Pattern Recognition

Assignment (3)

Speech Emotion Recognition

Team members

Nouran Hisham 6532

1D model notebook link: https://colab.research.google.com/drive/1-zdl8N3CMHmAE5OUFQNp33c96vzqP0W1?usp=sharing

2D model notebook link:
https://colab.research.google.com/drive/1cz8mibrjFwwpczWOk2y6eZNEJ2kW9R4j?usp=sharing

Note: Detailed comments are added in the notebooks for further explanation

Note: Detailed output is present in the notebooks.

Problem Statement:

Speech is the most natural way of expressing ourselves as humans. It is only natural then to extend this communication medium to computer applications. We define speech emotion recognition (SER) systems as a collection of methodologies that process and classify speech signals to detect the embedded emotions.

Imported libraries to be used:

```
[ ] import pandas as pd
    import numpy as np

    import os
    import sys

    import librosa as lr
    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 warnings
    if not sys.warnoptions:
        warnings.simplefilter("ignore")
    warnings.filterwarnings("ignore", category=DeprecationWarning)
```

1D model

Download the Dataset and Understand the Format:

We will use CREMA dataset that is available at the following link:
https://www.kaggle.com/dmitrybabko/speech-emotion-recognition-en

While loading the audio files, we used a sample rate of 16,000, trimmed any voice less than 60 dB, made the length of all audio files 3 seconds, and applied zero padding technique.

```
! pip install -q kaggle
! cp kaggle.json ~/.kaggle/
! chmod 600 ~/.kaggle/kaggle.json
! kaggle datasets list
! kaggle datasets download dmitrybabko/speech-emotion-recognition-en
! unzip speech-emotion-recognition-en.zip
```

```
def LoadAudio(Crema):
    emotions = []
    Audio_List=[]
    samp_freq=[]
    paths = []

    for wav in os.listdir(Crema):
        path=Crema+"/"+wav
        paths.append(path)
        audio,sampling_freq=lr.load(path,sr=8000)
        yt, index = lr.effects.trim(audio, top_db=60)

        if len(yt) > (2*8000):
            yt = yt[:2*8000]
        else:
            padding = (2*8000) - len(yt)
            offset = padding // 2
            yt = np.pad(yt, (offset,2*8000- len(yt) - offset), 'constant')

        mean = np.mean(yt)
        std = np.std(yt)
        out = np.ones( (len(yt)) )
        yt= np.divide((yt - mean),std,out=out, where=std!=0)
```

```
        Audio_List.append(yt)
        samp_freq.append(sampling_freq)

        info = wav.partition(".wav")[0].split("_")
        if info[2] == 'SAD':
            emotions.append(0)
        elif info[2] == 'ANG':
            emotions.append(1)
        elif info[2] == 'DIS':
            emotions.append(2)
        elif info[2] == 'FEA':
            emotions.append(3)
        elif info[2] == 'HAP':
            emotions.append(4)
        elif info[2] == 'NEU':
            emotions.append(5)
        else:
            emotions.append(6)

    return Audio_List,samp_freq,emotions, paths
```

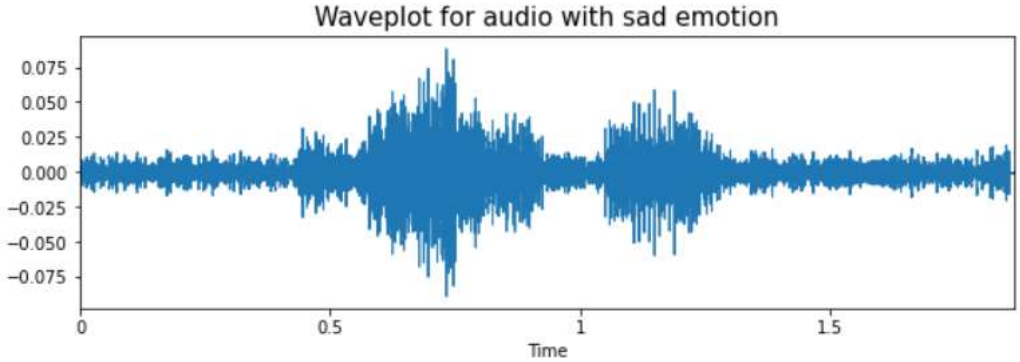
```
[ ] Audio_List,samp_freq,Labels, paths=  LoadAudio('Crema')

    emotions_df = pd.DataFrame(Labels, columns=['Emotions'])
    paths_df = pd.DataFrame(paths, columns=['Paths'])
    crema_df = pd.concat([emotions_df, paths_df], axis=1)
    data_paths = crema_df

    print(len(Audio_List))
    print(len(Labels))
```

```
7442
7442
```

Listen and plot the waveform of the audio files:

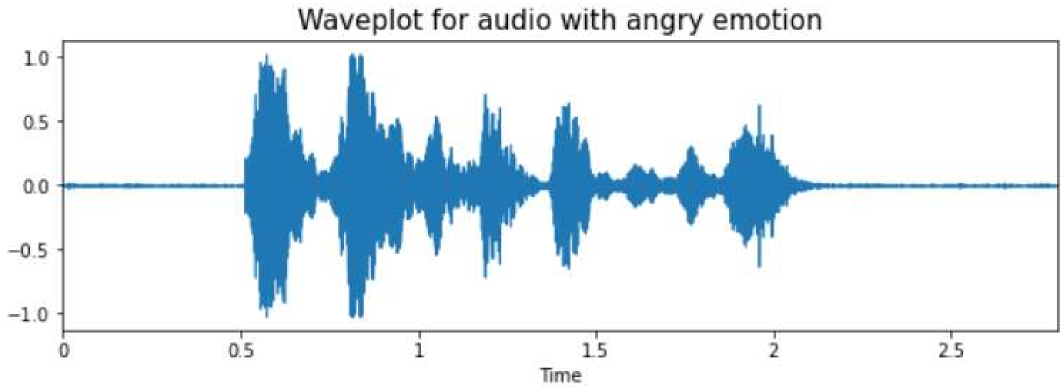
```
[ ] def visualize(data, sample_rate, emotion):
        plt.figure(figsize=(10, 3))
        plt.title('Waveplot for audio with '+str(emotion)+' emotion', size=15)
        librosa.display.waveplot(data, sr=sample_rate)
        plt.show()
```

```
[ ] audio_path = np.array(crema_df.Paths[crema_df.Emotions==0])[0]
    data, sample_rate = librosa.load(audio_path)
    visualize(data, sample_rate, "sad")
    Audio(audio_path)
```



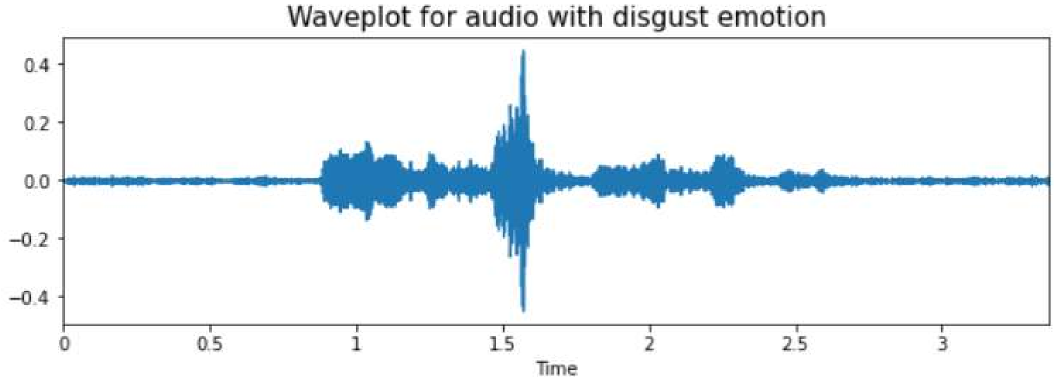
0:01 / 0:01

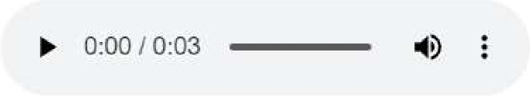
```
[ ] audio_path = np.array(crema_df.Paths[crema_df.Emotions==1])[1]
    data, sample_rate = librosa.load(audio_path)
    visualize(data, sample_rate, "angry")
    Audio(audio_path)
```



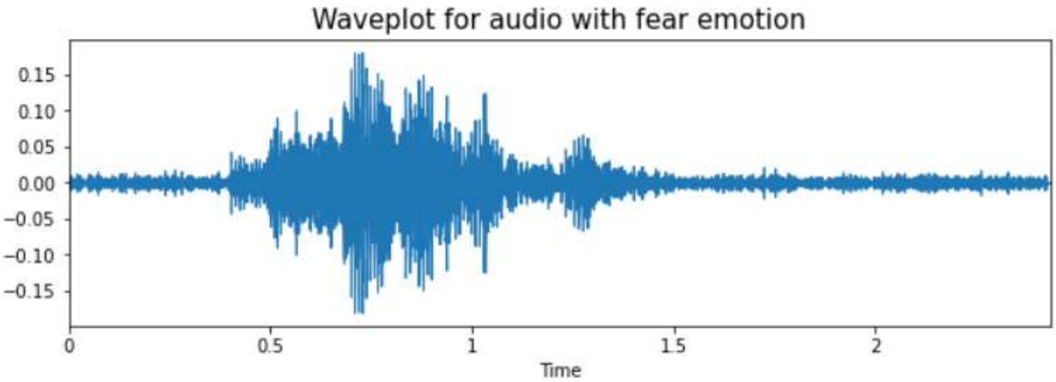
0:00 / 0:02

```
[ ] audio_path = np.array(crema_df.Paths[crema_df.Emotions==2])[2]
    data, sample_rate = librosa.load(audio_path)
    visualize(data, sample_rate, "disgust")
    Audio(audio_path)
```

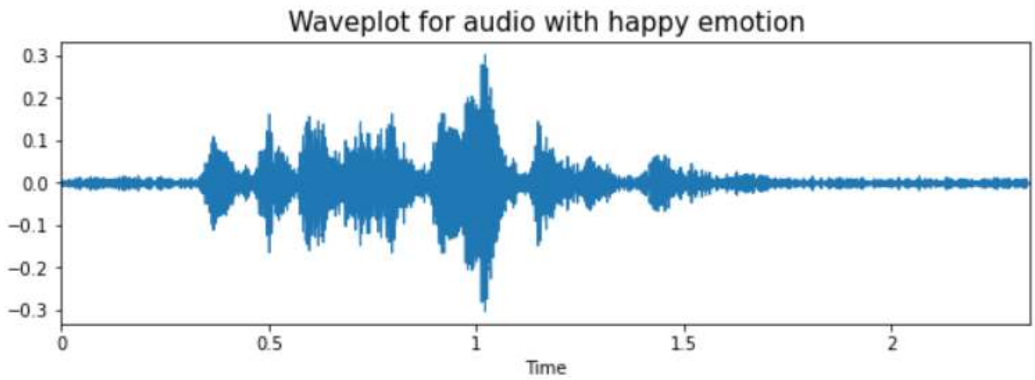




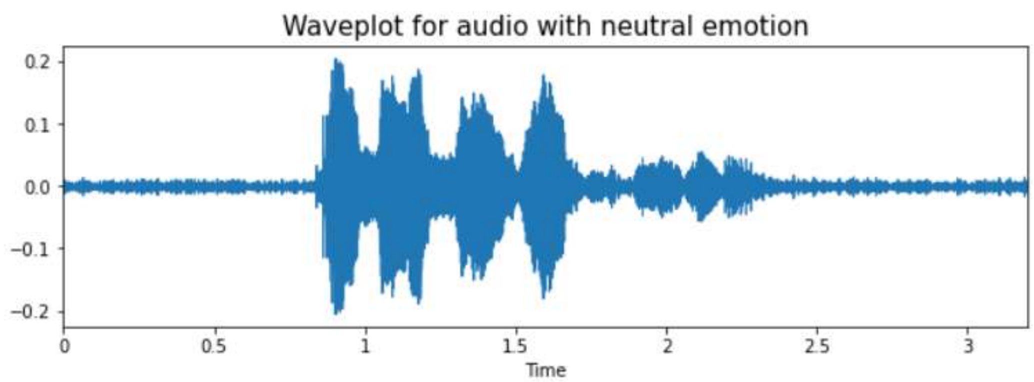

```
[ ] audio_path = np.array(crema_df.Paths[crema_df.Emotions==3])[3]
    data, sample_rate = librosa.load(audio_path)
    visualize(data, sample_rate, "fear")
    Audio(audio_path)
```



```
audio_path = np.array(crema_df.Paths[crema_df.Emotions==4])[4]
data, sample_rate = librosa.load(audio_path)
visualize(data, sample_rate, "happy")
Audio(audio_path)
```




```
audio_path = np.array(crema_df.Paths[crema_df.Emotions==5])[5]
data, sample_rate = librosa.load(audio_path)
visualize(data, sample_rate, "neutral")
Audio(audio_path)
```



Data splitting:

We split the original data into 70% train set and 30% test set, then we split the train set into 95% training data and 5% validation data.

```
train_data, test_data, labels_train1D, labels_test1D = train_test_split(Audio_List,Labels, test_size=0.30, random_state=42)
train_data,val_data, labels_train1D, labels_val1D = train_test_split(train_data,labels_train1D, test_size=0.05, random_state=42)
print(len(train_data))
print(len(val_data))
print(len(test_data))
```

```
4948
261
2233
```

Data augmentation:

We attempted using data augmentation to train the model on more data in different forms than the original audio files to lower the chances of overfitting the model on the given training data.

We used noise and shifting of the audio files as data augmentation. However, best results reached were when the data was augmented with just noise.

```
[ ]  def noise(data):
         noise_amp = 0.035*np.random.uniform()*np.amax(data)
         data = data + noise_amp*np.random.normal(size=data.shape[0])
         return data

     def shift(data):
         shift_range = int(np.random.uniform(low=-5, high = 5)*1000)
         return np.roll(data, shift_range)
```

```
[ ]  train_Aug=[]
     for i in range(len(train_data2)):
       train_Aug.append(train_data2[i])

     for i in range(len(train_data2)):
       noise_data = noise(train_data2[i])
       train_Aug.append(noise_data)
       labels_train1D2.append(labels_train1D2[i])

     for i in range(len(train_data2)):
       shift_data = shift(train_data2[i])
       train_Aug.append(shift_data)
       labels_train1D2.append(labels_train1D2[i])

     print(len(train_Aug))
     print(len(labels_train1D2))
     print(len(train_data2))
```

Features extraction:

List of features we used:

Zero Crossing Rate (zcr): The rate of sign-changes of the signal during the duration of a particular frame.

Energy (rms): The sum of squares of the signal values, normalized by the respective frame length.

Chroma short-term Fourier transformation (stft): STFT represents information about the classification of pitch and signal structure. It depicts the spike with high values (as evident from the color bar net to the graph) in low values (dark regions).

Mel-Frequency Cepstral Coefficients (mfcc): The MFCC feature extraction technique basically includes windowing the signal, applying the DFT, taking the log of the magnitude, and then warping the frequencies on a Mel scale, followed by applying the inverse DCT.

```
def zcr(data, frame_length=1024, hop_length=512):
    zcr = lr.feature.zero_crossing_rate(y=data, frame_length=frame_length, hop_length=hop_length)

    return np.squeeze(zcr)
```

```
[ ] def extract_features(Audio_List, samp_freq, frame_length=2048, hop_length=512, sr):
        result = np.array([])
        result = np.hstack((result,
                            zcr(Audio_List, frame_length, hop_length)))

        rms = np.mean(lr.feature.rms(y=Audio_List).T, axis=0)
        result = np.hstack((result, rms))

        stft = np.abs(lr.stft(Audio_List))
        chroma_stft = np.mean(lr.feature.chroma_stft(S=stft, sr=8000).T, axis=0)
        result = np.hstack((result, chroma_stft))

        mfcc = np.mean(lr.feature.mfcc(y=Audio_List, sr=8000).T, axis=0)
        result = np.hstack((result, mfcc))

        return result
```

Feature space:

We attempted models with the feature space including the audio files themselves while down sampling them to 8000 Hz only to not take up all the RAM and cause the session to crash but the test accuracy was really low and hasn't improved until we used our feature space as only the features we extracted from the audio file as listed above.

Audio included:

```
features=[]
for i in range(len(train_data)):

    res1 = extract_features(train_data[i],8000)
    result = np.array(res1)
    features.append(result)

features_val=[]
for i in range(len(val_data)):

    res1 = extract_features(val_data[i],8000)
    result = np.array(res1)
    features_val.append(result)

features_test=[]
for i in range(len(test_data)):

    res1 = extract_features(test_data[i],8000)
    result = np.array(res1)
    features_test.append(result)
```

```
[ ]
features = np.array(features)
features = features[:,:,np.newaxis]
print(features.shape)

features_val = np.array(features_val)
features_val = features_val[:,:,np.newaxis]
print(features_val.shape)

features_test = np.array(features_test)
features_test = features_test[:,:,np.newaxis]
print(features_test.shape)
```

```
train_data = np.array(train_data)
train_data = train_data[:,:,np.newaxis]
print(train_data.shape)

val_data = np.array(val_data)
val_data = val_data[:,:,np.newaxis]
print(val_data.shape)

test_data = np.array(test_data)
test_data = test_data[:,:,np.newaxis]
print(test_data.shape)
```

```
(4948, 16000, 1)
(261, 16000, 1)
(2233, 16000, 1)
```

```
final_train_aug = np.hstack((features, train_data))
final_train_aug = np.array(final_train_aug)
print(final_train_aug.shape)

final_val = np.hstack((features_val, val_data))
final_val = np.array(final_val)
print(final_val.shape)

final_test = np.hstack((features_test, test_data))
final_test = np.array(final_test)
print(final_test.shape)
```

Audio not included:

```
features_Aug=[]
for i in range(len(train_data)):

    res1 = extract_features2(train_data[i],16000)
    result = np.array(res1)
    features_Aug.append(result)

features_val2=[]
for i in range(len(val_data)):

    res1 = extract_features2(val_data[i],16000)
    result = np.array(res1)
    features_val2.append(result)

features_test2=[]
for i in range(len(test_data)):

    res1 = extract_features2(test_data[i],16000)
    result = np.array(res1)
    features_test2.append(result)
```

The best model:

After many attempts that are all present in the notebook, we reached a **train accuracy of 64%, validation accuracy of 51% and test accuracy of 48.3%.**

We tried to lower the chanced of overfitting by adding dropout layers, L2 regularization and data augmentation as mentioned before.

We used reduce on plateau to modify the learning rate when the model reaches a steady state and also early stopping to stop the model even if the epochs are not done yet if it’s not learning anymore and the results are repetitive.

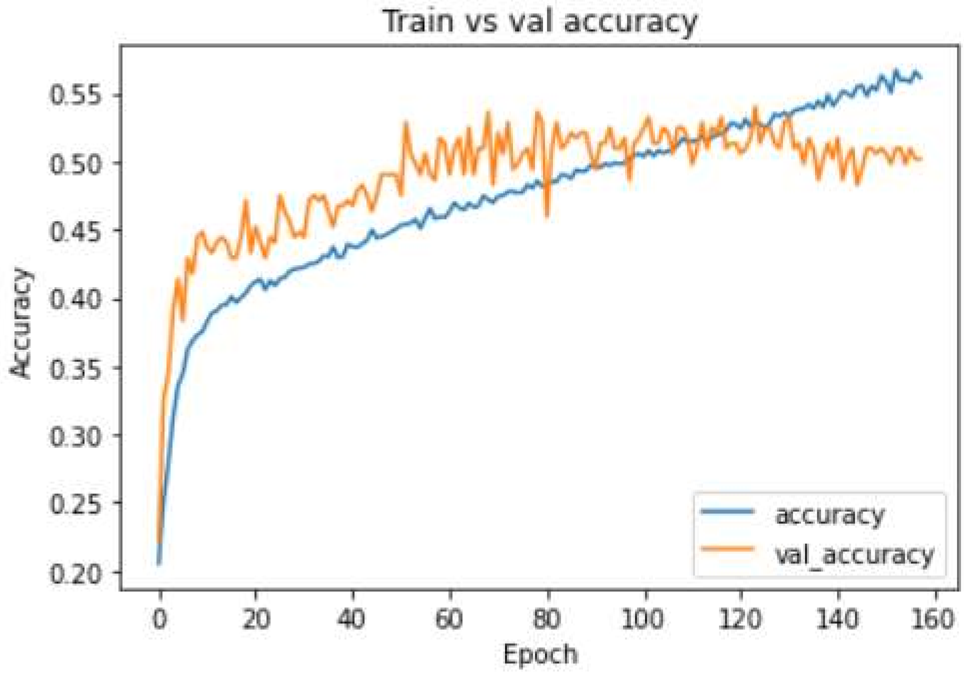
The model:

```python
model = tf.keras.Sequential()
model.add(layers.Conv1D(512, kernel_size=(7), activation='relu',strides=1,use_bias=True,bias_initializer="zeros",kernel_regularizer=l2(0), input_shape=(features_Aug.shape[1],1)))
model.add(layers.MaxPooling1D(pool_size=(5),strides=2))
model.add(layers.Dropout(0.4))

model.add(layers.Conv1D(512, kernel_size=(5), activation='relu',strides=1,use_bias=True,bias_initializer="zeros",kernel_regularizer=l2(0), input_shape=(features_Aug.shape[1],1)))
model.add(layers.MaxPooling1D(pool_size=(5),strides=2))
model.add(layers.Dropout(0.4))

model.add(layers.Conv1D(128, kernel_size=(3), activation='relu',strides=1,use_bias=True,bias_initializer="zeros",kernel_regularizer=l2(0), input_shape=(features_Aug.shape[1],1)))
model.add(layers.MaxPooling1D(pool_size=(5),strides=2))
model.add(layers.Dropout(0.4))

model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dropout(0.4))

model.add(layers.Dense(6, activation='softmax'))
opt = keras.optimizers.Adam(learning_rate=0.0001)
model.compile(loss='sparse_categorical_crossentropy', optimizer=opt ,metrics=['accuracy'])
model.summary()
```

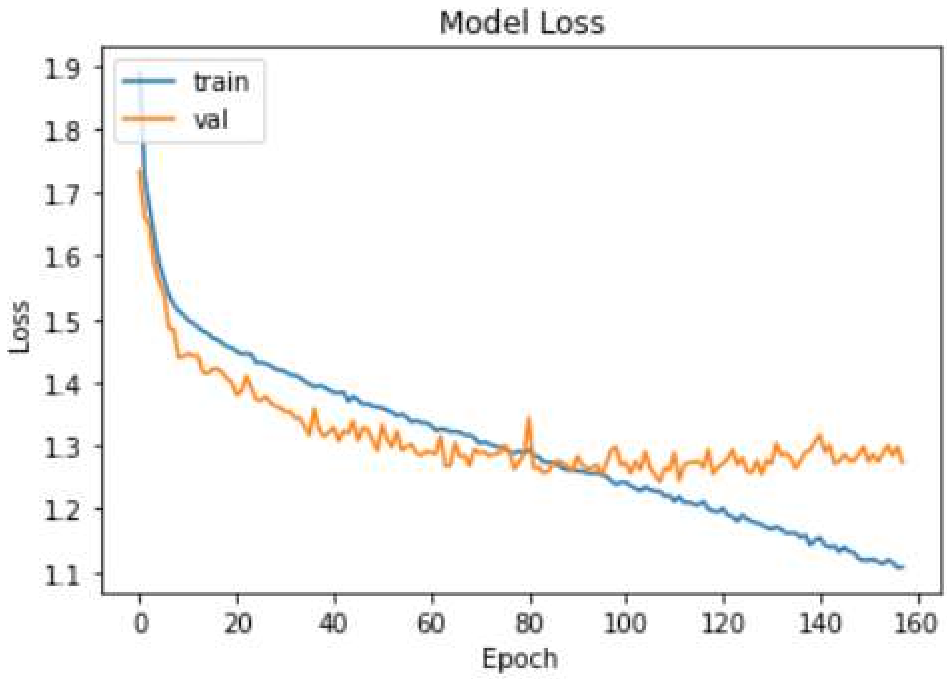
The model summary:

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_8 (Conv1D)           (None, 121, 512)          4096
 max_pooling1d_8 (MaxPooling  (None, 59, 512)          0
 1D)
 dropout_11 (Dropout)        (None, 59, 512)           0
 conv1d_9 (Conv1D)           (None, 55, 512)           1311232
 max_pooling1d_9 (MaxPooling  (None, 26, 512)          0
 1D)
 dropout_12 (Dropout)        (None, 26, 512)           0
 conv1d_10 (Conv1D)          (None, 24, 128)           196736
 max_pooling1d_10 (MaxPoolin  (None, 10, 128)          0
g1D)
 dropout_13 (Dropout)        (None, 10, 128)           0
 flatten_3 (Flatten)         (None, 1280)
0
 dense_6 (Dense)             (None, 256)               327936
 dropout_14 (Dropout)        (None, 256)
0
 dense_7 (Dense)             (None, 6)                 1542
=================================================================
Total params: 1,841,542
Trainable params: 1,841,542
Non-trainable params: 0
```

Accuracy plot:



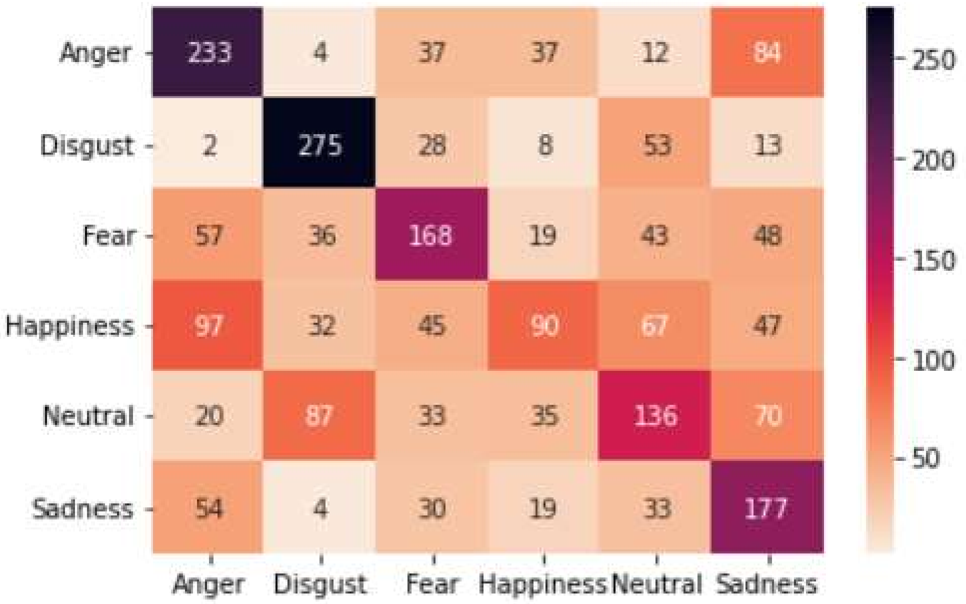
Loss plot:



Model evaluation:

	precision	recall	f1-score	support
0	0.50	0.57	0.54	407
1	0.63	0.73	0.67	379
2	0.49	0.45	0.47	371
3	0.43	0.24	0.31	378
4	0.40	0.36	0.38	381
5	0.40	0.56	0.47	317
accuracy			0.48	2233
macro avg	0.48	0.48	0.47	2233
weighted avg	0.48	0.48	0.47	2233

```
val accuracy: 50.191569328308105
train accuracy: 64.61870074272156
test accuracy 48.32064487236901
```



2D Model

We loaded the data the same way we did in the 1D model, the splitting of the data was also the same, the only different parts were the features we extracted from the data and how we augmented the data.

Feature extraction:

Mel spectrogram: Mel spectrogram is a spectrogram that is converted to a Mel scale. A spectrogram is a visualization of the frequency spectrum of a signal, where the frequency spectrum of a signal is the frequency range that is contained by the signal. The Mel scale mimics how the human ear works, with research showing humans don't perceive frequencies on a linear scale. Humans are better at detecting differences at lower frequencies than at higher frequencies.

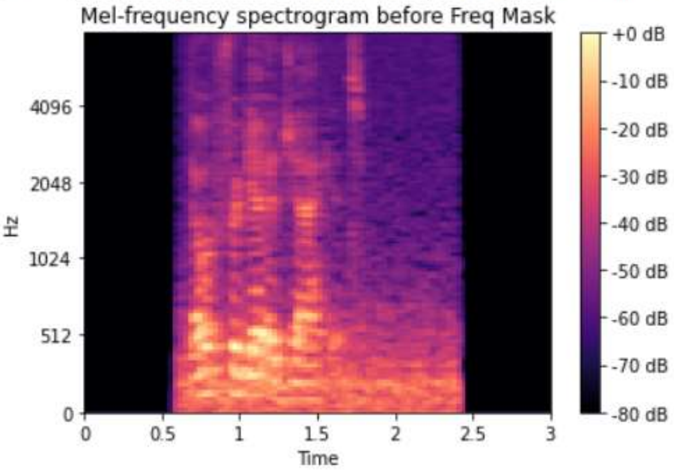
```
import tensorflow_io as tfio

def mel_spectrogram(train_Aug):
    S = lr.feature.melspectrogram(train_Aug, sr=16000, n_fft=2048, hop_length=512, n_mels=128)
    mel_spec_db=lr.power_to_db(S, ref=np.max)
    return mel_spec_db
```

```
train_set_spectorgram = np.zeros((len(train_Aug)*2,128,94))
for i in range(len(train_Aug)):
  train_set_spectorgram[i]= mel_spectrogram(train_Aug[i])
```

The plot:

```
[8]  fig, ax = plt.subplots()
     mel_spectrogram = librosa.display.specshow(train_set_spectorgram[0], x_axis='time',
                              y_axis='mel', sr=16000,
                              fmax=8000, ax=ax)
     fig.colorbar(mel_spectrogram, ax=ax, format='%+2.0f dB')
     ax.set(title='Mel-frequency spectrogram before Freq Mask')
```

```
[Text(0.5, 1.0, 'Mel-frequency spectrogram before Freq Mask')]
```



Data Augmentation:

Frequency masking: An audio compression technique that eliminates sounds that are quieter when compared to sounds with similar frequencies that are much louder.

```
[9]  def freq_mask(mel_spec_db):
         freq_mask = tfio.audio.freq_mask(mel_spec_db, param=10)
         return freq_mask
     j=0
     for i in range(len(train_Aug),len(train_Aug)*2):
       train_set_spectorgram[i]= freq_mask(train_set_spectorgram[j])
       labels_train.append(labels_train[j])
       j+=1
```

The best model:

After many attempts that are all present in the notebook, we reached a **train accuracy of 57%, validation accuracy of 50% and test accuracy of 52%.**

We tried to lower the chanced of overfitting by adding dropout and batch normalization layers, L1 and L2 regularization and data augmentation as mentioned before.

We used reduce on plateau to modify the learning rate when the model reaches a steady state and also early stopping to stop the model even if the epochs are not done yet if it's not learning anymore and the results are repetitive.

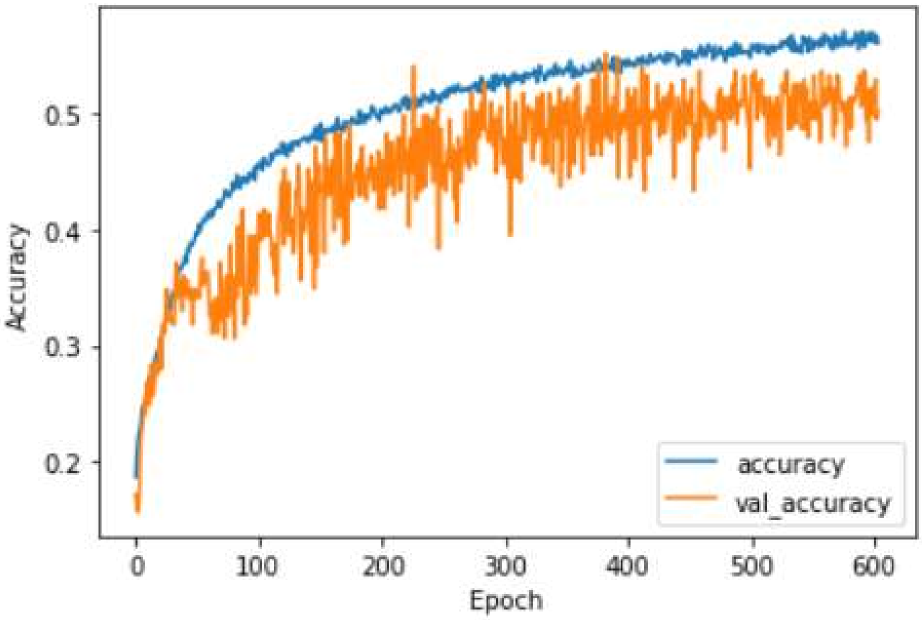
The model:

```
[12] nclass =len(np.unique(Labels))
     inp = Input(shape=(128, 94, 1))

     norm_inp = BatchNormalization()(inp)
     img_1 = Convolution2D(16, kernel_size=(3, 3), activation=activations.relu,kernel_regularizer=l1_l2(l1=0.01, l2=0.01))(norm_inp)
     img_1 = Convolution2D(16, kernel_size=(3, 3), activation=activations.relu,kernel_regularizer=l1_l2(l1=0.01, l2=0.01))(img_1)
     img_1 = MaxPooling2D(pool_size=(3, 3))(img_1)
     img_1 = Dropout(rate=0.1)(img_1)
     img_1 = Convolution2D(32, kernel_size=3, activation=activations.relu,kernel_regularizer=l1_l2(l1=0.01, l2=0.01))(img_1)
     img_1 = Convolution2D(32, kernel_size=3, activation=activations.relu,kernel_regularizer=l1_l2(l1=0.01, l2=0.01))(img_1)
     img_1 = MaxPooling2D(pool_size=(3, 3))(img_1)
     img_1 = Dropout(rate=0.1)(img_1)
     img_1 = Convolution2D(128, kernel_size=3, activation=activations.relu,kernel_regularizer=l1_l2(l1=0.01, l2=0.01))(img_1)
     img_1 = GlobalMaxPool2D()(img_1)
     img_1 = Dropout(rate=0.1)(img_1)

     dense_1 = BatchNormalization()(Dense(128, activation=activations.relu)(img_1))
     dense_1 = BatchNormalization()(Dense(128, activation=activations.relu)(dense_1))
     dense_1 = Dense(nclass, activation=activations.softmax)(dense_1)

     model = models.Model(inputs=inp, outputs=dense_1)
     opt = keras.optimizers.RMSprop(lr=0.00001, decay=1e-6)
     model.compile( loss = losses.sparse_categorical_crossentropy,optimizer = opt, metrics=['accuracy'])
     model.summary()
```

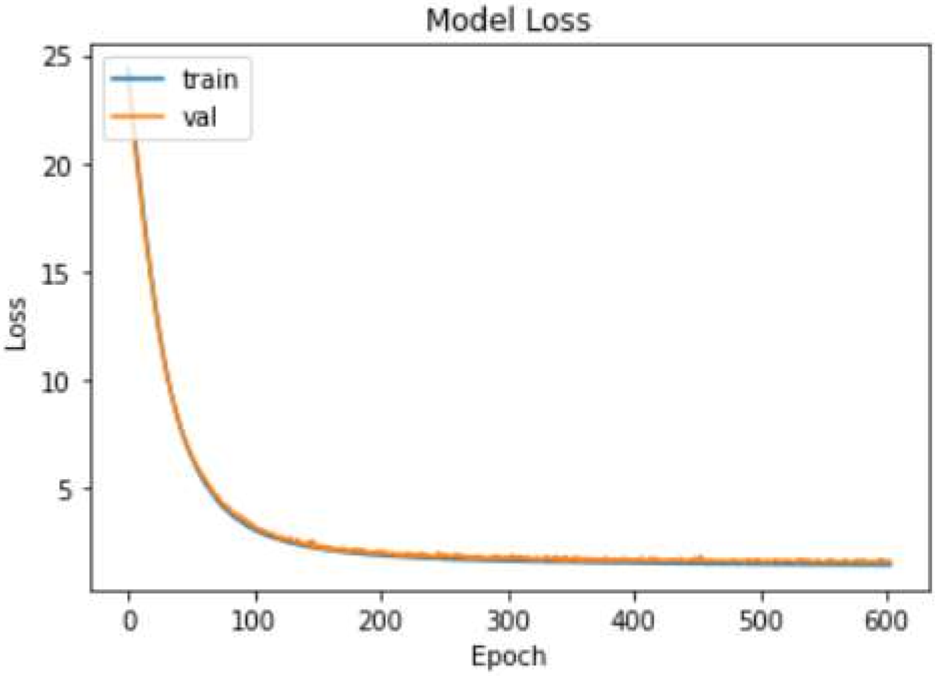
The model summary:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 128, 94, 1)]      0
 batch_normalization (BatchN  (None, 128, 94, 1)       4
 ormalization)
 conv2d (Conv2D)             (None, 126, 92, 16)       160
 conv2d_1 (Conv2D)           (None, 124, 90, 16)       2320
 max_pooling2d (MaxPooling2D  (None, 41, 30, 16)       0
 )
 dropout (Dropout)           (None, 41, 30, 16)
0
 conv2d_2 (Conv2D)           (None, 39, 28, 32)        4640
 conv2d_3 (Conv2D)           (None, 37, 26, 32)
9248
 max_pooling2d_1 (MaxPooling  (None, 12, 8, 32)        0
 2D)
 dropout_1 (Dropout)         (None, 12, 8, 32)         0
 conv2d_4 (Conv2D)           (None, 10, 6, 128)        36992
 global_max_pooling2d (Globa  (None, 128)              0
 lMaxPooling2D)
 dropout_2 (Dropout)         (None, 128)               0
 dense (Dense)               (None, 128)               16512
 batch_normalization_1 (Batc  (None, 128)              512
 hNormalization)
 dense_1 (Dense)             (None, 128)               16512
 batch_normalization_2 (Batc  (None, 128)              512
 hNormalization)
 dense_2 (Dense)             (None, 6)                 774
=================================================================
Total params: 88,186
Trainable params: 87,672
Non-trainable params: 514
```

Accuracy plot:



Loss plot:



Model evaluation:

```
              precision    recall  f1-score   support

           0       0.47      0.75      0.58       404
           1       0.76      0.48      0.59       386
           2       0.43      0.51      0.47       372
           3       0.51      0.40      0.45       367
           4       0.57      0.40      0.47       393
           5       0.54      0.59      0.56       311

    accuracy                           0.52      2233
   macro avg       0.55      0.52      0.52      2233
weighted avg       0.55      0.52      0.52      2233
```

```
val accuracy: 50.191569328308105
train accuracy: 57.083672285079956
test accuracy 52.21674876847291
```