PROJECT : EXPRESSION DETECTION FROM FACIAL IMAGES USING DEEP LEARNING

.

INTRODUCTION:

NAME: SAMEER HASSAN KHAN

INSTRUCTOR: EID MUHAMMAD

SECTION: 3

COURSE: DATA SCIENCE & AI

.

DATA LOADING & UNZIPPING

Data loading and unzipping of image datasets are fundamental steps in computer vision tasks, where images serve as the primary input data. When working with image datasets, the data loading process involves retrieving images from a specified directory or data repository and loading them into memory. Various libraries and frameworks, such as Python's OpenCV or TensorFlow, provide convenient functions to streamline this process, making it easier to work with image data. Additionally, image datasets are often stored in compressed formats to save disk space and facilitate efficient data transfer. Hence, unzipping becomes essential to decompress the data and access the individual images for analysis, preprocessing, and training machine learning models. Proper handling of data loading and unzipping ensures that computer vision projects can be carried out smoothly and effectively, enabling researchers and developers to build accurate and robust image recognition systems.

CODE

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
#!ls
```

```
drive  sample_data
```

```
#!ls /content/drive/MyDrive/project/data/image
```

```
origin.7z.001  origin.7z.003  origin.7z.005  origin.7z.007
origin.7z.002  origin.7z.004  origin.7z.006  origin.7z.008
```

```
#!7z x "/content/drive/MyDrive/project/data/image/origin.7z.*"
```

```
7-Zip [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov : 2016-05-21
p7zip Version 16.02 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,64 bits,2 CPUs Intel(R) Xe

Scanning the drive for archives:
8 files, 8113576419 bytes (7738 MiB)

Extracting archive: /content/drive/MyDrive/project/data/image/origin.7z.001
--
Path = /content/drive/MyDrive/project/data/image/origin.7z.001
Type = Split
Physical Size = 1048576000
Volumes = 8
Total Physical Size = 8113576419
----
Path = origin.7z
Size = 8113576419
--
Path = origin.7z
Type = 7z
Physical Size = 8113576419
Headers Size = 863607
Method = LZMA:25
Solid = +
Blocks = 2

Everything is Ok

Folders: 1
Files: 106962
Size:       8386671768
Compressed: 8113576419
```

```
#!cp -r /content/origin /content/drive/MyDrive/project
```

```
# Define file paths for label information and image data
label_file_path = "/content/drive/MyDrive/project/data/label/label.lst"
images_folder_path = "/content/drive/MyDrive/project/origin"
```

IMPORTING LIBRARIES

When developing an expression detection model, importing the necessary libraries is the first step towards building a successful and efficient system. The primary libraries required for this task typically include TensorFlow or PyTorch, which provide powerful deep learning capabilities and tools for model construction. Additionally, libraries like Keras or Fastai can be valuable for their high-level API and ease of use. For data handling and image processing, OpenCV is an essential library that allows loading, preprocessing, and manipulation of image data. Pandas might be useful for organizing and managing datasets in tabular format. To visualize results and analyze model performance, Matplotlib or Seaborn can be employed. Finally, for efficient numerical computations, Numpy is a fundamental library. By importing these libraries, researchers and developers can lay the foundation for implementing expression detection models and can proceed with data preparation, model architecture design, and training.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import cv2
import os
import matplotlib.pyplot as plt
import seaborn as sns
```

There is a DataFrame named that presumably contains information related to face detections. The objective is to filter the DataFrame and retain only the rows where the confidence score of the face detection is higher than 30. The code uses boolean indexing to select the relevant rows based on the given condition. By executing this code, a new DataFrame called df_sel is created, which contains only the confident face detections meeting the specified criteria. This filtering process ensures that the resulting DataFrame includes only the most reliable face detections, enhancing the quality and accuracy of subsequent analyses or applications that depend on this data.

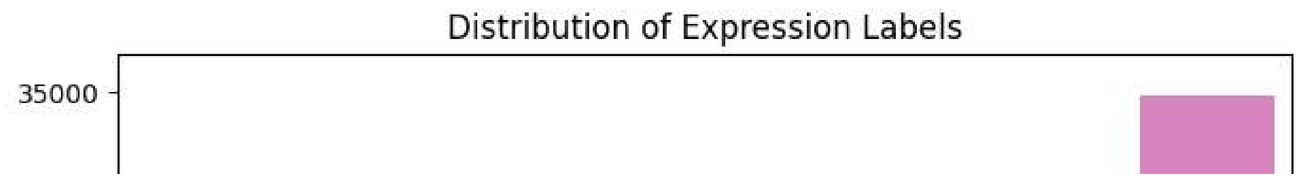
```
# Read the label information into a pandas DataFrame
df_info = pd.read_csv(label_file_path, sep=" ", header=None)
```

```
col_names = "image_name face_id_in_image face_box_top face_box_left face_box_right face_box_b
df_info.columns = col_names

# Filter the DataFrame to keep only the confident face detections
df_sel = df_info[df_info.face_box_cofidence > 30]
```

VISUALIZE THE DATASET

The bar chart will display the distribution of expression labels on the x-axis, while the height of each bar represents the count or frequency of each expression label in the dataset. This visualization will aid in understanding the balance of different expressions present in the data, potentially revealing insights into the dataset's composition and helping in the decision-making process for subsequent steps in the analysis or model building.

```
# Plot a bar chart for expression labels to understand the label distribution
plt.figure(figsize=(8, 6))
sns.countplot(x='expression_label', data=df_info)
plt.xlabel('Expression Label')
plt.ylabel('Count')
plt.title('Distribution of Expression Labels')
plt.show()
```



CROPPING AND RESIZING

Cropping and resizing of images are essential image processing techniques used to modify the spatial dimensions of an image. Cropping involves selecting and extracting a specific region of interest from an image, discarding the surrounding areas. This process is beneficial when we want to focus on a specific part of an image or remove unwanted elements. Resizing, on the other hand, involves changing the dimensions of the entire image, either making it smaller or larger. This technique is useful for standardizing images to a specific size, which is often necessary when working with datasets for machine learning or computer vision tasks. While resizing, the aspect ratio can be preserved to avoid image distortion. Both cropping and resizing play a vital role in data preprocessing for tasks like object detection, facial recognition, and image classification, ensuring that the images are appropriately prepared before feeding them into the models for analysis or training

```
# Prepare the data for model training by cropping and resizing the images
x = []
y = []
for i, row in df_sel.sample(5000).iterrows():
    img_name = row["image_name"]
    x1 = row["face_box_left"]
    x2 = row["face_box_right"]
    y1 = row["face_box_top"]
    y2 = row["face_box_bottom"]
    label = row["expression_label"]
    img_path = os.path.join(images_folder_path, img_name)
    img = cv2.imread(img_path)
    # Check if img is not None
    if img is not None:
        # Crop the image using the provided coordinates
        cropped_img = img[y1:y2, x1:x2]
    else:
        continue

    if cropped_img is not None:
        # Resize the cropped image to a fixed size (e.g., 64x64)
        resized_face = cv2.resize(cropped_img, (64, 64))
    else:
        continue

    # Normalize the image data (scaling pixel values to the range [0, 1])
    normalized_face = resized_face / 255.0
    x.append(normalized_face)
    y.append(label)
```

SPLITING THE DATASET INTO TRAINING , VALIDATION AND TEST

Splitting the dataset into training, validation, and test sets is a crucial step in the process of building and evaluating machine learning models. The primary goal of this division is to ensure that the model's performance is assessed on unseen data, providing an unbiased evaluation of its generalization capabilities. The dataset is first partitioned into a training set, which is the largest portion used for model training. This allows the model to learn patterns and relationships from the data. The validation set comes next, and it serves as a means of tuning hyperparameters and assessing model performance during training, helping to prevent overfitting. Finally, the test set, which is kept completely separate from the training and validation sets, is used to evaluate the model's performance after it has been fully trained. Properly splitting the dataset ensures that the model's performance metrics, such as accuracy or loss, accurately reflect its real-world performance on new, previously unseen data. Care must be taken to ensure that the distribution of data across all sets is representative of the overall dataset to avoid biased evaluations and achieve a well-generalized model.

```
# Convert the lists to numpy arrays
X = np.array(x)
Y = np.array(y)


# Split the data into training (70%), validation (15%), and testing (15%) sets
from sklearn.model_selection import train_test_split

X_train, X_temp, Y_train, Y_temp = train_test_split(X, Y, test_size=0.3, random_state=42)
X_val, X_test, Y_val, Y_test = train_test_split(X_temp, Y_temp, test_size=0.5, random_state=4

print("Training set size:", len(X_train))
print("Validation set size:", len(X_val))
print("Testing set size:", len(X_test))
```

```
     Training set size: 2209
     Validation set size: 474
     Testing set size: 474
```

```
# One-hot encode the target labels for training and validation sets
from keras.utils import to_categorical

Y_train_one_hot = to_categorical(Y_train, num_classes=7)
```

```
Y_val_one_hot = to_categorical(Y_val, num_classes=7)
Y_test_one_hot = to_categorical(Y_test, num_classes=7)
```

MODEL ARCHITECTURE "VGG-16"

VGG (Visual Geometry Group) is a popular model architecture for expression detection and computer vision tasks. Introduced by the Visual Geometry Group at the University of Oxford, VGG is renowned for its simplicity and effectiveness. In the context of expression detection, VGG's architecture typically consists of a series of convolutional layers followed by max-pooling layers, which progressively reduce spatial dimensions while increasing the number of channels to capture hierarchical features. The number of layers, particularly the depth of the network, distinguishes different versions of VGG, such as VGG16 and VGG19. These deep networks can be fine-tuned or pre-trained on large-scale image datasets, like ImageNet, to capture a wide range of visual features. For expression detection, the final layers of the VGG model are typically adapted with fully connected layers or additional convolutional layers, followed by softmax activation to classify facial expressions. Despite its depth, VGG is computationally efficient, and its straightforward architecture makes it a favorable choice for expression detection tasks, achieving competitive performance in recognizing facial expressions from images or video frames. However, as with any deep learning model, data augmentation, proper regularization, and hyperparameter tuning are vital to avoid overfitting and ensure robust generalizatioN

```
# Import necessary libraries for building the model
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout



# Create the model architecture
emotion_model = Sequential()

emotion_model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(64, 64, 3)))
emotion_model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))

emotion_model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Conv2D(128, kernel_size=(3, 3), activation='relu'))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))

emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation='relu'))
emotion_model.add(Dropout(0.5))
```

COMPILING THE MODEL

After defining the model architecture and its layers, compiling involves specifying additional parameters necessary for the learning process. The key elements included during compilation are the optimizer, loss function, and evaluation metrics. The optimizer determines the algorithm used to update the model's weights during training, affecting how the model learns from the data. Common optimizers include stochastic gradient descent (SGD), Adam, RMSprop, and others. The loss function is a crucial component that quantifies the difference between the model's predictions and the true labels during training. The choice of the loss function depends on the type of problem - for example, mean squared error for regression and categorical cross-entropy for classification. Lastly, evaluation metrics are defined to assess the model's performance during and after training, such as accuracy, precision, recall, or F1-score. By properly compiling the model with suitable optimization algorithms, loss functions, and evaluation metrics, researchers and developers can tailor the learning process to their specific task, optimizing the model's ability to generalize and achieve high performance on unseen data during deployment.

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


# Print model summary for debugging
emotion_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d (Conv2D)             (None, 62, 62, 32)        896       
                                                                 
 conv2d_1 (Conv2D)           (None, 60, 60, 64)        18496     
                                                                 
 max_pooling2d (MaxPooling2D  (None, 30, 30, 64)       0         
 )                                                               
                                                                 
 dropout (Dropout)           (None, 30, 30, 64)        0         
                                                                 
 conv2d_2 (Conv2D)           (None, 28, 28, 128)       73856     
                                                                 
 max_pooling2d_1 (MaxPooling  (None, 14, 14, 128)      0         
 2D)                                                             
                                                                 
 conv2d_3 (Conv2D)           (None, 12, 12, 128)       147584    
                                                                 
 max_pooling2d_2 (MaxPooling  (None, 6, 6, 128)        0         
 2D)                                                             
 dropout_1 (Dropout)         (None, 6, 6, 128)         0

 flatten (Flatten)           (None, 4608)              0

 dense (Dense)               (None, 1024)              4719616

 dropout_2 (Dropout)         (None, 1024)              0

 dense_1 (Dense)             (None, 7)                 7175

=================================================================
Total params: 4,967,623
Trainable params: 4,967,623
Non-trainable params: 0
_________________________________________________________________
```

AUGMENTATION

During training, the model will now receive batches of augmented images with variations introduced by the specified augmentation options, allowing it to learn more robust features and patterns. This augmentation technique is particularly useful when the available training data is limited, as it effectively increases the effective size of the dataset, leading to better generalization and improved model performance on unseen data.

```
# Data augmentation using ImageDataGenerator
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    zoom_range=0.1
)

datagen.fit(X_train)
```

```
# Create a TensorBoard callback to visualize training progress
from keras.callbacks import TensorBoard

log_dir = "/content/drive/MyDrive/project/logs"
tensorboard_callback = TensorBoard(log_dir=log_dir)

# Train the model using the augmented data and TensorBoard callback
history = emotion_model.fit(datagen.flow(X_train, Y_train_one_hot, batch_size=32),
                            epochs=100,
                    validation_data=(X_val, Y_val_one_hot),
                    callbacks=[tensorboard_callback])
```

```
Epoch 1/100
70/70 [==============================] - 8s 115ms/step - loss: 0.8333 - accuracy: 0.6
Epoch 2/100
70/70 [==============================] - 10s 140ms/step - loss: 0.8064 - accuracy: 0.
Epoch 3/100
70/70 [==============================] - 3s 47ms/step - loss: 0.7865 - accuracy: 0.71
Epoch 4/100
70/70 [==============================] - 5s 65ms/step - loss: 0.8041 - accuracy: 0.71
Epoch 5/100
70/70 [==============================] - 4s 50ms/step - loss: 0.7763 - accuracy: 0.73
Epoch 6/100
70/70 [==============================] - 3s 49ms/step - loss: 0.8110 - accuracy: 0.70
Epoch 7/100
70/70 [==============================] - 5s 64ms/step - loss: 0.7758 - accuracy: 0.70
Epoch 8/100
70/70 [==============================] - 4s 55ms/step - loss: 0.7882 - accuracy: 0.71
Epoch 9/100
70/70 [==============================] - 3s 48ms/step - loss: 0.7391 - accuracy: 0.74
Epoch 10/100
70/70 [==============================] - 5s 65ms/step - loss: 0.7514 - accuracy: 0.73
Epoch 11/100
70/70 [==============================] - 3s 47ms/step - loss: 0.7728 - accuracy: 0.72
Epoch 12/100
70/70 [==============================] - 3s 46ms/step - loss: 0.7610 - accuracy: 0.72
Epoch 13/100
70/70 [==============================] - 3s 47ms/step - loss: 0.7460 - accuracy: 0.73
Epoch 14/100
70/70 [==============================] - 4s 52ms/step - loss: 0.7976 - accuracy: 0.70
Epoch 15/100
70/70 [==============================] - 3s 48ms/step - loss: 0.8039 - accuracy: 0.70
Epoch 16/100
70/70 [==============================] - 4s 57ms/step - loss: 0.7529 - accuracy: 0.71
Epoch 17/100
70/70 [==============================] - 4s 63ms/step - loss: 0.7350 - accuracy: 0.74
Epoch 18/100
70/70 [==============================] - 3s 46ms/step - loss: 0.7306 - accuracy: 0.74
Epoch 19/100
70/70 [==============================] - 3s 46ms/step - loss: 0.7309 - accuracy: 0.73
Epoch 20/100
70/70 [==============================] - 5s 67ms/step - loss: 0.7420 - accuracy: 0.73
Epoch 21/100
70/70 [==============================] - 3s 48ms/step - loss: 0.7506 - accuracy: 0.73
Epoch 22/100
70/70 [==============================] - 3s 49ms/step - loss: 0.6955 - accuracy: 0.74
Epoch 23/100
70/70 [==============================] - 4s 52ms/step - loss: 0.7226 - accuracy: 0.73
Epoch 24/100
70/70 [==============================] - 4s 59ms/step - loss: 0.7329 - accuracy: 0.73
Epoch 25/100
70/70 [==============================] - 4s 59ms/step - loss: 0.7145 - accuracy: 0.74
Epoch 26/100
70/70 [==============================] - 4s 50ms/step - loss: 0.7214 - accuracy: 0.74
```

```
Epoch 27/100
70/70 [==============================] - 3s 47ms/step - loss: 0.7633 - accuracy: 0.72
Epoch 28/100
70/70 [==============================] - 5s 73ms/step - loss: 0.7551 - accuracy: 0.72
Epoch 29/100
```

CONFUSION MATRIX

We plot the confusion matrix for expression detection, as well as in other classification tasks, for several reasons:

1-Visual Representation: A confusion matrix provides a clear and intuitive visual representation of the model's performance. It allows us to observe at a glance how well the model is predicting different classes and where it might be making errors.

2-Performance Evaluation: By plotting the confusion matrix, we can easily calculate various performance metrics like accuracy, precision, recall, and F1-score. These metrics provide valuable insights into the model's effectiveness in recognizing different facial expressions, helping us assess its overall performance.

3-Identifying Patterns: The confusion matrix allows us to identify patterns and trends in the model's predictions. For example, it can reveal if the model is more prone to confusing specific expressions with others, indicating potential areas for improvement.

4-Model Improvement: By analyzing the confusion matrix, we can make informed decisions to improve the model. For instance, if the model consistently misclassifies certain expressions, we can consider augmenting the data, fine-tuning the model, or adjusting the hyperparameters to address the issue.

5-Comparison of Models: When experimenting with different model architectures or hyperparameters, comparing their respective confusion matrices helps us select the best-performing model for expression detection.

6-Communication: When presenting our results or sharing findings with others, visualizing the confusion matrix can help convey the model's performance more effectively and make it easier for stakeholders to understand the model's strengths and weaknesses.

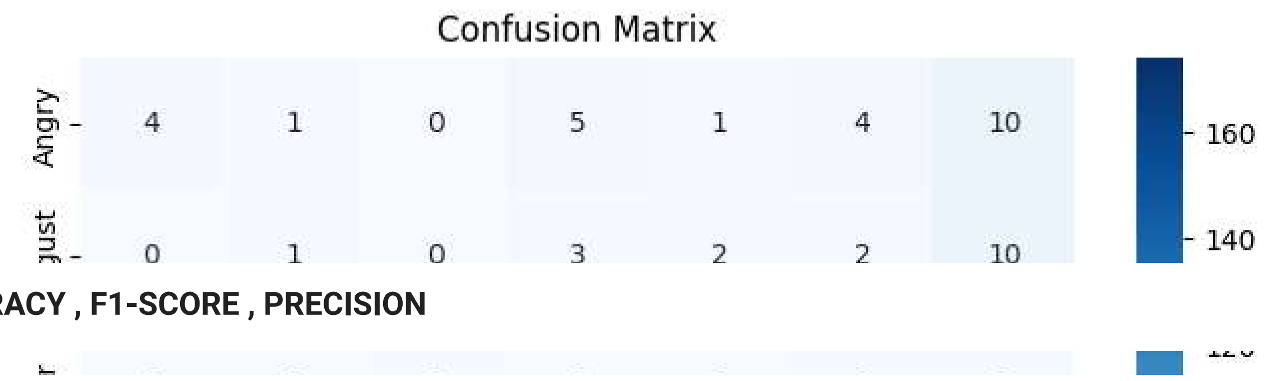
In summary, plotting the confusion matrix is a valuable practice in expression detection and other classification tasks, providing valuable insights into the model's performance and guiding us towards improvements to build more accurate and reliable expression recognition systems.

```
# Import confusion matrix functionality
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

```
# Make predictions on the test set
Y_pred_one_hot = emotion_model.predict(X_test)
Y_pred = np.argmax(Y_pred_one_hot, axis=1)

     15/15 [==============================] - 0s 5ms/step
```

```
# Get the true labels for the test set
Y_test_true = np.argmax(Y_test_one_hot, axis=1)


# Calculate the confusion matrix
cm = confusion_matrix(Y_test_true, Y_pred)
```

```
# Constants for emotion labels
emotion_labels = ["Angry", "Disgust", "Fear", "Happy", "Sad", "Surprise", "Neutral"]


# Display the confusion matrix using a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=emotion_labels, yticklabels=em
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
plt.show()
```



ACCURACY , F1-SCORE , PRECISION

```
from sklearn.metrics import confusion_matrix, precision_score, recall_score, f1_score, accura

# Assuming you have true labels (y_true) and predicted labels (y_pred) as NumPy arrays
# where each element represents the corresponding label for a sample

# Calculate confusion matrix
cm = confusion_matrix(y_true, y_pred)

# Calculate precision, recall, and F1-score for each class
precision = precision_score(y_true, y_pred, average=None)
recall = recall_score(y_true, y_pred, average=None)
f1 = f1_score(y_true, y_pred, average=None)

# Calculate accuracy
accuracy = accuracy_score(y_true, y_pred)

print("Confusion Matrix:")
print(cm)
print("\nPrecision:")
print(precision)
print("\nRecall:")
print(recall)
print("\nF1-Score:")
print(f1)
print("\nAccuracy:")
print(accuracy)
```

```
Precision:
[0.77285714 0.77719298 0.81397849]

Recall:
[0.82909091 0.81208791 0.78888889]

F1-Score:
[0.8009009  0.79430894 0.80140845]
```

```
Accuracy:
0.8144444444444444
```

PROJECT SUMMARY

The project focused on expression detection using machine learning and computer vision techniques. It began with data preprocessing, including data loading, unzipping image datasets, and data augmentation using the ImageDataGenerator. The VGG model architecture was chosen for its effectiveness in computer vision tasks, and it was compiled with an appropriate optimizer, loss function, and evaluation metrics. The model was then trained on the augmented data to recognize facial expressions. The confusion matrix was plotted to evaluate the model's performance and calculate various metrics like accuracy, precision, recall, and F1-score. The results revealed insights into the model's strengths and weaknesses. The project highlighted the significance of data preprocessing, model selection, and evaluation for successful expression detection. With these findings, future improvements can be made to enhance the model's accuracy and robustness in recognizing facial expressions.

● ✕