[정보시스템설계]

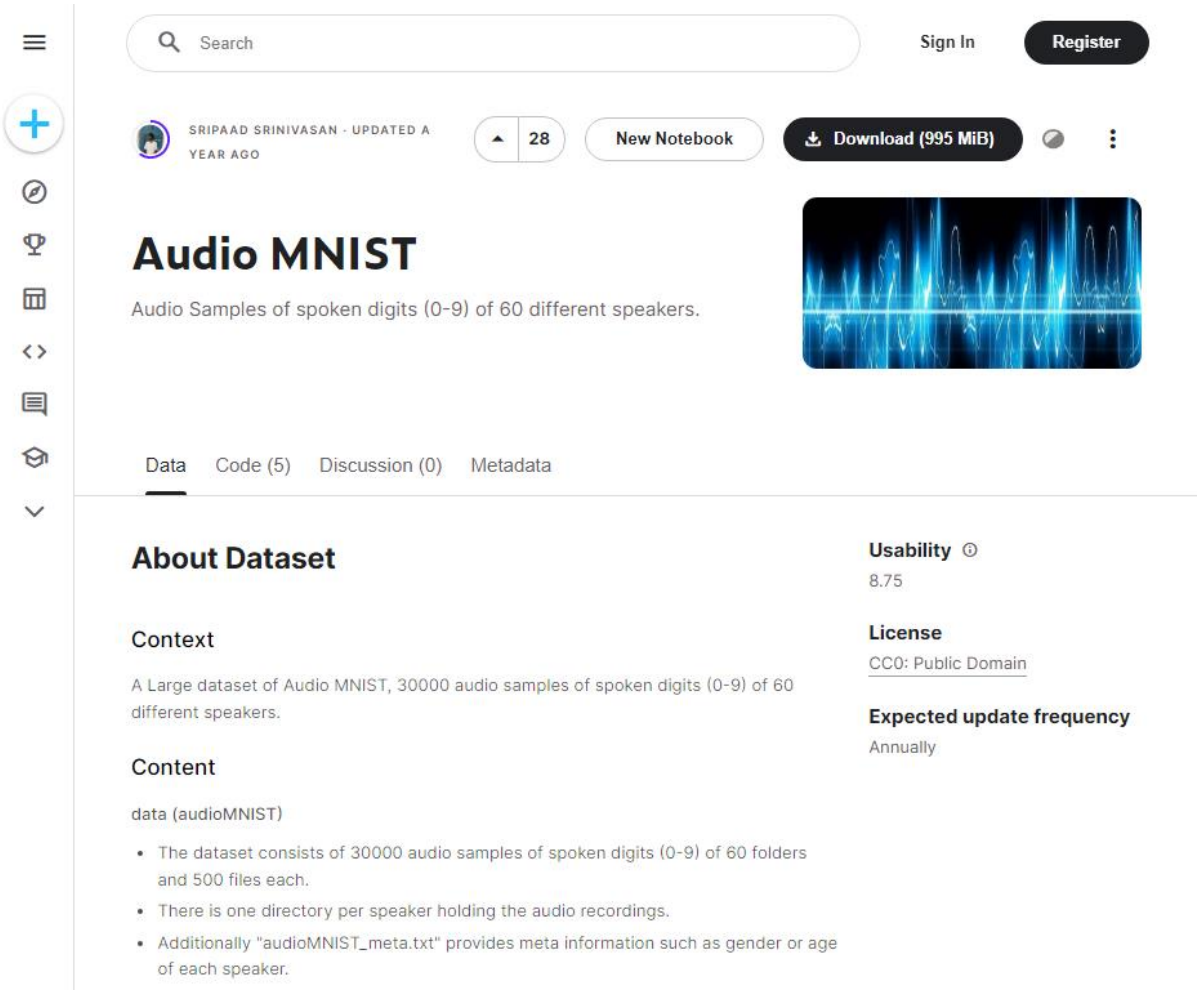
CNN 을 활용한
숫자 오디오 데이터 분류

Term Project IMEN335-00

고려대학교 공과대학
산업경영공학부

송은서 2019131438
박지영 2019170839
장예린 2019170853

1. 문제 정의



60 명의 사람이 녹음한 오디오 데이터셋이다. 0 부터 9 까지의 숫자를 발음하는 것을 녹음한 데이터이며 총 30000 개의 관측치로 이루어져 있다. 이 중 일부만을 Sampling 하여 학습 데이터로 사용하였다. 이 데이터를 통해 구축할 모델은, 해당 오디오에서 어떤 숫자를 말하고 있는지를 분류하는 분류 모델이다.

이번 프로젝트를 통해 오디오 데이터에 대한 딥러닝 분석을 시도해보고자 하였다. 오디오 데이터를 통해 그래프를 얻고 이를 이미지로 표현한 후 CNN 모델을 이용해 분류에 사용하고자 하였다. 오디오 데이터 분석에 자주 사용한다고 하는 Mel Spectrogram, MFCC 의 두가지 방법을 사용하고자 한다.

2. 데이터 탐색

2.1. 필요한 모듈 불러오기

```
import warnings
warnings.filterwarnings(action='ignore')

import seaborn as sns
from glob import glob
import tensorflow as tf
import IPython.display as ipd
import matplotlib.pyplot as plt
```

-오디오 전처리를 위한 라이브러리

```
import librosa
import librosa.display as dsp
from IPython.display import Audio
```

-데이터 전처리를 위한 라이브러리

```
import os
import numpy as np
import pandas as pd
from tqdm import tqdm
```

-모델 구축을 위한 라이브러리

```
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Dropout, Flatten, GlobalAveragePooling2D, Conv2D, MaxPool2D
from tensorflow.keras.layers import ZeroPadding2D, BatchNormalization, Input, DepthwiseConv2D, Add, LeakyReLU, ReLU
from tensorflow.keras.optimizers import Adam, SGD
```

-모델 학습을 위한 라이브러리

```
from sklearn.model_selection import StratifiedKFold
from tensorflow.keras.models import load_model
from sklearn.metrics import accuracy_score
```

-모델의 재현성을 위하여 Random Seed 고정

```
import random

def seed_everything(seed):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    tf.random.set_seed(seed)

seed_everything(929)
```

2.2. 데이터 EDA

-데이터셋 불러오기

```
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')
```

2.2.1. 데이터 확인

```
train.head()
```

	file_name	label
0	001.wav	9
1	002.wav	0
2	004.wav	1
3	005.wav	8
4	006.wav	0

```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   file_name  400 non-null    object
 1   label      400 non-null    int64
dtypes: int64(1), object(1)
memory usage: 6.4+ KB
```

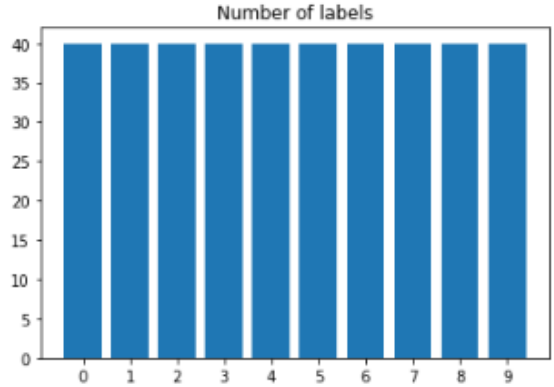
→모든 label 값이 int type 임을 확인할 수 있다.

2.2.2. Label 개수 확인

```
train.label.groupby(train.label).count()
```

```
label
0    40
1    40
2    40
3    40
4    40
5    40
6    40
7    40
8    40
9    40
Name: label, dtype: int64
```

이를 Plot 을 활용해 다음과 같이 시각화하였다.



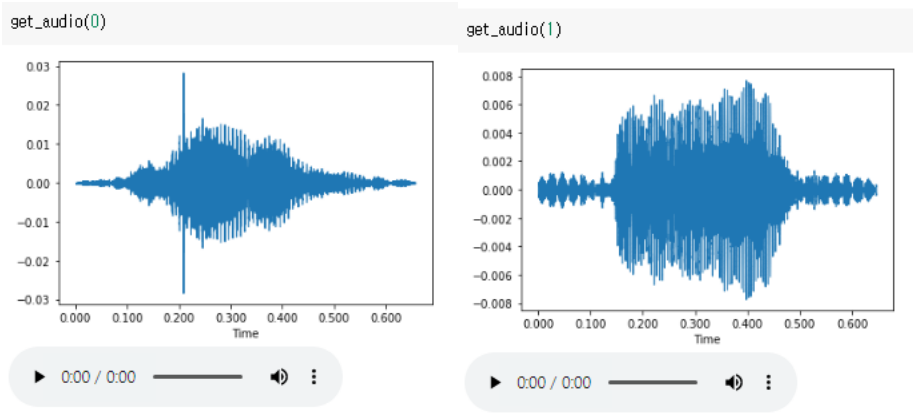
10 가지의 숫자 0~9 에 해당하는 data 가 균일하게 40 개씩 분포하고 있는 것을 확인하였다.

2.2.3. Audio 시각화 1 : 개별 오디오 관측치 불러와서 관찰하기

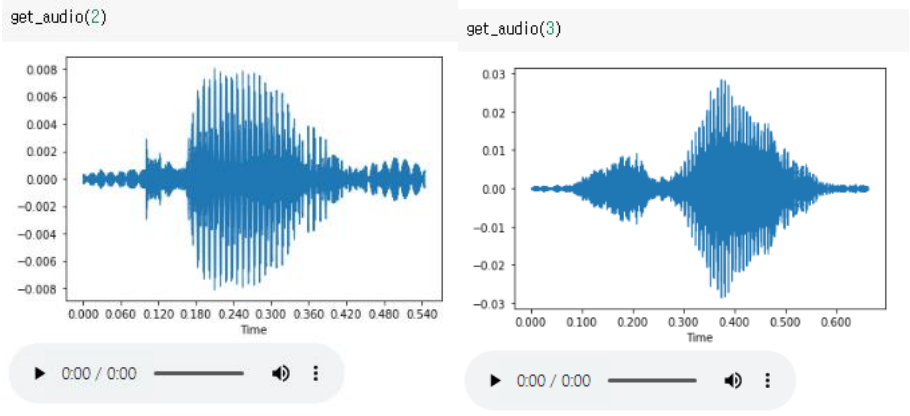
오디오 데이터(wav 형식)에서 각 숫자마다 한 가지씩의 data 를 추출하여 소리를 들어보았다. 관련 code 는 아래와 같다.

```
def get_audio(num = 0):
    # Audio Sample Directory
    sample = os.listdir('train')
    temp = train[train.label == num].file_name
    file_name = temp[temp.index[0]]

    file = 'train/' + file_name
    # Get Audio from the location
    data,sample_rate = librosa.load(file)

    # Plot the audio wave
    dsp.waveshow(data,sr=sample_rate)
    plt.show()

    # Show the widget
    return Audio(data=data,rate=sample_rate)
```

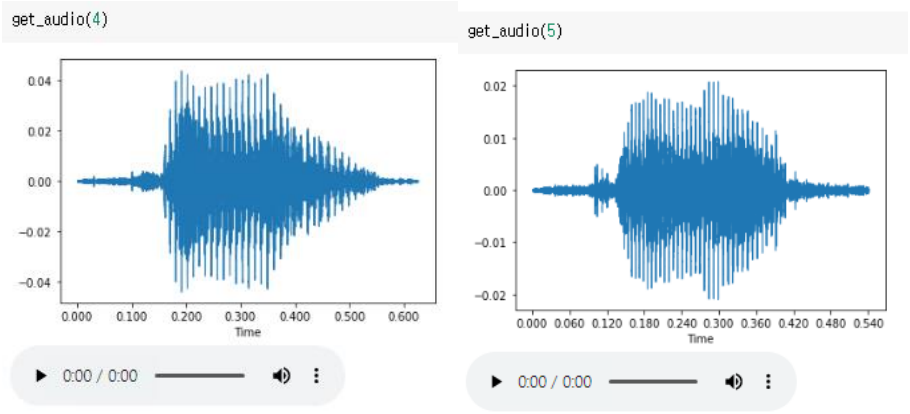
-숫자 0, 1 의 오디오



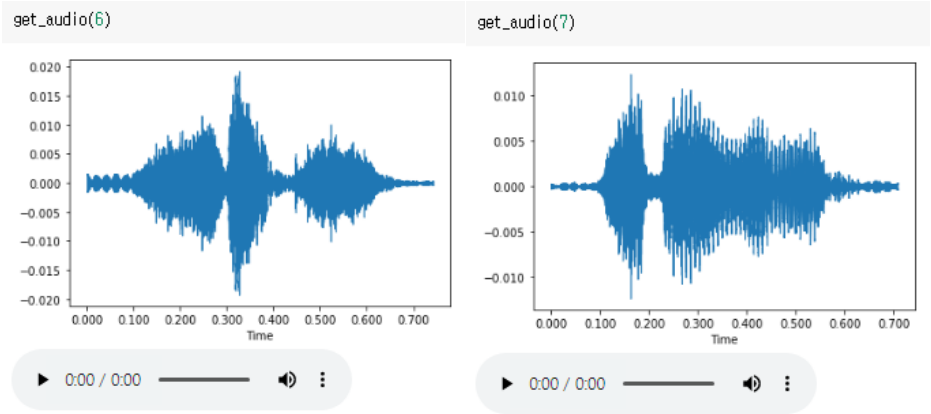
-숫자 2, 3 의 오디오



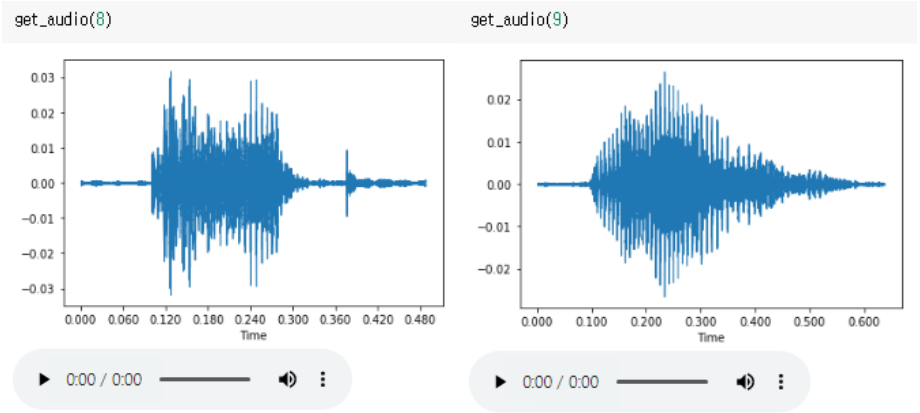
-숫자 4, 5 의 오디오



-숫자 6, 7 의 오디오



-숫자 8, 9 의 오디오



2.2.4. Audio 시각화 2 : 개별 오디오관측치의 퓨리에 변환 관찰하기

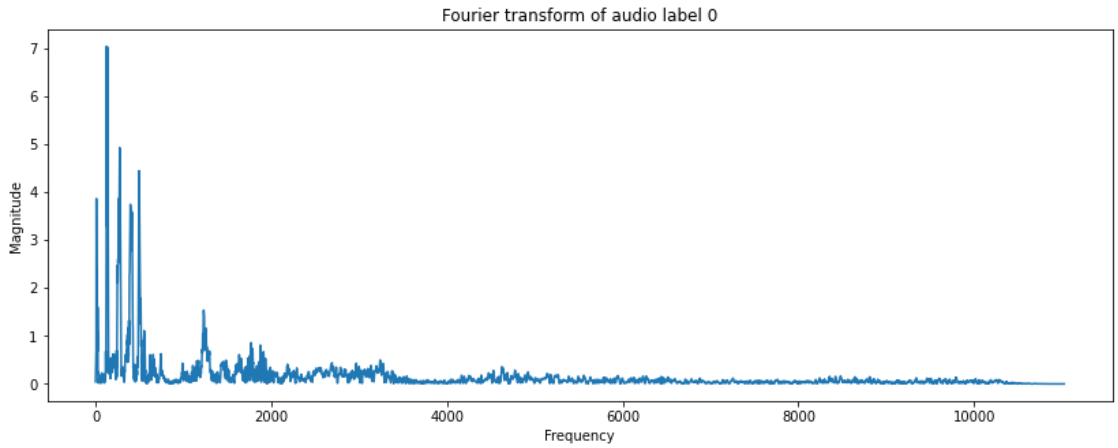
소리는 기본적으로 특정 주파수(frequency)를 갖는 sine 함수들의 합이다. 이를 통해 오디오 데이터 내에서 특정 구간에서 주파수 성분이 어떻게 구성되어 있는지를 확인할 수 있다. 그래서 오디오 데이터 분석을 할 때 파형 자체를 이용하기도 하고, 주파수를 분석하는 기법을 많이

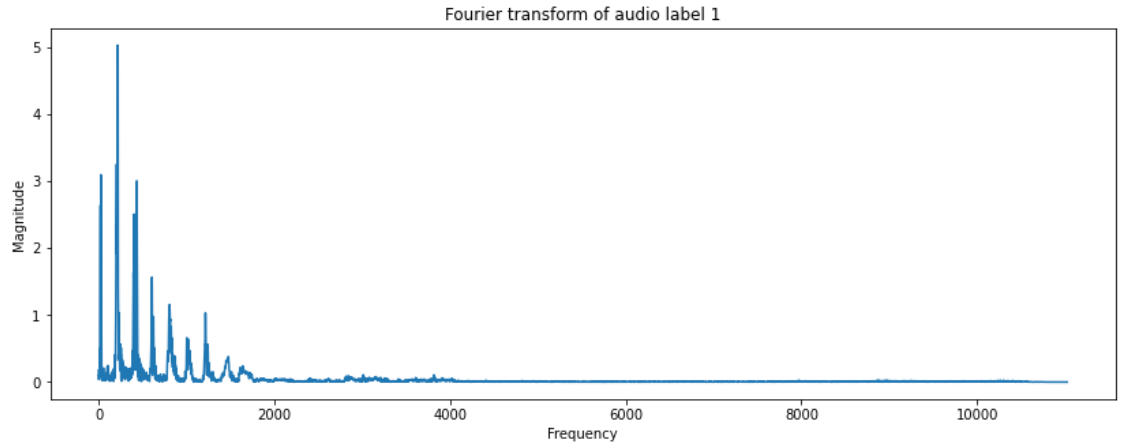
사용한다고 한다. 오디오 데이터를 분석하기 위해, 아날로그 데이터로 되어있는 오디오 데이터를 디지털 신호로 변환해야 한다.

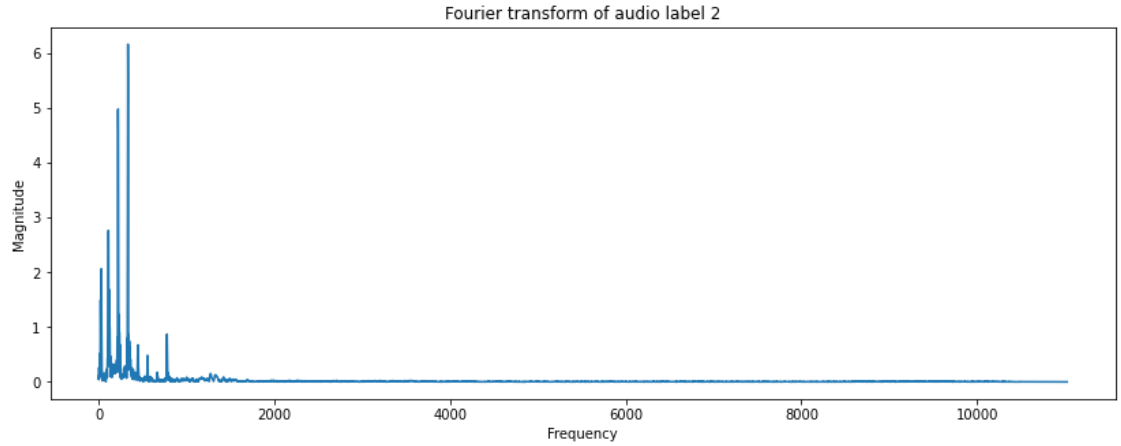
이를 위해 퓨리에 변환(Fourier Transform)을 해야 한다. 퓨리에 변환이란, 입력 신호를 다양한 주파수를 갖는 주기 함수들로 분해하는 것을 뜻한다. 주기 함수들을 분해함으로써 오디오 데이터에서 노이즈 및 배경 소리로부터 실제로 유용한 소리의 데이터를 추출하는 것이다. 퓨리에 변환을 통해 원본 오디오 데이터를 형성하는 주파수의 정도를 파악하고 시각화해보도록 하겠다. 이를 위해 구현한 함수는 다음과 같다.

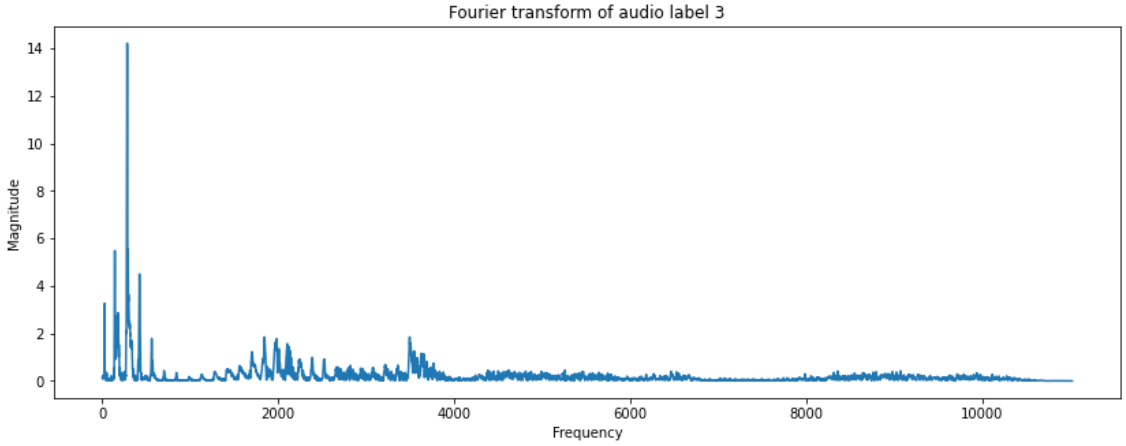
```
def fourier_transform(num = 0):
    # Audio Sample Directory
    sample = os.listdir('train')
    temp = train[train.label == num].file_name
    file_name = temp[temp.index[0]]

    file = 'train/' + file_name
    # Get Audio from the location
    data,sample_rate = librosa.load(file)

    fft = np.fft.fft(data)

    magnitude = np.abs(fft)
    frequency = np.linspace(0, sample_rate, len(magnitude))

    left_frequency = frequency[:int(len(frequency)/2)]
    left_magnitude = magnitude[:int(len(magnitude)/2)]

    fig = plt.figure(figsize = (14,5))
    plt.plot(left_frequency, left_magnitude)
    plt.xlabel("Frequency")
    plt.ylabel("Magnitude")
    plt.title(f"Fourier transform of audio label {num}")
    plt.show()
```

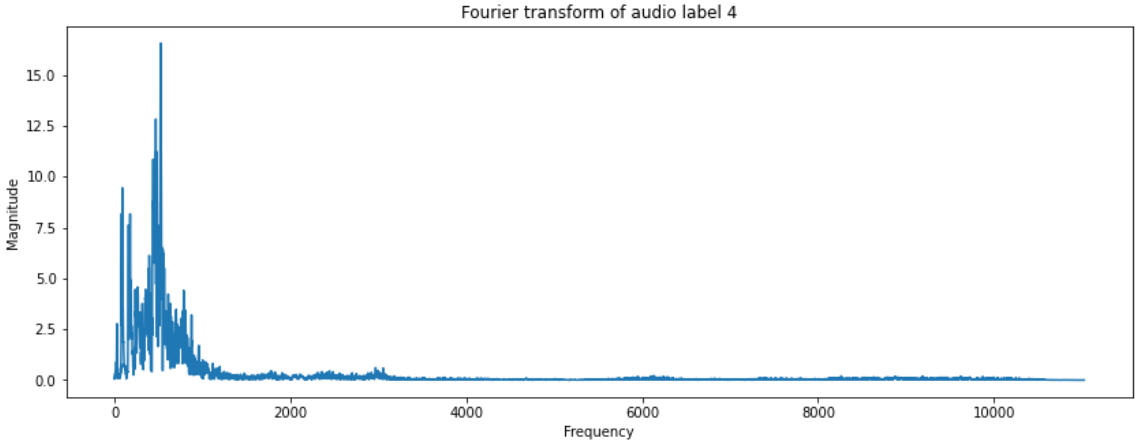
각 숫자에 대해, 주파수에 따른 퓨리에 변환 결과는 다음과 같다.

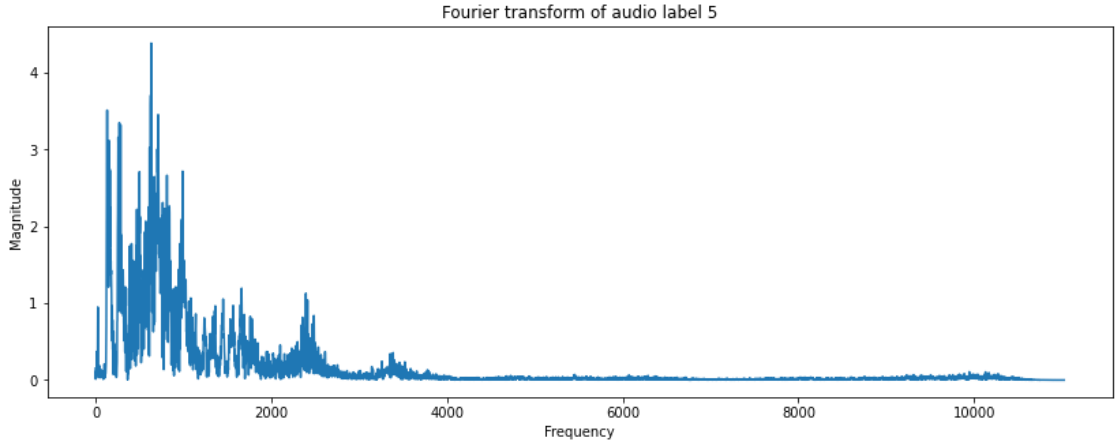


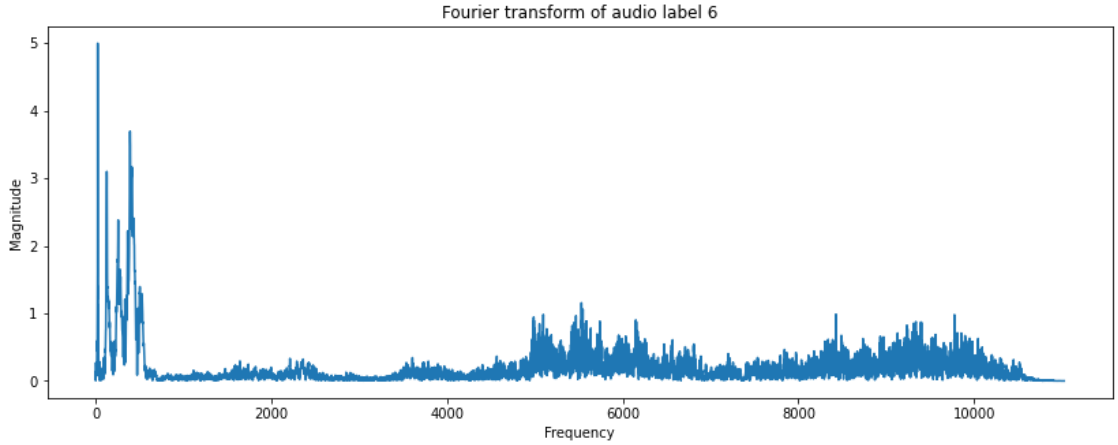

Fourier transform of audio label 1
Magnitude
Frequency
0
1
2
3
4
5
0
2000
4000
6000
8000
10000

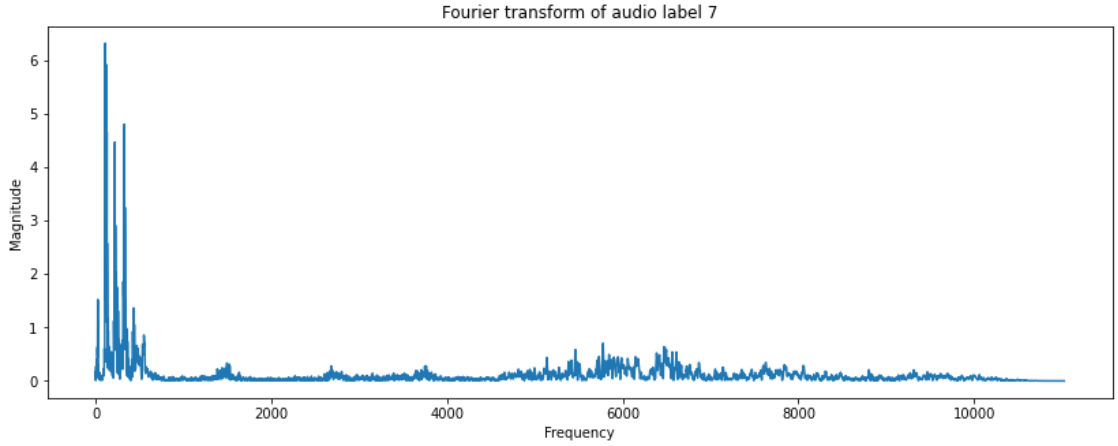

Fourier transform of audio label 2
Magnitude
Frequency
0
1
2
3
4
5
6
0
2000
4000
6000
8000
10000

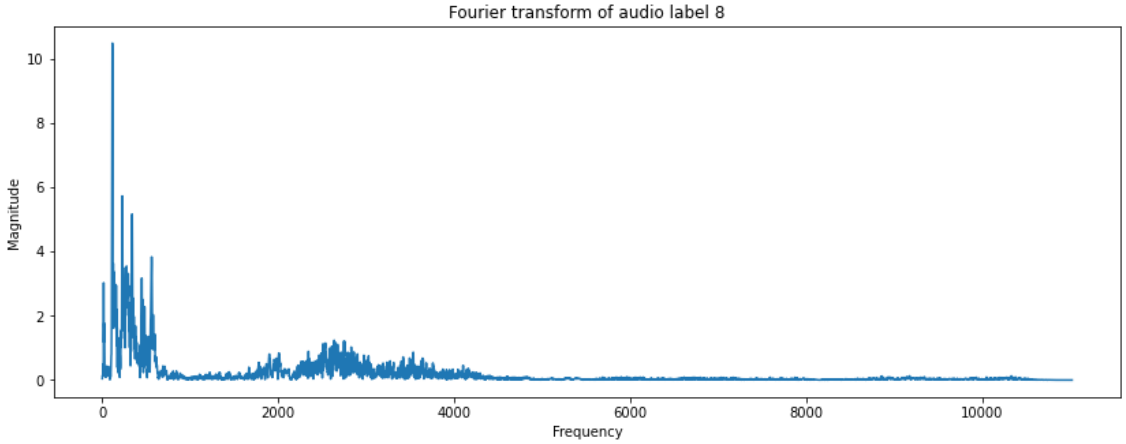

Fourier transform of audio label 3
Magnitude
Frequency
0
2
4
6
8
10
12
14
0
2000
4000
6000
8000
10000

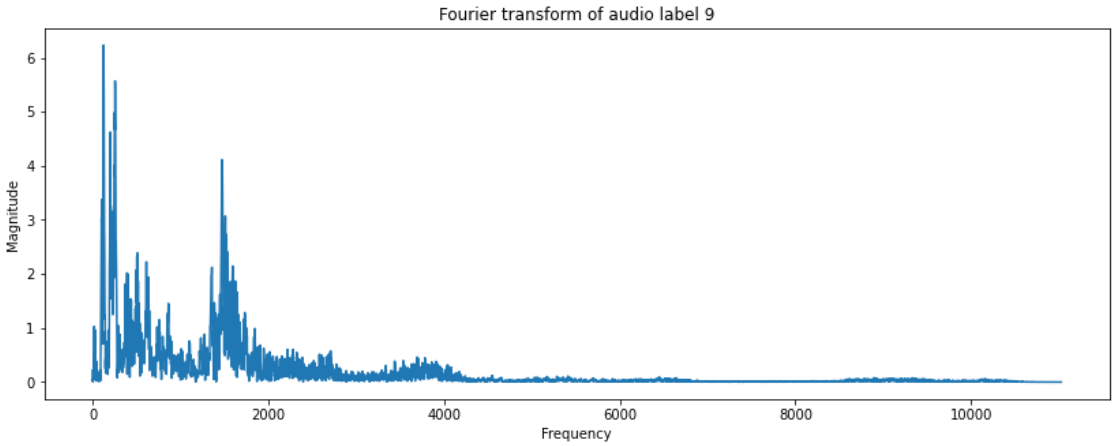

Fourier transform of audio label 4
Magnitude
Frequency
0.0
2.5
5.0
7.5
10.0
12.5
15.0
0
2000
4000
6000
8000
10000


Fourier transform of audio label 5
Magnitude
Frequency
0
1
2
3
4
0
2000
4000
6000
8000
10000


Fourier transform of audio label 6
Magnitude
Frequency
0
1
2
3
4
5
0
2000
4000
6000
8000
10000


Fourier transform of audio label 7
Magnitude
Frequency
0
1
2
3
4
5
6
0
2000
4000
6000
8000
10000


Fourier transform of audio label 8
Magnitude
Frequency
0
2
4
6
8
10
0
2000
4000
6000
8000
10000



이를 통해, 숫자에 따라 에너지의 분포가 상이하다는 것을 확인할 수 있었다. 이후 전처리 단계에서 모델에 분석할 수 있는 형태로 변환하기 위한 벡터화 알고리즘을 수행하고자 하였다.

3. 데이터 전처리

3.1. 오디오 데이터 Load

librosa 라이브러리를 사용하여 오디오 데이터를 불러온 이후에 오디오 데이터의 특징을 추출하겠다. 우선 예시로 train 오디오 데이터의 첫번째 data 를 불러왔다.

```
data, sample_rate = librosa.load('train/001.wav', sr = 16000)
print('sample_rate:', sample_rate, ', audio shape:', data.shape)
print('length:', data.shape[0]/float(sample_rate), 'secs')
```

```
sample_rate: 16000 , audio shape: (10192,)
length: 0.637 secs
```

Audio shape 와 sampling rate 를 이용하여 오디오의 길이를 계산할 수 있다. 위 코드를 통해 계산한 결과, 0.637 secs 의 길이를 갖는다는 것을 알 수 있었다.

이 때 ‘Sampling Rate’란 1 초에 오디오 신호를 몇 번 sampling 할 지를 결정하는 값으로 주파수와 동일하며 이 값은 아날로그 정보를 얼마나 잘게 쪼갤 지를 정한다. 즉 초당 몇 개의 sample 을 가지는 data 를 사용할 것인지를 결정하는 값이 된다. 아날로그 data 를 잘게 쪼갤수록 정보의 손실은 줄어들지만 data 의 크기가 늘어난다.

Sampling rate 의 default 값은 22050Hz 이지만 일반적으로 인간의 청각 영역은 8kHz(8000 sample/second)이므로, 사람 목소리의 경우 sampling rate 는 이의 두배인 16000Hz 안에 포함된다고 하여 sr 값을 16000 으로 설정하였다. 우리가 갖고 있는 숫자 오디오 데이터는 사람의 목소리 data 이기 때문이다.

이후 더 나은 해상도를 위해 sr 값을 더 여유롭게 20000 으로 설정하고 몇 개의 데이터를 더 sampling 하여 살펴보았다.

```
for i in os.listdir('./train')[:3]:
  print(f'{i}')
  data, sample_rate = librosa.load(f'./train/{i}', sr = 20000)
  print('sample_rate:', sample_rate, ', audio shape:', data.shape)
  print('length:', data.shape[0]/float(sample_rate), 'secs')
  print('₩n')
```

```
004.wav
sample_rate: 20000 , audio shape: (12910,)
length: 0.6455 secs


005.wav
sample_rate: 20000 , audio shape: (9753,)
length: 0.48765 secs


012.wav
sample_rate: 20000 , audio shape: (10803,)
length: 0.54015 secs
```

모두 다른 길이를 갖고 있음을 알 수 있었다. 이후 Data Augmentation 단계에서 길이를 맞추는 과정을 수행하고자 한다.

3.2. 오디오 데이터 특징 추출

오디오 raw data 를 그대로 사용하면 parameter 가 너무 많아지기도 하고 data 용량도 너무 커지기 때문에 입력된 신호에서 noise 및 배경 소리로부터 실제로 유용한 소리의 특징만을 추출하고자 한다. 이를 위해 Mel Spectrogram, MFCC 두가지 방법을 이용하고자 한다.

3.2.1. Mel-Scale

특징 추출 전에, 우선 Mel-Scale 이라는 개념을 먼저 소개하고자 한다. Mel 은 사람의 달팽이관을 모티브로 따온 값이다. 달팽이관은 주파수가 낮은 대역(500~1500Hz)에서는 변화하는 주파수를 잘 감지하지만 주파수가 높은 대역(10000Hz~11000Hz)으로 갈수록 간격이 넓어져서 주파수 변화를 감지를 잘 하지 못한다. 즉, 사람은 높은 주파수보다 낮은 주파수의 소리를 더 예민하게 받아들인다. 이러한 원리를 이용하여 오디오 데이터를 Filtering, Scaling 할 수 있다고 한다. 이 때 사용되는 기준을 Mel-Scale 이라고 한다. 주파수 단위를 Mel-Scale 로 변환하는 공식은 다음과 같다.

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right)$$

3.2.2. Mel Spectrogram

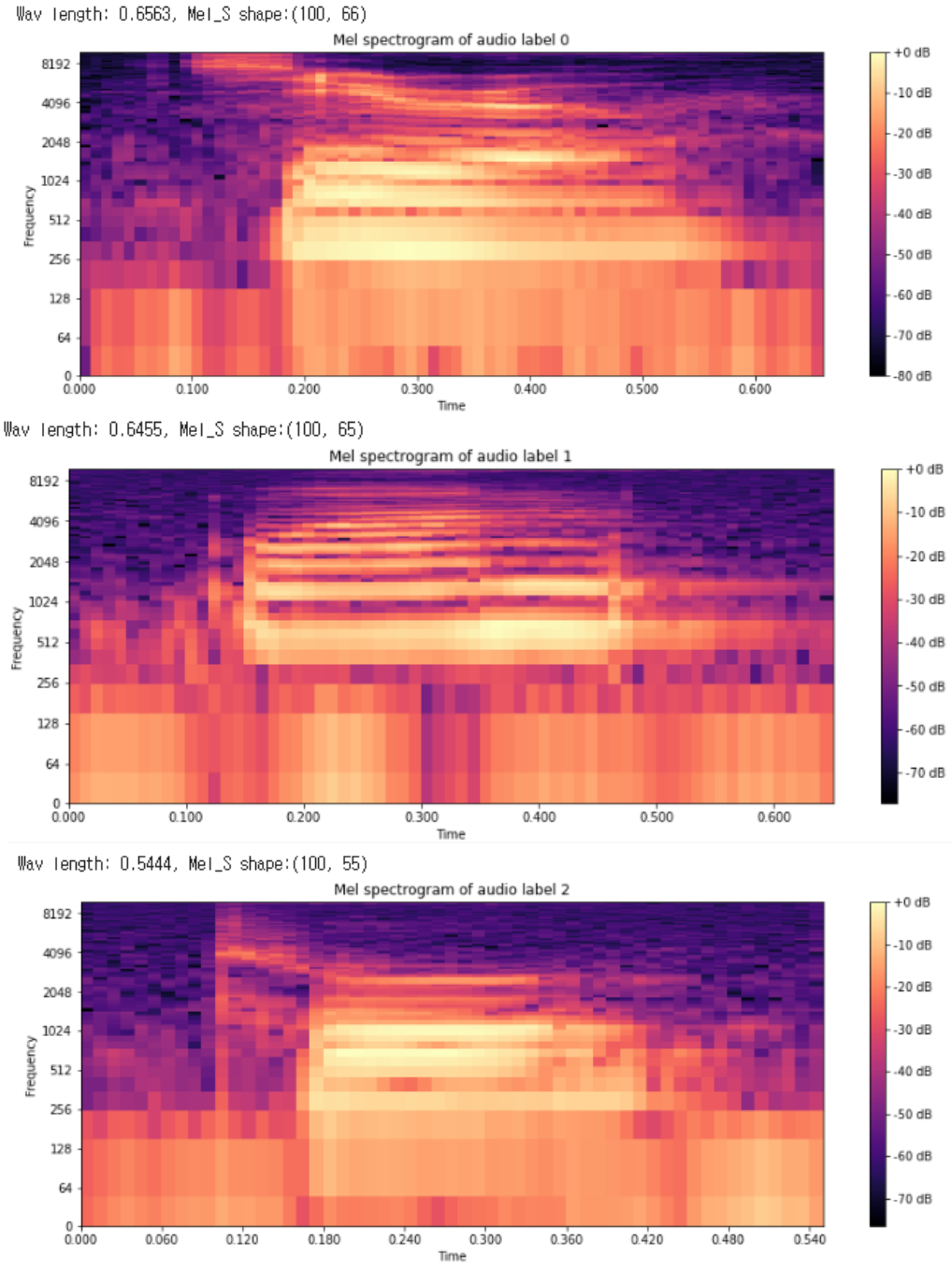
Spectrogram 이란 소리나 파동을 시각화하여 파악하기 위한 도구로, 파형(waveform)과 스펙트럼(spectrum)의 특징이 조합되어 있다. Mel Spectrogram 은 주파수를 Mel-Scale 로 변환한 형태의 spectrogram 이다. 이 mel spectrogram 을 시각화하고자, librosa.feature.melspectrogram 을 사용할 수 있으며 해당 메소드의 몇몇 parameter 에 대해 다음과 같이 간략히 소개할 수 있다.

** y : 오디오의 시계열 데이터*
** sr : y 의 sampling rate*
** n_fft : FFT(Fast Fourier Transform, 고속 퓨리에 변환) window 의 길이. Window size 라고도 부름. Frame 의 length 를 결정하는 파라미터. (sampling rate * frame_length 의 값과 동일함) (ex. 사람의 목소리는 대부분 16000Hz 안에 포함되는데, 일반적으로 자연어 처리에서 오디오는 25m 크기를 기본으로 하고 있음. -> 16000Hz 인 오디오에서 25m 의 오디오 크기를 가지면 n_fft 는 16000*0.025=400 의 값을 가짐)*
** hop_length : window 간 거리 (sampling rate * frame_stride 의 값과 동일함)*
** win_length : window 의 길이 (default = n_fft)*

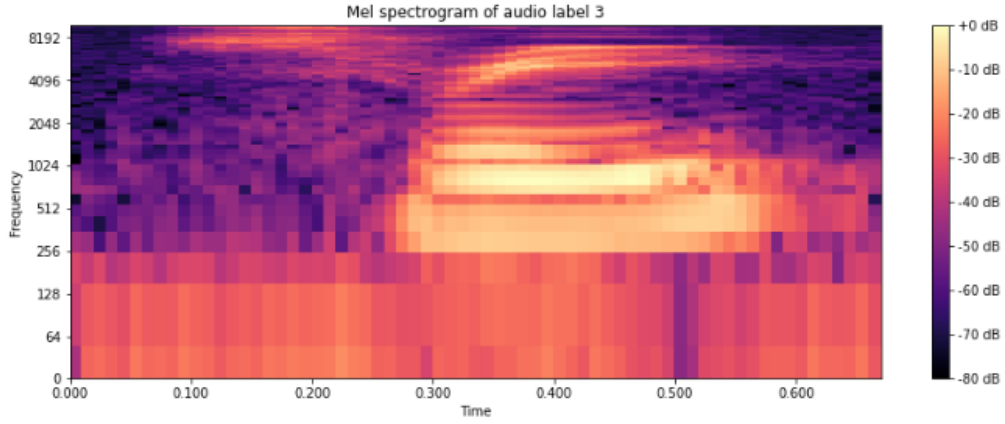
이를 이용하여 Mel Spectrogram 의 시각화를 구현한 함수는 아래와 같다.

```
def Mel_s(num = 0, frame_length = 0.025, frame_stride = 0.010):
    sample = os.listdir('train')
    temp = train[train.label == num].file_name
    file_name = temp[temp.index[0]]

    file = 'train/' + file_name
    data,sample_rate = librosa.load(file, sr = 20000)

    input_nfft = int(round(20000*frame_length))
    input_stride = int(round(20000*frame_stride))

    S = librosa.feature.melspectrogram(y=data, n_mels=100, n_fft=input_nfft, hop_length=input_stride)

    print("Wav length: {}, Mel_S shape:{}".format(len(data)/sample_rate, np.shape(S)))

    S_dB  = librosa.power_to_db(S, ref=np.max)
    fig = plt.figure(figsize = (14,5))
    librosa.display.specshow(S_dB,
                             sr=sample_rate,
                             hop_length=input_stride,
                             x_axis='time',
                             y_axis='log')
    plt.xlabel("Time")
    plt.ylabel("Frequency")
    plt.colorbar(format='%+2.0f dB')
    plt.title(f"Mel spectrogram of audio label {num}")
    plt.show()
```

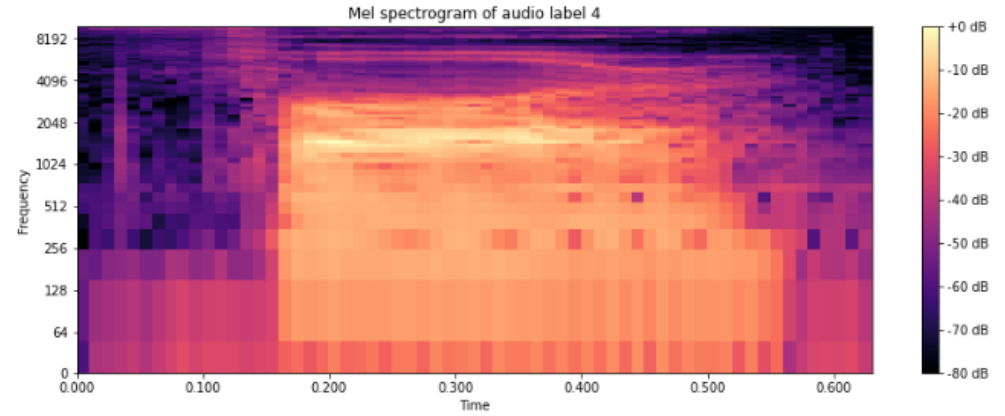
위 함수를 이용해 각 숫자에 대한 Mel Spectrogram 을 시각화하였다.



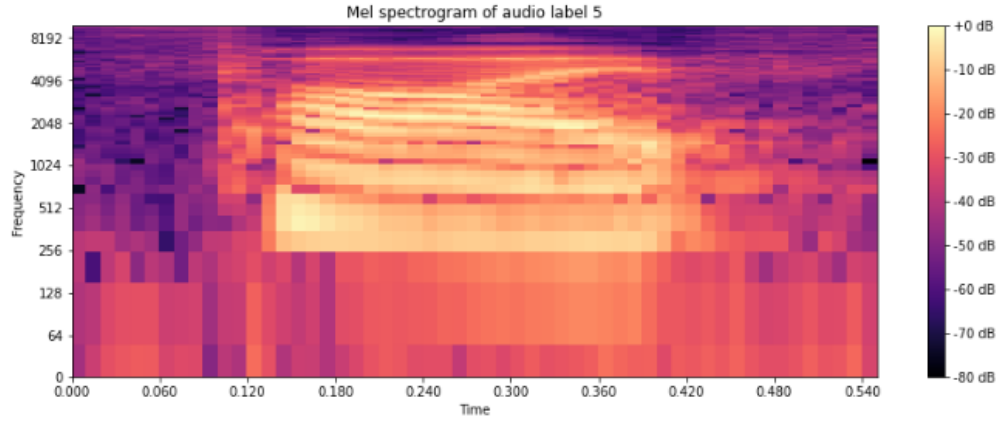
Wav length: 0.66165, Mel_S shape:(100, 67)



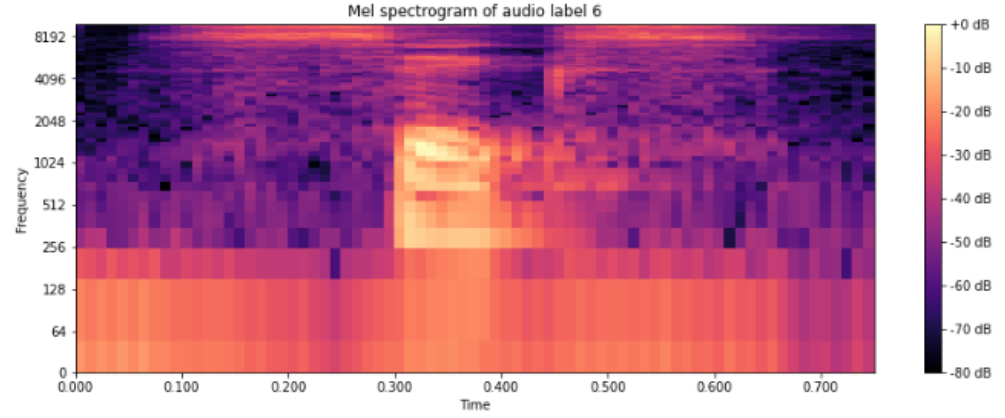
Wav length: 0.62555, Mel_S shape:(100, 63)

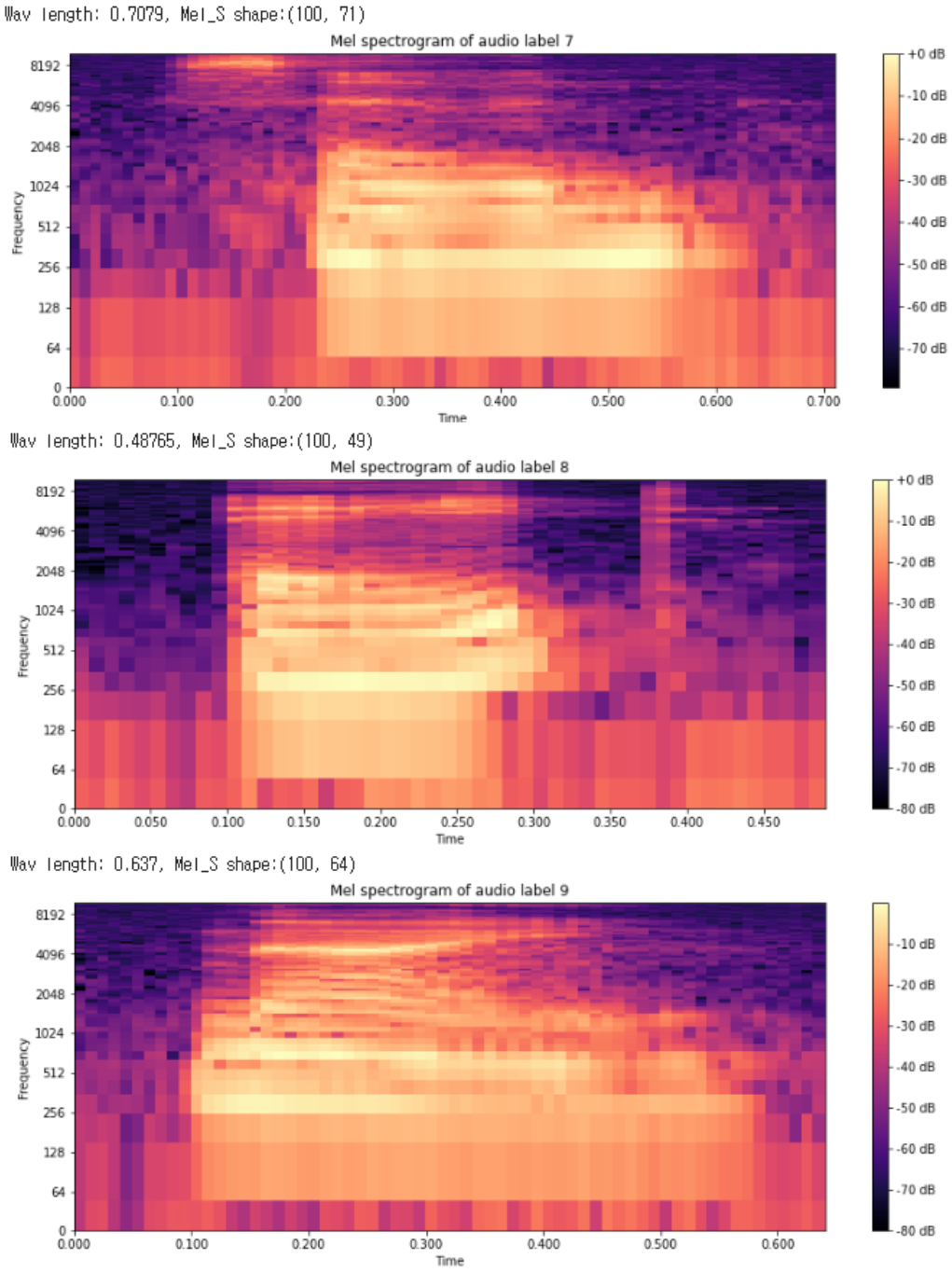


Wav length: 0.54015, Mel_S shape:(100, 55)



Wav length: 0.7418, Mel_S shape:(100, 75)



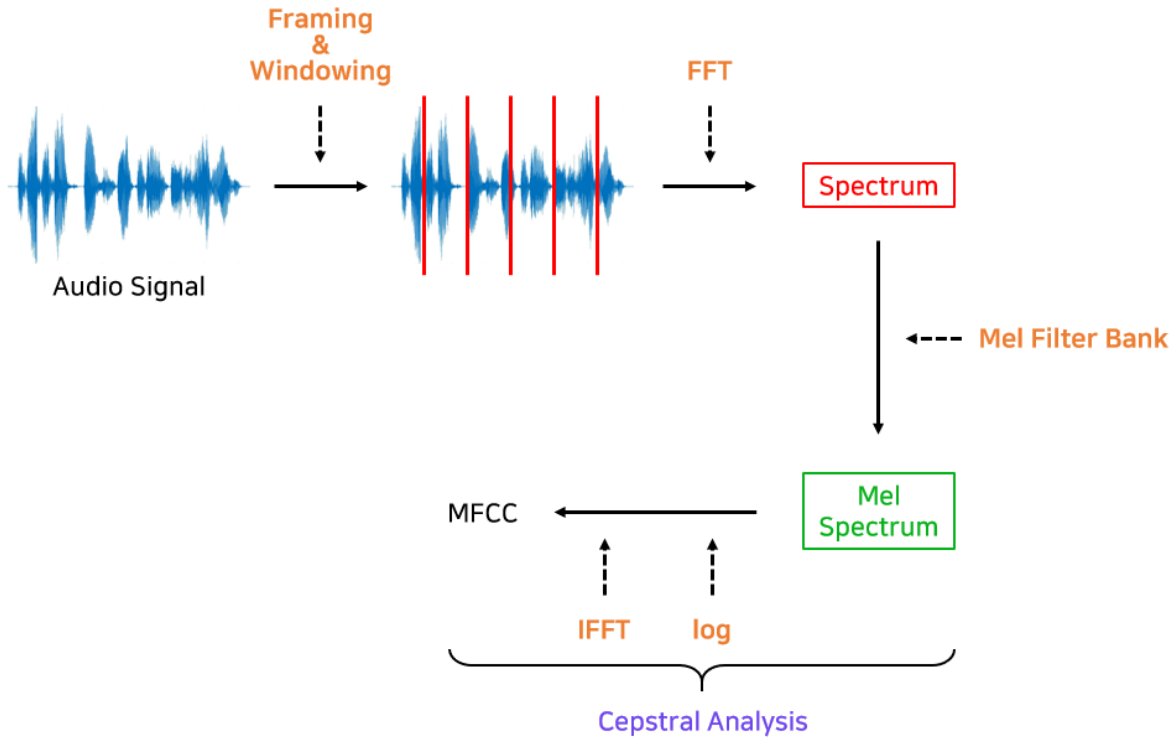


3.2.3. MFCC(Mel-Frequency Cepstral Coefficient)

오디오 데이터 전체에 퓨리에 변환을 수행한다면 '안녕하세요'와 같은 하나의 오디오 데이터에 대해 사람마다 지속 시간이 달라질 수 있다. 따라서 이러한 다양한 시간에 대해 같은 오디오 데이터를 학습시키기가 어렵다. 이와 같은 문제를 해결하기 위하여 MFCC 알고리즘을 이용한다.

MFCC 는 오디오 데이터를 특징벡터화하는 알고리즘이다. 입력된 소리 전체를 대상으로 하는 것이 아니라, 사람이 인지하기 좋은 Mel-Scale 로 오디오 데이터를 모두 20~40ms 으로 나눈 후 이

구간에 대한 스펙트럼을 분석하여 퓨리에 변환을 한 특징 추출 방법이다. 이 때, 사람의 오디오는 20~40ms 사이에서는 음소(현재 내고 있는 발음)가 바뀔 수 없다는 연구 결과들을 기반으로 음소는 해당 시간 내에 바뀔 수 없다고 가정한다고 한다. 따라서 MFCC 에서는 오디오 데이터를 모두 20~40ms 단위로 쪼갠 후 쪼갠 단위에 대해 Mel 값을 추출하여 Feature 로 사용한다. 이를 수행하고자 librosa.feature.mfcc 메소드를 이용할 수 있으며 해당 메소드의 몇몇 파라미터 중, n_mfcc 는 반환될 mfcc 의 개수를 정해주는 역할을 한다. 더 다양한 데이터의 특징을 추출하기 위해서는 더 큰 값을 지정한다.



MFCC 추출 과정을 간략하게 나타낸 도표는 위와 같으며 MFCC 를 이용하여 첫번째 오디오 데이터의 Feature 를 추출한 결과는 다음과 같다.

```
def extract_features(file):
    audio, sample_rate = librosa.load(file, sr = 20000)
    extracted_features = librosa.feature.mfcc(y=audio,
                                              sr=sample_rate,
                                              n_mfcc=40)

    extracted_features = np.mean(extracted_features.T,axis=0)
    return extracted_features
```

```
extract_features('train/001.wav')
```

```
array([-5.55574951e+02,  1.07603752e+02, -6.78596115e+00,  3.71658058e+01,
        1.97651196e+01,  7.19014883e+00,  9.94960606e-01, -1.09570913e+01,
       -3.01008511e+00,  3.47170544e+00, -1.53333163e+00, -2.76586843e+00,
       -7.03771830e+00, -2.01748013e+00,  8.78361225e+00, -7.18724298e+00,
       -8.45485878e+00,  3.20041347e+00, -6.64840746e+00, -6.60384536e-01,
       -7.78916311e+00, -8.09230518e+00, -7.82652950e+00,  3.41185117e+00,
       -8.96139050e+00,  2.65029597e+00, -6.52310801e+00,  2.83142519e+00,
       -3.19256163e+00,  5.09376240e+00,  5.65017760e-01,  3.17868686e+00,
        9.28390741e-01,  3.76043701e+00, -4.59410000e+00,  1.10403955e-01,
       -3.12741399e+00,  4.81157191e-02, -8.49821186e+00,  6.84093963e-03],
      dtype=float32)
```

3.2.4. Mel Spectrogram, MFCC 비교

Mel Spectrogram 의 경우 주파수끼리 correlate 하기 때문에 도메인이 한정적인 문제에서 더 좋은 성능을 보이고, MFCC 의 경우 이를 de-correlate 해주기 때문에 일반적인 상황에서 더 좋은 성능을 보인다고 한다.

Mel Spectrogram 과 MFCC 를 비교하기 위해 각 오디오 데이터를 어떻게 표현하는지를 시각화하기 위해 다음과 같이 데이터를 변환하였다.

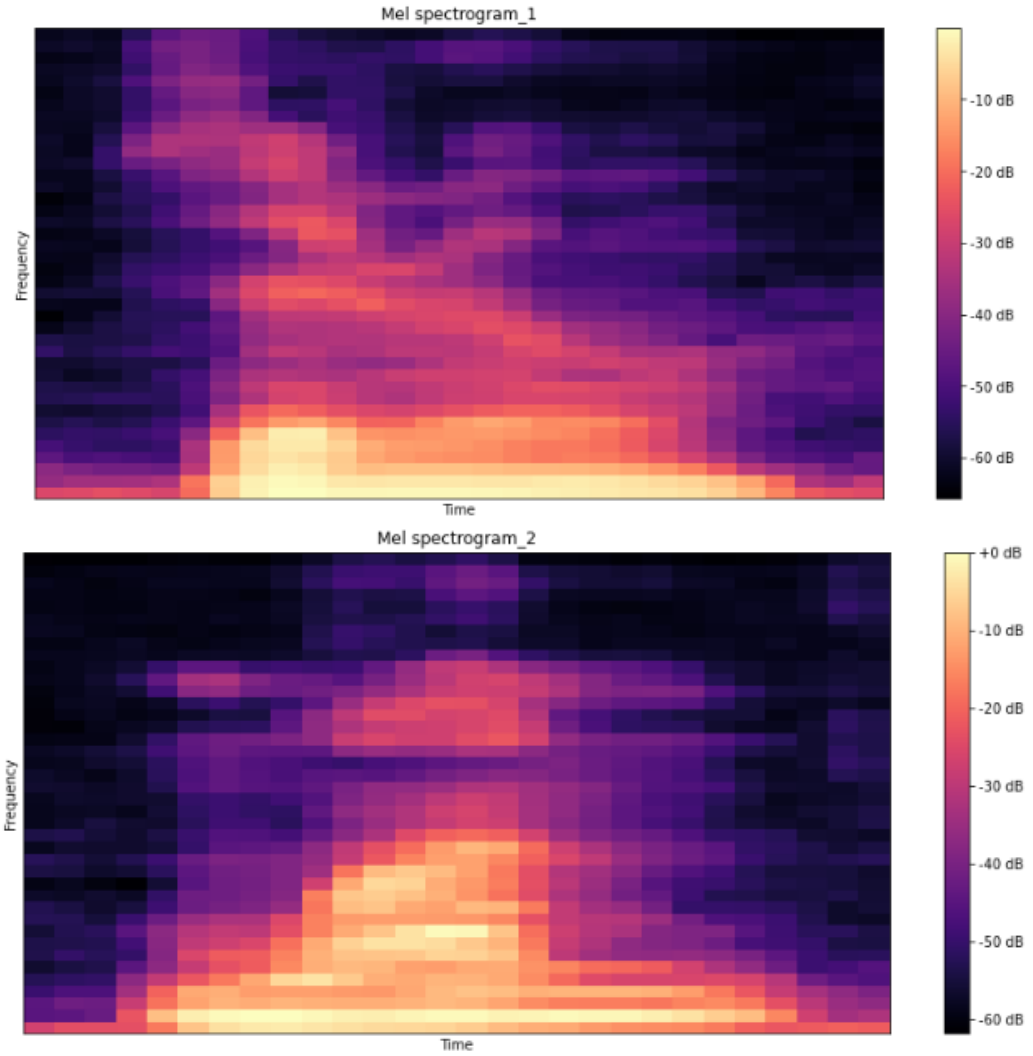
```
from tqdm.notebook import tqdm

train_file_names = train["file_name"].to_numpy()
test_file_names = test["file_name"].to_numpy()
target = train["label"].to_numpy()

def load_audio(file_names, target, path):
  audios = []
  for audio in tqdm(file_names):
    # librosa를 이용하여 데이터 로드
    an_audio, _ = librosa.load(path+audio, sr=20000)
    audio_array = np.array(an_audio)
    audios.append(audio_array)
  audios = np.array(audios)

  targets = target.copy()

  return audios, targets
```

```
audio_train, target_train = load_audio(train_file_names, target, path='./train/')
audio_test, _ = load_audio(test_file_names, np.array([None]), path='./test/')
```

```
100% 400/400 [00:28<00:00, 23.09it/s]
100% 200/200 [01:17<00:00, 2.91it/s]
```

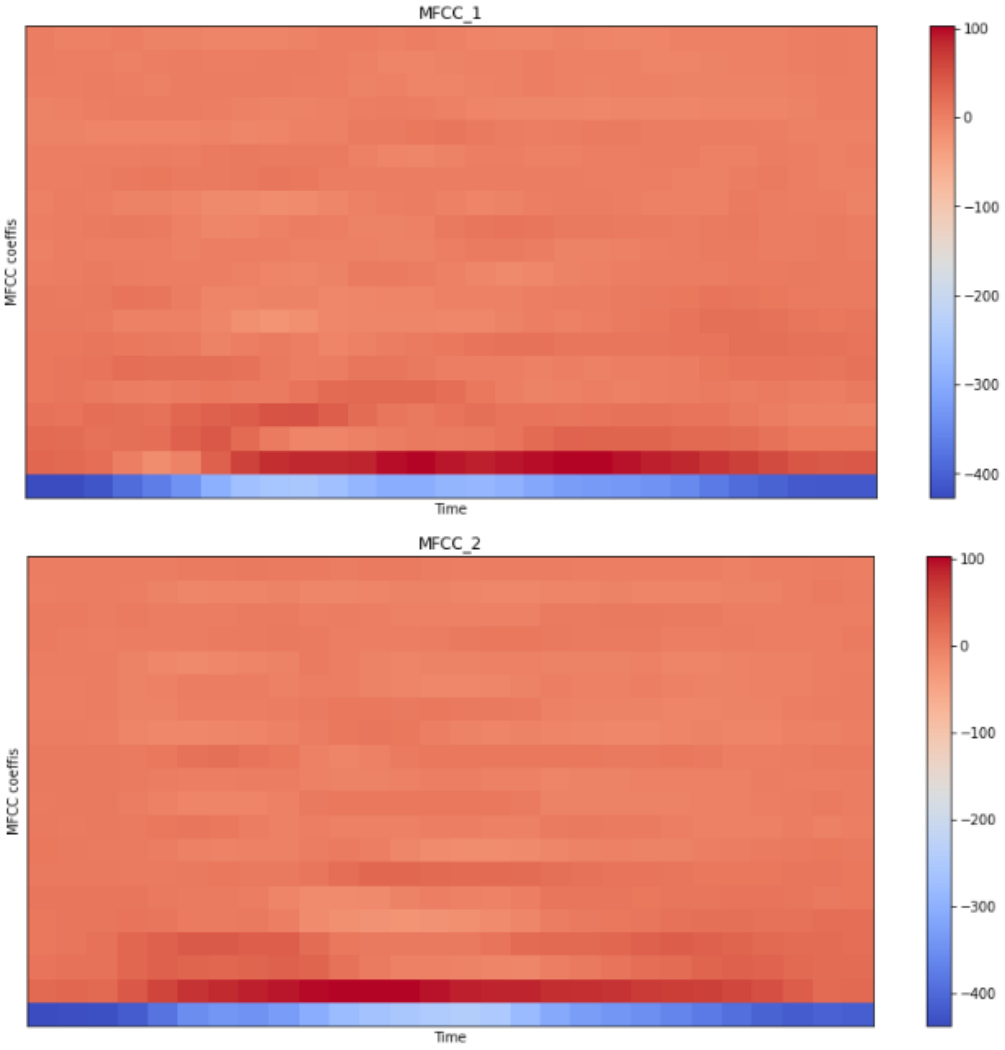
우선 mel spectrogram 이 서로 다른 데이터를 어떻게 구분하는 지를 표현하였다.

```
#mels끼리 비교

plt.figure(figsize=(14,14))
ax = plt.subplot(2, 1, 1)


ax = plt.subplot(2, 1, 1)
S = librosa.feature.melspectrogram(audio_train[15], sr=20000, n_mels=40)
log_S = librosa.power_to_db(S, ref=np.max)
librosa.display.specshow(log_S, sr=20000)
plt.title('Mel spectrogram_1')
plt.xlabel("Time")
plt.ylabel("Frequency")
plt.colorbar(format='%+2.0f dB')

ax = plt.subplot(2, 1, 2)
S = librosa.feature.melspectrogram(audio_train[16], sr=20000, n_mels=40)
log_S = librosa.power_to_db(S, ref=np.max)
librosa.display.specshow(log_S, sr=20000)
plt.title('Mel spectrogram_2')
plt.xlabel("Time")
plt.ylabel("Frequency")
plt.colorbar(format='%+2.0f dB')
```

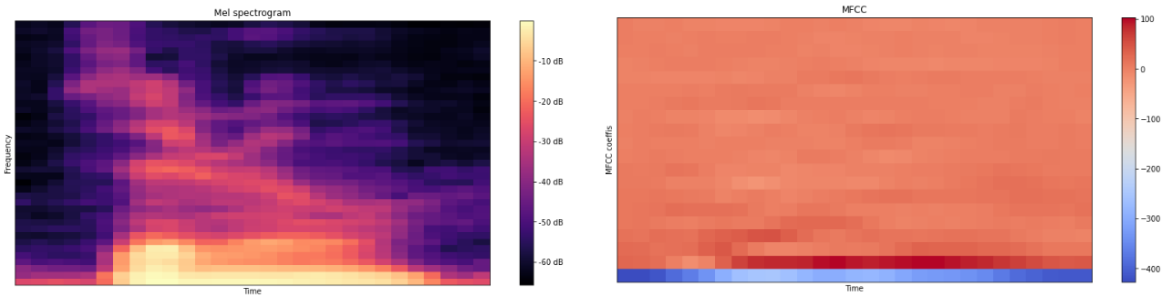


이어서 동일한 데이터에 대해 mfcc 가 수행한 결과도 표현하였다.

```
#mfcc끼리 비교

plt.figure(figsize=(14,14))
ax = plt.subplot(2, 1, 1)


ax = plt.subplot(2, 1, 1)
mfcc = librosa.feature.mfcc(audio_train[15], sr=20000, n_mels=40)
librosa.display.specshow(mfcc, sr=20000)
plt.title('MFCC_1')
plt.xlabel("Time")
plt.ylabel("MFCC coeffis")
plt.colorbar()

ax = plt.subplot(2, 1, 2)
mfcc = librosa.feature.mfcc(audio_train[16], sr=20000, n_mels=40)
librosa.display.specshow(mfcc, sr=20000)
plt.title('MFCC_2')
plt.xlabel("Time")
plt.ylabel("MFCC coeffis")
plt.colorbar()
```



조금 더 쉬운 비교를 위해 동일한 데이터에 대해 mel spectrogram(왼쪽)과 mfcc(오른쪽)를 동시에 시각화하였다.



이와 같은 시각화 결과를 통해서는, Mel Spectrogram 이 데이터를 더 잘 구분하는 것으로 판단된다. 실제 결과는 모델에서 확인해보고자 한다.

3.3. Data Augmentation

데이터 EDA 단계에서 살펴본 바와 같이, 오디오 데이터들의 길이가 다 다르다. 그래서 이에 따라 mel Spectrogram 과 mfcc 역시 모두 길이가 제각각인 결론을 반환한다.

CNN 에 학습시킬 input 데이터로의 전처리를 진행하고 있으므로, translation variance(input 의 위치가 달라져도 output 이 동일한 값을 갖는다는 개념)에 따라, 여유로운 크기의 고정 사이즈를 정하고 고정 사이즈에 맞추어 랜덤으로 앞이나 뒤에 padding 을 실시하고자 하였다. 각 파일마다 데시벨의 차이가 있을 것으로 예상하여 min_max scaling 을 수행하였다.

```
def random_pad(mels, pad_size, mfcc=True):

  pad_width = pad_size - mels.shape[1]
  rand = np.random.rand()
  left = int(pad_width * rand)
  right = pad_width - left

  if mfcc:
    mels = np.pad(mels, pad_width=((0,0), (left, right)), mode='constant')
    local_max, local_min = mels.max(), mels.min()
    mels = (mels - local_min)/(local_max - local_min)
  else:
    local_max, local_min = mels.max(), mels.min()
    mels = (mels - local_min)/(local_max - local_min)
    mels = np.pad(mels, pad_width=((0,0), (left, right)), mode='constant')

  return mels
```

위 함수를 통해 padding 을 수행하고자 하였다. 길이가 가장 긴 오디오 파일의 정보가 잘리지 않을 정도의 pad_size 를 생성하였다. 즉 pad_size 는 sampling rate 에 따라 달라질 것이다. 또한 melspectrogram 과 mfcc 의 return 값이 정사각형이 될 수 있도록 n=40 으로 설정하였다. Return 값이 커질수록 feature 값이 많아지는 것이므로 성능에 영향을 미칠 것으로 예상하였다.

```
size = 40
pad_size = 40
repeat_size = 5
```

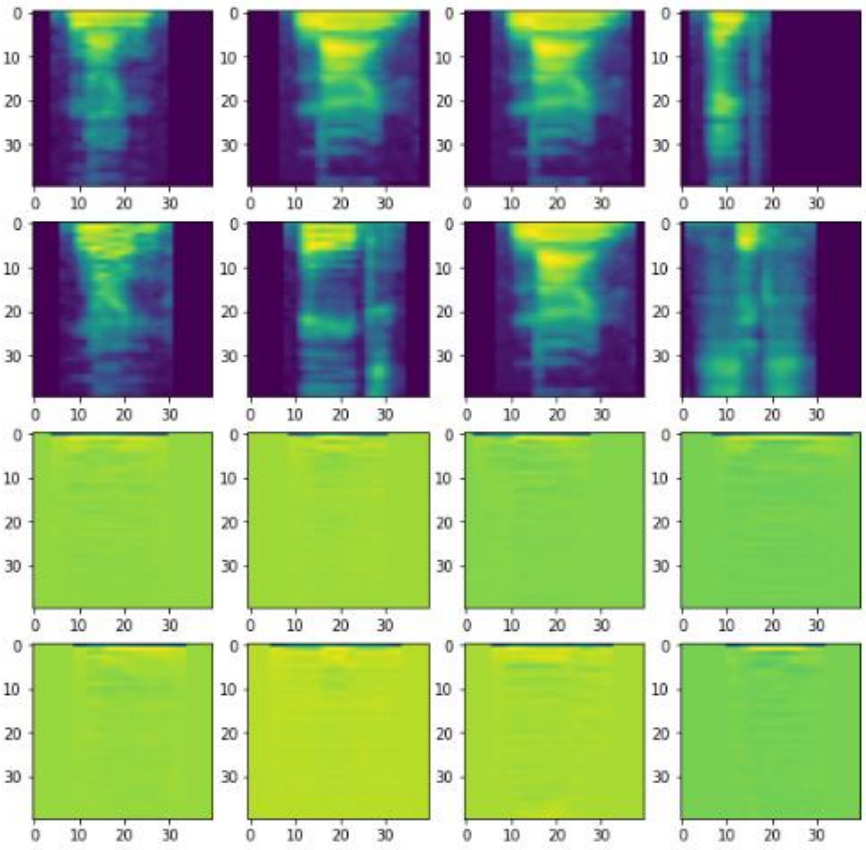
Data 의 augmentation 이 일어날 것으로 예상하여 하나의 데이터당 padding 을 5 번 수행하였다. 아래의 코드들을 통해 melspectrogram 과 mfcc 데이터셋이 생성되었다. 이 때 증강한 데이터만큼 target 값도 복제했다.

```
audio_mels = []
audio_mfcc = []

for y in audio_train:
  mels = librosa.feature.melspectrogram(y, sr=20000, n_mels=size)
  mels = librosa.power_to_db(mels, ref=np.max)

  mfcc = librosa.feature.mfcc(y, sr=20000, n_mfcc=size)

  for i in range(repeat_size):
    audio_mels.append(random_pad(mels, pad_size=pad_size, mfcc=False))
    audio_mfcc.append(random_pad(mfcc, pad_size=pad_size, mfcc=True))

audio_mels_array_test = []
audio_mfcc_array_test = []

for y in audio_test:
  mels = librosa.feature.melspectrogram(y, sr=20000, n_mels=size)
  mels = librosa.power_to_db(mels, ref=np.max)

  mfcc = librosa.feature.mfcc(y, sr=20000, n_mfcc=size)

  audio_mels_array_test.append(random_pad(mels, pad_size=pad_size, mfcc=False))
  audio_mfcc_array_test.append(random_pad(mfcc, pad_size=pad_size, mfcc=True))
```

3.4. 전처리 결과 확인

전처리 단계를 거친 결과는 다음 코드를 통해 시각화하였다.

```
target_num = 2
target_num_idx = np.where(target_train==target_num)[0]
target_num_idx = np.random.choice(target_num_idx,16)

plt.figure(figsize=(10, 10))
ax = plt.subplot(4, 4, 1)

for i, idx in enumerate(target_num_idx):
  ax = plt.subplot(4, 4, i + 1)
  if i<8:
    plt.imshow(audio_mels_array[idx], aspect='auto')
  else:
    plt.imshow(audio_mfcc_array[idx], aspect='auto')
```



위 그래프의 위 두 행은 mel spectrogram 으로, 아래 두 행은 mfcc 로 생성한 이미지이다.

4. 모델 구성(방법론)

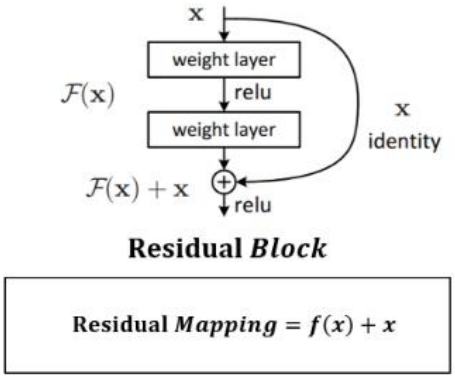
4.1. 모델링

전처리를 마친 데이터를 딥러닝 알고리즘 중 하나인 Convolution Neural Networks (CNN)을 사용하여 해당 오디오 데이터에서 말하고 있는 숫자가 무엇인지 분류했다. CNN 은 합성곱 연산을 적용하여 신경망을 연산하는 것으로, 이미지 및 자연어 해석, 영상 분류 등 다양한 분야에 활용되고 있으며 특히 오디오 분석 기법에서 딥러닝 기술이 접목되면서 과거에 비해 큰 기술 향상을 이루어 낸다는 연구가 보고된 바 있다.[1] 따라서 본 프로젝트에서는 숫자 오디오 분류 모델을 구축하는 데에 CNN 알고리즘을 적용할 것이다. CNN Layer 를 쌓기에 앞서, CNN 관련 이론을 학습하여 모델링 아이디어를 얻었다.

[1] (2021) CNN 알고리즘을 활용한 음성신호 중 비음성 구간 탐지 모델 연구

4.1.1. ResBlock(Residual Block Idea)

Residual Block 은 CNN 에서 레이어가 깊어질 때 발생하는 Vanishing Gradient 문제를 해결하기 위해 착안된 개념으로, 학습한 function 에 input 값을 더해주며 수행된다. 다음 그림에서 Residual Block 의 형태를 확인할 수 있다.



즉 Residual Block 을 통해 input 값인 identity 를 그대로 가져와 weight layer 를 통과한 F(x)에 identity, x 를 더해 Residual Mapping 을 해주어 layer 가 깊어질수록 identity 가 소실되어 feature update 가 안되는 문제를 해결한 것이다. 그리고 이러한 Residual Block 을 쌓는 구조를 ResNet (Residual Network)라고 하며, ResNet 의 등장으로 CNN 의 성능이 크게 향상되어 현재까지도 주목받고 있다고 한다.[2] 특히 ResNet 의 ResBlock 은 오디오 분야에서 많이 사용된다고 하여 이를 응용하여 다음과 같이 모델 구축에 사용하였다.

```
def residual_block(x, filters_in, filters_out):
    shortcut = x
    x = BatchNormalization()(x)
    x = ReLU()(x)
    x = Conv2D(filters_in, kernel_size=(1, 1), strides=(1, 1), padding="same",kernel_initializer='he_normal')(x)

    x = BatchNormalization()(x)
    x = ReLU()(x)
    x = Conv2D(filters_in, kernel_size=(3, 3), strides=(1, 1), padding="same",kernel_initializer='he_normal')(x)

    x = BatchNormalization()(x)
    x = ReLU()(x)
    x = Conv2D(filters_out, kernel_size=(1, 1), strides=(1, 1), padding="same",kernel_initializer='he_normal')(x)

    shortcut_channel = x.shape.as_list()[0]

    if shortcut_channel != filters_out:
        shortcut = Conv2D(filters_out, kernel_size=(1, 1), strides=(1, 1), padding="same",kernel_initializer='he_normal')(shortcut)

    x = Add()([x, shortcut])
    return ReLU()(x)
```

[2] (2015) Deep Residual Learning for Image Recognition

4.1.2. Model Building : CNN Layer

Conv2D, MaxPooling2D, Dropout, Dense Layer 를 쌓고 BatchNormalization 을 수행하여 CNN Layer 를 구성하였으며, 앞서 정의한 residual_block() 함수를 사용하여 x 값을 더해주는 연산을 수행하였다. 다음은 쌓은 layer 들의 구체적인 순서와 filter 크기를 포함한, 모델을 구축하는 함수이다.

```
def build_model():

  inputs = tf.keras.layers.Input(shape=(size,pad_size,1))

  outputs = Conv2D(16,(3,3),activation=None,padding='same',kernel_initializer='he_normal')(inputs)
  outputs = BatchNormalization()(outputs)
  outputs = ReLU()(outputs)
  outputs = MaxPool2D((2,2))(outputs)

  outputs = residual_block(outputs, 16, 32)
  outputs = MaxPool2D((2,2))(outputs)
  outputs = residual_block(outputs, 32, 32)
  outputs = residual_block(outputs, 32, 64)
  outputs = MaxPool2D((2,2))(outputs)
  outputs = residual_block(outputs, 64, 64)

  outputs = GlobalAveragePooling2D()(outputs)

  outputs = Dense(32,activation=None,kernel_initializer='he_normal')(outputs)
  outputs = BatchNormalization()(outputs)
  outputs = ReLU()(outputs)
  outputs = Dropout(0.5)(outputs)

  outputs = Dense(10,activation='softmax')(outputs)
  model = Model(inputs=inputs, outputs=outputs)
  model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

  return model
```

이를 통해 생성한 모델의 summary 결과는 다음과 같다.

```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 40, 40, 1)]  0           []

conv2d (Conv2D)                 (None, 40, 40, 16)   160         ['input_1[0][0]']

batch_normalization (BatchNorm  (None, 40, 40, 16)  64          ['conv2d[0][0]']
alization)

re_lu (ReLU)                    (None, 40, 40, 16)   0           ['batch_normalization[0][0]']

max_pooling2d (MaxPooling2D)    (None, 20, 20, 16)   0           ['re_lu[0][0]']

batch_normalization_1 (BatchNo  (None, 20, 20, 16)  64          ['max_pooling2d[0][0]']
rmalization)

re_lu_1 (ReLU)                  (None, 20, 20, 16)   0           ['batch_normalization_1[0][0]']
```

```
conv2d_1 (Conv2D)              (None, 20, 20, 16)   272         ['re_lu_1[0][0]']

batch_normalization_2 (BatchNo  (None, 20, 20, 16)  64          ['conv2d_1[0][0]']
rmalization)

re_lu_2 (ReLU)                 (None, 20, 20, 16)   0           ['batch_normalization_2[0][0]']

conv2d_2 (Conv2D)              (None, 20, 20, 16)   2320        ['re_lu_2[0][0]']

batch_normalization_3 (BatchNo  (None, 20, 20, 16)  64          ['conv2d_2[0][0]']
rmalization)

re_lu_3 (ReLU)                 (None, 20, 20, 16)   0           ['batch_normalization_3[0][0]']

conv2d_3 (Conv2D)              (None, 20, 20, 32)   544         ['re_lu_3[0][0]']

conv2d_4 (Conv2D)              (None, 20, 20, 32)   544         ['max_pooling2d[0][0]']

add (Add)                      (None, 20, 20, 32)   0           ['conv2d_3[0][0]',
                                                                 'conv2d_4[0][0]']

re_lu_4 (ReLU)                 (None, 20, 20, 32)   0           ['add[0][0]']

max_pooling2d_1 (MaxPooling2D)  (None, 10, 10, 32)  0           ['re_lu_4[0][0]']

batch_normalization_4 (BatchNo  (None, 10, 10, 32)  128         ['max_pooling2d_1[0][0]']
rmalization)
re_lu_5 (ReLU)                 (None, 10, 10, 32)   0           ['batch_normalization_4[0][0]']

conv2d_5 (Conv2D)              (None, 10, 10, 32)   1056        ['re_lu_5[0][0]']

batch_normalization_5 (BatchNo  (None, 10, 10, 32)  128         ['conv2d_5[0][0]']
rmalization)

re_lu_6 (ReLU)                 (None, 10, 10, 32)   0           ['batch_normalization_5[0][0]']

conv2d_6 (Conv2D)              (None, 10, 10, 32)   9248        ['re_lu_6[0][0]']

batch_normalization_6 (BatchNo  (None, 10, 10, 32)  128         ['conv2d_6[0][0]']
rmalization)

re_lu_7 (ReLU)                 (None, 10, 10, 32)   0           ['batch_normalization_6[0][0]']

conv2d_7 (Conv2D)              (None, 10, 10, 32)   1056        ['re_lu_7[0][0]']

conv2d_8 (Conv2D)              (None, 10, 10, 32)   1056        ['max_pooling2d_1[0][0]']

add_1 (Add)                    (None, 10, 10, 32)   0           ['conv2d_7[0][0]',
                                                                 'conv2d_8[0][0]']

re_lu_8 (ReLU)                 (None, 10, 10, 32)   0           ['add_1[0][0]']
batch_normalization_7 (BatchNo  (None, 10, 10, 32)  128         ['re_lu_8[0][0]']
rmalization)

re_lu_9 (ReLU)                 (None, 10, 10, 32)   0           ['batch_normalization_7[0][0]']

conv2d_9 (Conv2D)              (None, 10, 10, 32)   1056        ['re_lu_9[0][0]']

batch_normalization_8 (BatchNo  (None, 10, 10, 32)  128         ['conv2d_9[0][0]']
rmalization)

re_lu_10 (ReLU)                (None, 10, 10, 32)   0           ['batch_normalization_8[0][0]']

conv2d_10 (Conv2D)             (None, 10, 10, 32)   9248        ['re_lu_10[0][0]']

batch_normalization_9 (BatchNo  (None, 10, 10, 32)  128         ['conv2d_10[0][0]']
rmalization)
```

```
re_lu_11 (ReLU)                 (None, 10, 10, 32)   0           ['batch_normalization_9[0][0]']

conv2d_11 (Conv2D)              (None, 10, 10, 64)   2112        ['re_lu_11[0][0]']

conv2d_12 (Conv2D)              (None, 10, 10, 64)   2112        ['re_lu_8[0][0]']

add_2 (Add)                     (None, 10, 10, 64)   0           ['conv2d_11[0][0]',
                                                                 'conv2d_12[0][0]']
re_lu_12 (ReLU)                 (None, 10, 10, 64)   0           ['add_2[0][0]']

max_pooling2d_2 (MaxPooling2D)  (None, 5, 5, 64)     0           ['re_lu_12[0][0]']

batch_normalization_10 (BatchN  (None, 5, 5, 64)     256         ['max_pooling2d_2[0][0]']
ormalization)

re_lu_13 (ReLU)                 (None, 5, 5, 64)     0           ['batch_normalization_10[0][0]']

conv2d_13 (Conv2D)              (None, 5, 5, 64)     4160        ['re_lu_13[0][0]']

batch_normalization_11 (BatchN  (None, 5, 5, 64)     256         ['conv2d_13[0][0]']
ormalization)

re_lu_14 (ReLU)                 (None, 5, 5, 64)     0           ['batch_normalization_11[0][0]']

conv2d_14 (Conv2D)              (None, 5, 5, 64)     36928       ['re_lu_14[0][0]']

batch_normalization_12 (BatchN  (None, 5, 5, 64)     256         ['conv2d_14[0][0]']
ormalization)

re_lu_15 (ReLU)                 (None, 5, 5, 64)     0           ['batch_normalization_12[0][0]']

conv2d_15 (Conv2D)              (None, 5, 5, 64)     4160        ['re_lu_15[0][0]']

conv2d_16 (Conv2D)              (None, 5, 5, 64)     4160        ['max_pooling2d_2[0][0]']

add_3 (Add)                     (None, 5, 5, 64)     0           ['conv2d_15[0][0]',
                                                                 'conv2d_16[0][0]']
re_lu_16 (ReLU)                 (None, 5, 5, 64)     0           ['add_3[0][0]']

global_average_pooling2d (Glob  (None, 64)           0           ['re_lu_16[0][0]']
alAveragePooling2D)

dense (Dense)                   (None, 32)           2080        ['global_average_pooling2d[0][0]'
                                                                 ]
batch_normalization_13 (BatchN  (None, 32)           128         ['dense[0][0]']
ormalization)

re_lu_17 (ReLU)                 (None, 32)           0           ['batch_normalization_13[0][0]']

 dropout (Dropout)               (None, 32)           0          ['re_lu_17[0][0]']

 dense_1 (Dense)                 (None, 10)           330        ['dropout[0][0]']

==================================================================================================
Total params: 84,522
Trainable params: 83,562
Non-trainable params: 960
```

위와 같이 CNN 모델을 구축하는 데에 성공했다.

4.2. 학습

모델을 학습시키기 위해 5 fold StratifiedKFold 를 수행하였다. StratifiedKFold 는 target 에 속성 값의 수를 동일하게 하여, 일반 kfold 를 수행할 때 종종 발생하는 데이터가 한 곳으로 몰리는 현상을 방지할 수 있는 교차 검증 기법이다. 즉 label 값의 분포를 반영하지 못하는 문제를 해결하기 위해 사용하는 방법 중 하나이다.

이를 구현한 코드는 아래와 같으며, monitor = 'val_loss'의 결과로 나오는 결과창도 아래에 이어서 첨부하였다.

```
acc_list = []
pred_list = []
skf = StratifiedKFold(n_splits=5)
history_mel_list = []
history_mfcc_list = []

for fold,(train_index, val_index) in enumerate(skf.split(audio_mels_array, repeated_target)):

  print(f'₩n********** {fold+1} fold **********')

  preds_val_list = []
  ### melspectrogram ###
  model = build_model()
  x_train, x_val = audio_mels_array[train_index], audio_mels_array[val_index]
  y_train, y_val = repeated_target[train_index], repeated_target[val_index]
  filepath = f"model.res_test_0615_mels_{fold}.hdf5"
  callbacks = [tf.keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss',
                                                  verbose=0, save_best_only=True, mode='min')]
  history = model.fit(x_train, y_train, batch_size=32, epochs=50,
                      validation_data=(x_val,y_val), callbacks=callbacks, verbose=1)
  history_mel_list.append(history)
  model = load_model(filepath)
  preds_val = model.predict(x_val)
  preds_val_list.append(preds_val)
  preds_val_label = np.argmax(preds_val, axis=1)
  pred_list.append(model.predict(audio_mels_array_test))
  print(f'mels_model_acc : {accuracy_score(y_val,preds_val_label):.4f}')

  ### mfcc ###
  model = build_model()
  x_train, x_val = audio_mfcc_array[train_index], audio_mfcc_array[val_index]
  y_train, y_val = repeated_target[train_index], repeated_target[val_index]
  filepath = f"model.res_test_0615_mfcc_{fold}.hdf5"
  callbacks = [tf.keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss',
                                                  verbose=0, save_best_only=True, mode='min')]
  history = model.fit(x_train, y_train, batch_size=32, epochs=50,
                      validation_data=(x_val,y_val), callbacks=callbacks, verbose=1)
history_mfcc_list.append(history)
model = load_model(filepath)
preds_val = model.predict(x_val)
preds_val_list.append(preds_val)
preds_val_label = np.argmax(preds_val, axis=1)
pred_list.append(model.predict(audio_mfcc_array_test))
print(f'mfcc_model_acc : {accuracy_score(y_val,preds_val_label):.4f}')
```

```
  ### ensemble ###
  val_pred_result = preds_val_list[0].copy()
  for i in range(1, len(preds_val_list)): # 추후 추가될 가능성이 있는 model을 위해 for문 생성함
      val_pred_result += preds_val_list[i]
  val_pred_label = np.argmax(val_pred_result, axis=1)
  en_acc = accuracy_score(y_val,val_pred_label)
  acc_list.append(en_acc)
  print(f'ensemble_model_acc : {en_acc:.4f}')

print(f'₩n₩nmean_acc : {np.mean(acc_list):.4f}')
```

```
********** 1 fold **********
Epoch 1/50
50/50 [==============================] - 11s 154ms/step - loss: 2.1102 - accuracy: 0.2937 - val_loss: 2.3389 - val_accuracy: 0.1150
Epoch 2/50
50/50 [==============================] - 5s 108ms/step - loss: 1.4396 - accuracy: 0.5238 - val_loss: 2.7510 - val_accuracy: 0.1025
Epoch 3/50
50/50 [==============================] - 5s 107ms/step - loss: 1.0966 - accuracy: 0.6725 - val_loss: 3.2328 - val_accuracy: 0.1175
Epoch 4/50
50/50 [==============================] - 5s 106ms/step - loss: 0.8233 - accuracy: 0.7837 - val_loss: 3.3784 - val_accuracy: 0.1100
Epoch 5/50
50/50 [==============================] - 5s 108ms/step - loss: 0.6300 - accuracy: 0.8619 - val_loss: 3.4111 - val_accuracy: 0.2550
Epoch 6/50
50/50 [==============================] - 6s 112ms/step - loss: 0.5074 - accuracy: 0.8925 - val_loss: 2.9134 - val_accuracy: 0.2350
Epoch 7/50
50/50 [==============================] - 7s 147ms/step - loss: 0.4082 - accuracy: 0.9250 - val_loss: 1.7802 - val_accuracy: 0.3425
Epoch 8/50
50/50 [==============================] - 5s 108ms/step - loss: 0.3333 - accuracy: 0.9394 - val_loss: 1.8801 - val_accuracy: 0.4400
Epoch 9/50
50/50 [==============================] - 6s 115ms/step - loss: 0.2839 - accuracy: 0.9500 - val_loss: 0.9081 - val_accuracy: 0.6975
Epoch 10/50
50/50 [==============================] - 6s 115ms/step - loss: 0.2467 - accuracy: 0.9581 - val_loss: 0.7382 - val_accuracy: 0.7475
Epoch 11/50
50/50 [==============================] - 6s 115ms/step - loss: 0.2089 - accuracy: 0.9631 - val_loss: 0.4822 - val_accuracy: 0.8250
Epoch 12/50
50/50 [==============================] - 6s 116ms/step - loss: 0.1803 - accuracy: 0.9719 - val_loss: 0.3922 - val_accuracy: 0.8925
Epoch 13/50
50/50 [==============================] - 7s 141ms/step - loss: 0.1712 - accuracy: 0.9688 - val_loss: 0.3431 - val_accuracy: 0.9150
Epoch 14/50
50/50 [==============================] - 6s 123ms/step - loss: 0.1556 - accuracy: 0.9737 - val_loss: 0.3715 - val_accuracy: 0.8850
Epoch 15/50
50/50 [==============================] - 7s 137ms/step - loss: 0.1446 - accuracy: 0.9744 - val_loss: 0.3006 - val_accuracy: 0.9025
Epoch 16/50
50/50 [==============================] - 6s 112ms/step - loss: 0.1387 - accuracy: 0.9694 - val_loss: 0.4117 - val_accuracy: 0.8600
Epoch 17/50
50/50 [==============================] - 6s 111ms/step - loss: 0.1507 - accuracy: 0.9663 - val_loss: 0.6104 - val_accuracy: 0.8075
Epoch 18/50
50/50 [==============================] - 6s 111ms/step - loss: 0.1243 - accuracy: 0.9775 - val_loss: 0.3711 - val_accuracy: 0.8925
Epoch 19/50
50/50 [==============================] - 6s 111ms/step - loss: 0.1123 - accuracy: 0.9775 - val_loss: 0.5694 - val_accuracy: 0.8225
Epoch 20/50
50/50 [==============================] - 6s 120ms/step - loss: 0.1154 - accuracy: 0.9719 - val_loss: 0.2608 - val_accuracy: 0.9225
Epoch 21/50
50/50 [==============================] - 6s 117ms/step - loss: 0.0925 - accuracy: 0.9812 - val_loss: 0.2355 - val_accuracy: 0.9250
Epoch 22/50
50/50 [==============================] - 5s 109ms/step - loss: 0.0997 - accuracy: 0.9775 - val_loss: 0.2619 - val_accuracy: 0.9100
Epoch 23/50
50/50 [==============================] - 6s 111ms/step - loss: 0.0975 - accuracy: 0.9781 - val_loss: 0.2896 - val_accuracy: 0.9125
Epoch 24/50
50/50 [==============================] - 6s 111ms/step - loss: 0.0916 - accuracy: 0.9775 - val_loss: 0.3533 - val_accuracy: 0.8875
Epoch 25/50
50/50 [==============================] - 5s 110ms/step - loss: 0.0939 - accuracy: 0.9762 - val_loss: 0.3046 - val_accuracy: 0.8975
Epoch 26/50
```

총 5 개의 fold 에 대해 위와 같이 loss 값과 accuracy 값을 같이 출력하였다.

다음 두 출력 결과에서 볼 수 있듯이 각 fold 마다 mel spectrogram 을 학습시켰을 때의 평균 accuracy 와 mfcc 를 학습시켰을 때의 평균 accuracy, 이들을 활용한 앙상블 모델의 결과값이 출력되었다. 한 데이터에 대해 두 모델이 각각 예측한 확률을 더한 값들 중, 가장 큰 값을 보이는 argument 를 예측 label 로 사용하는 앙상블 모델을 생성하였다.

```
Epoch 48/50
50/50 [==============================] - 6s 122ms/step - loss: 0.0737 - accuracy: 0.9756 - val_loss: 1.1365 - val_accuracy: 0.7975
Epoch 49/50
50/50 [==============================] - 6s 128ms/step - loss: 0.0928 - accuracy: 0.9719 - val_loss: 1.3401 - val_accuracy: 0.7300
Epoch 50/50
50/50 [==============================] - 6s 121ms/step - loss: 0.0706 - accuracy: 0.9775 - val_loss: 0.5873 - val_accuracy: 0.8475
mels_model_acc : 0.9275
Epoch 1/50
50/50 [==============================] - 9s 128ms/step - loss: 2.1208 - accuracy: 0.2156 - val_loss: 8.1543 - val_accuracy: 0.1000
Epoch 2/50
50/50 [==============================] - 6s 113ms/step - loss: 1.7123 - accuracy: 0.3931 - val_loss: 2.4706 - val_accuracy: 0.1375
Epoch 3/50
50/50 [==============================] - 5s 109ms/step - loss: 1.4420 - accuracy: 0.5144 - val_loss: 2.8020 - val_accuracy: 0.1000
Epoch 4/50
50/50 [==============================] - 6s 111ms/step - loss: 0.0618 - accuracy: 0.9775 - val_loss: 1.1076 - val_accuracy: 0.7400
Epoch 48/50
50/50 [==============================] - 6s 111ms/step - loss: 0.0684 - accuracy: 0.9800 - val_loss: 0.5597 - val_accuracy: 0.8700
Epoch 49/50
50/50 [==============================] - 6s 110ms/step - loss: 0.1439 - accuracy: 0.9600 - val_loss: 3.3443 - val_accuracy: 0.5000
Epoch 50/50
50/50 [==============================] - 6s 111ms/step - loss: 0.0993 - accuracy: 0.9706 - val_loss: 5.8045 - val_accuracy: 0.2300
mfcc_model_acc : 0.9100
ensemble_model_acc : 0.9500
```

4.3. 분석 결과

앙상블 모델은 각 fold 당 한 개의 결과를 반환하므로, 아래와 같이 총 5 개의 결과가 나온다.

```
# ensemble 모델 결과 (5개)
print(f'앙상블 모델의 결과는 {acc_list}과 같으며 평균은 {np.mean(acc_list):.4f}이다.')
```

```
앙상블 모델의 결과는 [0.9725, 0.9625, 0.995, 0.955, 0.95]과 같으며 평균은 0.9670이다.
```

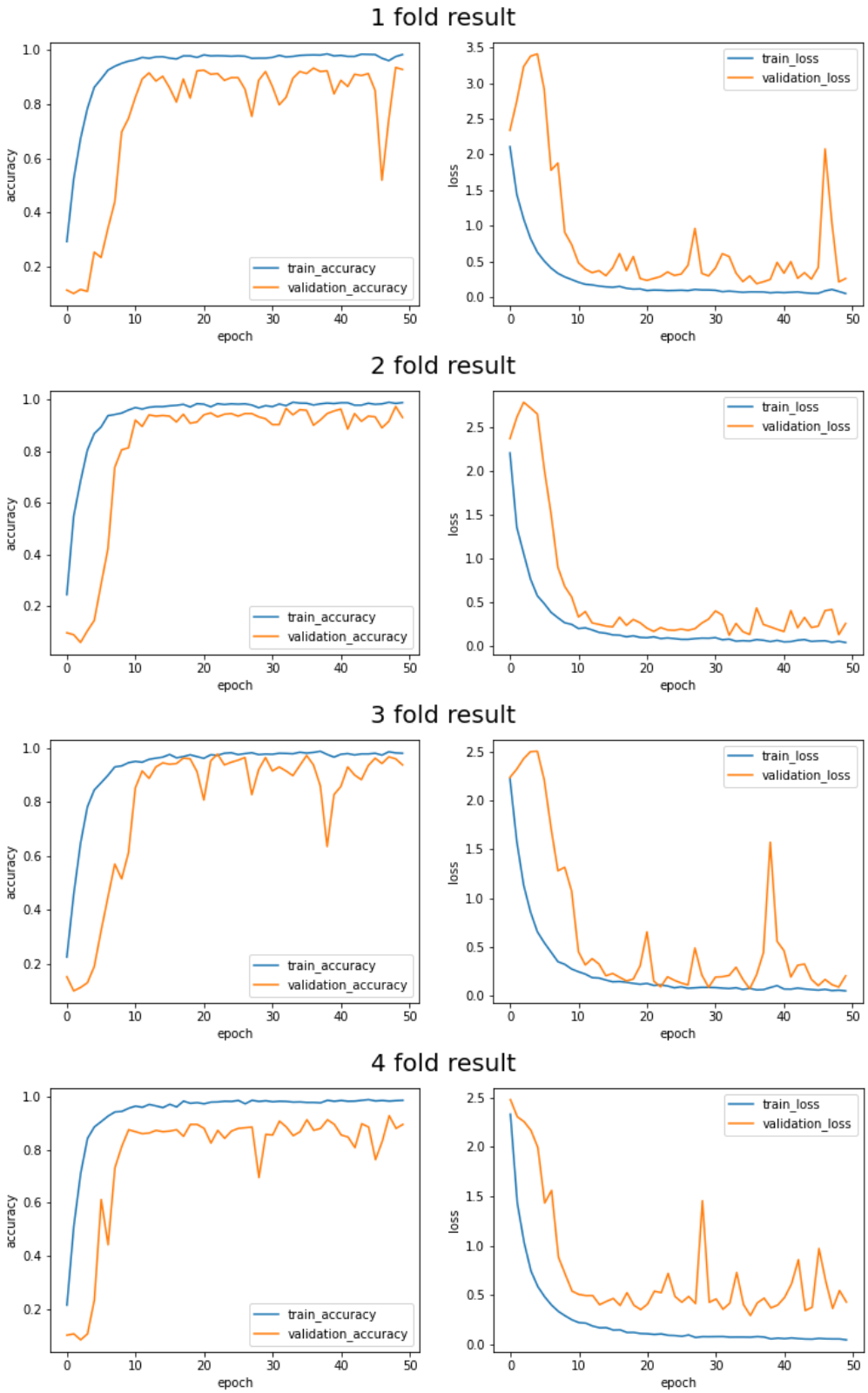
이들의 평균은 약 96.70%으로 매우 좋은 예측 성능을 보이는 것을 알 수 있다.

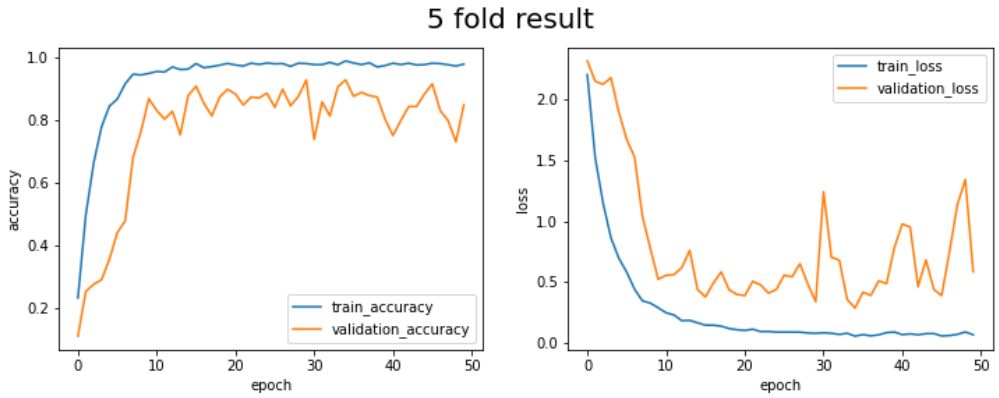
4.4. 결론

Mel spectrogram 과 MFCC 각각에 대해, 각 fold 에서 학습 성능과 예측 성능이 어떻게 변화하는지를 알고자 다음과 같은 코드를 실행하여 결과를 시각화하였다.

```
for i, _ in enumerate(skf.split(audio_mels_array, repeated_target)) :
  fig = plt.figure(figsize = (12, 4))
  plt.subplot(1, 2, 1)
  plt.plot(history_mel_list[i].history['accuracy'])
  plt.plot(history_mel_list[i].history['val_accuracy'])
  plt.ylabel('accuracy')
  plt.xlabel('epoch')
  plt.legend(['train_accuracy', 'validation_accuracy'], loc='best')

  plt.subplot(1, 2, 2)
  plt.plot(history_mel_list[i].history['loss'])
  plt.plot(history_mel_list[i].history['val_loss'])
  plt.ylabel('loss')
  plt.xlabel('epoch')
  plt.legend(['train_loss', 'validation_loss'], loc='best')
  plt.suptitle(f'{i + 1} fold result', fontsize = 20)
  plt.savefig(f'mels_{i + 1}.png')
plt.show()
```

우선 Mel spectrogram 에 대한 결과는 다음과 같다.

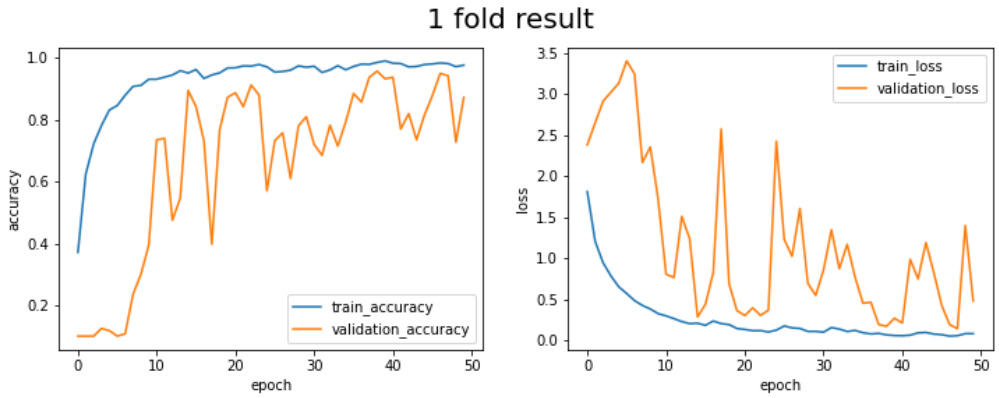


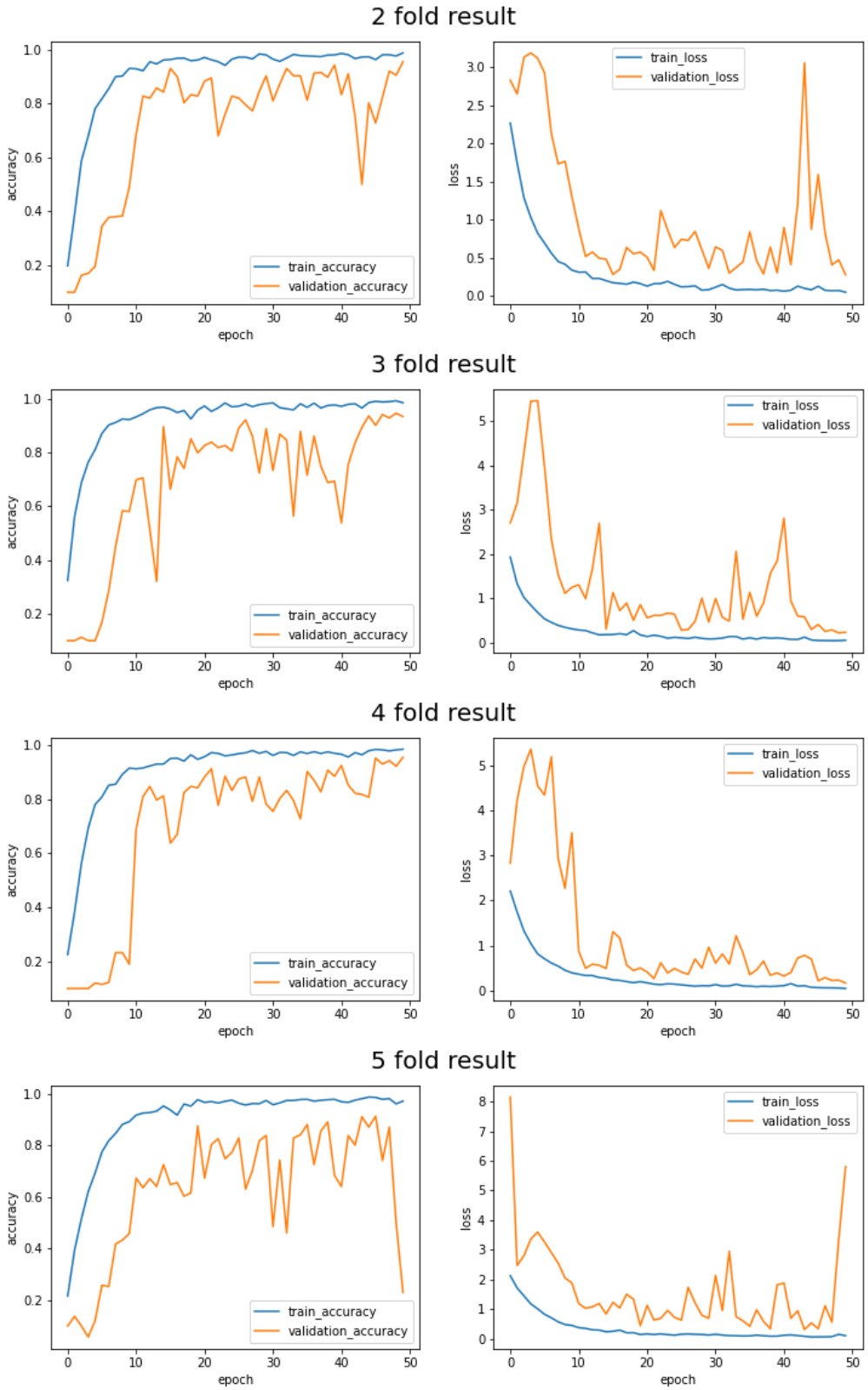


대부분의 fold 에서 epoch 이 진행될수록 accuracy 가 높아지는 것을 볼 수 있으며, 이와 동시에 loss 값은 감소하는 경향이 있다는 것을 볼 수 있었다.

```
for i, _ in enumerate(skf.split(audio_mels_array, repeated_target)) :
  fig = plt.figure(figsize = (12, 4))
  plt.subplot(1, 2, 1)
  plt.plot(history_mfcc_list[i].history['accuracy'])
  plt.plot(history_mfcc_list[i].history['val_accuracy'])
  plt.ylabel('accuracy')
  plt.xlabel('epoch')
  plt.legend(['train_accuracy', 'validation_accuracy'], loc='best')

  plt.subplot(1, 2, 2)
  plt.plot(history_mfcc_list[i].history['loss'])
  plt.plot(history_mfcc_list[i].history['val_loss'])
  plt.ylabel('loss')
  plt.xlabel('epoch')
  plt.legend(['train_loss', 'validation_loss'], loc='best')
  plt.suptitle(f'{i + 1} fold result', fontsize = 20)
  plt.savefig(f'mfcc_{i + 1}.png')
plt.show()
```

다음은 위 코드를 이용하여 시각화한 MFCC 에 대한 결과이다.





Mel spectrogram 과 비교했을 때, 전반적인 경향은 비슷하지만 검증 데이터셋에 대한 accuracy 와 loss 모두가 mel spectrogram 에 비교하여 많이 불안정한 것을 볼 수 있었다.

이를 수치로 표현한 값은 아래와 같다.

```
# stratifiedkfold 결과 (average) 사용

train_accs_mel, train_accs_mfcc, val_accs_mel, val_accs_mfcc = [], [], [], []
for i, _ in enumerate(skf.split(audio_mels_array, repeated_target)) :
  train_accs_mel.append((i, history_mel_list[i].history['accuracy'][-1], 'train_accuracy_mel'))
  train_accs_mfcc.append((i, history_mfcc_list[i].history['accuracy'][-1], 'train_accuracy_mfcc'))
  val_accs_mel.append((i,history_mel_list[i].history['val_accuracy'][-1], 'val_accuracy_mel'))
  val_accs_mfcc.append((i, history_mfcc_list[i].history['val_accuracy'][-1], 'val_accuracy_mfcc'))
accs_avg = {}
for accs_list in [train_accs_mel, train_accs_mfcc, val_accs_mel, val_accs_mfcc] :
  accs_sum = 0
  for i in range(len(accs_list)) :
    accs_sum += accs_list[i][1]
  print(f'{accs_list[i][2]:<20} 의 평균은 {accs_sum / len(accs_list) :.3f}')
```

```
train_accuracy_mel   의 평균은 0.983
train_accuracy_mfcc  의 평균은 0.981
val_accuracy_mel     의 평균은 0.908
val_accuracy_mfcc    의 평균은 0.789
```

앞서 그래프에서도 볼 수 있었듯, accuracy 는 검증 데이터에서보다는 학습 데이터에서 더 높으며 mfcc 에서보다 mel spectrogram 에서 더 높은 값을 보였다. 이는 앞서 데이터 전처리 단계에서 mel spectrogram 과 mfcc 를 비교한 결과를 해석했을 때와 같은 결과이다.

5. 참고자료

Resblock 설명 (tistory)

'Voice Recognition Using MFCC Algorithm' 논문