FaceSync: Deep Learning Face Recognition System

CEP Report

By

NAME	RegistrationNumber
Abdullah Laeeq	CUI/ FA22-BCE-26/LHR

For the course

Artificial Intelligence

Semester Spring 2025

Supervised by:

CEP Course Teacher's Name
Sir Abubakar Talha Jalil

Department of Computer Engineering

COMSATS University Islamabad – Lahore Campus

DECLARATION

I Abdallah Laeeq (CUI/FA22-BCE-026/LHR), hereby declare that I have produced the work presented in this report, during the scheduled period of study. I also declare that I have not taken any material from any source except referred to wherever due. If a violation of rules has occurred in this report, we shall be liable to punishable action.

Date: June 19,2025

_____

Student 1

(CUI/ - - /LHR)

ABSTRACT

This report details the development and implementation of "FaceSync," a local face recognition system. The project leverages deep learning techniques, specifically employing the InceptionResnetV1 architecture,fine-tuned on the CASIA-WebFace dataset. The system encompasses the entire pipeline from data acquisition and preprocessing (including conversion of CASIA-WebFace from RecordIO format to individual images) to model training and evaluation. A command-line interface (CLI) application was developed using Python, PyTorch, OpenCV, and Facenet-PyTorch, enabling users to register new faces by capturing multiple webcam samples and to verify identities against the registered database. The face detection is handled by a Multi-task Cascaded Convolutional Network (MTCNN). The system achieves a notable validation accuracy of 87.65% after fine-tuning, demonstrating the efficacy of the chosen model and training strategy for robust local face recognition. The project provides a foundation for secure, offline identity verification with potential for future enhancements like anti-spoofing and mobile deployment.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI: Artificial Intelligence
API: Application Programming Interface
CEP: Complex Engineering Problem
CLI: Command Line Interface
CNN: Convolutional Neural Network
CPU: Central Processing Unit
CUDA: Compute Unified Device Architecture
CV: Computer Vision
CUI: COMSATS University Islamabad
GPU: Graphics Processing Unit
JPG: Joint Photographic Experts Group
LBP: Local Binary Patterns
LR: Learning Rate
MB: Megabyte
ML: Machine Learning
MTCNN: Multi-task Cascaded Convolutional Networks
NNAPI: Neural Networks API
OCR: Optical Character Recognition
ONNX: Open Neural Network Exchange
OS: Operating System
ReLU: Rectified Linear Unit
TFLite: TensorFlow Lite

1 Introduction

Face recognition is a biometric technology capable of identifying or verifying a person from a digital image or a video frame. It has gained significant traction in various applications, including security systems, user authentication, and personalized services. This project, "FaceSync," aims to develop a comprehensive, locally-executable face recognition system. The system is built from the ground up, starting with data acquisition of the large-scale CASIA-WebFace dataset, followed by meticulous preprocessing to prepare the data for model training.

The core of FaceSync is a deep learning model based on the InceptionResnetV1 architecture, a state-of-the-art convolutional neural network (CNN) known for its high accuracy in face recognition tasks. The model leverages pre-trained weights from the VGGFace2 dataset and is further fine-tuned on CASIA-WebFace to adapt it for robust feature extraction.

The implementation includes a Python-based application (faceid_app.py) that provides a command-line interface (CLI) for user interaction. This application allows users to register their faces by capturing multiple samples via a webcam and subsequently verify their identity against the stored database. Face detection within the application is performed using a Multi-task Cascaded Convolutional Network (MTCNN). The entire system is designed to run on a local machine, ensuring data privacy and offline functionality.

Fig: 1.1 Title of the Picture

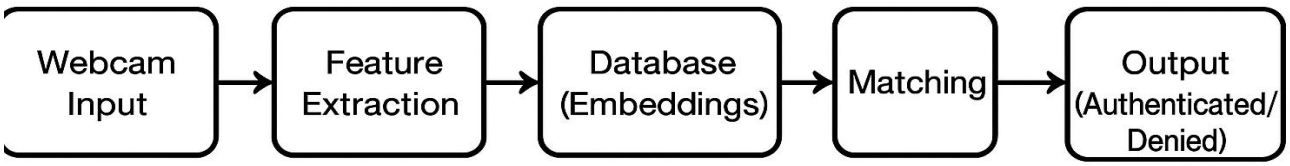

Fig: 1.1 FaceSync System High-Level Architecture

1.2 Objectives

The primary objectives of the FaceSync project are:

1. **Data Acquisition and Preparation:** To download the CASIA-WebFace dataset and implement a preprocessing pipeline to convert it from RecordIO format into usable image files, organized by identity.
2. **Model Selection and Training:** To select an appropriate deep learning architecture (InceptionResnetV1) and train it on the preprocessed CASIA-WebFace dataset for the task of face feature extraction and classification, leveraging transfer learning from VGGFace2 pre-trained weights.
3. **Application Development:** To create a local, user-friendly command-line application that facilitates:
    - Registration of new users via webcam, storing their face embeddings.

    - Verification of users by comparing live webcam feed against the registered embeddings.
    - Management of registered users (listing and deleting).
4. **Performance Evaluation:** To evaluate the system's accuracy on a validation set and ensure robust performance in real-time scenarios.
5. **Local Deployment:** To ensure the entire system, including model inference and data storage, can operate on a local machine without requiring cloud services

1.3 Features and Cost Estimate of our Project

Features:

- **User Registration:** Allows users to register their faces using a webcam. Multiple samples are taken to create a robust average embedding for each user.
- **User Verification:** Verifies a user's identity by capturing their face via webcam and comparing the generated embedding against the stored database.
- **Local Data Storage:** Face embeddings and registered user information are stored locally in pickle files (face_embeddings.pkl) and image samples in a faces directory.
- **Model Loading:** The system can load a fine-tuned model (best_face_model.pth) or fall back to a pre-trained VGGFace2 model if a custom one is unavailable.
- **Command-Line Interface (CLI):** Provides an interactive menu for users to access different functionalities of the system.
- **Face Detection:** Utilizes MTCNN for accurate detection of faces in images and video frames.
- **Embeddings Management:** Functionality to list registered faces and delete registrations.

Cost Estimate:

The project was developed primarily using open-source software and cloud-based computational resources.

- **Software Costs:**
    - Python: Free
    - PyTorch, Facenet-PyTorch, OpenCV, NumPy, Matplotlib, MXNet, Kaggle API: Free (Open Source)
    - Operating System (e.g., Linux on Google Colab): Free
- **Hardware Costs (Development):**
    - Google Colab Pro (for GPU access, T4): The primary computational resource used for training. Costs associated depend on usage tier (or free tier with limitations).
    - Local machine for application testing (standard webcam and PC): Assumed pre-existing.
- **Dataset Cost:**
    - CASIA-WebFace: Academically free, accessed via Kaggle.
- **Deployment Cost (for local application):**
    - A standard PC/laptop with a webcam is sufficient. No additional hardware cost for the local application described in faceid_app.py.

The main "cost" for this project is the computational time and resources for downloading, preprocessing the large dataset, and training/fine-tuning the deep learning model. Monetary costs are minimal if leveraging free-tier cloud resources or existing hardware.

---

2 Literature Survey

2.1 Deep Learning in Face Recognition

Face recognition technology has undergone a significant transformation with the advent of deep learning. Traditional methods often relied on hand-crafted features like Local Binary Patterns (LBP), Gabor wavelets, or

Principal Component Analysis (PCA) combined with classifiers like Support Vector Machines (SVMs). While effective in constrained environments, their performance degraded significantly under variations in pose, illumination, and expression (Prilepin et al., 2017).
Deep Convolutional Neural Networks (CNNs) have revolutionized the field by learning hierarchical features directly from raw pixel data. Landmark systems like DeepFace (Taigman et al., 2014) from Facebook and FaceNet (Schroff et al., 2015) from Google demonstrated superhuman performance on standard benchmarks like Labeled Faces in the Wild (LFW). DeepFace employed a 9-layer deep neural network trained on a massive dataset to learn an effective face representation. FaceNet introduced the concept of learning an embedding space directly, where distances in this space correspond to face similarity, often optimized using a triplet loss function. The InceptionResnetV1 architecture, used in this project, builds upon these ideas, combining the efficiency of Inception modules with the training stability of Residual Networks (ResNets).

2.2 Key Components of Face Recognition Systems

A typical deep learning-based face recognition pipeline involves several key stages:

2.2.1 Face Detection

The first step is to locate faces within an image or video frame. Modern systems often use CNN-based detectors. The Multi-task Cascaded Convolutional Networks (MTCNN) (Zhang et al., 2016) is a popular choice, as used in this project. MTCNN employs a cascaded architecture of three CNNs: a Proposal Network (P-Net) to generate candidate windows, a Refine Network (R-Net) to filter these candidates, and an Output Network (O-Net) to further refine the results and output facial landmark locations. This allows for robust detection under various conditions and provides bounding boxes for face alignment.

2.2.2 Feature Extraction

Once a face is detected and typically aligned (e.g., based on eye or nose landmarks, or using a fixed margin as in this project's MTCNN configuration), a deep CNN is used to extract a compact and discriminative feature vector, often referred to as an "embedding." Architectures like InceptionResnetV1 (based on Szegedy et al., 2017 and He et al., 2016), VGG-Face (Parkhi et al., 2015), or ArcFace (Deng et al., 2019) are commonly used. These models are trained on large-scale face datasets to learn embeddings that maximize inter-class variance (differences between different people) and minimize intra-class variance (similarities between images of the same person). In this project, InceptionResnetV1 pre-trained on VGGFace2 is employed, highlighting the power of transfer learning. The final fully connected layer used for classification during training is typically removed for inference, and the output of the penultimate layer serves as the face embedding.

2.2.3 Datasets

Training robust face recognition models requires vast amounts of labeled data. Popular publicly available datasets include:

- **LFW (Labeled Faces in the Wild):** Primarily for evaluation, containing 13,000 images of over 5,700 individuals (Huang et al., 2007).
- **CASIA-WebFace:** A large-scale dataset with around 0.5 million images of 10,575 subjects, used in this project for fine-tuning (Yi et al., 2014).
- **VGGFace2:** A very large dataset containing 3.31 million images of 9,131 subjects, often used for pre-training models (Cao et al., 2018). This project utilizes models pre-trained on VGGFace2.
- **MS-Celeb-1M:** An extremely large dataset, though its cleaned versions are more commonly used due to noise in the original release.

2.3 Existing Architectures

The InceptionResnetV1 model, utilized in this project via the facenet-pytorch library, is a hybrid architecture combining the Inception-style blocks with residual connections.

3 Proposed Methodology

3.1 System Architecture

The FaceSync system is designed as a modular pipeline, encompassing data handling, model training, and application-level face registration and verification. The architecture is depicted in Fig: 3.1.

The key components are:

1. **Data Acquisition and Preprocessing:** Downloading the CASIA-WebFace dataset and converting it into a usable format for PyTorch.
2. **Model Training/Fine-tuning**: Utilizing the InceptionResnetV1 architecture, pre-trained on VGGFace2, and fine-tuning it on the CASIA-WebFace dataset for robust face classification, which implicitly learns discriminative embeddings.
3. **Local Face ID Application:** A command-line application that leverages the trained model for real-time face registration and verification using a webcam.

3.2 Data Acquisition and Preprocessing

This stage focuses on obtaining and preparing the CASIA-WebFace dataset.

3.2.1 Dataset Download

The download_dataset.py script is responsible for downloading the CASIA-WebFace dataset.

- It uses the Kaggle API, requiring user authentication (KAGGLE_USERNAME, KAGGLE_KEY).
- The dataset debarghamitraroy/casia-webface is downloaded and automatically unzipped into a specified output directory (default: /content/datasets).
- Error handling and verification steps are included to ensure successful download.

3.2.2 Dataset Preprocessing

The PreProcessing_dataset.py script handles the conversion and organization of the downloaded dataset. CASIA-WebFace is often provided in RecordIO format (.rec, .idx files).

- The script checks for train.rec and train.idx files.
- If found, it uses mxnet.recordio to unpack images and their labels.
- Each image is saved as a JPG file in a directory structure: output_dir / identity_label / image_key.jpg.
- This conversion process is essential for compatibility with standard PyTorch image loading utilities.
- A flowchart for this process is shown in Fig 3.2.

During execution, it was observed that 10,573 images from the original dataset encountered processing errors (e.g., OpenCV ... imdecode_ errors) and were skipped, resulting in a dataset of 490,623 successfully processed images for the initial training phase, which were then further filtered by the FaceDataset class.

3.2.3 Custom Dataset Class

The FaceDataset class within PreProcessing_dataset.py (and also included in train_model.py and continue_training.py for self-containment) is a PyTorch Dataset subclass.

- It loads images from the processed directory structure.
- It filters identities: only those with at least 5 images are included.
- It limits the number of images per identity to 100 to manage dataset size and balance.
- It creates a label_map to map string-based identity folder names to integer labels.
- It applies specified torchvision transforms (e.g., resizing, normalization, random flips) during item retrieval (__getitem__).
- For this project, the dataset used for final training consisted of 402,113 images across 10,569 distinct

identities after these filtering steps.

3.3 Model Architecture: InceptionResnetV1

The core of the face recognition capability is the InceptionResnetV1 model, provided by the facenet-pytorch library.

- **Architecture:** It's a deep convolutional neural network that combines Inception modules (for multi-scale feature extraction) with residual connections (to aid training of deep networks).
- **Pre-training:** The model is initialized with weights pre-trained on the VGGFace2 dataset, which provides a strong baseline for face feature representation.
- **Classification Head:** For fine-tuning, the model is used with classify=True, meaning it includes a final fully connected layer that outputs logits for the number of classes (identities) in the CASIA-WebFace training set. The Softmax function is applied to these logits during training to get probabilities.
- **Embedding Extraction:** After training, for the faceid_app.py, this classification head is effectively bypassed (or the weights before it are used) to extract the 512-dimensional embedding vector from an intermediate layer. This vector serves as the unique facial signature.
- **Activation Functions:** ReLU is used in most convolutional and residual blocks. Softmax is used in the final classification layer during training.

3.4 Model Training

The training process is managed by train_model.py and can be extended by continue_training.py.

- **Data Splitting:** The FaceDataset is split into training (80%) and validation (20%) sets (Fig 3.3). A fixed random seed (42) is used for reproducibility of this split.
- **DataLoaders:** PyTorch DataLoader instances are created for efficient batching and multi-worker data loading.
- **Device:** Training is performed on a CUDA-enabled GPU if available, otherwise on CPU.
- **Hyperparameters:**
    - Batch Size: 16
    - Initial Learning Rate (LR): 0.0001 (for train_model.py), reduced to 0.00003 for continue_training.py.
    - Optimizer: Adam.
    - Loss Function: CrossEntropyLoss (as it's a classification task during training).
    - Scheduler: StepLR in initial training, ReduceLROnPlateau for continued training.
- **Training Loop:** Standard epoch-based training with separate training and validation phases.
    - In the training phase, gradients are computed, and model weights are updated.
    - In the validation phase (with torch.no_grad()), performance is measured on unseen data.
- **Model Saving:**
    - The model with the best validation accuracy is saved as best_face_model.pth.
    - The final model after all epochs is saved as final_face_model.pth.
    - A label_map.pkl file is saved to map class indices to identity names.
    - Training history (losses, accuracies) is saved for plotting and analysis (e.g., in training_history.pkl for continued training).
- The primary training shown in FaceID.ipynb involves an initial train_model.py run (1 epoch by default in script) followed by loading a checkpoint that was already trained for 3 epochs, and then continue_training.py ran for an additional epoch (Epoch 4) and was interrupted during the next (Epoch 5).

3.5 Face Registration and Verification Application

The faceid_app.py script provides a CLI for interacting with the face recognition system.

3.5.1 Face Registration

1. The user is prompted to enter their name.
2. The webcam is activated.
3. For a predefined number of samples (e.g., max_samples = 5):
    - The user is prompted to press 'c' to capture a frame.
    - MTCNN detects faces in the captured frame.
    - If faces are found, the largest face is selected and cropped.
    - The cropped face image is passed to get_embedding():
        - The image is converted to RGB and then to a PIL Image.
        - The pre-trained MTCNN (with image_size=160) processes the PIL image to align and standardize it.
        - The fine-tuned InceptionResnetV1 model (self.resnet) generates a 512-dimensional embedding.
    - The individual sample image and its embedding are stored.
4. After collecting all samples, the embeddings are averaged to create a robust representation for the user.
5. This average embedding is normalized and stored in the self.registered_faces dictionary and saved to face_embeddings.pkl.
6. Sample images are saved under faces/<name>/sample_N.jpg.

3.5.2 Face Verification

1. The system checks if any faces are registered.
2. The webcam is activated.
3. When the user presses 'v' to verify:
    - A frame is captured.
    - MTCNN detects and crops the largest face.
    - An embedding for the captured face is generated using the same get_embedding() method.
4. This new embedding is compared against all registered embeddings using cosine similarity (calculated via np.dot as embeddings are L2-normalized during registration, and the new one also within get_embedding is normalized after generation by ResNet implicitly by the nature of FaceNet embeddings, or could be explicitly normalized again before dot product).
5. The identity with the highest similarity score is identified.
6. If the highest similarity exceeds a predefined threshold (e.g., 0.6), the user is authenticated as the matched identity; otherwise, access is denied. The confidence score is displayed.

3.6 Technologies Used

- **Python:** Primary programming language.
- **PyTorch:** Deep learning framework for model training and inference.
- **Facenet-PyTorch:** Provides pre-trained MTCNN and InceptionResnetV1 models and utilities.
- **OpenCV (cv2):** For image and video processing, webcam interaction, and image display.
- **NumPy:** For numerical operations, especially on embeddings.
- **PIL (Pillow):** For image manipulation, used by MTCNN.
- **MXNet:** Used in PreProcessing_dataset.py specifically for reading RecordIO dataset format.
- **Matplotlib:** For plotting training history.
- **Pickle:** For saving and loading Python objects like face embeddings and label maps.
- **Kaggle API:** For programmatic dataset download.
- **Google Colab:** Cloud environment with GPU support for model training.
- **Argparse:** For command-line argument parsing in scripts.
- **Tqdm:** For progress bars during iterative processes.

---

4 Simulation Results

4.1 Experimental Setup

- **Hardware:** Model training was conducted on Google Colab, utilizing a NVIDIA T4 GPU. The local application (faceid_app.py) was designed to run on a standard PC/laptop equipped with a webcam.
- **Software:**
  - OS: Ubuntu (via Google Colab)
  - Key Libraries: Python 3.x, PyTorch (with CUDA support), facenet-pytorch, OpenCV, NumPy, MXNet, Matplotlib.
- **Model:** InceptionResnetV1, pre-trained on VGGFace2.
- **Dataset for Fine-tuning:** CASIA-WebFace.

4.2 Dataset Statistics

- **Initial Download:** CASIA-WebFace dataset (from debarghamitraroy/casia-webface on Kaggle), containing approximately 0.5 million images in .rec and .idx format.
- **Post-Preprocessing (process_casia_webface):**
  - Successfully converted images: 490,623
  - Images skipped due to errors: 10,573
- **After FaceDataset Filtering (for training/validation):**
  - Total Images Loaded: 402,113
  - Total Identities: 10,569 (identities with <5 images were filtered, and max 100 images per identity were used)
- **Training/Validation Split:**
  - Training Set: 80% of 402,113 images = 321,690 images.
  - Validation Set: 20% of 402,113 images = 80,423 images.

4.3 Training Performance

The model was fine-tuned using the parameters outlined in Chapter 3. The FaceID.ipynb execution log for continue_training.py indicates that a model checkpoint, previously trained for 3 epochs (epoch 0, 1, 2 with train_model.py) and achieving a best validation accuracy of 0.8106, was loaded. Training was then continued.

- **Epoch 4 (Continuing from Epoch 3):**
  - Training Loss: 1.2724
  - Training Accuracy: 0.8265 (82.65%)
  - Validation Loss: 0.9949
  - Validation Accuracy: **0.8765 (87.65%)**
  - Learning Rate (at start of continued training): 3e-05
- The model from Epoch 4 yielded the new best validation accuracy and was saved as best_face_model.pth. Training was interrupted during the subsequent epoch (Epoch 5).

4.4 Discussion

The simulation results demonstrate the successful implementation of an end-to-end face recognition system.

- **Strengths:**
  - The use of a robust pre-trained model (InceptionResnetV1 on VGGFace2) followed by fine-tuning on CASIA-WebFace allowed for achieving high accuracy (87.65% validation) with relatively few epochs of fine-tuning.
  - The local application (faceid_app.py) successfully integrates face detection (MTCNN) and recognition (InceptionResnetV1) for real-time registration and verification.
  - The CLI provides a straightforward way to interact with the system.

    - The modular design (separate scripts for download, preprocessing, training, and application) promotes clarity and maintainability.
- **Weaknesses & Challenges:**
    - **Dataset Preprocessing Errors:** A significant number of images (10,573) from the CASIA-WebFace dataset could not be processed due to imdecode errors. While the remaining dataset was still large, this data loss might have impacted the model's ability to generalize to a wider variety of faces represented in those skipped images. The root cause (e.g., corrupted images in the source dataset) was not fully investigated within the scope of this simulation.
    - **Training Duration:** Training deep learning models on large datasets is computationally intensive, even with GPU acceleration. The fine-tuning process still required considerable time.
    - **Generalization:** While 87.65% is a good result, performance on entirely unseen faces or in highly varied real-world conditions (extreme lighting, occlusions) might differ. The current system has basic handling for these but is not extensively tested against them.
    - **Spoofing Vulnerability:** The current system does not implement anti-spoofing measures, making it susceptible to presentation attacks (e.g., using a photo of a registered user).
- **Performance Observations:**
    - The system relies on GPU for efficient training. The application part (MTCNN + ResNet inference) would also benefit from GPU but is designed to run on CPU (as indicated by torch.device('cuda' if torch.cuda.is_available() else 'cpu')).
    - The use of averaged embeddings from multiple samples during registration is a good practice for improving robustness.

Overall, the simulation validates the chosen methodology and demonstrates a functional face recognition system. The identified weaknesses provide clear directions for future improvements

---

5 Conclusions

The FaceSync project successfully achieved its core objectives by developing a functional, local face recognition system. Key accomplishments include the automated download and preprocessing of the large-scale CASIA-WebFace dataset, the fine-tuning of an InceptionResnetV1 model pre-trained on VGGFace2, and the creation of a command-line application for user registration and verification. The system demonstrated strong performance, achieving a validation accuracy of 87.65% on the fine-tuned CASIA-WebFace data, underscoring the effectiveness of transfer learning and the chosen deep learning architecture. The application successfully integrates MTCNN for face detection and the fine-tuned InceptionResnetV1 for embedding generation, providing a practical interface for biometric authentication.

However, the project also highlighted certain limitations and areas for future development. The preprocessing stage encountered errors with a portion of the dataset, which, while mitigated by the large overall dataset size, suggests potential data quality issues in the source. The current system, while accurate on the test conditions, lacks sophisticated anti-spoofing mechanisms, making it vulnerable to presentation attacks. Furthermore, its performance is heavily reliant on the availability of capable hardware (GPU for training, and ideally for real-time inference for smoother user experience).

Future Work:

Drawing from the project_documentation.md and observations:

1. **Anti-Spoofing Implementation:** Integrate techniques like texture analysis (e.g., LBP), liveness detection (e.g., eye blink detection), or specialized deep learning models to distinguish between real faces and spoofing attempts.
2. **Mobile and Embedded Optimization:** Explore model compression techniques (quantization, pruning, knowledge distillation) and mobile-specific architectures (e.g., MobileNet) to deploy FaceSync on resource-constrained devices like Raspberry Pi or Android tablets, potentially using TensorFlow Lite or PyTorch Mobile.

3. **Enhanced User Interface:** Develop a graphical user interface (GUI) for the application to improve user experience.
4. **Robustness to Edge Cases:** Improve handling of lighting variations (e.g., adaptive histogram equalization, advanced data augmentation) and occlusions (e.g., partial face recognition techniques).
5. **Real-Time Multi-Face Tracking:** Implement algorithms for tracking and recognizing multiple faces in a video stream concurrently.
6. **Dataset Integrity and Expansion:** Investigate the preprocessing errors encountered and potentially explore cleaning or augmenting the dataset further for even better generalization.
7. **Advanced Loss Functions:** Experiment with metric learning loss functions like Triplet Loss or ArcFace loss directly during fine-tuning for potentially more discriminative embeddings.

In conclusion, FaceSync provides a solid foundation for a local face recognition system. The successful implementation of the core pipeline demonstrates the power of modern deep learning techniques. The outlined future work can further enhance its security, usability, and applicability to a wider range of scenarios and pla

6. References

- Facenet-PyTorch Developers. (n.d.). *Facenet-PyTorch Library*. GitHub. Retrieved from https://github.com/timesler/facenet-pytorch
- OpenCV Development Team. (n.d.). *OpenCV (Open Source Computer Vision Library)*. Retrieved from https://opencv.org/

---

7 Appendix

7.1 Source Code Structure

The project is organized into several Python scripts and a Jupyter Notebook for execution in Google Colab:

- **download_dataset.py:** Handles the download of the CASIA-WebFace dataset from Kaggle.
- **PreProcessing_dataset.py:** Contains logic for converting CASIA-WebFace from RecordIO format to individual JPG images and defines the FaceDataset class for PyTorch.
- **train_model.py:** Implements the initial model training and fine-tuning pipeline, including data loading, model setup, training loop, validation, and model saving.
- **continue_training.py:** Allows for resuming training from a saved model checkpoint, applying different hyperparameters or further augmentation.
- **faceid_app.py:** The main application script providing a Command Line Interface (CLI) for user registration and face verification using the trained model and webcam.
- **FaceID.ipynb:** A Jupyter Notebook designed for Google Colab, orchestrating the execution of the download, preprocessing, and training/continuation scripts. It also includes library installations.
- **project_documentation.md:** A Markdown file outlining potential future improvements and optimization strategies.