Multimodal Sentiment Analysis

121AD0013

Madhav Sharma

Introduction

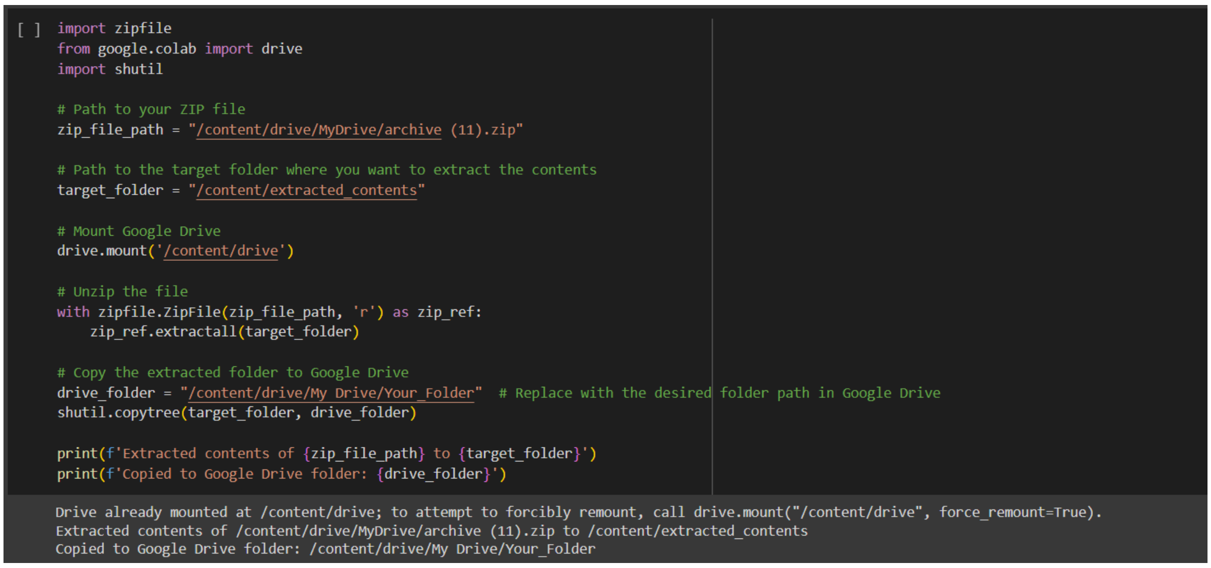
Sentiment analysis is a popular challenge in the Machine Learning Domain, several accurate models are made by Chinese and Korean Scientists, the main issue is everyone has separate faces when angry, happy, sad, and excited their facial features are different hence accuracy of the model is affected greatly

The dataset used must have similar or same person faces to enable the detection of emotions, what is actually in the backend is working of Hein's 7 features for facial detection which is a recent paper and is part of Computer Vision which we will not analyse today

The model gave us a satisfactory accuracy of 0.8 or 80% which is OK for small architecture and fewer resources

1. Un-Zipping the File:

The dataset is available at: https://www.kaggle.com/datasets/ameyamote030/einterface-image-dataset The dataset was uploaded in zip Format in Google Drive then unzipped from Python code as unzipping is less time taking in Python Code, We have used "zip file" library and mounted the drive and saved the data in the target folder

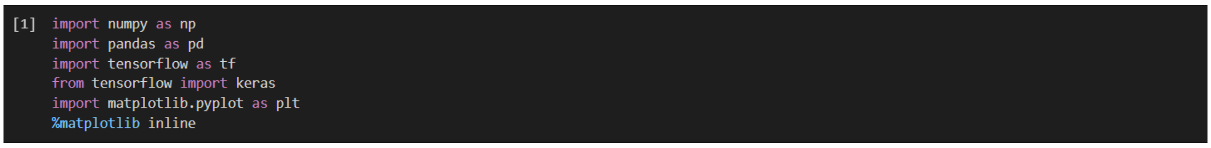


```
[ ] import zipfile
    from google.colab import drive
    import shutil

    # Path to your ZIP file
    zip_file_path = "/content/drive/MyDrive/archive (11).zip"

    # Path to the target folder where you want to extract the contents
    target_folder = "/content/extracted_contents"

    # Mount Google Drive
    drive.mount('/content/drive')

    # Unzip the file
    with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
        zip_ref.extractall(target_folder)

    # Copy the extracted folder to Google Drive
    drive_folder = "/content/drive/My Drive/Your_Folder"  # Replace with the desired folder path in Google Drive
    shutil.copytree(target_folder, drive_folder)

    print(f'Extracted contents of {zip_file_path} to {target_folder}')
    print(f'Copied to Google Drive folder: {drive_folder}')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
Extracted contents of /content/drive/MyDrive/archive (11).zip to /content/extracted_contents
Copied to Google Drive folder: /content/drive/My Drive/Your_Folder
```

2. Importing Necessary Libraries:

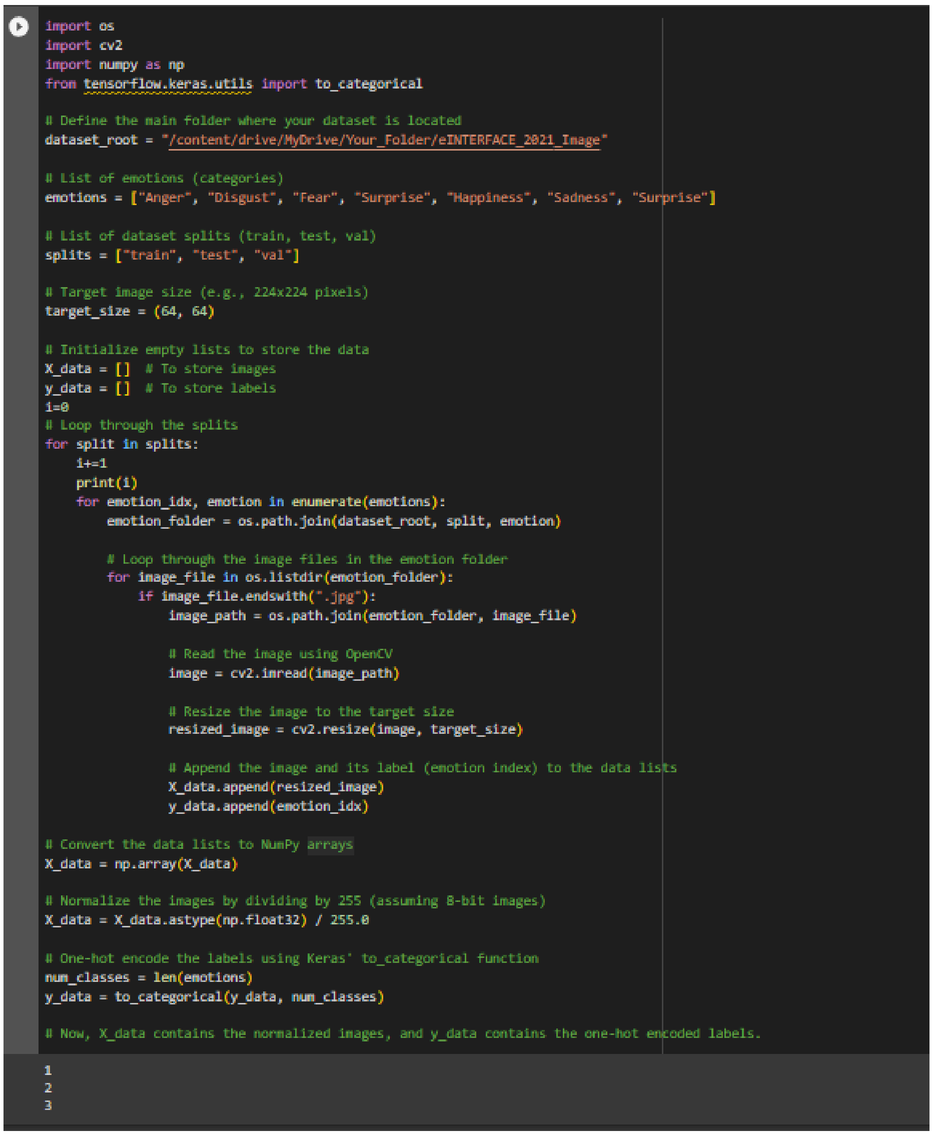
We have to import these to perform regular chores in Machine Learning Model



```
[1] import numpy as np
    import pandas as pd
    import tensorflow as tf
    from tensorflow import keras
    import matplotlib.pyplot as plt
    %matplotlib inline
```

3. Data Preprocessing:

- We have one directory which contains testing, training, and validation datasets the data is first stored in an image array, then converted into the same size of 64*64 due to hardware constraints, now what we have done is normalized the array to lie between [0,1] to do this we know that maximum pixel value will be 255.0 and minimum will be 0.0 hence we have directly divided by 255.0 Now final dataset is available for train test and val testing
- 80-20 ratio was maintained for training and testing: and then 50-50 between training and validation,
- The main purpose of testing is to test if the model is not overfit or underfit while the purpose of validation is to solely prevent overfitting the model

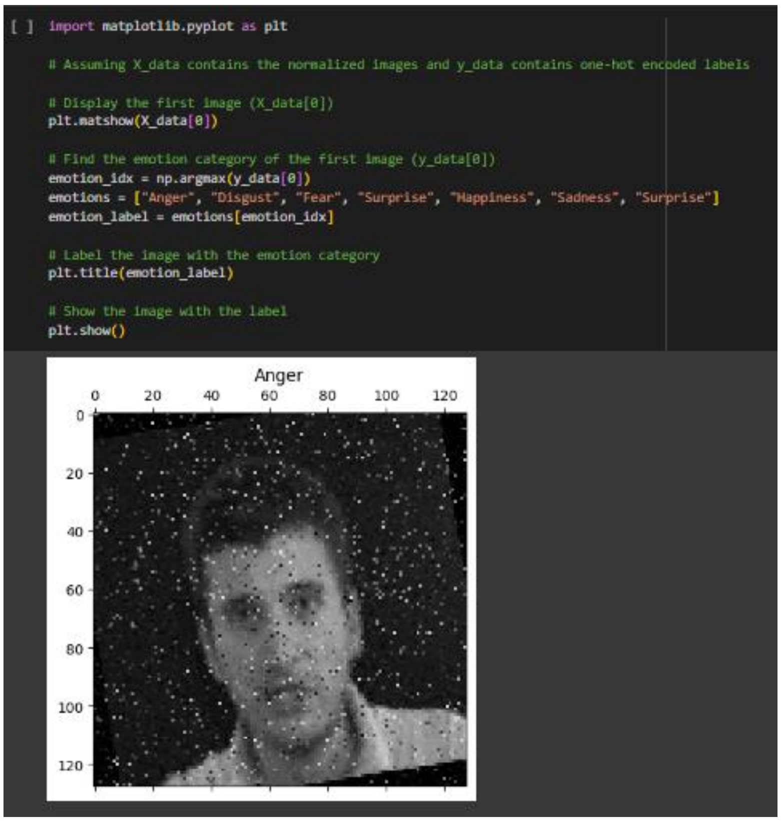


```
import os
import cv2
import numpy as np
from tensorflow.keras.utils import to_categorical

# Define the main folder where your dataset is located
dataset_root = "/content/drive/MyDrive/Your_Folder/eINTERFACE_2021_Image"

# List of emotions (categories)
emotions = ["Anger", "Disgust", "Fear", "Surprise", "Happiness", "Sadness", "Surprise"]

# List of dataset splits (train, test, val)
splits = ["train", "test", "val"]

# Target image size (e.g., 224x224 pixels)
target_size = (64, 64)

# Initialize empty lists to store the data
X_data = []  # To store images
y_data = []  # To store labels
i=0
# Loop through the splits
for split in splits:
    i+=1
    print(i)
    for emotion_idx, emotion in enumerate(emotions):
        emotion_folder = os.path.join(dataset_root, split, emotion)

        # Loop through the image files in the emotion folder
        for image_file in os.listdir(emotion_folder):
            if image_file.endswith(".jpg"):
                image_path = os.path.join(emotion_folder, image_file)

                # Read the image using OpenCV
                image = cv2.imread(image_path)

                # Resize the image to the target size
                resized_image = cv2.resize(image, target_size)

                # Append the image and its label (emotion index) to the data lists
                X_data.append(resized_image)
                y_data.append(emotion_idx)

# Convert the data lists to NumPy arrays
X_data = np.array(X_data)

# Normalize the images by dividing by 255 (assuming 8-bit images)
X_data = X_data.astype(np.float32) / 255.0

# One-hot encode the labels using Keras' to_categorical function
num_classes = len(emotions)
y_data = to_categorical(y_data, num_classes)

# Now, X_data contains the normalized images, and y_data contains the one-hot encoded labels.
```

```
1
2
3
```

4. Plotting Image for Mental Satisfaction:

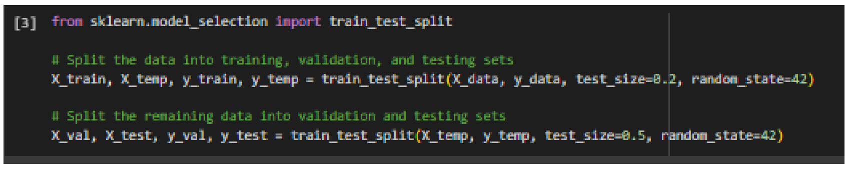
We have plotted the first image so that the label corresponds to that particular image so there is no mismatch between the label and image, we have used plt.matshow() command which is used to plot 2D-Numpy Arrays



```
[ ]  import matplotlib.pyplot as plt

     # Assuming X_data contains the normalized images and y_data contains one-hot encoded labels

     # Display the first image (X_data[0])
     plt.matshow(X_data[0])

     # Find the emotion category of the first image (y_data[0])
     emotion_idx = np.argmax(y_data[0])
     emotions = ["Anger", "Disgust", "Fear", "Surprise", "Happiness", "Sadness", "Surprise"]
     emotion_label = emotions[emotion_idx]

     # Label the image with the emotion category
     plt.title(emotion_label)

     # Show the image with the label
     plt.show()
```

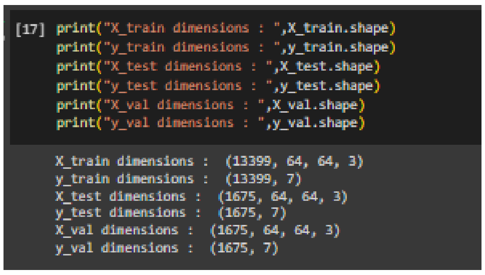
5. Train-Test Split

Here we have done a split between train and test , test and validation



```
[3]  from sklearn.model_selection import train_test_split

     # Split the data into training, validation, and testing sets
     X_train, X_temp, y_train, y_temp = train_test_split(X_data, y_data, test_size=0.2, random_state=42)

     # Split the remaining data into validation and testing sets
     X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42)
```

6. Metadata Exploration:



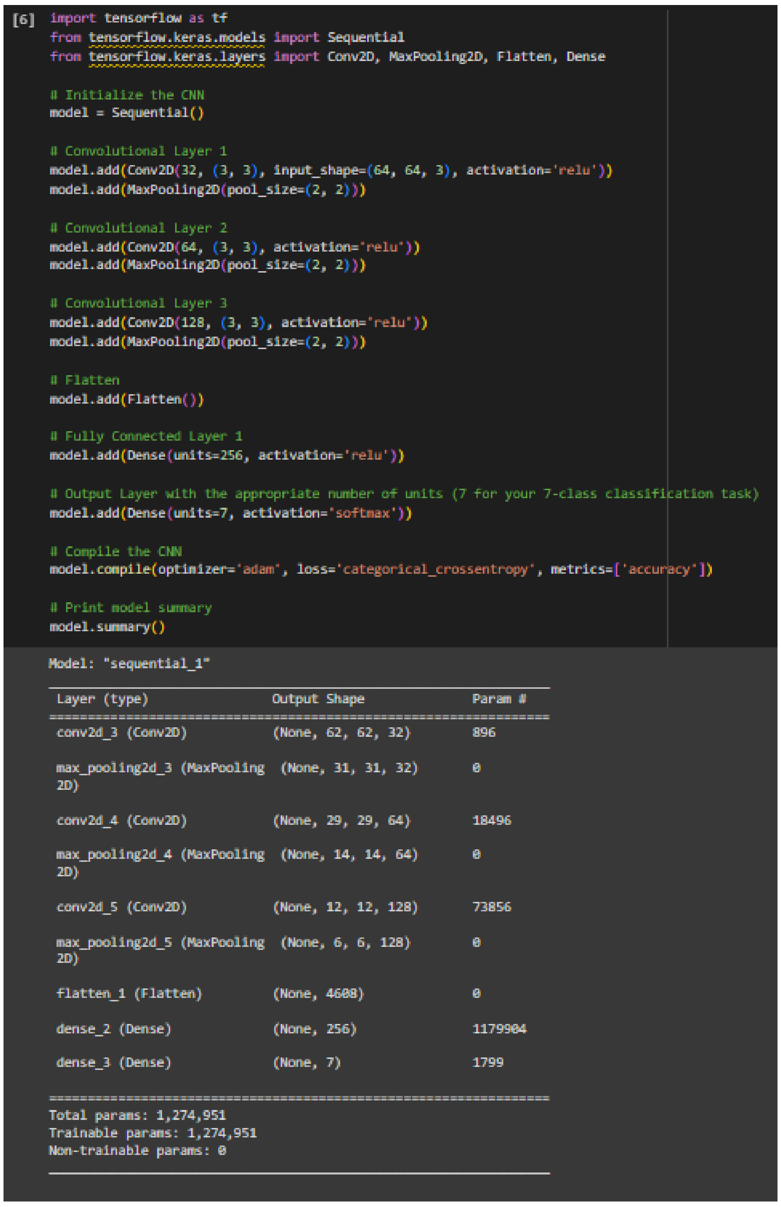
```
[17] print("X_train dimensions : ",X_train.shape)
     print("y_train dimensions : ",y_train.shape)
     print("X_test dimensions : ",X_test.shape)
     print("y_test dimensions : ",y_test.shape)
     print("X_val dimensions : ",X_val.shape)
     print("y_val dimensions : ",y_val.shape)

     X_train dimensions :  (13399, 64, 64, 3)
     y_train dimensions :  (13399, 7)
     X_test dimensions :  (1675, 64, 64, 3)
     y_test dimensions :  (1675, 7)
     X_val dimensions :  (1675, 64, 64, 3)
     y_val dimensions :  (1675, 7)
```

7. Defining Model Architecture:

- Initialize the CNN: The Sequential model is initialized. The Sequential model allows you to build a neural network layer by layer.
- Convolutional Layers: Three convolutional layers are added to the model. The first layer (Conv2D) has 32 filters of size 3x3, uses ReLU activation, and takes input images of size 64x64

pixels. A max-pooling layer (MaxPooling2D) with a 2x2 pool size follows each convolutional layer. The second convolutional layer has 64 filters, and the third has 128 filters, both using ReLU activation.

- Flatten Layer: The Flatten layer is used to flatten the 3D output to a 1D vector, preparing the data for the fully connected layers.
- Fully Connected Layers: One fully connected layer is added with 256 units and ReLU activation. The output layer has 7 units, corresponding to the 7 classes in your classification task. It uses the softmax activation function, which is suitable for multiclass classification.

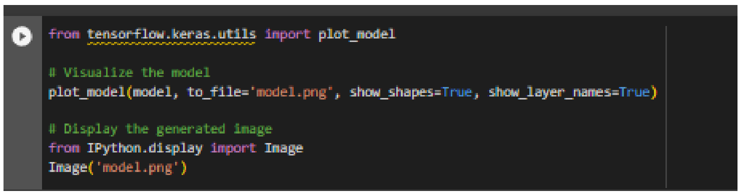


```
[6] import tensorflow as tf
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

    # Initialize the CNN
    model = Sequential()

    # Convolutional Layer 1
    model.add(Conv2D(32, (3, 3), input_shape=(64, 64, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Convolutional Layer 2
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Convolutional Layer 3
    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Flatten
    model.add(Flatten())

    # Fully Connected Layer 1
    model.add(Dense(units=256, activation='relu'))

    # Output Layer with the appropriate number of units (7 for your 7-class classification task)
    model.add(Dense(units=7, activation='softmax'))

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

    # Print model summary
    model.summary()
```

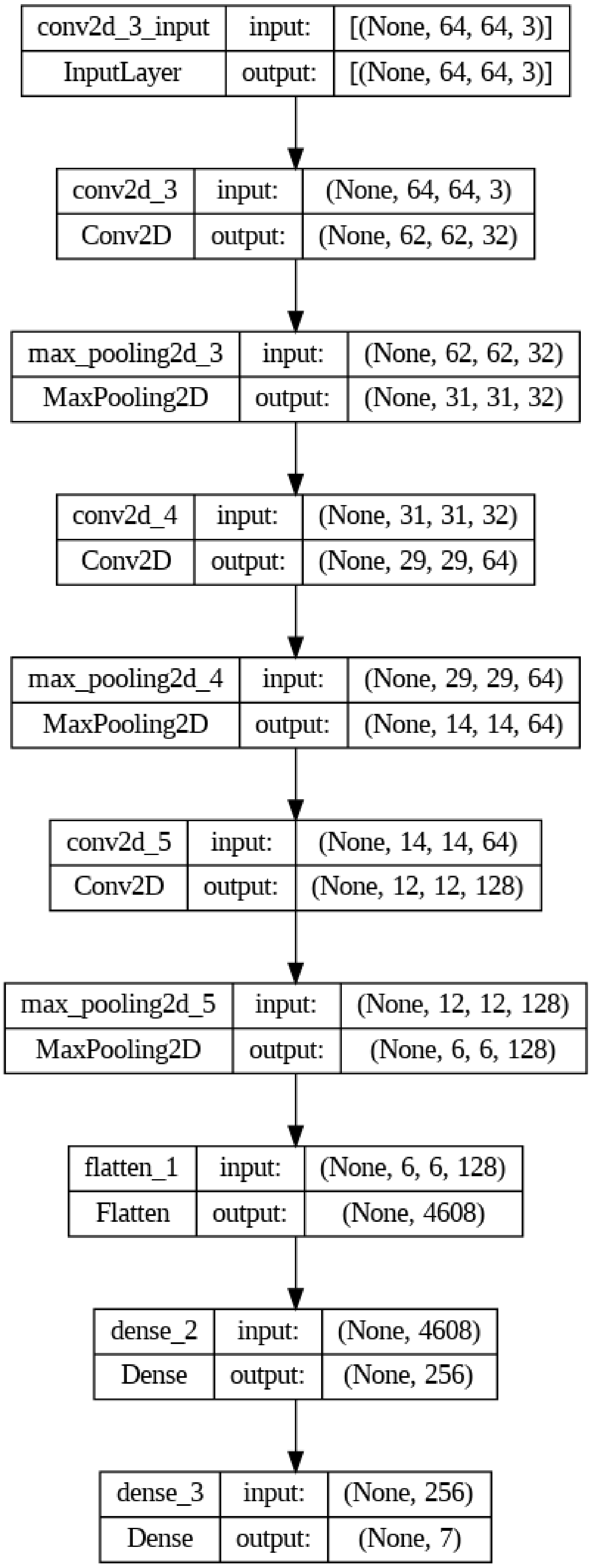
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 62, 62, 32)        896

 max_pooling2d_3 (MaxPooling  (None, 31, 31, 32)       0
 2D)

 conv2d_4 (Conv2D)           (None, 29, 29, 64)        18496

 max_pooling2d_4 (MaxPooling  (None, 14, 14, 64)       0
 2D)

 conv2d_5 (Conv2D)           (None, 12, 12, 128)       73856

 max_pooling2d_5 (MaxPooling  (None, 6, 6, 128)        0
 2D)

 flatten_1 (Flatten)         (None, 4608)              0

 dense_2 (Dense)             (None, 256)               1179904

 dense_3 (Dense)             (None, 7)                 1799

=================================================================
Total params: 1,274,951
Trainable params: 1,274,951
Non-trainable params: 0
_________________________________________________________________
```

8. Visualising CNN model

The visual memory of model is necessary toi understand its working we can use tensorflow plot_model function to plot the model

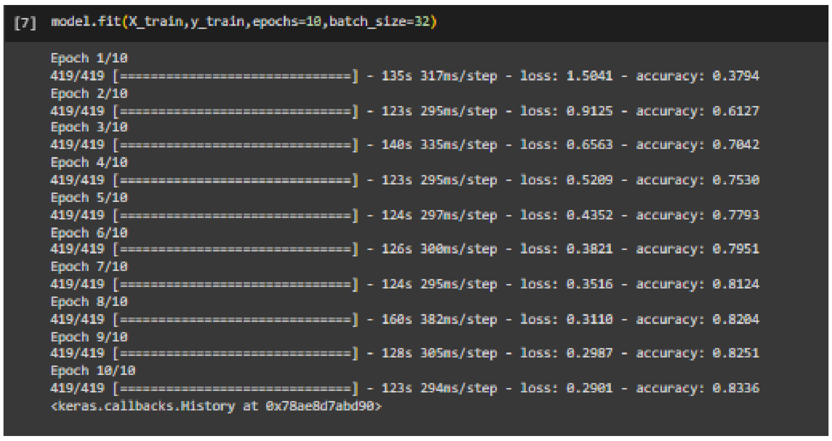


```
from tensorflow.keras.utils import plot_model

# Visualize the model
plot_model(model, to_file='model.png', show_shapes=True, show_layer_names=True)

# Display the generated image
from IPython.display import Image
Image('model.png')
```


conv2d_3_input | input: | [(None, 64, 64, 3)]
InputLayer | output: | [(None, 64, 64, 3)]
conv2d_3 | input: | (None, 64, 64, 3)
Conv2D | output: | (None, 62, 62, 32)
max_pooling2d_3 | input: | (None, 62, 62, 32)
MaxPooling2D | output: | (None, 31, 31, 32)
conv2d_4 | input: | (None, 31, 31, 32)
Conv2D | output: | (None, 29, 29, 64)
max_pooling2d_4 | input: | (None, 29, 29, 64)
MaxPooling2D | output: | (None, 14, 14, 64)
conv2d_5 | input: | (None, 14, 14, 64)
Conv2D | output: | (None, 12, 12, 128)
max_pooling2d_5 | input: | (None, 12, 12, 128)
MaxPooling2D | output: | (None, 6, 6, 128)
flatten_1 | input: | (None, 6, 6, 128)
Flatten | output: | (None, 4608)
dense_2 | input: | (None, 4608)
Dense | output: | (None, 256)
dense_3 | input: | (None, 256)
Dense | output: | (None, 7)

9. Fitting the model:

The number of epochs was set to 10 and batch size to 32 due to time constraints the model is pretty well trained



```
[7] model.fit(X_train,y_train,epochs=10,batch_size=32)

    Epoch 1/10
    419/419 [==============================] - 135s 317ms/step - loss: 1.5041 - accuracy: 0.3794
    Epoch 2/10
    419/419 [==============================] - 123s 295ms/step - loss: 0.9125 - accuracy: 0.6127
    Epoch 3/10
    419/419 [==============================] - 140s 335ms/step - loss: 0.6563 - accuracy: 0.7042
    Epoch 4/10
    419/419 [==============================] - 123s 295ms/step - loss: 0.5209 - accuracy: 0.7530
    Epoch 5/10
    419/419 [==============================] - 124s 297ms/step - loss: 0.4352 - accuracy: 0.7793
    Epoch 6/10
    419/419 [==============================] - 126s 300ms/step - loss: 0.3821 - accuracy: 0.7951
    Epoch 7/10
    419/419 [==============================] - 124s 295ms/step - loss: 0.3516 - accuracy: 0.8124
    Epoch 8/10
    419/419 [==============================] - 160s 382ms/step - loss: 0.3110 - accuracy: 0.8204
    Epoch 9/10
    419/419 [==============================] - 128s 305ms/step - loss: 0.2987 - accuracy: 0.8251
    Epoch 10/10
    419/419 [==============================] - 123s 294ms/step - loss: 0.2901 - accuracy: 0.8336
    <keras.callbacks.History at 0x78ae8d7abd90>
```

10.Testing the Model:

The most important phase is the testing phase tells us how overall model works, we will now see the metrics used:

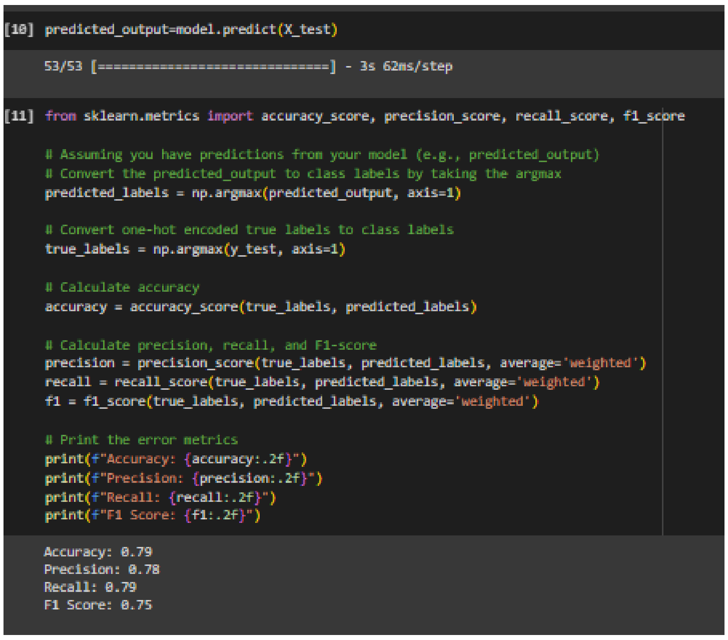
- **Accuracy:** Accuracy measures the overall correctness of a model's predictions. It's the ratio of correctly predicted instances to the total instances in the dataset. Formula: (True Positives + True Negatives) / (True Positives + True Negatives + False Positives + False Negatives) Accuracy is a good metric when the classes are balanced, meaning there are roughly equal numbers of positive and negative instances. However, it may not be informative when dealing with imbalanced datasets.
- **Precision:** Precision quantifies the accuracy of positive class predictions. It measures the ratio of true positive predictions to all positive predictions (both true positives and false positives). Formula: True Positives / (True Positives + False Positives) High precision is important when false positives are costly or when you want to minimize the chances of incorrectly classifying a negative instance as positive.
- **Recall (Sensitivity or True Positive Rate):** Recall assesses the model's ability to identify all positive instances correctly. It measures the ratio of true positive predictions to all actual positive instances (both true positives and false negatives). Formula: True Positives / (True Positives + False Negatives) High recall is crucial when missing positive instances (false negatives) is costly or when you want to ensure that positive instances are not overlooked.
- **F1 Score:** The F1 score is the harmonic mean of precision and recall. It provides a balanced measure that considers both false positives and false negatives. Formula: 2 * (Precision * Recall) / (Precision + Recall) The F1 score is a good choice when you want to strike a balance between precision and recall. It's especially useful in situations where class imbalance exists.

About TP, TN, FP, FN:

- **True Positives (TP)**: Instances that were correctly predicted as positive by the model. These are the actual positive cases that the model identified correctly.

- **True Negatives (TN)**: Instances that were correctly predicted as negative by the model. These are the actual negative cases that the model identified correctly.

- **False Positives (FP)**: Instances that were incorrectly predicted as positive by the model when they were actually negative. These are also known as Type I errors.
- **False Negatives (FN)**: Instances that were incorrectly predicted as negative by the model when they were actually positive. These are also known as Type II errors.



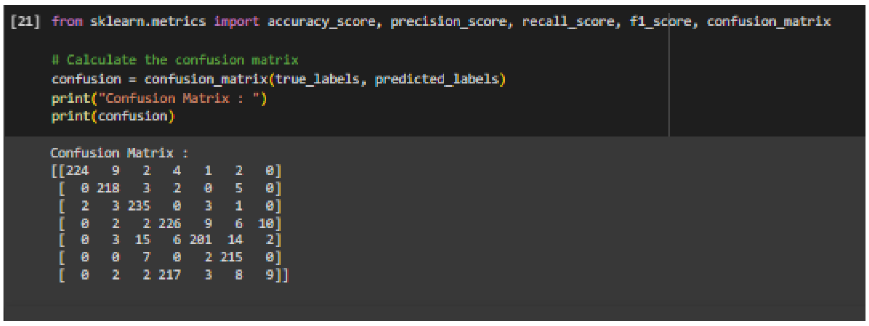
```
[10] predicted_output=model.predict(X_test)

     53/53 [==============================] - 3s 62ms/step

[11] from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

     # Assuming you have predictions from your model (e.g., predicted_output)
     # Convert the predicted_output to class labels by taking the argmax
     predicted_labels = np.argmax(predicted_output, axis=1)

     # Convert one-hot encoded true labels to class labels
     true_labels = np.argmax(y_test, axis=1)

     # Calculate accuracy
     accuracy = accuracy_score(true_labels, predicted_labels)

     # Calculate precision, recall, and F1-score
     precision = precision_score(true_labels, predicted_labels, average='weighted')
     recall = recall_score(true_labels, predicted_labels, average='weighted')
     f1 = f1_score(true_labels, predicted_labels, average='weighted')

     # Print the error metrics
     print(f"Accuracy: {accuracy:.2f}")
     print(f"Precision: {precision:.2f}")
     print(f"Recall: {recall:.2f}")
     print(f"F1 Score: {f1:.2f}")

     Accuracy: 0.79
     Precision: 0.78
     Recall: 0.79
     F1 Score: 0.75
```

11.Confusion Matrix:

A confusion matrix is a table that is used to describe the performance of a classification model on a set of test data for which the true values are known. It provides a detailed breakdown of the model's predictions, making it a valuable tool for evaluating the model's performance, especially in classification tasks.

The confusion matrix is typically organized into four main categories:

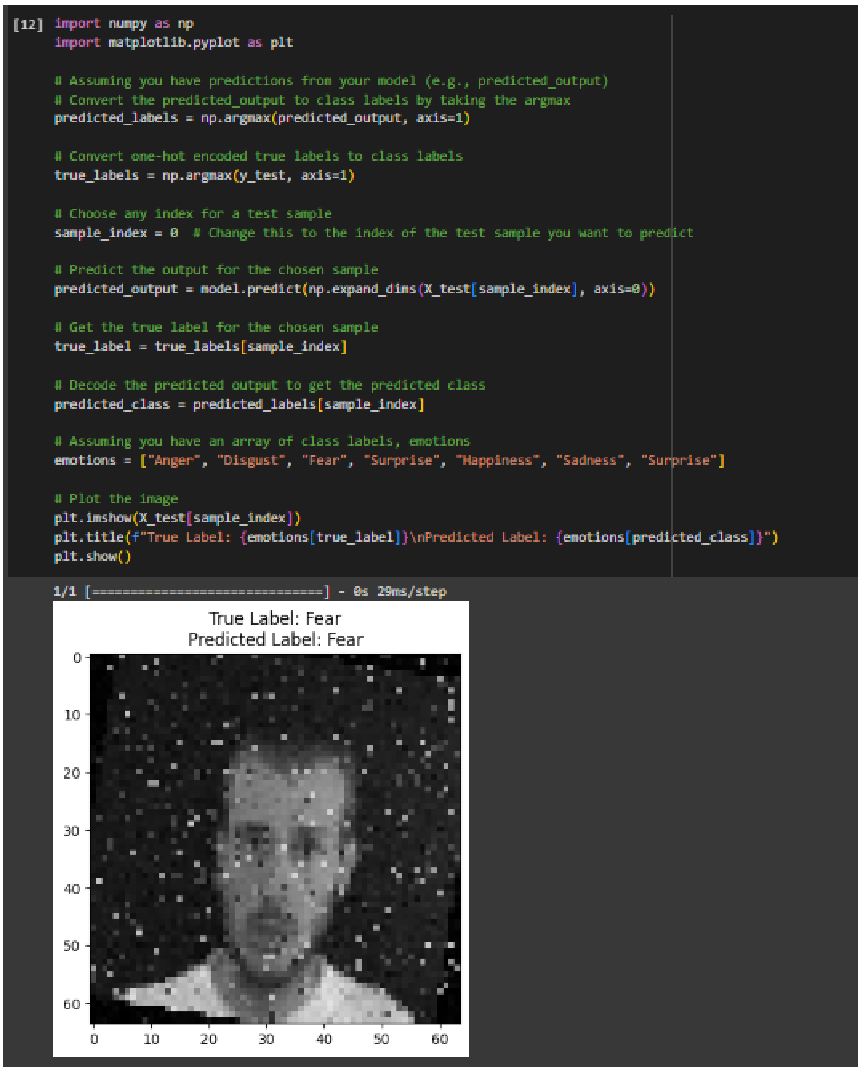
- True Positives (TP): The model correctly predicted the positive class.
- True Negatives (TN): The model correctly predicted the negative class.
- False Positives (FP): The model incorrectly predicted the positive class (a type I error).
- False Negatives (FN): The model incorrectly predicted the negative class (a type II error).



```
[21] from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix

     # Calculate the confusion matrix
     confusion = confusion_matrix(true_labels, predicted_labels)
     print("Confusion Matrix : ")
     print(confusion)

     Confusion Matrix :
     [[224   9   2   4   1   2   0]
      [  0 218   3   2   0   5   0]
      [  2   3 235   0   3   1   0]
      [  0   2   2 226   9   6  10]
      [  0   3  15   6 201  14   2]
      [  0   0   7   0   2 215   0]
      [  0   2   2 217   3   8   9]]
```

12. Prediction Phase:

Prediction using a input is very important as it tells us is the model suitable for real world problems , the model predicted well, the model also performed good on noisy dataset(almost 80% accuracy)



```
[12] import numpy as np
     import matplotlib.pyplot as plt

     # Assuming you have predictions from your model (e.g., predicted_output)
     # Convert the predicted_output to class labels by taking the argmax
     predicted_labels = np.argmax(predicted_output, axis=1)

     # Convert one-hot encoded true labels to class labels
     true_labels = np.argmax(y_test, axis=1)

     # Choose any index for a test sample
     sample_index = 0  # Change this to the index of the test sample you want to predict

     # Predict the output for the chosen sample
     predicted_output = model.predict(np.expand_dims(X_test[sample_index], axis=0))

     # Get the true label for the chosen sample
     true_label = true_labels[sample_index]

     # Decode the predicted output to get the predicted class
     predicted_class = predicted_labels[sample_index]

     # Assuming you have an array of class labels, emotions
     emotions = ["Anger", "Disgust", "Fear", "Surprise", "Happiness", "Sadness", "Surprise"]

     # Plot the image
     plt.imshow(X_test[sample_index])
     plt.title(f"True Label: {emotions[true_label]}\nPredicted Label: {emotions[predicted_class]}")
     plt.show()

     1/1 [==============================] - 0s 29ms/step
```

True Label: Fear
Predicted Label: Fear

Summary:

1) The model had 9 layers in total constituting of Covloution layer, Max Pooling Layer, Flattening Layer, and Input and Output layers
2) The model showed good results for images and was seen robust to outliers
3) The following confusion matrix was observed:

224	9	2	4	1	2	0
0	218	3	2	0	5	0
2	3	235	0	3	1	0
0	2	2	226	9	6	10
0	3	15	6	201	14	2
0	0	7	0	2	215	0
0	2	2	217	3	8	9

The model has following accuracy metrics:

Accuracy	0.79
Precision	0.78
Recall	0.79

F1 Score	0.75