Real-Time Face Recognition System with Custom-Trained Deep Learning Models

Team Members:

- **Ahmed Kamal FathAllah**, **ID:** 17010210
- **George Seleim Abd-Allah Khaleel**, **ID:** 20010436

Department:

- **Computer Science and Engineering Department (CSED)**, **Alexandria University, supervisors:**
- **Dr. Salah Selim**, Email: **salahss9@yahoo.com**

Co-Advisor:

- **Dr. Magdy Abdel-Azeem**, Email: **magdy_aa@hotmail.com**

Project Details:

Graduation Project

This report provides a comprehensive overview of the facial recognition system. The system encompasses various components, including face detection, face recognition, and a user interface, all integrated into a cohesive application. The report details the models used, the project structure, installation procedures, usage examples, and underlying code implementations.

System Overview

The facial recognition system is developed using Python and integrates several libraries for deep learning, computer vision, and graphical user interface (GUI) development. Its core components leverage:

- **CustomTkinter**: Utilized for creating a robust, modern, and enhanced graphical user interface, featuring rich interactive elements and a tabbed navigation system.
- **PyTorch**: Serves as the deep learning framework for model training and inference, enabling the implementation of custom neural network architectures.
- **OpenCV**: Employed for critical computer vision tasks, including image capture, real-time video stream processing, pre-processing, and the seamless integration of various face detection algorithms.
- **SQLite**: Functions as a lightweight, embedded database solution for securely storing user information, including facial embeddings and access permissions, ensuring data persistence and integrity.
- **ESP32-CAM Integration**: The system supports image capture from network-connected IP cameras (e.g., ESP32-CAM) in addition to standard built-in webcams, providing significant flexibility in deployment scenarios.

The system incorporates substantial enhancements focused on user configurability and real-time operational feedback. Key improvements include the ability to dynamically switch between multiple face detection models, select different camera sources, and manage registered individuals directly through an intuitive, tab-based GUI. The primary face recognition model, based on a ResNet architecture with specific modifications, is robustly integrated to ensure high performance and adaptability in diverse real-world applications.

Features

The updated facial recognition system incorporates a comprehensive set of features designed for flexibility, accuracy, and user experience:

- **Real-time Face Detection and Recognition**: The system performs continuous face detection and recognition from live video streams, making it suitable for interactive applications such as access control or attendance monitoring.

- **Configurable Camera Sources**: Users can dynamically select between a **built-in webcam** and an **ESP32-CAM** (via HTTP stream) as the active video input source directly from the graphical interface.
- **Multiple Face Detection Model Support**: The application provides the flexibility to switch between different face detection algorithms:
    - **YuNet Detector**: Highly efficient and accurate, especially for varying image qualities.
    - **Haar Cascade Detector**: A classic, lightweight, and computationally less intensive option.
    - **Custom CNN Detector**: A custom-trained convolutional neural network for specialized face detection.
- **Custom-Trained Deep Learning Models**: The system utilizes custom-trained models for both face detection and recognition, optimized for specific performance characteristics and datasets.
- **User Registration and Authentication System**:
    - **Registration**: Allows for secure enrollment of new individuals by capturing and averaging multiple facial samples to create a robust biometric profile.
    - **Authentication**: Performs real-time verification of detected faces against the registered database to grant or deny access.
- **Access Control Management**: Configurable access permissions (allowed/denied) can be assigned to registered individuals, enabling policy-based access control.
- **Enhanced GUI Interface**: Built with CustomTkinter, the graphical user interface features intuitive **tabbed navigation**, separating functionalities into:
    - **Main Controls**: For core registration, login, and real-time operational feedback.
    - **Settings**: To configure camera sources and actively switch between face detection models.
    - **Manage Persons**: To view, update, and manage registered individuals in the database.
- **Queue-Based Embedding Averaging**: Enhances recognition robustness and accuracy by collecting and averaging multiple face embeddings over a short period, mitigating issues caused by momentary pose variations or environmental factors. The number of samples to average (`num_samples`) is configurable.
- **Thread-Safe Database Operations**: Ensures stable performance and data integrity by implementing thread-safe access mechanisms for the SQLite database, preventing concurrency issues.
- **Real-time Operational Feedback**: Provides immediate visual updates through dedicated GUI elements:

- A **status label** offers concise, real-time messages on ongoing processes.
- A **GUI textbox** serves as an in-app log, displaying detailed application results, registration confirmations, and login outcomes.
- The **recognition threshold** and **last match distance** are displayed in real-time, offering transparent insights into the authentication process.

Models

The system incorporates sophisticated deep learning models for both face detection and face recognition, offering a robust and configurable approach to biometric identification.

Face Detection

The system employs a flexible and configurable approach to face detection, allowing users to select from **multiple integrated models** based on their specific needs and environmental conditions. This flexibility ensures robustness across varying image qualities and computational resources.

- **YuNet Detector (Default and Recommended)**:
  - **Model**: `face_detection_yunet_2023mar.onnx`
  - **Integration**: Directly integrated via OpenCV's `cv2.FaceDetectorYN` class. YuNet is a highly efficient and accurate real-time face detector, particularly well-suited for scenarios with varying image quality, including those from ESP32-CAM streams.
  - **Key Features**:
    - **Input Size**: `320x320` pixels (input frames are resized to this dimension before processing).
    - **Confidence Threshold**: `0.9` (configurable minimum confidence score for a detection to be considered valid).
    - **Non-Maximum Suppression (NMS) Threshold**: `0.3` (filters out overlapping detections).
    - **Top-K**: `5000` (specifies the maximum number of detections to retain after NMS).
  - **Rationale for Preference**: This model was preferred over the custom-trained bounding box model (v5) for real-time application in lower-quality streams because the custom model, while effective on high-quality datasets like CelebA, showed less robustness with typical ESP32-CAM output.
- **Haar Cascade Detector**:
  - **Model**: Standard OpenCV Haar Cascade Classifier (`haarcascade_frontalface_default.xml`, often provided with OpenCV installations).
  - **Integration**: Implemented through OpenCV's `cv2.CascadeClassifier`.

    - **Characteristics**: A classic, computationally lightweight, and historically significant face detection method. While generally faster, it may be less accurate and prone to false positives compared to deep learning-based models, especially in complex lighting or pose variations. It serves as a readily available and efficient fallback option.
- **Custom CNN Detector**:
    - **Model**: `bbox_v5_randomly_augmented_epoch_3.pth` (or similar version from `models/bbox_models/v5/`)
    - **Architecture**: This is a custom-trained Convolutional Neural Network (CNN) designed for face detection, as detailed in the "FaceBBoxModel" architecture section of this report.
    - **Training**: Trained on the CelebA dataset, which contains over 200,000 images with precise bounding box annotations.
    - **Input/Output**: Takes images of size `224x224` pixels as input and outputs four coordinates (`x, y, width, height`) representing the detected face's bounding box.
    - **Usage**: While robust on clean, high-resolution images, its performance on lower-quality or unconventional camera feeds (like those from ESP32-CAM) may vary. It is provided as a configurable option for users who might prefer a custom-trained solution or for specific testing scenarios.

Face Recognition

The core face recognition model is based on a ResNet architecture with residual connections, referred to as ResArkSGD.

- **Architecture**: ArcFace-based architecture with residual connections, designed for robust feature learning.
- **Embedding Dimension**: 512-dimensional embedding vectors are generated as unique numerical representations of facial features.
- **Loss Function**: ArcFace loss, which enhances feature discrimination by applying an angular margin penalty during training.
- **Preprocessing**: Includes face alignment using detected facial landmarks to ensure consistent input to the model.
- **Training Data**: Trained on the VGGFace2-HQ Cropped dataset.
- **Testing Data**: Evaluated using the PINS Face Recognition Dataset.

- **Model Versions**: Several versions of this model exist, each with varying accuracy, such as an augmented version with 94% accuracy, and non-augmented versions achieving 88% and 95% accuracy.

Model Architectures

FaceBBoxModel (Bounding Box Detector)

The `FaceBBoxModel` is a custom-designed Convolutional Neural Network (CNN) responsible for detecting faces within an image and predicting their precise bounding box coordinates. The model's architecture is structured as a series of convolutional layers for feature extraction, followed by fully connected layers for bounding box regression.

- **Convolutional Layers**: The model begins with five convolutional layers, each playing a critical role in feature extraction. These layers are organized as follows:
    - **Conv1**: Applies 16 filters with a kernel size of 3×3, a stride of 1, and padding of 1 to the input image (3 color channels RGB). This layer extracts initial low-level features.
    - **Conv2**: Uses 32 filters, also with a 3×3 kernel, stride of 1, and padding of 1. It builds on the features learned in the previous layer.
    - **Conv3**: Employs 64 filters with the same kernel size, stride, and padding. It continues to refine feature maps.
    - **Conv4**: Utilizes 128 filters with the same kernel size, stride, and padding. This layer extracts more complex features.
    - **Conv5**: The fifth and final convolutional layer uses 256 filters with a 3×3 kernel, stride of 1, and padding of 1. This layer extracts the deepest features.
- **Batch Normalization**: Each convolutional layer is immediately followed by a batch normalization layer (`nn.BatchNorm2d`). This normalizes the output of the previous layer, leading to more stable and faster training.
- **Max Pooling**: After each convolutional and batch normalization layer, a max pooling layer (`nn.MaxPool2d`) with a kernel size of 2×2 and a stride of 2 is applied. This reduces spatial dimensions, decreases parameters, controls overfitting, and makes the model robust to small input variations.

- **Fully Connected Layers**: Following the convolutional layers, the feature maps are flattened into a vector and passed through three fully connected (linear) layers (`nn.Linear`) for bounding box prediction.
    - The first fully connected layer transforms the flattened feature map to a 512-dimensional vector, followed by a ReLU activation function.
    - A dropout layer (`nn.Dropout`) with a dropout rate of is applied to prevent overfitting by randomly setting a fraction of neurons to zero during training.
    - The second fully connected layer transforms the 512-dimensional vector to a 128-dimensional vector, also followed by a ReLU activation.
    - The third fully connected layer reduces the vector to 4 dimensions, corresponding to the bounding box coordinates (x,y,width,height).
- **Output**: The model outputs the bounding box coordinates, which are then used to crop the face from the original image for further processing by the face recognition model.

Enhanced FaceRecognitionModel (Face Recognition)

The `EnhancedFaceRecognitionModel` is a sophisticated deep learning model designed for generating high-quality embeddings for face recognition. This model employs a deep architecture comprising initial convolutional layers, followed by multiple residual blocks and deep residual blocks, and finally fully connected layers to produce the face embeddings.

- **Initial Convolutional Layers**:
    - The model begins with a sequence of initial convolutional layers (`self.conv1`). This sequence includes a 2D convolutional layer (`nn.Conv2d`) with 64 filters, a kernel size of 7×7, a stride of 2, and padding of 3. This layer extracts initial, low-level features from the 3-channel (RGB) input image.
    - The output is then passed through a batch normalization layer (`nn.BatchNorm2d`) with 64 channels for normalization and faster training.
    - A ReLU (Rectified Linear Unit) activation function introduces non-linearity.
    - Finally, a max pooling layer (`nn.MaxPool2d`) with a kernel size of 3×3, a stride of 2, and padding of 1 downsamples feature maps, reducing spatial dimensions and computational load while retaining important information.

- **Deep Residual Layers**: The core of the model consists of four distinct sets of residual layers:
    - **Layer 1**: (`self.layer1`) Comprises 3 standard residual blocks, each with 64 input and output channels, and a stride of 1. These blocks learn feature mappings and incorporate residual connections to mitigate the vanishing gradient problem.
    - **Layer 2**: (`self.layer2`) Consists of 4 standard residual blocks, each with 64 input channels and 128 output channels, with a stride of 2 for the first block.
    - **Layer 3**: (`self.layer3`) Contains 6 standard residual blocks, each with 128 input channels and 256 output channels, with a stride of 2 for the first block.
    - **Layer 4**: (`self.layer4`) Is made up of 3 deep residual blocks, each with 256 input channels and 512 output channels, with a stride of 2 for the first block. This deep residual block differs from standard residual blocks by adding an extra convolutional layer, increasing the model's capacity to learn complex patterns.
- **Residual Blocks**:
    - **Standard Residual Block (`ResidualBlock`)**: Contains two convolutional layers (`nn.Conv2d`) with a kernel size of 3, padding of 1, and batch normalization (`nn.BatchNorm2d`) after each convolution. ReLU activation is applied after the first convolution. A shortcut connection is added to the output of the second convolutional layer, either passing the input directly or using a 1×1 convolution with batch normalization if feature map dimensions differ.
    - **Deep Residual Block (`DeepResidualBlock`)**: Similar to the standard residual block but includes an additional convolutional layer (`nn.Conv2d`) and batch normalization layer, allowing for the learning of more complex features.
- **Fully Connected Layers**: After the deep residual layers, feature maps are flattened and passed through two fully connected layers (`self.fc_layers`).
    - The first fully connected layer (`nn.Linear`) transforms the flattened feature map to a 1024-dimensional vector, followed by batch normalization (`nn.BatchNorm1d`) and a ReLU activation.
    - The second fully connected layer (`nn.Linear`) maps the 1024-dimensional vector to a 512-dimensional embedding vector, also followed by batch normalization.

- **Output**: The model outputs a 512-dimensional normalized embedding vector that represents the unique facial features of the input. This embedding is then used to determine the similarity between different faces.
- **ArcFace Loss**: The model is trained using an ArcFace loss function (`ArcFaceLoss`), specifically designed to improve the discriminative power of face embeddings. ArcFace loss applies an angular margin penalty to the target logit, enhancing feature discrimination. During training, the ArcFace loss uses a scale and margin that typically evolve (e.g., s=30,m=0.5 for early epochs, increasing to s=45,m=0.7 for later epochs).
- **Weight Initialization**: The model's weights are initialized using Kaiming Normal initialization for convolutional layers, Xavier Normal initialization for linear layers, and batch normalization layers are initialized to 1 for weights and 0 for biases.

Project Structure

The project is organized into several Python files and directories, reflecting a modular design for clarity and maintainability:

```
Graduation_Project/
├── app_V2.py              # Main application with enhanced features
and GUI
├── app_V1.py              # Previous main application version
(retained for reference)
├── embeddings.py          # Face recognition model implementation
and embedding generation
├── bounding_box.py        # Custom CNN face detection model
implementation (and Haar Cascade integration)
├── bounding_box_yunet.py  # YuNet face detector implementation
└── models/
    ├── bbox_models/       # Directory containing various face
detection models
        ├── v5/            # Custom trained models (e.g.,
bbox_v5_randomly_augmented_epoch_3.pth)
        └── YuNet/         # YuNet face detector models (e.g.,
face_detection_yunet_2023mar.onnx)
    └── resarksgd/         # Face recognition models (e.g.,
resarksgdaug94.pth, resarksgd95.pth)
```

Installation

The installation process involves the following steps:

1. **Cloning the Repository**: The project is cloned from a GitHub repository using the following command:

```
git clone https://github.com/AhmedKamal75/Graduation_Project.git cd
Graduation_Project
```

2. **Creating and Activating a Virtual Environment**: A virtual environment is created and activated to manage the project's dependencies:

```
python3 -m venv .venv # Using .venv is a common convention source
.venv/bin/activate # Linux/macOS
.venv\Scripts\activate # Windows
```

3. **Installing Dependencies**: The required Python packages are installed using pip:

```
pip install -r requirements.txt
```

4. **Downloading the Pre-trained Models**: Download the necessary pre-trained models and place them in their respective directories:

    Face Recognition Model (e.g., `resarksgdaug94.pth` or `resarksgd95.pth`):

- Download from: Kaggle Model - 95% accuracy or -94% accuracy.
- Place the downloaded `.pth` file in the `models/resarksgd/` directory.

  YuNet Face Detector Model (`face_detection_yunet_2023mar.onnelx`):

- Download if not already present. Ensure it is in `models/bbox_models/YuNet/`.

  Custom CNN Face Detector Model (e.g., `bbox_v5_randomly_augmented_epoch_3.pth`):

- Download if you intend to use this model. Ensure it is in `models/bbox_models/v5/.`

5. **No Manual Path Updates Needed**: Model paths are specified at the top of the `if __name__ == "__main__"` block in `app_V2.py`.. Ensure the downloaded model files exist at these specified paths before running the application.

Usage

The application provides a comprehensive Graphical User Interface (GUI) for easy interaction, organized into distinct tabs: "Main Controls," "Settings," and "Manage Persons."

1. Run the main application

```
python app_V2.py
```

2. The GUI will launch, displaying a live video feed on the left and a control panel with tabbed navigation on the right.

Registration Process (Main Control Tab)

1. Navigate to the "Main Controls" tab.
2. Locate the "Registration" section.
3. Enter the person's `Name` in the provided input field.
4. Optionally, check or uncheck the "Allowed Access" checkbox to define initial access permissions.
5. Click the "Register Person" button.
6. The system will initiate automatic capture of multiple facial samples (default: 5 samples, configurable by modifying `num_samples` in `app_V2.py`). During this process, ensure your face is visible in the camera feed and held as still as possible for optimal sample collection.
7. The "Application Results & Logs" area will provide real-time status updates and a final verification message indicating the success or failure of the registration.

Authentication Flow (Main Controls Tab)

1. Navigate to the "Main Controls" tab.
2. Stand clearly in front of the active camera.
3. Click the "Login Person" button in the "Login" section.
4. The system will capture facial samples, extract embeddings, and compare them against the registered database.
5. **Real-time Recognition Feedback**:
    - The "Application Results & Logs" area will display detailed recognition results, including the matched person's name, access status, and the

precise `Match Distance` (cosine distance) between the detected face's embedding and the best match in the database.
    - The `Last Match Distance` label will update dynamically.
    - A status label will provide quick visual feedback on login success or failure.
6. **Adjusting Recognition Sensitivity**:
    - Use the "Recognition Threshold" slider in the "Recognition Threshold" section to adjust the system's sensitivity.
    - **Lower values** (e.g., 0.1) require higher similarity for a match, leading to stricter authentication.
    - **Higher values** (e.g., 0.4) are more lenient, allowing for matches with greater facial variation. Experiment with this slider to find the optimal balance for your environment.

Setting Management (Setting Tab)

1. Navigate to the "Settings" tab.
2. **Camera Settings**:
    - In the "Camera Settings" section, select your desired "Camera Type" from the dropdown menu: "Built-in Cam" (for your local webcam) or "ESP32 Cam" (for an IP camera).
    - If "ESP32 Cam" is selected, enter its stream URL (e.g., `http://192.168.1.5/cam-hi.jpg`) in the "ESP32 Cam URL" entry field. This field will be disabled if "Built-in Cam" is selected.
    - Click "Apply Camera Settings" to initiate the camera switch. The video feed will restart with the new source.
3. **Face Detection Model**:
    - In the "Face Detection Model" section, select your preferred "Model Type" from the dropdown: "YuNet Detector", "Haar Cascade Detector", or "Custom CNN Detector".
    - Click "Apply Model Settings" to switch the active face detection algorithm. The change will be applied instantly, and subsequent face detections will use the newly selected model.

Manage Persons (Manage Persons Tab)

1. Navigate to the "Manage Persons" tab.
2. This tab displays a comprehensive list of all registered persons, including their unique `ID`, `Name`, `Allowed` access status, and `Registered At` timestamp.

3. **Deleting Persons**: To remove a person from the database, click the "Delete" button located next to their entry in the list. A confirmation dialog will appear before deletion.
4. **Refreshing the List**: If the list does not automatically update after a registration or deletion, click the "Refresh Person List" button to manually reload the data from the database.

Code Implementation

GUI Elements (`app_V2.py`)

The application's graphical user interface, primarily managed within `app_V2.py`, is built using CustomTkinter, offering a modern and interactive experience. It features a main window structured with a video feed on the left and a dynamic control panel on the right, which is organized into a `CTkTabview` for logical separation of functionalities.

- **Main Application Structure**:
  - `self.root`: The main `CTk` window, configured for optimal layout using `grid` for video feed and control panel.
  - `self.video_frame` & `self.video_label`: Responsible for displaying the live camera feed. `_update_frame_on_gui()` dynamically resizes the displayed frame to fit the label, ensuring responsiveness across different window sizes.
  - `self.control_panel_frame`: Houses the `CTkTabview` which contains all interaction elements.
- **`CTkTabview` - Tabbed Navigation**:
  - `self.tab_view`: The central component for organizing UI elements, allowing users to switch between "Main Controls," "Settings," and "Manage Persons" tabs.
- **"Main Controls" Tab (`self.main_tab`)**:
  - **Registration Section (`self.reg_frame`)**: Includes a `CTkEntry` for `name_entry` and a `CTkCheckBox` for `is_allowed_var`, along with the `register_btn` (`command=self.register_person`).
  - **Login Section (`self.login_frame`)**: Contains the `login_btn` (`command=self.login_person`).

    - **Recognition Threshold Section (`self.threshold_frame`)**: Features `self.threshold_slider` (controlling `self.threshold`), `self.threshold_label` for current value display, and `self.min_distance_label` to show the closest match distance from the last login.
    - **Application Results & Logs Section (`self.results_logs_section_frame`)**:
        - `self.status_label`: Provides immediate, concise feedback to the user regarding ongoing operations (e.g., "Connecting to camera...", "Face detected").
        - `self.results_text`: A `CTkTextbox` that displays detailed application logs, registration confirmations, and login results.
        - `_setup_gui_logger()`: A custom logging handler that redirects `INFO` and higher-level log messages directly to `self.results_text`, ensuring in-app visibility of system activity.
- **"Settings" Tab (`self.settings_tab`)**:
    - **Camera Settings Frame (`self.camera_settings_frame`)**:
        - `self.camera_type_optionmenu`: A `CTkOptionMenu` allowing selection between "Built-in Cam" and "ESP32 Cam" (`command=self.on_camera_type_selected`).
        - `self.esp32_url_entry`: A `CTkEntry` for inputting the ESP32-CAM stream URL, which is dynamically enabled/disabled by `_update_esp32_url_entry_state()`.
        - `self.apply_camera_settings_btn`: Triggers `self.apply_camera_settings()` to switch camera sources.
    - **Face Detection Model Frame (`self.bbox_model_settings_frame`)**:
        - `self.bbox_model_type_optionmenu`: A `CTkOptionMenu` for selecting the face detection model ("YuNet Detector", "Haar Cascade Detector", "Custom CNN Detector") (`command=self.on_bbox_model_type_selected`).
        - `self.apply_bbox_model_btn`: Triggers `self.apply_bbox_model_settings()` to load the selected face detector.
- **"Manage Persons" Tab (`self.manage_persons_tab`)**:
    - `self.person_list_canvas_container`, `self.person_list_canvas`, `self.person_list_inner_frame`:

These components, along with scrollbars (`self.person_list_v_scrollbar`, `self.person_list_h_scrollbar`), create a dynamically scrollable list view for registered persons.

  - `self.person_row_frames`: Stores references to individual `CTkFrame` widgets, each representing a registered person.
  - `_load_persons_to_gui()`: Populates the list by fetching data from `PersonDatabase`, creating a row for each person, and including a "Delete" button. It also manages canvas scroll region updates (`_on_inner_frame_configure`).
  - `_handle_delete_person_from_list(person_id)`: Callback for the delete buttons, removing the specified person from the database and refreshing the GUI list.
  - `self.refresh_persons_btn`: Manually triggers `_load_persons_to_gui()`.
- **Core GUI Interaction Functions**:
  - `_set_interaction_buttons_state(state)`: A crucial helper function that universally enables or disables various GUI buttons (register, login, apply settings, delete) during long-running operations (like capturing samples) to prevent concurrent actions and ensure thread safety.
  - `_start_camera_capture()`: Initializes the selected camera (built-in or ESP32) and starts `self.capture_thread` which runs `_capture_frames_loop()`.
  - `_stop_camera_capture()`: Gracefully stops the camera capture thread and releases OpenCV resources.
  - `_capture_frames_loop()`: Runs in a separate thread, continuously captures frames, calls `extract_face()` for processing, and updates `self.current_frame` for GUI display. Includes robust error handling and re-acquisition logic for camera streams.
  - `_update_frame_on_gui()`: Scheduled by `self.root.after()`, it converts `self.current_frame` to `CTkImage` and updates `self.video_label`, providing the live video feed.
  - `register_person()` and `login_person()`: These methods initiate their respective processes in separate threads (`threading.Thread`) to keep the GUI responsive, providing detailed status updates via `self.status_label` and `self.results_text`.

    - `exit_program()`: Handles a clean shutdown, ensuring all threads are joined and resources released.

Face Recognition Model (**`embeddings.py`**)

The face recognition model's implementation, primarily within `embeddings.py`, focuses on efficient embedding generation and preprocessing. The `EmbeddingPredictor` class provides:

- **`preprocess_face(face)`**: Responsible for preparing a cropped face image for input into the neural network. This involves a sequence of transformations: resizing to 224×224 pixels, converting color channels from BGR to RGB, transposing dimensions from HWC (Height, Width, Channel) to CHW (Channel, Height, Width), and normalizing pixel values to the range .
- **`generate_embedding(face)`**: Performs a forward pass through the loaded PyTorch model to compute a 512-dimensional embedding vector. This vector is a unique numerical representation of the facial features, designed to capture identity-specific information.
- **`capture_multiple_embeddings(num_samples)`**: This function is crucial for enhancing the robustness of the recognition system by collecting and averaging multiple embeddings. In `app_V2.py`, this function has been significantly improved to provide clear user feedback:
    - It now visually updates the `self.status_label` in the GUI, indicating "Detecting face and collecting samples..." during the process.
    - The function attempts an initial brief scan (e.g., up to 2 seconds) to confirm a face is detected before proceeding with full sample collection, providing early feedback if no face is found in the initial period.
    - A timeout mechanism (e.g., 5 seconds) ensures that the application doesn't get stuck indefinitely if the target `num_samples` cannot be fully collected. If a timeout occurs but *some* samples have been acquired, it proceeds with the available samples; otherwise, it reports a failure.
    - The embedding queue (`self.embedding_queue`) is explicitly cleared at the start of each new capture operation to ensure only fresh, relevant samples are used for averaging, preventing stale data.
    - Upon successful collection (either full or partial due to timeout), the `status_label` confirms "Samples collected successfully."

Face Detection Models (**`bounding_box.py`**, **`bounding_box_yunet.py`**)

The system integrates three distinct face detection models, managed dynamically by `app_V2.py` based on user selection:

- **`YuNetFaceDetector`** (**`bounding_box_yunet.py`**):
    - **Model Loading**: Loads the YuNet ONNX model efficiently using `cv2.FaceDetectorYN.create()`.
    - **`predict_bounding_box(frame)`**: Detects faces within the input `frame` using the loaded YuNet model and returns a list of bounding box coordinates (often including facial landmarks).
- **`BoundingBoxDetector`** (**`bounding_box.py`**):
    - **Model Loading**: Can be initialized with a path to a Haar Cascade XML file (e.g., `cv2.data.haarcascades + 'haarcascade_frontalface_default.xml'`).
    - **`predict_bounding_box(frame)`**: Utilizes OpenCV's `detectMultiScale` method for Haar Cascade-based face detection, returning a list of detected bounding boxes.
- **`BoundingBoxPredictor`** (**`bounding_box.py`**):
    - This class represents the custom CNN face detection model.
    - **Model Loading**: Loads a PyTorch-trained CNN model (e.g., `bbox_v5_randomly_augmented_epoch_3.pth`).
    - **`predict_bounding_box(frame)`**: Performs inference using the custom PyTorch CNN model to predict bounding box coordinates for faces in the input `frame`.

All bounding box predictor classes include a `crop_face(frame, bbox)` method to precisely extract the face region based on the detected bounding box, which is then passed to the embedding predictor.

Database Operations(**`PersonDatabase`** in **`app_V2.py`**)

The `PersonDatabase` class, nested within `app_V2.py`, handles all interactions with the SQLite database (`face_database.db`). It is designed to be thread-safe, utilizing a `threading.Lock()` to prevent concurrency issues during database access from multiple threads (e.g., GUI thread, capture thread).

- **`__init__(db_file='face_database.db')`**: Initializes the SQLite connection and ensures the `persons` table exists.
- **`_create_table()`**: Creates the `persons` table with columns for `id` (timestamp-based PRIMARY KEY), `name`, `embedding` (stored as BLOB), `is_allowed` (INTEGER for boolean), and `created_at` (TEXT).
- **`add_person(embedding, name, is_allowed)`**: Inserts a new person's data into the database. A unique `person_id` is generated using a timestamp (`datetime.now().strftime("%Y%m%d_%H%M%S")`). The embedding (NumPy array) is converted to a binary format (`sqlite3.Binary(embedding.tobytes())`) for storage.
- **`find_match(embedding, threshold=0.3)`**: Queries the database to find the closest matching person for a given input `embedding`. It iterates through stored embeddings, calculates cosine similarity (and converts to cosine distance `1 - similarity`), and returns the person's ID, data, and the minimum distance if a match is found within the specified `threshold`. It also logs the `min_distance` for debugging and display on the GUI.
- **`get_all_persons()`**: Retrieves all registered persons from the database, including their `id`, `name`, `is_allowed` status, and `created_at` timestamp. This data is used to populate the "Manage Persons" tab.
- **`delete_person(person_id)`**: Deletes a person's entry from the database based on their unique `person_id`. It returns `True` if a row was successfully deleted, `False` otherwise.

Face Recognition Model (`arkface_residual_connections_part_2.ipynb`)

The face recognition model is implemented with PyTorch, incorporating several advanced techniques and layers:

- **Data Loading and Preprocessing:** Dataset such as VGGFace2 is used for training the model, Pins is used for testing the model. The datasets are preprocessed using transformations that include resizing, normalization, and converting to tensors. Albumentations is used for applying image augmentations.
- **Dataset Classes:** Custom dataset classes (`PinsDataset` and `VGGFace2Dataset`) are implemented to manage image loading and labeling. These classes handle

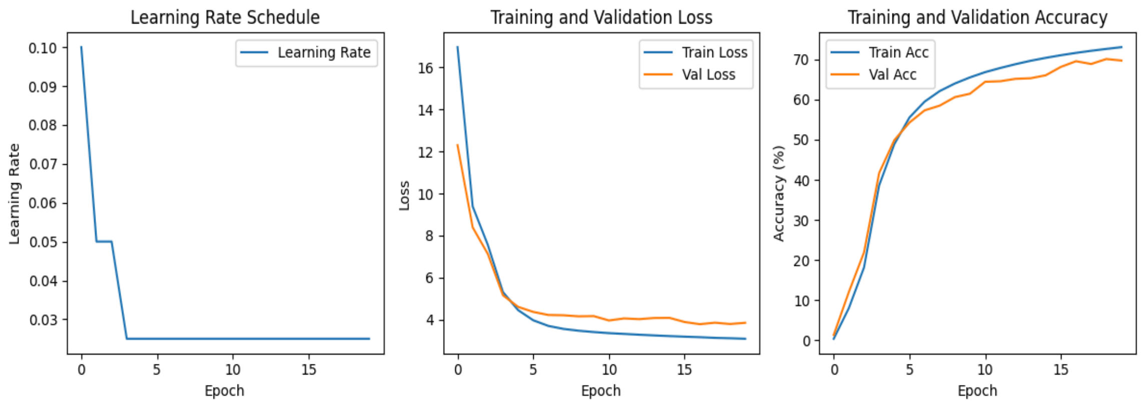
the retrieval of images and their corresponding labels and also allow for the creation of subsets.

- **ArcFace Loss:** The `ArcFaceLoss` class implements the ArcFace loss function, which is designed to enhance feature discrimination during training. It does this by adding an angular margin penalty to the target logit.
- **Residual Blocks:** The model uses residual blocks (`ResidualBlock` and `DeepResidualBlock`) to build deep convolutional layers. These blocks help with training deep neural networks.
- **Model Architecture:** The `EnhancedFaceRecognitionModel` class defines the main structure of the face recognition network. It uses a combination of convolutional layers, residual blocks, and fully connected layers. The model includes initial convolutional layers, deep residual layers, simplified fully connected layers, and the ArcFace loss.
- **Model Training Process for `EnhancedFaceRecognitionModel`:**
- The training of the `EnhancedFaceRecognitionModel` is orchestrated within the `train_model` function. This process encompasses data handling, optimization, and performance evaluation, and includes detailed steps as below:
- **Data Loading**:
  - The training process starts by loading the training and validation datasets using PyTorch's `DataLoader` class.
  - The datasets are split into batches of a defined `batch_size`. The batch size is set to 64 in the provided configuration.
  - The data loaders also handle shuffling of the data to ensure that the model is exposed to a diverse set of samples during each epoch.
  - The data loaders also use `num_workers` parameter to load the data in parallel. `pin_memory=True` is used when using a CUDA device to load the data into the GPU memory more efficiently. The number of workers is 0 when using CPU and 4 when using a CUDA device.
- **Optimizer and Loss Function Setup:**
  - The model employs the **Stochastic Gradient Descent (SGD) optimizer** with **Nesterov momentum**. This optimizer is selected for its effectiveness in training deep neural networks.
  - The optimizer parameters include a learning rate (`learning_rate`), momentum (`momentum`), and weight decay (`weight_decay`).
  - The learning rate is initially set to 0.1 in the provided configuration.
  - The momentum is set to 0.9 and weight decay is set to 1e-4.
  - A **MultiStepLR scheduler** is used to decay the learning rate during training. This scheduler decreases the learning rate by a factor (`gamma`) at

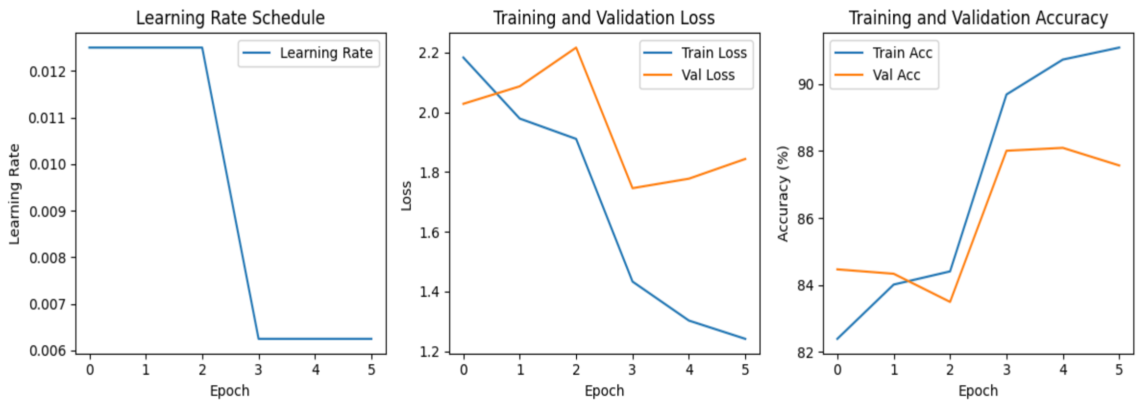
predefined epochs (`milestones`). The learning rate decreases by a factor of 0.5 at epochs.
    - A **CrossEntropyLoss** is used as the loss function for training the model. This is a standard loss function for multi-class classification problems.
- **Training Loop:**
    - The training process is carried out for a specified number of epochs.
    - In each epoch, the model iterates through the training dataset in batches.
    - During the training phase, the model is set to training mode (`model.train()`), and for each batch, the following steps are executed:
        - The input images and labels are transferred to the appropriate device (CPU or GPU).
        - The model performs a forward pass, producing the outputs and embeddings.
        - The loss is calculated using the `CrossEntropyLoss`.
        - The gradients are calculated and backpropagated.
        - The model parameters are updated using the optimizer with clipped gradients to prevent exploding gradients.
        - The training loss and accuracy for the batch are recorded.
    - After each epoch, the average training loss and accuracy are calculated and recorded.
- **Validation Phase:**
    - After the training phase in each epoch, the model's performance is evaluated on a validation dataset.
    - The model is set to evaluation mode (`model.eval()`), which turns off operations like dropout and batch normalization that are only used during training.
    - The model iterates through the validation dataset, and for each batch, it performs a forward pass, calculates the loss, and records the validation loss and accuracy.
    - The gradients are not computed during the validation phase to save memory and time.
    - After each epoch, the average validation loss and accuracy are calculated and recorded.
- **Learning Rate Scheduling:**
    - The learning rate scheduler is stepped after each epoch, adjusting the learning rate according to the specified milestones.
    - The current learning rate is printed at the beginning of each epoch.
- **Model Saving:**

  - The model with the best validation loss is saved during training.

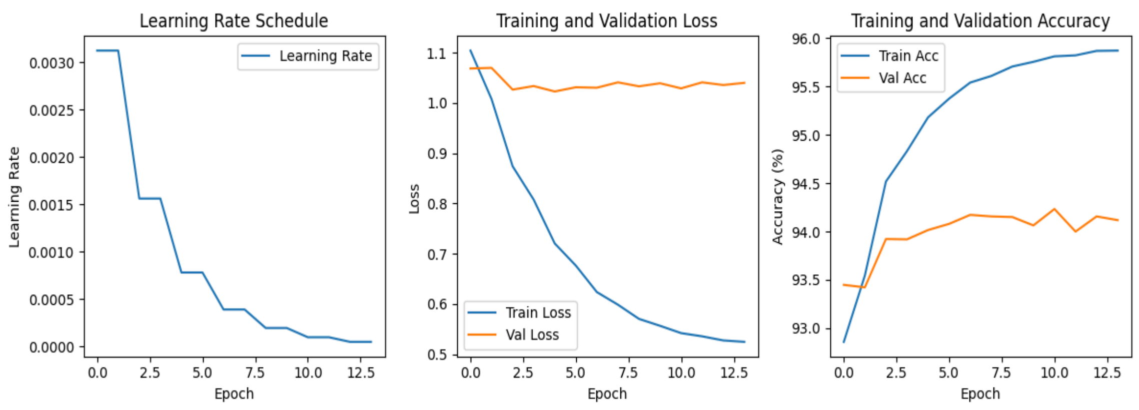
- **Performance Monitoring:**
  - The training and validation losses and accuracies are tracked across epochs using python lists and plotted at the end of training to visualize the training process.
  - The learning rate schedule is also tracked and plotted.
- These plots help assess the convergence and identify potential problems like overfitting.
- **Training Results**
  - 5 sets of training results are done, and the result is plotted:
    - The first result after 20 epochs of training:



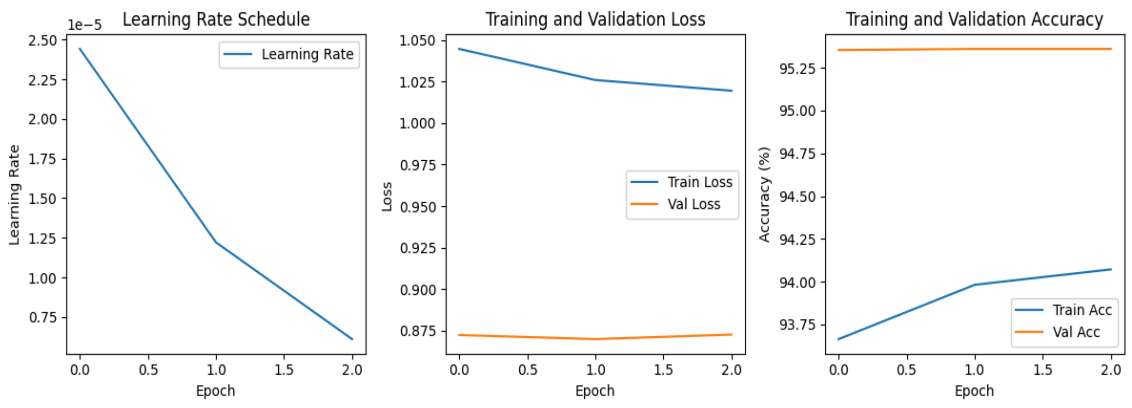
    - The second result after 6 epochs of training:



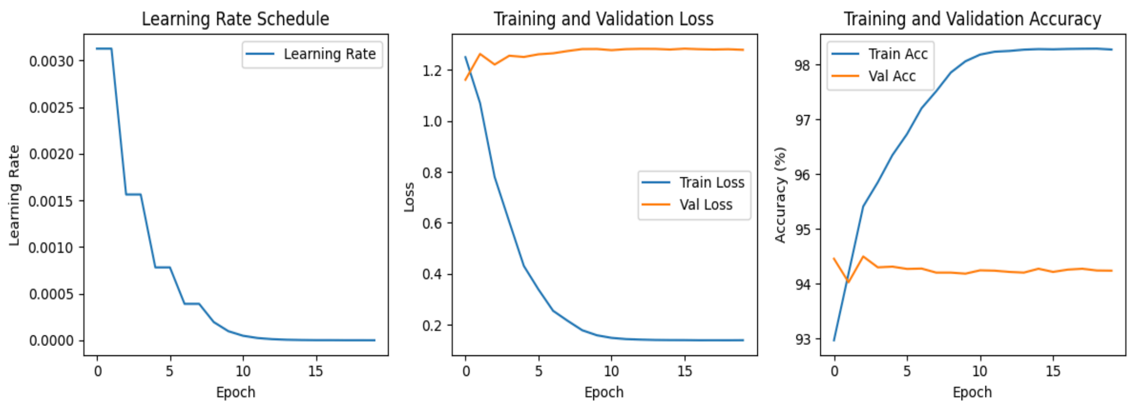
    - The third result after 14 epochs of training:



- The fourth result after 3 epochs of training:



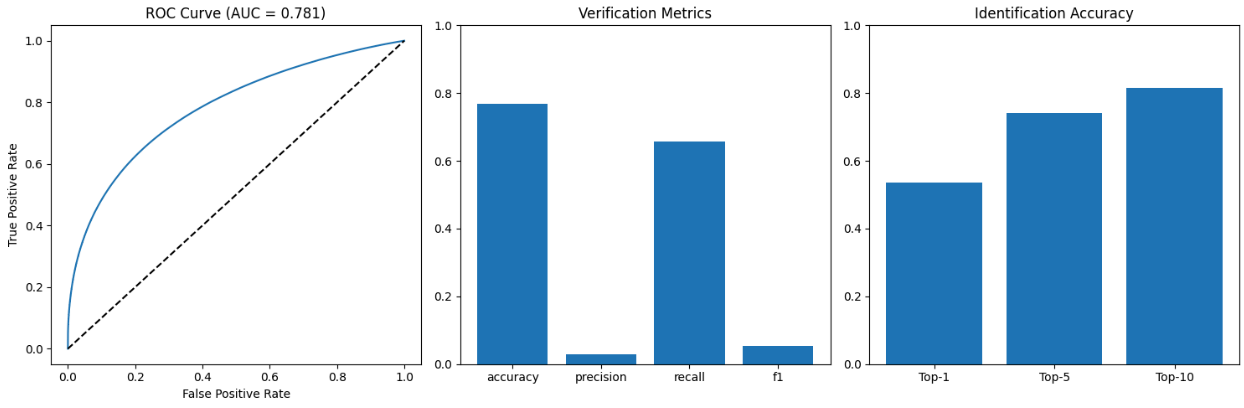
- The fifth result after 20 epochs of training with heavy augmentations:

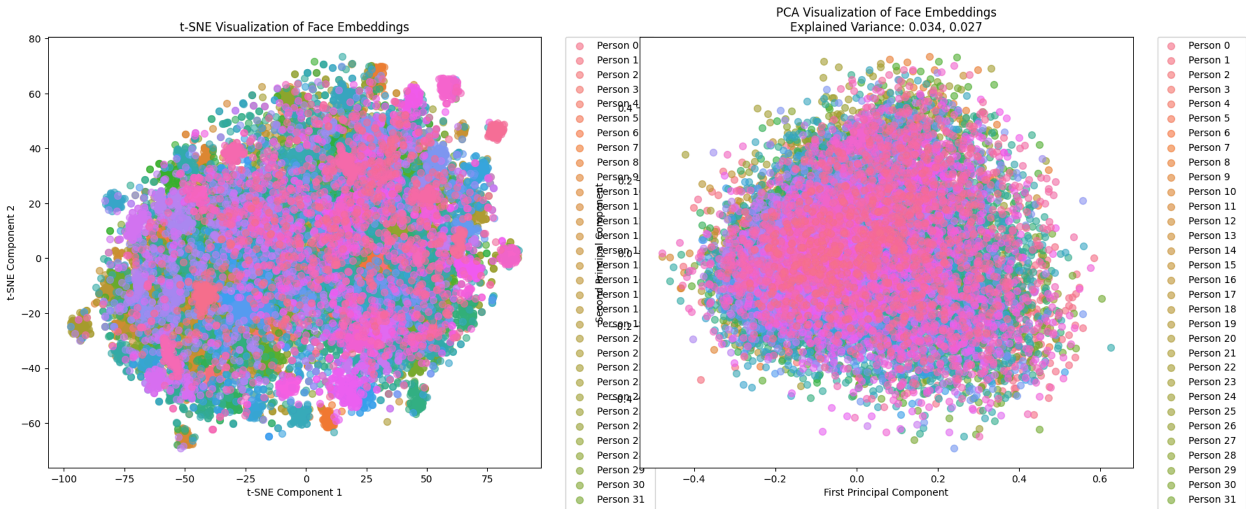


- These results show a trend that as the training goes on the training and validation loss decreases and the training and validation accuracy increases.

- **Testing Results:**
  - The testing results are obtained from running the trained model of 95% accuracy on the entire PINS dataset. The dataset has 105 people and 17534 images. These tests include:

- **Verification Results:**

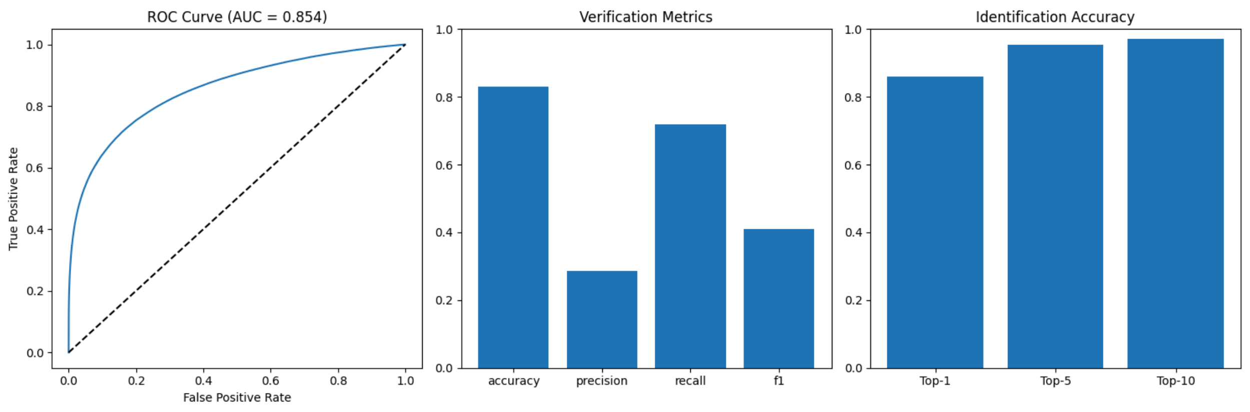


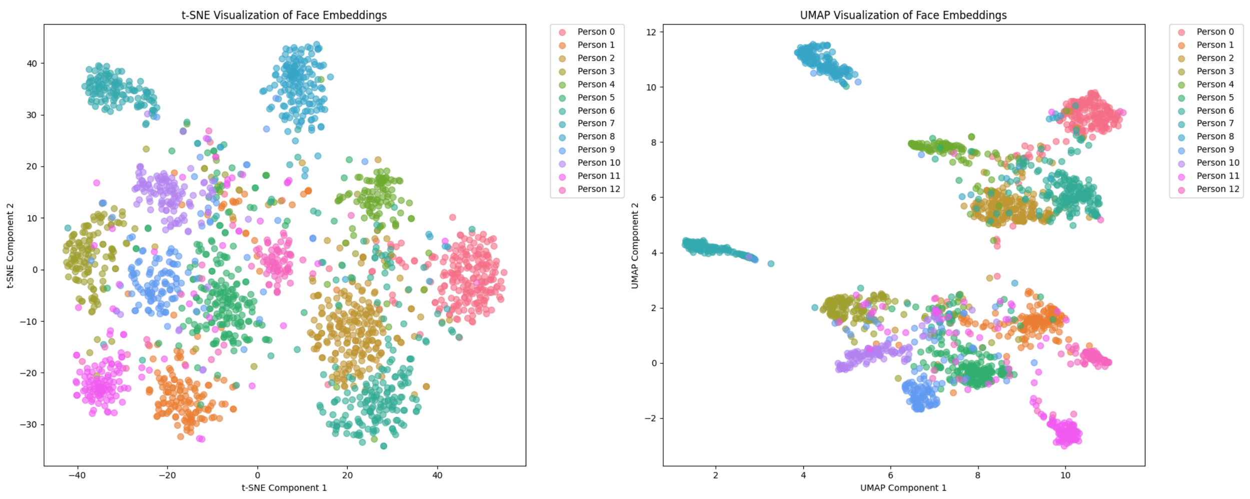


  - The verification results include **accuracy: 0.7825, precision: 0.0292, recall: 0.6485, f1: 0.0558, auc: 0.7837, best_threshold: 0.1122**
  - These results are calculated by comparing embeddings from images of the same person with embeddings from images of different people.
  - The **accuracy** metric provides a general measure of the verification performance, showing that the model can classify the images correctly 78% of the time, but when looking at the other metric there seems to be a big imbalance between **precision and recall**.
  - The **precision** metric is at 0.0292 indicating that out of all the pairs classified as being of the same person, only 2.92% are the same person. This is very low and indicates many false positives.
  - The **recall** metric is at 0.6485, indicating that the model can identify about 65% of the images that are of the same person.
  - The **F1 score** is only 0.0558, which reflects the imbalance between precision and recall.

        - The **AUC** which is the area under the ROC curve, is 0.7837 which is better than random performance.
        - The best threshold was found to be 0.1122 which is the optimal threshold when using the ROC curve.
    - **Identification Results:**
        - **Top 1 Accuracy: 0.5450.**
        - **Top 5 Accuracy: 0.7459.**
        - **Top 10 Accuracy: 0.8223**
    - **Optimal Threshold Results:**
        - The model finds the best threshold that maximizes precision while maintaining an acceptable recall (at least 0.2 in this case). The optimal threshold was found to be 0.31, at which the precision is 0.153, recall is 0.204 and f1 is 0.175.

    - **Unconditional Optimal Threshold Results:**
        - The model also finds the optimal threshold that maximizes precision unconditionally. The unconditional optimal threshold was found to be 0.99, at which the precision is 1.0, recall is 0.006 and f1 is 0.01.
- Another test was performed on a subset of the PINS dataset consisting of **2000 images** from **13 different identities**.





- Verification Results:
  - Accuracy: 0.8292
  - Precision: 0.2865
  - Recall: 0.7179
  - F1 Score: 0.4095
  - AUC: 0.8540
  - Best Threshold: 0.1263
- Identification Results:
  measure how often the **correct identity** appears within the **top X most likely predictions** made by the model
  - Top-1 Accuracy: 0.8610
  - Top-5 Accuracy: 0.9535
  - Top-10 Accuracy: 0.9720
- Optimal Threshold Results (for max precision with recall >= 0.2):
  - Optimal Threshold: 0.4
  - Precision: 0.9121
  - Recall: 0.2111

    - **F1 Score: 0.3428**
- **Unconditional Optimal Threshold Results (for max precision):**
    - **Unconditional Optimal Threshold: 0.99**
    - **Unconditional Precision: 1.0000**
    - **Unconditional Recall: 0.0061**
    - **Unconditional F1 Score: 0.0121**

Comparison with Full PINS Dataset Results

- **Verification:**
    - The subset shows a significantly higher accuracy (0.8292 vs 0.7825), precision (0.2865 vs 0.0292) and F1 score (0.4095 vs 0.0558), and AUC (0.8540 vs 0.7837) compared to the full dataset test. The subset and full dataset recall values are also different at (0.7179 vs 0.6485).
    - **This indicates that the model performs much better on the subset for face verification, with fewer false positives and better overall classification performance.**
- **Identification:**
    - The subset shows a significantly higher accuracy across the board for identification results; (top 1: 0.8610 vs 0.5450, top 5: 0.9535 vs 0.7459, top 10: 0.9720 vs 0.8223).
    - **This shows that the model is much more accurate at identifying faces in the top-k positions in the subset compared to the full dataset**.
- **Optimal Threshold:**
    - The subset produces a much higher precision with a good F1 score when finding the optimal threshold on the condition that recall is above 0.2 (precision: 0.9121 vs 0.153, F1: 0.3428 vs 0.175). However, the recall is much lower (0.2111 vs 0.204) with the subset test, likely as a consequence of the optimization parameters.

Opinion on the Differences

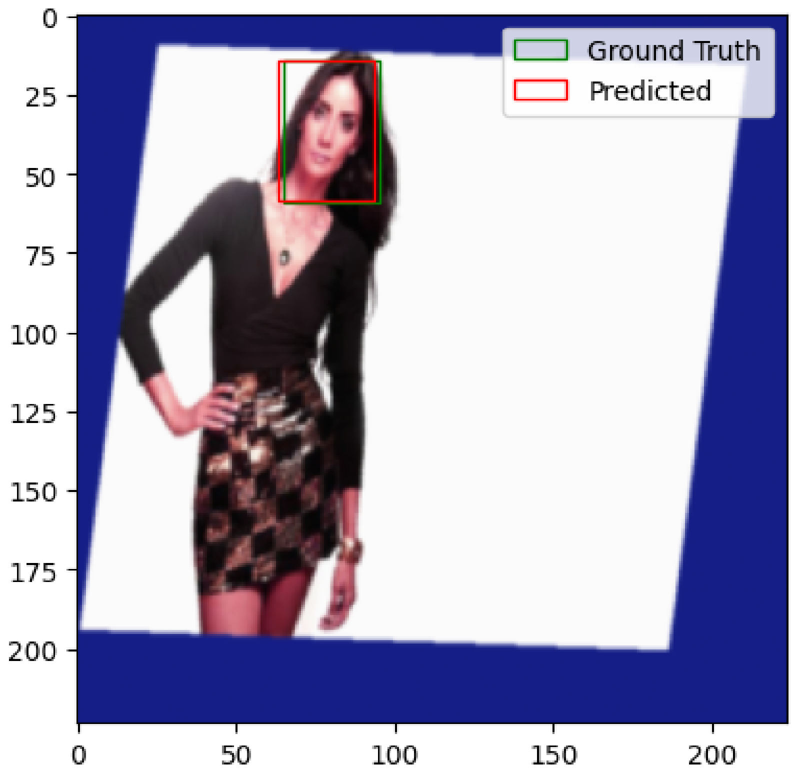
- **Dataset Complexity:** The most significant factor contributing to the difference is the complexity of the dataset.
    - The full PINS dataset contains 105 different identities and over 17,000 images, while the subset has only 13 identities and 2000 images.
    - **The reduced number of classes (identities) and samples in the subset makes the face recognition and verification tasks much easier for the model to handle compared to the full dataset.**

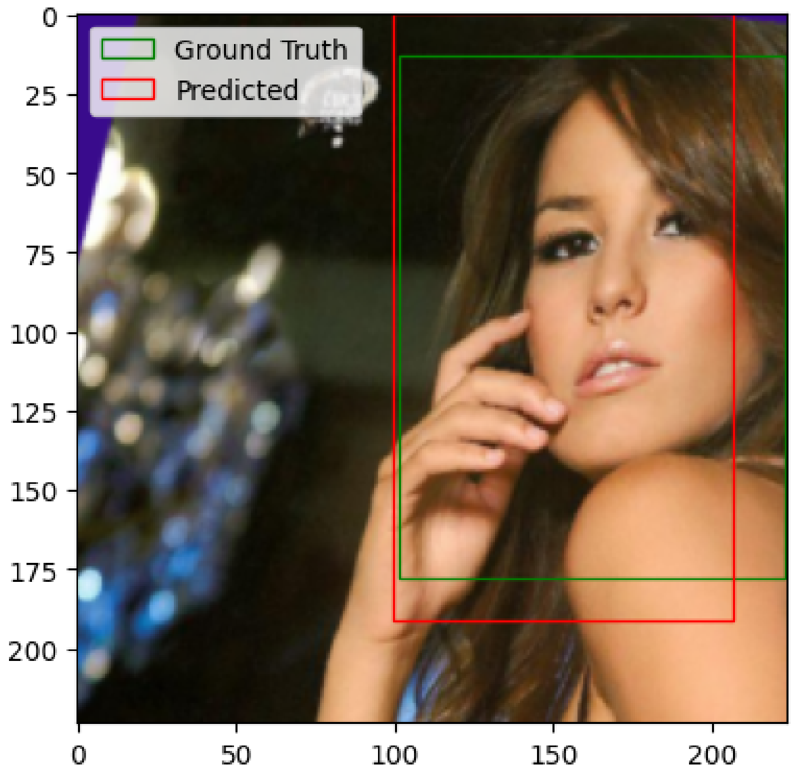
- The model performs significantly better on the subset of the PINS dataset than on the full dataset, as indicated by the verification and identification metrics.
- The difference is likely due to the reduced complexity of the subset.
- It's crucial to evaluate models on diverse and representative datasets to get an accurate assessment of their real-world performance.

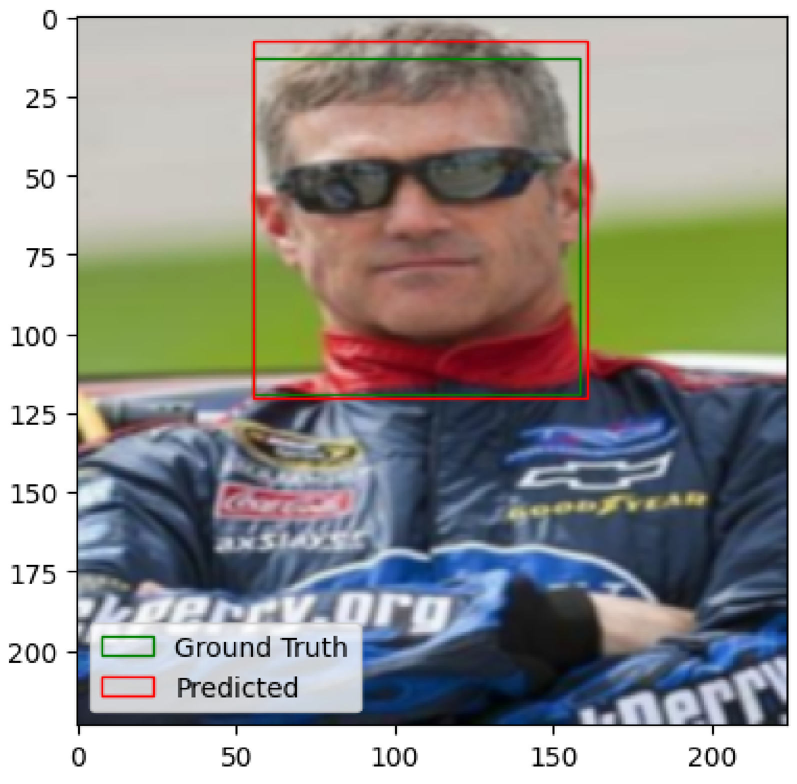
Face Detection Model (`embeddings.py bounding_box.py and facial_detection_celeba_bbox_optimized.py`)

The face detection model implementation involves:

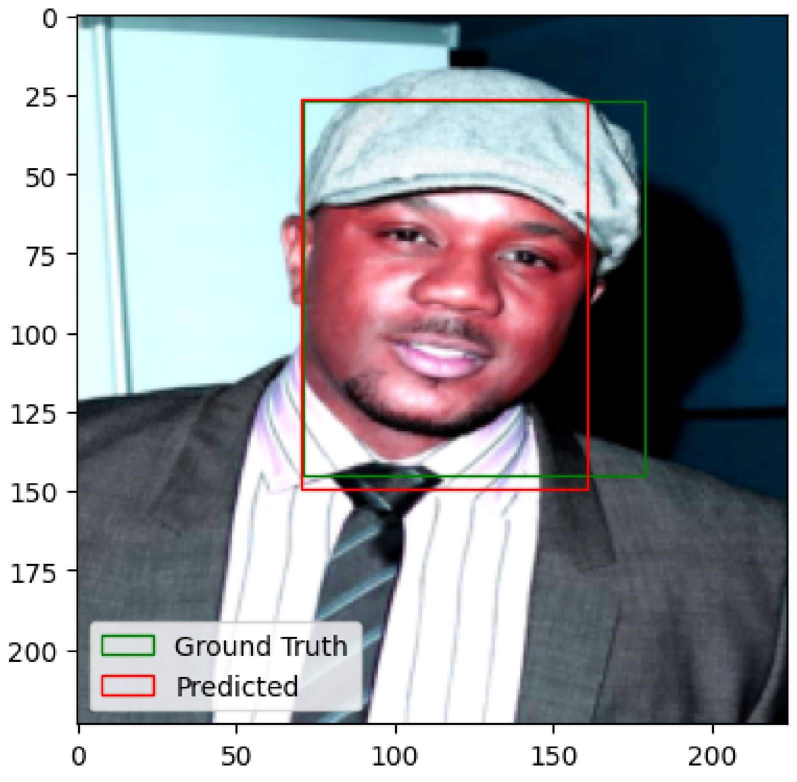
- **Model Architecture:** The `FaceBBoxModel` class defines the structure of the face detection network. It utilizes convolutional layers, batch normalization, max pooling, and fully connected layers for bounding box prediction.
- **Data Loading:** The CelebA dataset is loaded from Hugging Face for training the bounding box model. This dataset contains images of celebrities with bounding box annotations.
- **Dataset Class:** A custom dataset class is created to manage the images and their bounding box information and applies transformations to the images and bounding boxes such as resizing and normalization.
- **Training Process**: The training process involves iterating through the dataset and updating the model parameters using an optimizer and the loss function. The training function includes gradient scaling for faster training and utilizes the Adam optimizer with a learning rate scheduler.
- **Data Visualization:** A function is implemented to display images with their bounding box annotations. This visualization is useful for confirming the dataset and the model's outputs.
- These are some of the outputs of the model on the testing set of the celebA:


Ground Truth
Predicted
0
25
50
75
100
125
150
175
200
0
50
100
150
200


Ground Truth
Predicted
0
25
50
75
100
125
150
175
200
0
50
100
150
200



0
25
50
75
100
125
150
175
200
0
50
100
150
200
Ground Truth
Predicted



0
25
50
75
100
125
150
175
200
0
50
100
150
200
Ground Truth
Predicted

Key Components and Functions:

- **`app.py`**: This file serves as the central hub for the facial recognition application, integrating the GUI, camera interface, and deep learning models for face detection and recognition. It controls the overall flow of the application. It manages the user interface, processes user actions, captures camera data, performs face detection, extracts embeddings, and manages registration and authentication.
- **`insightface_app.py`**: This alternatively implements the core functionalities by utilizing the `InsightFace` library. This file offers a way to implement facial recognition tasks using different tools and models from the main application.
- **`embeddings.py`**: This file contains the implementation of the face recognition model based on the `EnhancedFaceRecognitionModel` class. This includes the forward pass through the network to extract embeddings, the ArcFace loss function, and other components specific to the model.
- **`bounding_box.py`**: This is the implementation of the custom face detection model. This file includes the convolutional architecture and training process of the bounding box model. It is an implementation that predicts the bounding box coordinates of detected faces.
- **`bounding_box_yunet.py`**: This is another face detection implementation using the `YuNet` face detection model, which is used as an alternative face detection approach.

Summary

The described facial recognition system is a sophisticated and robust application designed for real-time face detection and recognition. It integrates custom-trained deep learning models with widely adopted computer vision libraries. Implemented primarily in Python, the system leverages a range of specialized libraries to ensure functionality, performance, and user-friendliness. The application aims to provide a complete solution, from model inference to practical deployment via an intuitive Graphical User Interface (GUI), incorporating advanced techniques in deep learning, loss function optimization, and rigorous evaluation metrics.

Key Components and Features:

- **Real-time Face Detection and Recognition**: The system performs face detection and recognition in real-time, making it suitable for interactive security, attendance, or access control applications.
- **Configurable Camera Sources**: Supports flexible input from both standard built-in webcams and network-connected ESP32-CAMs, configurable directly within the GUI.
- **Multiple Face Detection Models**: Users can dynamically select and switch between three distinct face detection algorithms: the highly efficient **YuNet Detector**, the classic **Haar Cascade Detector**, and a **Custom CNN Detector**, providing adaptability to various scenarios.
- **Custom-Trained Deep Learning Models**: Utilizes custom-trained deep learning models for both face detection (Custom CNN) and recognition (ArcFace-based), tailored for specific performance characteristics.
- **User Registration and Authentication**: Provides a robust system for registering new individuals by capturing and averaging multiple face samples, and subsequently authenticating them via live camera feed.
- **Access Control Management**: Allows for granular access permissions (allowed/denied) to be assigned to registered users, enabling policy-based access.
- **Enhanced GUI Interface**: A comprehensive graphical user interface, built with CustomTkinter, offers intuitive tabbed navigation ("Main Controls," "Settings," "Manage Persons") for streamlined user interaction and system management.
- **Queue-Based Embedding Averaging**: Averages multiple face embeddings collected in real-time to enhance recognition accuracy and robustness against momentary fluctuations in pose or lighting.
- **Thread-Safe Database Operations**: Ensures smooth, concurrent functioning and data integrity through thread-safe access to the SQLite database.
- **Real-time Feedback**: Provides immediate visual feedback through a status label and a dedicated logging textbox within the GUI, showing recognition results, match distances, and system messages.

Model Details:

- **Face Detection Model**:
  - A custom-trained Convolutional Neural Network (CNN) is available, trained on the CelebA dataset (`224x224` input, `x, y, width, height` output).
  - Alternatively, the YuNet face detector model (`face_detection_yunet_2023mar.onnx`) is implemented as a robust

    - and efficient alternative, particularly optimized for real-time streams including those from ESP32-CAM.
    - The Haar Cascade Detector is also integrated as a lightweight option.
- **Face Recognition Model**:
    - An ArcFace-based architecture with residual connections (ResArkSGD) is utilized.
    - The model's embedding dimension is 512.
    - Trained with an ArcFace loss function with dynamic scaling and margin parameters to maximize feature discrimination.
    - Uses face alignment as a preprocessing step.
    - Trained on the VGGFace2-HQ cropped dataset and tested on the PINS face recognition dataset.

Project Structure:

The project is organized as follows:

- `app_V2.py`: The main application file, encompassing the GUI, camera interface, and integration logic for all models.
- `app_V1.py`: The previous version of the main application (retained for historical context).
- `embeddings.py`: Contains the implementation of the face recognition model and embedding generation logic.
- `bounding_box.py`: Contains the implementation of the custom face detection model and the integration of Haar Cascade.
- `bounding_box_yunet.py`: Contains the implementation of the YuNet face detection model.
- `models/`: Directory housing all trained deep learning models.

Installation and Usage:

- **Requirements**: The system requires Python 3.8+ and ideally a CUDA-capable GPU for faster processing, though CPU inference is supported.
- **Setup**: The repository can be cloned from GitHub, a virtual environment set up, and required dependencies installed via `pip`. Pre-trained models must be downloaded and placed in their specified directories.
- **Operation**: The main application is launched by running `app_V2.py`.
- **Interaction**: The system supports user registration by capturing multiple face samples and authentication via a live camera feed. Users can dynamically

configure camera sources, switch between various face detection models, adjust the recognition threshold, and manage registered individuals directly through the intuitive tabbed GUI.

Resources and Citations

- **CelebA Dataset**: The CelebA dataset is used for training the custom face detection model. This dataset contains over 200,000 celebrity images with bounding box and facial landmark annotations:
  - https://huggingface.co/datasets/hfaus/CelebA_bbox_and_facepoints
- **VGGFace2-HQ Cropped Dataset**: The VGGFace2-HQ Cropped dataset is used for training the face recognition model:
  - https://www.kaggle.com/datasets/zenbot99/vggface2-hq-cropped
- **PINS Face Recognition Dataset**: The PINS dataset is used for testing the face recognition model:
  - https://www.kaggle.com/datasets/hereisburak/pins-face-recognition
- **Training Notebooks**: The following notebooks contain code for training the face detection and recognition models:
  - **Face Detection (CelebA)**:
    - https://www.kaggle.com/code/ahmedkamal75/facial-detection-celeba-bbox-optimized
  - **Face Recognition (ArcFace)**:
    - `models/resarksgd/arkface-residual-connections-part-2.ipynb`
    - https://www.kaggle.com/code/ahmedkamal75/arkface-residual-connections-part-2
- **GitHub Repository**: The code for the face recognition system can be found at:
  - https://github.com/AhmedKamal75/Graduation_Project
- **Pre-trained Models**: The pre-trained models can be downloaded from the following links:
  - https://www.kaggle.com/models/ahmedkamal75/bbox_model_v5_epoch_8/
  - https://www.kaggle.com/models/ahmedkamal75/bbox_v5_augmented_epoch_50
  - https://www.kaggle.com/models/ahmedkamal75/resarksgd-acc-88/
  - https://www.kaggle.com/models/ahmedkamal75/resarksgd95/
  - `resarksgdaug94`

License

This project is licensed under the MIT License.