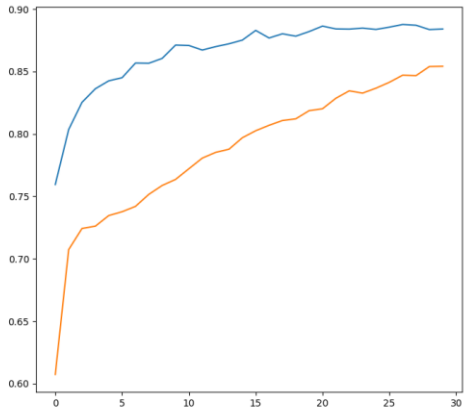
Problem 3. Make Your Own Conv Layers

Conv2d는 주로 컴퓨터 비전 영역에서 많이 사용하므로, 음성 인식에는 적절하지 않은 것으로 판단되어 conv1d를 그대로 사용하였다.

```
self.conv_layer = nn.Sequential( nn.Conv1d(n_mels, out_channels=hidden_size, kernel_size=3),
                                 nn.BatchNorm1d(hidden_size), #정규화
                                 nn.MaxPool1d(3),
                                 nn.ReLU(),
                                 nn.Conv1d(hidden_size, out_channels=hidden_size, kernel_size=3),
                                 nn.BatchNorm1d(hidden_size),
                                 nn.MaxPool1d(3),
                                 nn.ReLU(),
                                 nn.Conv1d(hidden_size, out_channels=hidden_size, kernel_size=3),
                                 nn.BatchNorm1d(hidden_size),
                                 nn.MaxPool1d(3),
                                 nn.Sigmoid()
                                )
```

바꾼 방식과 그 이유 : Batch normalization을 추가해주었다. 값의 평균을 0으로 만들어 변형된 결과(오버피팅)가 나오지 않고, valid set에서도 general하게 정확도를 높이고자 하였다. 마지막에는 ReLU대신 Sigmoid 함수를 넣어봤다. Sigmoid 함수는 layer 수가 많아질 때 중간에 값이 소실될 수 있는 문제가 있는데, 마지막에 결과를 낼 때만 넣으면 큰 문제가 없을 것 같다고 생각했다.





아쉽게도 default 모델에 비해 낮은 성능의 값이 나왔다. Loss 값도 나의 conv_layer을 적용한 경우가 더 높았고, epoch 수를 늘림에 따라 우상향 하였지만, 같은 epoch에 대해 valid set에 대한 정확도도 더 낮게 나오는 모습이 나왔다. 4번 문제에서 layer을 수정할 예정이다.

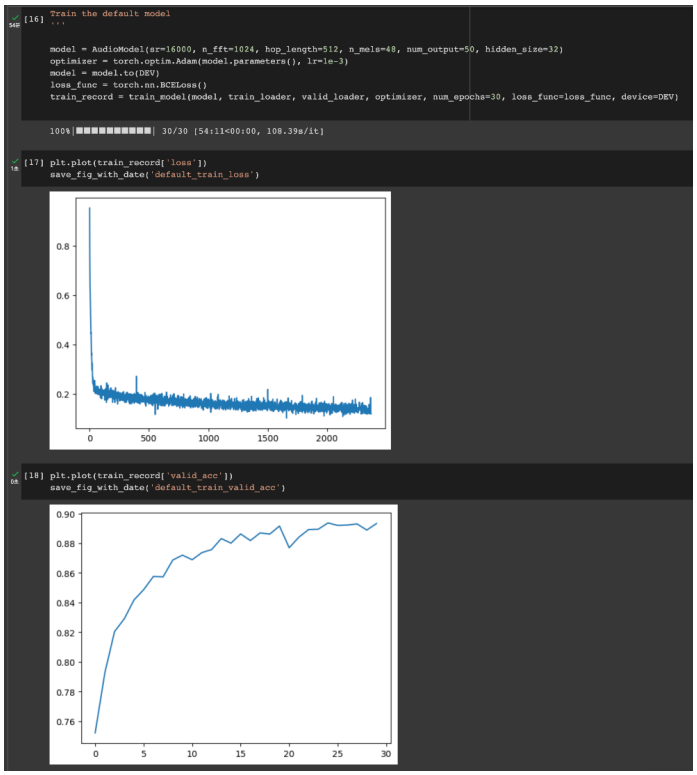
Probelm 4. Try Various Settings and Report

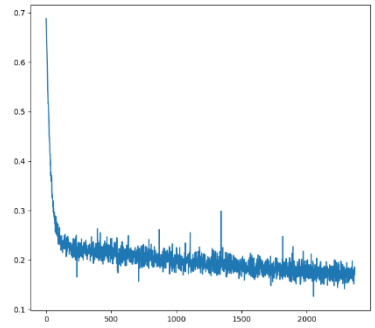
Probelm 3에 이어서, default보다 성능을 높이기 위해 총 세번의 시도를 하였다. Conv layer을 두번 수정해보았고, hidden layer의 값을 늘려보았다.

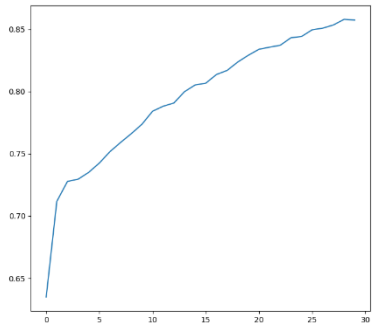
1. pooling과 활성화함수의 순서를 바꿔보았다.

기존 ConvLayer에서, 3개의 레이어 각각 MaxPool1d와 ReLU의 순서도 바꿔보았다. 활성화 함수를 적용하고 pooling하면 더 좋은 성능을 내지 않을까 하는 기대에서였다.

(중간에 컴퓨터가 colab이 다운되어 default train record와 나의 conv_layer을 적용한 모델을 따로 출력하였다.)



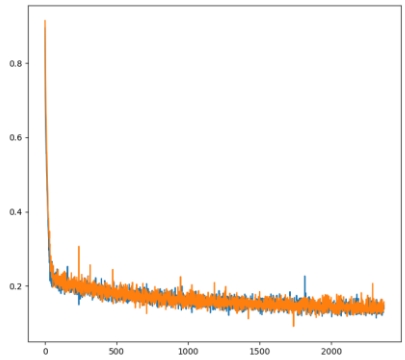


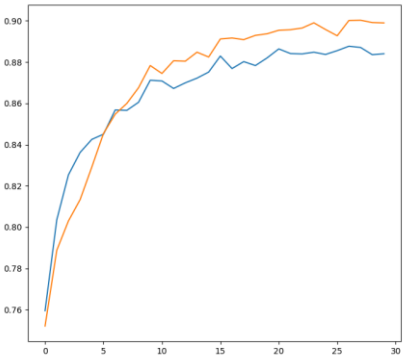


아쉽게도 3번에서 정의한 layer와 비슷한 성능을 보였다. 왼쪽은 default값, 오른쪽은 나의 신경망으로 합성한 결과이다. Y축을 보면 default model이 내가 만든 레이어에 비해 더 정확도가 높은 것을 알 수 있다. 이를 통해 해당 상황에서는, pooling과 활성화함수의 순서가 큰 의미가 없다는 것을 알 수 있었다.

2. 활성화함수 변경

마지막 활성화함수를 sigmoid에서 ReLU로 바꿔주었다.

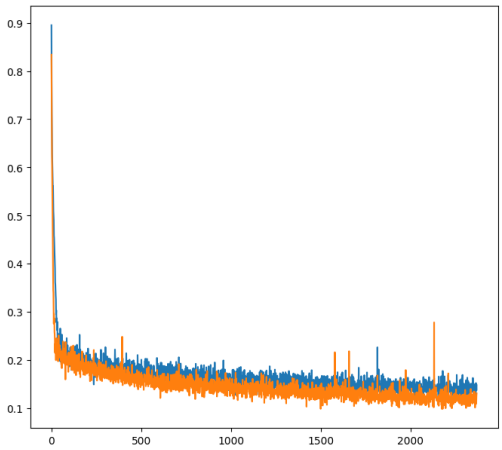


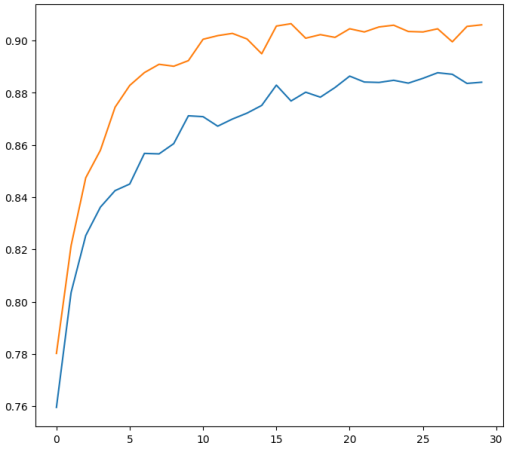


드디어 의도한 결과를 얻을 수 있었다. 비용함수에서 default 모델과 loss값 자체는 유사하나, Batch normalization을 통해 기존 default model보다 안정된 형태의 loss를 보여주고, valid set에 대해 약 10 이상의 epoch에서 더 높은 정확도를 볼 수 있다. 여기서는 두 가지를 알 수 있었는데, 첫 번째는 0과 1 두가지만 있는 classification 문제가 아닌 이상 sigmoid보다는 ReLU가 확실히 성능이 좋다는 것이다. 두 번째로는 기존 모델에 비해 성능을 향상시키고 싶을 때는 다른 조건을 고정한 채 한 조건만 바꾸는 것이 비교하기 좋다는 것이다. 예를 들면 이전 모델의 경우 default 모델에서 batch norminalization을 진행하고, 마지막 함수를 시그모이드로 바꾸었는데, 그러면 성능이 잘 안나오는 이유가 전자때문인지 후자때문인지 알기가 어렵다.

3. 하이퍼 파라미터의변화 - 4-2번 모델에서 하이퍼 파라미터의 결과를 바꿔보았다. Hidden size를 32에서 64로 늘려주었다. hidden size를 늘리게 되면 시간은 조금 더 걸리더라도 더 높은 성능을 보일 것이라는 기대가 있었다. 훈련 시간은 1시간 2분으로, 4-2번 모델에 비해 2분정도 더 소요되었다.

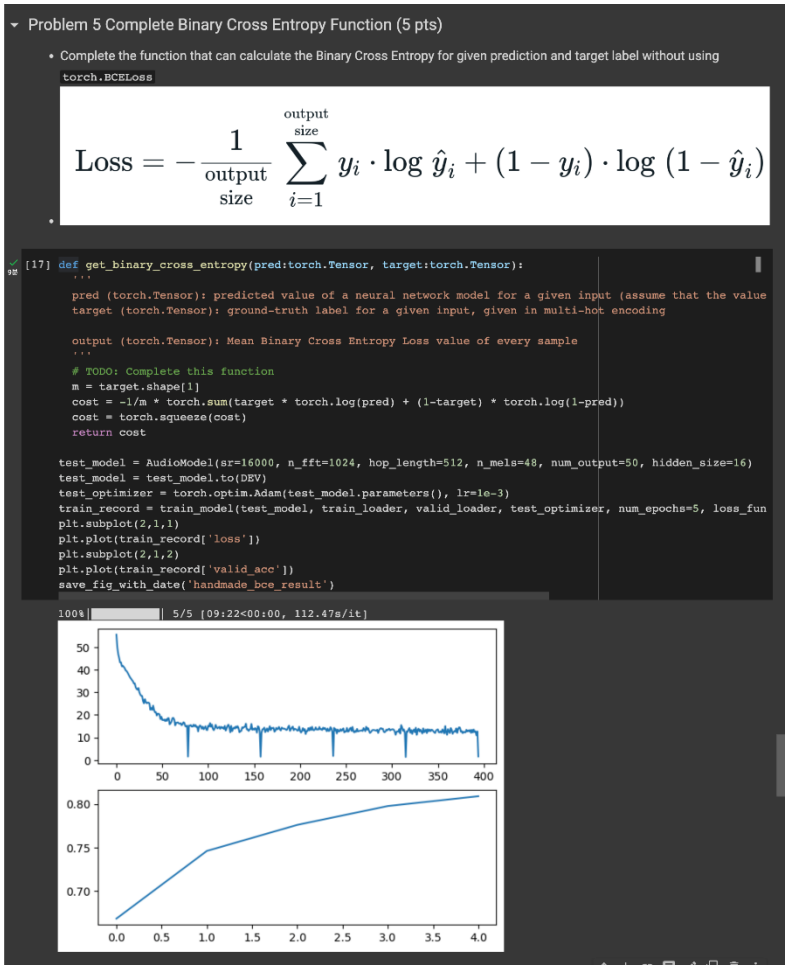
```
your_model = YourModel(sr=16000, n_fft=1024, hop_length=512, n_mels=48, num_output=50, hidden_size=64) #hidden 32 -> 64
optimizer = torch.optim.Adam(your_model.parameters(), lr=1e-3)
your_model = your_model.to(DEV)
your_train_record = train_model(your_model, train_loader, valid_loader, optimizer, num_epochs=30, loss_func=torch.nn.BCELoss(), device=DEV)

## Save the figure with comparison of default setting
plt.figure(figsize=(8,16))
plt.subplot(2,1,1)
plt.plot(train_record['loss'])
plt.plot(your_train_record['loss'])
plt.subplot(2,1,2)
plt.plot(train_record['valid_acc'])
plt.plot(your_train_record['valid_acc'])
save_fig_with_date('comparison_with_default')
```





이전 모델은 loss값 자체는 default model 과 유사했는데, 해당 모델은 중간에 튀는 값이 있지만 loss값이 유의미하게 더 적게 나왔다. Valid accuracy는 이전 모델과 거의 유사한데 차이가 있다면, 이전 모델은 epoch 10 이하에서는 default값이 더 높은 정확도를 보였지만, 해당 모델은 처음부터 default 모델보다 더 높은 정확도가 나왔다. 또한, 이전 모델은 꾸준히 우상향하는 모습을 보였다면, 해당은 15-20 사이의 epoch에서 가장 높은 정확도를 보였으며, 그 이후에 오히려 정확도가 내려가는 모습이 보인다. 따라서 해당 경우에는 hidden size를 높이면 더 적은 epoch 에서도 상당히 높은 정확도를 보인다는 점을 알 수 있었다.

Problem 5 Complete Binary Cross Entropy Function



내장되어있는 BCE를 사용하지 않고 직접 비용 함수를 구현하는 문제이다. Torch 모듈에 있는 sum과 log를 이용하여 구현하였다. Output size는 shape의 1번째 인덱스와 같다.

Problem 6. Complete Precision-Recall Area Under Curve Function

```python
[91] def get_precision_and_recall(pred:torch.Tensor, target:torch.Tensor, threshold:float):
        '''
        This function calculates precision and recall of given (prediction, target, threshold)

        pred (torch.Tensor): predicted value of a neural network model for a given input
        target (torch.Tensor): ground-truth label for a given input, given in multi-hot encoding

        output
          precision (torch.Tensor): (Number of true positive)/(Number of total positive predictions)
          recall (torch.Tensor): (Number of true positive)/(Number of total positive ground-truth)

        IMPORTANT:
          To prevent division by zero, make the denominator greater than zero.

        TODO: Complete this function
        '''

        # Write your code here

        precision = None
        recall = None

        positive_pred = pred > threshold
        true_positive = positive_pred * target
        num_tp = true_positive.sum() #참을 참이라고 예측한 개수
        false_positive = positive_pred * (1-target)
        num_fp = false_positive.sum() #거짓을 참이라고 예측한 개수
        false_negative =  (positive_pred == False) * target
        num_fn = false_negative.sum() #참을 거짓이라고 예측한 개수
        precision = num_tp / (num_tp + num_fp + 0.00000001) #0으로 나누는 것 방지
        recall = num_tp / (num_tp + num_fn + 0.00000001)
        '''
        Be careful for not returning nan because of division by zero
        '''
        assert not (torch.isnan(precision) or torch.isnan(recall))
        return precision, recall

      def get_precision_recall_auc(pred:torch.Tensor, target:torch.Tensor, num_grid=500):
        '''
        This function returns PR_AUC value for a given prediction and target.
        Assume pred.shape == target.shape

        pred (torch.Tensor): predicted value of a neural network model for a given input
        target (torch.Tensor): ground-truth label for a given input, given in multi-hot encoding

        output (torch.Tensor): Area Under Curve value for Precision-Recall Curve

        TODO: Complete this function using get_precision_and_recall
        '''

        auc = 0
        prev = 0;
        gra = torch.zeros(num_grid, 2)
        for i, thresh in enumerate(reversed(torch.linspace(0, 1, num_grid))):
          prec, rec = get_precision_and_recall(pred, target, thresh)
          #print(prec, rec)
          auc += prec * (rec - prev)
          #print(prec, rec - prev)
          prev = rec
          #gra[i,:] = torch.tensor([prec, rec])

        #plt.plot(gra[:,1], gra[:,0])
        return auc
```

```
[92] '''
     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, 1, 0, 0, 0, 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)
     '''

     #test
     #threshold = 0.99
     #prec, recall = get_precision_and_recall(dummy_pred, dummy_target, threshold)
     #print(prec, recall)
     #threshold = 0.1
     #prec, recall = get_precision_and_recall(dummy_pred, dummy_target, threshold)
     #print(prec, recall)

     get_precision_recall_auc(dummy_pred, dummy_target)

     tensor(0.1753)
```

get_precision_and_recall() 에서는 precision과 recall 값을 구한다. precision은 True이라고 예측한 것들 중 실제로 True인 비율을 의미하고(TP / (TP + FP)), Recall은 실제 True인 것 중에서 True라고 예측한 것들의 비율이다. (TP / (TP + FN)) PR AUC Function은 recall을 x축, precision을 y축이고 곡선 아래 영역이 AUC 값일 것이다.

코드를 짜면서 변수의 이름을 헷갈려 중간에 많이 헤맸던 문제이다. 0으로 나뉘는 현상을 방지하기 위해 아주 작은 값을 분모에 더해주었다. get_precision_and_recall_auc()에서는 앞선 함수를 이용하여 auc의 값을 구할 것이다. 1부터 threshold의 값을 조금씩 줄여가며 위 함수를 호출하고, y값 * (x값 길이)를 통해 auc의 값을 누적하여 더하고 auc를 반환한다. 최종적으로 dummy set기준 0.1753 값이 나왔다.

Problem 7: Load audio and make prediction

먼저, slice_audio를 구현하였다.

```
[29] your_audio_path = 'yourmeaning.mp3' #TODO
     selected_model = your_model # Change it if you want to select model with different name

[30] def get_resampled_mono_audio_from_file(audio_file_path, target_sr):
       y, sr = torchaudio.load(audio_file_path)
       if sr!= selected_model.sr:
         y = torchaudio.functional.resample(y, orig_freq=sr, new_freq=selected_model.sr)
       if y.shape[0] > 1:
         y = torch.sum(y, dim=0) / y.shape[0]

       return y

     def slice_audio(audio_sample, sr, start_sec, end_sec):
       '''
       This function takes an audio sample, sampling rate, and start/end position of slice
       and returns the sliced audio sample.

       audio_sample (torch.Tensor): A sequence of audio samples in shape of (N,), where N is number of audio samples
       sr (int): Sampling rate of audio_sample
       start_sec (float): desired slice start position in seconds
       end_sec (float): desired slice end position in seconds

       output (torch.Tensor): A sequence of audio samples in shape of (int(sr*(end_sec-start_sec)), )


       TODO: Complete this function
       '''
       start_idx = int(start_sec * sr)
       end_idx = int(end_sec * sr)
       sliced_audio = audio_sample[start_idx:end_idx]
       return sliced_audio
```

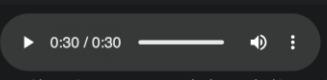
시작 초와 끝나는 초를 인자로 받고, samplig rate와 곱하여 텐서와 대응되는 인덱스를 만든다. 인덱스는 정수만 가능하므로 int로 형변환해준다. 해당 인덱스만큼 자르고 반환한다.

총 두가지 음악 파일을 이용하였고, 모델은 4-2(4번 문제에서 두번째로 구현한 모델)와 4-3을 이용하였다.

1. 아이유의 너의 의미 mp3 file을 사용하였다. 현재 태그중에 singer, soft, foreign 태그 정도가 나올 것이라고 예상하였다. 또한 초반 10초는 전주이기 때문에 10초부터 40초까지를 슬라이싱했다.

먼저, model은 4-2의 모델을 이용하였고 THRESHOLD를 0.2로 설정해보았다.

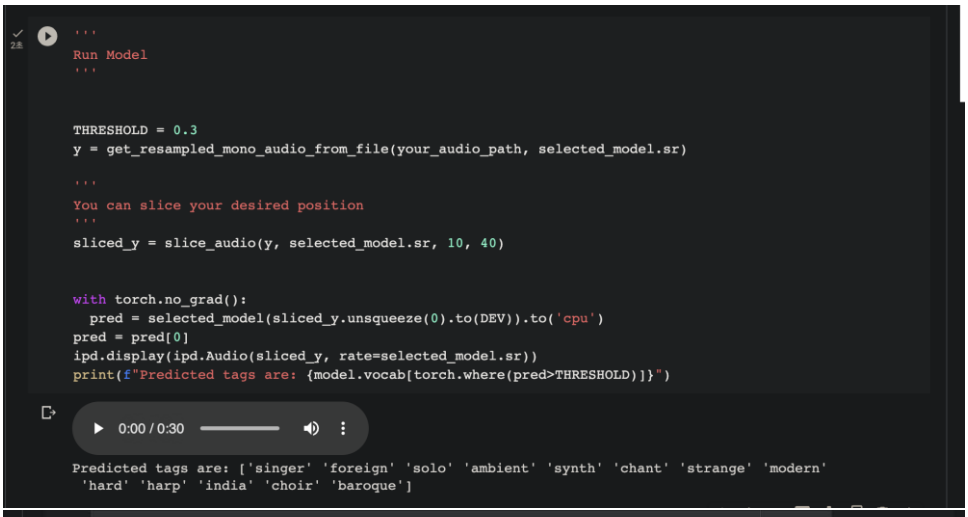
```
] '''
  Run Model
  '''


  THRESHOLD = 0.2
  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, 10, 40)


  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)]}")
```

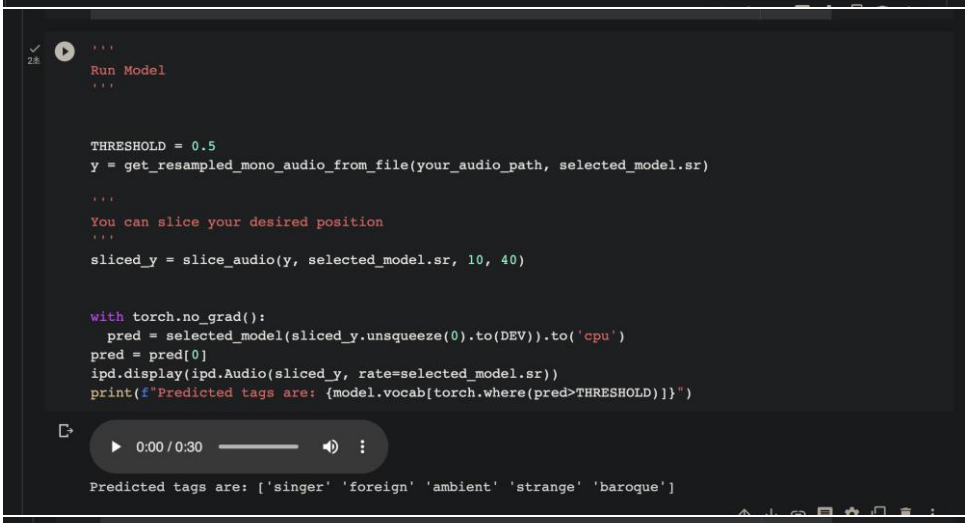


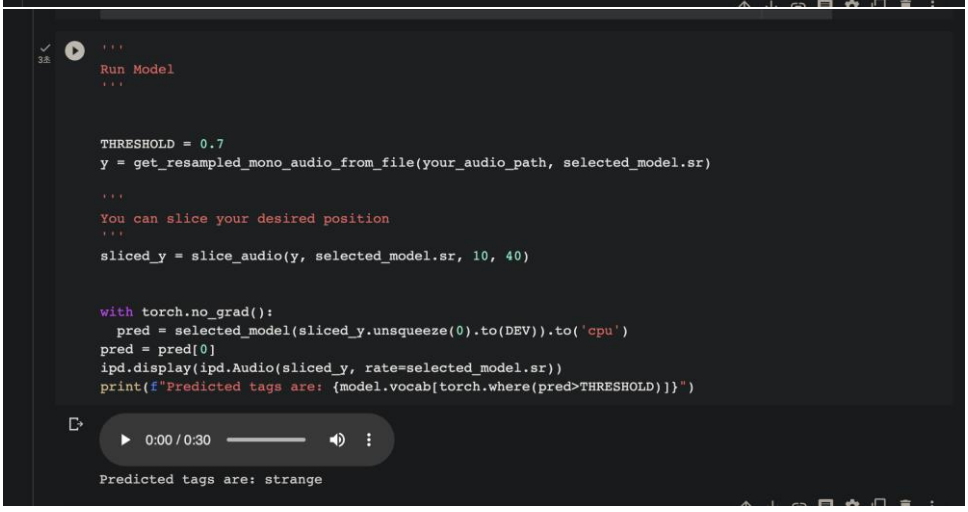

```
Predicted tags are: ['singer' 'harpsichord' 'sitar' 'heavy' 'foreign' 'classical' 'guitar'
 'quiet' 'solo' 'ambient' 'synth' 'drum' 'string' 'country' 'trance'
 'chant' 'strange' 'modern' 'hard' 'harp' 'pop' 'no singer' 'india' 'rock'
 'techno' 'beat' 'choir' 'baroque']
```

tag가 많이 나왔다. Singer, guiter, quiet, solo, drum, foreign 등 예상한 태그들도 있었고, rock, no singer등 적절하지 않은 태그도 나왔다.

TRESHOLD를 0.3, 0.5, 0.7로 늘려 보았다.

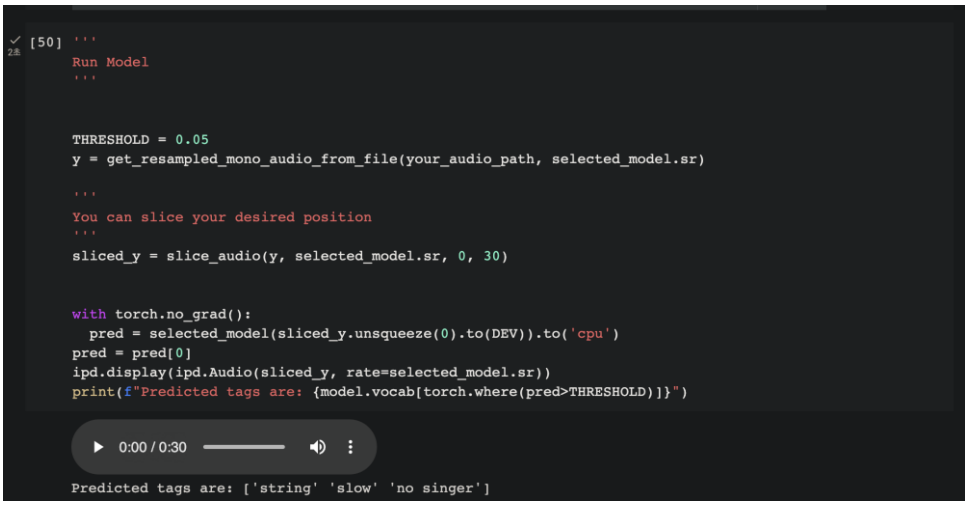






점점 태그가 줄어드는 걸 볼 수 있다. 0.5 정도가 적절한 것 같다. 영어 데이터로 학습하였는데 한국어 노래 이기 때문에 foreign, strange가 나오는 게 아닐까 추측해본다.

이제 4-3 모델을 사용하였다. 4-3 모델을 사용할 경우 threshold를 더 낮추어야 태그의 수가 나왔다.

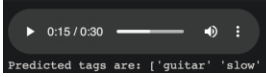


2. 두 번째 곡으로는 첫번째와 대조적인 곡 분위기인 뮤지컬 헤드윅의 tear me down을 선정해보았다. Rock, heavy 정도 태그가 예상된다. 모델은 4-3 모델을 사용하였다.

```
your_audio_path = 'TearMeDown.mp3' #TODO
selected_model = your_model # Change it if you want to select model with different name
```

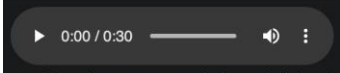
```
def get_resampled_mono_audio_from_file(audio_file_path, target_sr):
  y, sr = torchaudio.load(audio_file_path)
  if sr!= selected_model.sr:
    y = torchaudio.functional.resample(y, orig_freq=sr, new_freq=selected_model.sr)
  if y.shape[0] > 1:
    y = torch.sum(y, dim=0) / y.shape[0]

  return y

def slice_audio(audio_sample, sr, start_sec, end_sec):
  '''
  This function takes an audio sample, sampling rate, and start/end position of slice
  and returns the sliced audio sample.

  audio_sample (torch.Tensor): A sequence of audio samples in shape of (N,), where N is number of audi
  sr (int): Sampling rate of audio_sample
  start_sec (float): desired slice start position in seconds
  end_sec (float): desired slice end position in seconds

  output (torch.Tensor): A sequence of audio samples in shape of (int(sr*(end_sec-start_sec)), )


  TODO: Complete this function
  '''
  start_idx = int(start_sec * sr)
  end_idx = int(end_sec * sr)
  sliced_audio = audio_sample[start_idx:end_idx]
  return sliced_audio
```

```
'''
Run Model
'''


THRESHOLD = 0.2
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' 'slow' 'male' 'vocal']
```

```
THRESHOLD = 0.1
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: ['singer' 'female' 'guitar' 'slow' 'male' 'vocal']
```

노래를 들으며 생각한 태그와 거의 일치하는 모습을 보였다.