Project Report

전지훈

I. Introduction

전통적으로 CNN 기반 모델들은 컴퓨터비전(Computer Vision) 분야에서 가장 활발하게 활용되어 왔으나, 최근에는 Transformer와 Diffusion 등 새로운 방법론들의 등장으로 컴퓨터비전 분야에서 점점 찾아보기 어려워지고 있다. 하지만 Machine Listening 분야에서는 여전히 CNN 레이어들이 Feature Extraction을 비롯한 전반적인 과정에서 중추적인 역할을 담당하고 있기에, 본 프로젝트에서는 Machine Listening에 관한 모델을 구현해 보기로 하였다.

Machine Listening 분야에서는 ASR(Automatic Speech Recognition), 음성 합성 및 변조, 언어 인식, TTS(Text-to-Speech) 및 STS(Speech-to-Text) 등 다방면으로 폭넓은 연구가 이루어지고 있는데, 영상에서 추출된 오디오 데이터에 대한 체계적인 분류에 관하여는 선행연구가 많지 않음을 확인하였다. 이에 본 프로젝트에서는 지난 2017~2019년에 걸쳐 Google Research가 Youtube에서 추출하여 공개한 AudioSet이라는 데이터셋을 이용하여 음성 분류 모델을 구현해 보았다.

II. Dataset

AudioSet(https://research.google.com/audioset/dataset)은 지난 2018~2019년에 걸쳐 진행된 Youtube8M(https://research.google.com/youtube8m/) 프로젝트의 사이드 프로젝트로, 208만 4320개의 유튜브 비디오에서 추출한 오디오 데이터들로 구성되어 있으며, 각 오디오마다 관련 label이 annotation되어 있다. Label의 총 종류 수는 527가지이며, 특정 오디오 데이터에 2개 이상의 label이 annotation 되어 있는 경우도 매우 많다. 따라서, 해당 데이터셋에 대한 분류 태스크는 전통적으로 가장 흔한 이진 분류(Binary classification)나 다중 분류(Multiclass classification) 문제가 아니라, 다중라벨 분류(Multilabel classification) 문제에 속한다고 볼 수 있다.

데이터셋 버전으로 bal_train, eval, unbal_train의 총 3가지가 있는데, bal_train과 eval은 각 label의 출현 빈도수가 비교적 균형 있게 구성되어 있으며, 각각 22176개, 20383개의 오디오 데이터가 포함되어 있다. 반면, unbal_train은 이러한 구성이 보장되지 않은 전체 데이터셋이며, 포함된 총 오디오 데이터의 수는 204만 2895개이다.

Raw 오디오 데이터는 일반적으로 시간을 축으로 하여 이동하는 overlapping sliding

window를 이용하여 각 구간별로 푸리에변환(Fourier Transform)을 통해 frequency 값을 얻는 방식으로 전처리가 이루어진다. 이 때, sliding window 구성을 위해 각 시간 간격을 얼마나 잘게 쪼갰는지를 sampling rate을 통해 나타내는데, AudioSet에서는 대부분의 Machine Listening 분야 논문들과 마찬가지로 16kHz, 즉 초당 16000개로 쪼개는 샘플링 방식을 택하였다. 하지만 인간의 청각은 Hz 간 차이를 실제 그 수치에 비례하게 감지하지 못 하므로, 인간이 느끼는 실제 간격을 고려한 Mel-frequency로 앞선 푸리에변환 결과를 가공하는 작업이 요구된다. 해당 가공 과정의 코드는 Tensorflow의 AudioSet github repository (https://github.com/tensorflow/models/blob/master/research/audioset/vggish/vggish_input.py)에서 찾아볼 수 있었으며, Mel-frequency로의 변환 이후 CNN 등을 이용한 feature extraction (https://github.com/tensorflow/models/blob/master/research/audioset/vggish/vggish_slim.py)을 거친 결과값을 AudioSet이라는 이름의 데이터셋으로 공개하고 있었다.

본 프로젝트에서는 이들이 활용한 200여만개의 raw Youtube 비디오 데이터셋에 대해 음성 추출 및 초기 가공 등부터 다시 진행하기에 상당한 시간적 한계가 존재한다는 점을 고려하여, 이들이 공개한 오디오 데이터별 feature extraction 결과값을 이용하여 오디오 분류 태스크를 진행해 보기로 하였다.

한편, http://storage.googleapis.com/asia_audioset/youtube_corpus/v1/features/features.tar.gz에서 bal_train, eval, unbal_train의 3가지 버전 내 각 오디오별 feature 데이터를 다운받을 수 있었는데, 각 오디오 데이터별로 다음과 같은 TFRecord 파일 형식으로 저장되어 있어, pytorch 환경 상에서 활용하기 위해서는 몇 가지 과정을 거쳐야 했다.

[그림 1] AuidoSet 데이터 TFRecord 파일 형식

위 [그림 1]에서 확인할 수 있듯, 각 오디오 데이터는 크게 context feature와 embedding feature로 이루어져 있으며, context feature에는 해당 오디오가 추출된 비디오의 id (1가지), 해당 비디오 내에서 해당 오디오가 시작 및 종료되는 second (각각 1가지), 그리고 해당 오디오에 대해 사람들이 annotate한 label(2가지 이상인 경우 많음)들이 들어있다. 이 때, 절대다수의 오디오 데이터가 10초로 그 길이가 통일되어 있음을 확인할 수 있었다. 또한

embedding feature는 오디오 데이터의 10초 내 각 1초별로 128차원씩의 벡터 값으로 구성되어 있었으며, 각 128차원 벡터는 데이터 저장 및 호출 시의 용량 최적화를 위해 8비트로 양자화되어 있는 특징을 보였다.

III. TFRecord to Pandas

우선, 모델 학습에 필요한 오디오 feature 데이터 확보를 위해 torchdata 모듈의 datapipe를 활용하여 TFRecord 형식의 데이터를 전처리 과정에 활용하기 용이한 Pandas dataframe으로 변환하는 과정을 거쳤다. 이를 위해 다음과 같은 tfrecord_to_df라는 함수를 작성하였다.

```python
def tfrecord_to_df(data_name, time_size=10, sec_embed_dim=128, label_num=527):
    """
    Args:
        data_name (str): tfrecord dataset type (bal_train / eval / unbal_train)
        time_size (int, optional): Time horizon per each audio data. Defaults to 10.
        sec_embed_dim (int, optional): Embedding dimension per second. Defaults to 128.
        label_num (int, optional): Total number of distinct labels in dataset. Defaults to 527.
    """

    # Using torchdata, we can call tfrecord files into datapipe format
    datapipe1 = FileLister(f"./audioset_v1_embeddings/{data_name}", "*.tfrecord")
    datapipe2 = FileOpener(datapipe1,mode="b")
    tfrecord_loader_dp = datapipe2.load_from_tfrecord()

    data_list = []
    for record in tqdm(iter(tfrecord_loader_dp)):
        record_dict = dict()

        # Directly append tensor datatype values
        for tensor_item in ['start_time_seconds','end_time_seconds','labels']:
            if tensor_item == 'labels':
                # Since label values are not unique within a single audio data,
                # we need to append label data in the list format
                record_dict[tensor_item] = record[tensor_item].tolist()
            else:
                # We can simply append number value
                record_dict[tensor_item] = record[tensor_item].item()

        # First indexing is because len(record_dict['video_id']) == 1,
        # and Second indexing automatically decodes the quantized data into python string
        record_dict['video_id'] = record['video_id'][0][0]

        # Each embedding vector (per second) has the size of 1*128
        # So first indexing makes us explore the embedding data,
        # and second indexing automatically decodes the quantized data into python float
        record_dict['audio_embedding'] = [[embedding[0][i] for i in range(len(embedding[0]))] for embedding in record['audio_embedding']]
        data_list.append(record_dict)


    df = pd.DataFrame(data_list).set_index(['video_id'])
    df = df[~df.isnull()]  # Drop null row (Each audio tfrecord file has exactly 1 null row.)
    df = df[df['audio_embedding'].apply(lambda x: len(x)) == time_size]  # Eliminate audio data whose time length is not 10 sec

    # embeddings = np.vstack(df['audio_embedding'].apply(lambda x: np.vstack(x))).reshape(-1,time_size,sec_embed_dim)
    # np.save(f'{data_name}_embedding.npy',embeddings)

    def label_converter(x):
        output = np.zeros(label_num,dtype=int)
        for label in x:
            output[label] = 1
        return output

    # Convert dense label data into sparse data, whose 1-valued indices are original label data
    df['label'] = df['labels'].apply(lambda x: label_converter(x))

    # labels = np.vstack(df['label']).reshape(-1,label_num)
    # np.save(f'{data_name}_labels.npy',labels)

    df.to_parquet(f'{data_name}.parquet')
```

[그림 2] tfrecord_to_df 함수

tfrecord_to_df 함수에는 tfrecord를 pandas dataframe으로 바꾸는 것 이외에도, 몇 가지 전처리 로직이 구현되어 있다. 우선, 각 오디오 데이터의 labels 데이터가 multi-valued일 가능성이 높은 점을 고려하여, 이들은 우선 리스트 데이터 형태로 저장하게 하였으며, 양자화 저장되어 있는 video_id와 audio_embedding의 경우, 파이썬의 인덱싱 기능을 이용하여 각각 string과 float 데이터로 변환하게 하였다. 또한 각 오디오 데이터에 정확히 1행씩 들어 있는 null row를 삭제하고, 100%에 가까운 대부분 오디오 데이터가 10초 길이로 되어 있다는 점을 감안하여, 모델 학습 시의 통일성을 위해 10초가 아닌 오디오 데이터는 모두 drop하였다. 마지막으로, [1,3,5,6]과 같이 해당 오디오 데이터의 label들로 이루어진 labels 데이터를 모델 학습에 효율적으로 활용하기 위해, 총 라벨 종류 수인 527차원 리스트로 확장하고 label 값들에 해당하는 인덱스들은 1, 그렇지 않은 인덱스들은 0의 값을 갖도록 변환해 주었다.

IV. Data Preprocessing

```python
def df_to_tensor(data_names, top_label_num=top_label_num, embed_dim=embed_dim, embed_max=embed_max, label_num=label_num):
    """
    Args:
        data_names (list[str]): list of dataset versions to use
        top_label_num (int, optional): Number of most frequent labels to use. Defaults to 10.
        embed_dim (int, optional): Embedding dimension per second. Defaults to 128.
        embed_max (int, optional): Maximum Possible value in the embedding vectors. Defaults to 256.
        label_num (int, optional): Total number of distinct labels in dataset. Defaults to 527.

    Returns:
        tuple(normalized_embedding_tensor, label_tensor): tuple of embedding tensor after normalization, and the 527-dim label tensor
    """

    # Concat dataframes corresponding to the data_names
    df = pd.DataFrame([])
    for data_name in data_names:
        add_df = pd.read_parquet(f'{data_name}.parquet')
        add_df = add_df[~add_df.isnull()]
        print(f"{data_name}_df shape: ",add_df.shape)
        add_df = add_df[add_df['audio_embedding'].apply(lambda x: len(x)) == time_size]
        df = pd.concat([df,add_df],axis=0)
        del add_df

    def label_converter(x):
        output = np.zeros(label_num,dtype=int)
        for label in x:
            output[label] = 1
        return output
    # Until here, same code as in the tfrecord_to_df function

    # Convert "labels" column of df into np array
    label_df = pd.DataFrame(np.vstack(df['labels'].apply(lambda x: label_converter(x))).reshape(-1,label_num))
    top_label_idx = list(label_df.sum(axis=0).nlargest(top_label_num).index)  # Top 10 most frequent labels
    label_df = label_df[top_label_idx]  # Only use top10 label data
    label_df = label_df[label_df.sum(axis=1)>0]  # Drop audio data that does not contain any of top 10 label data
    label_tensor = torch.Tensor(label_df.to_numpy())  # Obtain pytorch tensor
    print(f"Total label shape: ",label_tensor.size())

    # Drop audio data that does not contain any of top 10 label data
    df['label'] = df['label'].apply(lambda x: x[top_label_idx])
    df = df[df['label'].apply(lambda x: sum(x))>0]

    # Convert "audio embedding" column of df into np array and torch tensor
    embeddings = np.vstack(df['audio_embedding'].apply(lambda x: np.vstack(x))).reshape(-1,time_size,embed_dim)
    embedding_tensor = torch.Tensor(embeddings)
    print(f"Total embedding shape: ", embedding_tensor.size())

    assert embedding_tensor.shape[0] == label_tensor.shape[0], "Feature and label dim does not coincide!"

    return embedding_tensor/embed_max, label_tensor   # normalize embedding_tensor by its maximum possible value
```

[그림 3] df_to_tensor 함수

III장의 과정을 통해 AudioSet에서 제공하는 TFRecord 형식의 파일들을 pandas dataframe들로 저장한 후, 이들을 불러와 모델 학습 및 테스트에 필요한 Pytorch Tensor 데이터셋들로 변환하는 과정을 위 [그림 3]의 df_to_tensor 함수를 구현하여 처리하였다. 이 함수에서는 사용할 AudioSet 데이터 버전들을 파라미터로 받아, 해당 버전의 데이터를 담고 있는 dataframe들을 merge하고, tfrecord_to_df 함수에서 거친 간단한 전처리 과정을 다시 한 번 거치게 하였다. 그 다음, labels 열을 numpy array로 변환하고, 가장 빈번하게 등장하는 10개의 라벨들을 찾아, 해당 라벨을 포함하고 있는 데이터들만 남기는 작업을 진행하였다. 전체 527개의 라벨들 중 이처럼 10개의 라벨들만을 이용하기로 한 이유는, 10개의 라벨들만으로 전체 데이터의 절반 이상이 구성되어 있고, infrequent한 군소 라벨들까지 지나치게 많이 활용하는 것은 간단한 분류 문제 해결을 목표로 하는 본 프로젝트에 부합하지 않는다고 판단하였기 때문이다. 또한 마지막에 embedding_tensor를 반환할 때 embedding 벡터의 각 element들이 가질 수 있는 최댓값이 255 (8bit 양자화되어 있었기 때문)인 점을 고려하여, embedding_tensor를 element-wise 256으로 나눠주는 normalize 연산을 수행하였다.

```
# Used bal_train and eval version of AudioSet data
embedding_all, label_all = df_to_tensor(['bal_train','eval'], top_label_num)

# Randomly splitting into train / validation / test data (8:1:1)
permute_idx = np.random.permutation(np.arange(embedding_all.shape[0]))
train_idx = permute_idx[:(permute_idx.shape[0]//5)*4]
val_idx = permute_idx[(permute_idx.shape[0]//5)*4:(permute_idx.shape[0]//10)*9]
test_idx = permute_idx[(permute_idx.shape[0]//10)*9:]
print(f"Train data Size: {len(train_idx)} , Val data Size: {len(val_idx)},Test data Size : {len(test_idx)}")

train_set = TensorDataset(embedding_all[train_idx], label_all[train_idx])
val_set = TensorDataset(embedding_all[val_idx], label_all[val_idx])
test_set = TensorDataset(embedding_all[test_idx], label_all[test_idx])

train_loader = torch.utils.data.DataLoader(dataset=train_set, batch_size=bs, shuffle=True, num_workers=cpu_count)
val_loader = torch.utils.data.DataLoader(dataset=val_set, batch_size=1, shuffle=False, num_workers=cpu_count)
test_loader = torch.utils.data.DataLoader(dataset=test_set, batch_size=1, shuffle=False, num_workers=cpu_count)
```

```
bal_train_df shape:  (21782, 5)
eval_df shape:  (19976, 5)
Total label shape:  torch.Size([22582, 10])
Total embedding shape:  torch.Size([22582, 10, 128])
Train data Size: 18064 , Val data Size: 2258,Test data Size : 2260
```

[그림 4] DataLoader 생성

본 프로젝트에서는 AudioSet의 bal_train과 eval_df를 합친 데이터를 활용하였으며, 합친 데이터를 8:1:1의 비율로 train, validation, test dataset으로 분할한 후, Pytorch 학습 과정에 활용하기 위해 DataLoader에 넣어주었다. bal_train과 eval_df를 합친 41758개의 audio 데이터 중, 본 프로젝트에서는 상위 10개 라벨을 포함하는 22582개의 audio 데이터를 활용하였다.

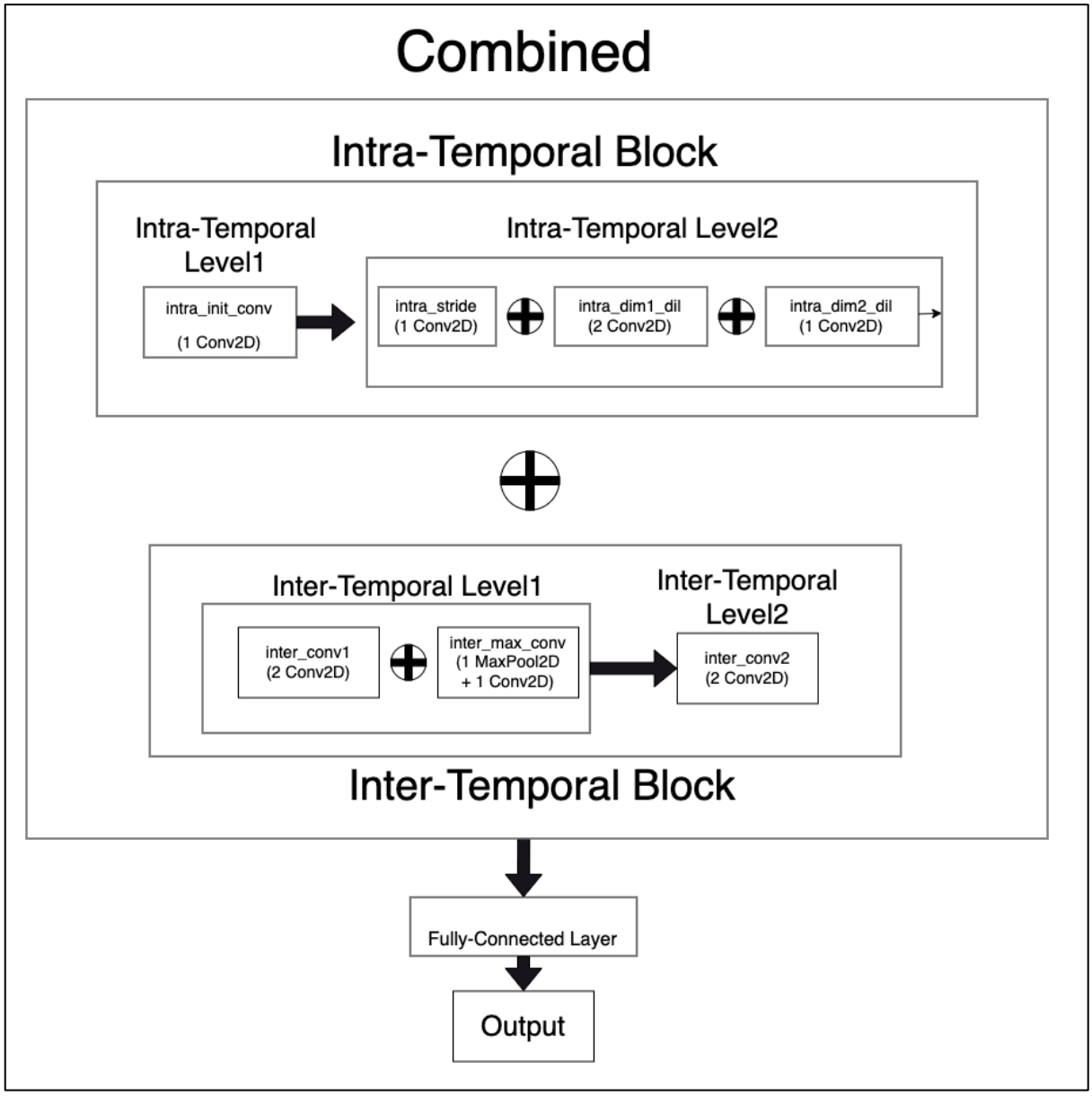
V. YoutubeAudioClassifier

CNN을 이용한 모델을 구현하는 본 과제의 목적을 고려하여, CNN layer들로 구성된 YoutubeAudioClassifer라는 모델을 Pytorch nn.module로 구현하였다.

```
class YoutubeAudioClassifier(nn.Module):
    def __init__(self, time_size=time_size, embed_dim=embed_dim, fc_dim=64, output_dim=top_label_num):
        """Consists of intra-temporal CNN blocks and inter-temporal CNN blocks, followed by 1 FC layer.

        Args:
            time_size (int, optional): Time horizon per each audio data. Defaults to 10.
            embed_dim (int, optional): Embedding dimension per second. Defaults to 128.
            fc_dim (int, optional): Dimension of final fully connected layer. Defaults to 64.
            output_dim (int, optional): Output dimension of the model. Defaults to 10.
        """

        super(YoutubeAudioClassifier, self).__init__()
        self.time_size = time_size
        self.embed_dim = embed_dim
        self.embed_dim1, self.embed_dim2 = self.split_embed_dim()
        self.fc_dim = fc_dim
        self.output_dim = output_dim


        ### For each layers, input and output dimensions are specified in the comment.

        # Level1 intra-temporal CNN block
        self.intra_init_conv = nn.Sequential(
            nn.Conv2d(self.time_size, self.time_size//2, 3, padding=1),    # 10*16*8 -> 5*16*8
            nn.BatchNorm2d(self.time_size//2),
            nn.ReLU()
        )

        # Three Level2 intra-temporal CNN blocks
        self.intra_stride = nn.Sequential(
            nn.Conv2d(self.time_size//2, self.time_size//4, 2, stride=2),  # 5*16*8 -> 2*8*4
            nn.BatchNorm2d(self.time_size//4),
            nn.ReLU()
        )
        self.intra_dim1_dil = nn.Sequential(
            nn.Conv2d(self.time_size//2, self.time_size//2, 3, dilation=(2,1)),  # 5*16*8 -> 5*12*6
            nn.BatchNorm2d(self.time_size//2),
            nn.ReLU(),
            nn.Conv2d(self.time_size//2, self.time_size//4, 3, dilation=(2,1)),  # 5*12*6 -> 2*8*4
            nn.BatchNorm2d(self.time_size//4),
            nn.ReLU()
        )
        self.intra_dim2_dil = nn.Sequential(
            nn.Conv2d(self.time_size//2, self.time_size//4, (2,3), dilation=(1,2), stride=(2,1)),  # 5*16*8 -> 2*8*4
            nn.BatchNorm2d(self.time_size//4),
            nn.ReLU()
        )


        # Two Level1 inter-termporal CNN blocks
        self.inter_conv1 = nn.Sequential(
            nn.Conv2d(1, 4, (3,5), stride=(1,2)),  # 1*10*128 -> 4*8*62
            nn.BatchNorm2d(4),
            nn.ReLU(),
            nn.Conv2d(4, 8, (3,8), dilation=(1,3)),  # 4*8*62 -> 8*6*41
            nn.BatchNorm2d(8),
            nn.ReLU()
        )
        self.inter_max_conv = nn.Sequential(
            nn.MaxPool2d((2,6), stride=(2,3), padding=(1,0)),  # 1*10*128 -> 1*6*41
            nn.Conv2d(1, 8, 3, padding=1), # 1*6*41 -> 8*6*41
            nn.BatchNorm2d(8),
            nn.ReLU()
        )

        # Level2 inter-temporal CNN block
        self.inter_conv2 = nn.Sequential(
            nn.Conv2d(8, 4, (3,3), stride=(1,2)),  # 8*6*41 -> 4*4*20
            nn.BatchNorm2d(4),
            nn.ReLU(),
            nn.Conv2d(4, 2, (3,5), dilation=(1,3), padding=(1,0)),  # 4*4*20 -> 2*4*8
            nn.BatchNorm2d(2),
            nn.ReLU()
        )

        # Concatenating two CNN blocks and normalize
        self.combine_norm = nn.Sequential(
            nn.BatchNorm2d(2),
            nn.ReLU()
        )

        self.fc = nn.Linear(self.fc_dim, self.output_dim)  # 64 -> 10



    # For balanced Width * Height split of input data for intra blocks
    def split_embed_dim(self):
        for i in reversed(np.arange(np.ceil(np.sqrt(self.embed_dim))+1)):
            if self.embed_dim % i == 0:
                return self.embed_dim // int(i) , int(i)
                break


    def forward(self, data):
        intra_data = data.view(-1, self.time_size, self.embed_dim1, self.embed_dim2)  # Input Data for intra-temporal CNN block
        inter_data = data.view(-1, 1, self.time_size, self.embed_dim)  # Input data for inter-temporal CNN block

        intra_block1_out = self.intra_init_conv(intra_data)
        # Concatenate three intra_block2 components
        intra_block2_out = self.intra_stride(intra_block1_out) + self.intra_dim1_dil(intra_block1_out) + self.intra_dim2_dil(intra_block1_out)

        # Concatenate two inter_block1 components
        inter_block1_out = self.inter_conv1(inter_data) + self.inter_max_conv(inter_data)
        inter_block2_out = self.inter_conv2(inter_block1_out).transpose(-2,-1)

        # Concatenate output of intra and inter temporal blocks, and normalize
        cnn_out = self.combine_norm(intra_block2_out + inter_block2_out).view(-1,self.fc_dim)

        # Final fully connected layer
        fc_out = self.fc(cnn_out)

        return F.softmax(fc_out,dim=1)
```

[그림 5] YoutubeAudioClassifer 모델

위 모델 구조를 도식화하면 다음과 같다.



[그림 6] 모델 구조 다이어그램

각 오디오 데이터의 feature가 10*128차원(128은 각 초별 임베딩 벡터, 10은 각 오디오 데이터의 길이)인 점을 고려하여, 시점 간 Convolution과 시점 내 Convolution에 중점을 두는 Block을 각각 생성하였다.

시점 내 Convolution에 중점을 두는 Intra-Temporal Block의 경우, 시점 수인 10을 input channel의 수로 하며, 각 시점별 128차원 feature에 대한 효과적인 feature 추출을 위해 이를 16*8 차원으로 변환한 데이터를 활용하였다. 가장 첫 Layer인 intra_init_conv에서는 filter size = (3,3), padding = (1,1)로 하여 input과 output의 차원을 일치시키면서 초기 convolution을 수행하도록 하였다. 그 다음으로는, stride convolution을 수행하는 Conv2D layer 1개와, 가로축 dilation convolution을 수행하는 Conv2D layer 2개와, 세로축 dilation convolution

을 수행하는 Conv2D layer 1개를 element-wise concatenate하도록 하였다. 이처럼 stride와 축별 dilation을 각각 독립적으로 수행 후 합치는 작업을 통해, 데이터의 각 element 간 다양한 dependency를 효과적으로 학습할 수 있을 것으로 판단하였다.

시점 간 Convolution에 중점을 두는 Inter-temporal Block의 경우, 10*128차원 input data를 1*10*128차원으로 cast한 후, Layer를 거칠수록 channel의 수가 늘어났다가 다시 감소하고, 그 외 차원은 지속적으로 감소하는 구조로 이루어져 있다. 첫번째 building block에서는 2개의 Conv2D layer로 이루어진 inter_conv1과 1개의 MaxPool2D 및 1개의 Conv2D로 이루어진 inter_max_conv를 element-wise concatenate 해 주었다. MaxPool을 활용한 데이터와의 결합을 추가한 이유는 ResNet에서와 같이 기존 input에 대해 특별히 convolution 연산을 가하지 않은 데이터와의 결합을 통해 convolution layer의 representation power를 증가시킬 수 있다고 보았기 때문이다. 1번째 building block을 통과한 데이터는 2개의 Conv2D layer로 이루어진 2번째 building block을 통과하게 된다.

마지막으로, Intra-temporal Block과 Inter-temporal Block을 element-wise concatenate한 후, batchnorm을 통해 정규화하는 과정을 거치고, 그 output이 1개의 Fully-connected layer를 통과하여 최종 output을 반환하게 된다. 이를 위해 Intra-temporal Block과 Inter-temporal Block 구현 과정에서 두 Block의 Output dimension이 일치할 수 있게끔 세심하게 kernel size와 stride, padding, dilation 등을 설정하였다. 또한 위 도식 및 설명에서는 생략하였지만, 각 Conv2D Layer는 기본적으로 BatchNorm2d와 ReLU 활성함수를 함께 통과하도록 설정하였다. 이는 Convolution layer에 BathcNorm2d와 ReLU가 이어지도록 활용할 때 가장 좋은 성능을 보였다는 많은 선행연구들을 따른 것이다.

VI. YoutubeAudioClassifierLight

위 V장에서 구현한 YoutubeAudioClassifer 모델의 학습 및 테스팅을 위한 과정은 Pytorch_lightning 모듈을 이용해 작성하였다. 또한 모델의 학습을 위한 손실함수(loss function)으로는 multi_label_soft_margin_loss와 crossentropy 중 하나를 활용할 수 있도록 하였다. 여기서 multi_label_soft_margin_loss는 본 프로젝트에서 구현하고자 하는 상황과 같은 multi-label 분류 문제를 위해 개발된 손실함수이다. Crossentropy는 원래 이진분류 혹은 다중분류 문제에 활용하는 형태 그대로 이용할 수 없기에, multi-valued label을 처리하고 이를 고려한 손실함수 계산이 가능하도록 추가적인 커스터마이징을 해 주었다. 한편, 모델의 Validation 및 Testing에 활용하는 metric으로는 Precision@1과 NDCG loss를 활용하였다. 마지막으로, 모델의 optimizer로는 Adam을 사용했다. 이를 구현하는 코드는 다음과 같다.

```python
class YoutubeAudioClassifierLight(pl.LightningModule):
    def __init__(self, model, lr, train_loss_type, Precision_k=1):
        """
        Args:
            model (nn.Module): Pytorch-implemened model.
            lr (int): Learning rate for training
            train_loss_type (string): Type of loss function used for training. "msml" or "CE" are allowed.
            Precision_k (int, optional): Number of topk used for Precision@k metric. Defaults to 1.
        """

        super().__init__()
        self.model = model
        self.lr = lr
        self.train_lossF = self.calc_loss(train_loss_type)
        self.ndcg_metric = self.calc_metric("ndcg")
        self.precision_metric = self.calc_metric("Precision@k", Precision_k=Precision_k)

    # Defining loss function for given parameter
    def calc_loss(self, loss_type):
        """
        Args:
            loss_type (string): "msml" or "CE" are allowed.

        Raises:
            ValueError: For "msml", if reduction_type parameter is not valid
            ValueError: If loss_type parameter is not among "msml","ce".

        Returns:
           function: Loss function to be used, corresponding to loss_type parameter.
        """
        if loss_type == 'msml':
            if device == torch.device("mps"):
                # In MPS device, F.logsigmoid is not supported.
                # It is well-known that softplus(beta = -1) is identical to logsigmoid.
                def multilabel_soft_margin_loss(input, target, weight = None, size_average = None, reduce = None, reduction = "mean"):
                    loss = -(target * F.softplus(input, beta = -1) + (1 - target) * F.softplus(-input, beta = -1))

                    if weight is not None:
                        loss = loss * weight

                    class_dim = input.dim() - 1
                    C = input.size(class_dim)
                    loss = loss.sum(dim=class_dim) / C  # only return N loss values

                    if reduction == "none":
                        ret = loss
                    elif reduction == "mean":
                        ret = loss.mean()
                    elif reduction == "sum":
                        ret = loss.sum()
                    else:
                        ret = input
                        raise ValueError(reduction + " is not valid")
                    return ret
                return multilabel_soft_margin_loss
            else:
                return nn.MultiLabelSoftMarginLoss()
        elif loss_type == 'CE':
            # Since label data in AudioSet dataset is not one-hot encoded (b/c possibly multi-valued),
            # Need to properly transform CrossEntropyLoss functon into specific form
            def MultiCELoss(input, target):
                loss = nn.CrossEntropyLoss()
                nonzeros = target.nonzero()
                idx = nonzeros[:,0]
                label = nonzeros[:,1]
                norm_input = input[idx] / torch.bincount(idx)[idx].view(-1,1)  # Normalize each inputdata in batch by its duplicate count
                return loss(norm_input, label)
            return MultiCELoss
        else:
            raise ValueError("Unsupported Loss function")

    # Defining metric function for given parameter
    def calc_metric(self, metric_type, Precision_k=1):
        """
        Args:
            metric_type (str): "ndcg" or "Precision@k" are allowed
            Precision_k (int, optional): Number of topk used for Precision@k metric. Defaults to 1.

        Raises:
            ValueError: If metric_type parameter is not among "ndcg","Precision@k".

        Returns:
            function: Metric function to be used, corresponding to metric_type parameter.
        """
        if metric_type == 'ndcg':
            return retrieval_normalized_dcg
        elif metric_type == 'Precision@k':
            def Precision_at_k(input, target):
                topk_idx = torch.topk(input,Precision_k,dim=1).indices
                return torch.stack([(target[i][topk_idx[i]]).float() for i in range(target.shape[0])]).mean()
            return Precision_at_k
        else:
            raise ValueError("Unsupported Metric function")


    def training_step(self, batch, batch_idx):
        # training_step defines the train loop.
        embedding, labels = batch
        train_loss = self.train_lossF(self.model(embedding), labels)
        self.log("train_loss", train_loss, on_step=True, on_epoch=True, prog_bar=True, logger=True)
        return train_loss

    def validation_step(self, batch, batch_idx):
        # this is the validation loop
        embedding, labels = batch
        valid_ndcg = self.ndcg_metric(self.model(embedding), labels)
        valid_precision = self.precision_metric(self.model(embedding), labels)
        self.log("validation_ndcg", valid_ndcg, prog_bar=True, logger=True)
        self.log("validation_precision", valid_precision, prog_bar=True, logger=True)
        return valid_ndcg, valid_precision

    def test_step(self, batch, batch_idx):
        # this is the test loop
        embedding, labels = batch
        test_ndcg = self.ndcg_metric(self.model(embedding), labels)
        test_precision = self.precision_metric(self.model(embedding), labels)
        self.log("test_ndcg", test_ndcg, prog_bar=True, logger=True)
        self.log("test_precision", test_precision, prog_bar=True, logger=True)
        return test_ndcg, test_precision

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=self.lr)
        return optimizer
```

[그림 7] YoutubeAudioClassifierLight

VII. Train, Validation, Test 과정

```
lr_list = [1, 0.1, 0.01, 0.001, 0.0001]
loss_type_list = ["msml", "CE"]

for lr in tqdm(lr_list):
    for loss_type in tqdm(loss_type_list):
        # Instantiate the model and the corresponding Pytorch_lightning object
        YACLight = YoutubeAudioClassifierLight(YoutubeAudioClassifier(),lr=lr,train_loss_type=loss_type)

        csv_logger = pl_loggers.CSVLogger("./logs", name=f"lr={lr},loss={loss_type}")
        n_epoch = 200

        if device == torch.device("mps"):
            trainer = pl.Trainer(accelerator="mps", max_epochs=n_epoch, logger=csv_logger, check_val_every_n_epoch=50)
        elif device == torch.device("cuda"):
            trainer = pl.Trainer(accelerator="gpu", max_epochs=n_epoch, logger=csv_logger, check_val_every_n_epoch=50)
        else:
            trainer = pl.Trainer(accelerator="cpu", max_epochs=n_epoch, logger=csv_logger, check_val_every_n_epoch=50)

        # Baseline metric (Before Training)
        trainer.test(model=YACLight, dataloaders=test_loader)

        # Train Process
        trainer.fit(model=YACLight, train_dataloaders=train_loader, val_dataloaders=val_loader)

        # Test Result (After training)
        trainer.test(model=YACLight, dataloaders=test_loader)
```

[그림 8] Train, Validation, Test 코드

위 [그림 8]의 코드에서 확인할 수 있듯, Learning rate은 1, 0.1, 0.01, 0.001, 0.0001의 5가지, epoch은 50, 100, 200의 3가지, Train loss function 종류로는 MultiLabelSoftMargin과 (Mutilabel classification 버전으로 변형된) CrossEntropy의 2가지를 실험해 보았으며, 이들을 한 가지씩 바꿔가며 총 30개 케이스에 대해 모델 학습 및 테스트를 진행하였다.

손실함수 중 MultiLabelSoftMargin의 경우, Pytorch에 구현되어 있는 것을 그대로 사용하였으나, CrossEntropy의 경우 기본적으로 정답 라벨이 1개인 Binary Classification 내지 Multi-class classficiation을 대상으로 개발이 되어 있는 관계로, 위 [그림 7]의 코드를 통해 각 batch별로 horizontal table(정답 label이 M개인 오디오 데이터를 정답 label이 1개인 M개의 행으로 펼치는 방법)로 변환하여 Pytorch의 CrossEntropy 함수 적용 후, 다시 원래 batch shape으로 변환함과 동시에, 각 data가 갖는 정답 label의 개수들로 계산된 값들을 normalize 해 주는 과정을 거쳤다. 이와 같은 normalize를 해 주지 않을 시, 정답 label의 수가 많은 오디오 데이터가 그렇지 않은 데이터에 비해 영향력이 지나치게 커지는 현상이 발생할 것으로 생각하여 CrossEntropy 계산에 이러한 조치를 취해 주었다.

Validation과 Test 시의 metric으로는 NDCG와 Precision@1을 사용하였다. 먼저, NDCG(Normalized Discounted Cumulative Gain)는 전통적으로 Information Retrieval 분야에서 널리 쓰여 온 metric으로, 모델의 N차원 output tensor가 relevance를 예측하고자 하는 N개의 item들 중 실제로 relevant한 item들에 높은 값을 부여하고, 그렇지 않은 item들에 낮은 값을 부여할수록 NDCG 값이 높게 도출된다. 구체적인 도출 수식은 다음과 같다.

$$NDCG_p = \frac{\sum_{i=1}^{p} \frac{2^{rel_i - 1}}{log_2(i+1)}}{\sum_{i=1}^{|REL_p|} \frac{rel_i}{log_2(i+1)}}$$

여기서 p는 특정 ranking을, rel_i는 sorting 이후 i번째에 ranked된 item의 relevance 정도를 나타내며, REL_p는 sorting 이후 p번째 rank까지의 relevant item들의 리스트를 나타낸다. NDCG의 가장 큰 특징은, relevance 수치를 rank 값이 낮을수록 (즉, relevance 높은 item일수록) 적게 discount하고, rank 값이 높을수록 (즉, relevance 낮은 item일수록) 많이 discount 한다는 점이다. 이를 통해 모델의 결과값이 실제 item들의 relevance를 올바른 순서로 예측하는지를 확인할 수 있다. Information Retrieval 도메인이 아닌 본 프로젝트에서 다루는 Multi-labeled classification의 평가에 이 metric을 적용해 보기로 한 것은, binary classification이나 Multi-class classification에서와 달리 정답 Label의 값이 하나가 아닌 상황에서는 이와 같은 metric을 통해 모델이 정답 Label에 높은 값을 부여하고 오답 Label에 낮은 값을 부여하는지를 잘 측정할 수 있다고 판단하였기 때문이다. 앞선 III장 및 IV장의 전처리 과정에서 Label 벡터의 각 element들이 0 or 1 값을 갖도록 하였기 때문에, 이에 대한 NDCG score는 1인 label들 (즉, 실제로 해당 오디오 데이터에 annotate되어 있는 라벨들)에 모델이 높은 값을 부여하고, 0인 label들 (즉, 해당 오디오 데이터에 annotate되어 있지 않은 라벨들)에 모델이 낮은 값을 부여하는지 여부를 측정하는 것이 된다.

두번째 평가 metric은 Precision@1으로, 이 역시 Information Retrieval 분야에서 많이 사용되어 온 metric이다. 통상 Precision@k라고 하여, 모델 output이 가리키는 Top-k item들 중에 실제로 relevant item의 비율이 얼마나 되는지를 계산하게 되는데, 본 프로젝트에서는 10개의 비교적 적은 수의 label들을 대상으로 하고 있으므로, 보다 엄격한 성능 측정을 위해 k=1로 두는 것이 바람직하다고 보았다. 따라서, 이를 이용해 모델의 output tensor에서 가장 값이 큰 position에 매칭되는 label이 실제로 해당 오디오 데이터에 annotate되어 있으면 1, 그렇지 않으면 0으로 계산하고, 데이터별로 계산된 이 값의 평균을 집계하는 방식으로 validation 및 testing을 진행하였다.

VIII. Train, Test 결과

앞서 VII장에서 세운 Train 실험 계획(Learning rate은 1, 0.1, 0.01, 0.001, 0.0001의 5가지, epoch은 50, 100, 200의 3가지, Train loss function 종류는 MultiLabelSoftMargin과 Mutilabel classification 버전으로 변형된 CrossEntropy의 2가지 ➔ 총 30개 케이스에)에 따라 학습을 진행한 결과는 다음 [표 1] ~ [표 3]과 같았다.

Learning Rate	Loss Function					
	MultiLabelSoftMargin			변형된 CrossEntropy		
	Epoch=50	Epoch=200	Epoch=1000	Epoch=50	Epoch=200	Epoch=1000
1	0.70264619	0.70264714	0.70264679	2.13571047	2.13529872	2.13478374
0.1	0.66927576	0.67196142	0.66866284	1.89397227	1.90137779	1.89564907
0.01	0.66569060	0.66409033	0.66365331	1.88386261	1.88233029	1.87634527
0.001	0.66413903	0.66358190	0.66093283	1.84795808	1.84299027	1.84172070
0.0001	0.66422563	0.66352713	0.66323381	1.87728548	1.84619772	1.84137558

[표 1] Train Loss

Learning Rate	Loss Function					
	MultiLabelSoftMargin			변형된 CrossEntropy		
	Epoch=50	Epoch=200	Epoch=1000	Epoch=50	Epoch=200	Epoch=1000
1	0.78203964	0.78203964	0.78706073	0.77174371	0.77174371	0.76749897
0.1	0.89088779	0.89747273	0.90349578	0.88983660	0.89135873	0.89123702
0.01	0.90710824	0.91247332	0.91328966	0.88971650	0.89371973	0.89537972
0.001	0.90621477	0.90722852	0.91656589	0.90710890	0.90983426	0.90777611
0.0001	0.90682452	0.90618157	0.90402197	0.89168971	0.90148425	0.90471839

[표 2] NDCG Metric (Test)

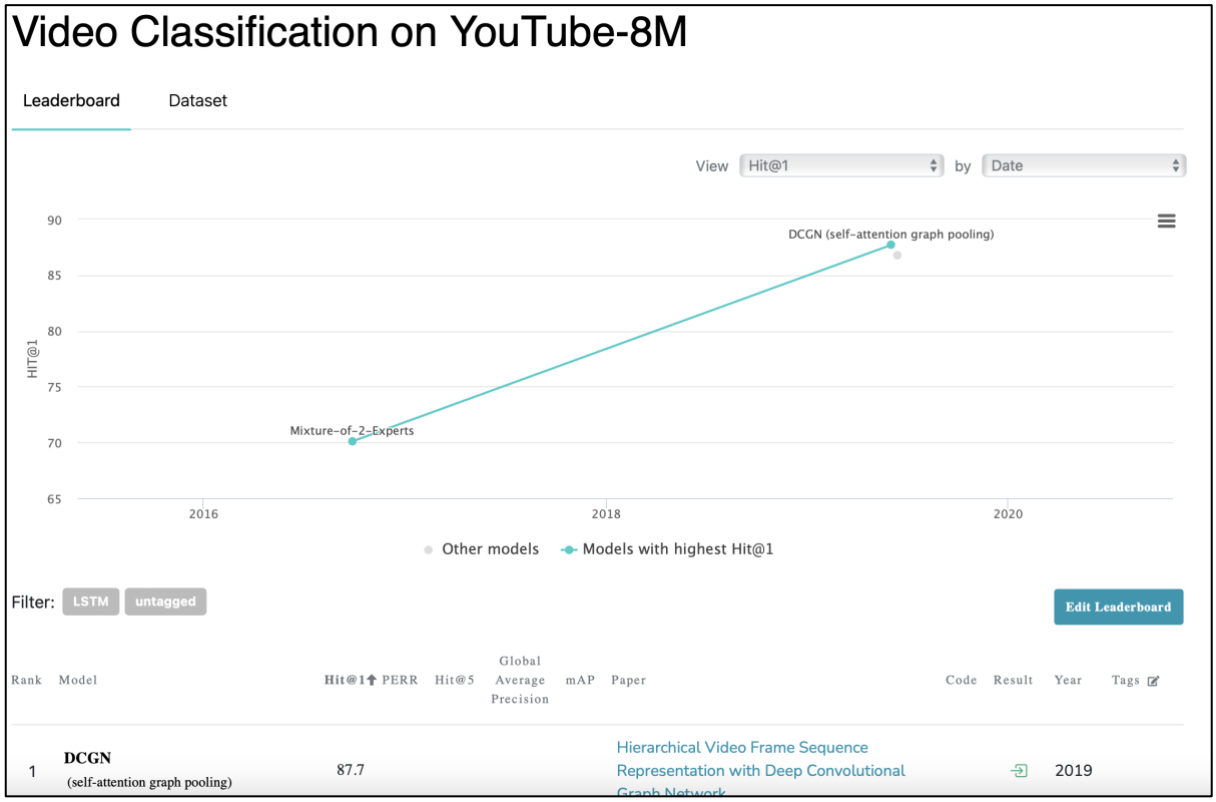
Learning Rate	Loss Function					
	MultiLabelSoftMargin			변형된 CrossEntropy		
	Epoch=50	Epoch=200	Epoch=1000	Epoch=50	Epoch=200	Epoch=1000
1	0.51727193	0.51727193	0.53185838	0.48937112	0.48937112	0.47566372
0.1	0.83082371	0.85296720	0.86858409	0.84278124	0.84410983	0.84380531
0.01	0.87466782	0.88219660	0.88893806	0.85385298	0.85163861	0.85486727
0.001	0.87865370	0.87599647	0.87610620	0.88219660	0.88131088	0.87433630
0.0001	0.87511074	0.87201064	0.86725664	0.85031002	0.86979627	0.87168139

[표 3] Precision@1 Metric (Test)

학습은 Google Cloud Platform 상의 n2-highmem-8 (8 CPU, 64GB RAM) 가상머신에서 cpu 병렬처리를 통해 진행하였으며, batch_size=16 기준으로 각 실험별 epoch 당 대략적으로 train에 18초씩, 전체 실험에는 총 50시간 가량 소요되었다.

위 [표 1] ~ [표 3] 각 표의 Loss function별로 초록색으로 표시된 셀들이 해당 Loss function에 대한 best performance case를 나타낸다. 결과를 종합해 보면, loss function으로는 MultiLabelSoftMargin을 사용하는 경우의 테스트 결과가 근소적으로 더 좋았고, 전체적으로 learning rate = 0.001, Epoch = 1000일 때 가장 좋은 성능을 보였으며, 해당 경우의 NDCG 값은 0.9165, Precision@1 값은 0.8761으로 나타났다.

Youtube8MAudio 데이터셋에 대해 이와 같은 분류 태스크를 한 공개 논문이 없어, 가장 유사한 데이터셋인 Youtube8M 데이터셋에 대한 비디오 분류 논문들의 성능과 비교한 결과, 본 프로젝트에서 구현한 모델의 Precision@1 값이 기존 연구들 중 최고 성능을 보인 모델과 유사한 수준을 보인 것을 확인할 수 있었다.



[그림 9] Youtube-8M 데이터셋에 대한 기존 Video Classification 연구들의 성능

기존에 가장 좋은 성능을 보인 모델의 Precision@1 값은 0.877로, 본 프로젝트에서 구현한 모델의 learning rate = 0.001, Epoch = 1000일 때의 값인 0.8761과 유사하고, learning rate = 0.01, Epoch = 1000일 때의 값인 0.8889보다 낮은 수준을 보이고 있다. 비록 데이터셋이 완전히 일치하는 것은 아니지만, Youtube8MAudio 데이터셋이 Youtube8M 데이터셋에서 오디오 데이터만을 추출하여 구성한 데이터셋을 감안할 때, 오히려 본 프로젝트에서 다룬 Youtube8MAudio 데이터셋이 Youtube8M 데이터셋에 비해 더 한정된 정보를 갖고 있으므로, 본 프로젝트가 더 어려운 환경 하에서 더 좋은 성능을 보인 것으로 해석할 수 있다.