인공지능을 활용한
음악 장르 분류

- 최윤주

목차

1. 프로젝트 배경: 프로젝트 목표와 그 이유

오디오 데이터에 관심을 갖게 된 것은 생성형 AI 의 일종인 RVC 때문이었다. 가수들의 목소리만 추출한 데이터(20 분 이상)들로 학습한 모델을 통해 원하는 노래를 음성 변조를 시켜서 만들 수 있다. 애초에 가수에게 원하는 노래를 부르게 하려면 돈만으로도 되는 문제가 아닌데, 저작권 문제가 있지만 AI 로 완성도 높은 결과를 낼 수 있다는 것이 흥미로웠다.

하지만 RVC 는 전문 프로그래머가 만든 것이라 코드를 봐도 이해가 불가능했고 오디오 데이터에 관심이 가던 차에 음원을 특징 벡터화 하여 이미지로 나타낼 수 있는 기술과, 음성분류 논의가 꽤나 주요하게 진행되는 것을 인터넷에서 보았다. 음성 분석법에는 여러가지가 있는데 방법론(3 장)에서 쓰도록 하겠다. 음성 분류 주제에는 음악 장르 분류, 의료데이터(심장소리 등)를 이용한 진단법, 통역을 위한 추임새 구분 등 다방면의 분야에서 활용되고 있었다.

흥미를 가지게 된 취지와 가장 유사한 음악 장르 분류를 프로젝트 주제로 선정했다. 음악 장르 분류를 성공적으로 한다면 결과를 바탕으로 음악 추천 시스템을 만들 수 있다. 추천 시스템은 음원 플랫폼 상의 유저들을 끌어 모으는데 핵심적인 역할을 하므로 사업 아이템이 될 수 있다. 예시로 유튜브를 보면 유저가 클릭한 것과 비슷한 유형의 음악들을 끊임없이 추천해줘서 사이트 접속 지속 시간을 늘린다. 또 창작자들이 필요에 의해 음악을 공연, 영화, 광고등에 사용할 때 적절한 음원을 찾는데 어려움이 있다고 한다. 만약 영감을 받은 기존 창작물이 있다면 그것에 사용된 음악과 비슷한 유형의 음악들을 찾아내서 사용 가능하다.

프로젝트 목표는 최대한 많은 클래스(장르) 내에서 정확도 80%이상의 분류 모델을 만드는 것이다. 예를 들어, rock 이라는 큰 장르 내에서도 그런지, 하드, 펑크, 메탈, 모던, 프로그레시브, 얼터너티브등 세부적으로 장르를 나눌 수 있다. 음악 자체를 기준으로 한 추천 시스템을 만드는 데 기반이 되는 분류 모델을 만드는 것이 목적이다. 다만 추천시스템 자체는 더 많은 지식과 연구가 필요하므로 제외했다. 장르를 세부적으로 나눌수록 추천시스템의 완성도가 올라갈 것으로 전제한다.

앞서 말한 유튜브 추천 시스템은 사용자와 유사한 다른 사용자들의 시청 습관을 비교하여 이를 활용해 사용자가 시청하고 싶어할 만한 다른 콘텐츠를 추천하는 방식으로 동작한다. 이 방식은 아주 효과적이고 유튜브를 음악동영상 사이트 1 위로 만드는 데 큰 기여를 했지만 다른 사용자들이 많이 시청한 음원만을 추천하기 때문에(이 방식이 나쁘다고 할 수는 절대 없고 아직까지 가장 효과적인 방법이기는 하다.) 비교적 알려지지 않은, 사용자가 적은 음원에 관해서는 영원히 추천 동영상으로

선정되지 않을 수 있다. 그래서 음악 장르 자체에 대한 추천 시스템을 제안한다.

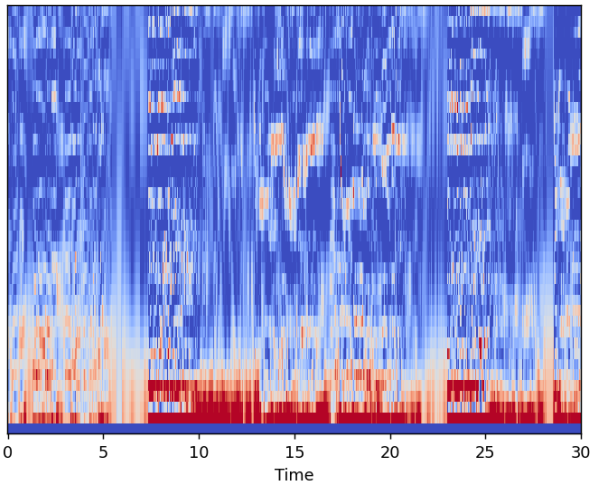
2. 데이터 설명

데이터는 케글(kaggle)에서 찾은 GTZAN 데이터셋을 사용했다. 본 데이터셋은 10 가지 클래스의 음악 장르로 나뉘어져 있고 각 클래스 당 30 초의 100 개 wav 파일을 포함한다. 클래스는 rock, reggae, pop, metal, jazz, hiphop, disco, country, classical, blues 이고 총 1000 개의 데이터 셋인데, 재즈 클래스의 파일 하나에 오류가 있어 실제 사용 가능한 데이터는 999 개이다.

클래스 수에 비하면 데이터 표본이 적기 때문에 각 클래스 당 추가적으로 5 개의 음원 파일을 30 초씩 나눈 파일을 추가하여 음원 길이에 따라 약 30~45 개 정도의 데이터를 추가하였다. (총 400 개 데이터 추가 생성) 데이터 양을 늘리기엔 이것도 현저히 수가 적지만, 추가할 노래를 선정해서 구하는 작업과 30 초씩 음원을 잘라 저장하는 것이 시간이 많이 필요한 작업이었다.

또, 트로트 장르 클래스를 추가하였다. 트로트(trot)는 한국, 일본 등에서 주로 듣는 음악 장르인데 GTZAN 데이터셋은 해외에서 만든 데이터라 서양음악 위주의 장르로만 구성되어 있어서 장르의 다양성을 추구하고자 트로트 장르를 추가하였다. 데이터는 유명 트로트 20 개의 음원 파일을 추출한 뒤 각 음원을 30 초씩 나누어 106 개의 트로트 표본 데이터셋을 구축하였다.

음원을 자르는 작업은 직접 진행하였다. 음원 데이터를 불러온 후 파이썬 pydub 라이브러리를 사용하여 음원을 30 초씩 분할하는 것을 자동화할 수 있지만 그러면 장르 특징을 가진 부분을 집중적으로 잘라내는 데 제약이 있어서 수작업을 선택했다. https://mp3cut.net/ko/라는 사이트에서 지원하는 음성 분할 서비스를 이용하여 음원의 장르적 특징을 가장 잘 반영하는 구간만을 30 초씩 잘라내었다. 음원 이미지화 전 처리 과정 중에 패딩 과정이 있으니 30.x 초 등의 정확한 분할은 크게 중요하지는 않았다.

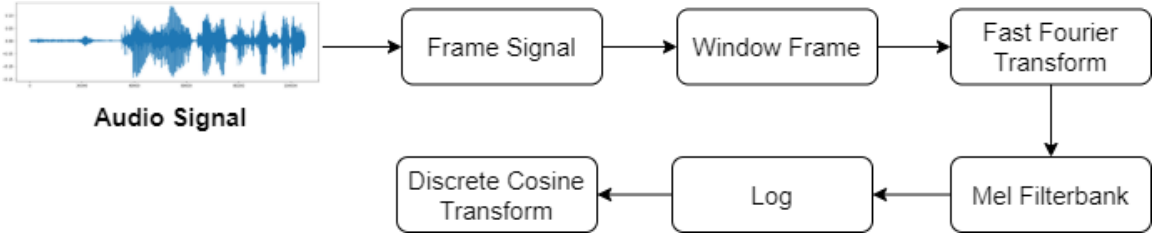


위 사진은 음원 데이터 전 처리 후 결과의 예시이다. 음원 파일들의 특징을 추출한 후 이를 이미지화 한 파일을 .png 확장자로 저장 후 이미지 파일로 음악 분류 모델을 만들 것이다.

3. 방법론 설명

앞서 음악 장르 분류를 프로젝트 주제로 선정한 이유 중 하나는 음원을 특징 벡터화 하여 이미지로 나타낼 수 있는 기술의 신기했기 때문이다. 음성의 이미지화에는 여러 방법이 있지만, 그 중에서도 오디오 신호를 분석하는 데 사용되는 두 가지 주요 특징 추출 방법인 멜 스펙트로그램(Mel spectrogram)과 MFCC(Mel-Frequency Cepstral Coefficient)를 중점으로 본 프로젝트를 진행한다.

멜 스펙트로그램과 MFCC 모두 주파수 표현 방식의 일종이며 소리의 고유한 특징값을 추출한다. MFCC 를 간략히 표현한다면 멜 스펙트럼에서 켑스트럴(Cepstral) 변환을 통해 추출된 값이다. MFCC 의 추출법을 설명하자면 다음과 같다:



먼저 오디오 신호를 원본 데이터로부터 추출 후(샘플링 비율에 따라 다르지만 보통 초당 20,000 개 이상이다.) 해당 신호를 잘게 분할한 후(20ms~30ms), 고속 푸리에 변환을 해서 스펙트럼 값을 구한다. 여기서 오디오 신호는 시간 축을 사용하는 시계열 데이터로 오디오 파형을 의미한다. 고속 푸리에 변환은 이산 푸리에 변환(DFT)의 계산량을 줄이는 알고리즘이며 디지털 신호처리에 많이 쓰인다. 고속 푸리에 변환은 시간에 대한 함수를 주파수 성분으로 분해하는 변환이다 이를 통해 함수를 주파수 영역에서 표현 가능하게 된다. 각 주파수에서의 진폭은 원 함수를 구성하던 주파수 성분의 크기를 나타내게 된다.

푸리에 변환을 알기 쉽게 시각화한 이미지는 다음과 같다:

스펙트럼이란, 각 주파수의 대역 별 세기를 나타낸다. 한 신호에서 어떤 주파수의 강하고 약한 정도를 알려준다. 스펙트럼은 고속 푸리에 변환을 통해 얻을 수 있다. 이후, 켑스트럴(Cepstral) 변환을 통해 스펙트럼으로부터 정보를 추출해 내는데 MFCC 는 그냥 스펙트럼을 사용하지 않고 멜 스펙트럼을 사용한다. 즉, 멜 스펙트럼으로부터 정보를 추출해 낸다.

멜 스펙트럼이 일반 스펙트럼과 다른 점은 실제 사람이 인식하는 청각 특성을 반영했다는 점이다. 일반적으로 사람은 높은 주파수의 변화에는 둔감하거나 아예 인식하지 못하는 반면, 낮은 주파수의 변화는 잘 알아챈다. 예시로 돌고래가 인식하는 고주파를 사람은 듣지 못하는 것이 있다. 주파수를 사람의 청각 특성에 맞게 변환하는데 사용되는 것이 멜 스케일이고, 멜 스펙트럼은 스펙트럼을 멜 스케일을 통해 변환하여 나타낸 것이다. 멜 스펙트럼을 사용한다면 사람이 인식 가능한 주파수 내에서 음원을 분류하는 데 효과적이다.

앞서 켑스트럴 변환을 통해 멜 스펙트럼으로부터 정보를 추출해 낸다고 하였다. 켑스트럴 변환 과정은 먼저 푸리에 변환(=주파수 변환)에 로그를 취한 주파수 영역의 로그 변환으로 켑스트럼을 얻고 이를 역 푸리에 변환하여(다시 시간 영역으로 변환) 켑스트럼을 얻는다. 로그를 사용하여 두개의 곱으로 된 함수를 합으로 나타내 주며 주파수 영역으로 변환된 신호 크기를 분리하여 나타낼 수 있다. 켑스트럴 변환으로 특징을 추출할 수 있다.

MFCC(Mel-Frequency Cepstral Coefficient)는 켑스트럴 변환 후 마지막으로 이산 코사인 변환을 하여 최종적으로 소리에 대한 특징을 추출하게 된다. 이산 코사인 변환은 신호의 에너지를 적은 수의 계수로 집중시키는 특성이 있기에 적은 수의 주요 계수만으로도 신호를 효과적으로 표현할 수 있다.

아래 코드는 하이퍼 파라미터 튜닝 전 코드로, 설명을 위해 첨부한다:

```
import numpy as np
import matplotlib.pyplot as plt
import os
import librosa
import librosa.display

path = 'C:₩₩Users₩₩cyj42₩₩Downloads₩₩music_Data₩₩music₩₩'

file_list = os.listdir(path)
file_list_py = [file for file in file_list if (file.endswith('.mp3') or file.endswith('.wav'))]
print("file_list_py len:", len(file_list_py))
ter=0
def pad_to_length(array, target_length):
    pad_width = ((0, 0), (0, max(0, target_length - array.shape[1])))
    padded_array = np.pad(array, pad_width=pad_width, mode='constant', constant_values=0)
    return padded_array

for i in file_list_py:
    ter+=1

    print("ter:",ter)

    audio, sr = librosa.load(path + i, sr=None)
    print('sr:', sr, ', audio shape:', audio.shape)
    print('length:', audio.shape[0] / float(sr), 'secs')
    print(i)
    mel_spectrogram = librosa.power_to_db(librosa.feature.melspectrogram(y=audio, sr=sr, n_fft=2048, hop_length=512, n_mels=128))
    padded_mel_spectrogram = pad_to_length(mel_spectrogram, target_length=1290)
    mfcc = librosa.feature.mfcc(S=librosa.power_to_db(padded_mel_spectrogram), n_mfcc=40)

    librosa.display.specshow(mfcc, sr=sr, x_axis='time', vmin=-20, vmax=80)
    #vmin, vmax에서 색상강도 조절해서 이미지가 더 잘 식별가능하게함

    #plt.savefig(f"C:₩₩Users₩₩cyj42₩₩Downloads₩₩mfcc_₩₩plot_{i}.png", dpi=128)
    plt.show()
```

이미지 생성 및 저장 방법을 설명하자면, 먼저 파이썬 os 라이브러리의 os.listdir()를 통해 파일이 있는 폴더 경로를 입력 후 파일들의 이름 정보들을 불러온다. 그후, 음악 확장자 파일만 다시 골라내는 코드를 거친 뒤 파일이름 정보들을 가진 리스트를 for 루프를 돌리며 소리데이터로부터 MFCC 방법을 사용하여 특징을 추출하고 이미지 화한 그래프를 파일경로를 지정하여 저장한다.

먼저 파이썬 librosa 라이브러리의 librosa.load ()함수를 사용하여 오디오 파일로부터 오디오 신호와 샘플링 속도를 추출해 낸다. 그 다음 파이썬 librosa 라이브러리의 librosa.feature.melspectrogram()을 통해 MFCC 에 앞서 멜 스펙트로그램을 구한다. 이 함수의 parameter 에 대한 설명은 다음과 같다.

y: 입력 오디오 신호

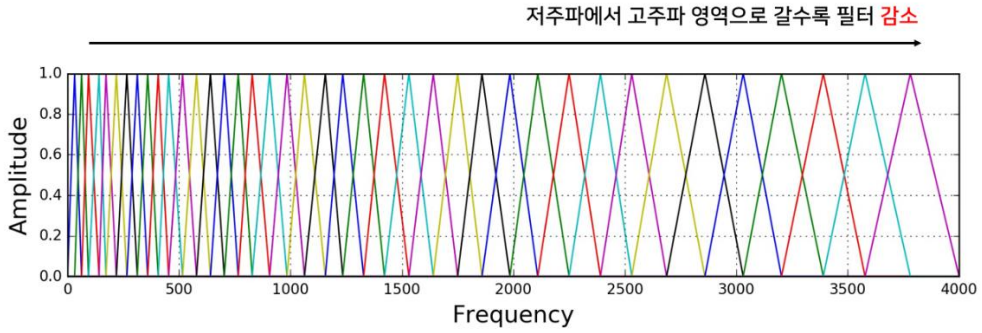
sr: 오디오 신호의 샘플링 속도이며, 이는 초당 갖는 데이터의 값이다. 기본값인 22050 으로 설정했다.

n_fft: 푸리에 변환을 계산하는 데 사용되는 window 의 크기이다. Window 란 신호를 잘게 나눈 각 프레임 내에서 프레임의 가장자리를 덜 강조하면서 프레임의 중앙 부분을 강조하는 역할을 한다. 오디오 신호를 분석할 때 신호를 짧은 세그먼트나 프레임으로 나누고 각 프레임에서 푸리에 변환을 하는 것이 일반적이다. 보통 512~2048 사이의 값으로 설정한다.

hop_length: 음성을 자른 프레임 간격으로, hop_length 가 작을수록 더 많은 프레임을 얻고 시간 해상도(시각적으로 무리없이 부드럽게 보여주는 능력)가 증가하지만 계산 또한 증가한다. 반면 hop_length 가 클수록 프레임 수가

줄어들고 시간 해상도는 낮아지지만 처리 속도가 빨라진다. 보통 256 이나 512 로 설정한다.

n_mels: 사용할 멜 필터의 개수이다. 멜 스펙트로그램은 스펙트럼에 멜 필터를 적용해서 얻는다. 멜 필터는 사람 청각의 고유 특징을 반영하게 해준다. 보통 256 이나 128 로 설정한다. 아래 사진자료는 멜 필터에 대한 설명이다:



함수의 parameter 는 여러 시행착오를 통해 적절한 값을 찾는 것이 좋다. 기계학습을 이용한 이미지 분류는 사람이 알아보기 편하면 모델 성능도 높아지므로 적절한 전 처리 방법을 알아내는 것이 중요하다.

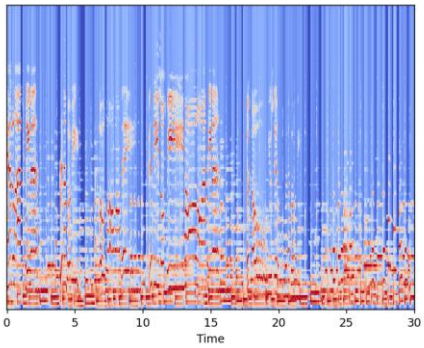
멜 스펙트로그램을 구한 후 패딩 작업을 거친다. 음원 파일이 모두 30 초라고 해도 모두 정확히 30 초인 것은 아니고 0.1 초 이내에서 차이가 있기 때문에 적절한 길이로 맞추어 주는 과정이 필요하다. 패딩은 따로 구현한 함수로 처리할 수 있다. 이 함수는 데이터의 길이가 모자라면 0 으로 채우고 설정한 길이보다 데이터가 더 길면 자른 결과를 반환하는 함수이다. 멜 스펙트로그램 그래프는 x 축은 시간, y 축은 주파수를 의미하기에 2 차원 데이터이므로 함수의 인풋 파라미터도 2 차원이어야 한다.

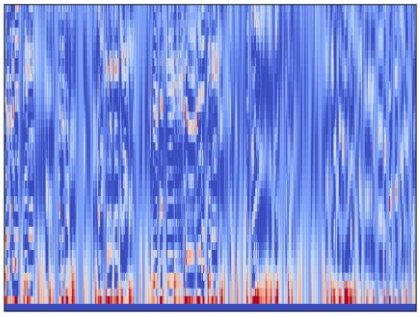
pad_to_length 함수의 parameter 중 target_length 은 데이터의 시간 축 길이를 의미한다. Target Length=Duration (seconds)×Frame Rate 이고 Frame Rate= Hop Length/Sampling Rate 으로 정의된다. 앞서 hop_length=512 로 설정했고 Sampling Rate 는 22050 이므로 Frame Rate 은 약 43 이다. 즉, Target Length=30×43, Target Length 는 1290 이다.

패딩 과정을 거친후, 파이썬 librosa 라이브러리의 MFCC 함수(librosa.feature.mfcc())를 사용해서 MFCC 값을 도출한다. 함수 parameter 인 n_mfcc 은 리턴 값으로 몇 개의 Mel-Frequency Cepstral Coefficients 를 받을지 결정한다. n_mfcc 값이 높을수록 특징이 많이 추출되어 이미지 화했을 때 픽셀의 쪼개심이 더 많다. 디폴트 값은 20 이지만 특징을 더 많이 추출하기 위해 이 값은 40 으로 설정했다. n_mfcc 의 최적 값은 세분화된 장르 분류와 관련된 작업이라면, 더 많은 수의 계수를 사용하는 것이 좋다. librosa.power_to_db() 함수는 필수는 아니지만 제곱(power) 스펙트로그램을 데시벨로 바꾸어서 시각화나 분석에 용이하게 만들기 때문에 사용된다. 데시벨(dB)은 신호의 크기를 나타내기 위해 오디오 처리에 일반적으로 사용되는

로그 단위로 로그 스케일은 사람이 음량의 변화를 인식하는 방식과 더 일치하기 때문에 오디오 분석에 주로 쓰인다.

아래 이미지는 같은 오디오 데이터를 (1) 멜 스펙트로그램만 적용한 이미지와 (2)MFCC 까지 적용한 이미지이다:





-멜 스펙트로그램 이미지에 비해 MFCC 이미지가 분절적이다. MFCC 계수 추출 시 설정한 값(n_mfcc)에 따라 이는 달라질 수 있다.

librosa.util.normalize() 함수를 사용하면 MFCC 결과를 정규화 할 수 있지만, 정규화 시킨 결과를 이미지화 했을 때 그래프의 식별가능성이 오히려 적어져서 사용하지 않았다.

librosa.display.specshow() 함수를 사용하여 이미지를 어떻게 보여주면 효과적일 것인지를 결정한다. Vmin, vmax 라는 parameter 를 통해 색상의 상한과 하한을 정해서 색상 분포를 훨씬 알아보기 쉬운 방향이 되도록 하였다. , vmin=-20, vmax=80 로 설정했을 때 가장 효과적이었다.

마지막으로 plt.savefig() 함수를 사용해서 해당 이미지를 원하는 경로에 저장한다. 해상도는 dpi=128 로, 모델이 학습하기 버겁지 않은 정도로 설정했다. 모델 학습 속도를 빠르게 하기 위해 이 값을 너무 작게 설정하면 이미지가 알아보기 힘들게 된다.

아래 코드는 이미지를 불러오고, 라벨 값을 저장하는 코드의 일부이다:

```
import cv2
import os
import numpy as np

path = '/content/drive/MyDrive/mfcc_/'
file_list = os.listdir(path)
file_list_py = [file for file in file_list if file.endswith('.png')]



#250, 420, 3
image_w = 420
image_h = 250

X = []  # 이미지저장
Y = []  # 라벨값저장
for i in file_list_py:
    print(path+i)
    img = cv2.imread(path+i)
    img = cv2.resize(img, (image_w, image_h))
    img = img / 255.0
    label=i[5:8]
    if label== 'roc':
        Y.append(0)
    elif label == 'reg':
        Y.append(1)

    elif label== 'pop':
        Y.append(2)

    elif label== 'met':
        Y.append(3)
```

본격적인 학습을 위해 이미지를 불러올 차례다. 이미지 처리 라이브러리인 cv2 를 import 한 후 사용한다. 이미지 파일의 경로를 입력한 후, 이미지 파일을 하나하나 불러온다. 불러올 이미지 해상도 폭은 원래 이미지 배율과 맞게 330, 높이는 250 로 설정했다. 이 값은 이미지 학습 시 코랩이 끊기지 않는 최댓값이었다. X 리스트에는 이미지를 추가하고 Y 리스트에는 이미지 파일 이름에 인덱싱된 장르 정보를 저장해서 라벨을 만들었다. 불러온 이미지는 정규화 과정(픽셀값을 255.0 으로 나누었다.)을 거쳐 픽셀값들을 0 과 1 사이로 만들었다. 정규화를 하면 이미지 학습에 효과적이다.

GTZAN 데이터가 적어서 추가로 데이터를 구했지만 여전히 학습 결과의 정확도를 높이기엔 매우 부족하다. 그래서 keras 의 ImageDataGenerator 를 사용해 기존 이미지 데이터 수를 늘리도록 했다. 만약 이 방법을 적용했을 때 결과가 적은 원본 데이터만을 사용한 결과보다 안 좋다면 쓰지 않도록 한다. 일반 이미지 데이터의 사물의 위치를 변화시킨다 하더라도 사물을 인식하는 데는 문제가 없지만 해당 데이터는 그래프 이미지이기 때문에 위치 정보가 중요하게 작용할 수 있기 때문이다. ImageDataGenerator 의 parameter 는 다음과 같다. shear_range: 이미지를 찌그러트리는 강도, zoom_range: 이미지 확대 또는 축소 범위를 지정, horizontal_flip: 이미지를 수평으로 뒤집는다. vertical_flip: 이미지를 수직으로 뒤집는다. rescale: 이미지의 픽셀 값의 범위를 지정한다. width_shift_range, height_shift_range 를 이용해서 이미지를 수직이나 수평으로 이동하는 정도를 설정한다. 다음은 이를 적용한 코드이다:

```
#image generator version
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense,Activation,Dropout,Conv2D
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import optimizers
import numpy as np

train_datagen=ImageDataGenerator(rescale=1./255,
                                 height_shift_range=0.1,
                                 zoom_scale=0.1
                                 )
train_generator=train_datagen.flow_from_directory('/content/drive/MyDrive/mfcc_train/',
                                                  target_size=(288,432),
                                                  batch_size=5,
                                                  class_mode='categorical')
test_datagen=ImageDataGenerator(rescale=1./255)
test_generator=test_datagen.flow_from_directory('/content/drive/MyDrive/mfcc_test/',
                                                target_size=(288,432),
                                                batch_size=5,
                                                class_mode='categorical')
```

ImageDataGenerator 는 입력한 parameter 들을 무작위로 적용시켜 모델이 학습하는 이미지 데이터를 늘린다. 이 방법론에서는 그래프에 영향이 가장 적을 것으로 예측되는 parameter 인 rescale 과 height_shift_range(수직이동), zoom_range (이미지 확대)만을 사용했다. 학습데이터와 검증데이터의 비는 8:2 로 했다.

모델 학습에는 직접 만든 CNN 모델과 keras 에서 불러온 VGG16 모델을 각각 사용해 보기로 하였다. VGG16 을 사용하면 imagenet 의 데이터들을 미리 학습해서 가져온 weight 를 사용해서 모델의 학습 수준을 높이고 시간은 단축할 수 있다. VGG16 은 ILSVRC 라는 머신러닝 이미지 분류 경진대회에서 2 위를 한 모델이다. 깊은 은닉층을 사용하고 따라서 weight 개수도 많다.

VGG16 구조는 13 개의 convolution 층을 가지고 3 개의 fully connected 층이 있다. convolution 층에서 이미지의 local 특징들을 추출하고 fully connected 층에서 2 차원 데이터를 flatten 하고 이미지 분류를 위한 재가공을 한다. VGG16 는 학습 파라미터 수를 감소시키기 위해 convolution 시 3*3 크기의 필터만을 사용했다.

아래 코드는 VGG16 모델을 사용한 코드이다:

```
from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
model_VGG16 = VGG16(weights='imagenet', include_top=False, input_shape=(250, 420, 3))
model_VGG16.trainable = False

V_model = Sequential()
V_model.add(model_VGG16)
V_model.add(Flatten())
V_model.add(Dense(units=64, activation='relu'))
model.add(Dropout(0.25))
V_model.add(Dense(units=11, activation='softmax'))
V_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

#모델저장경로
modelpath="/content/drive/MyDrive/mel_spec/best_model_VGG_.hdf5"
checkpointer=ModelCheckpoint(filepath=modelpath, monitor='val_loss',
                             verbose=1,save_best_only=True)
early_stopping_callback=EarlyStopping(monitor='val_loss',patience=10)
history=V_model.fit(x_train,y_train,validation_split=0.2,epochs=30,batch_size=100,
                    callbacks=[early_stopping_callback,checkpointer])
```

불러온 VGG16 구조의 fully connected 층은 사용 안 할 것이기 때문에 include_top=False 로 설정했다. 추가로 만든 fully connected 층은 Dense 연산 두 번과 드롭 아웃으로 이루어져 있다. 첫번째 Dense 연산에선 노드 개수를 64 개로 해서 연산을 수행하고 활성화 함수로는 relu 를 사용했다. 과적합을 막기 위해 드롭 아웃 시킨 후, 분류를 위해 다시 활성화함수로 소프트맥스 함수를 사용한 Dense 연산을 진행했다.

아래 코드는 일반 CNN 모델 코드이다:

```
model = Sequential()
model.add(Conv2D(64, kernel_size=3, padding='same', activation='relu', input_shape=(250,420, 3)))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(32, kernel_size=3, activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(11, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# 모델 저장 경로
modelpath = "best_model.hdf5"
checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_loss', verbose=1, save_best_only=True)
early_stopping_callback = EarlyStopping(monitor='val_loss', patience=10)
history = model.fit(x_train, y_train, validation_split=0.1, epochs=30, batch_size=100,
                    callbacks=[early_stopping_callback, checkpointer])
```

직접 만든 CNN 모델에서는 Conv2D 연산을 진행한 후 풀링층을 거친다. 또 Conv2D 연산 후 풀링 연산을 하고 과적합을 막기 위해 드롭 아웃을 시킨다. Fully-connected 층에서는 앞서 구한 특징 맵을 1 차원화 하고 다시

Dense 연산을 한 다음 드롭 아웃을 진행하고 마지막으로 소프트맥스 함수를 사용해서 다중분류(11 개 클래스)에 적합하게 한다.

모델 컴파일 부분에서는 optimizer 는 가장 널리 쓰이는 adam 을 사용했고 오차함수는 다중분류기 때문에 categorical_crossentropy 를 사용했다. 과적합을 방지하고 모델이 가장 잘 학습된 시기를 포착해서 저장하기 위해 ModelCheckpoint 함수와 EarlyStopping 함수를 사용했다. Validation data 의 loss 가 10 번 연속 줄지 않으면 학습이 중단되도록 했다.

학습률 조정은 하이퍼 파라미터 튜닝 방식이 아닌, Tenserflow 의 ReduceLROnPlateau 를 사용하여 검증셋이 향상되지 않을 때 학습률을 줄이는 방법을 쓰기로 했다. 이유는 학습률을 하나로 지정하기보단 모델 학습이 진행함에 따라 학습률이 작아져야 한다는 데 주목했다. ReduceLROnPlateau 를 통해 검증셋 오차가 5 에포크 동안 낮아지지 않으면 학습률을 0.1 배로 감소시켰다. (기본 시작 lr 은 0.001 이다.)

```
from tensorflow.keras.callbacks import ReduceLROnPlateau
lr_reducer = ReduceLROnPlateau(factor=0.1, patience=5, min_lr=1e-6)
```

아래는 다른 변수로 진행한 하이퍼 파라이터 튜닝 과정 코드의 일부이다:

```
param_grid = {
    'n_fft': [1024, 2048],
    'hop_length': [256, 512],
    'n_mels': [128, 256]
}
path = '/content/drive/MyDrive/music/'

file_list = os.listdir(path)
file_list_py = [file for file in file_list if file.endswith('.wav')]
print("file_list_py len:", len(file_list_py))

grid = ParameterGrid(param_grid)
Y = []


for params in grid:

    n_fft = params['n_fft']
    hop_length = params['hop_length']
    n_mels = params['n_mels']
    iter=0
    print(f"hyper parameter: n_fft: {n_fft}, hop_length: {hop_length}, n_mels: {n_mels}")
    for i in file_list_py:
        iter+=1
        print("iter: ",iter)
        audio, sr = librosa.load(path + i, sr=None)
        mel_spectrogram = librosa.power_to_db(librosa.feature.melspectrogram(y=audio, sr=sr, n_fft=n_fft, hop_length=hop_length, n_mels=n_mels))
        padded_mel_spectrogram = pad_to_length(mel_spectrogram, target_length=1290)
        mfcc = librosa.feature.mfcc(S=librosa.power_to_db(padded_mel_spectrogram), n_mfcc=40)

        librosa.display.specshow(mfcc, sr=sr, x_axis='time', vmin=-20, vmax=80)
        plt.savefig(f"/content/drive/MyDrive/grid/plot_{i}.png", dpi=128)

        label=i[5:8]
        if label== 'roc':
            Y.append(0)
```

```
file_list_img = os.listdir(path_img)
file_list_py_img = [file for file in file_list_img if file.endswith('.png')]

image_w = 256
image_h = 128
y_train = to_categorical(Y)

X = []
for i in file_list_py_img:
    print(path_img+i)
    img = cv2.imread(path_img+i)
    img = cv2.resize(img, (image_w, image_h))
    img = img / 255.0

    X.append(img)


model_VGG16 = VGG16(weights='imagenet', include_top=False, input_shape=(256,128,3))
model_VGG16.trainable = False

V_model = Sequential()
V_model.add(model_VGG16)
V_model.add(Flatten())
V_model.add(Dense(units=10, activation='softmax'))
V_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

#모델저장경로
modelpath="best_model_VGG.hdf5"
checkpointer=ModelCheckpoint(filepath=modelpath, monitor='val_loss',
                             verbose=1,save_best_only=True)
early_stopping_callback=EarlyStopping(monitor='val_loss',patience=10)
history=V_model.fit(X,y_train,validation_split=0.2,epochs=20,batch_size=100,
                    callbacks=[early_stopping_callback,checkpointer])
print("accuracy:",history.history['val_accuracy'],"%")
```

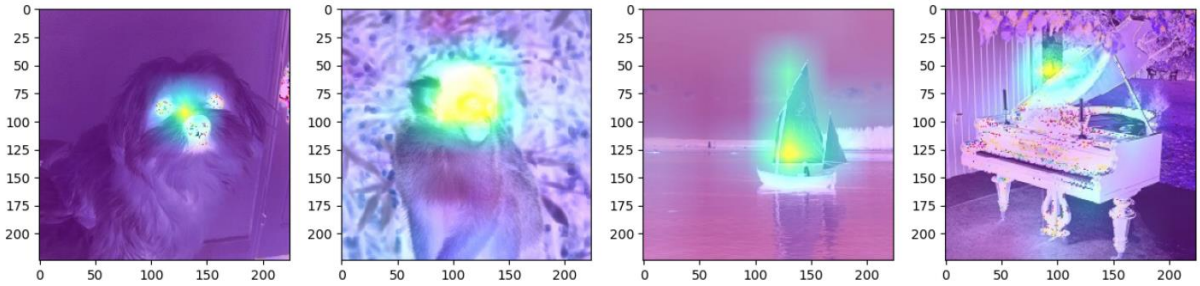
모델 성능을 향상시키기 위해 하이퍼 파라미터 튜닝을 진행한다. 하이퍼 파라미터란 사용자가 모델 학습 전에 입력하는 고정된 입력값이다. 데이터에 따라 최적의 파라미터 값은 다를 수 있다. 그래서 사용하는 방법이 하이퍼 파라미터 튜닝이다. 하이퍼 파라미터 튜닝 방법에는 그리드 서치(격자 탐색) 방법이 있다. 그리드 서치 방법은 하이퍼 파라미터에 넣을 수 있는 모든 값들을 리스트 같은 저장 공간에 넣어 놓고 각각 학습을 진행한 다음 결과가 가장 좋은 성능의 파라미터를 선택하는 방법이다.

멜 스펙트로그램 함수 인자에 임의의 값(일반적으로 쓰는 값들 중에 임의로 선택)을 설정하고 모델 학습을 진행했을 때의 학습데이터 정확도는 98%정도로 높았지만, 테스트 데이터셋의 정확도 결과는 좋지 못했다. 이는 학습이 적절히 이루어지고 있지 않다는 증거이다. 학습률은 다른 동적 방법으로 조절하기로 하고 멜 스펙트로그램의 함수 인자 자체의 파라미터 값을 찾는 게 더 중요하다고 판단했다.

멜 스펙트로그램의 parameter 중 바꿀 수 없는 인자는 제외하고 n_fft, hop_length, n_mels 를 대상으로 하이퍼 파라미터 튜닝을 진행했다. 'n_fft'에 쓰이는 값들은 [1024, 2048]가 주로 쓰인다. 또, 'hop_length'에는 [256, 512]를 표본으로 설정했다. 'n_mels'의 표본은 [128, 256]로 설정했다.

하이퍼 파라미터 튜닝 단계에선 GTZAN 데이터 세트의 절반만을 이용했다. 하나의 파라미터 조합을 학습시켜서 정확도와 오차를 알아내는데 걸리는 시간만 5 시간 이상이고, 오차의 차이만 확인하면 되기 때문에 원본데이터의 수를 더 추가할 필요는 없다고 판단했다. 같은 이유로 이미지의 크기 또한 256*128 로 줄였다. 학습 모델은 VGG16 을 사용했고 epoch 는 이전에 여러 번 실험한 결과를 바탕으로 20 정도면 오차의 차이를 판단할 수 있다고 생각해서 설정했다. 테스트셋은 따로 만들지 않고 model.fit 단계에서 가장 검증셋의 오차가 높게 나온 조합을 채택하기로 한다.

GradCAM 이란 XAI 의 일종으로 이미지의 어떤 부분이 분류에 큰 역할을 했는지 시각적으로 보여준다. 분류에 큰 영향을 미친 부분은 밝게 나타나고 적게 영향을 미칠수록 파란색(보라색)으로 나타난다. 적용 예시는 다음과 같다:



GradCAM 방법은 CAM 방법의 응용으로, CAM 이란 flatten 과정 전에 특성맵들을 모아 각각 평균값을 낸 후 평균값들과 최종 예측 사이에서 한 번 더 학습을 진행시켜 어떤 특성맵이 최종 결정에 큰 역할을 하는지에 관한 가중치들을 얻은 다음 이 가중치들을 원래 특성 맵에 곱해 가중치가 큰 특성 맵들만을 남겨 이미지 분류에서 중요한 역할을 하는 특성들 만을 남기는 방법이다. GradCAM 은 평균값을 계산하기 위해 모델의 구조를 바꾸어야 하는 번거로움을 피하고자 평균값 대신 기울기를 사용하는 방법이다.

물론 음악 분류는 오디오 데이터를 전처리한 이미지를 사용하는 것이기 때문에 원본 데이터에서 어느 부분이 분류에 영향을 크게 주었냐 와는 괴리가 있을 수 있다. 정확히 알기 위해서는 이미지에서 밝게 처리된 부분의 타임라인을 찾아가서 원본 오디오 데이터를 확인해야 할 것이다.

4. 방법론 결과 및 논의

먼저 하이퍼 파라미터 튜닝 결과를 설명하겠다. 모두 8 조합이었고 위에서 썼듯 검증셋의 오차가 가장 낮은 조합을 모델학습 시 채택하기로 한다. 순서대로 'n_fft', 'hop_length', 'n_mels'가 (1) 1024 256 128 일 때, 14epoch 에서 오차는 5.4570 로 이 조합 내 최저 수치이다. (2) 1024 256 256 일 때, 12epoch 에서 오차는 4.6091 로 이 조합 내 최저 수치이다. (3) 1024 512 128 일 때, 13epoch 에서 오차는 4.6931 로 이 조합 내 최저 수치이다. (4) 1024 512 256 일 때, 4 epoch 에서 오차는 3.2920 로 이 조합 내 최저 수치이다. (5) 2048 256 128 일 때, 3 epoch 에서 오차는 4.8341 로 이 조합 내 최저 수치이다. (6) 2048 256 256 일 때, 7 epoch 에서 오차는 3.4092 로 이 조합 내 최저 수치이다. (7) 2048 512 128 일 때, 5 epoch 에서 오차는 3.7309 로 이 조합 내 최저 수치이다. (8) 2048 512 256 일 때, 3 epoch 에서 오차는 3.7798 로 이 조합 내 최저 수치이다.

이 결과를 바탕으로 'n_fft', 'hop_length', 'n_mels'의 조합이 1024 512 256 일 때 가장 학습이 효과적이었다는 결론을 낼 수 있다.

ImageDataGenerator 를 썼을 때의 결과는 예상대로 좋지 않았다.

```
val_accuracy = history.history['val_accuracy']
print(val_accuracy)
```

```
[0.3799999952316284]
```

직접 만든 CNN 모델 학습 시 배치사이즈는 코랩 세션이 끊기지 않는 최대한도인 50 으로 설정했다. 학습은 20 에포크에서 더 이상 val_loss 가 줄지 않아 자동 중단되었다. 테스트셋으로 확인해 본 설계 모델의 정확도는 다음과 같았다:

```
#설계 모델 정확도
result=model.evaluate(x_test,y_test)
print("accuracy:",round(result[1]*100,0),"%")
```

```
5/5 [==============================] - 1s 79ms/step - loss: 2.1003 - accuracy: 0.4643
accuracy: 46.0 %
```

설계 모델 학습률 조정후(ReduceLROnPlateau 사용) 결과는:

```
Epoch 24/50
23/23 [==============================] - 7s 303ms/step - loss: 0.4701 - accuracy: 0.8411 - val_loss: 1.7430 - val_accuracy: 0.4683 - lr: 1.0000e-04
```

```
#설계 모델 정확도
result=model.evaluate(x_test,y_test)
print("accuracy:",round(result[1]*100,0),"%")
```

```
5/5 [==============================] - 1s 91ms/step - loss: 1.6982 - accuracy: 0.5071
accuracy: 51.0 %
```

VGG16 모델 적용 시 결과는

```
Epoch 30/30
21/21 [==============================] - ETA: 0s - loss: 1.5197 - accuracy: 0.4568
Epoch 30: val_loss did not improve from 1.71482
21/21 [==============================] - 10s 489ms/step - loss: 1.5197 - accuracy: 0.4568 - val_loss: 1.7657 - val_accuracy: 0.3452
```

```
[4] result=V_model.evaluate(x_test,y_test)
    print("accuracy:",round(result[1]*100,0),"%")

    5/5 [==============================] - 8s 741ms/step - loss: 1.9418 - accuracy: 0.2786
    accuracy: 28.0 %
```

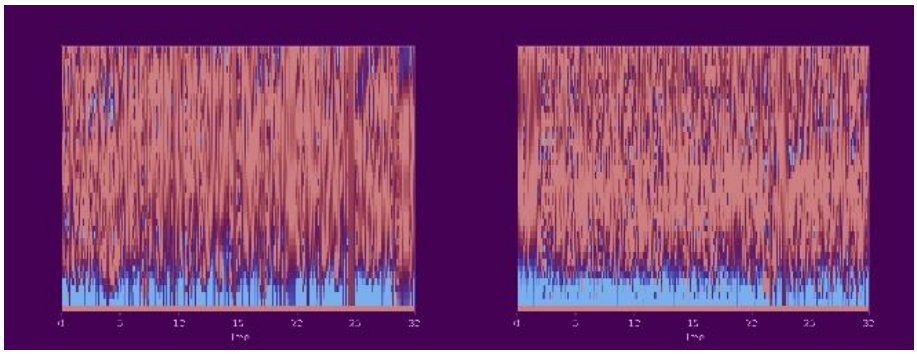
VGG16 모델 학습률 조정후(ReduceLROnPlateau 사용) 결과는 다음과 같다:

```
Epoch 49: val_loss did not improve from 1.75564
21/21 [==============================] - 10s 504ms/step - loss: 1.5028 - accuracy: 0.4429 - val_loss:
Epoch 50/50
21/21 [==============================] - ETA: 0s - loss: 1.4561 - accuracy: 0.4727
Epoch 50: val_loss did not improve from 1.75564
21/21 [==============================] - 10s 495ms/step - loss: 1.4561 - accuracy: 0.4727 - val_loss:
```

```
4] result=V_model.evaluate(x_test,y_test)
   print("accuracy:",round(result[1]*100,0),"%")

   5/5 [==============================] - 8s 741ms/step - loss: 1.6004 - accuracy: 0.3714
   accuracy: 37.0 %
```

GradCAM 적용 후 결과는 다음과 같다:



-의미가 있는 결과라고 보기 힘들다. 아무래도 이 이미지는 사물이 아니고, 시간에 따라 연속성을 가진 그래프이기에 어느 한 부분을 짚어서 결과에 주요한 영향을 미친 부분이라고 하기 힘들다.

멜 스펙트로그램만을 활용한 음성분류도 실제로 사용되고 정확도도 높은 편이다. 참고로 진행해 본 멜 스펙트로그램만을 사용한 분류 결과는 다음과 같다:

```
result=V_model.evaluate(x_test,y_test)
print("accuracy:",round(result[1]*100,0),"%")
```

```
4/4 [==============================] - 1s 244ms/step - loss: 0.9182 - accuracy: 0.6900
accuracy: 69.0 %
```

5. 연구 결과 및 정리

MFCC 는 멜 스펙트로그램에 Cepstral 분석을 통해 추출된 값(계수)들로, 멜 스펙트로그램의 특징들의 작은 집합을 축약해서 보여준다. 다시 말하면, MFCC 는 음성 신호의 스펙트로그램 정보를 보다 의미 있는 형태로 압축하는 역할을 한다. 어떤 스펙트로그램 영역 내 특징을 잘 뽑아낸다는 점에서 멜 스펙트로그램보다 발전된 방법이지만, 전체적인 오디오 특성이 작은 특성들까지 반영해야 한다면, 원본인 멜 스펙트로그램을 이용하는 것이 좋을 수 있다.

이는 오디오 분석에는 음성 인식(자연어 처리), 음악분류, 도시 내 소음 감지 등 여러 분야가 있기 때문이다. 따라서 필요에 따라 다른 방법론들을 사용해야 한다. 실험적으로 진행해 본 멜 스펙트로그램 결과가 더 좋았던 것을 보면, 음악 분류 분야에서는 MFCC 보다 멜 스펙트로그램이 적절하다고 할 수 있다. 선행 연구들을 보면 MFCC 는 자연어 처리 분야에 더 자주 쓰이는 듯하다.

VGG16 보다 직접 만든 모델의 결과가 좋았던 것은 놀라운 점이다. VGG16 의 이미지 분류 능력은 잘 알려져 있다. 이러한 결과는 VGG16 의 학습 대상은 일반 사물들이어서 사람이 알아보기도 난해한 그래프 이미지 인식에는 효과가 적었던 것으로 해석될 수 있다.

프로젝트 구상 시점에는 학습률 조정은 모델 학습 결과에 별로 영향을 미치지 않을 것이라고 생각했다. 하지만 학습률 조정을 거치며 정확도가 약 8%정도 상승하며 큰 변화가 생겼다. 이는 학습률을 검증셋 변화에 맞게 조절하여 학습 모델 성능이 향상되지 않으면 학습률을 감소시켜 모델 수렴이 빠르도록 도왔기 때문이다. 만약 일정한 학습률을 고수했다면, 모델은 어떤 특정 지역에서 weight 값이 진동하며 수렴에 어려움이 있었을 것이다.

최종 결과의 정확도는 51%로 클래스는 모두 11 개였다. 처음엔 이 결과 값이 낮다고 생각했지만, 음악 장르 특성상 트로트에도 클래식이 쓰이거나, 재즈 풍의 블루스 라거나, 컨트리 음악 같은 락의 경우에 장르를 명확히 구분짓는데는 어려움이 있다고 결론 내릴 수 있다. 물론 프로젝트 초반 목표는 음악 분류에 명확한 구분선을 만들자는 것이었지만, 데이터 양이 매우 적어서(각 클래스 당 약 140 개 정도의 표본) 만족할 만한 결과를 얻기 힘들었다.

또 추가로 데이터를 확보하는 과정에서 원본 데이터와의 통일성을 보장하지 못한 점도 한계라고 할 수 있다. 같은 음량 크기, 품질 등을 보장하지 못했기 때문이다.

결론으로 클래스가 11 개였던 점, 사람도 구별해내기 힘든 그래프에서 특징들을 51%확률로 가려냈다는 점에서 51%라는 성적은 결코 낮다고 볼 수 없다. 이를 통해 CNN 모델은 사람을 뛰어넘는 이미지 분류 능력이 있다는 것과 MFCC 나 멜 스펙트로그램이 오디오 분석에 유용한 방법론이라는 것을 주장할 수 있다.