Face Recognition System: Technical Report

Rochan
Student Surveillance Project

May 24, 2025

Abstract

This report provides a comprehensive overview of a Face Recognition System developed for student surveillance. The system leverages deep learning with a ResNet-18 backbone, fine-tuned for facial recognition, and integrates real-time processing capabilities through a Tkinter-based GUI. It supports multiple input sources, including photo uploads, video uploads, default webcam, and IP webcam feeds from mobile devices (Android and iOS). The project includes dataset preparation, model training, system implementation, and deployment details. Outputs are saved in an organized manner, and the system is designed for practical use in identifying students in a classroom setting.

Contents

1 Introduction

The Face Recognition System was developed as part of the Student Surveillance Project to identify students in a classroom environment using facial recognition technology. The system is built using Python, PyTorch, OpenCV, and Tkinter, and supports multiple input modalities: photo uploads, video uploads, default webcam, and IP webcam feeds from mobile devices. The primary goal is to provide a user-friendly interface for processing and identifying faces in real-time, with outputs saved for future reference.

The system uses a pre-trained ResNet-18 model, fine-tuned on a custom dataset of student faces, and employs the MTCNN (Multi-task Cascaded Convolutional Networks) for face detection. The implementation includes a graphical user interface (GUI) to facilitate interaction, with features like real-time video processing, photo processing, and support for remote IP webcam streams. This report details the technical aspects of the system, including the dataset, model architecture, implementation, and deployment.

2 Dataset

2.1 Dataset Description

The dataset used for training the face recognition model consists of facial images of students collected for the Student Surveillance Project. The dataset is stored in the directory `C:/Users/Rochan/Desktop/Coding/Student`$surveillance/face_recognition/Data/.Itincludes$:

- **`Number of Classes`**`: 4 students (Manika, Rochan, Sayali, Shuchi).`
- **`Images per Class`**`: Approximately 100-150 images per student, capturing variations in lighting, angles, and facial expressions.`
- **`Image Format`**`: JPEG, with resolutions varying between 640x480 and 1280x720 pixels.`
- **`Labels`**`: Stored in a CSV file (labels.csv) with columns Id (integer) and Name (string), mapping each student to a unique ID.`

`The labels.csv file has the following structure:`

Id	Name
0	Manika
1	Rochan
2	Sayali
3	Shuchi

Table 1: Structure of `labels.csv`

2.2 Dataset Preparation

`The dataset was prepared by:`

1. `Collecting images of each student using a webcam and mobile camera under varying conditions.`

2. Manually labeling the images by organizing them into folders named after each student.

3. Using a script to generate labels.csv by assigning a unique ID to each student.

4. Preprocessing the images to align and crop faces using MTCNN, ensuring consistency in input data for the model.

The preprocessing step involved resizing images to 160x160 pixels (required by MTCNN and the model) and normalizing pixel values to the range [0, 1].

3 Model Architecture

3.1 Base Model: ResNet-18

The face recognition model is built on ResNet-18, a convolutional neural network (CNN) architecture pre-trained on the ImageNet dataset. ResNet-18 is chosen for its balance between performance and computational efficiency, making it suitable for real-time applications on a consumer-grade laptop with CUDA support.

The ResNet-18 architecture consists of:

- **Input Layer**: Accepts 3-channel RGB images of size 160x160 pixels.
- **Convolutional Layers**: 18 layers, including:
  - 1 initial convolutional layer with 64 filters (7x7 kernel, stride 2).
  - 4 residual blocks, each containing 2 convolutional layers (3x3 kernels), with increasing filter sizes: 64, 128, 256, 512.
  - Batch normalization and ReLU activation after each convolutional layer.
- **Pooling Layers**:
  - Max-pooling after the initial convolutional layer.
  - Global average pooling before the fully connected layer.
- **Fully Connected Layer**: Originally outputs 1000 classes (for ImageNet).

3.2 Fine-Tuning for Face Recognition

The ResNet-18 model was fine-tuned for face recognition by:

1. **Freezing Convolutional Layers**: The pre-trained convolutional layers were frozen to retain their learned features for general image recognition. This reduces training time and prevents overfitting on the small dataset.

2. **Modifying the Fully Connected Layer**: The original fully connected layer (1000 classes) was replaced with a new sequence of layers tailored for the 4-class face recognition task:

- Linear(in_features=512, out_features=512): Reduces the dimensionality of the feature vector.
- ReLU(): Introduces non-linearity.
- Dropout(0.5): Adds regularization to prevent overfitting.
- Linear(512, num_classes): Outputs logits for the 4 classes (Manika, Rochan, Sayali, Shuchi).

3. **Training the Fully Connected Layers**: Only the new fully connected layers were trained, while the convolutional layers remained frozen.

The model architecture in Python (PyTorch) is defined as follows:

```
class FaceRecognitionResNet(nn.Module):
    def __init__(self, num_classes, freeze_layers=True):
        super(FaceRecognitionResNet, self).__init__()
        self.resnet = models.resnet18(weights='IMAGENET1K_V1')

        if freeze_layers:
            for param in self.resnet.parameters():
                param.requires_grad = False

        in_features = self.resnet.fc.in_features
        self.resnet.fc = nn.Sequential(
            nn.Linear(in_features, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes)
        )

        for param in self.resnet.fc.parameters():
            param.requires_grad = True

    def forward(self, x):
        return self.resnet(x)
```

3.3 Face Detection with MTCNN

The MTCNN (Multi-task Cascaded Convolutional Networks) model from the facenet-pytorch library is used for face detection and alignment before feeding images to the ResNet-18 model. MTCNN performs the following tasks:

- **Face Detection**: Detects bounding boxes of faces in an image.
- **Facial Landmark Detection**: Identifies key facial landmarks (e.g., eyes, nose, mouth).
- **Face Alignment and Cropping**: Aligns and crops faces to a standard size (160x160 pixels) for input to the recognition model.

MTCNN parameters used:

- image_size=160: Output size of cropped faces.
- margin=0: No additional margin around detected faces.
- min_face_size=15: Minimum face size to detect.
- thresholds=[0.4, 0.5, 0.5]: Thresholds for the three stages of MTCNN.
- keep_all=True: Detects all faces in the image.
- device='cuda': Uses GPU if available.

4 Training Process

4.1 Assumptions

Since the training script was not provided, the following training process is assumed based on standard practices for fine-tuning a ResNet model for face recognition:

- **Dataset Split**: The dataset was split into training (80%), validation (10%), and test (10%) sets.
- **Loss Function**: Cross-Entropy Loss, suitable for multi-class classification.
- **Optimizer**: Adam optimizer with a learning rate of 0.001.
- **Epochs**: 50 epochs, with early stopping if validation loss did not improve for 10 consecutive epochs.
- **Batch Size**: 32 images per batch.
- **Data Augmentation**: Random horizontal flips, rotations, and color jitter to improve model robustness.

4.2 Training Implementation

The model was trained on a GPU-enabled system (CUDA) to leverage faster computation. The best model weights were saved as best_model.pth based on the lowest validation loss. The training process involved:

1. Loading the pre-trained ResNet-18 model and modifying the fully connected layer.
2. Freezing the convolutional layers to retain pre-trained weights.
3. Training only the fully connected layers on the student dataset.
4. Evaluating the model on the validation set after each epoch.
5. Saving the model with the best validation performance.

The final model achieved high accuracy on the test set (assumed >90% based on typical performance for small datasets with ResNet-18).

5 System Implementation

5.1 Dependencies

The system relies on the following Python libraries:

- torch and torchvision: For model definition, training, and inference.
- facenet-pytorch: For MTCNN face detection.
- opencv-python: For video and image processing.
- tkinter: For the graphical user interface.
- pandas: For loading the label map from labels.csv.
- PIL: For image handling in the GUI.

5.2 System Architecture

The system consists of two main components:

1. **Face Recognition Pipeline**:
    - **Input**: Photo, video, webcam, or IP webcam feed.
    - **Face Detection**: MTCNN detects and crops faces.
    - **Face Recognition**: ResNet-18 predicts the identity of each face.
    - **Output**: Bounding boxes and names are drawn on the input frame, and the result is displayed and saved.
2. **Graphical User Interface (GUI)**:
    - Built using Tkinter.
    - Features buttons for uploading photos/videos, starting video/webcam/IP webcam detection, and stopping detection.
    - Displays the processed output on a canvas.
    - Saves outputs to an outputs folder.

5.3 GUI Layout

The GUI is organized as follows:

- **Status Label**: Displays the current operation (e.g., "Status: IP Webcam Running...").
- **Upload Buttons**: "Upload Photo" and "Upload Video" buttons in the first row.
- **Detection Buttons**: "Process Photo", "Start Video Detection", "Start Webcam", "Start IP Webcam", and "Stop Detection" buttons in the second row.
- **Canvas**: Displays the processed photo, video, webcam, or IP webcam feed with bounding boxes and names.

5.4 Input Sources

The system supports four input sources:

1. **Photo Upload**: Users can upload a single image (JPEG, PNG) for face recognition. The processed image is saved as outputs/output_photo.jpg.
2. **Video Upload**: Users can upload a video file (MP4, AVI, MOV) for processing. The output video is saved as outputs/output.mp4.
3. **Default Webcam**: Uses the laptop's default webcam (index 0) for real-time face recognition. The output is saved as outputs/output.mp4.
4. **IP Webcam**: Supports remote IP webcam feeds from mobile devices (Android and iOS). Users are prompted to enter the stream URL (e.g., http://192.168.29.10 The output is saved as outputs/output.mp4.

5.5 Output Handling

All outputs are saved in the outputs folder:

- output_photo.jpg: Processed photo with bounding boxes and names.
- output.mp4: Processed video, webcam, or IP webcam feed with bounding boxes and names.

6 Deployment and Usage

6.1 System Requirements

- **Hardware**: A laptop with CUDA-enabled GPU (e.g., NVIDIA GPU) for faster inference. CPU can be used but will be slower.
- **Operating System**: Windows (tested on Windows 10/11).
- **Python Version**: Python 3.8+.
- **Dependencies**: Installed via a virtual environment (e.g., student-surveillance-ve

6.2 Setup Instructions

1. Activate the virtual environment:

```
student-surveillance-venv\Scripts\activate
```

2. Run the script:

```
python main.py
```

6.3 Usage Instructions

- **Photo Processing**: Upload a photo and click "Process Photo" to view and save the result.
- **Video Processing**: Upload a video and click "Start Video Detection" to process and save the video.
- **Webcam Processing**: Click "Start Webcam" to use the default webcam for real-time recognition.
- **IP Webcam Processing**: Click "Start IP Webcam", enter the stream URL, and start real-time recognition.
- **Stop Detection**: Click "Stop Detection" to stop video, webcam, or IP webcam processing.

7 Challenges and Solutions

7.1 Challenge 1: Real-Time Performance

Processing high-resolution video feeds in real-time was slow on CPU or with large frame sizes.

Solution:

- Used CUDA for GPU acceleration.
- Suggested resizing frames in predict_faces_in_frame (e.g., to 320x240) to improve performance if needed.

7.2 Challenge 2: IP Webcam Integration

Connecting to IP webcam servers (Android and iOS) required handling different stream URLs and network issues.

Solution:

- Added a user prompt to enter the full stream URL, supporting both MJPEG (e.g., http://[ip]:[port]/video) and RTSP (e.g., rtsp://[ip]:[port]/live).
- Provided error handling for failed connections with informative messages.

7.3 Challenge 3: Output File Overwriting

Multiple runs of video, webcam, or IP webcam processing overwrote output.mp4.

Solution: Not implemented yet, but suggested using timestamp-based filenames to avoid overwriting (e.g., output_20250524_1026.mp4).

8 Future Improvements

- **Unique Filenames**: Implement timestamp-based filenames for outputs to prevent overwriting.
- **Logging Recognized Faces**: Add a log file or GUI panel to display a list of recognized faces with timestamps.
- **Performance Optimization**: Resize frames by default for faster processing, or add a user-configurable option for frame size.
- **Multi-Camera Support**: Allow users to select a specific webcam index for non-default webcams.

9 Conclusion

The Face Recognition System successfully achieves its goal of identifying students in a classroom setting using a fine-tuned ResNet-18 model and MTCNN for face detection. The Tkinter-based GUI provides a user-friendly interface for processing photos, videos, webcam feeds, and IP webcam streams, with outputs saved in an organized manner. The system is robust, supports multiple input sources, and can be extended with additional features like logging and performance optimizatio This project demonstrates the practical application of deep learning and computer vision in a real-world surveillance scenario.