

# Kaggle 프로젝트 보고서

과목 : 머신러닝

담당 교수 : 김성은 교수님

팀 : ST\_ML2025\_59

이름(학과) : 기현빈(기계시스템디자인공학과)

인공지능반도체 연계융합복수전공

# 머리말

본 보고서는 서울과학기술대학교 2025학년도 1학기 머신러닝 과목의 프로젝트 일환으로 작성되었습니다. 프로젝트의 주요 목표는 오디오 데이터를 분석하여 사람의 호흡 소리에서 '들숨(Inhale)'과 '날숨(Exhale)'을 정확하게 구분하는 이진 분류 모델을 개발하는 것입니다. 이를 위해 제공된 Kaggle 데이터셋을 활용하여 다양한 모델을 구축하고, 성능 최적화를 위한 하이퍼파라미터 튜닝 및 앙상블 학습 기법을 적용하였습니다.

이번 프로젝트에서는 CNN(Convolutional Neural Network) 모델을 활용하여, 주어진 오디오 데이터를 처리하고, 효율적인 특성 추출 및 분류 성능 향상을 목표로 하였습니다. 또한, 데이터 전처리 단계에서의 중요한 과정인 특성 추출, 패딩, 학습/검증 분할 등을 세심하게 처리하였으며, 최적의 모델을 위한 하이퍼파라미터 튜닝과 앙상블을 통해 성능을 극대화할 수 있었습니다.

이 보고서에서는 데이터셋의 구성과 전처리 과정, 사용된 예측 모델 및 성능 평가 방법에 대해 자세히 설명하며, 프로젝트 수행 과정에서의 성과와 결과를 종합적으로 다룰 것입니다. 또한, 모델 성능 향상을 위한 전략과 과정에서 겪은 시행착오 및 그 해결 방안을 함께 논의할 예정입니다.

# 목차

1. 데이터셋 설명
2. 데이터 전처리
3. 파생 feature 추출
4. 딥러닝 모델
5. 성능평가
6. 프로젝트 마무리

# 1. 데이터셋 설명

## 제공되는 파일

train/ (폴더)

모델 학습에 사용되는 .wav 형식의 오디오 파일들이 담겨 있습니다. 각 파일은 한 사람의 호흡 소리(들숨 또는 날숨)를 담고 있습니다.

test/ (폴더)

모델 성능 평가에 사용되는 .wav 형식의 오디오 파일들이 담겨 있습니다. 이 파일들에 대해 들숨/날숨을 예측해야 합니다.

train.csv

학습 세트 정보 파일입니다. train/ 폴더의 오디오 파일들에 대한 정답 레이블을 포함합니다. 이 파일을 사용하여 모델을 학습시킬 수 있습니다.

test.csv

테스트 세트 정보 파일입니다. 여러분이 예측해야 할 test/ 폴더 안의 오디오 파일들의 ID 목록입니다.

이 파일에 있는 모든 ID에 대해 예측값을 제출해야 합니다.

데이터 열(Columns) 설명

train.csv

file\_name: 오디오 파일의 고유 식별자(파일 이름)입니다. train/ 폴더의 .wav 파일과 일치합니다.

label: 해당 오디오 파일의 정답 레이블입니다.

I: 들숨 (Inhale)

E: 날숨 (Exhale)

test.csv

ID: 예측해야 할 오디오 파일의 고유 식별자입니다

## 2. 데이터 전처리

### 1. 데이터 셋 로드 전처리

```
for _, row in tqdm(train_df.iterrows(), total=len(train_df)):
    base_id = row['ID'].replace('_I_', '_').replace('_E_', '_') + '.wav'
    if base_id not in actual_files:
        continue
    file_path = os.path.join(TRAIN_DIR, base_id)
    try:
        audio, sr = librosa.load(file_path, sr=None)
        feat = extract_mel_spec_from_y(audio, sr)
        X.append(feat)
        y.append(1 if row['Target'] == 'I' else 0)
    except:
        continue
```

train\_df['ID']의 형식은 \_I\_, \_E\_와 같이 특수 구분자를 포함하고 있는데, 실제 오디오 파일명(.wav)과 매칭하기 위해 해당 구분자를 '\_'로 통일해 이름을 정규화함.

예: '20190123\_10\_42\_02\_I\_001' → '20190123\_10\_42\_02\_001.wav'

정규화된 파일명이 실제 데이터 폴더에 존재하는지 확인하고, 누락된 파일은 무시함.  
파일이 존재하는 경우 librosa.load()를 통해 오디오 데이터를 불러오고,  
사용자 정의 함수 extract\_mel\_spec\_from\_y()를 이용해 feature를 추출함.  
레이블은 Target 컬럼 기준으로 'I' → 1, 'E' → 0으로 변환해 저장함.

### 2. feature 추출

```
mfcc_feat = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)
delta = librosa.feature.delta(mfcc_feat)
delta2 = librosa.feature.delta(mfcc_feat, order=2)
full_feat = np.concatenate([mfcc_feat, delta, delta2], axis=0) # (39, time_steps)
feature_vector = np.mean(full_feat, axis=1) # → (39,)
```

오디오 신호로부터 MFCC(Mel-Frequency Cepstral Coefficients) 13차원을 추출함.  
MFCC는 사람의 청각 시스템을 모델링한 음색 정보이며, 음성인식과 음향분석에서 핵심 feature이다.

MFCC, Delta, Delta-Delta 총 39차원 feature에 대해 시간 축(axis=1) 기준으로  
평균을 취해, 고정된 길이의 1차원 벡터 (39,)로 요약한다.  
모델에 직접 입력하거나, 이후 다른 feature들과 결합 가능함.

### 3. 파생 feature 추출

#### 1. Mel-Spectrogram

시간 축과 주파수 축을 가진 2차원 표현으로, 음성신호를 이미지처럼 다룰 수 있다.  
일반적인 스펙트로그램과 달리, 사람의 청각특성(저음에 민감)을 반영한 MEL-scale을 사용했다.  
CNN계열 모델이 잘 학습할 수 있도록 시각적 구조를 제공함.

#### 2. RMS (Root Mean Square Energy)

주어진 구간의 신호 에너지 크기를 나타내며, 소리의 세기를 수치화했다.  
음소 간 볼륨차이나 무음(숨뱃.pause 등)을 감지할 수 있는 유용한 loudness특성이다.  
용도:강약이 반복되는 숨소리 등에서 유용한 패턴 제공.

#### 3. ZCR (Zero Crossing Rate)

오디오신호가 0을 기준으로 양수/음수로 바뀌는 횟수를 계산함.  
고주파(거친 숨소리)일수록 ZCR이 높게 측정된다.

#### 4. Spectral Contrast

주파수 대역에서 강한 에너지(피크)와 약한 에너지의 차이를 측정한 값이다.  
일정 주파수 대역에서의 에너지 차이를 기반으로 질감(Timbre)의 다양성을 포착할 수 있다.  
용도: 폐의 상태나 숨소리의 잡음 등에서 차이를 감지할 수 있다.

#### 5. MFCC + Delta + Delta-Delta

MFCC란 음색(Timbre) 정보를 정량적으로 표현한 벡터이다.  
Delta: 1차 차분(속도), Delta-Delta: 2차 차분(가속도)  
총 13(MFCC) x 3 = 39개의 특성 → 발성 방식, 조음변화를 잘 포착 할 수 있다.  
용도: 소리의 구조적 변화 감지(연속적인 호흡의 흐름)

### 6. 데이터 누락 처리

파일 누락, 읽기 실패 등 오류 발생 시 해당 샘플 건너뛰  
error\_count로 누락건수를 로깅하여 전체 처리 성공률을 확인했다.  
20190123\_10\_42\_02\_006.wav : 손상된 파일로, 정상적으로 로드되지않아 자동으로 제외됨.

### 7. 길이 정규화(Padding)

```
X_pad = pad_sequences(X, padding='post', dtype='float32')
y_arr = np.array(y)
return X_pad, y_arr
```

추출한 특징을 패딩하여 X\_pad로 반환하고, 레이블 배열 y\_arr를 반환한다.  
Keras에서 제공하는 pad\_sequences를 사용하여 추출한 feature를 padding을 통해  
각 오디오 파일길이를 동일하게 맞추었다.  
padding="post" : 뒤에 0을 채우는 방식(앞부분 정보손실 방지)  
keras의 pad\_sequences를 활용하고, 결과는 numpy → pytorch에서 그대로 tensor로 변환해 사용함.

## 8. 학습/검증 분할

`train_test_split(..., test_size=0.2, stratify=y_all)`

전체 데이터를 학습 80%, 검증 20%로 나누어 편향된 분할 방지

**stratify=y\_all** : 전체데이터를 80:20으로 나누되, 클래스(label) 비율을 유지하여 분할한다.

이로써 데이터 분류과정에서 모델이 한쪽클래스에 치우치지 않고 학습하고, 더욱더 공정한 평가를 할 수 있다.

## 9. 테스트셋 처리

학습과정에서 사용한 전처리 방식을 동일하게 테스트 셋에도

적용해야 입력형식이 일치하고, 모델성능평가가 정확히 가능하다.

따라서 테스트셋도 동일한 방식의 feature 추출 및 padding 수순을 따랐다.

오류 발생 시 `np.zeros((1, 112))`을 대체 입력으로 사용하여 일관된 입력 형태 유지했다.

# 4. 딥러닝 모델

### 4-1) 모델 선정 이유

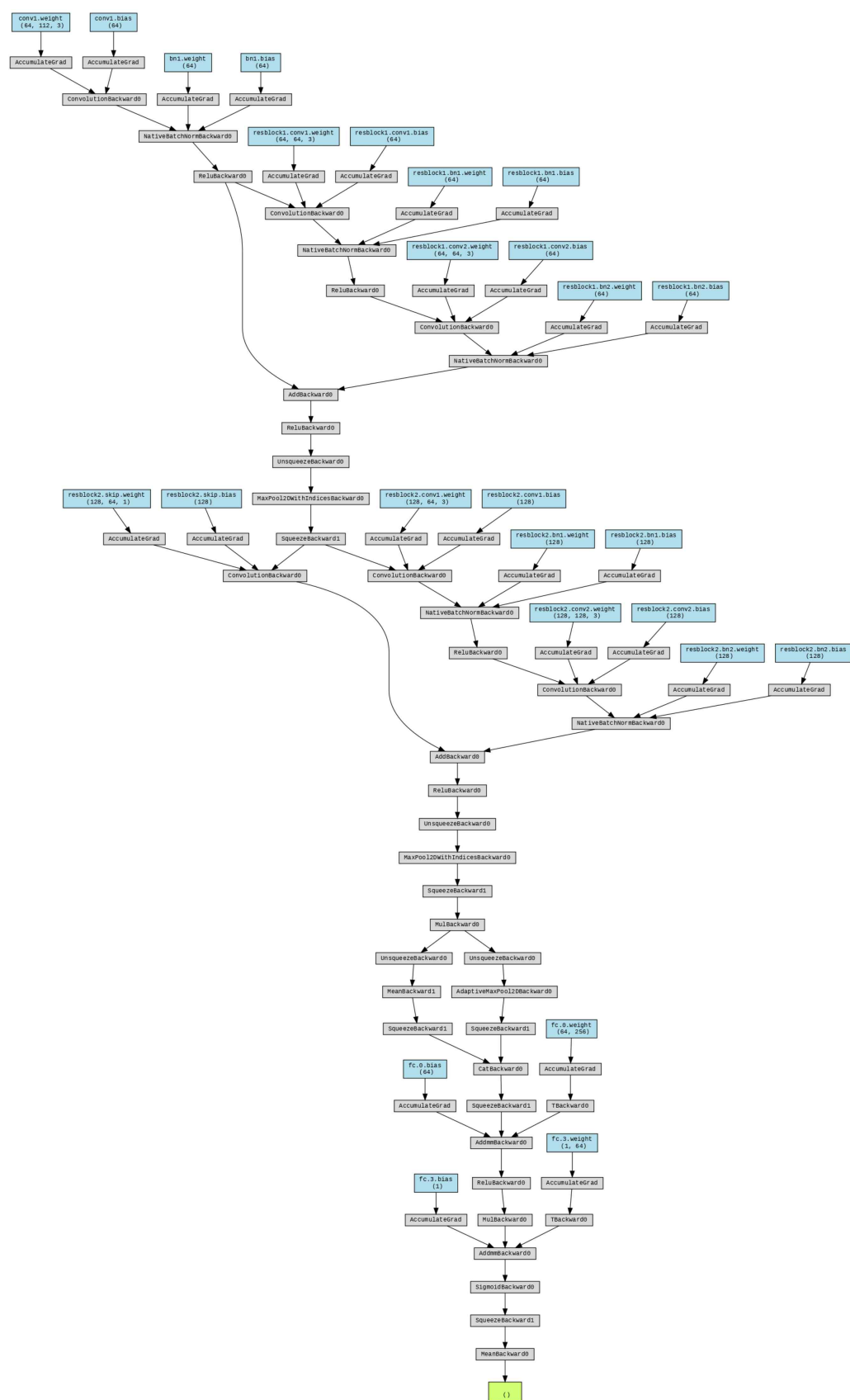
이 프로젝트에서는 CNN(Convolutional Neural Network)을 사용하여 오디오 데이터를 ‘들숨’/‘날숨’으로 이진분류하는 모델을 만들고자 한다. CNN은 주로 이미지 처리 및 영상 분석에 사용되는 딥러닝 모델이다. 모델 특성상 이미지나 비디오와 같은 2D 데이터의 공간적 패턴을 인식하는데 매우 유효하지만, 최근에는 음성 데이터와 시계열 데이터 등에도 사용되고 있다. CNN의 큰 장점은 데이터로부터의 매우 효과적인 특성 추출 능력에 있다. CNN모델 설계의 중점은 feature 설정 보다는 CNN의 구조를 쌓는 것에 있다. CNN의 주요 구성요소는 Convolution layer, Pooling layer, Activation Function , Residual layer 등으로 특히 overfitting을 완화하기 위한 방법에 중점을 두었다.

### 4-2) 모델 구조

layer	특징
Convolution1d + BN + ReLU	입력 feature(112차원)를 64채널로 변환
ResidualBlock(64 → 64)	깊이 유지하면서 학습 안정성 확보
MaxPool1d	시계열 길이 압축
ResidualBlock(64 → 128)	더 복잡한 feature 추출
MaxPool1d	추가 시계열 압축
Dropout	과적합 방지
GAP + GMP	Global Average/Max Pooling → flatten
FC Layer	256 → 64 → 1 → sigmoid 이진 출력

본 프로젝트에서 설계한 레이어의 대략적 구조와 특성이다.

torchviz 라이브러리를 이용한 시각화는 다음과 같다.





#### 4-3) 모델 구조 성능 향상 과정

##### 1. Residual layer의 추가를 통한 overfitting 완화

초기 CNN 모델

layer	특징
Convolution1d + BatchNorm + ReLU	입력feature(112차원)을 64채널로 변환
MaxPool1d	시계열 길이 압축
Conv1d(64 → 128)	채널 수 증가로 더 복잡한 패턴 학습
MaxPool1d	추가적인 시계열 길이 압축
Conv1d(128 → 256)	깊이 있는 학습
MaxPool1d	추가적인 시계열 길이 압축
AdaptiveAvgPool1d(1)	전역 평균 풀링 시계열 길이를 1로 줄임
Linear(256 → 64)	완전 연결
Dropout(0.5)	정규화 과적합 방지
Linear(64 → 1)	출력: 시그모이드 활성화로 이진 분류 (I/E)

위와 같이 모델을 구성한 결과, accuracy는 일정 수준에서 정체되었으며,

이는 다양한 데이터를 일반화하지 못하는 overfitting 현상이 발생한 것으로 해석된다.

이를 완화하기 위해 **Residual Block**을 모델에 도입하였다.

Residual Block은 입력값을 연산 결과에 더하는 skip connection 구조를 가지며,

주요 정보의 흐름을 보존하면서도 불필요한 특징에 대한 과적합을 방지하는 데 도움을 준다.

이러한 구조는 모델이 중요한 특징만을 학습하도록 유도하며,

학습 안정성과 일반화 성능을 향상시키는 데 기여하였다.

##### <초기 모델의 overfitting 결과>

기본 제공 data를 통한 성능(accuracy): 0.7688  
kaggle 제출 기준 성능(accuracy): 0.722

##### 2. hyper parameter 튜닝을 통한 성능 향상

###### # step 별 시도와 시행착오

###### step 1. threshold 조정

초기설계과정에서 최적의 threshold만을 찾아 성능을 향상시키고자 하였다.

이 과정에서는 F1-score를 기준으로 threshold 값을 찾았고, 그 결과 성능이 다소 향상되었다.

그러나 하이퍼파라미터 튜닝 없이 threshold만 조정하는 것은 성능 향상에 큰 영향을 미치지 않았다.

```
best_th = 0.0
best_f1 = 0.0
```

```
for th in np.arange(0.3, 0.7, 0.01):
    preds = (y_probs > th).astype(int)
    score = f1_score(y_true, preds)
    if score > best_f1:
        best_f1 = score
        best_th = th
```

```
Epoch [1/20], Train Loss: 0.7476, Val Loss: 0.6753, Train Acc: 0.5711, Val Acc: 0.5825
Epoch [2/20], Train Loss: 0.6374, Val Loss: 0.6411, Train Acc: 0.6499, Val Acc: 0.6312
Epoch [3/20], Train Loss: 0.6102, Val Loss: 0.6033, Train Acc: 0.6765, Val Acc: 0.6575
Epoch [4/20], Train Loss: 0.6000, Val Loss: 0.5955, Train Acc: 0.6871, Val Acc: 0.6763
Epoch [5/20], Train Loss: 0.5749, Val Loss: 0.6075, Train Acc: 0.7074, Val Acc: 0.6800
Epoch [6/20], Train Loss: 0.5400, Val Loss: 0.6303, Train Acc: 0.7302, Val Acc: 0.6600
Epoch [7/20], Train Loss: 0.5393, Val Loss: 0.6132, Train Acc: 0.7393, Val Acc: 0.6600
Epoch [8/20], Train Loss: 0.5164, Val Loss: 0.5435, Train Acc: 0.7474, Val Acc: 0.7312
Epoch [9/20], Train Loss: 0.5104, Val Loss: 0.8868, Train Acc: 0.7540, Val Acc: 0.5587
Epoch [10/20], Train Loss: 0.4997, Val Loss: 0.5975, Train Acc: 0.7627, Val Acc: 0.6975
Epoch [11/20], Train Loss: 0.4758, Val Loss: 0.5566, Train Acc: 0.7781, Val Acc: 0.7212
Epoch [12/20], Train Loss: 0.4537, Val Loss: 0.5894, Train Acc: 0.7868, Val Acc: 0.7250
Epoch [13/20], Train Loss: 0.4490, Val Loss: 0.6046, Train Acc: 0.7934, Val Acc: 0.7275
Epoch [14/20], Train Loss: 0.4475, Val Loss: 0.5587, Train Acc: 0.7987, Val Acc: 0.7338
Epoch [15/20], Train Loss: 0.4471, Val Loss: 0.5600, Train Acc: 0.7959, Val Acc: 0.7175
Epoch [16/20], Train Loss: 0.4197, Val Loss: 0.5216, Train Acc: 0.8112, Val Acc: 0.7612
Epoch [17/20], Train Loss: 0.4163, Val Loss: 0.5305, Train Acc: 0.8118, Val Acc: 0.7600
Epoch [18/20], Train Loss: 0.4146, Val Loss: 0.5557, Train Acc: 0.8215, Val Acc: 0.7388
Epoch [19/20], Train Loss: 0.4086, Val Loss: 0.5115, Train Acc: 0.8249, Val Acc: 0.7650
Epoch [20/20], Train Loss: 0.3997, Val Loss: 0.5136, Train Acc: 0.8240, Val Acc: 0.7662
```

🔍 Best Threshold: 0.34, Best F1-score: 0.7832

	precision	recall	f1-score	support
0.0	0.85	0.60	0.70	395
1.0	0.70	0.90	0.78	405
accuracy			0.75	800
macro avg	0.77	0.75	0.74	800
weighted avg	0.77	0.75	0.74	800

🔗 Final Accuracy with Best Threshold (0.34): 0.7488

## 주안점

하이퍼 파라미터를 최적화 하지않은 채 단순 best threshold만을 찾는 것은 성능향상에 도움이 되지 않았다. 이에 하이퍼파라미터 튜닝을 한 후 best threshold를 찾는 것이 성능향상에 더 효과적일 것이라고 예상함.

## step 2. 기본 hyper parameter & 모델 구조 hyper parameter 동시 튜닝

#기본 hyperparameter : dropout\_rate, lr, weight\_decay, gamma

#모델 구조 hyper parameter : n\_mels, kernel\_size, hidden\_dim

```
def objective(trial):
    dropout_rate = trial.suggest_float("dropout_rate", 0.3, 0.7)
    lr = trial.suggest_float("lr", 1e-4, 1e-2, log=True)
    weight_decay = trial.suggest_float("weight_decay", 1e-6, 1e-2, log=True)
```

```

gamma = trial.suggest_float("gamma", 0.3, 0.9)
n_mels = trial.suggest_categorical("n_mels", [32, 48, 64, 80, 96, 128])
kernel_size = trial.suggest_categorical("kernel_size", [3, 5, 7])
hidden_dim = trial.suggest_categorical("hidden_dim", [32, 64, 128])
🔗 Best Params: {'dropout_rate': 0.3161628959039737, 'lr': 0.0002780309918774388, 'weight_decay':
0.00015947443251787687, 'gamma': 0.378404862404653, 'n_mels': 64, 'kernel_size': 5, 'hidden_dim': 64}
🔗 Best Threshold: 0.35000000000000003
🔗 Best F1: 0.7918088737201365

```

```

Epoch 1 - Train Loss: 0.6477, Val Loss: 0.5890, Val Acc: 0.6950
Epoch 2 - Train Loss: 0.5620, Val Loss: 0.5632, Val Acc: 0.7113
Epoch 3 - Train Loss: 0.5250, Val Loss: 0.5383, Val Acc: 0.7250
Epoch 4 - Train Loss: 0.4962, Val Loss: 0.5470, Val Acc: 0.7087
Epoch 5 - Train Loss: 0.4699, Val Loss: 0.6106, Val Acc: 0.6850
Epoch 6 - Train Loss: 0.4259, Val Loss: 0.5056, Val Acc: 0.7525
Epoch 7 - Train Loss: 0.4007, Val Loss: 0.4865, Val Acc: 0.7638
Epoch 8 - Train Loss: 0.3814, Val Loss: 0.5087, Val Acc: 0.7288
Epoch 9 - Train Loss: 0.3586, Val Loss: 0.4811, Val Acc: 0.7625
Epoch 10 - Train Loss: 0.3506, Val Loss: 0.5707, Val Acc: 0.7188
Epoch 11 - Train Loss: 0.3052, Val Loss: 0.4864, Val Acc: 0.7650
Epoch 12 - Train Loss: 0.2845, Val Loss: 0.4795, Val Acc: 0.7725
Epoch 13 - Train Loss: 0.2793, Val Loss: 0.4814, Val Acc: 0.7725
Epoch 14 - Train Loss: 0.2692, Val Loss: 0.4789, Val Acc: 0.7675
Epoch 15 - Train Loss: 0.2545, Val Loss: 0.4898, Val Acc: 0.7588
Epoch 16 - Train Loss: 0.2351, Val Loss: 0.4858, Val Acc: 0.7588
Epoch 17 - Train Loss: 0.2341, Val Loss: 0.4858, Val Acc: 0.7675
Epoch 18 - Train Loss: 0.2344, Val Loss: 0.4835, Val Acc: 0.7662
Epoch 19 - Train Loss: 0.2229, Val Loss: 0.4896, Val Acc: 0.7562
Epoch 20 - Train Loss: 0.2163, Val Loss: 0.4860, Val Acc: 0.7700
Epoch 21 - Train Loss: 0.2163, Val Loss: 0.4874, Val Acc: 0.7688
Epoch 22 - Train Loss: 0.2151, Val Loss: 0.4862, Val Acc: 0.7625
Epoch 23 - Train Loss: 0.2135, Val Loss: 0.4866, Val Acc: 0.7662
Epoch 24 - Train Loss: 0.2120, Val Loss: 0.4877, Val Acc: 0.7638
🛑 Early stopping at epoch 24
🔍 Best threshold: 0.30, F1: 0.7880

```

### 주안점(major problem)

기존 계획대로 최적의 하이퍼파라미터 조합을 찾는다는 성공했다.

하지만 모델구조 하이퍼파라미터 **kernel\_size**, **hidden\_dim**, **n\_mels** 는 모델의 필터 사이즈를 바꾸기 때문에, 추후 앙상블 과정에서 각 모델마다 입출력 크기를 맞춰주는 것이 매우 어려웠다.

앙상블을 목표로 파라미터 후보군들을 찾았지만, 앙상블 시도 과정에서 각 모델의 입출력 사이즈를 동일하게 맞추는 것이 매우 복잡한 작업이란걸 알게 되었고, 대안점을 모색했다.

### step 3. 기본 hyper parameter 튜닝

#### #튜닝 파라미터 종류 축소

결국, 모델 구조는 그대로 유지한 채 weight\_decay, gamma, dropout\_rate, lr 만을 변경하며 최적화 모델을 찾고자 했다. 이번튜닝은 모델구조를 그대로 유지한 채 학습률, 규제정도 등 국소적 부분만 변경하므로 앙상블을 시도할 때 기존의 복잡했던 문제를 쉽게 해결할 수 있을 것이라 생각했다.

optuma 60 으로 많은 후보군집단을 탐색하였다.

Best Threshold: 0.3500000000000000

🔍 Best Params: {'dropout\_rate': 0.5320021982413212, 'lr': 0.0005454765231375181, 'weight\_decay': 0.0002854022873111989, 'gamma': 0.5446879190443025}

Epoch 1 - Train Loss: 0.6787, Val Loss: 0.6348, Val Acc: 0.6713  
Epoch 2 - Train Loss: 0.6188, Val Loss: 0.6289, Val Acc: 0.6300  
Epoch 3 - Train Loss: 0.5791, Val Loss: 0.5744, Val Acc: 0.7137  
Epoch 4 - Train Loss: 0.5498, Val Loss: 0.5846, Val Acc: 0.6813  
Epoch 5 - Train Loss: 0.5331, Val Loss: 0.5740, Val Acc: 0.6963  
Epoch 6 - Train Loss: 0.4969, Val Loss: 0.5436, Val Acc: 0.7087  
Epoch 7 - Train Loss: 0.4624, Val Loss: 0.5234, Val Acc: 0.7362  
Epoch 8 - Train Loss: 0.4428, Val Loss: 0.5537, Val Acc: 0.7037  
Epoch 9 - Train Loss: 0.4279, Val Loss: 0.5126, Val Acc: 0.7475  
Epoch 10 - Train Loss: 0.4043, Val Loss: 0.5166, Val Acc: 0.7488  
Epoch 11 - Train Loss: 0.3912, Val Loss: 0.4864, Val Acc: 0.7600  
Epoch 12 - Train Loss: 0.3602, Val Loss: 0.5313, Val Acc: 0.7275  
Epoch 13 - Train Loss: 0.3550, Val Loss: 0.5253, Val Acc: 0.7350  
Epoch 14 - Train Loss: 0.3318, Val Loss: 0.5122, Val Acc: 0.7462  
Epoch 15 - Train Loss: 0.3187, Val Loss: 0.4842, Val Acc: 0.7700  
Epoch 16 - Train Loss: 0.2960, Val Loss: 0.5024, Val Acc: 0.7562  
Epoch 17 - Train Loss: 0.2783, Val Loss: 0.4720, Val Acc: 0.7850  
Epoch 18 - Train Loss: 0.2789, Val Loss: 0.5055, Val Acc: 0.7488  
Epoch 19 - Train Loss: 0.2631, Val Loss: 0.4844, Val Acc: 0.7762  
Epoch 20 - Train Loss: 0.2499, Val Loss: 0.4904, Val Acc: 0.7675  
Epoch 21 - Train Loss: 0.2199, Val Loss: 0.4808, Val Acc: 0.7738  
Epoch 22 - Train Loss: 0.2187, Val Loss: 0.4843, Val Acc: 0.7800  
Epoch 23 - Train Loss: 0.2156, Val Loss: 0.5026, Val Acc: 0.7538  
Epoch 24 - Train Loss: 0.2085, Val Loss: 0.4875, Val Acc: 0.7750  
Epoch 25 - Train Loss: 0.2077, Val Loss: 0.4870, Val Acc: 0.7837  
🔍 Best threshold: 0.37, F1: 0.7972

#### step 4. 앙상블학습 및 후보 모델 선택

하이퍼 파라미터 튜닝 탐색을 성공적으로 마친 후 추후 앙상블할 때 우수한 조합을 선정하였다.  
선정기준은 모델 성능이 우수하고, 개별 모델이 서로다른 파라미터 조합을 갖추도록  
선택했다. 앙상블 기법은 개별트리들이 서로 다른 파라미터 조합을 가지고 있을수록  
서로를 보완해서 더욱 우수한 성능향상을 이끌어 낼 수 있기 때문이다. (diversity 보장)

ensemble parameter 후보군 : trial 8, trial 12, trial 54을 각각 pt파일로 저장.

##### 4-1. 3-model ensemble (soft voting)

3개 trial을 모두 앙상블해보았다.

leaderboard public score: 0.77428

##### 주안점

확실히 이전모델들에 비해 성능이 향상된 것을 경험했다.

하지만 예상보다 성능 향상의 폭이 크지는 않았다.

세 모델 중 Trial 12 모델 성능이 다른 두 모델에 비해 상대적으로 낮은 F1 score 0.77을 기록하였고,  
이는 다른 두 모델의 F1 score 0.783과 비교하여 다소 부족한 결과였다.

이 결과를 바탕으로, 하이퍼 파라미터의 다양성은 보장됐지만 개별모델의 성능이 나머지 2개에 비해 떨

어지기 때문에 오히려 **노이즈**로 작용했을 수도 있다는 의문점을 가졌다.

#### 4-2. 2-model ensemble (soft voting)

개별모델 F1 score도 **0.78**로 우수하고 하이퍼파라미터 조합의 **diversity**도 보장됐기 때문에 2개 모델로 앙상블했을 때 시너지가 좋을 것이라 예상했음. (**일반화 & 앙상블 효과 기대**)

```
# Soft Voting (2-Model Ensemble: Trial 8 + Trial 54)
ensemble_settings = [
    {"model_path": os.path.join(kaggle_data, "trial_8.pt"), "dropout_rate": 0.5320},
    {"model_path": os.path.join(kaggle_data, "trial_54.pt"), "dropout_rate": 0.3638},
]

ensemble_probs = []
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
X_test_tensor = X_test_tensor.to(device)

for setting in ensemble_settings:
    model = CNNResSimple(input_dim=112, dropout_rate=setting['dropout_rate']).to(device)
    model.load_state_dict(torch.load(setting['model_path'], map_location=device))
    model.eval()
    y_test_probs = []
    with torch.no_grad():
        for i in range(0, len(X_test_tensor), 64):
            batch = X_test_tensor[i:i+64]
            preds = model(batch)
            y_test_probs.extend(preds.cpu().numpy())
    ensemble_probs.append(np.array(y_test_probs))
```

#### 주안점

2model ensemble 결과 leaderboard public score **0.780**을 달성했다.

trial 12를 제외하고 나머지 2개 모델만을 앙상블한게 성능향상에 큰 기여를 했다고 분석했다.  
앞선 trial 12의 경우 예상한대로 앙상블에서 오히려 **noise**로 작용한 것이 아닌지 의심된다.

#### 4-3. 2-model ensemble (weighted soft voting)

기존의 soft voting은 두 모델의 판단을 단순히 평균내서 데이터를 구분하는 방식이었다.  
그래서 weighted soft voting으로 두 모델의 **판단 가중치**를 다르게 주면  
성능이 좀 더 향상될 수 있을지 궁금했다.

기존 2-model ensemble에서 **모델 가중치 비율만 Optuna를 통해 조정**해서 최적값을 찾기로 했다.  
결과적으로 Optuna를 사용해 **4:6** 비율로 판단하는 것이 검증 성능이 가장 높게 나왔다.  
하지만 Kaggle public test에서는 오히려 성능이 떨어진 것을 경험했다.  
앞선 3-model ensemble보다도 더 낮은 성능을 기록했다.  
이는 아마도 검증 데이터에 **overfitting**된 상황일 것이라 추측한다.

# cf. 3-model ensemble : **0.77428** / soft\_weighted ensemble: **0.77285**

## 5. 최종제출 ( public & private leaderboard score)

kaggle leaderboard에는 자동으로 가장 높은 public score을 기록한 모델이 최종등록되기 때문에 2-model ensemble이 선택되었다.

#public score : 0.78 → 22등으로 꽤 준수한 성능을 기록했다.

#private score : 0.766 → 5등

key\_point: ensemble을 통한 강한 일반화

## 5. 성능 평가




주 성능평가요소는 **F1 Score**를 사용하였다. F1 Score는 이진 분류 문제에서 모델의 성능을 평가하기 위한 지표 중 하나로, 정밀도(Precision)와 재현율(Recall)의 조화 평균(Harmonic Mean)을 제공한다. 이는 특히 데이터 불균형이 심한 경우, 단순히 정확도(Accuracy)만을 사용하는 것보다 유용하다.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

### <프로젝트 결과분석>

**Weighted soft voting** 모델은 public score에서 가장 성능이 저조했지만, 실제 test score를 종합했을 때 **0.783**이라는 가장 높은 정확도를 기록했다. 반면, 내가 제출한 2model\_ensemble의 test 성능은 가장 저조했음을 알게 되었다. weighted soft voting의 public score가 낮게 나와서 overfitting을 예상했지만, 실제 테스트 데이터셋에는 더 잘 맞아떨어졌다고 생각한다.

Private test leaderboard 순위는 다음과 같다.

Submission and Description	Private Score ①	Public Score ①	Selected
 <b>submission_ensemble_weighted.csv</b> Complete · Kihyunbin · 10d ago	<b>0.78333</b>	<b>0.77285</b>	<input type="checkbox"/>
 <b>submission_ensemble_2models.csv</b> Complete · Kihyunbin · 11d ago	<b>0.76666</b>	<b>0.78000</b>	<input type="checkbox"/>
 <b>submission_ensemble.csv</b> Complete · Kihyunbin · 11d ago	<b>0.77000</b>	<b>0.77428</b>	<input type="checkbox"/>

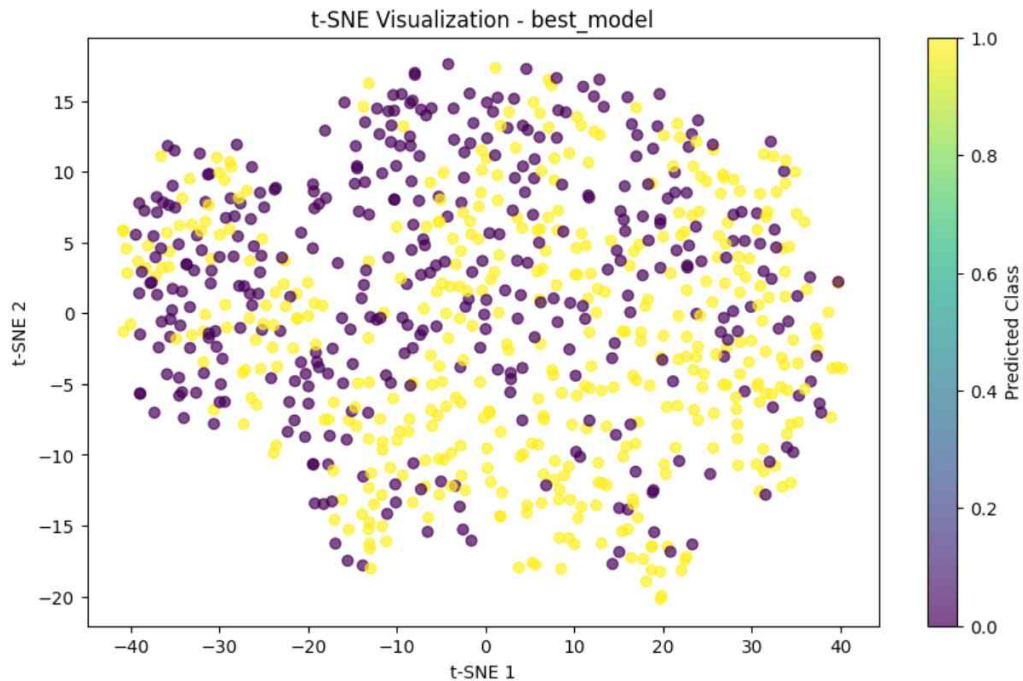
결국 딥러닝에서 가장 중요한 point는 미지의 데이터에 대한 일반화 라고 정리할 수 있을 것 같다.

머신러닝에 비해 입력 데이터에 따른 성능 차이가 엄청 크다는 걸 실감했다.

결과적으로, 기존의 **weighted 2-model ensemble 모델**이 가장 좋은 성능을 낸 걸 알게 되었다.

초기 의문점대로 단순 평균보다는 적당한 가중치를 조절해서 최적의 판단을 하는 것이 가장 좋은 성능을 자랑한다는 것을 이번 프로젝트를 하며 배웠다.

<T-SNE graph 첨부>



<배움의 자세>

**\*양상블 모델의 한계:** 양상블은 무조건 많은 모델을 적용한다고 좋은 게 아니라는 걸 알았다. 개별 성능이 확실한 모델들끼리 양상블하고 파라미터 조합은 폭 넓게 가져가는 것이 가장 좋은 무기가 될 것이다.

**\*하이퍼파라미터 튜닝의 중요성:** 1차 프로젝트와 달리, 2차 프로젝트 (딥러닝 계열)에서는 하이퍼파라미터만 조금 바뀌도 성능차이가 엄청났다. 머신러닝에서는 **feature engineering**의 중요성을 실감한 반면 딥러닝에서는 **hyper parameter tuning**의 중요성을 실감했다.

**\*세심한 코딩:** 높은 성능을 얻으려면 세심한 코딩이 정말 중요하다는 걸 깨달았다. 모두가 **chat-gpt**에 의존하는 상황에서도 결국 자신만의 경험과 노하우로 작은 차이를 만들어 성능을 비약적으로 발전시킬 수 있다는 것을 배웠다.

## 6. 프로젝트 마무리

프로젝트를 마치며..

앞선 1차 프로젝트와 더불어 이번학기 머신러닝 과목은 2개의 과정을 통해 개인의 머신러닝 통찰력을 크게 향상시키는 계기가 됐다. 여러 시행착오 속에서도 포기하지 않고 오직 좋은 성능을 가진 알고리즘을 만들기 위해 새벽같이 일어나는 스스로의 모습을 보며 오랜만에 뿌듯함을 느꼈다.

앞으로도 계속해서 도전하는 서울과기대 학생이 되고싶다.

