face

December 2, 2024

[1]:
```
# Core Libraries
import os  # For file and directory management
from pathlib import Path  # For advanced path manipulations
import gc  # For garbage collection to manage memory
import numpy as np  # For numerical operations and arrays

# Image Processing Libraries
import cv2  # OpenCV for image reading and manipulation
from PIL import Image  # PIL for image operations

# Face Detection and Preprocessing
from mtcnn import MTCNN  # Multi-task Cascaded Convolutional Networks for face␣
 ↪detection
from tensorflow.keras.preprocessing.image import load_img, img_to_array  # For␣
 ↪image loading and preprocessing
from tensorflow.keras.applications import MobileNetV2  # Pre-trained␣
 ↪MobileNetV2 model
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input  #␣
 ↪Preprocessing for MobileNetV2 inputs

# Data Augmentation Libraries
import imgaug.augmenters as iaa  # For creating augmentations
from tqdm import tqdm  # Progress bar for loops

# Machine Learning and Model Training Libraries
from sklearn.preprocessing import LabelEncoder  # For encoding class labels
from sklearn.model_selection import train_test_split  # For splitting datasets
from sklearn.metrics import classification_report, confusion_matrix  # For␣
 ↪model evaluation

# Deep Learning Framework
from tensorflow.keras.models import Sequential, Model, load_model  # Model␣
 ↪creation and loading
from tensorflow.keras.layers import (
    Dense, Dropout, BatchNormalization, GlobalAveragePooling2D, Conv2D, Input
)
from tensorflow.keras.optimizers import Adam  # Optimizer
```

```
from tensorflow.keras.callbacks import EarlyStopping  # For early stopping␣
↪during training
from tensorflow.keras.utils import to_categorical  # For converting labels to␣
↪one-hot encoding
from tensorflow.keras import regularizers  # For applying regularization to␣
↪model layers

# Visualization Libraries
import matplotlib.pyplot as plt  # Plotting library
import seaborn as sns  # For creating heatmaps and other visualizations

# Parallel Processing Libraries
from concurrent.futures import ThreadPoolExecutor  # For parallel processing

# Utility Libraries
import joblib  # For saving and loading Python objects
```

Data augments

[2]:
```
# Paths
path = r"D:\study\code\project\Face_Recognition\facedataset"
aug_path = r"D:\study\code\project\Face_Recognition\augmented_data"
```

[3]:
```
# Define augmentations with reasonable limits
augmenters = [
    iaa.Affine(rotate=(-20, 20)),                          # Random rotation
    iaa.Fliplr(0.5),                                       # Horizontal flip
    iaa.GaussianBlur(sigma=(0, 1.5)),                      # Limited Gaussian␣
↪blur
    iaa.GammaContrast(gamma=(0.8, 1.2)),                   # Mild gamma contrast
    iaa.AdditiveGaussianNoise(scale=(0, 0.05 * 255)),      # Mild Gaussian noise
    iaa.Multiply((0.9, 1.1)),                              # Small brightness␣
↪adjustment
    iaa.Affine(translate_percent=(-0.05, 0.05)),           # Small translations
    iaa.AddToHueAndSaturation(value=(-5, 5)),              # Mild hue/saturation␣
↪adjustment
    iaa.Grayscale(alpha=(0.0, 0.5)),                       # Partial grayscale
    iaa.Crop(percent=(0, 0.1)),                            # Limited random␣
↪cropping
    iaa.Resize({"height": (0.9, 1.1), "width": (0.9, 1.1)})  # Small resizing␣
↪adjustments
]
```

[4]:
```
# Ensure the output directory exists
os.makedirs(aug_path, exist_ok=True)
```

```python
[ ]: # Process images
for folder in os.listdir(path):
    folder_path = os.path.join(path, folder)
    aug_folder_path = os.path.join(aug_path, folder)
    os.makedirs(aug_folder_path, exist_ok=True)  # Create folder for augmented␣
 ↪data if not exists

    print(f"Processing folder: {folder}")

    for img_name in os.listdir(folder_path):
        img_path = os.path.join(folder_path, img_name)

        # Read the image using PIL and convert to numpy array
        image = np.array(Image.open(img_path))

        for i, augmenter in enumerate(augmenters):
            seq = iaa.Sequential([augmenter])  # Apply one augmenter at a time
            augmented_image = seq(image=image)  # Apply augmentation

            # Save augmented image
            aug_img_name = f"{os.path.splitext(img_name)[0]}_aug_{i}.jpg"
            aug_img_path = os.path.join(aug_folder_path, aug_img_name)

            # Convert to RGB for saving using OpenCV
            augmented_image = cv2.cvtColor(augmented_image, cv2.COLOR_BGR2RGB)
            cv2.imwrite(aug_img_path, augmented_image)

            print(f"Saved: {aug_img_name}")

print("Data augmentation complete.")
```

Face Detection and Dataset Preparation

```python
[6]: # Paths
input_path = r"D:\study\code\project\Face_Recognition\augmented_data"
output_x_file = "face_embeddings.npy"
output_y_file = "labels.npy"
log_file_path = "process_log.txt"
mapping_file = "class_label_mapping.txt"
```

```python
[7]: # Initialize variables
x, y = [], []
skipped_count = 0
```

```python
[8]: # Map folder names (classes) to numerical labels
label_map = {folder: idx for idx, folder in enumerate(sorted(os.
 ↪listdir(input_path)))}
```

```python
# Save and display the class-to-label mapping
print("Class-to-Label Mapping:")
with open(mapping_file, "w") as file:
    file.write("Class-to-Label Mapping:\n")
    for class_name, label in label_map.items():
        print(f"Class: {class_name} -> Label: {label}")
        file.write(f"Class: {class_name} -> Label: {label}\n")
print(f"\nClass-to-label mapping saved to: {mapping_file}")
```

```
Class-to-Label Mapping:
Class: K_P -> Label: 0
Class: Md_azam -> Label: 1
Class: Mukul_Bindal -> Label: 2
Class: abha -> Label: 3
Class: abhishek -> Label: 4
Class: abhishek_chauhan -> Label: 5
Class: ajita -> Label: 6
Class: akshat_goyal -> Label: 7
Class: akshat_jain -> Label: 8
Class: ankita -> Label: 9
Class: anurag -> Label: 10
Class: bhoomika -> Label: 11
Class: jatin_jha -> Label: 12
Class: jewal_sharma -> Label: 13
Class: mansi -> Label: 14
Class: neelesh -> Label: 15
Class: prabhat -> Label: 16
Class: priyanshu -> Label: 17
Class: rahul_sharma -> Label: 18
Class: raj_singh -> Label: 19
Class: rohan -> Label: 20
Class: satyam -> Label: 21
Class: shruti_tripathi -> Label: 22
Class: sojal -> Label: 23
Class: suneha_goyal -> Label: 24
Class: vanshita -> Label: 25
Class: vivek_mishra -> Label: 26

Class-to-label mapping saved to: class_label_mapping.txt
```

[9]:
```python
# Initialize MTCNN detector
detector = MTCNN()
```

[10]:
```python
# Load Pre-trained Model (MobileNetV2)
base_model = MobileNetV2(weights="imagenet", include_top=False,
 ↪input_shape=(224, 224, 3))
```

```
embedding_model = Model(inputs=base_model.input, outputs=base_model.output)
```

[11]:
```
# Function to log messages
def write_log(message):
    with open(log_file_path, "a") as log_file:
        log_file.write(message + "\n")
```

[12]:
```
# Function to verify if a cropped region contains a valid face
def is_valid_face(face_region):
    if face_region is None or face_region.size == 0:
        return False
    if np.var(face_region) < 10:  # Check for minimal pixel variance
        return False
    return True
```

[ ]:
```
# Function to process a single image
def process_image(img_path, label):
    global skipped_count
    try:
        # Read image
        img = cv2.imread(img_path)
        if img is None:
            raise ValueError(f"Image not found or could not be read: 
 ↪{img_path}")

        # Resize image for faster MTCNN detection
        img = cv2.resize(img, (640, 480))

        # Convert BGR to RGB for MTCNN
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # Detect faces
        results = detector.detect_faces(img)
        if not results:
            raise ValueError("No face detected")

        # Extract face
        x1, y1, width, height = results[0]['box']
        x1, y1 = max(0, x1), max(0, y1)  # Ensure coordinates are within bounds
        x2, y2 = x1 + width, y1 + height
        face = img[y1:y2, x1:x2]

        # Verify face validity
        if not is_valid_face(face):
            raise ValueError("Invalid face region (low pixel variance)")

        # Preprocess face for MobileNetV2
```

```
        face = cv2.resize(face, (224, 224))
        face = preprocess_input(face)
        face = np.expand_dims(face, axis=0)

        # Generate face embeddings
        embedding = embedding_model.predict(face)[0]

        # Log successful processing
        write_log(f"Processed image: {img_path}")

        return embedding, label
    except Exception as e:
        skipped_count += 1
        error_message = f"Skipped {skipped_count}: {e} (Image: {img_path})"
        print(error_message)
        write_log(error_message)
        return None, None
```

[ ]:
```
# Process images in batches
def process_folder(folder):
    folder_path = os.path.join(input_path, folder)
    img_paths = [(os.path.join(folder_path, img_name), label_map[folder]) for␣
 ↪img_name in os.listdir(folder_path)]

    # Process images using ThreadPoolExecutor
    results = []
    with ThreadPoolExecutor(max_workers=4) as executor:  # Limit workers to␣
 ↪prevent memory issues
        results = list(tqdm(executor.map(lambda args: process_image(*args),␣
 ↪img_paths), total=len(img_paths)))

    return results
```

[ ]:
```
# Main script
if __name__ == "__main__":
    print("Processing images...")
    write_log("Starting image processing...")

    # Get sorted folder names to ensure consistent label mapping
    folders = sorted(os.listdir(input_path))
    for folder in tqdm(folders, desc="Processing folders"):
        write_log(f"Processing folder: {folder}")
        results = process_folder(folder)
        for embedding, label in results:
            if embedding is not None:
                x.append(embedding)
                y.append(label)
```

```
        # Release memory after each folder
        gc.collect()

    # Save processed embeddings and labels
    print(f"Saving processed data: {len(x)} embeddings, {len(y)} labels")
    write_log(f"Saving processed data: {len(x)} embeddings, {len(y)} labels")
    np.save(output_x_file, np.array(x, dtype=np.float32))
    np.save(output_y_file, np.array(y, dtype=np.int32))  # Save labels as␣
↪integers
    write_log(f"Data saved: {output_x_file}, {output_y_file}")
    print(f"Data saved: {output_x_file}, {output_y_file}")
    print(f"Total skipped images: {skipped_count}")
    write_log(f"Total skipped images: {skipped_count}")
```

Model Training

[16]:
```
# Paths
embeddings_file = "face_embeddings.npy"
labels_file = "labels.npy"
model_save_path = "face_recognition_model.keras"
label_encoder_path = "label_encoder.pkl"
```

[17]:
```
# Load embeddings and labels
print("Loading embeddings and labels...")
x = np.load(embeddings_file)
y = np.load(labels_file)
```

```
Loading embeddings and labels…
```

[18]:
```
print(f"Embeddings shape: {x.shape}, Labels shape: {y.shape}")
```

```
Embeddings shape: (8904, 7, 7, 1280), Labels shape: (8904,)
```

[19]:
```
# Encode labels
print("Encoding labels...")
encoder = LabelEncoder()
y_encoded = encoder.fit_transform(y)
y_categorical = to_categorical(y_encoded)  # Convert to one-hot encoding for␣
↪neural network
```

```
Encoding labels…
```

[20]:
```
# Save the label encoder for later use
joblib.dump(encoder, label_encoder_path)
print(f"Label encoder saved to: {label_encoder_path}")
```

```
Label encoder saved to: label_encoder.pkl
```

[21]:
```
# Split data into training and test sets
print("Splitting data into training and test sets...")
x_train, x_test, y_train, y_test = train_test_split(
    x, y_categorical, test_size=0.2, random_state=42, stratify=y_encoded
)
```

Splitting data into training and test sets…

[22]:
```
# Define the model
model = Sequential([
    # Explicit Input layer
    Input(shape=(x.shape[1], x.shape[2], 1280)),
    # Add Conv2D layer for feature extraction if input has spatial dimensions
    Conv2D(64, (3, 3), activation='relu', padding='same',␣
 ↪kernel_regularizer=regularizers.l2(0.01)),
    BatchNormalization(),
    Dropout(0.3),

    Conv2D(128, (3, 3), activation='relu', padding='same',␣
 ↪kernel_regularizer=regularizers.l2(0.01)),
    BatchNormalization(),
    Dropout(0.3),

    GlobalAveragePooling2D(),  # Reduce dimensions while retaining features

    # Fully connected layers for classification
    Dense(512, activation='relu', kernel_regularizer=regularizers.l2(0.01)),
    BatchNormalization(),
    Dropout(0.5),

    Dense(256, activation='relu', kernel_regularizer=regularizers.l2(0.01)),
    BatchNormalization(),
    Dropout(0.4),

    Dense(len(encoder.classes_), activation='softmax',␣
 ↪kernel_regularizer=regularizers.l2(0.01))
])
```

[23]:
```
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
```

[24]:
```
# Model summary
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 7, 7, 64)	737,344
batch_normalization (BatchNormalization)	(None, 7, 7, 64)	256
dropout (Dropout)	(None, 7, 7, 64)	0
conv2d_13 (Conv2D)	(None, 7, 7, 128)	73,856
batch_normalization_1 (BatchNormalization)	(None, 7, 7, 128)	512
dropout_1 (Dropout)	(None, 7, 7, 128)	0
global_average_pooling2d (GlobalAveragePooling2D)	(None, 128)	0
dense_7 (Dense)	(None, 512)	66,048
batch_normalization_2 (BatchNormalization)	(None, 512)	2,048
dropout_2 (Dropout)	(None, 512)	0
dense_8 (Dense)	(None, 256)	131,328
batch_normalization_3 (BatchNormalization)	(None, 256)	1,024
dropout_3 (Dropout)	(None, 256)	0
dense_9 (Dense)	(None, 27)	6,939

Total params: 1,019,355 (3.89 MB)

Trainable params: 1,017,435 (3.88 MB)

Non-trainable params: 1,920 (7.50 KB)

[25]:
```
# Train the model with early stopping
print("Training the model...")
```

```python
early_stopping = EarlyStopping(monitor='val_loss', patience=5,␣
    ↪restore_best_weights=True)
history = model.fit(
    x_train, y_train,
    validation_data=(x_test, y_test),
    epochs=50,
    batch_size=32,
    callbacks=[early_stopping],
    verbose=1
)
```

```
Training the model…
Epoch 1/50
223/223              14s 44ms/step -
accuracy: 0.6022 - loss: 8.9290 - val_accuracy: 0.9646 - val_loss: 5.1741
Epoch 2/50
223/223              9s 40ms/step -
accuracy: 0.9604 - loss: 3.9597 - val_accuracy: 0.9523 - val_loss: 2.4333
Epoch 3/50
223/223              8s 37ms/step -
accuracy: 0.9610 - loss: 1.9955 - val_accuracy: 0.9736 - val_loss: 1.3413
Epoch 4/50
223/223              8s 37ms/step -
accuracy: 0.9670 - loss: 1.2027 - val_accuracy: 0.8905 - val_loss: 1.3012
Epoch 5/50
223/223              8s 36ms/step -
accuracy: 0.9575 - loss: 1.0512 - val_accuracy: 0.9483 - val_loss: 1.0504
Epoch 6/50
223/223              8s 36ms/step -
accuracy: 0.9578 - loss: 0.9454 - val_accuracy: 0.9214 - val_loss: 1.0132
Epoch 7/50
223/223              8s 36ms/step -
accuracy: 0.9498 - loss: 0.9596 - val_accuracy: 0.9517 - val_loss: 1.0151
Epoch 8/50
223/223              8s 35ms/step -
accuracy: 0.9583 - loss: 0.9484 - val_accuracy: 0.8613 - val_loss: 1.2597
Epoch 9/50
223/223              8s 34ms/step -
accuracy: 0.9539 - loss: 0.9894 - val_accuracy: 0.9090 - val_loss: 1.1396
Epoch 10/50
223/223              8s 34ms/step -
accuracy: 0.9498 - loss: 0.9842 - val_accuracy: 0.7479 - val_loss: 1.6284
Epoch 11/50
223/223              8s 35ms/step -
accuracy: 0.9549 - loss: 0.9949 - val_accuracy: 0.9512 - val_loss: 0.9673
Epoch 12/50
223/223              8s 34ms/step -
accuracy: 0.9474 - loss: 0.9802 - val_accuracy: 0.7024 - val_loss: 1.9225
```

```
Epoch 13/50
223/223            8s 34ms/step -
accuracy: 0.9443 - loss: 1.0670 - val_accuracy: 0.9220 - val_loss: 1.1144
Epoch 14/50
223/223            8s 35ms/step -
accuracy: 0.9516 - loss: 0.9793 - val_accuracy: 0.9315 - val_loss: 1.0442
Epoch 15/50
223/223            8s 34ms/step -
accuracy: 0.9490 - loss: 0.9857 - val_accuracy: 0.9242 - val_loss: 1.0029
Epoch 16/50
223/223            8s 34ms/step -
accuracy: 0.9536 - loss: 0.9216 - val_accuracy: 0.8742 - val_loss: 1.1950
```

[26]:
```
# Save the trained model
model.save(model_save_path)
print(f"Model saved to: {model_save_path}")
```

```
Model saved to: face_recognition_model.keras
```

Evaluation

[27]:
```
# Evaluate the model on test data
print("Evaluating the model on test data...")
loss, accuracy = model.evaluate(x_test, y_test, verbose=0)
print(f"Test Accuracy: {accuracy:.4f}")
```

```
Evaluating the model on test data…
Test Accuracy: 0.9512
```

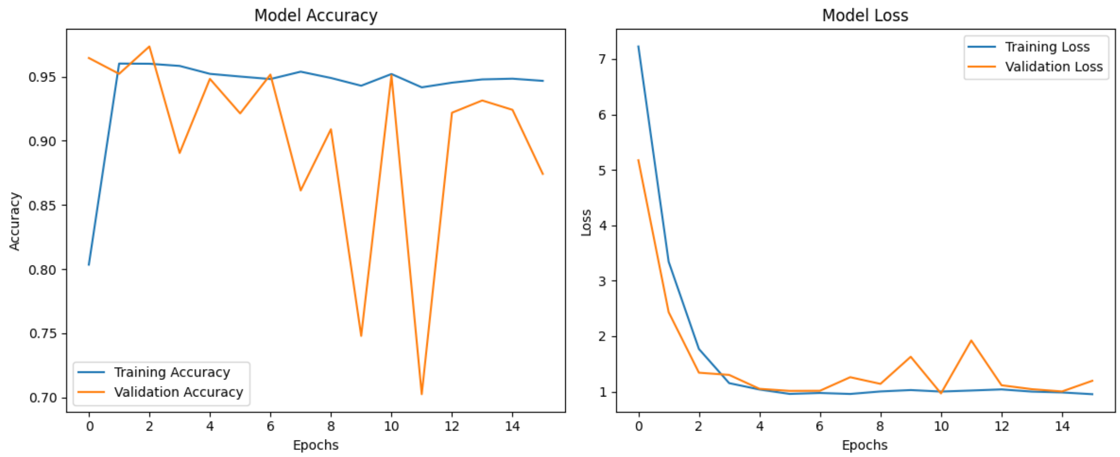
[28]:
```
# Plot training history
def plot_training_history(history):
    # Accuracy
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Model Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    # Loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Model Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
```

```
    plt.tight_layout()
    plt.show()

print("Plotting training history...")
plot_training_history(history)
```

Plotting training history…



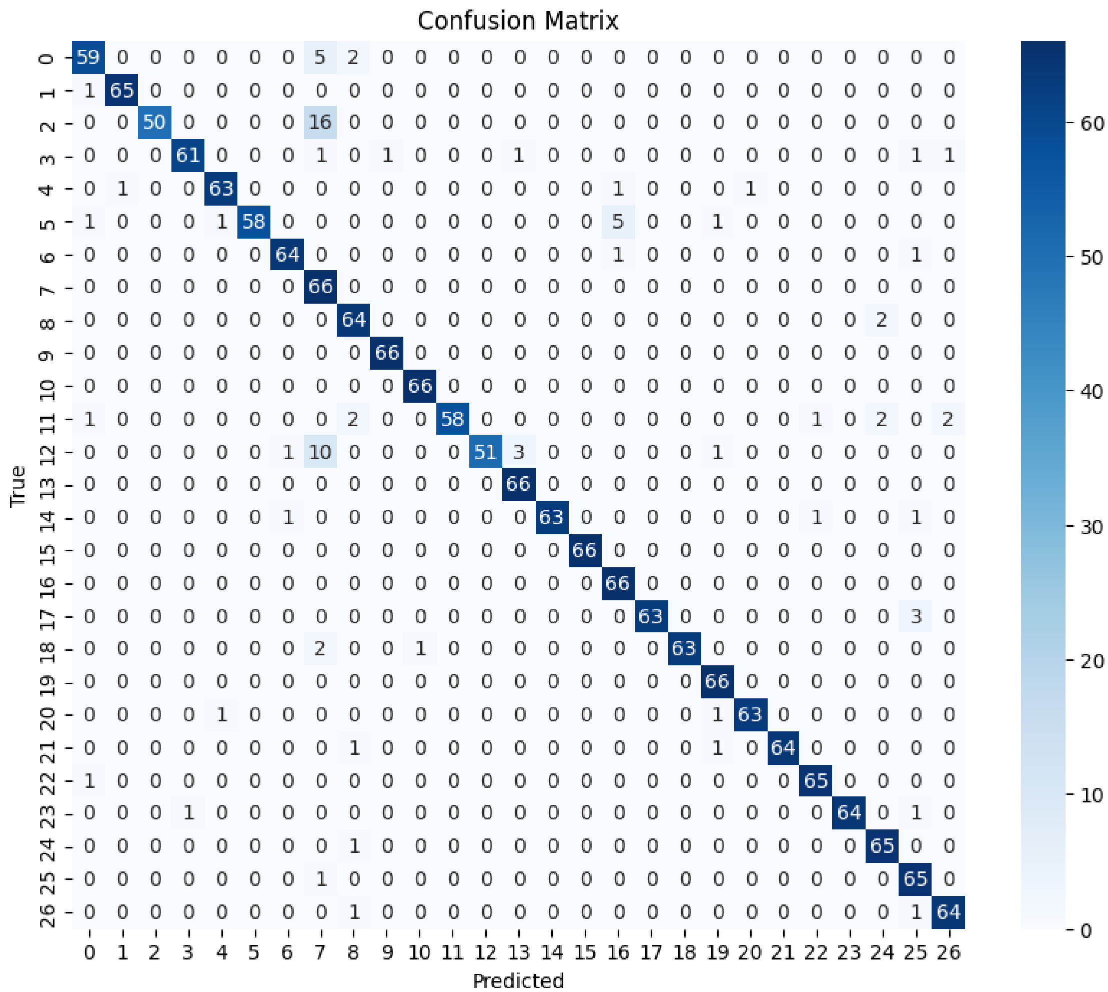
[29]:

```
# Confusion Matrix and Classification Report
print("Generating confusion matrix and classification report...")
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)

cm = confusion_matrix(y_true_classes, y_pred_classes)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=encoder.
 ↪classes_, yticklabels=encoder.classes_)
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()

print("Classification Report:")
print(classification_report(y_true_classes, y_pred_classes,␣
 ↪target_names=encoder.classes_.astype(str)))
```

Generating confusion matrix and classification report…
56/56   1s 14ms/step



Classification Report:

	precision	recall	f1-score	support
0	0.94	0.89	0.91	66
1	0.98	0.98	0.98	66
2	1.00	0.76	0.86	66
3	0.98	0.92	0.95	66
4	0.97	0.95	0.96	66
5	1.00	0.88	0.94	66
6	0.97	0.97	0.97	66
7	0.65	1.00	0.79	66
8	0.90	0.97	0.93	66
9	0.99	1.00	0.99	66
10	0.99	1.00	0.99	66
11	1.00	0.88	0.94	66
12	1.00	0.77	0.87	66

```
          13       0.94      1.00      0.97        66
          14       1.00      0.95      0.98        66
          15       1.00      1.00      1.00        66
          16       0.90      1.00      0.95        66
          17       1.00      0.95      0.98        66
          18       1.00      0.95      0.98        66
          19       0.94      1.00      0.97        66
          20       0.98      0.97      0.98        65
          21       1.00      0.97      0.98        66
          22       0.97      0.98      0.98        66
          23       1.00      0.97      0.98        66
          24       0.94      0.98      0.96        66
          25       0.89      0.98      0.94        66
          26       0.96      0.97      0.96        66

    accuracy                           0.95      1781
   macro avg       0.96      0.95      0.95      1781
weighted avg       0.96      0.95      0.95      1781
```

[30]:
```python
# Function for inference
def recognize_face(face_embedding):
    """Predict the label of a given face embedding."""
    prediction = model.predict(np.expand_dims(face_embedding, axis=0))
    predicted_class = np.argmax(prediction)
    label = encoder.inverse_transform([predicted_class])[0]
    confidence = prediction[0][predicted_class]
    return label, confidence
```

[31]:
```python
# Test example
test_embedding = x_test[0]  # Use a test embedding as an example
predicted_label, confidence = recognize_face(test_embedding)
print(f"Predicted Label: {predicted_label}, Confidence: {confidence:.2f}")
```

```
1/1              0s 15ms/step
Predicted Label: 21, Confidence: 1.00
```

Saveing the Model

[32]:
```python
# Loading and Deploying the Model
print("Testing model loading and deployment...")
loaded_model = load_model(model_save_path)
print("Model loaded successfully!")
```

```
Testing model loading and deployment…
Model loaded successfully!
```

[33]:
```python
# Load the embedding extractor (pre-trained model)
```

```
embedding_model = MobileNetV2(weights='imagenet', include_top=False,␣
↪input_shape=(224, 224, 3))
```

[34]:
```
def extract_embeddings(image_path):
    """Extract embeddings from an image using a pre-trained model."""
    img = load_img(image_path, target_size=(224, 224))  # Resize to match␣
↪embedding model input
    img_array = img_to_array(img)  # Convert to array
    img_array = np.expand_dims(img_array, axis=0)  # Add batch dimension
    img_array = preprocess_input(img_array)  # Preprocess for the embedding␣
↪model

    # Generate embeddings
    embeddings = embedding_model.predict(img_array)  # Shape: (1, 7, 7, 1280)
    return embeddings
```

[35]:
```
def predict_face(image_path, classifier_model):
    """Predict the label of an image."""
    # Extract embeddings
    embeddings = extract_embeddings(image_path)  # Shape: (1, 7, 7, 1280)
    # Predict using the classifier model
    predictions = classifier_model.predict(embeddings)  # Expecting (1, 7, 7,␣
↪1280)
    predicted_class = np.argmax(predictions)
    confidence = predictions[0][predicted_class]

    return predicted_class, confidence
```

[36]:
```
# Test Deployment Example
image_path = r"D:\study\code\project\Face_Recognition\test data\12345.jpg" #␣
↪Update this path
predicted_class, confidence = predict_face(image_path, loaded_model)
class_map = {i: label for i, label in enumerate(encoder.classes_)}
print(f"Predicted Class: {class_map[predicted_class]}, Confidence: {confidence:.
↪2f}")
```

```
1/1            1s 883ms/step
1/1            0s 117ms/step
Predicted Class: 19, Confidence: 0.88
```

[ ]: