# CHAPTER THREE

3.1 Introduction

This chapter describes the the methodology adopted in developing the deep face detection model. It explains the project design, tools and technologies used, data flow architecture, model development, preprocessing, and testing procedures. The goal is to ensure that the project is reproducible, scalable, and optimized for real-time face detection in both images and live video streams.

3.2 System Design and Architecture

The face detection system was designed using a modular and layered architecture for flexibility and maintainability. The design includes the following core components:

1. Image/Video Input Layer   
 - Captures input either from static images or webcam streams using OpenCV.

2. Preprocessing Layer   
 - Converts frames to grayscale for compatibility with Haar Cascade classifiers.  
 - Resizes input if needed for speed optimization.

3. Detection Layer   
 - Loads a pre-trained Haar Cascade model from OpenCV.  
 - Applies face detection to the image or frame.  
 - (Optional) TensorFlow modules for future deep model integration.

4. Visualization Layer   
 - Draws bounding boxes around detected faces.  
 - Displays the live output in a real-time video window.

5. Evaluation and Output Layer   
 - Measures detection speed (FPS) and accuracy in test scenarios.  
 - Logs detection results if required.

**Figure: System Architecture Diagram**

PREPROCESSING

DISPLAY & OUTPUT

DETECTION LAYER

IMAGE / VIDEO

3.3 Tools and Technologies Used

|  |  |
| --- | --- |
| **Tool** | **Purpose** |
| Python | Primary programming language |
| OpenCV | Image processing, Haar Cascade detection |
| TensorFlow | Deep learning framework for future integration |
| NumPy | Numerical computations and matrix operations |
| Jupyter Lab | Development environment for writing/testing code |
| Webcam | Real-time video input |

Although the main model implementation was based on OpenCV’s Haar Cascade classifier, the framework is extensible to support TensorFlow-based CNN models for improved performance.

3.4 Dataset and Preprocessing

Dataset Usage  
This project did not involve training a new face detection model. Instead, it relied on pre-trained Haar Cascade XML files from OpenCV's repository (haarcascade\_frontalface\_default.xml). These classifiers are trained on thousands of labeled face images and non-face images.

For future enhancement or custom dataset training, popular datasets like WIDER FACE, FDDB, or LFW are suitable for training deep learning models such as MTCNN, YOLO, or SSD.

Preprocessing Steps  
- Convert all frames or images to grayscale (Haar requires grayscale input).  
- Optionally resize image for performance.  
- Normalize pixel values (in TensorFlow versions).  
- Detect faces using detectMultiScale() method from OpenCV.

Example:  
gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)  
faces = face\_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5)

3.5 Model Development and Integration

Phase 1: Haar Cascade Implementation  
- Load Haar Cascade model.  
- Capture webcam feed or load an image.  
- Process each frame to detect faces.  
- Draw rectangles on detected faces using cv2.rectangle().

Phase 2: TensorFlow Framework Setup (Future Phase)  
- Create an abstract detection class.  
- Add placeholders for deep models like MTCNN, YOLOv3, or SSD.  
- Integrate TensorFlow/Keras model loading with OpenCV pipelines.

Example:  
model = tf.keras.models.load\_model('path\_to\_model')  
predictions = model.predict(preprocessed\_frame)

Though TensorFlow integration was minimal in the uploaded notebook, the system was designed to allow smooth migration to deep learning-based detection with minimal code restructuring.

3.6 Real-Time Face Detection Implementation

Implementation Steps  
1. Initialize webcam using OpenCV.  
2. Read each frame in a loop.  
3. Convert the frame to grayscale.  
4. Detect faces using the Haar classifier.  
5. Draw bounding boxes around faces using labelme.  
6. Display the processed frame with annotations.  
7. Exit on user keystroke.

**Sample Code:**cap = cv2.VideoCapture(0)

while cap.isOpened():

\_ , frame = cap.read()

frame = frame[50:500, 50:500,:]

rgb = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

resized = tf.image.resize(rgb, (120,120))

yhat = facetracker.predict(np.expand\_dims(resized/255,0))

sample\_coords = yhat[1][0]

if yhat[0] > 0.5:

# Controls the main rectangle

cv2.rectangle(frame,

tuple(np.multiply(sample\_coords[:2], [450,450]).astype(int)),

tuple(np.multiply(sample\_coords[2:], [450,450]).astype(int)),

(255,0,0), 2)

# Controls the label rectangle

cv2.rectangle(frame,

tuple(np.add(np.multiply(sample\_coords[:2], [450,450]).astype(int),

[0,-30])),

tuple(np.add(np.multiply(sample\_coords[:2], [450,450]).astype(int),

[80,0])),

(255,0,0), -1)

# Controls the text rendered

cv2.putText(frame, 'face', tuple(np.add(np.multiply(sample\_coords[:2], [450,450]).astype(int),

[0,-5])),

cv2.FONT\_HERSHEY\_SIMPLEX, 1, (255,255,255), 2, cv2.LINE\_AA)

cv2.imshow('EyeTrack', frame)

if cv2.waitKey(1) & 0xFF == ord('q'):

break

cap.release()

cv2.destroyAllWindows()

3.7 Evaluation Strategy

The evaluation of the face detection system was based on the following criteria:

|  |  |
| --- | --- |
| **Metric** | **Description** |
| **Detection Speed** | Measured in frames per second (FPS) |
| **Accuracy** | Number of correctly detected faces vs actual faces |
| **False Positives** | Non-face regions incorrectly marked as faces |
| **False Negatives** | Actual faces not detected |
| **Hardware Utilization** | CPU usage, memory footprint in real-time detection |

While not deeply quantified in the current version, this evaluation framework will guide future experiments with TensorFlow models.

3.8 Summary

This chapter discussed the system architecture, tools, and techniques used in the implementation of a real-time face detection system using Python and OpenCV. While the current version uses Haar Cascades, the modular design supports future integration of deep learning-based models such as MTCNN and YOLO. The approach emphasizes simplicity, speed, and adaptability for deployment in low-resource environments.