|  |  |  |
| --- | --- | --- |
| **FACES** | **MODELS & DETECTOR** | **PERCENTAGE** |
| Figure 1 - Azim | VGG-Face (Model) & Retinaface (Backend) |  |
| OpenFace & Retinaface |  |
| Facenet & Retinaface |  |
| Facenet512 & Retinaface |  |

|  |  |  |
| --- | --- | --- |
| Figure 2 - Najmi | VGG-Face & Retinaface |  |
| OpenFace & Retinaface |  |
| Facenet & Retinaface |  |
| Facenet512 & Retinaface |  |
|  |  |  |
| Figure 3 - Syah | VGG-Face & Retinaface |  |
| OpenFace & Retinaface |  |
| Facenet & Retinaface |  |
| Facenet512 & Retinaface |  |
| Figure 4 - Fahmi | VGG-Face & Retinaface |  |
| OpenFace & Retinaface |  |
| Facenet & Retinaface |  |
| Facenet512 & Retinaface |  |
|  |  |  |

**ABOUT CODE**

**Explanation**

1. model\_name='VGG-Face':
   * This specifies which deep learning model to use for creating face embeddings (numerical representations of facial features)
   * VGG-Face is a convolutional neural network model specifically trained for face recognition
   * It converts face images into 2622-dimensional feature vectors that capture facial characteristics
   * Other possible models in DeepFace include 'Facenet', 'OpenFace', 'DeepFace', 'DeepID', 'ArcFace', but this code uses VGG-Face
2. detector\_backend='opencv':
   * This specifies which face detection method to use to locate faces in images
   * 'opencv' refers to OpenCV's cascade classifier for face detection
   * DeepFace supports several other detector backends including:
     + 'mtcnn': Multi-task Cascaded Convolutional Networks
     + 'ssd': Single Shot Detector
     + 'dlib': Dlib's HOG + Linear SVM face detector
     + 'retinaface': RetinaFace detector
     + 'mediapipe': MediaPipe face detector

Interestingly, while this code specifies 'opencv' as the detector\_backend in DeepFace, it actually uses MediaPipe for the main face detection through the detect\_faces method. The OpenCV detector is only used during the embedding creation process.

The choice of these parameters affects:

* Speed of detection/recognition
* Accuracy of results
* Resource usage (memory and CPU/GPU)
* Minimum face size that can be detected
* Tolerance to different face angles and lighting conditions

**Actual Code**

import cv2  
import os  
import numpy as np  
from deepface import DeepFace  
import mediapipe as mp  
import tkinter as tk  
from tkinter import ttk, filedialog, messagebox  
from PIL import Image, ImageTk  
  
  
class FaceRecognitionGUI:  
 def \_\_init\_\_(self, root):  
 self.root = root  
 self.root.title("Face Recognition System")  
 self.root.geometry("1200x800")  
  
 # Make the root window responsive  
 self.root.columnconfigure(0, weight=1)  
 self.root.rowconfigure(0, weight=1)  
  
 # Initialize face recognition system  
 self.recognition\_system = FaceRecognitionSystem()  
  
 # Create GUI elements  
 self.create\_gui()  
  
 def create\_gui(self):  
 # Create main frame  
 main\_frame = ttk.Frame(self.root, padding="10")  
 main\_frame.grid(row=0, column=0, sticky="nsew")  
  
 # Configure main frame grid weights  
 main\_frame.columnconfigure(1, weight=3) # Right frame takes more space  
 main\_frame.columnconfigure(0, weight=1) # Left frame takes less space  
 main\_frame.rowconfigure(0, weight=1)  
  
 # Create left frame for controls  
 left\_frame = ttk.Frame(main\_frame, padding="5")  
 left\_frame.grid(row=0, column=0, sticky="nsew")  
 left\_frame.columnconfigure(0, weight=1)  
  
 # Database section  
 ttk.Label(left\_frame, text="Database Management", font=('Arial', 12, 'bold')).grid(row=0, column=0, pady=5,  
 sticky="w")  
  
 self.database\_path = tk.StringVar(value="./employee\_database")  
 ttk.Label(left\_frame, text="Database Path:").grid(row=1, column=0, pady=2, sticky="w")  
 ttk.Entry(left\_frame, textvariable=self.database\_path).grid(row=2, column=0, pady=2, sticky="ew")  
 ttk.Button(left\_frame, text="Browse Database", command=self.browse\_database).grid(row=3, column=0, pady=5,  
 sticky="ew")  
 ttk.Button(left\_frame, text="Load Database", command=self.load\_database).grid(row=4, column=0, pady=5,  
 sticky="ew")  
  
 # Input image section  
 ttk.Separator(left\_frame, orient='horizontal').grid(row=5, column=0, pady=10, sticky='ew')  
 ttk.Label(left\_frame, text="Image Processing", font=('Arial', 12, 'bold')).grid(row=6, column=0, pady=5,  
 sticky="w")  
  
 ttk.Button(left\_frame, text="Select Image", command=self.browse\_image).grid(row=7, column=0, pady=5,  
 sticky="ew")  
 ttk.Button(left\_frame, text="Process Image", command=self.process\_image).grid(row=8, column=0, pady=5,  
 sticky="ew")  
 ttk.Button(left\_frame, text="Save Result", command=self.save\_result).grid(row=9, column=0, pady=5, sticky="ew")  
  
 # Status section  
 ttk.Separator(left\_frame, orient='horizontal').grid(row=10, column=0, pady=10, sticky='ew')  
 ttk.Label(left\_frame, text="Status", font=('Arial', 12, 'bold')).grid(row=11, column=0, pady=5, sticky="w")  
  
 # Make status text expand with window  
 self.status\_text = tk.Text(left\_frame, wrap=tk.WORD)  
 self.status\_text.grid(row=12, column=0, pady=5, sticky="nsew")  
 left\_frame.rowconfigure(12, weight=1)  
  
 # Create right frame for image display  
 right\_frame = ttk.Frame(main\_frame, padding="5")  
 right\_frame.grid(row=0, column=1, sticky="nsew")  
  
 # Configure right frame grid weights  
 right\_frame.columnconfigure(0, weight=1)  
 right\_frame.rowconfigure(1, weight=1)  
 right\_frame.rowconfigure(3, weight=1)  
  
 # Image display areas  
 ttk.Label(right\_frame, text="Input Image", font=('Arial', 12, 'bold')).grid(row=0, column=0, pady=5)  
  
 # Create frames to contain the image labels  
 self.input\_image\_frame = ttk.Frame(right\_frame)  
 self.input\_image\_frame.grid(row=1, column=0, sticky="nsew")  
 self.input\_image\_frame.columnconfigure(0, weight=1)  
 self.input\_image\_frame.rowconfigure(0, weight=1)  
  
 self.input\_image\_label = ttk.Label(self.input\_image\_frame)  
 self.input\_image\_label.grid(row=0, column=0)  
  
 ttk.Label(right\_frame, text="Processed Image", font=('Arial', 12, 'bold')).grid(row=2, column=0, pady=5)  
  
 self.output\_image\_frame = ttk.Frame(right\_frame)  
 self.output\_image\_frame.grid(row=3, column=0, sticky="nsew")  
 self.output\_image\_frame.columnconfigure(0, weight=1)  
 self.output\_image\_frame.rowconfigure(0, weight=1)  
  
 self.output\_image\_label = ttk.Label(self.output\_image\_frame)  
 self.output\_image\_label.grid(row=0, column=0)  
  
 # Initialize variables  
 self.input\_image\_path = None  
 self.processed\_image = None  
 self.photo\_images = [] # Keep reference to prevent garbage collection  
 self.original\_input\_image = None # Store original input image  
 self.original\_processed\_image = None # Store original processed image  
  
 # Bind resize event  
 self.root.bind("<Configure>", self.on\_window\_resize)  
  
 def on\_window\_resize(self, event):  
 # Only handle window resize events, not other widget configure events  
 if event.widget == self.root:  
 # Update images if they exist  
 if self.input\_image\_path and hasattr(self, 'original\_input\_image'):  
 self.display\_image(self.input\_image\_path, self.input\_image\_label, True)  
 if hasattr(self, 'original\_processed\_image') and self.original\_processed\_image is not None:  
 self.display\_processed\_image(True)  
  
 def update\_status(self, message):  
 self.status\_text.insert(tk.END, f"{message}\n")  
 self.status\_text.see(tk.END)  
 self.root.update\_idletasks()  
  
 def browse\_database(self):  
 folder\_path = filedialog.askdirectory()  
 if folder\_path:  
 self.database\_path.set(folder\_path)  
 self.update\_status(f"Database path set to: {folder\_path}")  
  
 def load\_database(self):  
 try:  
 self.recognition\_system = FaceRecognitionSystem(self.database\_path.get())  
 self.update\_status("Database loaded successfully")  
 except Exception as e:  
 messagebox.showerror("Error", f"Failed to load database: {str(e)}")  
 self.update\_status(f"Error loading database: {str(e)}")  
  
 def browse\_image(self):  
 file\_path = filedialog.askopenfilename(  
 filetypes=[("Image files", "\*.jpg \*.jpeg \*.png \*.bmp \*.gif \*.tiff")]  
 )  
 if file\_path:  
 self.input\_image\_path = file\_path  
 self.update\_status(f"Selected image: {file\_path}")  
 self.display\_image(file\_path, self.input\_image\_label)  
  
 def process\_image(self):  
 if not self.input\_image\_path:  
 messagebox.showwarning("Warning", "Please select an input image first")  
 return  
  
 try:  
 self.update\_status("Processing image...")  
 self.processed\_image = self.recognition\_system.process\_image(self.input\_image\_path)  
  
 if self.processed\_image is not None:  
 self.display\_processed\_image()  
 self.update\_status("Image processed successfully")  
 else:  
 self.update\_status("Failed to process image")  
  
 except Exception as e:  
 messagebox.showerror("Error", f"Failed to process image: {str(e)}")  
 self.update\_status(f"Error processing image: {str(e)}")  
  
 def save\_result(self):  
 if self.processed\_image is None:  
 messagebox.showwarning("Warning", "No processed image to save")  
 return  
  
 file\_path = filedialog.asksaveasfilename(  
 defaultextension=".jpg",  
 filetypes=[("JPEG files", "\*.jpg"), ("All files", "\*.\*")]  
 )  
  
 if file\_path:  
 cv2.imwrite(file\_path, self.processed\_image)  
 self.update\_status(f"Result saved to: {file\_path}")  
  
 def get\_display\_size(self):  
 # Calculate the maximum size for images based on window size  
 right\_frame\_width = self.root.winfo\_width() \* 0.7 # 70% of window width  
 right\_frame\_height = (self.root.winfo\_height() \* 0.45) # 45% of window height for each image  
 return right\_frame\_width, right\_frame\_height  
  
 def display\_image(self, image\_path, label, resize=False):  
 try:  
 if not resize:  
 # Load the original image first time  
 self.original\_input\_image = Image.open(image\_path)  
  
 # Get current display size  
 max\_width, max\_height = self.get\_display\_size()  
  
 # Calculate scaling factor while maintaining aspect ratio  
 image = self.original\_input\_image.copy()  
 scale = min(max\_width / image.width, max\_height / image.height)  
 new\_size = (int(image.width \* scale), int(image.height \* scale))  
  
 # Resize image  
 image = image.resize(new\_size, Image.Resampling.LANCZOS)  
 photo = ImageTk.PhotoImage(image)  
  
 # Update label  
 label.configure(image=photo)  
 self.photo\_images.append(photo) # Keep a reference  
  
 except Exception as e:  
 messagebox.showerror("Error", f"Failed to display image: {str(e)}")  
  
 def display\_processed\_image(self, resize=False):  
 try:  
 if not resize:  
 # Store the original processed image first time  
 self.original\_processed\_image = Image.fromarray(cv2.cvtColor(self.processed\_image, cv2.COLOR\_BGR2RGB))  
  
 # Get current display size  
 max\_width, max\_height = self.get\_display\_size()  
  
 # Calculate scaling factor while maintaining aspect ratio  
 image = self.original\_processed\_image.copy()  
 scale = min(max\_width / image.width, max\_height / image.height)  
 new\_size = (int(image.width \* scale), int(image.height \* scale))  
  
 # Resize image  
 image = image.resize(new\_size, Image.Resampling.LANCZOS)  
 photo = ImageTk.PhotoImage(image)  
  
 # Update label  
 self.output\_image\_label.configure(image=photo)  
 self.photo\_images.append(photo) # Keep a reference  
  
 except Exception as e:  
 messagebox.showerror("Error", f"Failed to display processed image: {str(e)}")  
  
  
class FaceRecognitionSystem:  
 def \_\_init\_\_(self, database\_path='./employee\_database'):  
 self.database\_path = database\_path  
 self.known\_embeddings = {}  
 self.recognition\_threshold = 0.3  
  
 # Initialize MediaPipe Face Detection  
 self.mp\_face\_detection = mp.solutions.face\_detection  
 self.mp\_drawing = mp.solutions.drawing\_utils  
 self.face\_detection = self.mp\_face\_detection.FaceDetection(  
 model\_selection=1,  
 min\_detection\_confidence=0.5  
 )  
  
 print("Initializing face embeddings...")  
 self.\_load\_known\_faces()  
  
 def \_load\_known\_faces(self):  
 *"""Pre-load and cache face embeddings for known faces"""* for person\_folder in os.listdir(self.database\_path):  
 folder\_path = os.path.join(self.database\_path, person\_folder)  
 if os.path.isdir(folder\_path):  
 try:  
 image\_files = [f for f in os.listdir(folder\_path)  
 if f.lower().endswith(('.png', '.jpg', '.jpeg'))]  
 if image\_files:  
 image\_path = os.path.join(folder\_path, image\_files[0])  
 embedding = DeepFace.represent(  
 img\_path=image\_path,  
 model\_name='Facenet512',  
 enforce\_detection=False,  
 detector\_backend='retinaface'  
 )  
 if embedding and len(embedding) > 0:  
 self.known\_embeddings[person\_folder] = embedding[0]['embedding']  
 print(f"Loaded embedding for {person\_folder}")  
 except Exception as e:  
 print(f"Error loading embedding for {person\_folder}: {e}")  
  
 def \_get\_face\_embedding(self, image\_path):  
 *"""Get embedding for face in image"""* try:  
 embedding = DeepFace.represent(  
 img\_path=image\_path,  
 model\_name='Facenet512',  
 enforce\_detection=False,  
 detector\_backend='retinaface'  
 )  
 if embedding and len(embedding) > 0:  
 return embedding[0]['embedding']  
 return None  
 except Exception as e:  
 print(f"Error getting embedding: {e}")  
 return None  
  
 def \_compare\_embeddings(self, embedding1, embedding2):  
 *"""Compare two face embeddings using cosine similarity"""* try:  
 if embedding1 is None or embedding2 is None:  
 return 0  
  
 vec1 = np.array(embedding1).flatten()  
 vec2 = np.array(embedding2).flatten()  
  
 similarity = np.dot(vec1, vec2) / (np.linalg.norm(vec1) \* np.linalg.norm(vec2))  
 return similarity  
 except Exception as e:  
 return 0  
  
 def detect\_faces(self, frame):  
 *"""Detect faces in the frame using MediaPipe"""* frame\_rgb = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)  
 results = self.face\_detection.process(frame\_rgb)  
  
 faces = []  
 if results.detections:  
 frame\_height, frame\_width, \_ = frame.shape  
 for detection in results.detections:  
 bbox = detection.location\_data.relative\_bounding\_box  
 x = int(bbox.xmin \* frame\_width)  
 y = int(bbox.ymin \* frame\_height)  
 w = int(bbox.width \* frame\_width)  
 h = int(bbox.height \* frame\_height)  
  
 x = max(0, x)  
 y = max(0, y)  
 w = min(w, frame\_width - x)  
 h = min(h, frame\_height - y)  
  
 faces.append((x, y, w, h, detection.score[0]))  
  
 return faces  
  
 def recognize\_face(self, image\_path):  
 *"""Recognize face in the image"""* frame\_embedding = self.\_get\_face\_embedding(image\_path)  
 if frame\_embedding is None:  
 return []  
  
 recognized\_faces = []  
 for name, known\_embedding in self.known\_embeddings.items():  
 similarity = self.\_compare\_embeddings(frame\_embedding, known\_embedding)  
 if similarity > self.recognition\_threshold:  
 confidence = similarity \* 100  
 recognized\_faces.append((name, confidence))  
  
 recognized\_faces.sort(key=lambda x: x[1], reverse=True)  
 return recognized\_faces  
  
 def draw\_face\_box(self, frame, x, y, w, h, name, confidence, detection\_score):  
 *"""Draw bounding box and labels on the face"""* cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)  
  
 label = f"{name} ({confidence:.1f}%) [Det: {detection\_score:.2f}]"  
  
 font = cv2.FONT\_HERSHEY\_SIMPLEX  
 font\_scale = 0.6  
 thickness = 2  
 text\_size = cv2.getTextSize(label, font, font\_scale, thickness)[0]  
  
 cv2.rectangle(frame,  
 (x, y - text\_size[1] - 10),  
 (x + text\_size[0], y),  
 (0, 255, 0),  
 cv2.FILLED)  
  
 cv2.putText(frame,  
 label,  
 (x, y - 5),  
 font,  
 font\_scale,  
 (0, 0, 0),  
 thickness)  
  
 def process\_image(self, input\_image\_path):  
 *"""Process a single image and perform face recognition"""* frame = cv2.imread(input\_image\_path)  
 if frame is None:  
 raise Exception(f"Could not read image {input\_image\_path}")  
  
 faces = self.detect\_faces(frame)  
 if not faces:  
 print("No faces detected in the image")  
 return frame  
  
 recognized\_faces = self.recognize\_face(input\_image\_path)  
  
 if recognized\_faces:  
 name, confidence = recognized\_faces[0]  
 for (x, y, w, h, detection\_score) in faces:  
 self.draw\_face\_box(frame, x, y, w, h, name, confidence, detection\_score)  
 else:  
 for (x, y, w, h, detection\_score) in faces:  
 cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)  
 cv2.putText(frame, f"Unknown [Det: {detection\_score:.2f}]",  
 (x, y - 5), cv2.FONT\_HERSHEY\_SIMPLEX, 0.6, (255, 0, 0), 3)  
  
 return frame  
  
  
def main():  
 root = tk.Tk()  
 app = FaceRecognitionGUI(root)  
 root.mainloop()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()