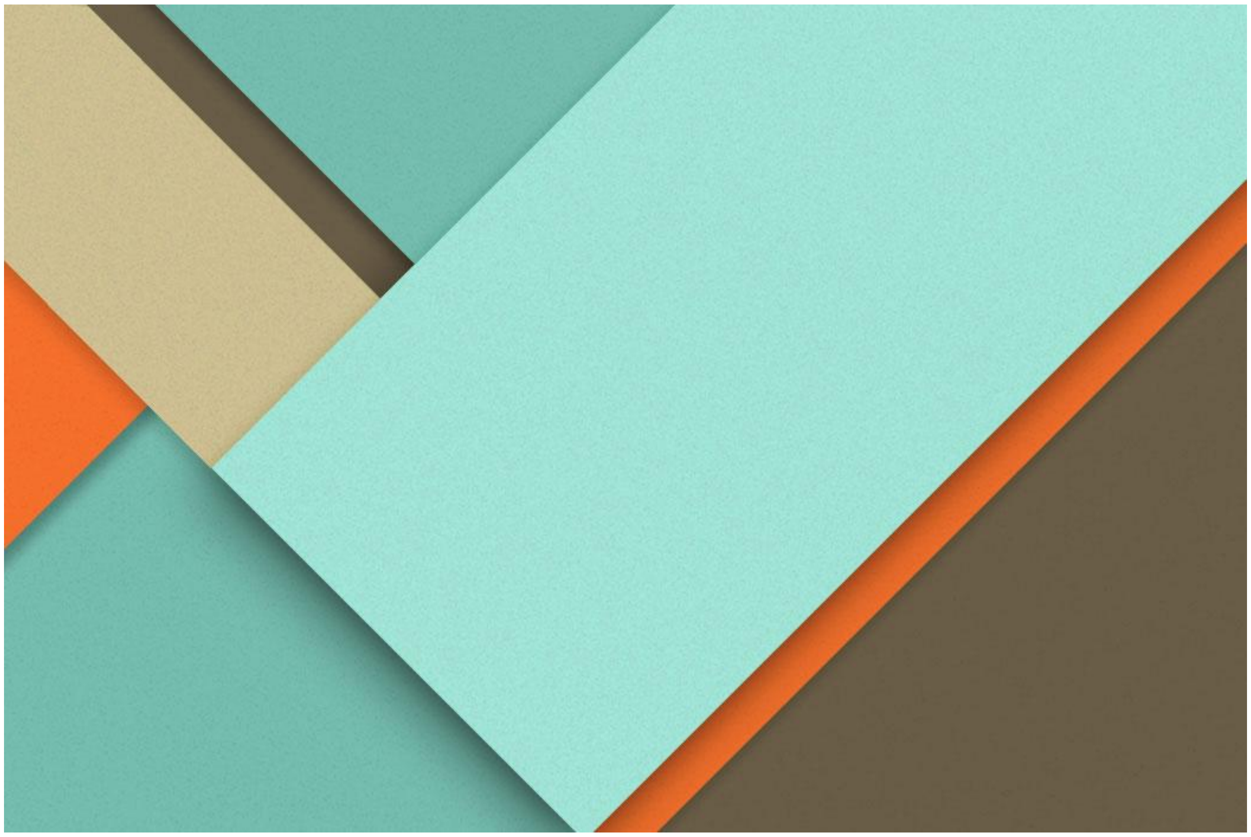


Speech Emotion Recognition

Pattern Recognition

—

Name	ID
Toka Ashraf Abo Alwafa	19015539
Rowan Nasser Edrees	19015686
Nada Mohamed Ibrahim	19016782

Problem Statement

The problem statement describes speech emotion recognition (SER) systems as a collection of methodologies that process and classify speech signals to detect the embedded emotions. The assignment involves downloading the CREMA dataset, creating two feature spaces from the audio, building a CNN model from scratch, and comparing the performance of different models based on accuracy, F-Score, and confusion matrices. The steps required are: 1) download and understand the format of the dataset, 2) create the feature space, 3) build the model, and 4) compare the performance of the models.

Code Flow

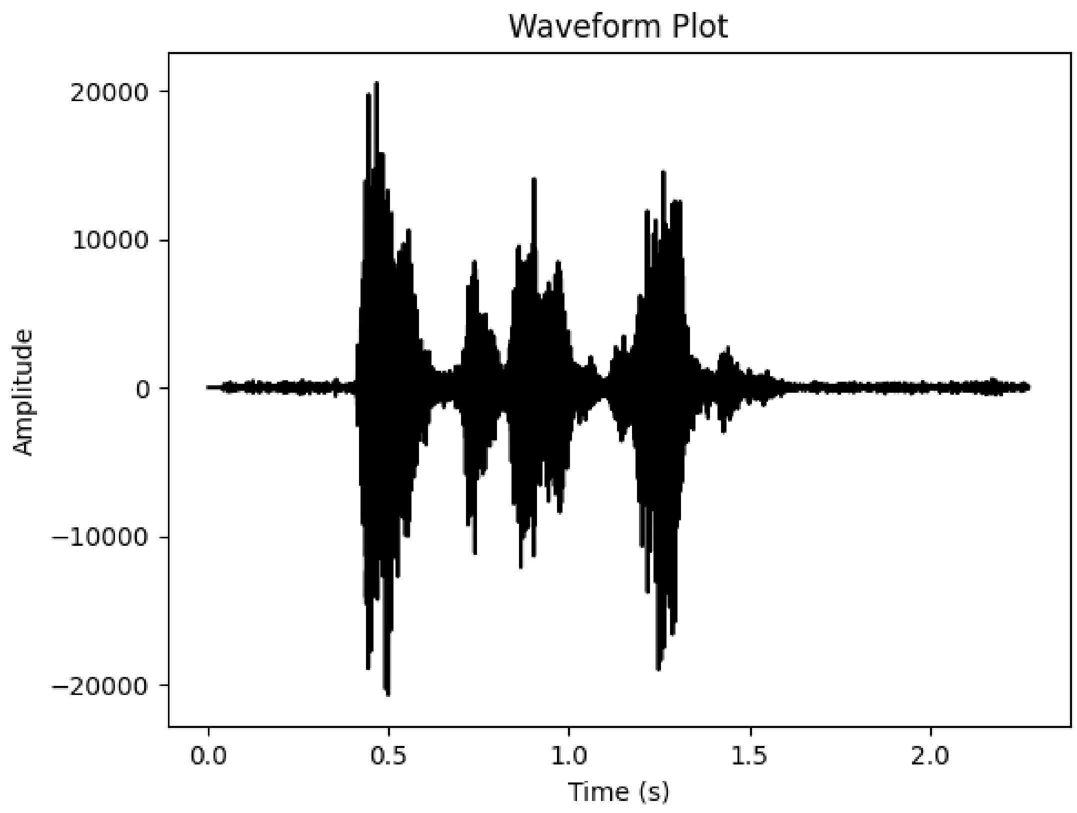
I. Download Dataset

We are using a set of shell commands to to install the Kaggle package, create a directory for storing Kaggle API credentials, copy the API credentials file to the directory, set file permissions for the credentials file, and extract the contents of a zipped dataset file related to speech emotion recognition analysis.

II. Formatting the Dataset

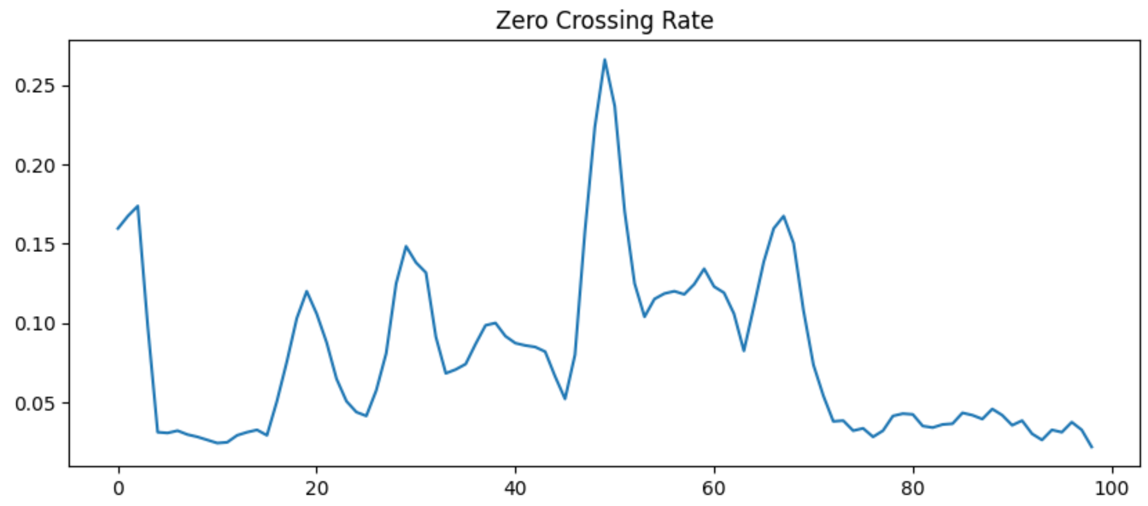
- Firstly, We read the names of audio files from a specified folder path and extract the emotion labels from the filenames based on a predefined mapping in a dictionary. The emotion labels are assigned to a numeric code and stored in a list, which can be used for training a speech emotion recognition model. The resulting labels list contains the numeric emotion labels for each audio file, and the audio_files list contains the names of all the audio files in the folder.
- Then, We play the audio files whose names were read in the previous step. It uses the soundfile and sounddevice libraries to read and play the audio data from each WAV file, respectively. The audio files are played in a loop using the audio_files list, and the program waits until the audio playback is complete before continuing.
- Lastly, We use the wave and matplotlib libraries to plot the waveform of the audio files. It loops over each audio file specified in the audio_files list and reads the audio data from each WAV file using the wave.open() function. The waveform of the audio data is then plotted using the plt.plot() function, and the x-axis shows the time in seconds, and the y-axis shows the amplitude of the audio signal. Finally, the plt.show() function displays the waveform plot for each

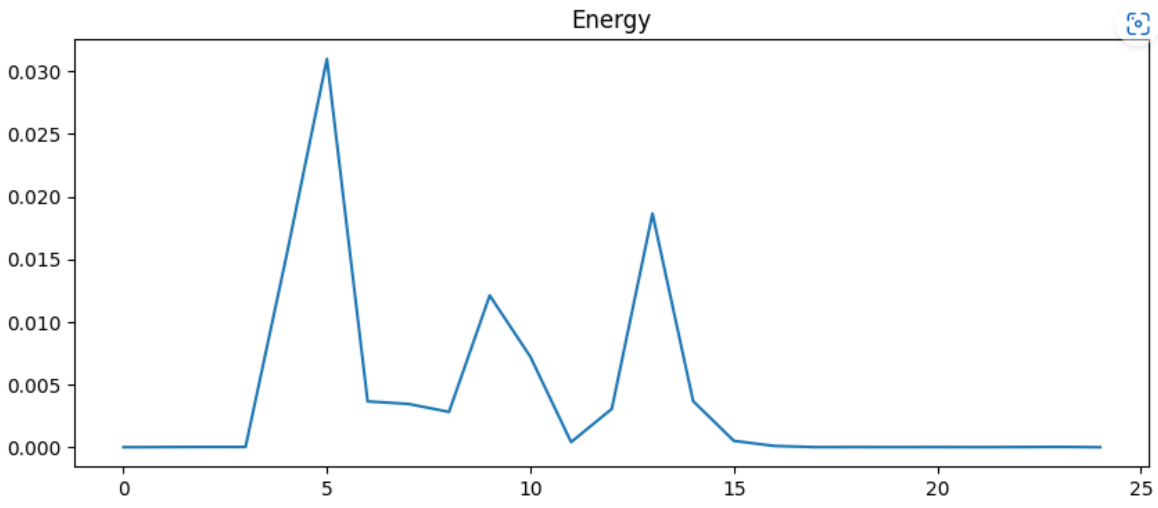
audio file.

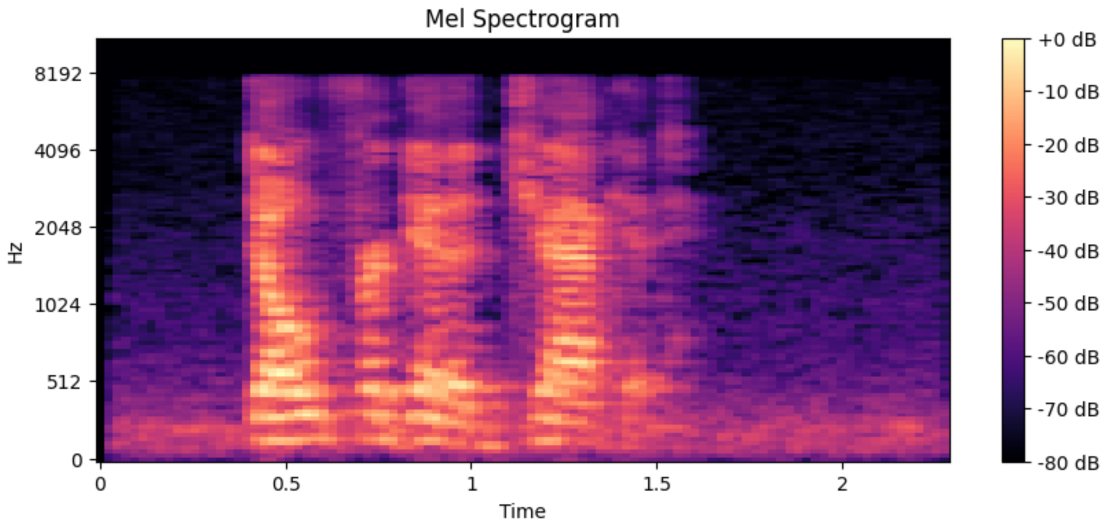


III. Create Feature Space

1. **Visualizing the zero-crossing rate, energy and mel spectrogram**
   We define a function called plot_mel_spectogram() which takes five parameters as input: y (audio time series), sr (sampling rate), zcr (zero-crossing rate), energy (energy of the audio signal), mel_spec_db (mel spectrogram in decibels), and hop_length (number of samples between each frame). The function then generates three plots using the matplotlib and librosa libraries: the zero-crossing rate, energy of the audio signal, and mel spectrogram. These plots help to visualize and understand the characteristics of the audio signal.


Zero Crossing Rate
0.25
0.20
0.15
0.10
0.05
0
20
40
60
80
100


Energy
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
5
10
15
20
25


Mel Spectrogram
8192
4096
2048
1024
512
0
Hz
0
0.5
1
1.5
2
Time
+0 dB
-10 dB
-20 dB
-30 dB
-40 dB
-50 dB
-60 dB
-70 dB
-80 dB

2. **Adding padding for the input audio files:**
   Why: As adding padding to the input feature space before building a CNN model can help to improve the performance of the model by preserving spatial dimensions, improving edge detection, and preventing information loss.
   How: Firstly, loading an audio file as a floating point time series using built in function of load from library of librosa that takes the file path as a parameter and returns:
   - y: audio time series.
   - sr: sampling rate of y
   Secondly, checking the length of y if more than the maximum size chosen then set y to be of the length of this maximum size but if less so calculate the padding by subtracting the length of y from the maximum size and dividing this by 2 then rounding it down to the nearest integer then adding this calculated padding to the start of y and adding the rest (max_size - len(y) - padding) to the end of y after that appending this y to an array to use it to extract the feature space.

3. **Extracting the feature spaces:**
   We process the audio files using the Librosa library, compute various features of the audio signal, such as zero-crossing rate, energy, and mel-spectrogram, and plot the mel-spectrogram using the plot_mel_spectogram() function. The for loop iterates over each audio file after:
   - loading it and adding the padding , calculates the zero-crossing rate of the audio signal using librosa.feature.zero_crossing_rate(),and calculates the root mean square then concatenates arrays along the second axis (columns) and finally append it to an array that representes the dataset of the 1-D model time domain after reshaping it.
   - computes the energy of the audio signal using a frame-by-frame approach then reshaping it and appending to an array .
   - calculates the mel-spectrogram of the audio signal using librosa.feature.melspectrogram(), reshape the dimension, and finally plot the mel-spectrogram using the plot_mel_spectogram() function.

IV. Splitting of The Dataset

Splitting the dataset to 70% training and validation and 30% test

Then splitting the 70% training and validation to 5% validation and 95% training.

V. CNN Model of Time Domain Feature (Zero Crossing Rate)

The model consists of three convolutional layers, max pooling layers, and two fully connected layers:

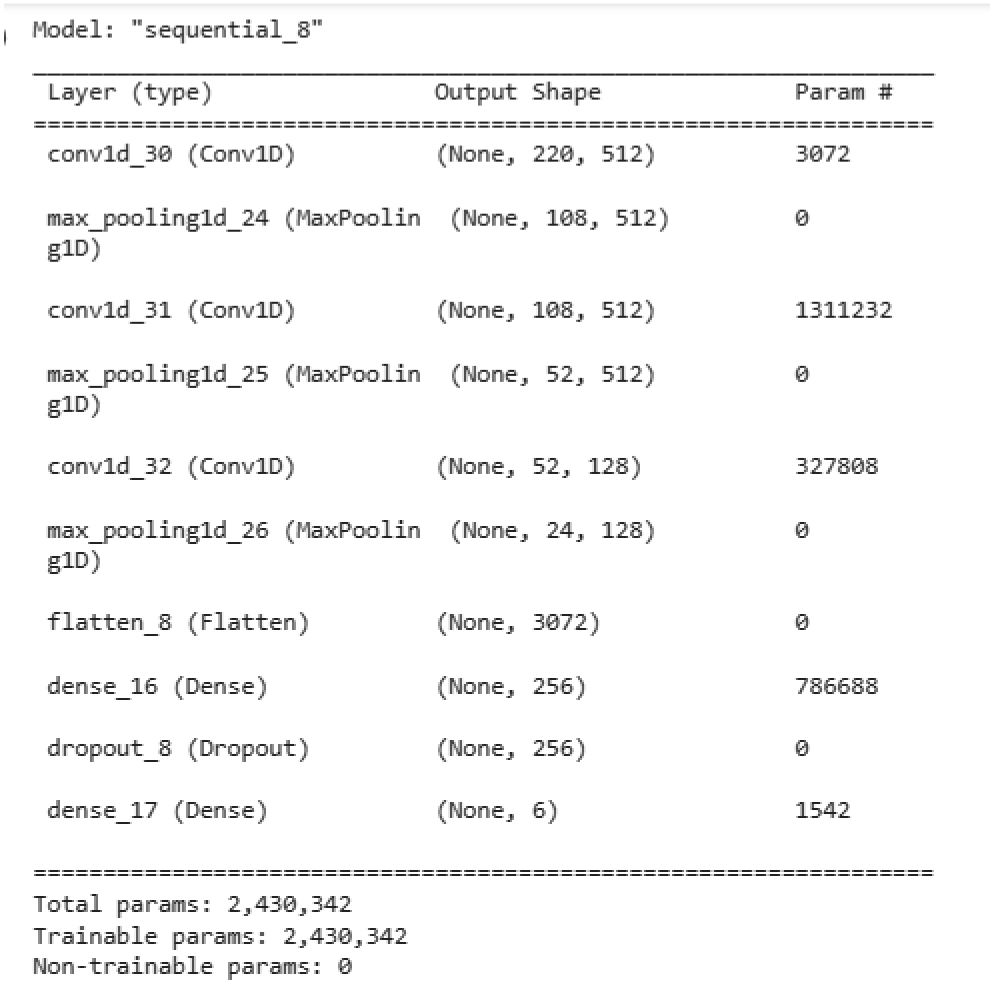
- All layers consists of kernel size of 5 , 1 stride, padding is set to same (this means that the output from the convolutional layer will have the same spatial dimensions as the input by adding padding equally to the beginning and end of the input) and activation is set to relu (that returns the input value if it is positive and zero otherwise and helps to avoid the vanishing gradient problem).
- Number of filters differs from layers to another (first layer >> 512 , second layer >> 512 , third layer >> 128):

  each filter applies a convolution operation to the input data, which involves sliding the filter over the input and computing a dot product between the filter and a small region of the input at a time.

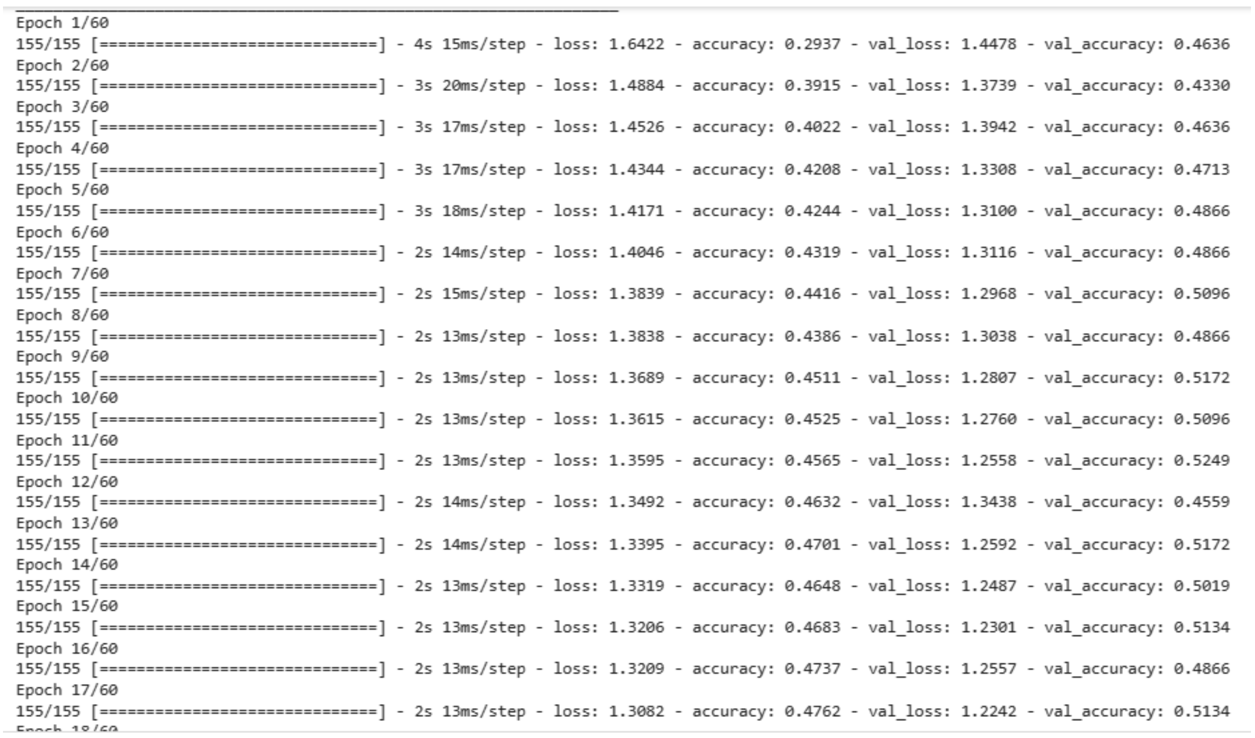
- Each convolutional layer is followed by a max-pooling layer to downsample the output and reduce the dimensionality by taking the maximum value over non-overlapping windows of size 6 (pooling size) , and sliding the window by 2 (number of strides) steps at a time.
- The output from the layers is flatten into 1-D array to be ready to enter the fully connected layer.
- Fully connected layer of 256 units of neurons and activation of relu is added.
- Drop out layer is added to the model by fraction of 40% to prevent the overfitting by any one neuron.
- The output layer is added with 6 units of neurons and activation of softmax (that specifies that the output values should be normalized so that they represent a probability distribution over the 6 classes).

- Adam optimizer is added with learning rate 0.0001 to update the weights of the model during the training to minimize the loss function.
- compiling the model for training and getting its summary.
- Finally , fitting the compiled model on the training data with 60 epoches (number of times to iterate over the entire training dataset) and 32 batch size (number of samples to use in each batch) and using the validation data to evaluate the performance of the model after each epoch.

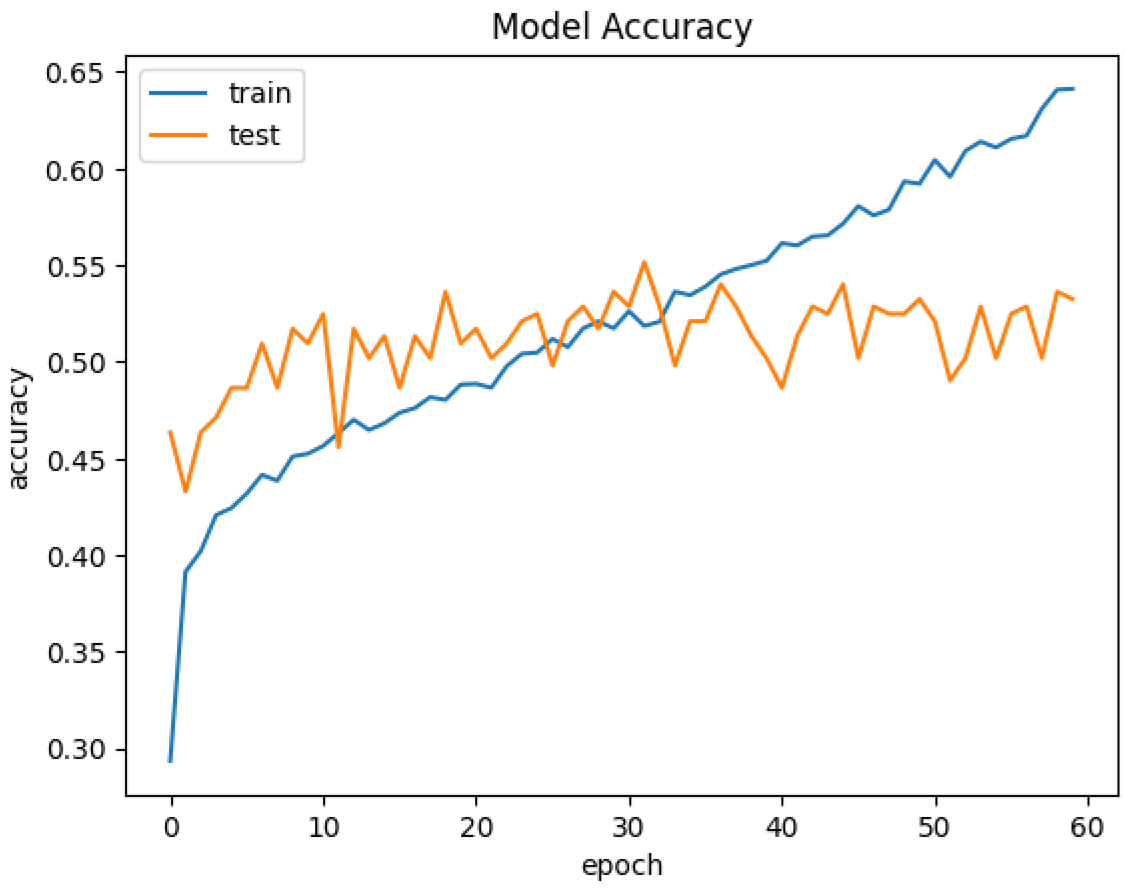
- **Results after applying this model:**

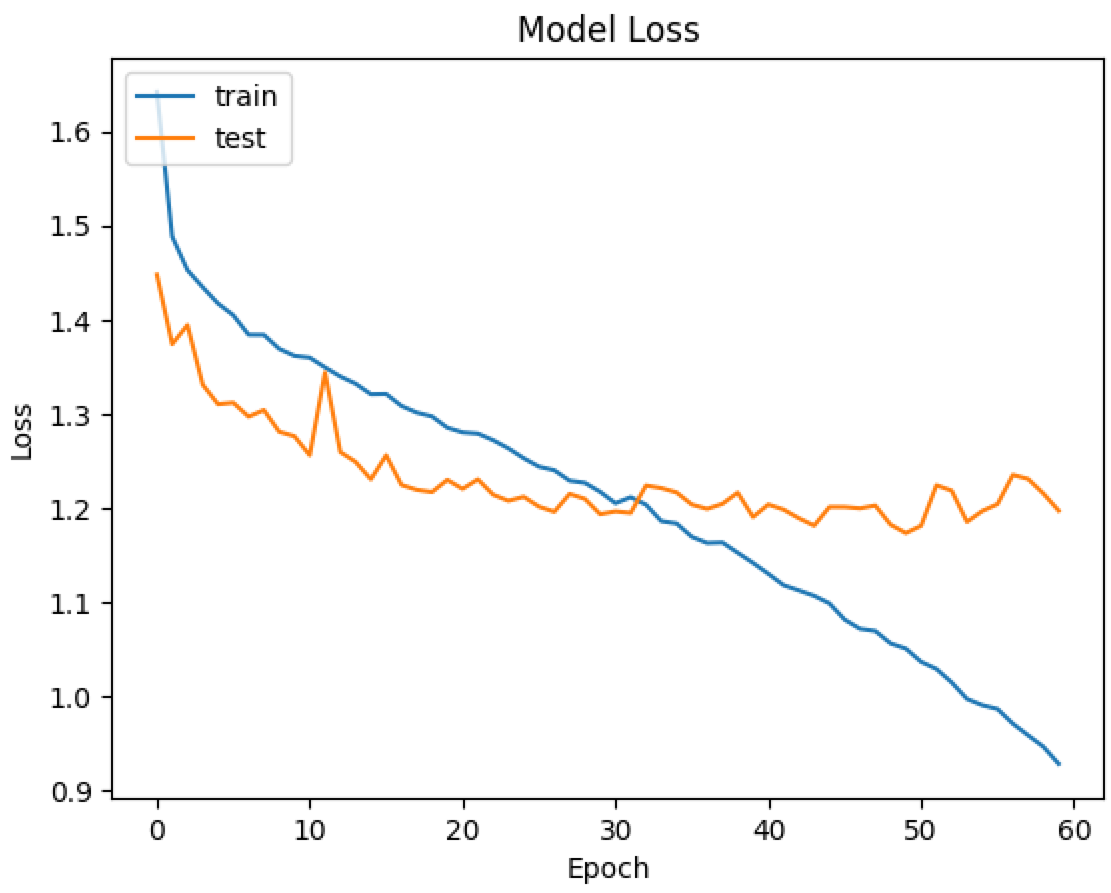


```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_30 (Conv1D)          (None, 220, 512)          3072

 max_pooling1d_24 (MaxPoolin  (None, 108, 512)         0
 g1D)

 conv1d_31 (Conv1D)          (None, 108, 512)          1311232

 max_pooling1d_25 (MaxPoolin  (None, 52, 512)          0
 g1D)

 conv1d_32 (Conv1D)          (None, 52, 128)           327808

 max_pooling1d_26 (MaxPoolin  (None, 24, 128)          0
 g1D)

 flatten_8 (Flatten)         (None, 3072)              0

 dense_16 (Dense)            (None, 256)               786688

 dropout_8 (Dropout)         (None, 256)               0

 dense_17 (Dense)            (None, 6)                 1542

=================================================================
Total params: 2,430,342
Trainable params: 2,430,342
Non-trainable params: 0
```



```
Epoch 1/60
155/155 [==============================] - 4s 15ms/step - loss: 1.6422 - accuracy: 0.2937 - val_loss: 1.4478 - val_accuracy: 0.4636
Epoch 2/60
155/155 [==============================] - 3s 20ms/step - loss: 1.4884 - accuracy: 0.3915 - val_loss: 1.3739 - val_accuracy: 0.4330
Epoch 3/60
155/155 [==============================] - 3s 17ms/step - loss: 1.4526 - accuracy: 0.4022 - val_loss: 1.3942 - val_accuracy: 0.4636
Epoch 4/60
155/155 [==============================] - 3s 17ms/step - loss: 1.4344 - accuracy: 0.4208 - val_loss: 1.3308 - val_accuracy: 0.4713
Epoch 5/60
155/155 [==============================] - 3s 18ms/step - loss: 1.4171 - accuracy: 0.4244 - val_loss: 1.3100 - val_accuracy: 0.4866
Epoch 6/60
155/155 [==============================] - 2s 14ms/step - loss: 1.4046 - accuracy: 0.4319 - val_loss: 1.3116 - val_accuracy: 0.4866
Epoch 7/60
155/155 [==============================] - 2s 15ms/step - loss: 1.3839 - accuracy: 0.4416 - val_loss: 1.2968 - val_accuracy: 0.5096
Epoch 8/60
155/155 [==============================] - 2s 13ms/step - loss: 1.3838 - accuracy: 0.4386 - val_loss: 1.3038 - val_accuracy: 0.4866
Epoch 9/60
155/155 [==============================] - 2s 13ms/step - loss: 1.3689 - accuracy: 0.4511 - val_loss: 1.2807 - val_accuracy: 0.5172
Epoch 10/60
155/155 [==============================] - 2s 13ms/step - loss: 1.3615 - accuracy: 0.4525 - val_loss: 1.2760 - val_accuracy: 0.5096
Epoch 11/60
155/155 [==============================] - 2s 13ms/step - loss: 1.3595 - accuracy: 0.4565 - val_loss: 1.2558 - val_accuracy: 0.5249
Epoch 12/60
155/155 [==============================] - 2s 14ms/step - loss: 1.3492 - accuracy: 0.4632 - val_loss: 1.3438 - val_accuracy: 0.4559
Epoch 13/60
155/155 [==============================] - 2s 14ms/step - loss: 1.3395 - accuracy: 0.4701 - val_loss: 1.2592 - val_accuracy: 0.5172
Epoch 14/60
155/155 [==============================] - 2s 13ms/step - loss: 1.3319 - accuracy: 0.4648 - val_loss: 1.2487 - val_accuracy: 0.5019
Epoch 15/60
155/155 [==============================] - 2s 13ms/step - loss: 1.3206 - accuracy: 0.4683 - val_loss: 1.2301 - val_accuracy: 0.5134
Epoch 16/60
155/155 [==============================] - 2s 13ms/step - loss: 1.3209 - accuracy: 0.4737 - val_loss: 1.2557 - val_accuracy: 0.4866
Epoch 17/60
155/155 [==============================] - 2s 13ms/step - loss: 1.3082 - accuracy: 0.4762 - val_loss: 1.2242 - val_accuracy: 0.5134
```

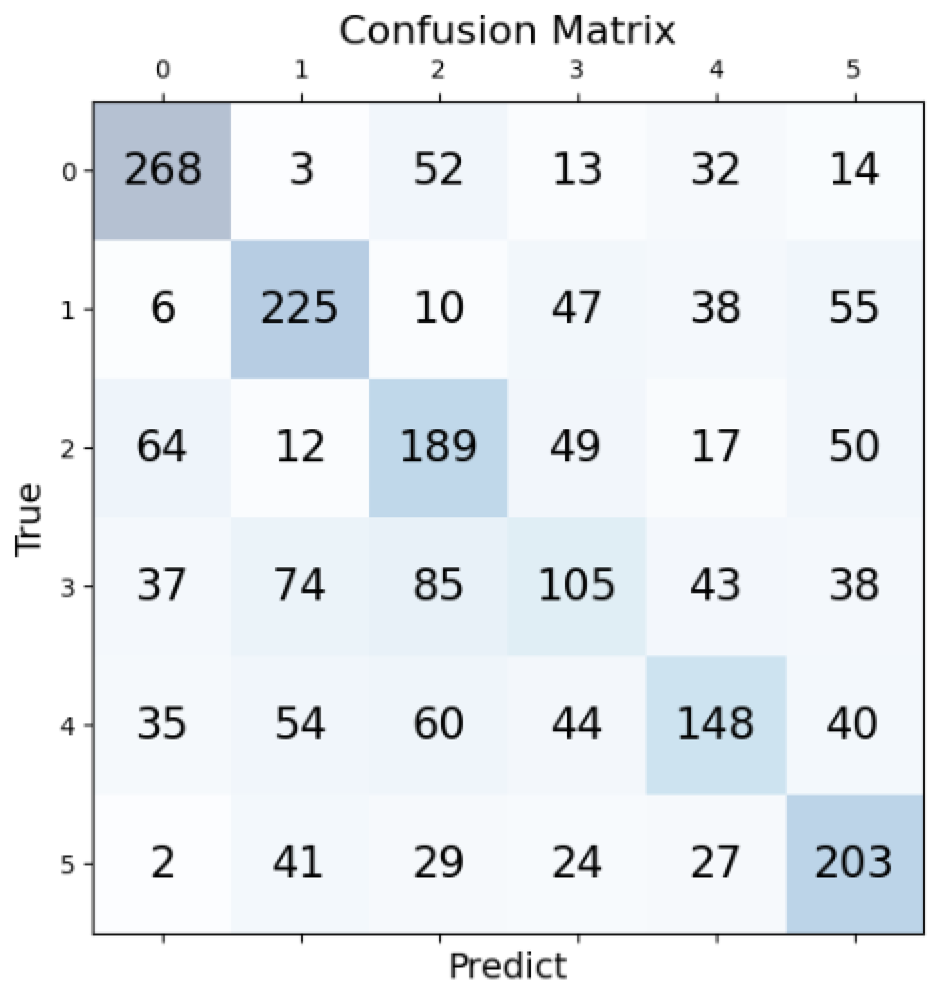




After calculating the accuracy:

```
Accuracy of CNN model 1D = 0.5096282958984375
F1 Score of CNN model 1D = 0.5020600306403878
```

```
              precision    recall  f1-score   support

         ANG       0.65      0.70      0.68       382
         SAD       0.55      0.59      0.57       381
         HAP       0.44      0.50      0.47       381
         FEA       0.37      0.27      0.32       382
         DIS       0.49      0.39      0.43       381
         NEU       0.51      0.62      0.56       326

    accuracy                           0.51      2233
   macro avg       0.50      0.51      0.50      2233
weighted avg       0.50      0.51      0.50      2233
```



VI. Comparison Between Different Models of 1D

Number of layers=3 , Number of Epoches=30

filters	acc	f-score
(512, 512, 128)	0.486789	0.471517
(256, 128, 64)	0.490819	0.484056
(128, 64, 32)	0.473802	0.462482

Number of layers=3 , Number of Epoches=60

filters	acc	f-score
(512, 512, 128)	0.50515	0.49898
(256, 128, 64)	0.50380	0.50006
(128, 64, 32)	0.48096	0.47346

Number of Epoches=30

Layers	acc	f-score
3	0.486789	0.471517
5	0.493058	0.495989
7	0.486789	0.480589

VII. CNN Model of Spectrogram Feature(Spectrogram)

As 1d model we put some of the variables of conv2D in constant numbers :

- All layers consists of kernel size of (5, 5) , (1,1) stride, padding is set to 'same' and activation is set to 'relu'

- Number of filters differs from layers to another (first layer >> 512 , second layer >> 512 , third layer >> 128):

- Each convolutional layer is conducted pooling operation for size 4 and included a dropout of 0.4 and batch normalization

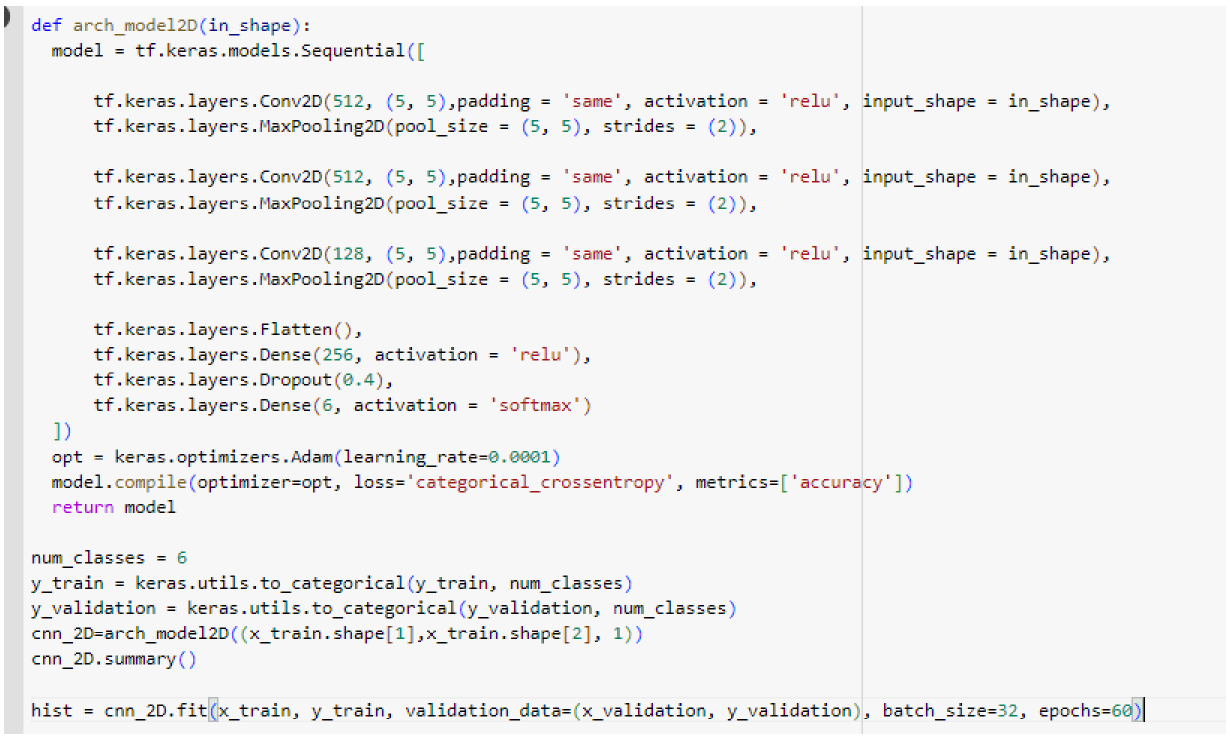
- The output from the layers is flatten into 2-D array to be ready to enter the fully connected layer.

- Fully connected layer of 256 units of neurons and activation of relu is added.

- The output layer is added with 6 units of neurons and activation of softmax

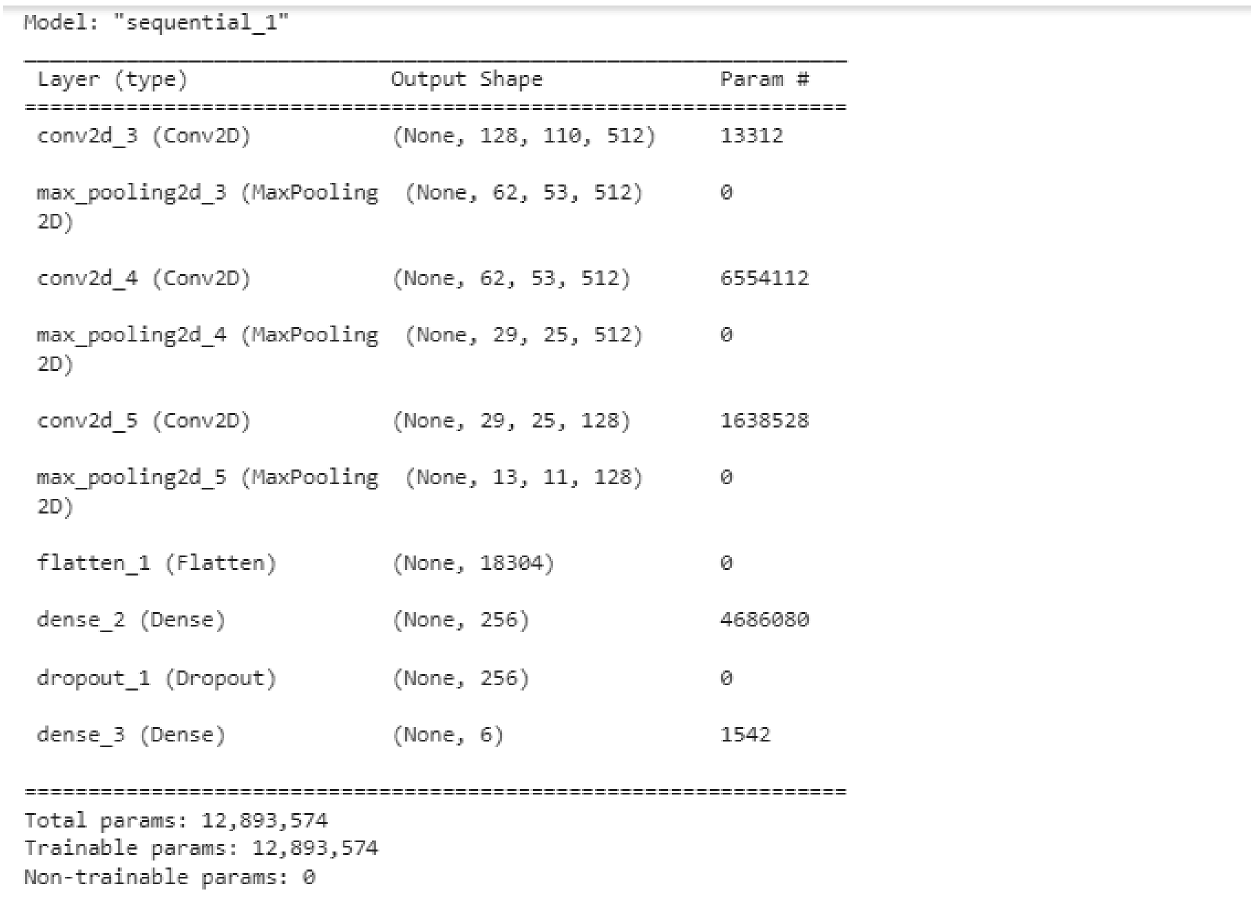
- Adam optimizer is added with learning rate 0.0001

- compiling the model for training and getting its summary.
- Finally , fitting the compiled model on the training data with 60 epoches and 32 batch size

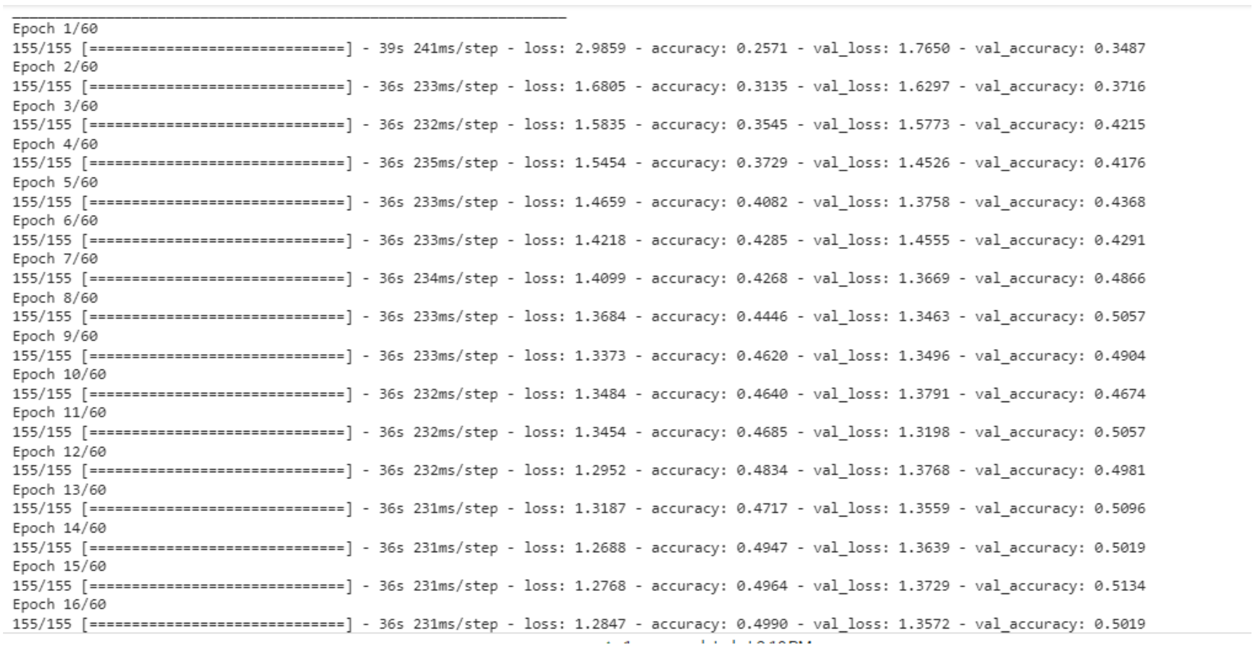


```
def arch_model2D(in_shape):
  model = tf.keras.models.Sequential([

      tf.keras.layers.Conv2D(512, (5, 5),padding = 'same', activation = 'relu', input_shape = in_shape),
      tf.keras.layers.MaxPooling2D(pool_size = (5, 5), strides = (2)),

      tf.keras.layers.Conv2D(512, (5, 5),padding = 'same', activation = 'relu', input_shape = in_shape),
      tf.keras.layers.MaxPooling2D(pool_size = (5, 5), strides = (2)),

      tf.keras.layers.Conv2D(128, (5, 5),padding = 'same', activation = 'relu', input_shape = in_shape),
      tf.keras.layers.MaxPooling2D(pool_size = (5, 5), strides = (2)),

      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(256, activation = 'relu'),
      tf.keras.layers.Dropout(0.4),
      tf.keras.layers.Dense(6, activation = 'softmax')
  ])
  opt = keras.optimizers.Adam(learning_rate=0.0001)
  model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
  return model

num_classes = 6
y_train = keras.utils.to_categorical(y_train, num_classes)
y_validation = keras.utils.to_categorical(y_validation, num_classes)
cnn_2D=arch_model2D((x_train.shape[1],x_train.shape[2], 1))
cnn_2D.summary()

hist = cnn_2D.fit(x_train, y_train, validation_data=(x_validation, y_validation), batch_size=32, epochs=60)
```

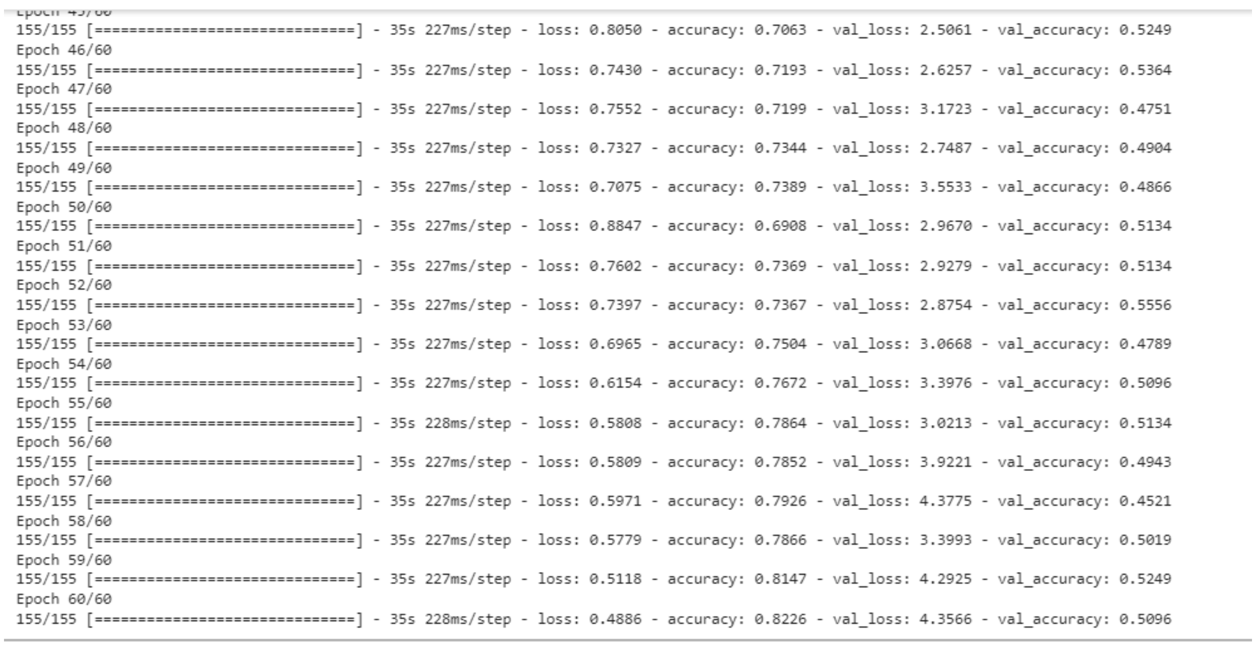
- **Results after applying this model:**



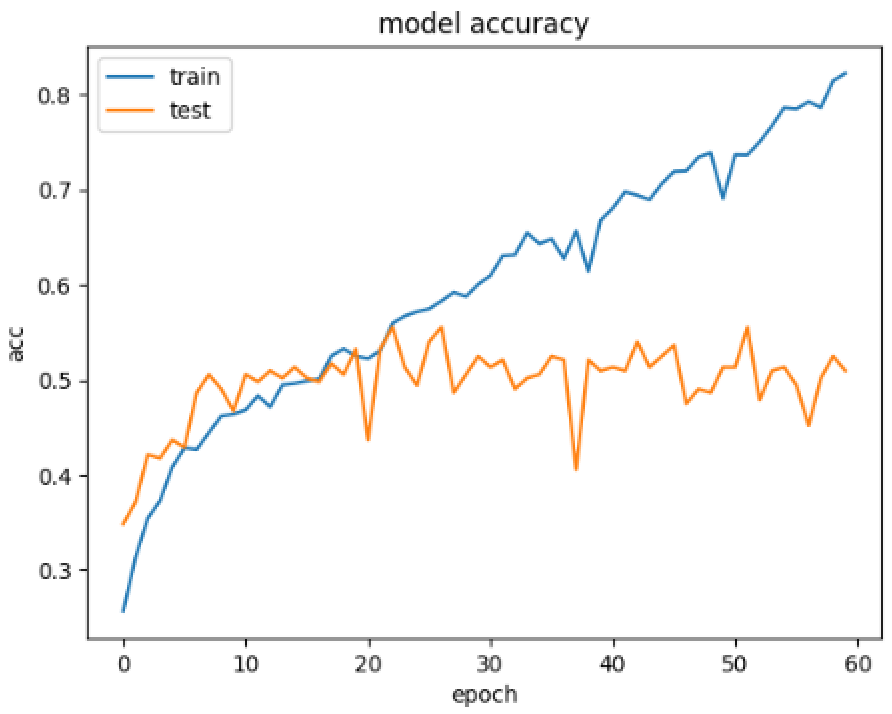
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 128, 110, 512)     13312

 max_pooling2d_3 (MaxPooling  (None, 62, 53, 512)      0
 2D)

 conv2d_4 (Conv2D)           (None, 62, 53, 512)       6554112

 max_pooling2d_4 (MaxPooling  (None, 29, 25, 512)      0
 2D)

 conv2d_5 (Conv2D)           (None, 29, 25, 128)       1638528

 max_pooling2d_5 (MaxPooling  (None, 13, 11, 128)      0
 2D)

 flatten_1 (Flatten)         (None, 18304)             0

 dense_2 (Dense)             (None, 256)               4686080

 dropout_1 (Dropout)         (None, 256)               0

 dense_3 (Dense)             (None, 6)                 1542

=================================================================
Total params: 12,893,574
Trainable params: 12,893,574
Non-trainable params: 0
```

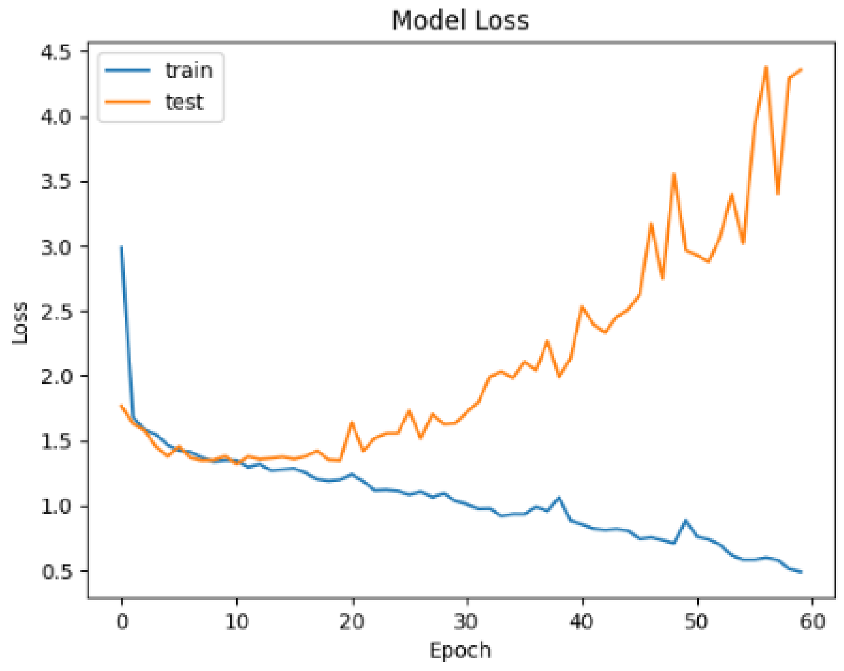


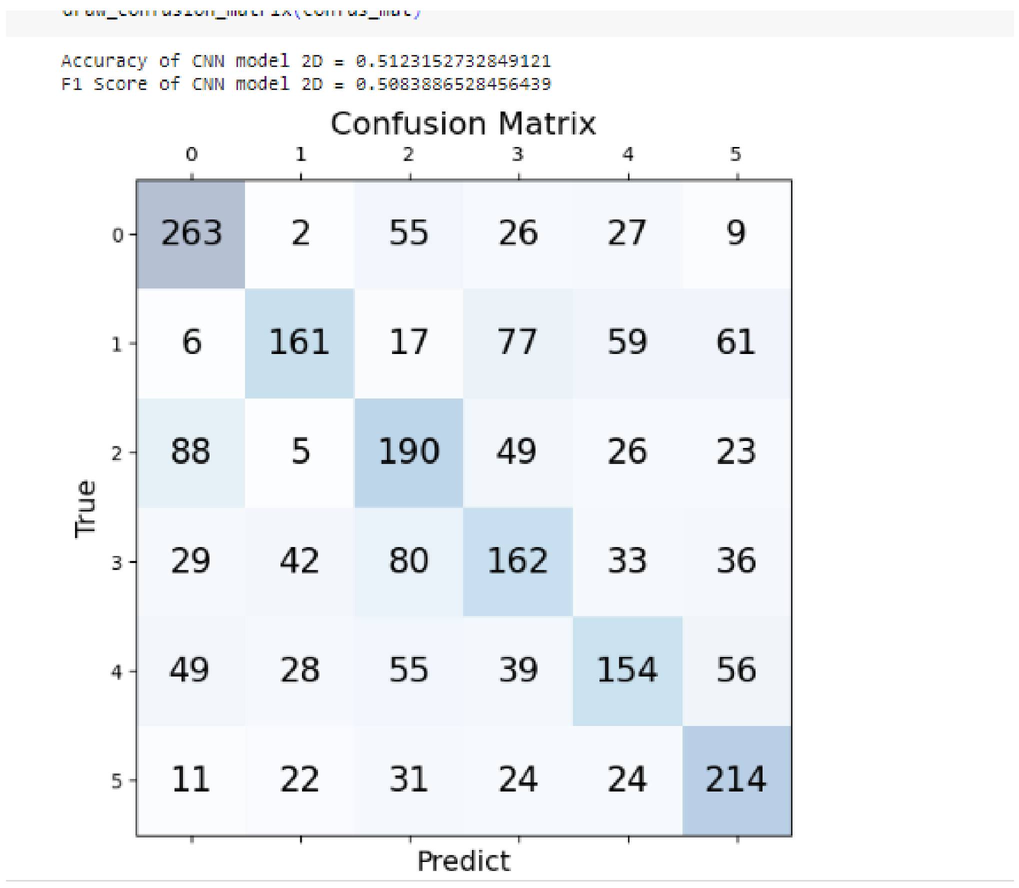
```
Epoch 1/60
155/155 [==============================] - 39s 241ms/step - loss: 2.9859 - accuracy: 0.2571 - val_loss: 1.7650 - val_accuracy: 0.3487
Epoch 2/60
155/155 [==============================] - 36s 233ms/step - loss: 1.6805 - accuracy: 0.3135 - val_loss: 1.6297 - val_accuracy: 0.3716
Epoch 3/60
155/155 [==============================] - 36s 232ms/step - loss: 1.5835 - accuracy: 0.3545 - val_loss: 1.5773 - val_accuracy: 0.4215
Epoch 4/60
155/155 [==============================] - 36s 235ms/step - loss: 1.5454 - accuracy: 0.3729 - val_loss: 1.4526 - val_accuracy: 0.4176
Epoch 5/60
155/155 [==============================] - 36s 233ms/step - loss: 1.4659 - accuracy: 0.4082 - val_loss: 1.3758 - val_accuracy: 0.4368
Epoch 6/60
155/155 [==============================] - 36s 233ms/step - loss: 1.4218 - accuracy: 0.4285 - val_loss: 1.4555 - val_accuracy: 0.4291
Epoch 7/60
155/155 [==============================] - 36s 234ms/step - loss: 1.4099 - accuracy: 0.4268 - val_loss: 1.3669 - val_accuracy: 0.4866
Epoch 8/60
155/155 [==============================] - 36s 233ms/step - loss: 1.3684 - accuracy: 0.4446 - val_loss: 1.3463 - val_accuracy: 0.5057
Epoch 9/60
155/155 [==============================] - 36s 233ms/step - loss: 1.3373 - accuracy: 0.4620 - val_loss: 1.3496 - val_accuracy: 0.4904
Epoch 10/60
155/155 [==============================] - 36s 232ms/step - loss: 1.3484 - accuracy: 0.4640 - val_loss: 1.3791 - val_accuracy: 0.4674
Epoch 11/60
155/155 [==============================] - 36s 232ms/step - loss: 1.3454 - accuracy: 0.4685 - val_loss: 1.3198 - val_accuracy: 0.5057
Epoch 12/60
155/155 [==============================] - 36s 232ms/step - loss: 1.2952 - accuracy: 0.4834 - val_loss: 1.3768 - val_accuracy: 0.4981
Epoch 13/60
155/155 [==============================] - 36s 231ms/step - loss: 1.3187 - accuracy: 0.4717 - val_loss: 1.3559 - val_accuracy: 0.5096
Epoch 14/60
155/155 [==============================] - 36s 231ms/step - loss: 1.2688 - accuracy: 0.4947 - val_loss: 1.3639 - val_accuracy: 0.5019
Epoch 15/60
155/155 [==============================] - 36s 231ms/step - loss: 1.2768 - accuracy: 0.4964 - val_loss: 1.3729 - val_accuracy: 0.5134
Epoch 16/60
155/155 [==============================] - 36s 231ms/step - loss: 1.2847 - accuracy: 0.4990 - val_loss: 1.3572 - val_accuracy: 0.5019
```



```
155/155 [==============================] - 35s 227ms/step - loss: 0.8050 - accuracy: 0.7063 - val_loss: 2.5061 - val_accuracy: 0.5249
Epoch 46/60
155/155 [==============================] - 35s 227ms/step - loss: 0.7430 - accuracy: 0.7193 - val_loss: 2.6257 - val_accuracy: 0.5364
Epoch 47/60
155/155 [==============================] - 35s 227ms/step - loss: 0.7552 - accuracy: 0.7199 - val_loss: 3.1723 - val_accuracy: 0.4751
Epoch 48/60
155/155 [==============================] - 35s 227ms/step - loss: 0.7327 - accuracy: 0.7344 - val_loss: 2.7487 - val_accuracy: 0.4904
Epoch 49/60
155/155 [==============================] - 35s 227ms/step - loss: 0.7075 - accuracy: 0.7389 - val_loss: 3.5533 - val_accuracy: 0.4866
Epoch 50/60
155/155 [==============================] - 35s 227ms/step - loss: 0.8847 - accuracy: 0.6908 - val_loss: 2.9670 - val_accuracy: 0.5134
Epoch 51/60
155/155 [==============================] - 35s 227ms/step - loss: 0.7602 - accuracy: 0.7369 - val_loss: 2.9279 - val_accuracy: 0.5134
Epoch 52/60
155/155 [==============================] - 35s 227ms/step - loss: 0.7397 - accuracy: 0.7367 - val_loss: 2.8754 - val_accuracy: 0.5556
Epoch 53/60
155/155 [==============================] - 35s 227ms/step - loss: 0.6965 - accuracy: 0.7504 - val_loss: 3.0668 - val_accuracy: 0.4789
Epoch 54/60
155/155 [==============================] - 35s 227ms/step - loss: 0.6154 - accuracy: 0.7672 - val_loss: 3.3976 - val_accuracy: 0.5096
Epoch 55/60
155/155 [==============================] - 35s 228ms/step - loss: 0.5808 - accuracy: 0.7864 - val_loss: 3.0213 - val_accuracy: 0.5134
Epoch 56/60
155/155 [==============================] - 35s 227ms/step - loss: 0.5809 - accuracy: 0.7852 - val_loss: 3.9221 - val_accuracy: 0.4943
Epoch 57/60
155/155 [==============================] - 35s 227ms/step - loss: 0.5971 - accuracy: 0.7926 - val_loss: 4.3775 - val_accuracy: 0.4521
Epoch 58/60
155/155 [==============================] - 35s 227ms/step - loss: 0.5779 - accuracy: 0.7866 - val_loss: 3.3993 - val_accuracy: 0.5019
Epoch 59/60
155/155 [==============================] - 35s 227ms/step - loss: 0.5118 - accuracy: 0.8147 - val_loss: 4.2925 - val_accuracy: 0.5249
Epoch 60/60
155/155 [==============================] - 35s 228ms/step - loss: 0.4886 - accuracy: 0.8226 - val_loss: 4.3566 - val_accuracy: 0.5096
```


model accuracy
train
test
acc
epoch
0.8
0.7
0.6
0.5
0.4
0.3
0
10
20
30
40
50
60


Model Loss
train
test
Loss
Epoch
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
10
20
30
40
50
60



```
draw_confusion_matrix(confus_mat)
```

```
Accuracy of CNN model 2D = 0.5123152732849121
F1 Score of CNN model 2D = 0.5083886528456439
```

Confusion Matrix

True \ Predict	0	1	2	3	4	5
0	263	2	55	26	27	9
1	6	161	17	77	59	61
2	88	5	190	49	26	23
3	29	42	80	162	33	36
4	49	28	55	39	154	56
5	11	22	31	24	24	214

VIII. Comparison Between Different Models of 2D filters:

n	acc	f-score
(512, 512, 128)	0.512315	0.508388
(256, 128, 64)	0.5015674	0.4904014
(128, 64, 32)	0.4626063	0.4499058

Layers:

having more layers than required will risk the overfitting

n	acc	f-score
1	0.4415584	0.4272983
3	0.4626063	0.4499058
5	0.4572324	0.4343508
8	0.4482758	0.42482015

Filters= (128, 64, 32)

Epochs	Acc	f-score
30	0.4626063	0.4499058
60	0.4832064	0.4785069

IX. Comparison Between Both Features:

In audio : 2d is more efficient given something like a melspectrogram, in time series where you take previous inputs to predict the next ones, the sparsity of the data is relevant to the task.

If the data is using very little timesteps, convolutions across each variable (1d cnn) will work quite well, but with much larger batches and timesteps, 2d cnn has more theoretical feature interactions, and thus more "learnability"