```
pip install tensorflow==2.9.1
```

SOURCE CODE:

Importing libraries:

```python
# import system libs
import os
import time
import shutil
import pathlib
import itertools
from PIL import Image

# import data handling tools
import cv2
import numpy as np
import pandas as pd
import seaborn as sns
sns.set_style('darkgrid')
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
# import Deep learning Libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam, Adamax
from tensorflow.keras.metrics import categorical_crossentropy
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Activation, Dropout, BatchNormalization
from tensorflow.keras import regularizers
```

```python
# Ignore Warnings
import warnings
warnings.filterwarnings("ignore")

print ('modules loaded')
```

Setting Training path:

```python
train_dir = 'train'
filepaths = []
labels = []

folds = os.listdir(train_dir)
for fold in folds:
    foldpath = os.path.join(train_dir, fold)
    filelist = os.listdir(foldpath)
    for file in filelist:
        fpath = os.path.join(foldpath, file)
        filepaths.append(fpath)
        labels.append(fold)

# Concatenate data paths with labels into one dataframe
Fseries = pd.Series(filepaths, name= 'filepaths')
Lseries = pd.Series(labels, name='labels')
train_df = pd.concat([Fseries, Lseries], axis= 1)
```

```python
train_df
```

	filepaths	labels
0	train\angry\im0.png	angry
1	train\angry\im1.png	angry
2	train\angry\im10.png	angry
3	train\angry\im100.png	angry
4	train\angry\im1000.png	angry
...	...	...
28704	train\surprised\im995.png	surprised
28705	train\surprised\im996.png	surprised
28706	train\surprised\im997.png	surprised
28707	train\surprised\im998.png	surprised
28708	train\surprised\im999.png	surprised

28709 rows × 2 columns

Generate test data paths with labels:

```
# Generate  test data paths with labels
test_dir = 'test'
filepaths = []
labels = []

folds = os.listdir(test_dir)
for fold in folds:
    foldpath = os.path.join(test_dir, fold)
    filelist = os.listdir(foldpath)
    for file in filelist:
        fpath = os.path.join(foldpath, file)
        filepaths.append(fpath)
        labels.append(fold)

# Concatenate data paths with labels into one dataframe
Fseries = pd.Series(filepaths, name= 'filepaths')
Lseries = pd.Series(labels, name='labels')
test_df = pd.concat([Fseries, Lseries], axis= 1)
```
[7]

```
test_df
```
[8]

	filepaths	labels
0	test\angry\im0.png	angry
1	test\angry\im1.png	angry
2	test\angry\im10.png	angry
3	test\angry\im100.png	angry
4	test\angry\im101.png	angry
...	...	...
7173	test\surprised\im95.png	surprised
7174	test\surprised\im96.png	surprised
7175	test\surprised\im97.png	surprised
7176	test\surprised\im98.png	surprised
7177	test\surprised\im99.png	surprised

7178 rows × 2 columns

Valid and test data frame:

```
# valid and test dataframe
valid_df, test_df = train_test_split(test_df,  train_size= 0.6, shuffle= True, random_state= 123)
```
[9]

```python
# crobed image size
batch_size = 16
img_size = (224, 224)
channels = 3
img_shape = (img_size[0], img_size[1], channels)

tr_gen = ImageDataGenerator()
ts_gen = ImageDataGenerator()
train_gen = tr_gen.flow_from_dataframe( train_df, x_col= 'filepaths', y_col= 'labels', target_size= img_size, class_mode= 'categorical',
                                        color_mode= 'rgb', shuffle= True, batch_size= batch_size)

valid_gen = ts_gen.flow_from_dataframe( valid_df, x_col= 'filepaths', y_col= 'labels', target_size= img_size, class_mode= 'categorical',
                                        color_mode= 'rgb', shuffle= True, batch_size= batch_size)

test_gen = ts_gen.flow_from_dataframe( test_df, x_col= 'filepaths', y_col= 'labels', target_size= img_size, class_mode= 'categorical',
                                       color_mode= 'rgb', shuffle= False, batch_size= batch_size)
```

```
Found 28709 validated image filenames belonging to 7 classes.
Found 4306 validated image filenames belonging to 7 classes.
Found 2872 validated image filenames belonging to 7 classes.
```

Visualizing a Batch of Training Images with Class Labels:

```python
g_dict = train_gen.class_indices      # defines dictionary {'class': index}
classes = list(g_dict.keys())       # defines list of dictionary's kays (classes), classes names : string
images, labels = next(train_gen)      # get a batch size samples from the generator

plt.figure(figsize= (20, 20))

for i in range(16):
    plt.subplot(4, 4, i + 1)
    image = images[i] / 255       # scales data to range (0 - 255)
    plt.imshow(image)
    index = np.argmax(labels[i])  # get image index
    class_name = classes[index]   # get class of image
    plt.title(class_name, color= 'blue', fontsize= 12)
    plt.axis('off')
plt.show()
```

Defining the Model Architecture with EfficientNetB0 Backbone:

```
#Create Model Structure
img_size = (224, 224)
channels = 3
img_shape = (img_size[0], img_size[1], channels)
class_count = len(list(train_gen.class_indices.keys())) # to define number of classes in dense layer

# create pre-trained model (you can built on pretrained model such as :  efficientnet, VGG , Resnet )
# we will use efficientnetb3 from EfficientNet family.
base_model = tf.keras.applications.efficientnet.EfficientNetB0(include_top= False, weights= "imagenet", input_shape= img_shape, pooling= 'max')
# base_model.trainable = False
model = Sequential([
    base_model,
    BatchNormalization(axis= -1, momentum= 0.99, epsilon= 0.001),
    Dense(256, kernel_regularizer= regularizers.l2( 0.016), activity_regularizer= regularizers.l1(0.006),
                bias_regularizer= regularizers.l1(0.006), activation= 'relu'),
    Dropout(rate= 0.45, seed= 123),
    Dense(class_count, activation= 'softmax')
])

model.compile(Adamax(learning_rate= 0.001), loss= 'categorical_crossentropy', metrics= ['accuracy'])

model.summary()
```

Layer (type)	Output Shape	Param #
efficientnetb0 (Functional)	(None, 1280)	4,049,571
batch_normalization (BatchNormalization)	(None, 1280)	5,120
dense (Dense)	(None, 256)	327,936
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 7)	1,799

```
Total params: 4,384,426 (16.73 MB)

Trainable params: 4,339,843 (16.56 MB)

Non-trainable params: 44,583 (174.16 KB)
```

Training the Model with Generate test data paths with labels:

```
batch_size = 20   # set batch size for training
epochs =  1  # number of all epochs in training

history = model.fit( (function) validation_steps: Any )se= 1, validation_data= valid_gen,
                    validation_steps= None, shuffle= False)
```

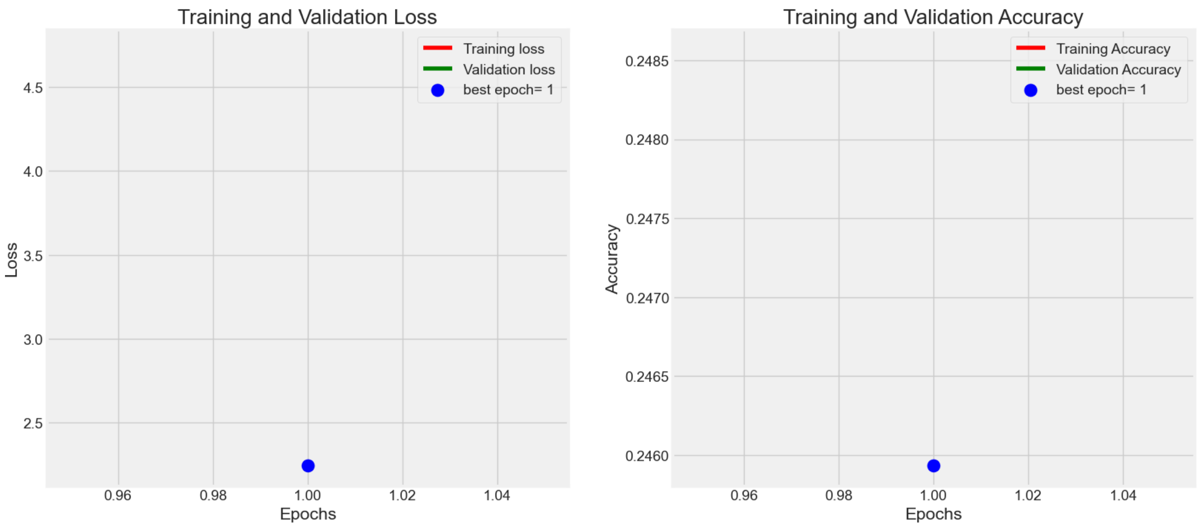
```
1795/1795 ━━━━━━━━━━━━━━━━━━━━ 5828s 3s/step - accuracy: 0.2415 - loss: 7.4846 - val_accuracy: 0.2459 - val_loss: 2.2470
```

Evaluating Training and Validation Performance Over Epochs:

```
# Define needed variables
tr_acc = history.history['accuracy']
tr_loss = history.history['loss']
val_acc = history.history['val_accuracy']
val_loss = history.history['val_loss']
index_loss = np.argmin(val_loss)
val_lowest = val_loss[index_loss]
index_acc = np.argmax(val_acc)
acc_highest = val_acc[index_acc]

Epochs = [i+1 for i in range(len(tr_acc))]
loss_label = f'best epoch= {str(index_loss + 1)}'
acc_label = f'best epoch= {str(index_acc + 1)}'
```

```python
# Plot training history
plt.figure(figsize= (20, 8))
plt.style.use('fivethirtyeight')
plt.subplot(1, 2, 1)
plt.plot(Epochs, tr_loss, 'r', label= 'Training loss')
plt.plot(Epochs, val_loss, 'g', label= 'Validation loss')
plt.scatter(index_loss + 1, val_lowest, s= 150, c= 'blue', label= loss_label)
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(Epochs, tr_acc, 'r', label= 'Training Accuracy')
plt.plot(Epochs, val_acc, 'g', label= 'Validation Accuracy')
plt.scatter(index_acc + 1 , acc_highest, s= 150, c= 'blue', label= acc_label)
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.tight_layout
plt.show()
```



Evaluating Model Performance on Training, Validation, and Test Sets:

```python
ts_length = len(test_df)
test_batch_size = max(sorted([ts_length // n for n in range(1, ts_length + 1) if ts_length%n == 0 and ts_length/n <= 80]))
test_steps = ts_length // test_batch_size

train_score = model.evaluate(train_gen, steps= test_steps, verbose= 1)
valid_score = model.evaluate(valid_gen, steps= test_steps, verbose= 1)
test_score = model.evaluate(test_gen, steps= test_steps, verbose= 1)

print("Train Loss: ", train_score[0])
print("Train Accuracy: ", train_score[1])
print('-' * 20)
print("Validation Loss: ", valid_score[0])
print("Validation Accuracy: ", valid_score[1])
print('-' * 20)
print("Test Loss: ", test_score[0])
print("Test Accuracy: ", test_score[1])
```

```
359/359 ━━━━━━━━━━━━━━━━━━━━ 174s 486ms/step - accuracy: 0.2554 - loss: 2.2483
359/359 ━━━━━━━━━━━━━━━━━━━━ 130s 361ms/step - accuracy: 0.2507 - loss: 2.2459
359/359 ━━━━━━━━━━━━━━━━━━━━ 95s 263ms/step - accuracy: 0.2479 - loss: 2.2508
Train Loss:  2.2467174530029297
Train Accuracy:  0.259575217962265
--------------------
Validation Loss:  2.2470076084136963
Validation Accuracy:  0.24593590199947357
--------------------
Test Loss:  2.2502918243408203
Test Accuracy:  0.24721448123455048
```

Generating Predictions on Test Data:

```
Click to add a breakpoint ct(test_gen, steps=len(test_gen))
y_pred = np.argmax(preds, axis=1)
[16]
... 180/180 ━━━━━━━━━━━━━━━━━━━━ 96s 524ms/step
```

Visualizing Model Performance with a Confusion Matrix:

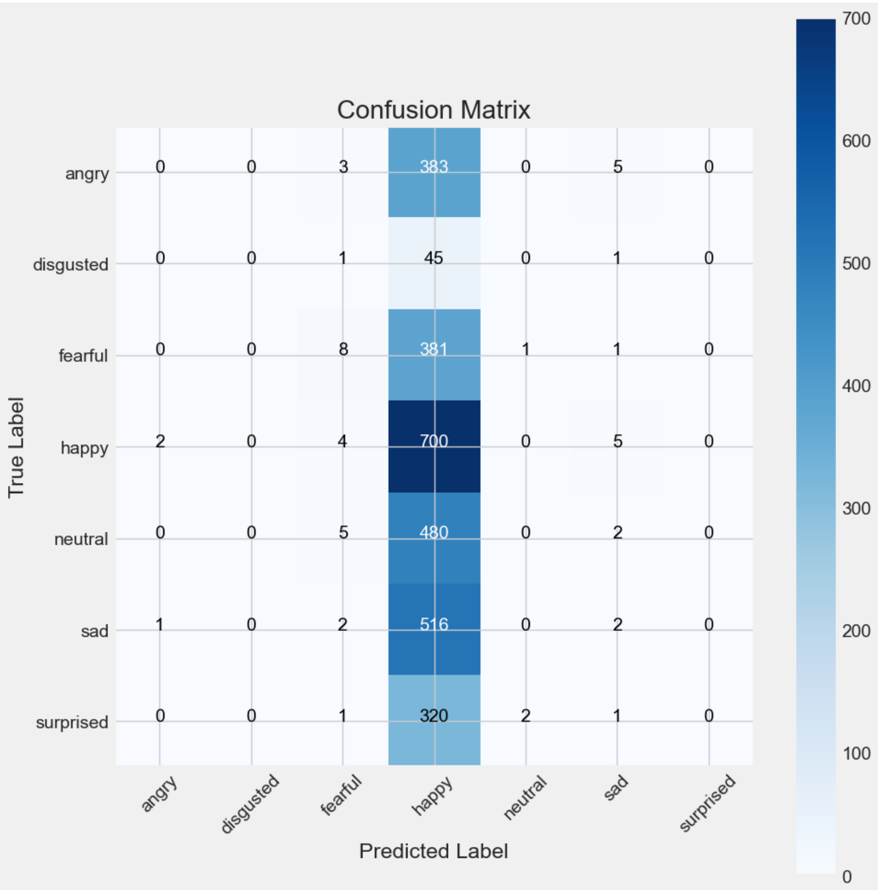
```python
g_dict = test_gen.class_indices
classes = list(g_dict.keys())

# Confusion matrix
cm = confusion_matrix(test_gen.classes, y_pred)

plt.figure(figsize= (10, 10))
plt.imshow(cm, interpolation= 'nearest', cmap= plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()
tick_marks = np.arange(len(classes))
plt.xticks(tick_marks, classes, rotation= 45)
plt.yticks(tick_marks, classes)
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, cm[i, j], horizontalalignment= 'center', color= 'white' if cm[i, j] > thresh else 'black')

plt.tight_layout()
plt.ylabel('True Label')
plt.xlabel('Predicted Label')

plt.show()
```



Generating Classification Report for Test Data:

```
# Classification report
print(classification_report(test_gen.classes, y_pred, target_names= classes))
```

```
              precision    recall  f1-score   support

       angry       0.00      0.00      0.00       391
   disgusted       0.00      0.00      0.00        47
     fearful       0.33      0.02      0.04       391
       happy       0.25      0.98      0.40       711
     neutral       0.00      0.00      0.00       487
         sad       0.12      0.00      0.01       521
   surprised       0.00      0.00      0.00       324

    accuracy                           0.25      2872
   macro avg       0.10      0.14      0.06      2872
weighted avg       0.13      0.25      0.10      2872
```

Saving the Trained Model for Future Use:

```
#Save the model
model.save('model.h5')
```

Loading and Compiling the Saved Model:

```
loaded_model = tf.keras.models.load_model('model.h5', compile=False)
loaded_model.compile(Adamax(learning_rate= 0.001), loss= 'categorical_crossentropy', metrics= ['accuracy'])
```

Making Predictions on a Single Image:

```
image_path = 'test/neutral/im90.png'
image = Image.open(image_path)

# Preprocess the image
img = image.resize((224, 224))
img_array = tf.keras.preprocessing.image.img_to_array(img)
img_array = tf.expand_dims(img_array, 0)

# Make predictions
predictions = loaded_model.predict(img_array)
class_labels = classes
score = tf.nn.softmax(predictions[0])
print(f"{class_labels[tf.argmax(score)]}")
plt.imshow(img)
plt.axis('off')
plt.show()
```

Results Interpretation

Detected Emotion:

- Once the model processes the input image, it outputs the predicted emotion, indicating the emotional state represented in the image.

Model Prediction Accuracy:

Users can evaluate the model's performance by considering how well the predicted emotion aligns with the visible expressions in the image. For improved accuracy, using a diverse dataset with a wide range of emotional expressions and higher-quality images is recommended.

RESULT :

CONCLUSION:

In conclusion, utilizing Convolutional Neural Networks (CNNs) for emotion detection has proven to be an effective approach, surpassing traditional methods in accuracy and reliability. CNNs' capacity to learn complex features from images enables them to identify emotional expressions under various conditions, such as different lighting and angles. The success of CNNs in this domain highlights their adaptability, making them suitable for applications ranging from user experience enhancement in technology to mental health assessments.