VOICE EMOTION DETECTION

Introduction:

The purpose of this report is to explore the implementation of a hybrid deep learning model capable of **recognizing emotions** in human speech by analyzing the **acoustic characteristics of the audio signals**. The project employs Python for audio data handling, extraction of temporal and spectral features, design of deep learning models, and performance evaluation.

Manually classifying emotions in voice recordings is time-consuming, inconsistent, and lacks scalability. This project addresses the challenge by developing an intelligent system that can automatically detect emotions such as anger, happiness, sadness, fear, disgust, and neutral through audio analysis. .

The emotion classifier is built using a **hybrid neural network architecture**, combining **Convolutional Neural Networks (CNNs)** for capturing local spectral patterns from features like MFCCs, and **Bidirectional Long Short-Term Memory (BiLSTM) networks** for learning temporal dependencies across time. This combination enables the model to understand both short-term spectral changes and long-term emotional dynamics in speech.

The model is first trained on Training set and then tested over unseen Testing set and is evaluated using appropriate performance metrics such as **accuracy**, **precision**, **recall**, and **F1-score**.

Dataset: CREMA-D Emotional Speech Dataset

For this project, the dataset used for **Voice Emotion Detection** is the widely recognized ***CREMA-D (Crowd-Sourced Emotional Multimodal Actors Dataset)***. Developed by researchers at New York University, CREMA-D has become a standard benchmark in the field of speech-based emotion recognition.

Dataset Source:

The CREMA-D Emotional Speech Dataset was obtained from Kaggle (URL: https://www.kaggle.com/datasets/ejlok1/cremad)

Key Features of CREMA-D Dataset:

- **Emotion Diversity:** The dataset includes six distinct emotional expressions that reflect a wide range of human vocal emotions:
  - Angry
  - Disgust
  - Fear
  - Happy
  - Neutral
  - Sad
- **Audio Samples:** CREMA-D contains a total of 7,442 audio clips performed by 91 professional actors (48 male and 43 female), delivering 12 standardized sentences in different emotional tones and intensities.
- **Speaker Variation:** The dataset includes voices from a diverse group of actors, varying in age, gender, and ethnicity, helping the model generalize better across different types of speakers.
- **Intensity Levels:** Emotions are expressed at two intensity levels (Low and High), allowing the model to capture subtle variations in emotion expression.
- **Format and Quality:** All audio files are in WAV format, sampled at a consistent rate of 16 kHz, ensuring clarity and consistency in speech quality.

Importing the Necessary Libraries

```
import pandas as pd
import numpy as np

import os
import sys
import librosa
import librosa.display
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.model_selection import train_test_split

from IPython.display import Audio

import tensorflow as tf
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Flatten, Dropout, BatchNormalization
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.losses import CategoricalCrossentropy
from tensorflow.keras.layers import Bidirectional, LSTM, Dense
from tensorflow.keras.regularizers import l2
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers import Nadam
from sklearn.preprocessing import StandardScale
import warnings
if not sys.warnoptions:
    warnings.simplefilter("ignore")
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

```
2025-07-18 03:41:42.516950: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:477] Unable to register cuFFT factory: Attempting
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1752810102.885817      36 cuda_dnn.cc:8310] Unable to register cuDNN factory: Attempting to register factory for plugin cuDN
E0000 00:00:1752810102.991637      36 cuda_blas.cc:1418] Unable to register cuBLAS factory: Attempting to register factory for plugin cu
```

Pandas / NumPy: For data manipulation.

OS / SYS: File and system-level operations.

Librosa: For loading and processing audio data.

Seaborn / Matplotlib: Data visualization.

Scikit-learn: For preprocessing and evaluation.

IPython Display: To play audio files inside notebook.

TensorFlow / Keras: Deep learning library used to build the emotion classification model.

Warnings: Hides deprecation warnings to keep output clean.

Data Preprocessing and Cleaning:

Downloading DataSet:

```
import kagglehub

# Download CREMA-D dataset
path = kagglehub.dataset_download("ejlok1/cremad")

print("Path to dataset files:", path)
```

```
Path to dataset files: /kaggle/input/cremad
```

Downloads the CREMA-D dataset from Kaggle using KaggleHub.

Accessing Raw Audio Files

```
import os

# List first few audio files
audio_dir = os.path.join(path, "AudioWAV")  # typical folder for CREMA-D
```

```
print("Total files:", len(os.listdir(audio_dir)))
print("Sample files:", os.listdir(audio_dir)[:5])
```

```
Total files: 7442
Sample files: ['1028_TSI_DIS_XX.wav', '1075_IEO_HAP_LO.wav', '1084_ITS_HAP_XX.wav', '1067_IWW_DIS_XX.wav', '1066_TIE_DIS_XX.wav']
```

Checks the folder where audio files are saved.

Prints total number of files and displays the first five.

Useful for verifying that the dataset has been downloaded correctly.

Metadata Extraction and Dataframe Creation

```
def parse_filename(filename):
    parts = filename.split('_')
    return {
        'filename': filename,
        'actor_id': parts[0],
        'emotion': parts[2],
        'intensity': parts[3],
    }

# Create dataframe
file_data = [parse_filename(f) for f in os.listdir(audio_dir) if f.endswith('.wav')]
df = pd.DataFrame(file_data)

# Show info
print(df.shape)
print(df.info())
print(df['emotion'].value_counts())
print("Missing values:\n", df.isnull().sum())
```

```
(7442, 4)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7442 entries, 0 to 7441
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   filename   7442 non-null   object
 1   actor_id   7442 non-null   object
 2   emotion    7442 non-null   object
 3   intensity  7442 non-null   object
dtypes: object(4)
memory usage: 232.7+ KB
None
emotion
DIS    1271
HAP    1271
SAD    1271
FEA    1271
ANG    1271
NEU    1087
Name: count, dtype: int64
Missing values:
 filename     0
actor_id     0
emotion      0
intensity    0
dtype: int64
```

This block parses the file names in the dataset. Each filename contains useful metadata like:

actor_id: the speaker,

emotion: emotion type (coded),

intensity: emotion intensity.

Then creates a Pandas DataFrame to organize this info.

Displays the shape, checks data types, and counts missing values. So missing value present

```
# Mapping dictionary
emotion_map = {
    'SAD': 'sad',
    'ANG': 'angry',
    'DIS': 'disgust',
    'FEA': 'fear',
    'HAP': 'happy',
    'NEU': 'neutral'
}

# Apply mapping
df['emotion_label'] = df['emotion'].map(emotion_map)
```

The original emotion codes (SAD, ANG, etc.) are mapped to human-readable labels using a dictionary.

A new column emotion_label is created for easy reference.

```
# Add full path to each audio file
df['file_path'] = df['filename'].apply(lambda x: os.path.join(audio_dir, x))
```

This adds the full file path of each audio file to the DataFrame.

Useful for loading audio files later using librosa.

```
# Check head
print(df.head())

# Optional: check for any missing mapped labels
print("Missing labels:", df['emotion_label'].isnull().sum())
```

```
              filename actor_id emotion intensity emotion_label  \
0  1028_TSI_DIS_XX.wav     1028     DIS    XX.wav       disgust
1  1075_IEO_HAP_LO.wav     1075     HAP    LO.wav         happy
2  1084_ITS_HAP_XX.wav     1084     HAP    XX.wav         happy
3  1067_IWW_DIS_XX.wav     1067     DIS    XX.wav       disgust
4  1066_TIE_DIS_XX.wav     1066     DIS    XX.wav       disgust

                                   file_path
0  /kaggle/input/cremad/AudioWAV/1028_TSI_DIS_XX.wav
1  /kaggle/input/cremad/AudioWAV/1075_IEO_HAP_LO.wav
2  /kaggle/input/cremad/AudioWAV/1084_ITS_HAP_XX.wav
3  /kaggle/input/cremad/AudioWAV/1067_IWW_DIS_XX.wav
4  /kaggle/input/cremad/AudioWAV/1066_TIE_DIS_XX.wav
Missing labels: 0
```

Displays the first few rows of the updated DataFrame.

Confirms that there are no missing labels after mapping

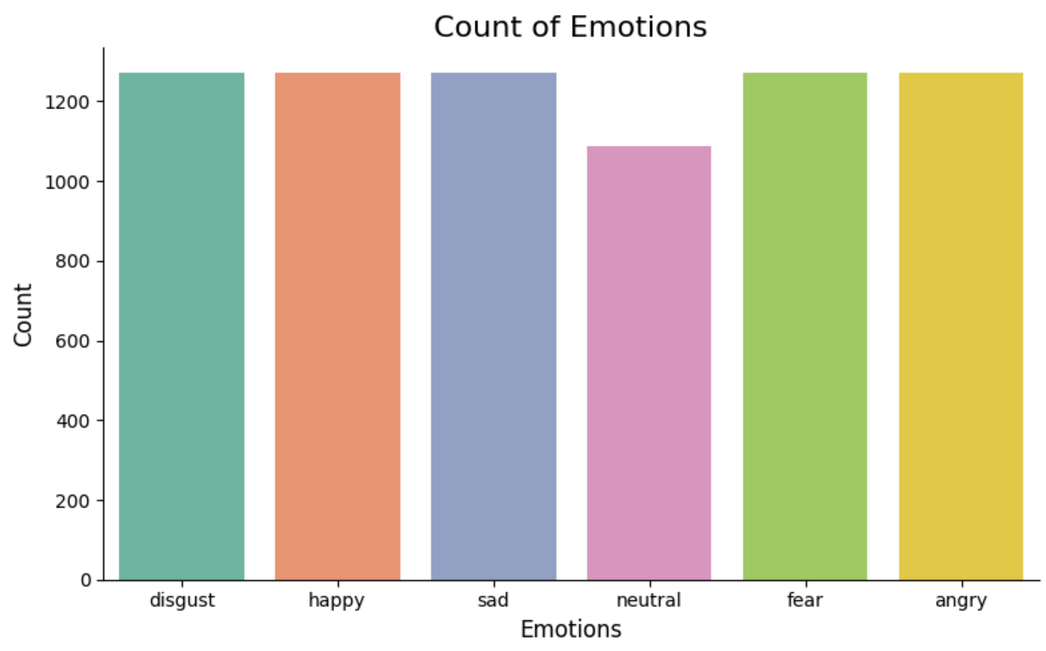
```
df.head()
```

	filename	actor_id	emotion	intensity	emotion_label	file_path
0	1028_TSI_DIS_XX.wav	1028	DIS	XX.wav	disgust	/kaggle/input/cremad/AudioWAV/1028_TSI_DIS_XX.wav
1	1075_IEO_HAP_LO.wav	1075	HAP	LO.wav	happy	/kaggle/input/cremad/AudioWAV/1075_IEO_HAP_LO.wav
2	1084_ITS_HAP_XX.wav	1084	HAP	XX.wav	happy	/kaggle/input/cremad/AudioWAV/1084_ITS_HAP_XX.wav
3	1067_IWW_DIS_XX.wav	1067	DIS	XX.wav	disgust	/kaggle/input/cremad/AudioWAV/1067_IWW_DIS_XX.wav
4	1066_TIE_DIS_XX.wav	1066	DIS	XX.wav	disgust	/kaggle/input/cremad/AudioWAV/1066_TIE_DIS_XX.wav

Emotion Distribution Visualization

```
plt.figure(figsize=(8, 5))
plt.title('Count of Emotions', size=16)

sns.countplot(x=df['emotion_label'], palette='Set2')

plt.ylabel('Count', size=12)
plt.xlabel('Emotions', size=12)
sns.despine(top=True, right=True, left=False, bottom=False)
```

```
plt.tight_layout()

plt.show()
```



Uses a bar plot to visualize the count of each emotion.

This helps identify class imbalance in the dataset (e.g., fewer "fear" samples).

Exploratory Data Analysis (EDA):

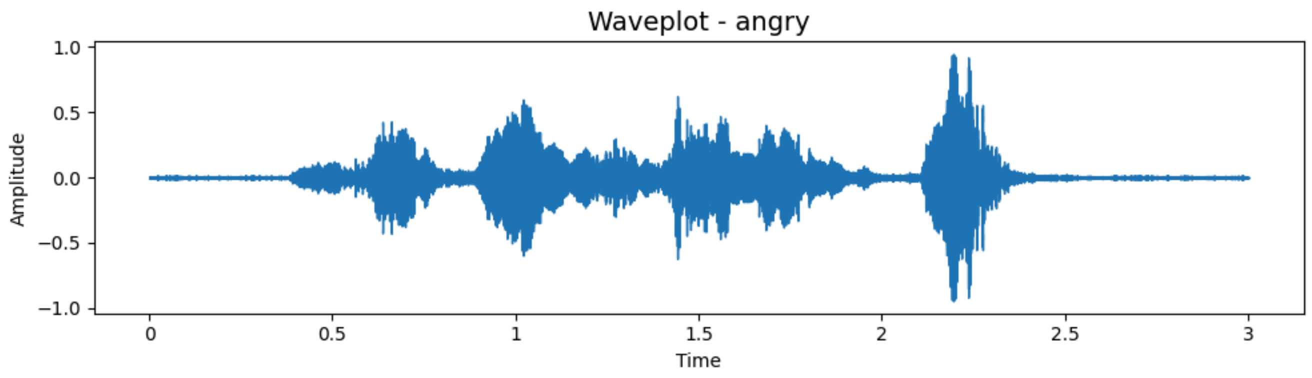
Audio Signal Visualization

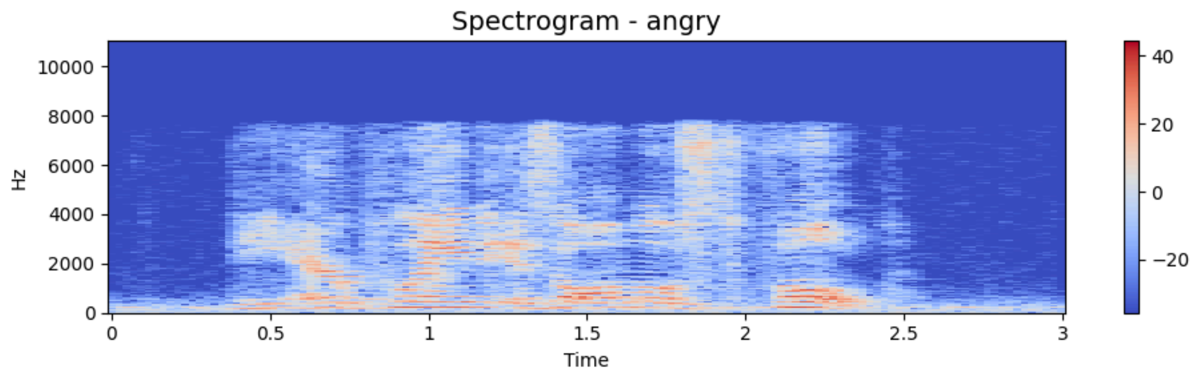
```
def create_waveplot(data, sr, emotion):
    plt.figure(figsize=(10, 3))
    librosa.display.waveshow(data, sr=sr)
    plt.title(f'Waveplot - {emotion}', size=14)
    plt.xlabel("Time")
    plt.ylabel("Amplitude")
    plt.tight_layout()
    plt.show()

def create_spectrogram(data, sr, emotion):
    X = librosa.stft(data)
    Xdb = librosa.amplitude_to_db(abs(X))
    plt.figure(figsize=(10, 3))
    librosa.display.specshow(Xdb, sr=sr, x_axis='time', y_axis='hz')
    plt.colorbar()
    plt.title(f'Spectrogram - {emotion}', size=14)
    plt.tight_layout()
    plt.show()
```

Sample Emotion Analysis

```
# Set target emotion
emotion = 'angry'  # Options: sad, happy, disgust, fear, angry, neutral

# Get first audio path for selected emotion
path = df[df['emotion_label'] == emotion]['file_path'].values[0]

# Load audio
data, sampling_rate = librosa.load(path)
```

```
# Create plots
create_waveplot(data, sampling_rate, emotion)
create_spectrogram(data, sampling_rate, emotion)

# Play audio
Audio(path)
```





0:00 / 0:03

Waveplot: Shows amplitude vs. time.

In this **“angry”** speech sample, you can see:

Sudden, high-amplitude spikes that indicate sharp bursts of energy

Clear segment transitions representing forceful articulation

These characteristics are typical of angry tone, which is often louder, more abrupt, and forceful.

Spectrogram: Shows frequency components over time.

Energy concentrated in the 1000–4000 Hz range — typical of strong voiced speech

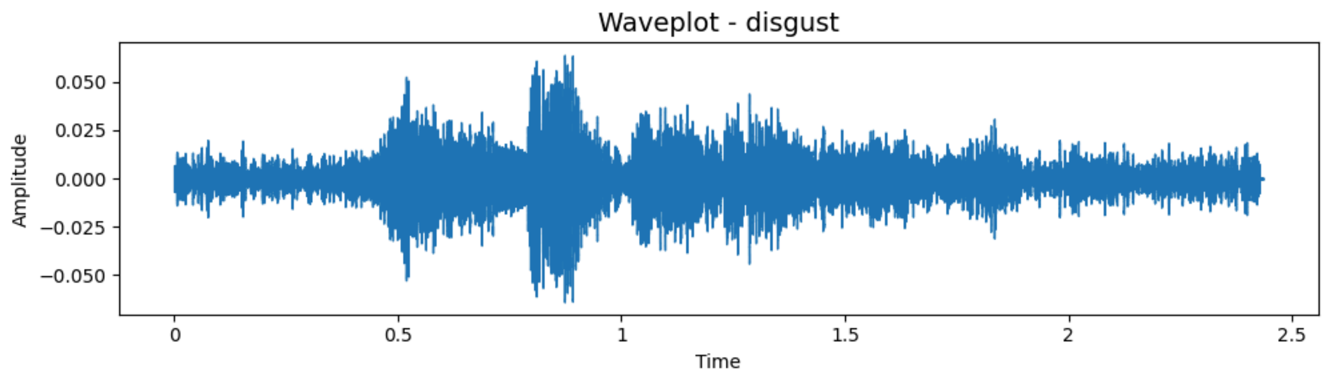
Frequent horizontal “stripes” — indicate pitch harmonics, which are rich and strong in angry speech

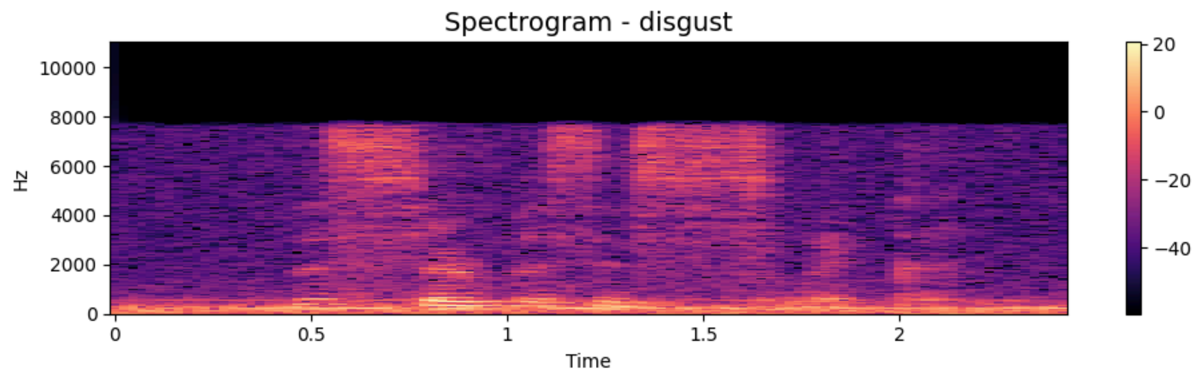
More red/orange regions = high energy intensity, supporting the “angry” label

Audio is also played using IPython.display.Audio.

```
emotions = df['emotion_label'].unique()

for emotion in emotions:
    path = df[df['emotion_label'] == emotion]['file_path'].values[0]
    data, sr = librosa.load(path)
    print(f"🔊 Emotion: {emotion}")
    create_waveplot(data, sr, emotion)
    create_spectrogram(data, sr, emotion)
    display(Audio(path))
```

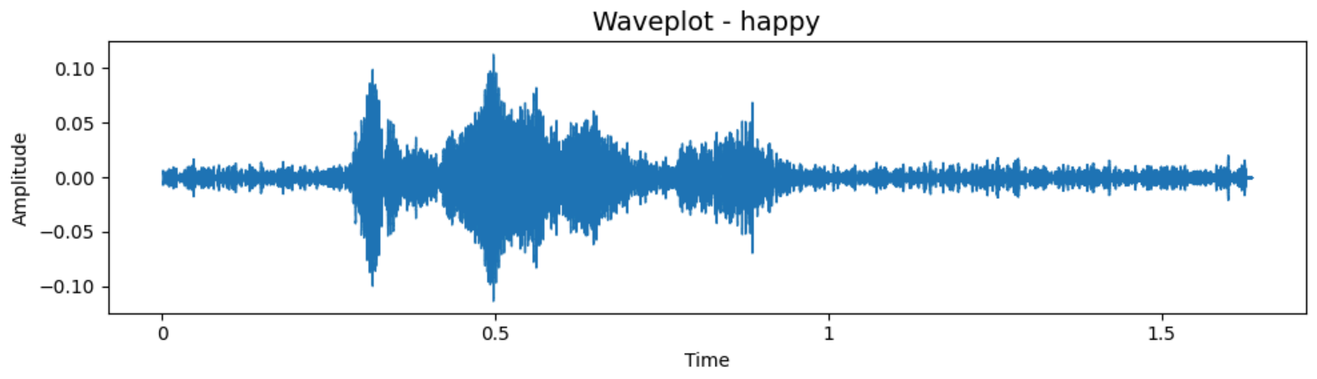
🔊 Emotion: disgust

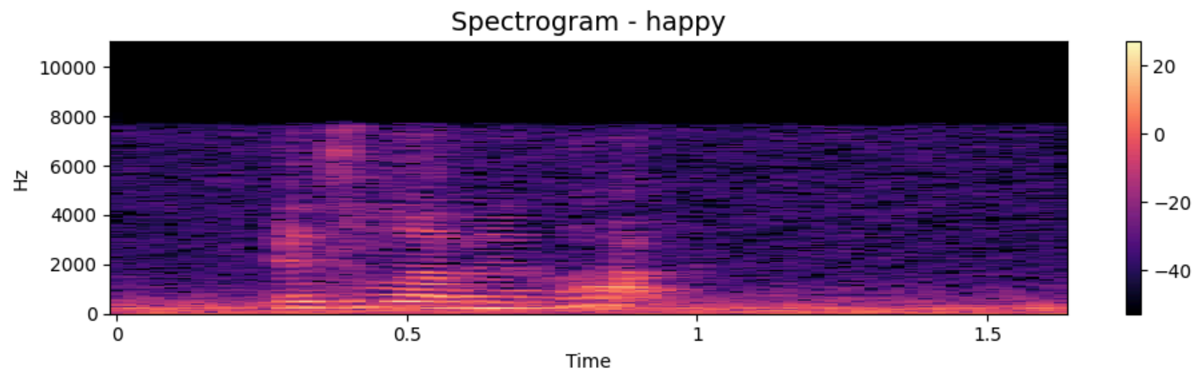




0:00 / 0:02

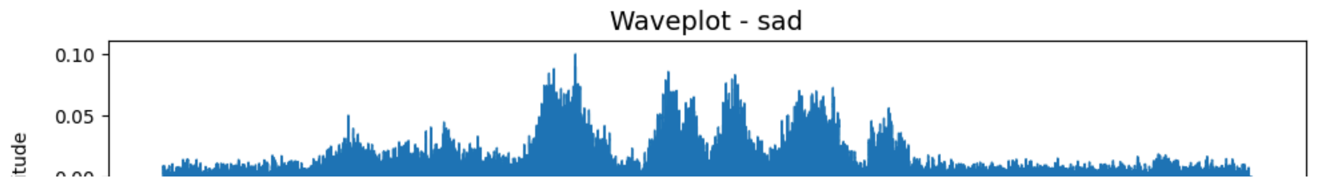
🔊 Emotion: happy

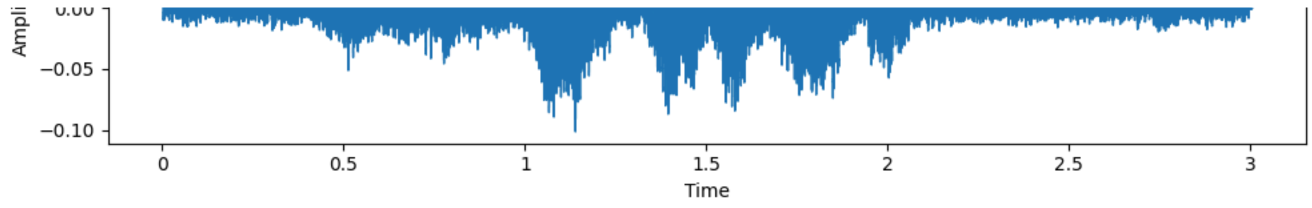


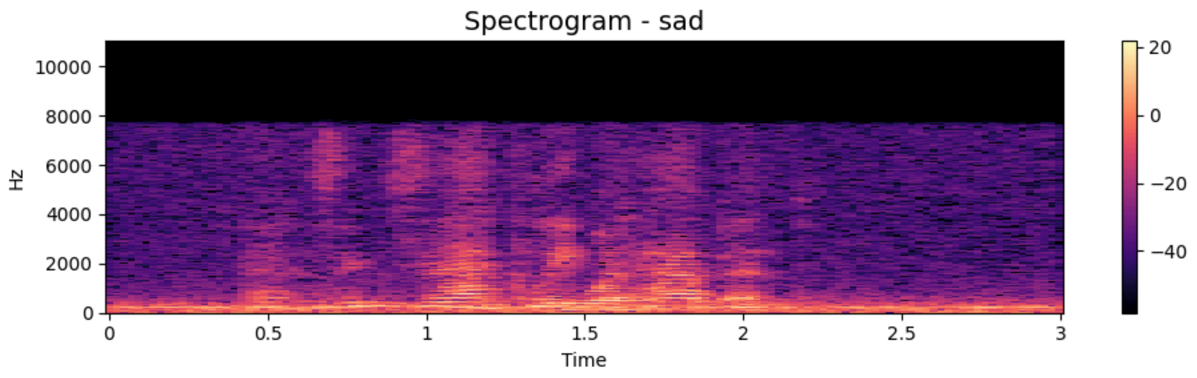


0:00 / 0:01

🔊 Emotion: sad

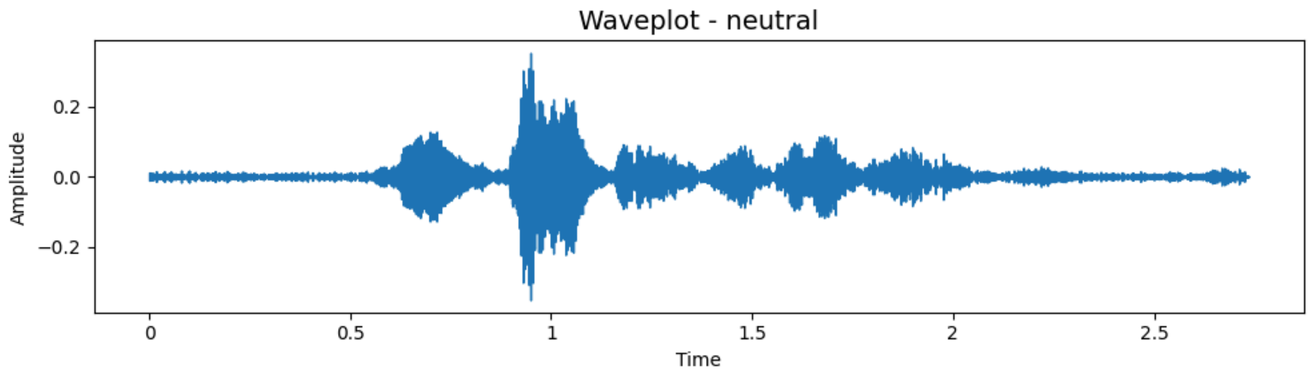


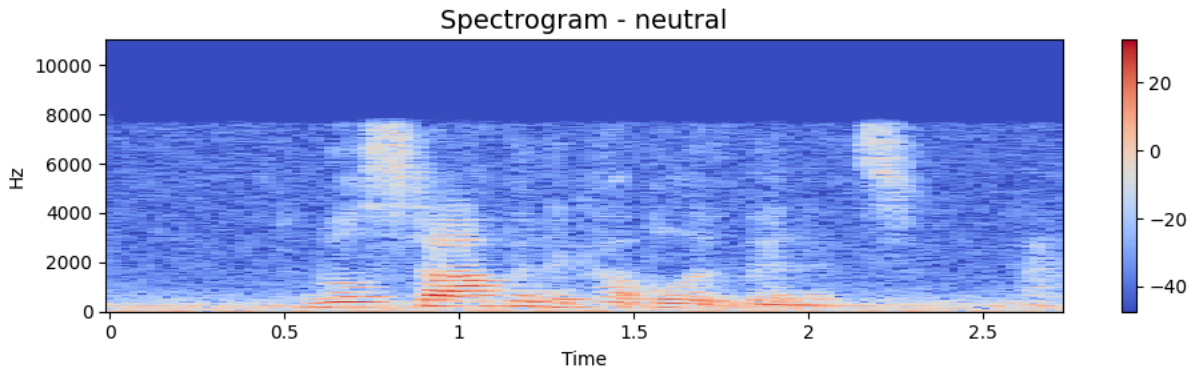




0:00 / 0:03

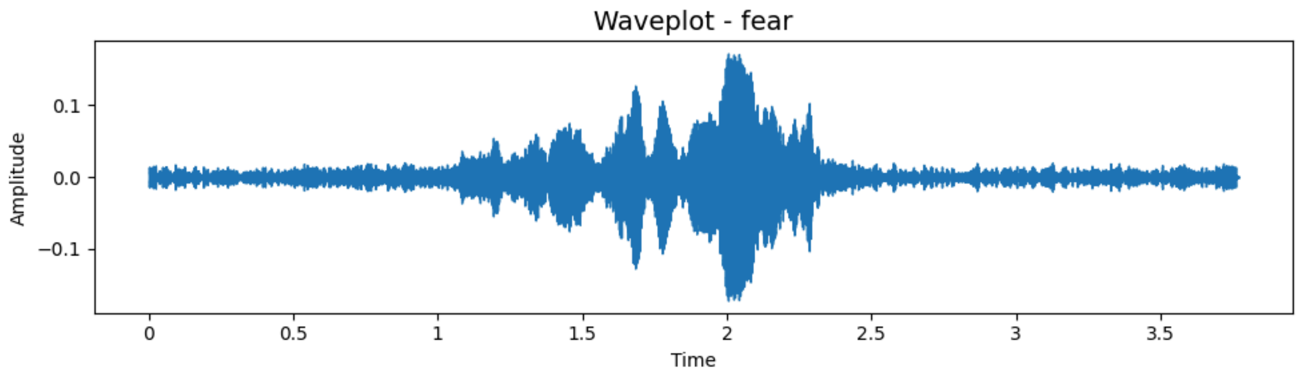
🔊 `Emotion: neutral`

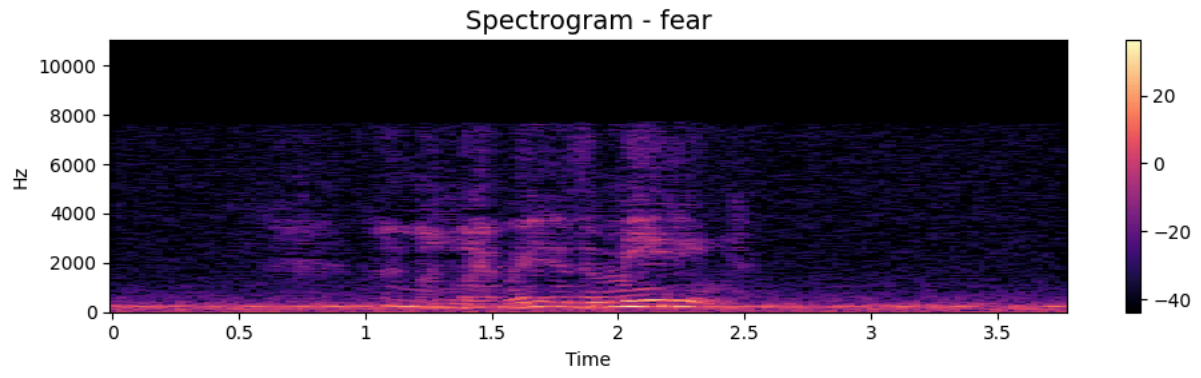




0:00 / 0:02

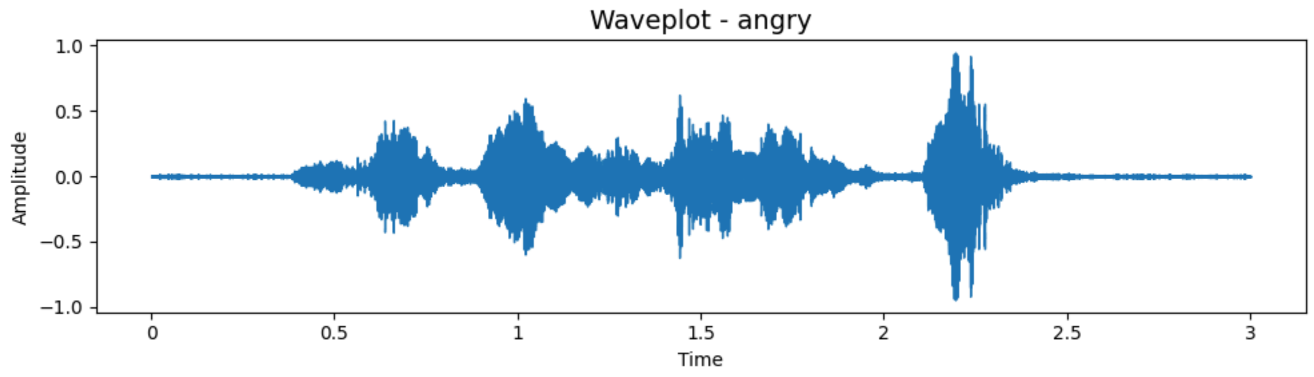
🔊 `Emotion: fear`

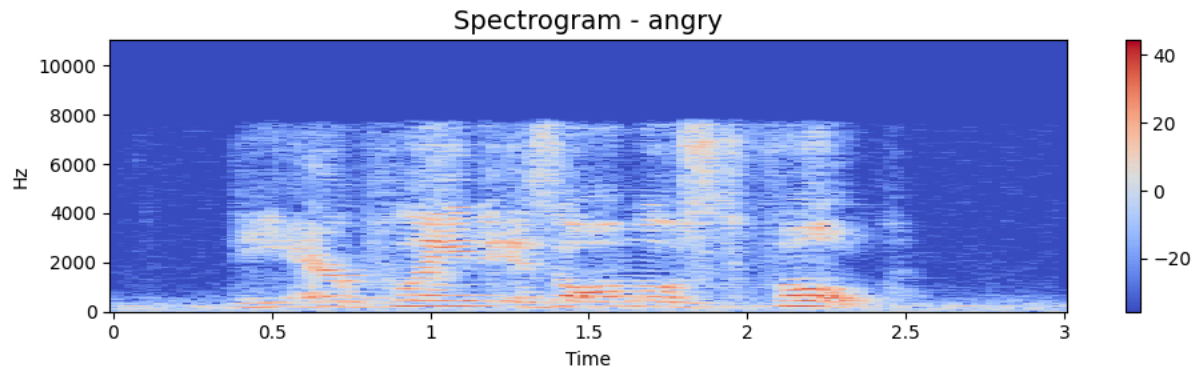




0:00 / 0:03

```
🔊 Emotion: angry
```





0:00 / 0:03

For each unique emotion, it loads a sample audio and displays its waveform and spectrogram.

Extracting Features: MFCCs(Mel-Frequency Cepstral Coefficients,Chroma, Mel Spectrogram, and ZCR)

```
import librosa
import numpy as np
from tqdm import tqdm

valid_labels = ['happy', 'angry', 'fear', 'sad', 'disgust', 'neutral']
max_len = 300
X = []
y = []

for _, row in tqdm(df.iterrows(), total=len(df)):
    try:
        label = row['emotion_label']
        if label not in valid_labels:
            continue

        signal, sr = librosa.load(row['file_path'], sr=22050)
        signal = signal / (np.max(np.abs(signal)) + 1e-7)  # Normalize volume

        # Extract features
        mfcc = librosa.feature.mfcc(y=signal, sr=sr, n_mfcc=13).T              # (T, 13)
        chroma = librosa.feature.chroma_stft(y=signal, sr=sr).T                # (T, 12)
        mel = librosa.feature.melspectrogram(y=signal, sr=sr, n_mels=128)
        mel_db = librosa.power_to_db(mel).T                                    # (T, 128)
        zcr = librosa.feature.zero_crossing_rate(y=signal).T                   # (T, 1)


        def pad(x):
            return np.pad(x, ((0, max_len - x.shape[0]), (0, 0)), mode='constant') if x.shape[0] < max_len else x[:max_len, :]

        mfcc = pad(mfcc)
        chroma = pad(chroma)
        mel_db = pad(mel_db)
        zcr = pad(zcr)


        combined = np.concatenate((mfcc, chroma, mel_db, zcr), axis=1)
        X.append(combined)
        y.append(label)

    except Exception as e:
        print(f"Error with {row['filename']}: {e}")

X = np.array(X)
y = np.array(y)
print("Feature shape:", X.shape)
```

```
100%|██████████| 7442/7442 [05:51<00:00, 21.15it/s]
Feature shape: (7442, 300, 154)
```

MFCCs – Captures short-term spectral envelope.

Chroma – Captures harmonic features.

Mel Spectrogram – Models how humans perceive frequencies.

ZCR – Indicates noisiness of signal.

Why we done this

MFCCs capture the tone and texture of speech, mimicking human hearing.

Chroma features help detect pitch and harmonic patterns related to emotions.

Mel Spectrogram provides a detailed view of frequency energy over time.

ZCR measures the noisiness or intensity, useful for detecting harsh emotions like anger.

Each feature is computed as a time-series matrix and padded or truncated to a consistent length of 300 time steps, ensuring uniform input dimensions for the model. The padded matrices are then concatenated along the feature axis to form a unified input tensor for each sample, resulting in a final shape of (samples, 300, total_features).

This hybrid feature representation enriches the model's ability to learn both spectral and temporal cues associated with various emotional states in speech.

Applying Model:

```
Convolutional Neural Networks(CNN) + Bidirectional LSTM (BiLSTM)
```

Architecture Overview

- **Convolutional Layers (CNN)**: The first part of the model includes two 1D convolutional layers that slide filters over the audio feature sequences (MFCC + Chroma + Mel + ZCR) to extract local time-frequency patterns.

  Each convolutional layer is followed by:

  Batch Normalization: for stabilizing training

  Max Pooling: for downsampling and reducing computation

  Dropout: to prevent overfitting
- **Bidirectional LSTM Layers (BiLSTM)**: After the CNN layers, two Bidirectional LSTM layers are added.These capture forward and backward temporal relationships in the speech signal, making the model sensitive to both past and future acoustic context.

  This is especially important for detecting nuanced emotional cues in speech that evolve over time.
- **Fully Connected Layers**: A dense layer with ReLU activation further processes the output from the BiLSTM block.The final output layer uses a Softmax activation to classify into one of the six emotion categories.

Model Configuration:

- **Input Shape:** (300, 154) representing padded sequences of extracted features.
- **Optimizer**: Nadam with a learning rate of 0.0005
- **Loss Function:** Categorical Crossentropy with label smoothing to improve generalization
- **Regularization:** L2 kernel regularizer applied to convolution layers

Callbacks:

EarlyStopping to halt training early if no improvement is seen in validation loss

ReduceLROnPlateau to dynamically reduce learning rate and escape plateaus

Training Strategy

The model is trained using:

A batch size of 32

Up to 50 epochs

With class weights applied to address emotion class imbalance

Validation accuracy and loss are monitored across epochs to select the best-performing model.

```
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.utils import to_categorical

le = LabelEncoder()
y_encoded = le.fit_transform(y)
y_onehot = to_categorical(y_encoded)

print("Class mapping:", dict(zip(le.classes_, le.transform(le.classes_))))
```

```
Class mapping: {'angry': 0, 'disgust': 1, 'fear': 2, 'happy': 3, 'neutral': 4, 'sad': 5}
```

Before training the model, the categorical emotion labels must be converted into a numerical format suitable for classification tasks. This is done in two key steps:

Label Encoding: The LabelEncoder from Scikit-learn assigns a unique integer to each emotion class (e.g., angry → 0, happy → 1, etc.), which allows the model to interpret the classes numerically.

One-Hot Encoding: The encoded labels are then transformed using Keras' to_categorical() function into one-hot vectors, which are required for multi-class classification using softmax activation and categorical crossentropy loss.

This dual transformation ensures the labels are in the optimal format for both model training and evaluation.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y_onehot, test_size=0.2, random_state=42, stratify=y_encoded
)
```

```
X_train[0]
```

```
array([[-3.13310486e+02,  1.30495758e+02,  2.75206966e+01, ...,
        -4.99217758e+01, -4.99569130e+01,  5.37109375e-03],
       [-2.89408691e+02,  1.37901230e+02,  2.59613342e+01, ...,
        -5.00998001e+01, -5.00998001e+01,  8.78906250e-03],
       [-2.92786041e+02,  1.31681763e+02,  3.47370033e+01, ...,
        -5.00998001e+01, -5.00998001e+01,  1.51367188e-02],
       ...,
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00, ...,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00, ...,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00, ...,
         0.00000000e+00,  0.00000000e+00,  0.00000000e+00]])
```

```
from sklearn.utils.class_weight import compute_class_weight

class_weights = compute_class_weight(
    class_weight='balanced',
    classes=np.unique(y_encoded),
    y=y_encoded
)
class_weights = dict(enumerate(class_weights))
print("Class Weights:", class_weights)
```

```
Class Weights: {0: 0.9758720167846839, 1: 0.9758720167846839, 2: 0.9758720167846839, 3: 0.9758720167846839, 4: 1.1410610242256976, 5: 0.
```

In datasets like CREMA-D, the distribution of emotion classes may not be perfectly balanced, leading to bias in model training where the model favors majority classes. To address this, class weights are computed and applied during training to give more importance to underrepresented classes.

Using Scikit-learn's compute_class_weight() function, balanced weights are calculated based on the frequency of each class. These weights are passed to the model's fit() function to ensure that the loss contribution of each class is fairly adjusted.

This approach improves model generalization across all emotion classes and helps mitigate bias toward more frequent labels.

```
print(X_train.shape)
print(X_test.shape)
print(y_test.shape)
print(y_train.shape)
```

```
(5953, 300, 154)
(1489, 300, 154)
(1489, 6)
(5953, 6)
```

```
# Show the first training sample
print("Train sample 1:")
print(X_train[0])
print("Label:", y_train[0])

# Show the first testing sample
print("\nTest sample 1:")
print(X_test[0])
print("Label:", y_test[0])
```

```
Train sample 1:
[[-3.13310486e+02  1.30495758e+02  2.75206966e+01 ... -4.99217758e+01
  -4.99569130e+01  5.37109375e-03]
 [-2.89408691e+02  1.37901230e+02  2.59613342e+01 ... -5.00998001e+01
  -5.00998001e+01  8.78906250e-03]
 [-2.92786041e+02  1.31681763e+02  3.47370033e+01 ... -5.00998001e+01
  -5.00998001e+01  1.51367188e-02]
 ...
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]]
Label: [0. 0. 0. 0. 1. 0.]

Test sample 1:
[[-4.40432831e+02  6.18224716e+01  5.19879684e+01 ... -4.30233231e+01
  -4.30233231e+01  1.41601562e-02]
 [-4.31649658e+02  7.17967224e+01  5.63582230e+01 ... -4.30233231e+01
  -4.30233231e+01  2.68554688e-02]
 [-4.33172546e+02  7.02131653e+01  5.58502045e+01 ... -4.30233231e+01
  -4.30233231e+01  3.36914062e-02]
 ...
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]]
Label: [1. 0. 0. 0. 0. 0.]
```

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train.reshape(-1, X_train.shape[-1])).reshape(X_train.shape)
X_test = scaler.transform(X_test.reshape(-1, X_test.shape[-1])).reshape(X_test.shape)
```

Before feeding the feature sequences into the neural network, a StandardScaler is applied to normalize the data. This step ensures that all features contribute equally to the model by transforming them to have zero mean and unit variance, which improves the convergence speed and stability during training.

Because the input data has a 3D shape (samples, time_steps, features), the features are reshaped to 2D, normalized using StandardScaler, and then reshaped back to the original 3D format. This approach applies scaling across all time steps consistently for each feature.

After normalization, the first samples from both the training and testing sets are printed. These show:

The numerical values of the time-aligned feature sequences

The padded zeros at the end (from shorter audio clips)

The one-hot encoded labels that indicate the correct emotion category

```
# CNN + BiLSTM with moderate depth
model = Sequential()

model.add(Conv1D(64, kernel_size=3, activation='relu', input_shape=(300, 154),kernel_regularizer=l2(0.00001)))
model.add(BatchNormalization())
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.3))

model.add(Conv1D(128, kernel_size=3, activation='relu',kernel_regularizer=l2(0.00001)))
model.add(BatchNormalization())
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.3))

model.add(Bidirectional(LSTM(64,return_sequences=True))) #change
model.add(Bidirectional(LSTM(32)))
# Second layer
model.add(BatchNormalization())   #change


model.add(Dense(64, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(6, activation='softmax'))

model.compile(
    optimizer=Nadam(learning_rate=0.0005),
    loss=CategoricalCrossentropy(label_smoothing=0.1),
    metrics=['accuracy']
)
```

```
model.summary()
```

```
I0000 00:00:1752810651.989263      36 gpu_device.cc:2022] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 13942 MB memc
I0000 00:00:1752810651.989971      36 gpu_device.cc:2022] Created device /job:localhost/replica:0/task:0/device:GPU:1 with 13942 MB memc
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                       ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ conv1d (Conv1D)                    │ (None, 298, 64)        │        29,632 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization                │ (None, 298, 64)        │           256 │
│ (BatchNormalization)               │                        │               │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling1d (MaxPooling1D)       │ (None, 149, 64)        │             0 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)                  │ (None, 149, 64)        │             0 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ conv1d_1 (Conv1D)                  │ (None, 147, 128)       │        24,704 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_1              │ (None, 147, 128)       │           512 │
│ (BatchNormalization)               │                        │               │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ max_pooling1d_1 (MaxPooling1D)     │ (None, 73, 128)        │             0 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_1 (Dropout)                │ (None, 73, 128)        │             0 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ bidirectional (Bidirectional)      │ (None, 73, 128)        │        98,816 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ bidirectional_1 (Bidirectional)    │ (None, 64)             │        41,216 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization_2              │ (None, 64)             │           256 │
│ (BatchNormalization)               │                        │               │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                      │ (None, 64)             │         4,160 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_2 (Dropout)                │ (None, 64)             │             0 │
├────────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                    │ (None, 6)              │           390 │
└────────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 199,942 (781.02 KB)
 Trainable params: 199,430 (779.02 KB)
 Non-trainable params: 512 (2.00 KB)
```

```
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=3,verbose=1)
```

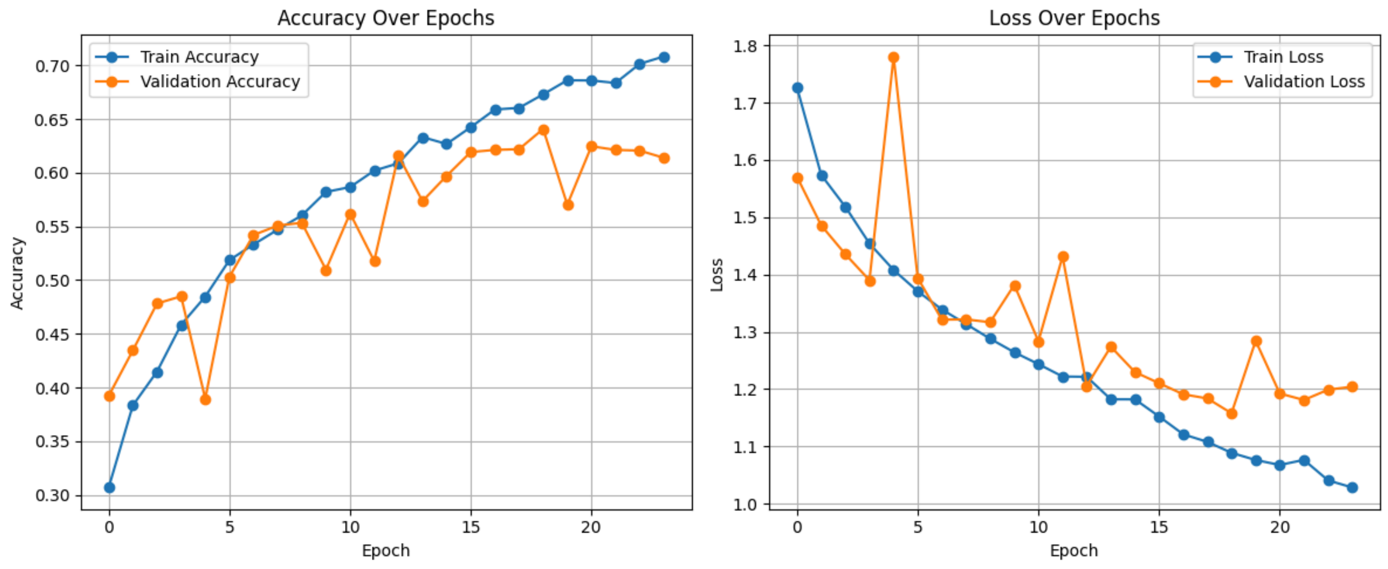
• Training The Model

```
history = model.fit(
    X_train, y_train, #change  X_train,y_train
    validation_data=(X_test, y_test),
    epochs=50,
    batch_size=32,
    class_weight=class_weights,
    callbacks=[early_stop, reduce_lr],
    verbose=1
)
```

```
Epoch 1/50
I0000 00:00:1752810670.585265     116 cuda_dnn.cc:529] Loaded cuDNN version 90300
187/187 ━━━━━━━━━━━━━━━━━━━━ 20s 33ms/step - accuracy: 0.2633 - loss: 1.8232 - val_accuracy: 0.3922 - val_loss: 1.5703 - learning_rate:
Epoch 2/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.3745 - loss: 1.6027 - val_accuracy: 0.4345 - val_loss: 1.4858 - learning_rate: 5
Epoch 3/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.4128 - loss: 1.5230 - val_accuracy: 0.4782 - val_loss: 1.4355 - learning_rate: 5
Epoch 4/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.4665 - loss: 1.4453 - val_accuracy: 0.4849 - val_loss: 1.3900 - learning_rate: 5
Epoch 5/50
```

```
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 23ms/step - accuracy: 0.4812 - loss: 1.4194 - val_accuracy: 0.3895 - val_loss: 1.7807 - learning_rate: 5
Epoch 6/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 5s 24ms/step - accuracy: 0.5289 - loss: 1.3625 - val_accuracy: 0.5030 - val_loss: 1.3938 - learning_rate: 5
Epoch 7/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 23ms/step - accuracy: 0.5185 - loss: 1.3521 - val_accuracy: 0.5420 - val_loss: 1.3213 - learning_rate: 5
Epoch 8/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.5527 - loss: 1.3044 - val_accuracy: 0.5507 - val_loss: 1.3217 - learning_rate: 5
Epoch 9/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.5628 - loss: 1.2836 - val_accuracy: 0.5534 - val_loss: 1.3167 - learning_rate: 5
Epoch 10/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.5865 - loss: 1.2569 - val_accuracy: 0.5097 - val_loss: 1.3818 - learning_rate: 5
Epoch 11/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.5895 - loss: 1.2564 - val_accuracy: 0.5621 - val_loss: 1.2836 - learning_rate: 5
Epoch 12/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.6053 - loss: 1.2171 - val_accuracy: 0.5178 - val_loss: 1.4313 - learning_rate: 5
Epoch 13/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 5s 24ms/step - accuracy: 0.6096 - loss: 1.2195 - val_accuracy: 0.6158 - val_loss: 1.2058 - learning_rate: 5
Epoch 14/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 5s 24ms/step - accuracy: 0.6364 - loss: 1.1743 - val_accuracy: 0.5735 - val_loss: 1.2741 - learning_rate: 5
Epoch 15/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.6345 - loss: 1.1766 - val_accuracy: 0.5970 - val_loss: 1.2297 - learning_rate: 5
Epoch 16/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.6454 - loss: 1.1426
Epoch 16: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.6454 - loss: 1.1427 - val_accuracy: 0.6192 - val_loss: 1.2103 - learning_rate: 5
Epoch 17/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 5s 24ms/step - accuracy: 0.6436 - loss: 1.1379 - val_accuracy: 0.6212 - val_loss: 1.1909 - learning_rate: 2
Epoch 18/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 23ms/step - accuracy: 0.6641 - loss: 1.0978 - val_accuracy: 0.6219 - val_loss: 1.1837 - learning_rate: 2
Epoch 19/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.6745 - loss: 1.0874 - val_accuracy: 0.6407 - val_loss: 1.1579 - learning_rate: 2
Epoch 20/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.6885 - loss: 1.0614 - val_accuracy: 0.5702 - val_loss: 1.2842 - learning_rate: 2
Epoch 21/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.6874 - loss: 1.0679 - val_accuracy: 0.6246 - val_loss: 1.1923 - learning_rate: 2
Epoch 22/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.6856 - loss: 1.0735
Epoch 22: ReduceLROnPlateau reducing learning rate to 0.0001250000059371814.
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.6856 - loss: 1.0735 - val_accuracy: 0.6212 - val_loss: 1.1810 - learning_rate: 2
Epoch 23/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.7022 - loss: 1.0457 - val_accuracy: 0.6206 - val_loss: 1.1989 - learning_rate: 1
Epoch 24/50
187/187 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.7042 - loss: 1.0367 - val_accuracy: 0.6138 - val_loss: 1.2042 - learning_rate: 1
```

```python
import matplotlib.pyplot as plt

# Accuracy plot
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy', marker='o')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy', marker='o')
plt.title('Accuracy Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)

# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss', marker='o')
plt.plot(history.history['val_loss'], label='Validation Loss', marker='o')
plt.title('Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```



The model's performance during training is visualized through accuracy and loss curves over the training epochs. These plots provide a clear picture of how well the model is learning and generalizing to unseen data.

Accuracy Over Epochs (Left Plot): The training accuracy shows a consistent upward trend, indicating that the model is effectively learning patterns from the training data.

The validation accuracy improves significantly during the early epochs and then stabilizes around 60–64%, suggesting that the model is generalizing reasonably well, although some variance remains due to class imbalance or noise.

Loss Over Epochs (Right Plot): The training loss steadily decreases across epochs, demonstrating successful convergence.

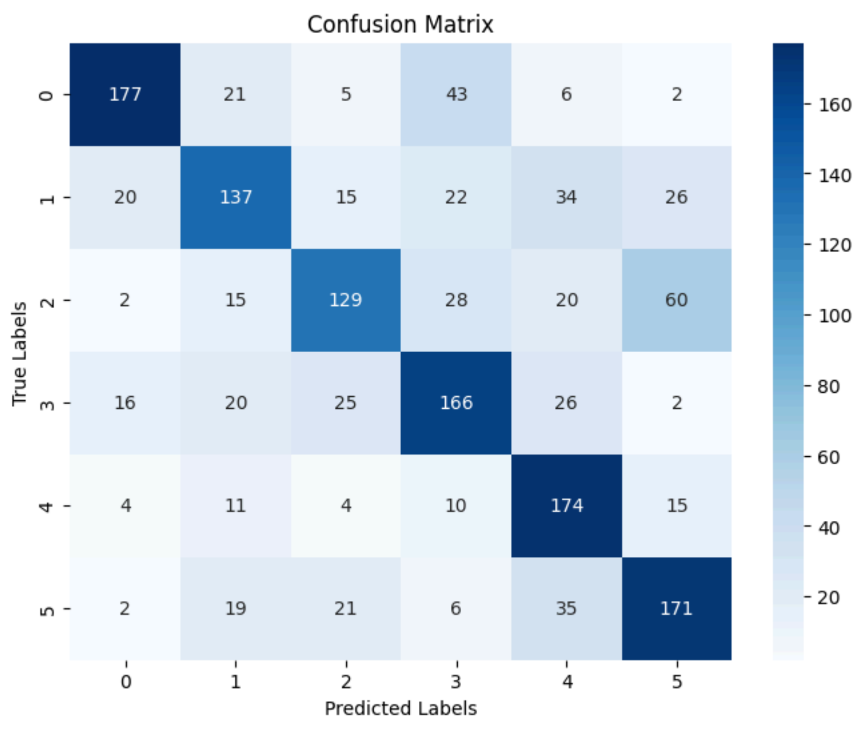
The validation loss shows occasional spikes, which may point to overfitting or noise sensitivity in the validation set. However, the general trend is downward, indicating progress.

- ## Testing the Model

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import numpy as np

# Predict class probabilities
y_pred_probs = model.predict(X_test)

# Convert probabilities to class labels
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = np.argmax(y_test, axis=1)

# Accuracy
test_accuracy = accuracy_score(y_true, y_pred)
print("Test Accuracy:", test_accuracy)
```

```
47/47 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step
Test Accuracy: 0.6406984553391538
```

```
import seaborn as sns
import matplotlib.pyplot as plt

cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```



- **Accuracy , Precision , Recall , F1-score of the Implemented Model**

```
print(classification_report(y_true, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.80      0.70      0.75       254
           1       0.61      0.54      0.57       254
           2       0.65      0.51      0.57       254
           3       0.60      0.65      0.63       255
           4       0.59      0.80      0.68       218
           5       0.62      0.67      0.65       254

    accuracy                           0.64      1489
   macro avg       0.65      0.64      0.64      1489
weighted avg       0.65      0.64      0.64      1489
```

Predicting the Emotions for an unseen Audio File

```
import numpy as np

# Class mapping
class_mapping = {'angry': 0, 'disgust': 1, 'fear': 2, 'happy': 3, 'neutral': 4, 'sad': 5}
index_to_class = {v: k for k, v in class_mapping.items()}

# Sample index from test set
sample_index = 1

# Select sample
```

```
sample = X_test[sample_index]  # shape (300, 154)
sample = np.expand_dims(sample, axis=0)  # shape becomes (1, 300, 154)

# Make prediction
prediction = model.predict(sample)
predicted_class_index = np.argmax(prediction, axis=1)[0]
predicted_class_name = index_to_class[predicted_class_index]

# Get true label
true_label_index = y_test[sample_index]
if isinstance(true_label_index, np.ndarray):  # If one-hot encoded
    true_label_index = np.argmax(true_label_index)

true_class_name = index_to_class[true_label_index]

# Output
print("Predicted Class Index:", predicted_class_index)
print("Predicted Class Name:", predicted_class_name)
print("Actual Class Index:", true_label_index)
print("Actual Class Name:", true_class_name)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step
Predicted Class Index: 3
Predicted Class Name: happy
Actual Class Index: 3
Actual Class Name: happy
```

```
import numpy as np

# Class mapping
class_mapping = {'angry': 0, 'disgust': 1, 'fear': 2, 'happy': 3, 'neutral': 4, 'sad': 5}
index_to_class = {v: k for k, v in class_mapping.items()}

# Sample index from test set
sample_index = 5

# Select sample
sample = X_test[sample_index]  # shape (300, 154)
sample = np.expand_dims(sample, axis=0)  # shape becomes (1, 300, 154)

# Make prediction
prediction = model.predict(sample)
predicted_class_index = np.argmax(prediction, axis=1)[0]
predicted_class_name = index_to_class[predicted_class_index]

# Get true label
true_label_index = y_test[sample_index]
if isinstance(true_label_index, np.ndarray):  # If one-hot encoded
    true_label_index = np.argmax(true_label_index)

true_class_name = index_to_class[true_label_index]

# Output
print("Predicted Class Index:", predicted_class_index)
print("Predicted Class Name:", predicted_class_name)
print("Actual Class Index:", true_label_index)
print("Actual Class Name:", true_class_name)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step
Predicted Class Index: 5
Predicted Class Name: sad
Actual Class Index: 5
Actual Class Name: sad
```

```
import numpy as np

# Class mapping
class_mapping = {'angry': 0, 'disgust': 1, 'fear': 2, 'happy': 3, 'neutral': 4, 'sad': 5}
index_to_class = {v: k for k, v in class_mapping.items()}

# Sample index from test set
sample_index = 10

# Select sample
sample = X_test[sample_index]  # shape (300, 154)
sample = np.expand_dims(sample, axis=0)  # shape becomes (1, 300, 154)

# Make prediction
prediction = model.predict(sample)
predicted_class_index = np.argmax(prediction, axis=1)[0]
predicted_class_name = index_to_class[predicted_class_index]
```