Deployment of a Real-Time Face Mask Classification System Using Browser Webcam Streaming and FastAPI

By

Yazhini Venkatraman

University of Victoria, BC, Canada

Report Submitted in Partial Fulfillment of Requirements for the Degree
Master of Engineering

in the

Department of Electrical & Computer Engineering

© Yazhini Venkatraman, 2026 University of Victoria
All rights reserved. This report may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Deployment of a Real-Time Face Mask Classification System Using Browser Webcam Streaming and FastAPI

By

Yazhini Venkatraman

University of Victoria, 2026

Supervisory Committee

Dr. Poman So

Department of Electrical and Computer Engineering

Dr. Navneet Popli

Department of Electrical and Computer Engineering

Abstract

This project presents a real-time face mask classification system designed to support safety monitoring in public and controlled environments such as workplaces, institutions, and healthcare facilities. The system detects a person's face and classifies mask usage into four categories: with mask, without mask, with N95 mask, and improper mask. A curated dataset of face images was preprocessed through face detection, cropping, resizing, normalization, and augmentation to improve the model's robustness under different lighting and orientation conditions.

The model is built using a MobileNet based convolutional neural network, chosen for its efficiency and suitability for real-time applications. A classical Single Shot Detector is used to localize faces before classification. The trained model is evaluated using standard metrics including accuracy, precision, recall, F1-score, and a confusion matrix and achieves strong performance across all four mask categories. A live webcam interface has also been implemented to demonstrate real-time inference and practical usability.

Overall, this work shows that a lightweight deep learning pipeline can reliably classify mask wearing conditions in real time on standard hardware. The system forms a basis for further improvements, such as handling complex occlusions, expanding the dataset with more diverse samples, and deploying the model as a standalone desktop or mobile application for real-world monitoring needs.

Table of Contents

Abstract **2**

Table of Contents **4**

List of Tables **5**

List of Figures **7**

Glossary **8**

Acknowledgements **9**

1 Introduction **10**
1.1 Report Objectives and Contributions 12

2 Literature Review **13**
2.1 Overview of Deep Learning Based Face Mask Detection 13
2.2 Taxonomy of Face Mask Detection Approaches 14
2.3 Feature-Extraction and Classification Pipelines 15
2.4 Segmentation and Fine-Grained Mask Analysis 16
2.5 IoT, Edge and Browser Based Deployment Frameworks 16
2.6 Datasets, Data Diversity, and Augmentation Techniques 17
2.7 Summary and Research Gaps 17

3 Experiment and Methodology **19**
3.1 Overview 19
3.2 System Architecture 19
3.2.1 Overall Workflow 19
3.2.2 Hardware Setup 20
3.2.3 Software Stack 20
3.3 Dataset Preparation 22
3.3.1 Dataset Sources and Class Schema 22
3.3.2 Data Preprocessing 23
3.3.3 Data Augmentation 24

3.3.4 Data Splitting . . . 25
3.4 Model Architecture . . . 26
3.4.1 Backbone and Preprocessing Interfaces . . . 26
3.4.2 Base Model Selection . . . 26
3.4.3 Model Modification . . . 27
3.4.4 Transformation from Input Image to Classification . . . 27
3.4.5 Hyperparameters . . . 28
3.5 Face Detection Baseline . . . 28
3.6 Evaluation Methodology . . . 29
3.6.1 Evaluation Metrics . . . 29
3.6.2 Validation and Testing . . . 29
3.7 Real-Time Implementation . . . 30
3.7.1 System Integration . . . 30
3.7.2 Frame Processing Pipeline . . . 30
3.7.3 Prototype Validation Using Capture-and-Predict Script . . . 31
3.7.4 Final Real-Time Continuous Detection System . . . 31
3.7.5 Latency & Performance Analysis . . . 32
3.7.6 Frame Annotation Conventions . . . 35
3.7.7 Browser Based Deployment . . . 35
3.8 Limitations and Challenges . . . 39

4 Results **42**
4.1 Accuracy Evaluation . . . 42
4.1.1 Evaluation Across EarlyStopping Patience Values . . . 42
4.1.2 Confusion Matrix Visualisation Across Patience Values . . . 42
4.1.3 Real-Time Detection Performance . . . 45
4.1.4 Browser Based Deployment Results . . . 46
4.1.5 Latency Analysis . . . 47
4.1.6 Browser Visualization Quality . . . 49
4.1.7 Summary of Findings . . . 50

5 Conclusion and Future Work **52**
5.1 Conclusion . . . 52
5.2 Future Work . . . 52

Bibliography **53**

A Source Code **56**

List of Tables

2.1 Comparison of object detection based face mask detection approaches . . 15

3.1 Software stack used for system implementation and evaluation. 21
3.2 Class Index Mapping Used Across Data Loaders and Model Head. 22
3.3 Table 3.2 Dataset Split Configuration and Output Directories. 26
3.4 Progressive Feature Transformation in MobileNet 27
3.5 Visualization color codes used in real-time overlays (OpenCV BGR). . . 35

4.1 Per-class metrics for patience $p = 1$ 43
4.2 Per-class metrics for patience $p = 2$ 44
4.3 Per-class metrics for patience $p = 3$ 45
4.4 Per-class metrics for patience $p = 4$ 46
4.5 Per-class metrics for patience $p = 5$ 47
4.6 Per-class metrics for patience $p = 6$ 47
4.7 Per-class metrics for patience $p = 7$ 47
4.8 Overall accuracy for patience values 1–7. 47
4.9 Comparison between desktop and Browser Based Interface 50

List of Figures

2.1 Face Detection and Preprocessing Stages, from [1]. 13
2.2 Overview of facial contour extraction. Without a Mask (a1) Background Removed;(a2) Extract contour; (a3) Smooth contour curve; With a surgical Mask (b1) Background Removed; (b2) Extract contour; (b3) Smooth contour curve, from [2]. 14
2.3 Network architecture of the G-Mask, from [3]. 16

3.1 Overall Workflow 20
3.2 Data Preprocessing 23
3.3 Face Detection and Preprocessing Stages 24
3.4 Evaluation - Without Mask 32
3.5 Evaluation - With Mask 33
3.6 Evaluation - With N95 Mask 33
3.7 Evaluation - Incorrect Mask Over Mouth 34
3.8 Evaluation - Incorrect Mask Over Chin 34
3.9 Browser based mask detection interface rendered in a browser. 36
3.10 Browser camera access prompt for the Browser based mask detection client. 37
3.11 Real-time continuous mask detection and classification result - Without Mask 38

4.1 Confusion matrix heatmap for patience $p = 1$. 43
4.2 Confusion matrix heatmap for patience $p = 2$. 44
4.3 Confusion matrix heatmap for patience $p = 3$. 44
4.4 Confusion matrix heatmap for patience $p = 4$. 45
4.5 Confusion matrix heatmap for patience $p = 5$. 45
4.6 Confusion matrix heatmap for patience $p = 6$. 46
4.7 Confusion matrix heatmap for patience $p = 7$. 46
4.8 Real-time continuous mask detection and classification result - Without Mask 48
4.9 Real-time continuous mask detection and classification result - With Mask 48
4.10 Real-time continuous mask detection and classification result - With N95 Mask 49

4.11 Real-time continuous mask detection and classification result - Incorrect Mask 50

Glossary

CNN	Convolutional Neural Network
DL	Deep Learning
ML	Machine Learning
SSD	Single Shot Detector
DNN	Deep Neural Network
FPS	Frames Per Second
ROI	Region of Interest
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative
F1-score	Harmonic mean of Precision and Recall
RGB	Red Green Blue
H5	Hierarchical Data Format used for saving models
Adam	Adaptive Moment Estimation Optimizer
N95	N95 Respirator Mask Category
CPU	Central Processing Unit
Keras	High-level Deep Learning API used with TensorFlow
TensorFlow	Deep Learning framework used for model training
OpenCV	Open Source Computer Vision Library
MTCNN	Multi-task Cascaded Convolutional Network
Albumentations	Image augmentation library used for preprocessing

Acknowledgements

First and foremost, I would like to express my heartfelt gratitude to my **parents** for their unconditional love, support, and encouragement throughout my academic journey. Their constant motivation and belief in me have been the foundation of all my achievements.

I extend my sincere thanks to my supervisors, **Dr. Poman So** and **Dr. Navneet Popli**, for their guidance, feedback, and continuous support during the development of this project. Their expertise and mentorship have been invaluable.

I am grateful to the **University of Victoria** and the Department of Electrical and Computer Engineering for providing the resources, learning environment, and academic opportunities that enabled the successful completion of this work.

Finally, I would like to acknowledge my friends, colleagues, and everyone who contributed their support through discussions and encouragement.

Chapter 1

Introduction

Face masks play a fundamental role in contamination control and biosafety across a wide range of professional environments. During the COVID-19 epidemic, people became much more conscious of the need to wear masks, but this need has been there for a long time in healthcare and scientific settings. In hospitals, surgical theaters, pharmaceutical cleanrooms, and biomedical laboratories, masks help maintain sterile conditions, protect patients, and prevent cross-contamination during sensitive procedures. Studies have shown that effective mask detection systems directly support compliance monitoring and improve public safety outcomes [1]. In research facilities, consistent mask usage also helps prevent biological sample contamination and ensures sticking to occupational safety standards. Consequently, reliable and continuous monitoring of proper mask usage is not merely a convenience but an operational and regulatory necessity.

Manual monitoring of mask compliance, however, is labor intensive and vulnerable to human error. Personnel responsible for oversight may experience visual fatigue, reduced attentiveness, or difficulty supervising multiple access points simultaneously. As institutions expand their use of automation, computer vision technologies have emerged as practical tools for supplementing or replacing manual inspection. Automated mask detection enables real-time enforcement, streamlined entry control, and improved surveillance. The increasing relevance of such systems is evident in recent work demonstrating the integration of mask detection with biometric authentication in real-time web applications [4]. These developments underscore the need for robust and deployable mask detection systems that function reliably in diverse real-world environments.

Advances in deep learning particularly Convolutional Neural Networks (CNNs) and one-stage object detectors have significantly improved the accuracy and efficiency of mask classification systems. Lightweight architectures such as MobileNet, VGG, Inception, and single-shot detectors like YOLO and SSD demonstrate strong performance on mask detection benchmarks. YOLOv8 have shown near-perfect identification accuracy on curated datasets, highlighting their suitability for real-time public safety monitoring [5]. CNN-based models leverage hierarchical feature representations, allowing them to recognize subtle variations in mask type, degree of coverage, and mask misplacement. As a result,

automated mask detection has become a key research area in applications such as airport screening, public transportation, healthcare triage, and access controlled facilities.

Despite these advancements, a distinction remains between many academic research prototypes and fully deployable real-world systems. Commercial solutions such as Hikvision mask detection modules, Invixium IXM TITAN access terminals, and Sirix Mask Detection AI demonstrate that deployment ready systems are feasible in practice. However, these systems are typically proprietary, with limited public disclosure of their training pipelines, architectural design choices, or evaluation methodologies.

In contrast, a substantial portion of the academic literature focuses primarily on maximizing classification accuracy on controlled datasets, often providing limited discussion of deployment oriented considerations. Consequently, there remains a need for transparent and reproducible academic systems that explicitly integrate model development, evaluation, and practical deployment considerations within a unified framework.

Recent IoT oriented frameworks highlight the difficulty of deploying deep learning models efficiently on resource constrained devices, even when accuracy is high [6]. In healthcare and research environments, technical performance alone is insufficient; systems must also adhere to ethical guidelines, institutional safety policies, and privacy standards. This project explicitly addresses these deployment oriented considerations through a modular system design that emphasizes real-time performance, usability, and reproducibility. The proposed solution employs a lightweight MobileNet based classifier combined with Caffe SSD face localization to achieve real-time inference on standard CPU based systems. While privacy preserving computation and large scale multi camera deployment are beyond the scope of the current implementation, the system architecture is designed to support future extensions in these directions.

To contextualize the proposed solution, Chapter 1 introduces the significance of mask compliance and the challenges associated with manual monitoring. Chapter 2 reviews major research contributions in mask detection, including CNN architectures, single-stage detectors, and dataset construction techniques. This literature review establishes the foundation for choosing a lightweight model architecture suitable for real-time deployment.

Chapter 3 presents the full experimental methodology, covering dataset preprocessing, face localisation using Caffe-SSD, augmentation techniques, and MobileNet-based classifier training. Also the deployment of the trained model within both a local real-time inference pipeline and a browser based, client server interface implemented using FastAPI and browser side scripting. Chapter 4 reports the evaluation results using accuracy, confusion matrices, classwise performance metrics.

Finally, Chapter 5 summarizes the contributions of this work and outlines future directions in deployment, multi-camera support, and privacy preserving computation. Potential applications include clinical laboratories, industrial cleanrooms, and any environment where contamination control is critical.

In summary, this project bridges the gap between high-accuracy deep learning research and practical, deployable mask detection solutions. By combining reliable mask classification, effective face localisation, and a user-friendly real-time inference described in Chapter 3, the proposed system provides a viable tool for compliance monitoring in safety-critical environments. The following chapters detail the system implementation methodology, evaluation results, and practical implications of the proposed approach.

1.1 Report Objectives and Contributions

The primary objective of this project is to design, implement, and evaluate a practical real-time face mask detection system tailored for healthcare, laboratory, and other safety critical environments. While modern deep learning methods offer strong classification accuracy, this project focuses on systematic model development, training, and evaluation using a publicly available dataset, with an emphasis on reliable multi-class mask classification rather than large-scale operational deployment.

Key contributions include:

- **Development of a four-class mask classifier** using MobileNet, capable of distinguishing *with_mask*, *without_mask*, *with_n95*, and *incorrect_mask* in real time.
- **Integration of an SSD-based face detector** for stable localisation under varying orientations and lighting.
- **Construction of a robust preprocessing and augmentation pipeline** for training stability and generalization.
- **Implementation of a real-time webcam detection framework** with bounding boxes, confidence scores, and user controlled activation.
- **User-centric, ethical design**, featuring local processing and operator initiated monitoring.
- **Comprehensive evaluation**, including accuracy, per class metrics, inference speed, and qualitative analysis.

Chapter 2

Literature Review

2.1 Overview of Deep Learning Based Face Mask Detection

The COVID-19 pandemic accelerated the development of intelligent face mask detection systems, driving rapid adoption of deep learning in public health monitoring. Convolutional neural networks (CNNs), transfer learning, and lightweight architectures have demonstrated strong capabilities in recognizing masked and unmasked faces, as well as differentiating mask types and wearing conditions [7].

Putri and Athoillah [1] achieved 94.7% accuracy using a CNN-based binary mask detection model. Tarnpradab et al. [4] developed a real-time Browser based recognition framework using VGG16, VGGFace, and InceptionResNetV2, achieving 93.3% accuracy and strong recall performance. Dewi et al. [5] employed YOLOv8 for high-precision mask detection, achieving a 99.1% mAP by merging two major datasets (FMD and MMD).

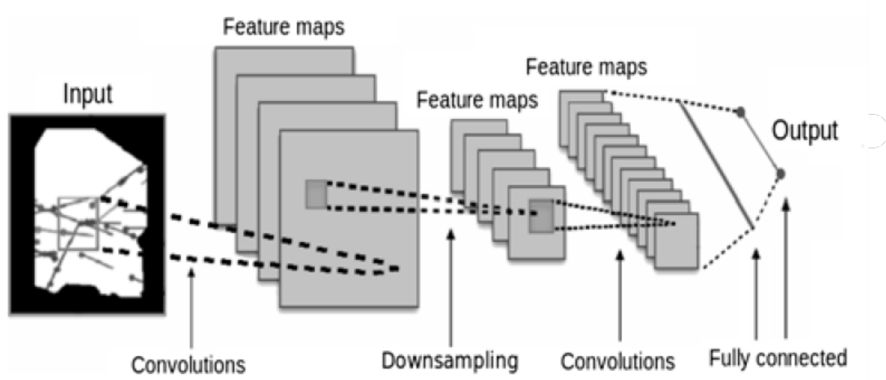

Figure 2.1: Face Detection and Preprocessing Stages, from [1].

Hybrid deep learning systems also gained traction. Dubey et al. [6] integrated ResNet50 and MobileNetV2 in a Hybrid Flame–Sailfish Optimization (HFSO) framework, achieving 97.5% accuracy across multiple datasets and supporting real-time edge deployment.

Transfer-learning-based classification approaches were also explored in [8], demonstrating that models such as MobileNet, Xception, and DenseNet can effectively generalize to varied mask types (two categories of qualified masks like N95 masks, disposable medical masks and others are unqualified masks) and environments like natural scenes (poor lighting conditions, object occlusion, dense crowds, multiple objects, etc..)

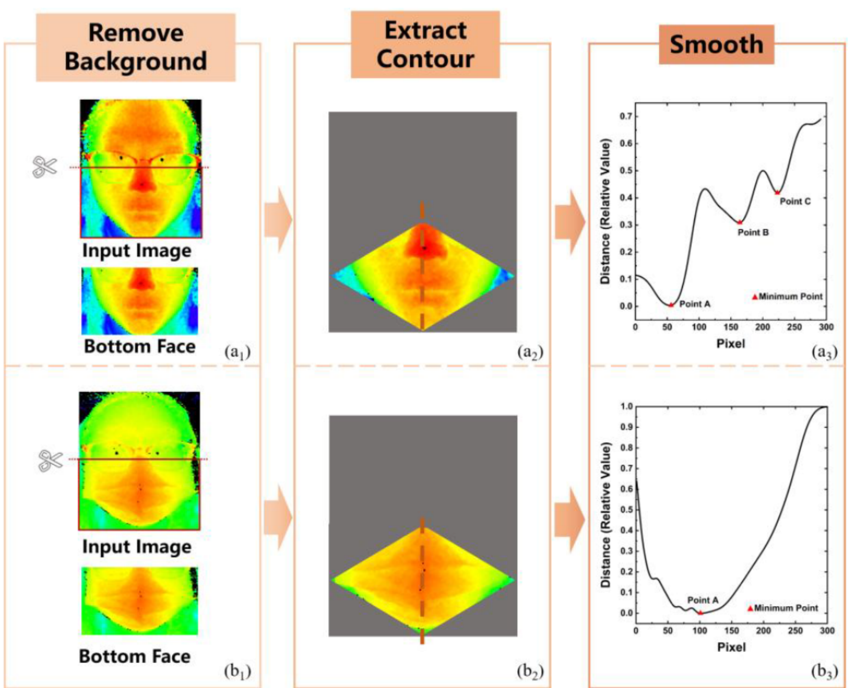

Figure 2.2: Overview of facial contour extraction. Without a Mask (a1) Background Removed;(a2) Extract contour; (a3) Smooth contour curve; With a surgical Mask (b1) Background Removed; (b2) Extract contour; (b3) Smooth contour curve, from [2].

A time-of-flight (ToF) camera captures depth images by measuring the distance between the camera and facial surfaces using reflected infrared light. From the depth image, a central facial contour is extracted, and the relative depth variation along this contour is computed. The distance values in Fig. 2.2 represent these relative depth differences, plotted as a function of contour position (pixel index), enabling discrimination between different mask types based on facial shape changes.

Comprehensive reviews such as Zhang et al. [2] and Nowrin et al. [7] highlight persistent challenges in mask detection, including dataset imbalance, variations in mask type, illumination sensitivity, and limited coverage of improperly worn masks. These identified challenges motivate the design choices adopted in the present work, which focuses on multi-class mask classification and improved handling of improper mask usage.

2.2 Taxonomy of Face Mask Detection Approaches

Zhang et al. [2] categorize mask detection approaches into:

1. Feature-extraction-and-classification pipelines,
2. Object-detection-based systems,
3. Multi-sensor-fusion and multimodal frameworks.

Similarly, Nowrin et al. [7] reviewed algorithms ranging from classical ML (SVM, PCA) to modern detectors, noting that deep CNNs consistently outperform traditional approaches. PCA-based recognition approaches, such as [9], further highlight limitations of pre-deep-learning systems in handling masked faces.

2.3 Feature-Extraction and Classification Pipelines

Feature extraction pipelines remain relevant for applications requiring lower computational complexity compared to large end-to-end detection architectures such as full scale YOLO or ResNet based models. Putri and Athoillah [1] achieved high accuracy with a simple CNN classifier applied to cropped face regions. Kamil et al. [10] used SVM-based classification in an online attendance system, showing the ease of integrating classical ML with browser based frameworks.

Sella Veluswami et al. [11] proposed SSDNET for face detection combined with a lightweight CNN classifier, achieving up to 99% accuracy. Transfer learning based classification using deep feature extractors (MobileNetV2, Inception, ResNet) is further detailed in [8], demonstrating strong generalization to diverse mask types.

Table 2.1: Comparison of object detection based face mask detection approaches

Method	**Detection Paradigm**	**Strengths**	**Limitations**
YOLO (v3–v8)	Single stage detector	High inference speed; suitable for real-time applications; strong accuracy speed trade-off	Performance can degrade for small or heavily occluded faces; often requires GPU for higher variants
SSD (Single Shot Detector)	Single stage detector	Computationally efficient; lower latency; suitable for CPU based or embedded systems	Lower accuracy compared to newer YOLO variants, especially under complex occlusions
Faster R-CNN / Mask R-CNN	Two stage detector	High localization accuracy; robust to scale variation; supports fine grained segmentation	High computational cost; slower inference; unsuitable for real-time or resource constrained deployment

2.4 Segmentation and Fine-Grained Mask Analysis

Segmentation based systems, although computationally heavier, provide detailed pixel level mask face boundary information. Lin et al. [3] demonstrated that combining Mask R-CNN with GIoU loss leads to improved segmentation accuracy, supporting more precise occlusion analysis and mask fit estimation. Such systems are beneficial for medical and industrial settings where mask quality assessment is required.

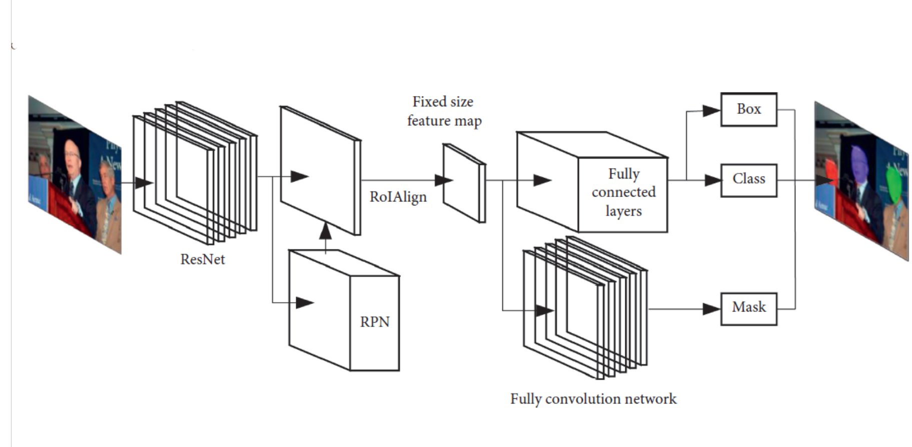

Figure 2.3: Network architecture of the G-Mask, from [3].

However, segmentation based models incur high computational overhead due to components such as region proposal networks, RoI Align operations, and pixel level mask prediction. Lightweight classification and single stage detection approaches reduce this complexity by removing segmentation branches, region proposal stages, and fine grained pixel supervision, retaining only the features necessary for bounding box detection and mask classification. As a result, these models are better suited for real-time inference on CPU only systems.

2.5 IoT, Edge and Browser Based Deployment Frameworks

Edge intelligence and IoT integration are essential for scalable mask detection, particularly in resource constrained environments. For example, Dubey et al. [6] proposed an edge oriented framework in which a hybrid ResNet50–MobileNetV2 model is optimized using a hybrid feature selection optimization (HFSO) algorithm and deployed on low power devices for real-time inference. Such architectures enable localized processing close to the data source, reducing latency and network overhead.

Web based frameworks emphasize accessibility and ease of interaction by leveraging browser based frontends connected to centralized inference services. Kamil et al. [10], for instance, implemented an online attendance and monitoring system where face analysis is

performed through a web interface without requiring local software installation, allowing users to upload images or video streams for server side processing.

At the infrastructure level, several studies focus on supporting large scale development pipelines rather than end user inference. Pham et al. [12] introduced an automated data generation and labeling framework that extracts training samples from live video streams, facilitating rapid dataset expansion and continuous model improvement.

2.6 Datasets, Data Diversity, and Augmentation Techniques

Dataset quality directly impacts the robustness of mask detection systems. PWMFD [13] (Public Mask Wearing Face Dataset) provides three mask-wearing categories, addressing a major gap in improperly worn mask classification. Pham et al. [12] contributed a 7110 image dataset generated from YouTube, demonstrating large-scale automatic labelling.

Rezaei et al. [14] and Hongtao et al. [15] highlight the influence of image quality and detector robustness, noting that low-resolution or noisy surveillance footage significantly degrades model performance.

Data augmentation is widely used to mitigate dataset limitations. Albumentations [16] offers efficient pipelines for geometric and photometric transformations, while Augmentor [17] provides advanced stochastic transformations such as elastic distortions and pipeline chaining. Broader surveys such as [18, 19, 20] emphasize augmentation's role in improving generalization and reducing overfitting across image classification tasks.

2.7 Summary and Research Gaps

Across the literature, face mask detection research has progressed rapidly since the onset of the COVID-19 pandemic (2020–2024), with a substantial increase in the number of deep learning–based approaches, benchmark datasets, and reported accuracy levels. Early works primarily addressed binary mask detection, while more recent studies have explored multi-class classification, real-time inference, and integration with object detection frameworks.

- Many existing studies focus on public surveillance or access control scenarios, with limited emphasis on healthcare and laboratory environments where mask compliance requirements and operating conditions differ significantly.

- A large portion of published work addresses binary mask detection (mask vs. no-mask), while fewer implementations explicitly consider detailed categories such as N95 masks or improperly worn masks, as noted in recent survey studies.

- Although high accuracy is frequently reported, numerous implementations rely on computationally intensive architectures (e.g., deep YOLO or two stage detectors), limiting their suitability for real-time inference on CPU-only systems.
- Privacy preserving considerations, including local only processing and operator controlled monitoring, receive limited attention in many vision based mask detection frameworks.

The present project seeks to address selected aspects of these gaps by focusing on multi class mask classification using a modular two stage pipeline, combining SSD based face localization with a MobileNet classifier. The system is designed for real-time execution in controlled environments, emphasizing classification reliability and demonstration of practical inference workflows.

Chapter 3

Experiment and Methodology

3.1 Overview

This chapter presents the complete experimental pipeline used to develop the real-time face mask detection system, covering all major stages from data preparation to deployment. The workflow begins with dataset construction, including face extraction, and augmentation. A MobileNet based classifier is then designed and trained using the processed dataset, followed by accuracy analysis. Finally, the trained model is integrated into a real-time detection framework that performs continuous face localisation and mask classification using live webcam input. Together, these stages form a cohesive methodology for building an efficient and practical mask detection system.

3.2 System Architecture

3.2.1 Overall Workflow

The preprocessing pipeline begins by scanning the dataset directory structure to enumerate all available image files under the four target classes. Each image undergoes automated face localization using a pre-trained OpenCV DNN-based SSD face detector (ResNet-10 backbone). The detected facial region is then cropped, resized to 224×224 pixels, and stored in a curated directory hierarchy under `processed/faces`. 224×224 pixels corresponds to the standard input resolution expected by ImageNet pretrained MobileNet models, ensuring architectural compatibility and effective transfer learning. This modular step ensures consistency in face extraction before feeding data to the model training pipeline. The workflow integrates the modules `common.py` and `preprocessing.py`, forming the backbone of the data preparation stage in the overall architecture. Following preprocessing, copying images into `processed/{Train,Val,Test}/<class>/`. This keeps the raw curated pool of images immutable, stored in separate directories (`train_dir`, `val_dir`, `test_dir`) and provides stable, reproducible entry points for all subsequent training and evaluation scripts.

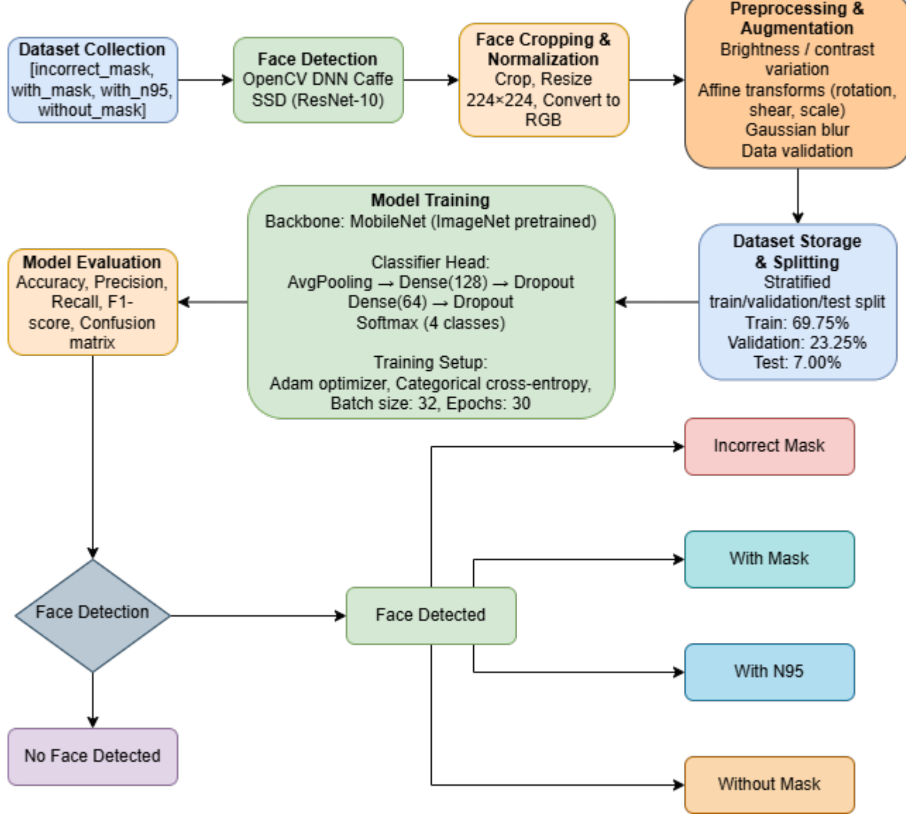

Figure 3.1: Overall Workflow

3.2.2 Hardware Setup

The real-time face mask detection system was implemented and tested on a standard workstation equipped with a Windows 10 64-bit operating system. Laptop's webcam is used for live video capture and frame-by-frame analysis for mask detection.

3.2.3 Software Stack

The implementation is developed in Python and relies on widely adopted computer vision and deep learning libraries to ensure portability and reproducibility. Core dependencies include: OpenCV for image I/O and drawing utilities, NumPy for tensor manipulation, TensorFlow/Keras for model definition and training, scikit-learn for evaluation helpers, Matplotlib for plotting, Albumentations for data augmentation, and TensorFlow Datasets (TFDS) for optional dataset access. The code is modularized via a `common.py` module that centralizes shared constants, paths, and label mappings across the data and training scripts. In addition, the classical OpenCV & Caffe SSD face detector (ResNet-10 backbone) is retained for rapid face localization in baseline experiments.

Rationale for Software Stack Selection:

TensorFlow and Keras is selected as the primary deep learning framework due to its strong support for pretrained model MobileNet, ease of model definition, and seam-

less integration with Python based workflows. Its high level Keras API simplifies rapid prototyping and flexibility for custom model design and deployment.

OpenCV is used for computer vision tasks due to its efficiency in real-time image processing, wide range of builtin functions for image manipulation, and native support for classical models such as the Caffe SSD face detector. Its optimized implementations make it suitable for CPU based inference.

Alternative frameworks such as PyTorch could also be used for deep learning tasks, and libraries such as Dlib or MediaPipe can perform face detection. However, TensorFlow/Keras and OpenCV are chosen due to their mature ecosystem, ease of integration, and strong compatibility with real-time deployment pipelines.

Table 3.1: Software stack used for system implementation and evaluation.

Component	**Library / Tool**	**Purpose**
Programming Language	Python 3.x	Primary implementation language for data preprocessing, model training, evaluation, and deployment.
Computer Vision	OpenCV (cv2)	Image I/O, webcam capture, preprocessing, and face region extraction.
Deep Learning Framework	TensorFlow / Keras	Model definition, training, validation, and inference of the MobileNet based mask classifier.
Face Detection Model	Caffe SSD (ResNet-10)	Pre-trained single shot detector used for robust and efficient face localization.
Data Augmentation	Albumentations	Offline augmentation during preprocessing to improve robustness to lighting, pose, and blur variations.
Training Augmentation	ImageDataGenerator	On-the-fly data augmentation during training to enhance generalization and reduce overfitting.
Numerical Computing	NumPy	Efficient array manipulation and tensor preprocessing.
Visualization	Matplotlib	Plotting of confusion matrix heatmaps for analysis.
Browser Backend	FastAPI	Lightweight REST based backend for browser accessible real-time inference.
Browser Frontend	HTML, JavaScript	Browser based user interface for webcam access, frame capture, and result visualization.

Key Libraries:

- `opencv-python`, `numpy`, `matplotlib`
- `tensorflow/keras` (MobileNet, VGG19 backbones; preprocessing)

- `scikit-learn` (metrics/utilities), `albumentations` (augmentation)
- `Pillow` (robust image loading), `imageio` (I/O convenience)

Project Paths: The project root for data is configured via:

```
image_path = C:/../../source
dataset_root = <image_path>/dataset
processed_root = <image_path>/processed
```

This separation preserves raw images under `dataset` and any curated or augmented artifacts under `processed`. All further training and evaluation scripts import these paths from `common.py`, avoiding duplication and path changes across experiments.

3.3 Dataset Preparation

3.3.1 Dataset Sources and Class Schema

The dataset was collected from publicly available face mask image repositories on Kaggle. The final curated dataset consists of 14,857 images distributed across the four defined classes.

The mask wearing state is modeled as a four class classification problem aligned with public safety monitoring needs. The label set is defined once in a shared module `common.py` to avoid inconsistencies across preprocessing and training scripts:

$$\texttt{CLASSES} = \{\texttt{incorrect_mask}, \texttt{with_mask}, \texttt{with_n95}, \texttt{without_mask}\}.$$

Retained two positive with mask categories to distinguish between generic masks and N95 types, while explicitly capturing improperly worn masks and without mask cases as separate safety critical states.

Label	Index
incorrect_mask	0
with_mask	1
with_N95	2
without_mask	3

Table 3.2: Class Index Mapping Used Across Data Loaders and Model Head.

The dataset is structured into four distinct categories representing different mask-wearing conditions. These classes are standardized across preprocessing, training, and inference scripts to maintain consistency in label encoding and evaluation.

3.3.2 Data Preprocessing

To ensure compatibility with the chosen backbones (MobileNet and VGG19), the canonical input resolution is set to:

$$\texttt{TARGET} = 224 \times 224.$$

Although 224×224 may appear small compared to full resolution images, this input size is standard for ImageNet pretrained convolutional backbones such as MobileNet and VGG19. In the present pipeline, the classifier operates on tightly cropped facial regions rather than full scene images, ensuring that mask related features occupy a significant portion of the input. This resolution provides sufficient spatial detail to distinguish mask types and wearing conditions while maintaining computational efficiency, which is critical for real-time inference on CPU based systems.

Using smaller resolutions (e.g., 128×128) may lead to loss of fine grained mask features, while **larger resolutions** (e.g., 512×512 or higher) increase computational cost without significant performance improvement.

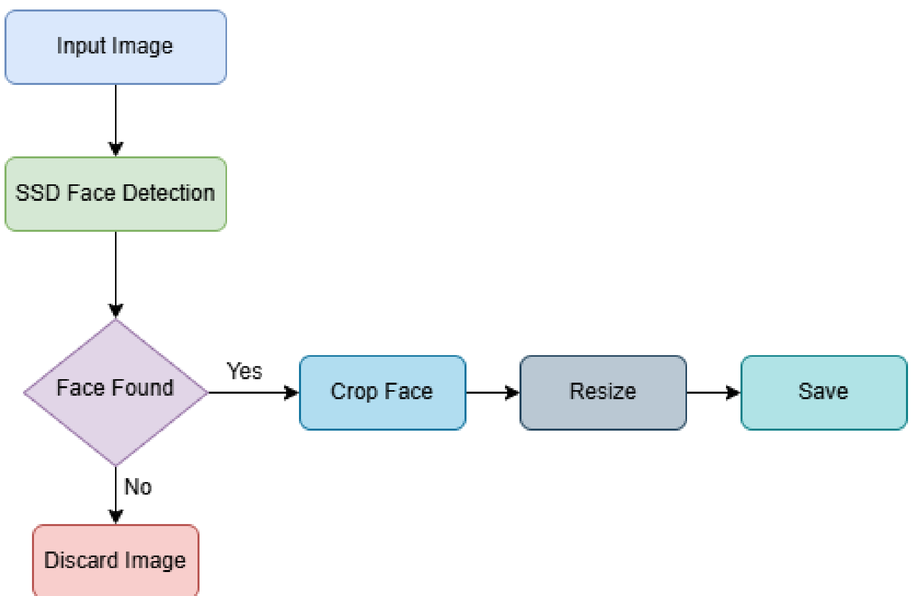

Figure 3.2: Data Preprocessing

Raw image samples are first collected from the structured folder tree defined in `common.py`. The script traverses each class folder recursively, registering image paths and labels. Before training, a preprocessing routine executes the following:

1. **Face Detection:** Each image is processed by the Caffe-based SSD model to detect facial regions.

2. **Cropping and Resizing:** The face with the highest confidence is cropped, resized to 224×224, and converted to RGB.

3. **Saving Curated Data:** Cropped faces are stored under `processed/faces/<class>/` maintaining the original label hierarchy.

4. **Error Handling:** Faulty or unreadable images are automatically skipped to preserve data quality.

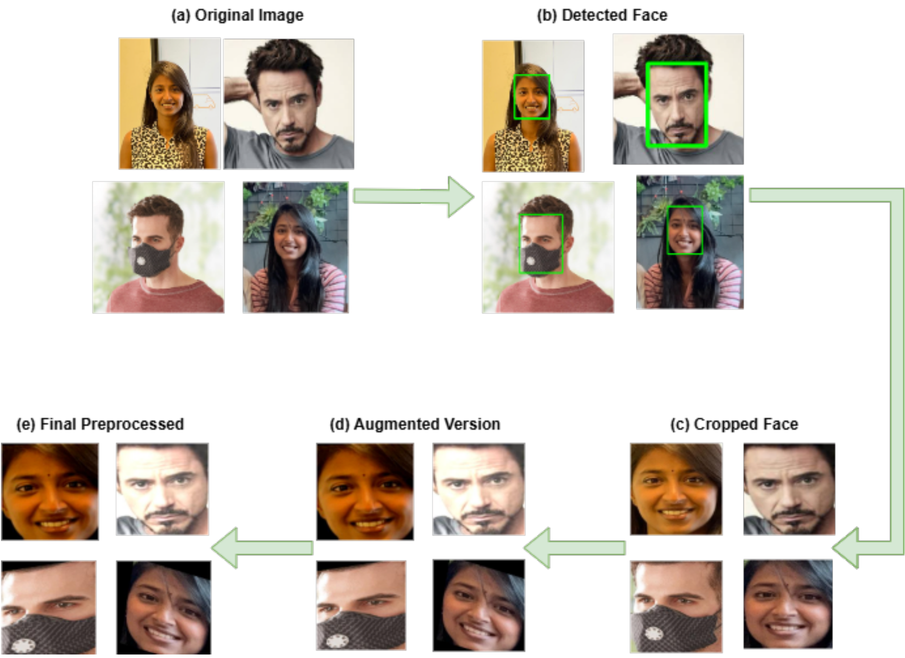

Figure 3.3: Face Detection and Preprocessing Stages

3.3.3 Data Augmentation

To improve generalization and reduce overfitting, data augmentation is applied during both the preprocessing stage and the model training stage. Augmentation increases the effective size and diversity of the training data by introducing controlled variations that simulate real-world conditions such as lighting, orientation, blur, and slight geometric distortions.

Albumentations based Augmentation (Preprocessing Stage). During face extraction, each cropped face passes through a stochastic augmentation pipeline implemented with the `Albumentations` library. The configured transformations include:

- **Random Brightness Contrast** to simulate varied illumination.
- **Affine transformations** (scale, translation, rotation, shear) to model pose differences.
- **Gaussian blur** to emulate motion blur or low-quality webcam capture.

This augmentation occurs for approximately 90% of curated samples, ensuring that the model encounters visually perturbed instances even before formal training begins. This step is particularly useful for improving robustness in real-time detection, where lighting and orientation vary continuously.

Keras `ImageDataGenerator` Augmentation (Training Stage). A second layer of augmentation is applied dynamically during model training via `ImageDataGenerator`. The following transformations are enabled:

- **Horizontal flipping** to account for left/right symmetry.
- **Zoom and shear transformations** to simulate camera perspective variations.
- **Brightness range adjustments** to strengthen illumination invariance.
- **Rotation** to handle mild non-frontal head orientations. In-plane rotation to improve robustness to minor head tilt and camera misalignment; this augmentation does not simulate true out-of-plane (yaw) head rotations.

These augmentations are applied on-the-fly to each training batch, such that the model is exposed to slightly different variations of the same images across epochs. This strategy is commonly used to encourage invariance to nuisance factors such as minor illumination changes, camera alignment, and pose variations. The combined use of Albumentations during preprocessing and ImageDataGenerator during training is therefore intended to improve generalization and reduce overfitting, particularly in real-time webcam based inference scenarios.

3.3.4 Data Splitting

The script reconstructs the dataset by scanning `<image_path>/processed/faces/<class>/` for common image extensions (`.jpg/.jpeg/.png/.bmp/.webp/.tif/.tiff`). The resulting lists {`face_dir`, `face_labels`} are split using `train_test_split` with **stratification** to preserve class priors and a fixed seed (`random_state=42`) to ensure reproducibility.
Configured split ratios

- Training Subset/Test: 93% / 7%
- Train/Val (from the 93% training subset): 75% for train / 25% for validation

After removing duplicate and low-quality samples, a total of **14546 curated face images** remained across all classes.

- Small dataset - Need to use more training data (e.g., 90/10 or 93/7)
- Large dataset - Can afford larger test split (e.g., 80/20)

- This work (14,546 images) - 93/7 chosen to maximize training while keeping 1019 test samples for reliable evaluation

These values reflect the exact stratified distribution used during training and final evaluation.

Curated files are then copied into a clean directory tree: `<image_path>/processed/{Train,Val,Test}/<class>/` which decouples model I/O from the raw curation folder and prevents leakage across folds which means that no image or augmented variant is shared between the training, validation, and test subsets.

Table 3.3: Table 3.2 Dataset Split Configuration and Output Directories.

Subset	Count	Proportion	Stratified	Destination Pattern
Train	10145	69.75%	Yes	`processed/Train/<class>/`
Validation	3382	23.25%	Yes	`processed/Val/<class>/`
Test	1019	7.00%	Yes	`processed/Test/<class>/`

Integrity validation. Before training, the script verifies decodability of every image using `PIL.Image.open`, removing unreadable or corrupted files and deleting their paths where possible. This prevents training-time decoder failures and ensures that the exported roots `train_dir`, `val_dir`, `test_dir` maintain consistent, non-overlapping datasets.

3.4 Model Architecture

3.4.1 Backbone and Preprocessing Interfaces

The codebase imports the Keras MobileNet backbone alongside its preprocessing utilities. MobileNet models are emphasized for their superior latency accuracy tradeoff in edge or real-time contexts. The shared module exposes the input size and preprocessing handle (`mobilenet_v2.preprocess_input`) to prevent configuration drift across training and inference scripts.

3.4.2 Base Model Selection

Adopted **MobileNet** (`include_top=False`, ImageNet weights) as the feature extractor owing to its favorable latency accuracy tradeoff for real-time applications and its depthwise separable convolutions that reduce parameter count and FLOPs while retaining strong representation capacity for face centric tasks. Inputs are standardized to $224 \times 224 \times 3$.

Table 3.4: Progressive Feature Transformation in MobileNet

Stage	What happens	Size Example
Input	Original image	$224 \times 224 \times 3$
Convolutional layers	Detect edges/textures	—
Downsampling	Reduce spatial size	112×112
More layers	Detect shapes	—
More downsampling	Further reduction	56×56
Continue	Higher-level features	28×28
Continue	Deeper abstraction	14×14
Final feature maps	Deep learned features	$\mathbf{7 \times 7 \times 1024}$

3.4.3 Model Modification

The backbone is frozen (`layer.trainable=False` for all layers) to preserve pretrained features during the first training phase. On top of the convolutional trunk, added a lightweight classifier head:

- `AveragePooling2D(pool_size=(7,7))` to spatially aggregate features;
- `Flatten()` to vectorize pooled features;
- `Dense(128, ReLU)` → `Dropout(0.3)`;
- `Dense(64, ReLU)` → `Dropout(0.3)`;
- `Dense(4, Softmax)` for the four classes.

This head balances capacity and generalization with moderate regularization (two dropout layers at 0.3).

3.4.4 Transformation from Input Image to Classification

An input image of size $224 \times 224 \times 3$ represents a standard RGB image, where 224×224 denotes the spatial resolution and 3 color Red, Green, Blue channels. At this stage, the image is represented in raw pixel form.

Feature Extraction using MobileNet

As the input image passes through the MobileNet backbone, it undergoes a series of convolutional and downsampling operations. These operations progressively reduce the spatial resolution while extracting abstract and meaningful features such as edges, textures, and patterns relevant to face and mask.

The final output of the MobileNet backbone is a tensor of size $7 \times 7 \times 1024$. This representation no longer corresponds to raw pixel values. Instead, it encodes high level features learned by the network in various stages.

Interpretation of $7 \times 7 \times 1024$

The 7×7 dimension represents a spatial grid, where each cell corresponds to a region of the original image. However, unlike pixels, each location in this grid contains a feature vector of length 1024, capturing the presence and strength of various learned patterns such as mask edges, facial contours, and texture variations.

Thus, each spatial location can be expressed as:

$$Cell = [f_1, f_2, \ldots, f_{1024}]$$

Classifier Head Processing (3D to 1D Transformation)

The classifier head converts this 3D feature representation into a 1D vector suitable for classification:

- **Average Pooling:** Each 7×7 feature map is reduced to a single value by averaging spatial information. This transforms the tensor from $7 \times 7 \times 1024$ to $1 \times 1 \times 1024$, by summarizing all the feature presence.
- **Flatten:** The $1 \times 1 \times 1024$ tensor is reshaped into a 1D vector of size 1024.
- **Dense Layers:** Fully connected layers learn combinations of these features to distinguish between different mask categories.
- **Softmax Output:** The final layer produces a probability distribution over the four classes: `incorrect_mask`, `with_mask`, `with_n95`, and `without_mask`. For example:

  $$[0.02,\ 0.85,\ 0.05,\ 0.08]$$

  indicates a high probability for the `with_mask` class.

3.4.5 Hyperparameters

Used `Adam` (default settings) with `categorical_crossentropy` and report `accuracy`. Training is conducted for **30 epochs** with a **batch size of 32**. The choice of 30 epochs serves as a practical upper bound to allow sufficient learning of feature representations. In transfer learning settings, pretrained convolutional networks typically converge within a limited number of epochs when fine tuned on moderately sized datasets [4, 6, 8]. Early stopping with patience values between 1–7 is employed to prevent overfitting and terminate training once validation performance stabilizes, ensuring efficient convergence.

3.5 Face Detection Baseline

For baseline face localization, included the OpenCV DNN Caffe SSD with a ResNet-10 backbone, referenced by:

```
caffe_model = <image_path>/../res10_300x300_ssd_iter_140000.caffemodel
dep_prototxt = <image_path>/../deploy.prototxt
```

Although single stage detectors can directly produce face and mask states, this classical detector serves two purposes: (i) rapid prototyping of a two-stage pipeline (face detection - mask classification), and (ii) a consistent baseline to quantify the latency/accuracy benefits when migrating to integrated detection heads. Subsequent sections report FPS and precision trade-offs for both approaches.

3.6 Evaluation Methodology

Model evaluation follows a two-stage process: (i) built-in Keras metrics computed during training and validation, and (ii) a manual post training analysis performed on the held-out test set. This ensures that both aggregate and class specific behaviours are captured.

3.6.1 Evaluation Metrics

The following metrics are used to assess model performance:

- **Accuracy** overall proportion of correctly classified samples.
- **Precision, Recall, and F1-score** computed per class to quantify error types.
- **Confusion matrix** summarises classwise prediction patterns.

Although accuracy is reported during training, the full metric suite is computed after training using manually derived TP, FP, TN, and FN counts.

3.6.2 Validation and Testing

Generalization performance is assessed on the held-out test set using:

```
model.evaluate(test_generator)
```

This reports the final test loss and accuracy after training. For deeper analysis, predictions on the entire test set are used to compute the confusion matrix and per-class metrics. The final trained model is exported for downstream real-time deployment as:

```
model.save('data/saved_model.h5')
```

This file is later loaded by the real-time inference modules to perform continuous mask classification.

3.7 Real-Time Implementation

3.7.1 System Integration

The real-time system integrates three primary components developed earlier in the project:

(a) Face Detection Module (Caffe SSD)

A lightweight Single Shot Detector (SSD) with a ResNet-10 backbone is loaded using:

```
cv2.dnn.readNetFromCaffe(prototxt, caffemodel).
```

This model provides reliable face localisation with minimal CPU load, suitable for real-time inference.

(b) MobileNet Based Mask Classifier

The trained model `saved_model.h5`, which contains the MobileNet backbone and custom dense layers, is imported using:

```
model = load_model(trained_model).
```

It outputs probabilities for four classes: {*incorrect_mask*, *with_mask*, *with_n95*, *without_mask*}.

(c) Capture and Display Module

Frames are continuously captured using OpenCV's `VideoCapture(0)`, annotated using class-specific colors from `common.py`, and displayed in real time.

3.7.2 Frame Processing Pipeline

Each frame ($H \times W \times 3,\ BGR$) from `VideoCapture(0)` is processed as follows:

1. **Face localisation (OpenCV DNN SSD).** A blob is built with mean subtraction and 300×300 resizing:

   $$\texttt{blobFromImage}(I_{BGR},\ 1.0,\ (300, 300),\ (104, 177, 123)).$$

   The detector outputs boxes (x_1, y_1, x_2, y_2) and confidences; boxes with $\texttt{CONF_THRESH} = 0.5$ are kept. Coordinates are scaled by $[W, H, W, H]$ and clipped to the image bounds.

2. **Crop & preprocessing.** Each face ROI is cropped and resized to `IMG_SIZE` = $(224, 224)$. For this deployment, pixel values are scaled to $[0, 1]$ by division with 255, matching the training generators.

3. **Classification.** The trained MobileNet-based model (`saved_model.h5`) predicts class probabilities over {*incorrect_mask*, *with_mask*, *with_n95*, *without_mask*}. The argmax yields the label and confidence.

4. **Visualization.** For each detection, a color-coded rectangle, label text and confidence percentage are rendered on the frame.

3.7.3 Prototype Validation Using Capture-and-Predict Script

Before implementing continuous real-time monitoring, a single-frame inference module (`capture_and_predict.py`) was used for controlled testing. The script allows the user to:

- Press `c` to capture a frame,
- Apply MobileNet preprocessing,
- Classify the captured image,
- Print full probability distributions,
- Save the frame for inspection.

This stage validated preprocessing correctness, class ordering, prediction consistency, and the model's behaviour on real-world samples.

3.7.4 Final Real-Time Continuous Detection System

After validating single-frame inference through the prototype, the final deployment module was implemented to enable **continuous real-time monitoring** using a live webcam stream. This system extends the same trained `saved_model.h5` architecture and preprocessing pipeline but introduces automated frame capture, face localization, and overlay visualization for every frame in the video feed.

Each incoming frame from the webcam is passed through the following stages:

1. **Frame Acquisition:** Continuous frame capture from the default camera using OpenCV's `VideoCapture()` API.

2. **Face Detection:** Real-time localization of faces using the SSD-based Caffe model integrated via `cv2.dnn`.

3. **Preprocessing:** Cropping and resizing to 224×224, RGB normalization, and scaling to $[-1, 1]$ as required by MobileNetV2.

4. **Prediction:** The trained deep network performs classification among the four mask states — *with_mask*, *without_mask*, *with_n95*, and *incorrect_mask*.

5. **Visualization:** Bounding boxes and labels are rendered on each detected face, color-coded using the scheme defined in `common.py`.

To maintain responsiveness, inference is performed asynchronously, and frame updates are buffered to avoid latency. The average processing time per frame was observed to remain within real-time constraints (20–30 frames per second) on a standard CPU-based system.

This final module represents the fully operational **end-to-end real-time face mask detection system** capable of continuous public safety monitoring using live video input. It combines preprocessing, deep learning inference, and dynamic visualization into a unified real-time framework.

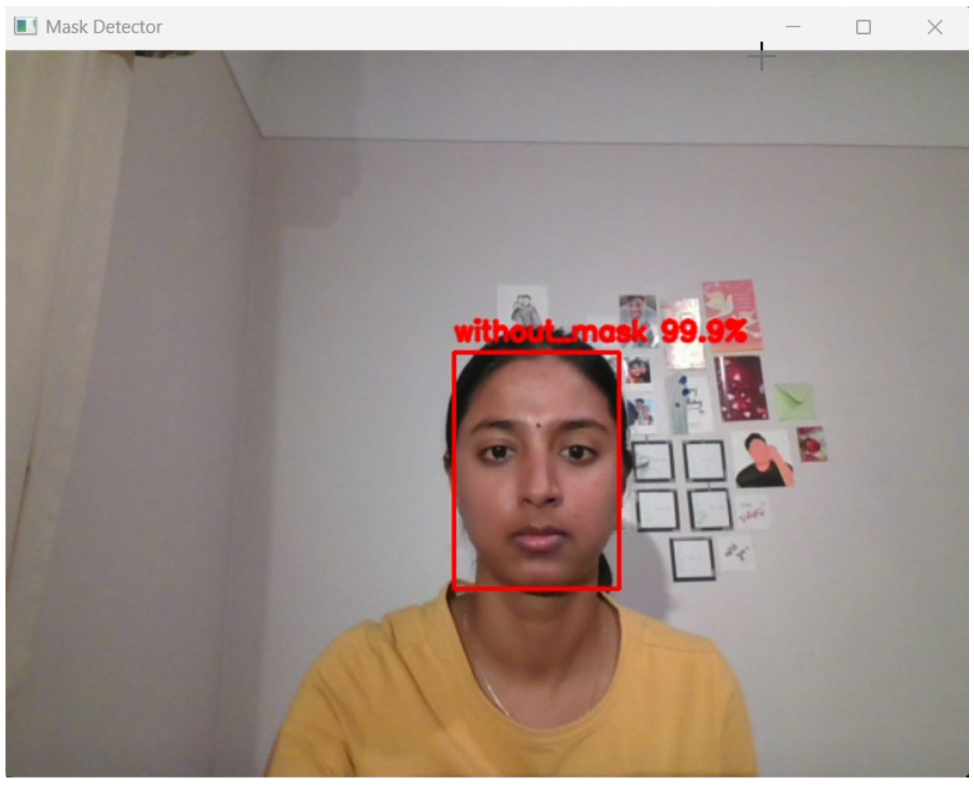

Figure 3.4: Evaluation - Without Mask

3.7.5 Latency & Performance Analysis

Measured end-to-end frame time by timestamping before face detection and after annotation. Reported metrics include mean FPS, median per-frame latency, and the 95th percentile:

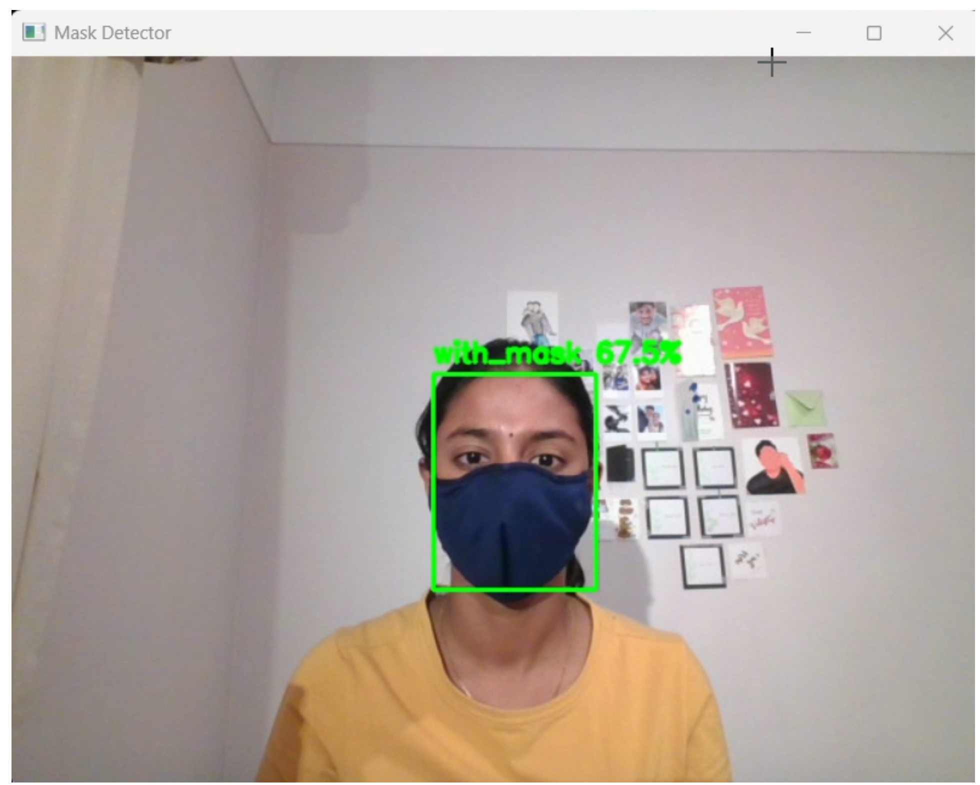

Figure 3.5: Evaluation - With Mask

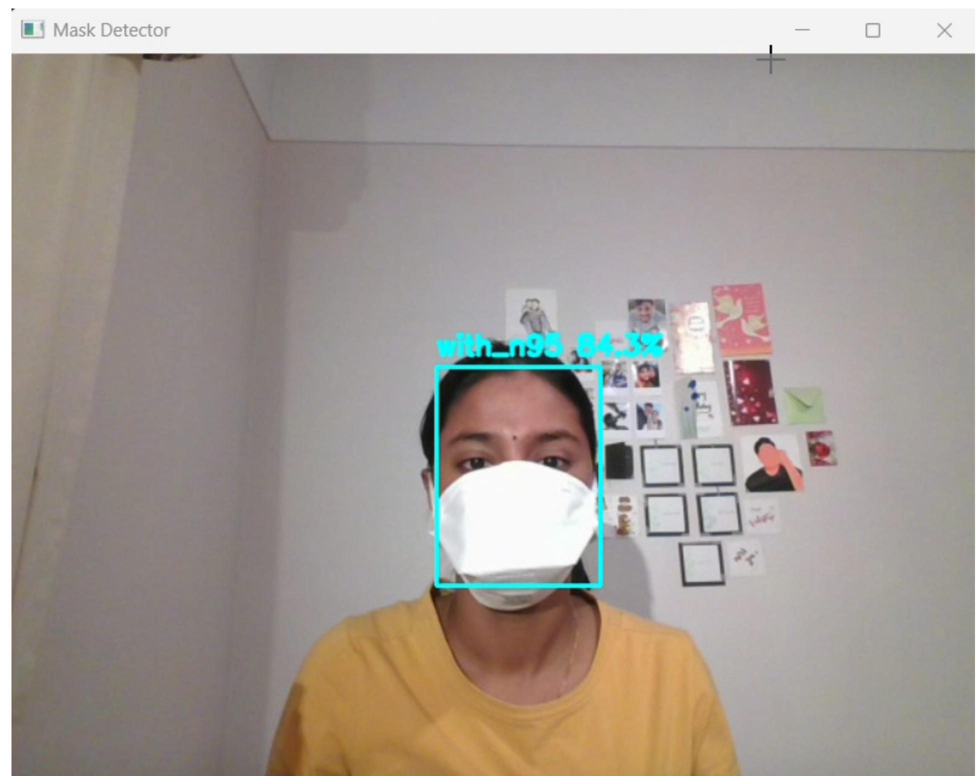

Figure 3.6: Evaluation - With N95 Mask

- **Mean FPS:** __ (CPU-only) **Median latency:** __ ms **P95:** __ ms.

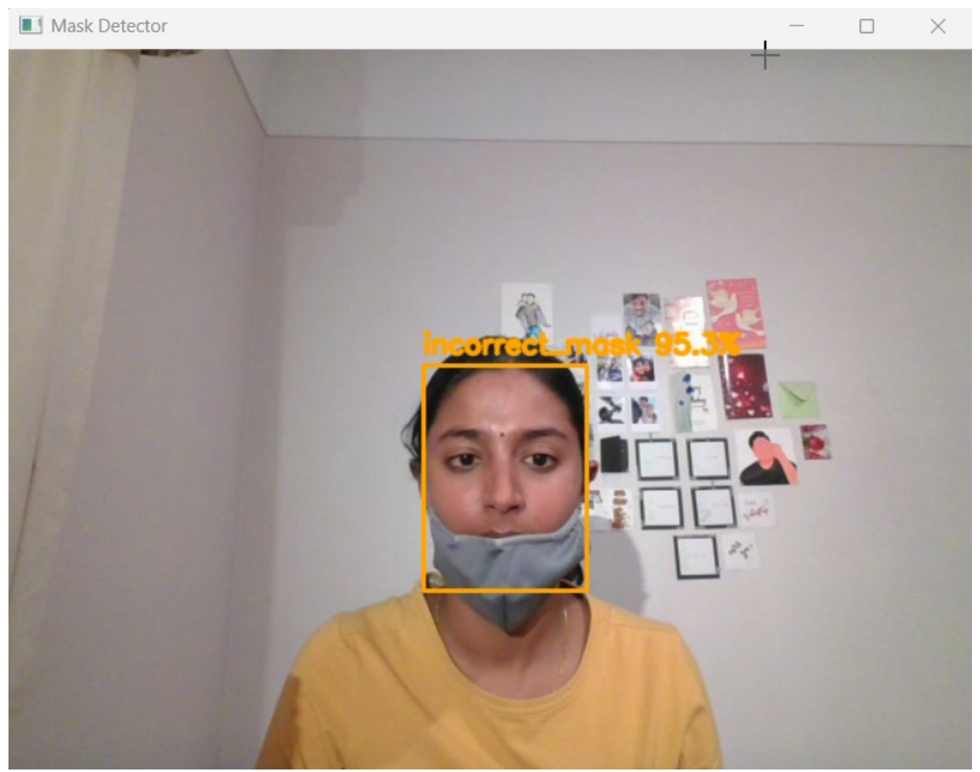

Figure 3.7: Evaluation - Incorrect Mask Over Mouth

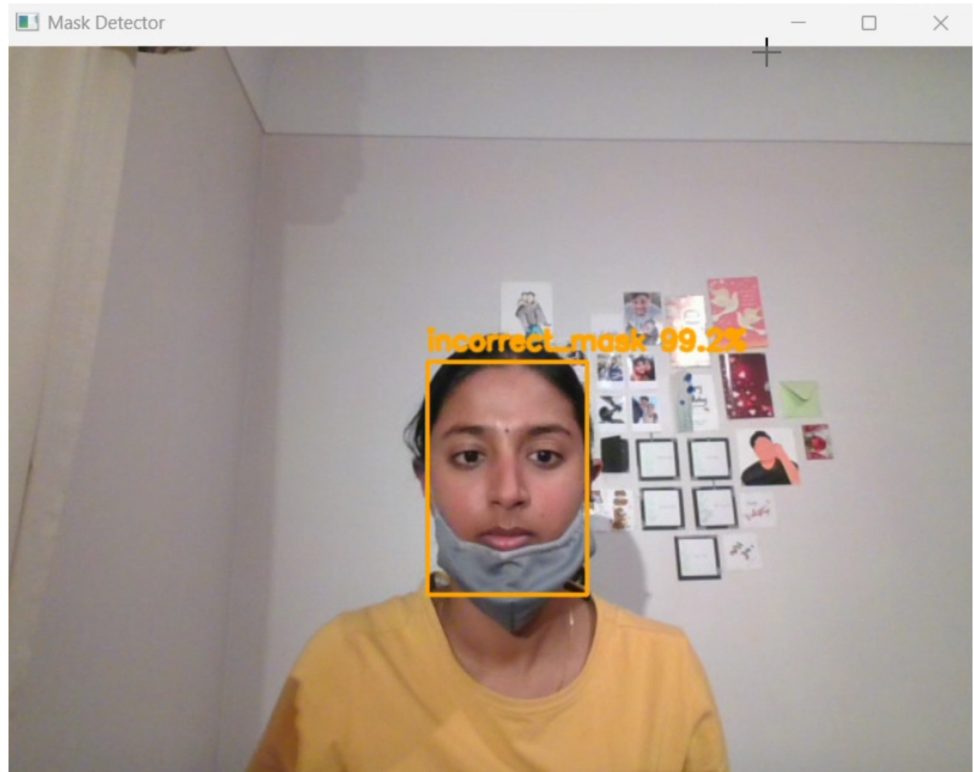

Figure 3.8: Evaluation - Incorrect Mask Over Chin

- **Per-stage breakdown:** face detection __ ms; preprocessing __ ms; classification __ ms; draw __ ms.

A sustained $\geq$ 20–30 FPS indicates real-time operation on the target hardware.

3.7.6 Frame Annotation Conventions

For interpretability in real-time demos, each predicted class is associated with a distinct BGR color for bounding boxes and labels:

Class	BGR Color
incorrect_mask	(0, 165, 255)
with_mask	(0, 255, 0)
with_n95	(255, 255, 0)
without_mask	(0, 0, 255)

Table 3.5: Visualization color codes used in real-time overlays (OpenCV BGR).

These assignments (orange/green/yellow-cyan/red) were chosen to align with intuitive risk signaling: green for correct mask usage; red for non-compliance; orange for improper wear; and a distinct yellowish tone for N95 identification. The same palette is reused in evaluation plots to maintain visual consistency between offline metrics and live demos.

3.7.7 Browser Based Deployment

In addition to the Python based desktop prototype, the system was extended into a lightweight Browser based application to enable real-time mask detection. This deployment demonstrates that the trained model can be exposed as a network service and accessed from commodity client devices without requiring local Python or GPU installations.

Backend Architecture

The backend is implemented using the FastAPI framework and is responsible for model loading, frame processing, and prediction. At server startup, the Caffe-based SSD face detector and the Keras classifier are loaded once into memory:

- **Face detector:** OpenCV DNN SSD with a ResNet-10 backbone, initialized via `readNetFromCaffe(dep_prototxt, caffe_model)`.
- **Mask classifier:** The trained MobileNet-based model loaded from `saved_model.h5` using `load_model()`.

A global image size of `IMG_SIZE` $= (224, 224)$ and confidence threshold `CONF_THRESH` $= 0.5$ are defined, matching the desktop inference pipeline.

The FastAPI application exposes two routes:

- `GET /`: Serves the main HTML page via Jinja2 templates (`index.html`), along with static JavaScript (`app.js`) and CSS resources.

- `POST /predict`: Accepts a single uploaded frame (multipart `file` field), decodes the image using `cv2.imdecode`, and passes the resulting BGR frame to the `face_mask_detector()` function.

Within `face_mask_detector()`, the backend performs the same processing steps as local evaluation:

1. Build a 300×300 blob with mean subtraction $(104, 177, 123)$ for the SSD detector.
2. Run forward inference to obtain face bounding boxes and confidences.
3. Filter detections by `CONF_THRESH`, clip and validate box coordinates.
4. Crop each face ROI, convert BGR to RGB, resize to 224×224, and rescale pixels to $[0, 1]$.
5. Run the classifier to obtain class probabilities, select the argmax label, and overlay a color-coded bounding box and text label (class name and confidence).

After annotation, the processed frame is JPEG-encoded using `cv2.imencode(".jpg", ...)`. The resulting bytes are converted to a hexadecimal string and returned to the client in a JSON response:

$$\{\ \texttt{"image"} : hex - encodedJPEG,\ \ \texttt{"rtt_ms"} : processingtime\ \}.$$

The `rtt_ms` field is measured using `time.perf_counter()` and provides an approximate per-request latency for performance analysis.

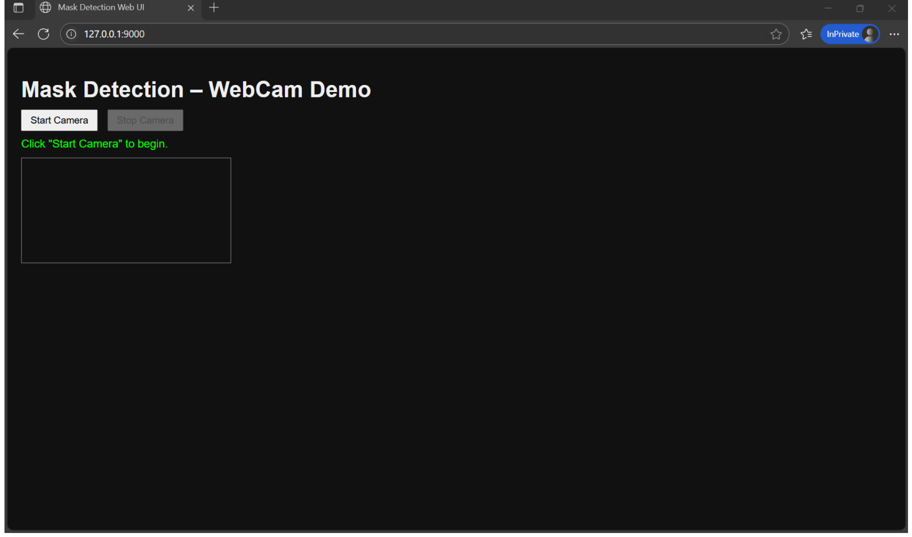

Figure 3.9: Browser based mask detection interface rendered in a browser.

Frontend Architecture

The frontend is implemented as a minimal HTML/JavaScript client that runs entirely in the browser. The core elements are:

- An HTML5 <video> element that receives the raw webcam stream.
- An HTML5 <canvas> element used both for capturing frames and displaying annotated results.
- Two control buttons (*Start Camera* and *Stop Camera*) and a status text label.

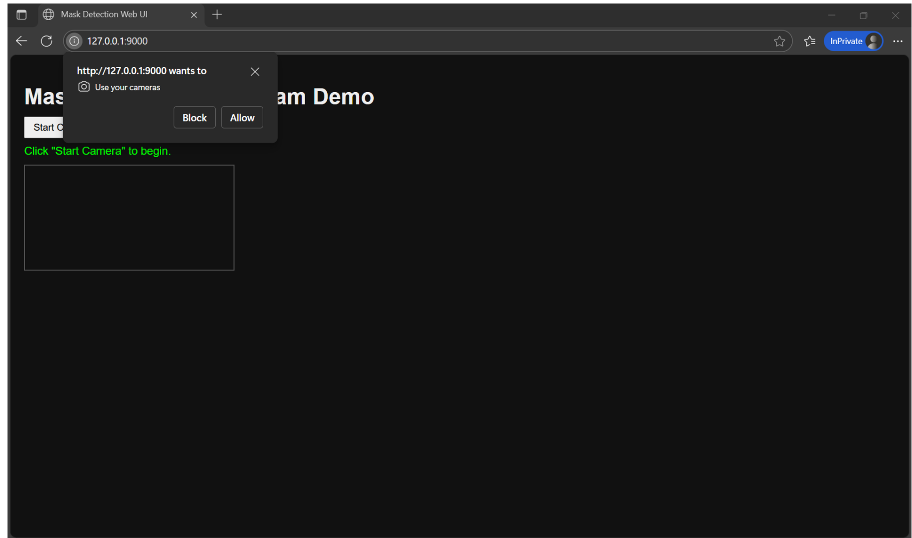

Figure 3.10: Browser camera access prompt for the Browser based mask detection client.

Upon clicking *Start Camera*, the browser requests access to the user's webcam via the navigator.mediaDevices.getUserMedia API, with a target resolution of 640×480. If the user grants permission, the resulting MediaStream is bound to the <video> element. The raw video is not shown directly (the <video> tag is hidden in CSS); instead, the script periodically copies frames onto the canvas for processing.

A JavaScript timer (setInterval) triggers the sendFrame() function every 500 ms:

1. The current video frame is drawn onto the canvas.
2. The canvas content is encoded as a JPEG blob via canvas.toBlob().
3. The JPEG blob is sent as a multipart POST request to /predict.
4. The JSON response is decoded: the hex string is converted back to a Uint8Array, wrapped in a Blob, and loaded as an Image object.
5. Once the annotated image loads, it is rendered back onto the same canvas, replacing the original frame.

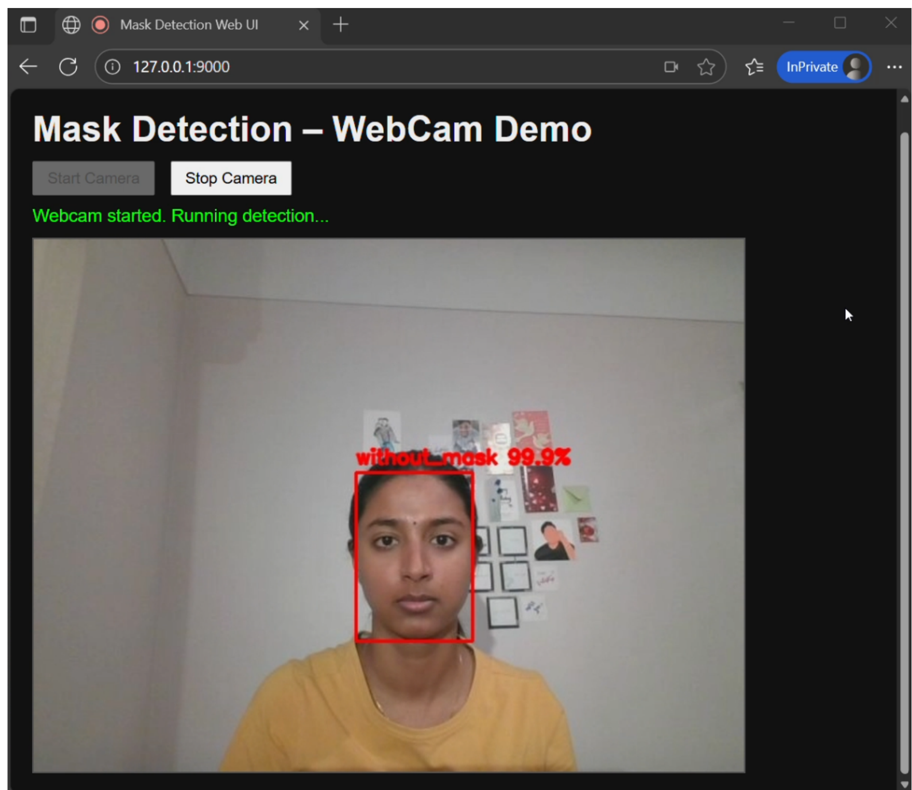

Figure 3.11: Real-time continuous mask detection and classification result - Without Mask

The *Stop Camera* button cancels the detection timer, stops all tracks in the `MediaStream`, and clears the canvas. No automatic startup is performed, ensuring that webcam capture only begins after explicit user interaction.

End-to-End Workflow

The end-to-end Browser based workflow can be summarized as:

1. The user navigates to the Browser based interface and clicks *Start Camera*.
2. The browser requests and, upon approval, starts the webcam stream.
3. Every 500 ms, the current frame is captured from the video element, encoded, and sent to the FastAPI backend.
4. The backend performs face detection and mask classification, annotates the frame, and returns the processed image in a JSON payload.
5. The frontend decodes the returned image and renders the annotated frame onto the canvas, providing real-time visual feedback.

This loop continues until the user explicitly clicks *Stop Camera*, after which both the detection interval and the webcam stream are terminated.

Purpose, Performance, and Significance

The Browser based extension demonstrates that the proposed system can be deployed as a thin client application, requiring only a modern browser on the user side. Key properties include:

- **Platform independence:** No Python, CUDA, or local model installation is required on the client; all inference is performed on the server.
- **Real-time responsiveness:** With a capture interval of 500 ms and typical per-frame processing times on the order of tens of milliseconds (as tracked by `rtt_ms`), the system achieves interactive frame rates suitable for basic monitoring.
- **Privacy-aware interaction:** Frames are processed on-the-fly and returned as annotated images; the client interface does not auto-start the camera and requires explicit user consent, aligning with privacy expectations in institutional environments.
- **Ease of integration:** The REST style `/predict` endpoint and simple JSON based response format decouple the inference backend from the client interface, in contrast to tightly coupled desktop applications or hardware specific edge deployments that require direct integration with device drivers or proprietary SDKs.

By providing a simple browser accessible deployment path in addition to the desktop prototype, the system illustrates its practical applicability to scenarios such as entrance screening stations, laboratory compliance monitoring, or temporary event checkpoints.

3.8 Limitations and Challenges

Although the proposed system demonstrates strong accuracy and stable real-time performance in controlled settings, several limitations were encountered during development, evaluation, and deployment.

Dataset and Model Limitations

The most significant challenge is the **dataset imbalance**. A large portion of the dataset consists of Asian faces, with comparatively fewer samples from other demographic groups. This imbalance may introduce unintended bias and reduce the model's ability to generalise across diverse populations. In addition, the `with_n95` category contains substantially fewer samples than the remaining classes, which contributed to lower recall during testing. The `incorrect_mask` class also remains difficult due to its wide intra-class variability, as improper mask usage can appear in many subtle and unstructured forms.

Furthermore, the `with_n95` class in the current dataset is predominantly composed of beak shaped N95 masks (characterised by a central vertical ridge). As a result, the model may exhibit limited generalisation to other N95 mask variants, such as cup shaped or flat fold designs, which differ in geometry and surface structure.

Operational Limitations in Real-Time Desktop Inference

During desktop based real-time testing, factors such as **lighting variation**, shadows, partial occlusions, and rapid head movements occasionally reduced the reliability of face localisation. Although the SSD based face detector performs adequately on standard CPUs, its computational cost can still limit throughput on resource constrained environments. Running the pipeline on minimal edge hardware would likely result in reduced frame rates unless a more lightweight detector is adopted.

GPU Dependency

The current implementation shows performance on a CPU without utilizing GPU acceleration. While GPU based execution could significantly improve inference speed and throughput, it was not adopted in this work to maintain system portability and ensure compatibility with standard desktop and edge devices where dedicated GPUs may not be available. Future implementations may leverage GPU acceleration for improved scalability and performance.

Limitations Observed in the Browser Based Client–Server Interface

The browser-integrated client-server interface introduces additional overhead due to frame capture, encoding, and backend inference. Because each frame must be:

1. Captured from the webcam,
2. Encoded as a JPEG blob on the client,
3. Sent via an HTTP request to the FastAPI backend,
4. Processed by the face detector and mask classifier,
5. Re-encoded as an annotated JPEG,
6. Returned to the browser for visualization.

When the browser and backend execute on the same system, overall responsiveness is primarily influenced by JPEG encoding/decoding overhead and backend processing time. In alternative deployment configurations where the browser and backend are hosted on

separate machines within a local network, network latency and Wi-Fi fluctuations may introduce additional delays, particularly in shared or congested environments. While the system operates smoothly with a 500 ms capture interval, these factors can reduce visual clarity for fast-moving subjects.

The FastAPI backend currently performs inference on a single CPU thread, which may limit scalability when multiple clients connect concurrently.

Summary of Challenges

These limitations highlight opportunities for:

- Expanding and balancing the dataset across demographics,
- Improving the robustness of face detection under uncontrolled conditions,
- Adopting lighter or hardware-accelerated detectors for edge deployment,
- Exploring browser-side inference (e.g., WebAssembly or TensorFlow.js) to reduce backend load and eliminate client–server transfer overhead in distributed deployments,
- Incorporating buffering or adaptive frame skipping for smoother Browser based performance.

Overall, the system is functional and accurate but exhibits several practical constraints that motivate future optimisation and extension.

Chapter 4

Results

4.1 Accuracy Evaluation

While Keras provides built-in metrics during training and validation, these aggregated values do not reveal class-specific behaviour or patterns in misclassification. To gain a deeper understanding of the model's performance, a **accuracy evaluation** was conducted using detailed predictions on the held-out test dataset. This evaluation incorporates the confusion matrix, class-wise accuracy, precision, recall, F1-scores, and an independently computed overall accuracy. Such analysis is essential for multi-class classification problems, particularly for safety-critical applications where different classes may have different operational significance.

4.1.1 Evaluation Across EarlyStopping Patience Values

To analyse the effect of the EarlyStopping hyperparameter on model stability and generalisation, the patience value was varied from $p = 1$ to $p = 7$, while keeping all other hyperparameters fixed. Each model was trained for a maximum of 30 epochs and terminated early when the validation loss did not improve for p consecutive epochs. The following tables summarise per-class precision, recall, F1-score, and accuracy for each patience configuration, along with the corresponding overall test-set accuracy.

4.1.2 Confusion Matrix Visualisation Across Patience Values

To complement the numeric metrics, Figures 4.1–4.7 show confusion matrix heatmaps for each EarlyStopping patience value ($p = 1 \ldots 7$).

In this context, the term *heatmap* refers to a color coded visualization in which color intensity represents the magnitude of values in the confusion matrix, rather than any physical temperature.

Example (with_mask class):

From the confusion matrix,

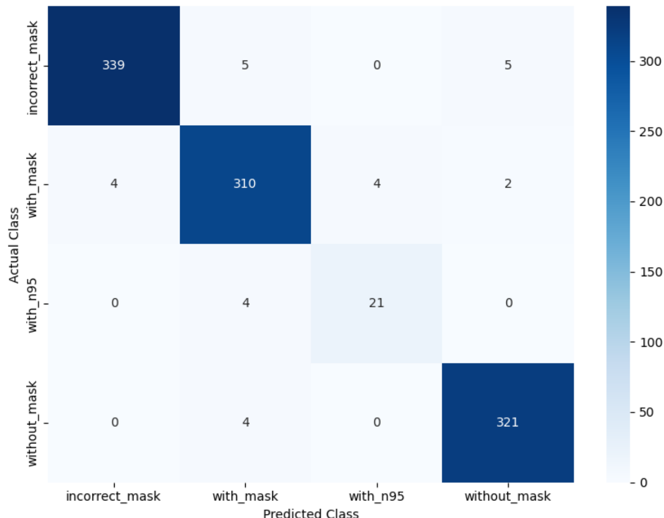

Figure 4.1: Confusion matrix heatmap for patience $p = 1$.

- TP = 310
- FN = 10
- FP = 13
- TN = 533 (866 - 310 - 10 - 13)

$$Precision = \frac{310}{310 + 13} = 0.9598$$

$$Recall = \frac{310}{310 + 10} = 0.9687$$

$$F1 - score = \frac{2 \cdot (0.96 \cdot 0.97)}{0.96 + 0.97} = 0.9642$$

$$Accuracy = \frac{310 + 533}{866} = 0.9774$$

Table 4.1: Per-class metrics for patience $p = 1$

Class	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Accuracy (%)**
incorrect$_{m}ask$	98.83	97.13	97.98	98.63
with_mask	95.98	96.87	96.42	97.74
with_n95	84.00	84.00	84.00	99.21
without_mask	97.87	98.77	98.32	98.92

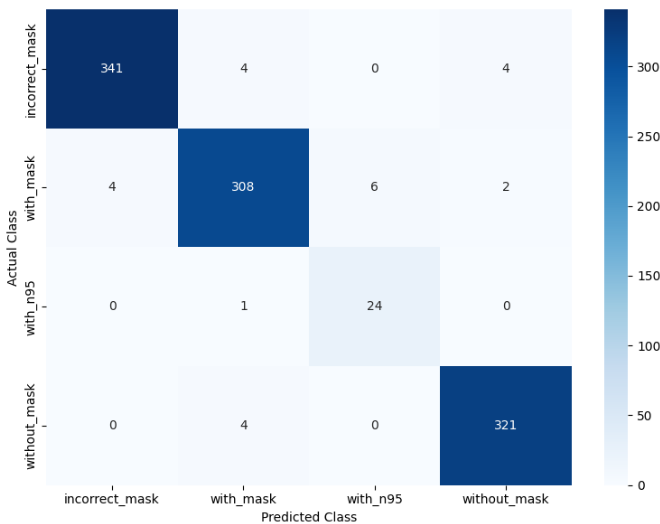

Figure 4.2: Confusion matrix heatmap for patience $p = 2$.

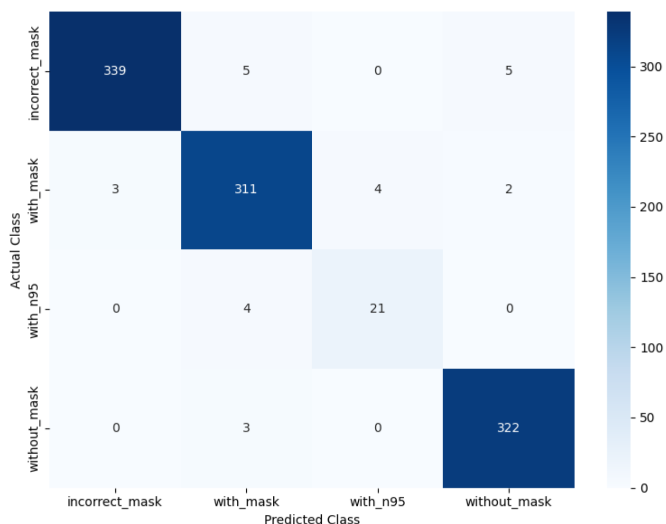

Figure 4.3: Confusion matrix heatmap for patience $p = 3$.

Table 4.2: Per-class metrics for patience $p = 2$

Class	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Accuracy (%)**
incorrect$_{m}ask$	98.84	97.71	98.27	98.82
with_mask	97.16	96.25	96.70	97.94
with_n95	80.00	96.00	87.27	99.31
without_mask	98.17	98.77	98.47	99.02

Summary of Findings

Among all configurations, the best performance was obtained at **patience** $p = 4$, achieving an overall accuracy of **97.55%**. This indicates that allowing moderate tolerance to temporary validation-loss fluctuations helps the model converge to a more stable and generalisable solution.

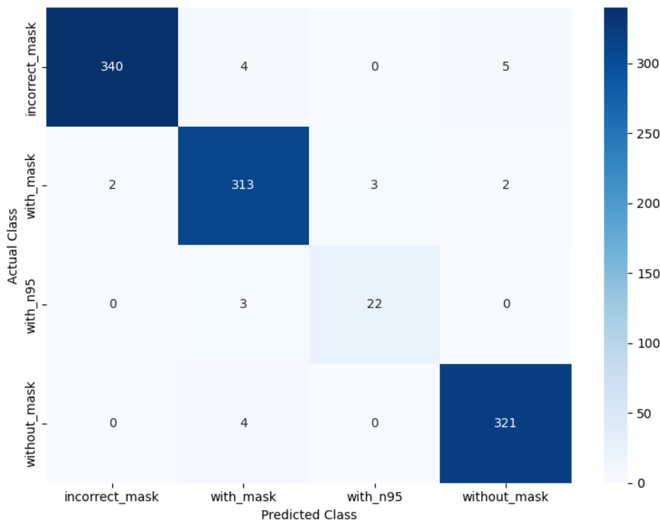

Figure 4.4: Confusion matrix heatmap for patience $p = 4$.

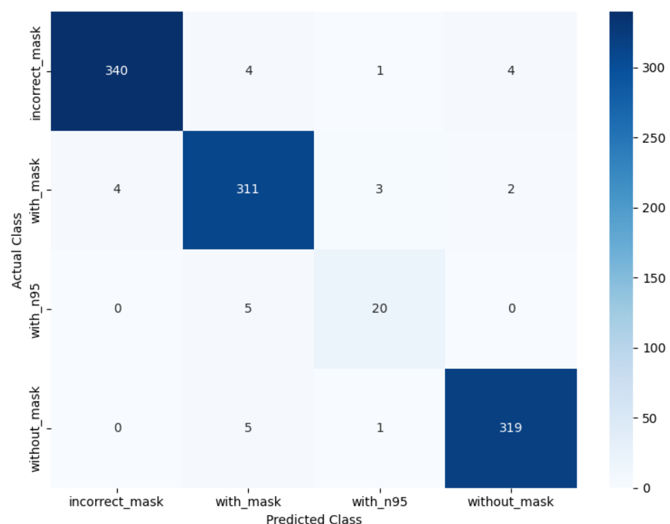

Figure 4.5: Confusion matrix heatmap for patience $p = 5$.

Table 4.3: Per-class metrics for patience $p = 3$

Class	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Accuracy (%)**
incorrect$_{m}ask$	99.12	97.13	98.12	98.72
with_mask	96.28	97.19	96.73	97.94
with_n95	84.00	84.00	84.00	99.21
without_mask	97.87	99.08	98.47	99.02

4.1.3 Real-Time Detection Performance

The standalone Python-based detector sustains **20–30 FPS** on CPU only hardware. The SSD detector consistently locates faces with high confidence, and the classifier performs stable frame by frame inference. These results demonstrate that the trained model is well suited for responsive real-time applications.

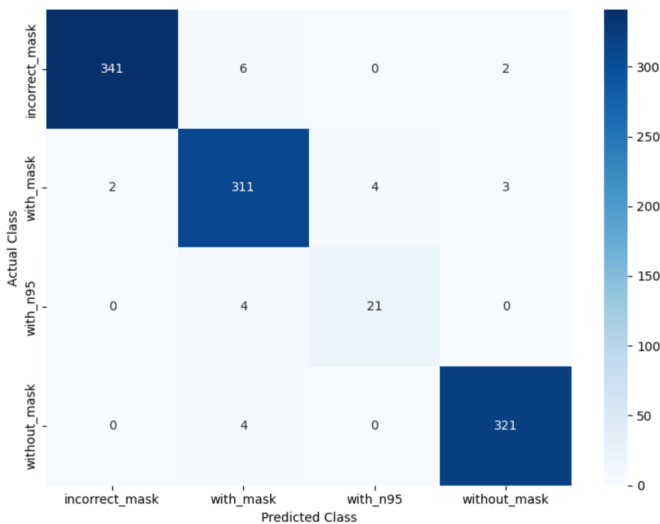

Figure 4.6: Confusion matrix heatmap for patience $p = 6$.

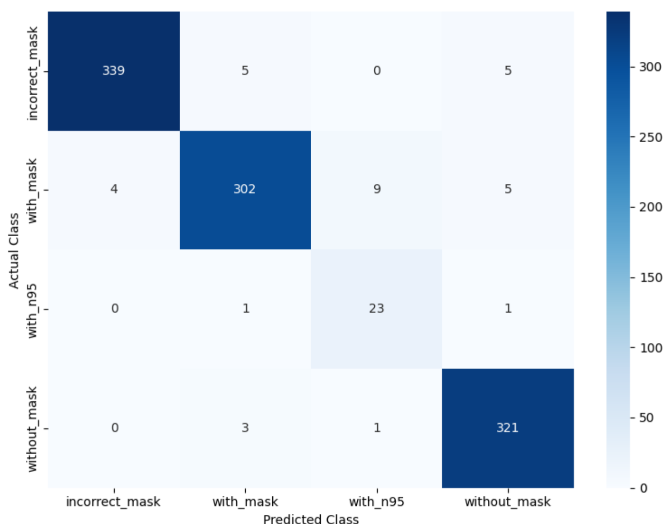

Figure 4.7: Confusion matrix heatmap for patience $p = 7$.

Table 4.4: Per-class metrics for patience $p = 4$

Class	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Accuracy (%)**
incorrect$_{m}ask$	99.42	97.42	98.41	98.92
with_mask	96.60	97.81	97.20	98.23
with_n95	88.00	88.00	88.00	99.41
without_mask	97.87	98.77	98.32	98.92

4.1.4 Browser Based Deployment Results

To evaluate system portability, the model was integrated into a FastAPI backend and a JavaScript-based browser client. Each browser frame is captured, JPEG encoded, sent to the server, processed by the SSD + MobileNet pipeline, and returned with annotations. This workflow mirrors the architecture described in the methodology chapter and allows

Table 4.5: Per-class metrics for patience $p = 5$

Class	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Accuracy (%)**
incorrect$_mask$	98.84	97.42	98.12	98.72
with_mask	95.69	97.19	96.43	97.74
with_n95	80.00	80.00	80.00	99.02
without_mask	98.15	98.15	98.15	98.82

Table 4.6: Per-class metrics for patience $p = 6$

Class	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Accuracy (%)**
incorrect$_mask$	99.42	97.71	98.55	99.02
with_mask	95.69	97.19	96.43	97.74
with_n95	84.00	84.00	84.00	99.21
without_mask	98.47	98.77	98.62	99.12

Table 4.7: Per-class metrics for patience $p = 7$

Class	**Precision (%)**	**Recall (%)**	**F1-Score (%)**	**Accuracy (%)**
incorrect$_mask$	98.83	97.13	97.98	98.63
with_mask	97.11	94.37	95.72	97.35
with_n95	69.70	92.00	79.31	98.82
without_mask	96.69	98.77	97.72	98.53

Table 4.8: Overall accuracy for patience values 1–7.

Patience	**Overall Accuracy (%)**
1	95.95
2	96.92
3	97.22
4	**97.55**
5	97.04
6	96.95
7	96.98

real-time detection without installing Python locally.

4.1.5 Latency Analysis

End-to-end timing was measured using server-side processing logs (`time.perf_counter()`). Latency includes:

- Webcam capture and JPEG encoding,
- HTTP upload to the backend,
- Face detection and classification,
- JPEG re-encoding and return to the browser.

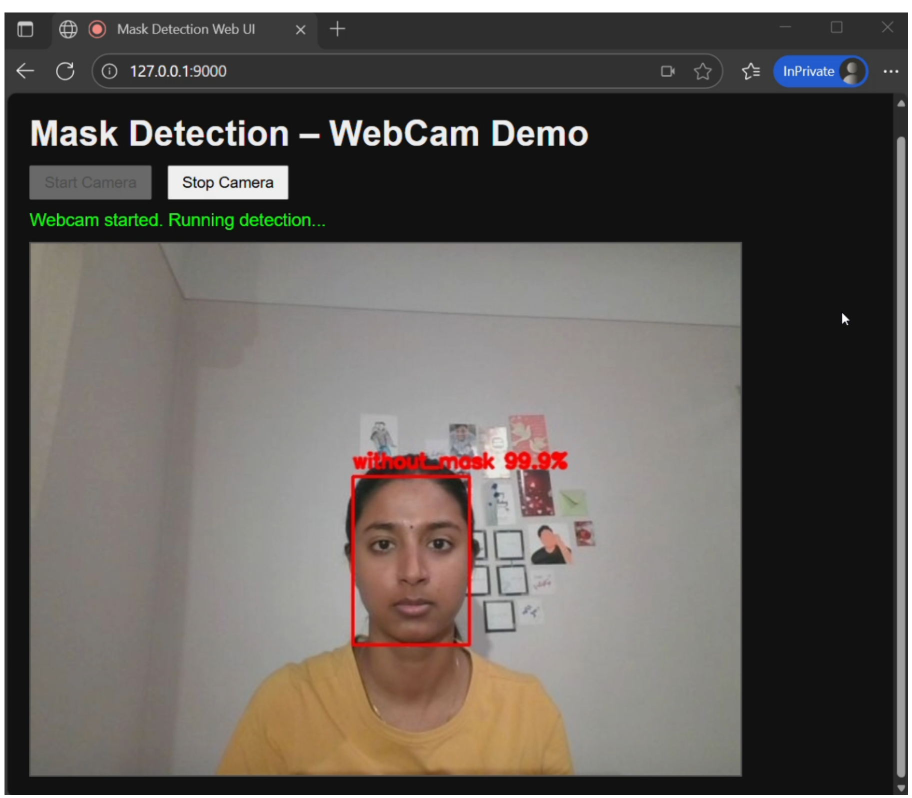

Figure 4.8: Real-time continuous mask detection and classification result - Without Mask

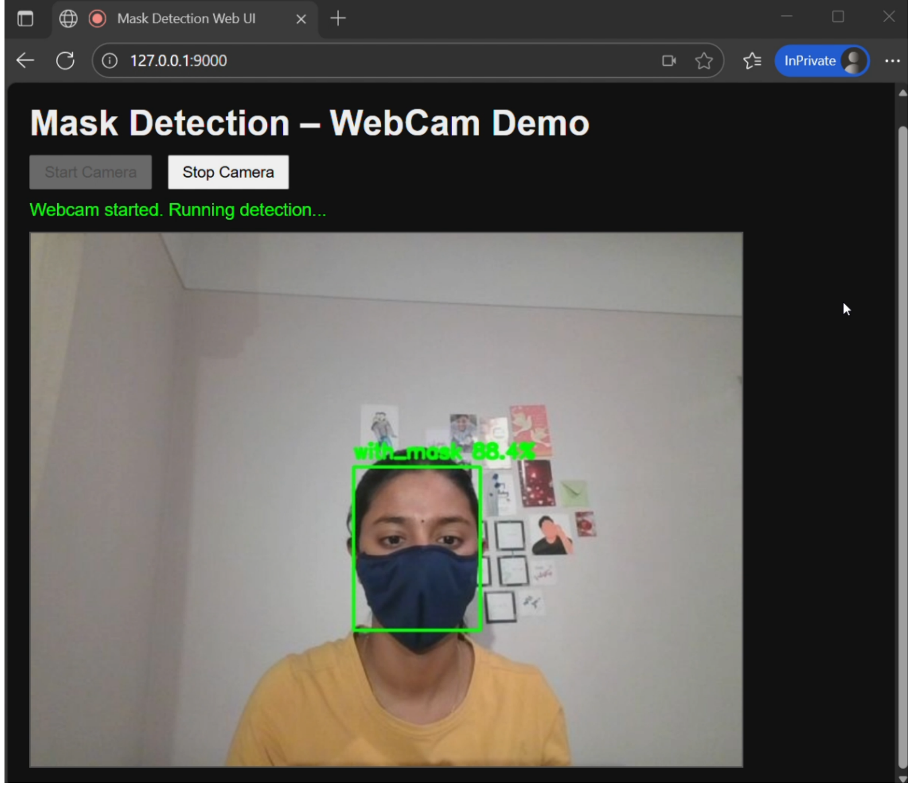

Figure 4.9: Real-time continuous mask detection and classification result - With Mask

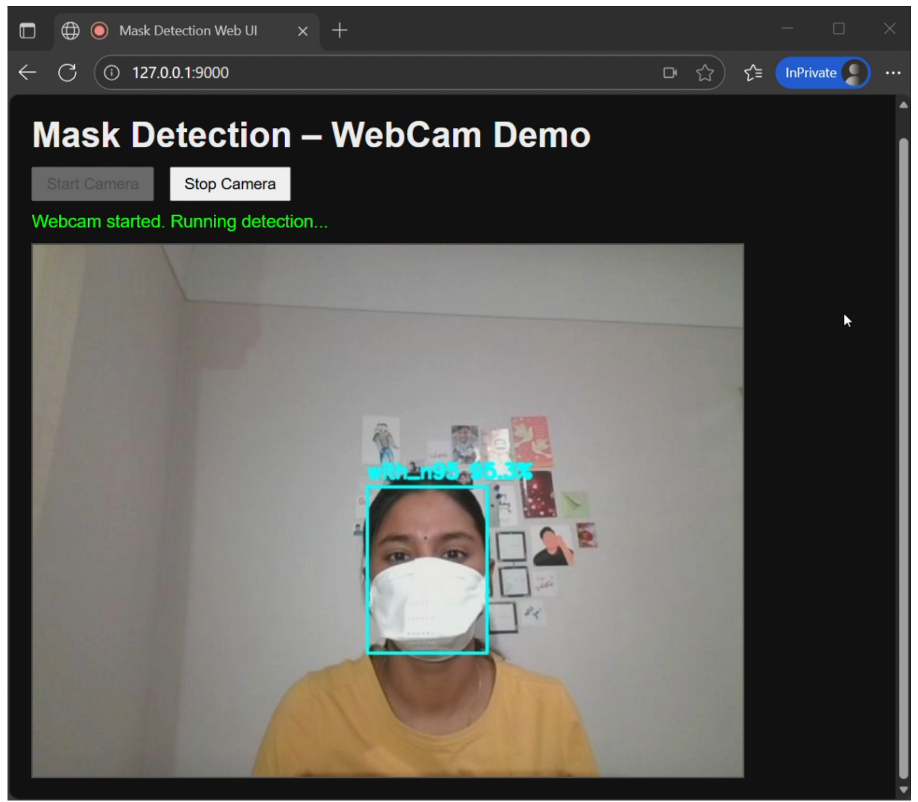

Figure 4.10: Real-time continuous mask detection and classification result - With N95 Mask

Average round trip time: 140–180 ms/frame

This delay persists even in a localhost (loopback) setup because the overhead is dominated by repeated client–server operations, which includes frame capture, JPEG encoding and decoding, HTTP request handling, and backend inference. These processing steps introduce some latency even in the absence of external network delays.

This corresponds to an effective rate of **5–7 FPS**, which remains suitable for compliance monitoring and kiosk-style interaction. The lower FPS relative to the desktop application is expected due to repeated JPEG encoding/decoding and client–server data transfer overhead.

4.1.6 Browser Visualization Quality

The browser interface provides smooth updates on the HTML canvas and supports explicit user control via Start/Stop buttons. All processing remains local to the machine (loopback), maintaining privacy while avoiding external cloud dependence.

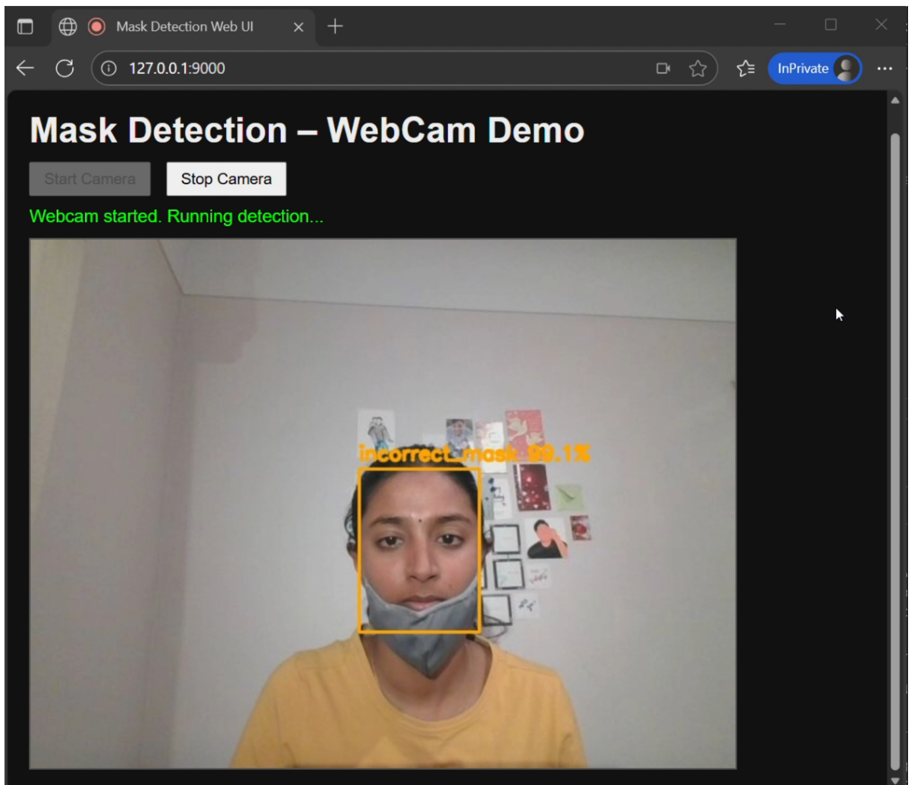

Figure 4.11: Real-time continuous mask detection and classification result - Incorrect Mask

Table 4.9: Comparison between desktop and Browser Based Interface

Metric	**Local Python App**	**Browser Based Interface**
FPS	20–30	5–7
Latency	40–60 ms	140–180 ms
User interaction	Terminal/UI window	Browser-based
Control	Full system access	Restricted, privacy safe
Processing location	Local CPU	Backend CPU via FastAPI

4.1.7 Summary of Findings

The experimental results demonstrate that the proposed system provides:

- High classification accuracy across the defined mask wearing categories,
- Stable and responsive real-time performance in the desktop based prototype,
- A functional browser-based client–server interface with acceptable end-to-end latency,
- Reliable end-to-end operation under typical indoor usage conditions.

The qualitative examples presented in this chapter are intended to illustrate overall system behavior rather than to isolate individual sources of error. In these examples, variations in mask color, shape, and material often co-occur with differences in mask

wearing category (e.g., regular mask, N95 mask, improperly worn mask). As such, the presented figures should not be interpreted as a controlled evaluation of robustness to specific mask attributes in isolation.

A systematic analysis of individual factors such as mask color, fabric type, or material composition would require controlled datasets in which only a single variable is varied at a time. Conducting such an attribute level study is beyond the scope of the present work and is identified as a direction for future investigation.

Overall, the results validate the proposed model and system architecture as a practical and reliable baseline for institutional mask compliance monitoring, while highlighting clear opportunities for further refinement and controlled robustness analysis.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This work presented the design and deployment of a real-time face mask classification system that combines a Caffe-based SSD face detector with a MobileNet backbone and a lightweight Browser interface. A dataset with four classes (*with_mask*, *without_mask*, *with_n95*, and *incorrect_mask*) was constructed, followed by systematic preprocessing, augmentation, and stratified splitting into training, validation, and test sets. Using transfer learning and an early stopping strategy, the final models consistently achieved test accuracies in the 97.55% with strong per-class precision and recall.

Beyond offline evaluation, the system was implemented as a browser accessible service. The terminal based version maintains ≈ 20–30 FPS on CPU hardware, while the FastAPI-based Browser interface delivers ≈ 5–7 FPS with end-to-end latencies of 140–180 ms per frame, which is sufficient for compliance monitoring and operator supervision. Overall, the results demonstrate that a lightweight, transfer-learning-based architecture can deliver reliable mask classification in real time while remaining accessible through standard browsers.

5.2 Future Work

Several directions can further enhance the robustness and applicability of the proposed system. First, dataset expansion is needed to reduce bias and improve generalisation across demographics and mask types. In particular, collecting more samples for the *with_n95* and *incorrect_mask* classes and including faces from a broader range of ethnicities, age groups, and lighting conditions would likely improve the most challenging decision boundaries.

Second, the two-stage SSD+MobileNet pipeline can be replaced or augmented with more recent single-stage detectors or joint detection–classification models (e.g., YOLO style architectures) to further reduce latency and potentially increase accuracy.

Third, the Browser based interface can be extended with additional features including role based dashboards, configurable alert thresholds, temporal smoothing over video sequences, and secure audit logging. Integrating the system with access control hardware (e.g., door controllers) and upstream facility management platforms would enable end-to-end automated workflows. Finally, privacy and security enhancements such as on-device encryption of logs or fine grained control over data retention represent an important area of future research to ensure that real-time mask monitoring can be deployed responsibly in sensitive healthcare and institutional settings.

Bibliography

[1] R. K. Putri and M. Athoillah, “Detection of facial mask using deep learning classification algorithm,” *Journal of Data Science and Intelligent Systems*, vol. 2, no. 1, pp. 58–63, 2024.

[2] J. Zhang, D. An, Y. Zhang, X. Wang, X. Wang, Q. Wang, Z. Pan, and Y. Yue, “A review on face mask recognition,” *Sensors (Basel, Switzerland)*, vol. 25, no. 2, p. 387, 2025.

[3] K. Lin, H. Zhao, J. Lv, C. Li, X. Liu, R. Chen, and R. Zhao, “Face detection and segmentation based on improved mask r-cnn,” *Discrete dynamics in nature and society*, vol. 2020, no. 1, p. 9242917, 2020.

[4] S. Tarnpradab, P. Poonpinij, N. Na Lumpoon, and N. Wattanapongsakorn, “Real-time masked face recognition and authentication with convolutional neural networks on the web application,” *Multimedia Tools and Applications*, vol. 84, no. 20, pp. 22655–22679, 2025.

[5] C. Dewi, D. Manongga, Hendry, E. Mailoa, and K. D. Hartomo, “Deep learning and yolov8 utilized in an accurate face mask detection system,” *Big Data and Cognitive Computing*, vol. 8, no. 1, p. 9, 2024.

[6] P. Dubey, P. Dubey, C. Iwendi, C. N. Biamba, and D. D. Rao, “Enhanced iot-based face mask detection framework using optimized deep learning models: a hybrid approach with adaptive algorithms,” *IEEE Access*, 2025.

[7] A. Nowrin, S. Afroz, M. S. Rahman, I. Mahmud, and Y.-Z. Cho, “Comprehensive review on facemask detection techniques in the context of covid-19,” *IEEE access*, vol. 9, pp. 106839–106864, 2021.

[8] X. Su, M. Gao, J. Ren, Y. Li, M. Dong, and X. Liu, “Face mask detection and classification via deep transfer learning,” *Multimedia Tools and Applications*, vol. 81, no. 3, pp. 4475–4494, 2022.

[9] M. S. Ejaz, M. R. Islam, M. Sifatullah, and A. Sarker, “Implementation of principal component analysis on masked and non-masked face recognition,” in *2019 1st*

international conference on advances in science, engineering and robotics technology (ICASERT), pp. 1–5, IEEE, 2019.

[10] M. H. M. Kamil, N. Zaini, L. Mazalan, and A. H. Ahamad, "Online attendance system based on facial recognition with face mask detection," *Multimedia tools and applications*, vol. 82, no. 22, pp. 34437–34457, 2023.

[11] J. R. Sella Veluswami, S. Prakash, N. Parekh, *et al.*, "Face mask detection using ssdnet and lightweight custom cnn," in *Proceedings of the International Conference on IoT Based Control Networks & Intelligent Systems-ICICNIS*, 2021.

[12] T.-N. Pham, V.-H. Nguyen, and J.-H. Huh, "Integration of improved yolov5 for face mask detector and auto-labeling to generate dataset for fighting against covid-19," *The Journal of supercomputing*, vol. 79, no. 8, p. 8966, 2023.

[13] X. Jiang, T. Gao, Z. Zhu, and Y. Zhao, "Real-time face mask detection method based on yolov3," *Electronics*, vol. 10, no. 7, p. 837, 2021.

[14] M. Rezaei, E. Ravanbakhsh, E. Namjoo, and M. Haghighat, "Assessing the effect of image quality on ssd and faster r-cnn networks for face detection," in *2019 27th Iranian Conference on Electrical Engineering (ICEE)*, pp. 1589–1594, IEEE, 2019.

[15] W. Hongtao and Y. Xi, "Object detection method based on improved one-stage detector," in *2020 5th International Conference on Smart Grid and Electrical Automation (ICSGEA)*, pp. 209–212, IEEE, 2020.

[16] A. Buslaev, V. I. Iglovikov, E. Khvedchenya, A. Parinov, M. Druzhinin, and A. A. Kalinin, "Albumentations: fast and flexible image augmentations," *Information*, vol. 11, no. 2, p. 125, 2020.

[17] M. D. Bloice, C. Stocker, and A. Holzinger, "Augmentor: an image augmentation library for machine learning," *arXiv preprint arXiv:1708.04680*, 2017.

[18] T. Kumar, R. Brennan, A. Mileo, and M. Bendechache, "Image data augmentation approaches: A comprehensive survey and future directions," *IEEE Access*, 2024.

[19] N. E. Khalifa, M. Loey, and S. Mirjalili, "A comprehensive survey of recent trends in deep learning for digital images augmentation," *Artificial Intelligence Review*, vol. 55, no. 3, pp. 2351–2377, 2022.

[20] A. Mikołajczyk and M. Grochowski, "Data augmentation for improving deep learning in image classification problem," in *2018 international interdisciplinary PhD workshop (IIPhDW)*, pp. 117–122, IEEE, 2018.

Appendix A

Source Code

Due to package limitations in Overleaf, the complete source code for this project is provided as appended PDF files following the main report. The source code was cloned locally from the project repository to ensure consistency with the submitted implementation.

common.py

```python
# Necessary imports and uploads

import cv2
import numpy as np
import tensorflow as tf
import sklearn
import matplotlib.pyplot as plt
import time
import glob
import os
import pathlib
import re
import random
import imageio.v2 as iio
import shutil
import albumentations as A


from tensorflow.keras.preprocessing.image import img_to_array, ImageDataGenerator
from tensorflow.keras.models import load_model
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input
from tensorflow.keras.applications import MobileNet, VGG19
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout, AveragePooling2D
import tensorflow_datasets as tfds
from PIL import Image, UnidentifiedImageError


image_path = "C:/Users/nsach/OneDrive/Desktop/meng_project/source"
dataset_name = "dataset"
processed_folder = "processed"
dataset_root = os.path.join(image_path, dataset_name)
processed_root = os.path.join(image_path, processed_folder)

# === Class definitions ===
CLASSES = ["incorrect_mask", "with_mask", "with_n95", "without_mask"]

# === Global constants ===
# standard input size for MobileNetV2, VGG19, etc
TARGET = 224

# === Optional visualization colors ===
COLORS = {
    "incorrect_mask": (0, 165, 255),  # orange
    "with_mask": (0, 255, 0),         # green
    "with_n95": (255, 255, 0),        # cyan-yellow
    "without_mask": (0, 0, 255)       # red
}
```

```
49
50  # === Mappings ===
51
52  CLASS_TO_IDX = {cls: i for i, cls in enumerate(CLASSES)}
53  IDX_TO_CLASS = {i: cls for i, cls in enumerate(CLASSES)}
54
55  caffe_model = f"{image_path}/face_detection_model/res10_300x300_ssd_iter_140000.caffemodel"
56  dep_prototxt = f"{image_path}/face_detection_model/deploy.prototxt"
```

preprocessing.py

```
1  from common import *
2
3
4  # Image generator paper dataset
5  print(image_path)
6  image_folder  = os.path.join(image_path, dataset_name)
7  image_folders = glob.glob(os.path.join(image_folder, "*"))
8  images = []
9
10
11 for folder in image_folders:
12   classifier = os.path.basename(folder)   # get last part of path (folder name)
13   sub_folders = glob.glob(os.path.join(folder, "*"))  # list subfolders inside this class
   folder
14   for sub_folder in sub_folders:
15     detail = os.path.basename(sub_folder) # get subfolder name
16     # print('classifier and sub folders: ', classifier, detail)
17     images_in_class = glob.glob(os.path.join(sub_folder, "*"))  # list all images inside
   subfolder
18     for image in images_in_class:
19       info = {"data": image, "class": classifier, "detail": detail}
20       images.append(info)
21
22 print(images[0])
23 # image = iio.imread(images[0]["data"])
24 # plt.imshow(image)
25 # plt.show()
26
27 image_dirs, image_labels = [], []
28
29 for img_info in images:
30     if img_info["class"] in CLASSES:
31         image_dirs.append(img_info["data"])
32         image_labels.append(img_info["class"])
33
34 print(f"Total images: {len(image_dirs)}; classes: {sorted(set(image_labels))}")
35
36 _face_detector = None
37
38 # Loads the caffe model and prototxt in _face_detector
39 def _get_face_detector():
40   global _face_detector
41   if _face_detector is None:
42     if not os.path.exists(dep_prototxt):
43       raise FileNotFoundError(f"Missing prototxt: {dep_prototxt}")
44     if not os.path.exists(caffe_model):
45       raise FileNotFoundError(f"Missing caffemodel: {caffe_model}")
46     _face_detector = cv2.dnn.readNetFromCaffe(dep_prototxt, caffe_model)
```

```
47   return _face_detector
48
49
50 # Define the Albumentations augmentation pipeline
51 aug = A.Compose([
52     # A.Flip(p=0.5),
53     A.RandomBrightnessContrast(p=0.2),
54     A.Affine(scale=(0.5, 1.25), translate_percent=(0.05, 0.05), rotate=(-30, 30), shear=(-10,
10), p=0.8),
55     A.GaussianBlur(blur_limit=(3, 7), p=0.5),
56 ])
57
58 def find_face_in_image(image):
59   # Drop alpha if present
60   if image.ndim == 3 and image.shape[-1] > 3:
61     image = image[:, :, :3]
62
63   face_detector = _get_face_detector()
64
65   h, w = image.shape[:2]
66   image_bgr = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
67
68   # Build DNN blob (BGR, mean-subtracted, 300×300)
69   blob = cv2.dnn.blobFromImage(image_bgr, 1.0, (300, 300), (104.0, 177.0, 123.0))
70   face_detector.setInput(blob)
71   detections = face_detector.forward()  # (1,1,N,7)
72
73   best = None; best_area = 0.0
74   for i in range(detections.shape[2]):
75     conf = float(detections[0, 0, i, 2])
76     if conf < 0.5:
77       continue
78     x1, y1, x2, y2 = detections[0, 0, i, 3:7]
79     x1, y1, x2, y2 = np.clip([x1, y1, x2, y2], 0, 1)
80     sx, sy, ex, ey = int(x1*w), int(y1*h), int(x2*w), int(y2*h)
81     area = max(0, ex - sx) * max(0, ey - sy)
82     if area > best_area:
83       best_area = area
84       best = (sx, sy, ex, ey)
85
86   if best is None:
87     return None
88
89   sx, sy, ex, ey = best
90   face = image_bgr[sy:ey, sx:ex, :]
91   face = cv2.resize(face, (TARGET, TARGET))
92   face = cv2.cvtColor(face, cv2.COLOR_BGR2RGB)
93   return face
94
95
```

```
96  def face_image_generator(image_dirs, class_labels):
97    for image_dir, label in zip(image_dirs, class_labels):
98      try:
99        image = iio.imread(image_dir)  # RGB
100       face = find_face_in_image(image)
101       if face is None:
102          continue
103
104       if random.random() < 0.9:
105         augface = aug(image=face)["image"]
106       else:
107          augface = face
108
109       assert augface.shape == (TARGET, TARGET, 3), f"Invalid output shape: {augface.shape}"
110       augface = tf.convert_to_tensor(augface, tf.uint8)
111       label_idx = CLASS_TO_IDX[label]
112       label = tf.convert_to_tensor(label_idx, tf.uint8)
113
114       yield augface, label
115
116     except Exception as e:
117       print(e)
118       continue
119
120
121 face_gen = face_image_generator(image_dirs, image_labels)
122
123 i=0
124 for face, label in face_gen:
125   i += 1
126   if i > 4:
127     break
128
129
130 # preprocess the data prior to running the actual model
131
132 def save_faces_in_dir(full_img_dirs, img_labels, folder_name, find_face=False,
    out_root=None):
133   """
134   Write curated faces to: <out_root>/<folder_name>/<class>/*.jpg
135   Defaults to <image_path>/processed if out_root is None.
136   """
137   saved_faces = 0
138   found_faces = 0
139   skipped_faces = 0
140
141   if out_root is None:
142     out_root = os.path.join(image_path, "processed")  # keep out of 'dataset'
143   base_dir = os.path.join(out_root, folder_name)
144
```

```python
145  # Make class dirs
146  class_dirs = {cls: os.path.join(base_dir, cls) for cls in CLASSES}
147  os.makedirs(base_dir, exist_ok=True)
148  for d in class_dirs.values():
149    os.makedirs(d, exist_ok=True)
150
151  face_dirs, face_labels = [], []
152  current_progress = 0
153
154  for label, image_dir in zip(img_labels, full_img_dirs):
155    if current_progress % 200 == 0 and len(img_labels) > 0:
156      print(f"Processing {folder_name} {current_progress / len(img_labels) * 100:.1f}%")
157    current_progress += 1
158
159    try:
160      label_idx  = CLASS_TO_IDX[label]   # validates the label
161      image_name = os.path.basename(image_dir)
162      new_image_dir = os.path.join(class_dirs[label], image_name)
163
164      if not os.path.isfile(new_image_dir):
165        img = iio.imread(image_dir)
166
167      # Drop alpha channel if present
168        if img.ndim == 3 and img.shape[-1] > 3:
169          img = img[:, :, :3]
170
171        if find_face:
172          face = find_face_in_image(img)
173          if face is None:
174            skipped_faces += 1
175            continue
176        else:
177          face = img
178
179        iio.imwrite(new_image_dir, face)  # RGB save
180        saved_faces += 1
181      else:
182        found_faces += 1
183
184      face_dirs.append(new_image_dir)
185      face_labels.append(label)
186
187    except Exception as e:
188    # Could be unreadable file, bad label, etc.
189      print("skip:", image_dir, "| reason:", e)
190      skipped_faces += 1
191      continue
192
193  print(f"{saved_faces = }")
194  print(f"{found_faces  = }")
```

```
195    print(f"{skipped_faces = }")
196    for cls, d in class_dirs.items():
197      print(d, len(glob.glob(os.path.join(d, "*"))))
198
199    return base_dir, face_dirs, face_labels
200
201
202
203  data_dir, face_dir, face_labels = save_faces_in_dir(image_dirs, image_labels, "faces",
     find_face=True)
204
```

train_val_test_split.py

```python
from common import *
from sklearn.model_selection import train_test_split
from PIL import Image, UnidentifiedImageError


# Reconstruct lists from the saved faces folder
faces_root = os.path.join(image_path, "processed", "faces")
classes = CLASSES

face_dir = []
face_labels = []

for cls in classes:
    cls_dir = os.path.join(faces_root, cls)
    # collect typical image extensions
    exts = ("*.jpg","*.jpeg","*.png","*.bmp","*.webp","*.tif","*.tiff")
    files = []
    for ext in exts:
        files.extend(glob.glob(os.path.join(cls_dir, ext)))
    # add to lists
    face_dir.extend(files)
    face_labels.extend([cls]*len(files))

print(f"Found {len(face_dir)} files under {faces_root}")

X_train, X_test, y_train, y_test = train_test_split(
    face_dir, face_labels,
    test_size=0.07, train_size=0.93,
    random_state=42, stratify=face_labels
)

X_train, X_val, y_train, y_val = train_test_split(
    X_train, y_train,
    test_size=0.25, train_size=0.75,
    random_state=42, stratify=y_train
)


def copy_files_into_copied_tree(full_img_dirs: list, img_labels: list, folder_name: str):
  """
  Copies files into: <image_path>/processed/<folder_name>/<class>/
  Example: source/processed/Train/with_mask/...
  """
  base_dir = os.path.join(image_path, "processed", folder_name)
  os.makedirs(base_dir, exist_ok=True)

  # Create class subfolders
  class_dirs = {cls: os.path.join(base_dir, cls) for cls in CLASSES}
```

```
49    for d in class_dirs.values():
50      os.makedirs(d, exist_ok=True)
51    face_dirs, face_labels = [], []
52    current_progress = 0
53
54    for label, image_dir in zip(img_labels, full_img_dirs):
55      if current_progress % 200 == 0:
56        print(f"Processing {folder_name} {current_progress / len(img_labels) * 100:.1f}%")
57        current_progress += 1
58
59      try:
60        image_name = os.path.basename(image_dir)
61        new_image_dir = os.path.join(class_dirs[label], image_name)
62
63        # Copy only if not already copied
64        if not os.path.isfile(new_image_dir):
65          shutil.copyfile(image_dir, new_image_dir)
66
67        face_dirs.append(new_image_dir)
68        face_labels.append(label)
69
70      except Exception as e:
71        print(f"Error copying {image_dir}: {e}")
72
73    print(f"Finished copying {len(face_dirs)} images to {base_dir}")
74    return base_dir, face_dirs, face_labels
75
76
77
78  # Create training directory
79  train_dir, train_faces, train_labels = copy_files_into_copied_tree(X_train, y_train, "Train")
80  print(len(train_faces))
81
82
83  # Create Testing Directory
84  test_dir, test_faces, test_labels = copy_files_into_copied_tree(X_test, y_test, "Test")
85  print(len(test_faces))
86
87
88  # Create Validation Directory
89  val_dir, val_faces, val_labels = copy_files_into_copied_tree(X_val, y_val, "Val")
90  print(len(train_faces), len(val_faces), len(test_faces))
91
92  # Validates the encoding of all of the face images. Images that did not encode properly cause
    issues while training
93
94  def validate_image_integrity(dirs, labels):
95    dirs_to_remove = []
96    labels_to_remove = []
97    for dir, label in zip(dirs, labels):
```

```
98        try:
99          Image.open(dir)
100       except (UnidentifiedImageError, FileNotFoundError) as e:
101         print(e, dir, label)
102         dirs_to_remove.append(dir)
103         labels_to_remove.append(label)
104
105     for dir, label in zip(dirs_to_remove, labels_to_remove):
106       try:
107         dirs.remove(dir)
108       except:
109         pass
110       try:
111         labels.remove(label)
112       except:
113         pass
114       try:
115         os.remove(dir)
116       except:
117         pass
118     return dirs, labels
119
120
121   # Exporting these directories so other scripts can use them
122   __all__ = ["train_dir", "val_dir", "test_dir"]
```

model_training.py

```
# model_training.py

from common import *
from train_val_test_split import train_dir, val_dir
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import MobileNet
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import AveragePooling2D, Flatten, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
import os

# === Data generators ===
training_datagenerator = ImageDataGenerator(
    rescale=1.0 / 255,
    horizontal_flip=True,
    zoom_range=0.2,
    shear_range=0.4,
    brightness_range=[0.8, 1.2],
    rotation_range=90,
)

datagenerator = ImageDataGenerator(rescale=1.0 / 255)

# Training Data
train_generator = training_datagenerator.flow_from_directory(
    directory=train_dir,
    target_size=(TARGET, TARGET),
    class_mode="categorical",
    batch_size=32,
)

# Validation Data
val_generator = datagenerator.flow_from_directory(
    directory=val_dir,
    target_size=(TARGET, TARGET),
    class_mode="categorical",
    batch_size=32,
)

# Model definition
mobilenet_model = MobileNet(
    weights="imagenet",
    include_top=False,
    input_shape=(TARGET, TARGET, 3),
)

for layer in mobilenet_model.layers:
    layer.trainable = False
```

```
49
50 model = Sequential()
51 model.add(mobilenet_model)
52 model.add(AveragePooling2D(pool_size=(7, 7)))
53 model.add(Flatten())
54 model.add(Dense(128, activation="relu", name="dense_128"))
55 model.add(Dropout(0.3))
56 model.add(Dense(64, activation="relu", name="dense_64"))
57 model.add(Dropout(0.3))
58 model.add(Dense(len(CLASSES), activation="softmax", name="output_layer"))
59
60 model.summary()
61
62 model.compile(
63     optimizer="adam",
64     loss="categorical_crossentropy",
65     metrics=["accuracy"],
66 )
67
68 # === EarlyStopping ===
69 early_stop = EarlyStopping(
70     monitor="val_loss",
71     patience=7,
72     restore_best_weights=True,
73     verbose=1,
74 )
75
76 # === Train model ===
77 history = model.fit(
78     train_generator,
79     epochs=30,                  # total possible epochs
80     validation_data=val_generator,
81     callbacks=[early_stop],     # early stopping applied
82 )
83
84 # Evaluate on validation set to get accuracy for comparison
85 val_loss, val_acc = model.evaluate(val_generator, verbose=0)
86 print(f"\nFinal validation accuracy: {val_acc * 100:.2f}%")
87 print(f"Final validation loss: {val_loss:.4f}")
88
89 best_val_acc = max(history.history["val_accuracy"])
90 print(f"Best validation accuracy during training: {best_val_acc * 100:.2f}%")
91
92 # Save model (same path used by other scripts)
93 out_path = os.path.join(image_path, "data", "saved_model.h5")
94 os.makedirs(os.path.dirname(out_path), exist_ok=True)
95 model.save(out_path)
96 print(f"Model saved to {out_path}")
```

evaluate_model.py

```python
# evaluate_model.py
from common import *
from train_val_test_split import test_dir
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import load_model
from sklearn.metrics import confusion_matrix
import numpy as np

# Load test set
datagenerator = ImageDataGenerator(rescale=1.0 / 255)

test_generator = datagenerator.flow_from_directory(
    directory=test_dir,
    target_size=(TARGET, TARGET),
    class_mode="categorical",
    batch_size=32,
    shuffle=False  # NECESSARY for proper TP/FP/TN/FN calculation
)

# Load trained model
model_path = os.path.join(image_path, "data", "saved_model_e30_p7.h5")
model = load_model(model_path)

# Predictions
y_prob = model.predict(test_generator)
y_pred = np.argmax(y_prob, axis=1)
y_true = test_generator.classes

class_indices = test_generator.class_indices
idx_to_class = {v: k for k, v in class_indices.items()}

# Confusion matrix
cm = confusion_matrix(y_true, y_pred)

print("\nConfusion Matrix (rows = true, cols = predicted):")
print("Class order:", [idx_to_class[i] for i in range(len(idx_to_class))])
print(cm, "\n")

num_classes = len(class_indices)

# TP, FP, TN, FN computation per class
print("\n=== Manual Metrics Per Class (TP, FP, TN, FN) ===")

for cls in range(num_classes):
    TP = cm[cls, cls]
    FP = np.sum(cm[:, cls]) - TP
    FN = np.sum(cm[cls, :]) - TP
    TN = cm.sum() - (TP + FP + FN)
```

```
49
50      manual_accuracy = (TP + TN) / (TP + TN + FP + FN)
51      precision = TP / (TP + FP + 1e-9)
52      recall = TP / (TP + FN + 1e-9)
53      f1 = 2 * precision * recall / (precision + recall + 1e-9)
54
55      print(f"\nClass: {idx_to_class[cls]}")
56      print(f"  TP = {TP}")
57      print(f"  FP = {FP}")
58      print(f"  TN = {TN}")
59      print(f"  FN = {FN}")
60      print(f"  Manual Accuracy: {manual_accuracy * 100:.2f}%")
61      print(f"  Precision: {precision * 100:.2f}%")
62      print(f"  Recall: {recall * 100:.2f}%")
63      print(f"  F1-score: {f1 * 100:.2f}%")
64
65  # Overall accuracy (all classes combined)
66  total_TP = sum(cm[i, i] for i in range(num_classes))
67  total_FP = sum(np.sum(cm[:, i]) - cm[i, i] for i in range(num_classes))
68  total_FN = sum(np.sum(cm[i, :]) - cm[i, i] for i in range(num_classes))
69  total_TN = cm.sum() - (total_TP + total_FP + total_FN)
70
71  overall_manual_acc = (total_TP + total_TN) / cm.sum()
72
73  print("\n=== Overall Manual Accuracy (TP+TN / Total) ===")
74  print(f"Overall Manual Accuracy: {overall_manual_acc * 100:.2f}%\n")
75
```

capture_and_predict.py

```python
from common import *  # gives TARGET, CLASSES, COLORS, caffe_model, dep_prototxt, image_path
from tensorflow.keras.models import load_model
import cv2
import numpy as np
import os

# === CONFIG ===
trained_model = f"{image_path}/data/saved_model.h5"

IMG_SIZE = (TARGET, TARGET)   # 224 x 224
CONF_THRESH = 0.5             # same as final_detect

# load models
print("Loading models...")
net = cv2.dnn.readNetFromCaffe(dep_prototxt, caffe_model)
model = load_model(trained_model)
print("Models loaded.")

def classify_face_bgr(face_bgr):
    """Crop -> RGB -> resize -> /255 -> predict."""
    face_rgb = cv2.cvtColor(face_bgr, cv2.COLOR_BGR2RGB)
    face_rgb = cv2.resize(face_rgb, IMG_SIZE).astype("float32") / 255.0
    x = np.expand_dims(face_rgb, 0)  # (1, 224, 224, 3)
    probs = model.predict(x, verbose=0)[0]
    idx = int(np.argmax(probs))
    return CLASSES[idx], float(probs[idx]), probs

def detect_and_classify_in_frame(frame_bgr):
    """Run SSD face detection, classify each face, draw boxes + labels."""
    h, w = frame_bgr.shape[:2]

    # build blob for SSD
    blob = cv2.dnn.blobFromImage(
        frame_bgr, 1.0, (300, 300),
        (104, 177, 123), swapRB=False, crop=False
    )
    net.setInput(blob)
    detections = net.forward()

    results = []  # (label, conf, probs, box)

    for i in range(detections.shape[2]):
        conf = float(detections[0, 0, i, 2])
        if conf < CONF_THRESH:
            continue

        box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
        x1, y1, x2, y2 = box.astype(int)
```

```
49          x1, y1 = max(0, x1), max(0, y1)
50          x2, y2 = min(w - 1, x2), min(h - 1, y2)
51          if x2 <= x1 or y2 <= y1:
52              continue
53
54          # crop face and classify
55          face_bgr = frame_bgr[y1:y2, x1:x2]
56          if face_bgr.size == 0:
57              continue
58
59          label, conf_cls, probs = classify_face_bgr(face_bgr)
60          results.append((label, conf_cls, probs, (x1, y1, x2, y2)))
61
62          # draw on frame
63          color = COLORS.get(label, (0, 255, 0))
64          text = f"{label} {conf_cls*100:.1f}%"
65          cv2.rectangle(frame_bgr, (x1, y1), (x2, y2), color, 2)
66          y_txt = y1 - 8 if y1 - 8 > 10 else y1 + 20
67          cv2.putText(
68              frame_bgr, text, (x1, y_txt),
69              cv2.FONT_HERSHEY_SIMPLEX, 0.6, color, 2, cv2.LINE_AA
70          )
71
72      return frame_bgr, results
73
74  def main():
75      cap = cv2.VideoCapture(0)
76      if not cap.isOpened():
77          raise RuntimeError("Could not open webcam. Check camera permissions.")
78
79      print("Press 'c' to capture & classify current frame, 'q' to quit.")
80
81      while True:
82          ret, frame = cap.read()
83          if not ret:
84              print("Failed to read frame from camera.")
85              break
86
87          # live preview
88          cv2.imshow("Webcam (press 'c' to classify, 'q' to quit)", frame)
89          key = cv2.waitKey(1) & 0xFF
90
91          if key == ord('c'):
92              # run detection + classification on this frame
93              disp = frame.copy()
94              disp, results = detect_and_classify_in_frame(disp)
95
96              # show annotated frame
97              cv2.imshow("Prediction", disp)
98
```

```
99                # save captured frame
100               os.makedirs("captures", exist_ok=True)
101               out_path = os.path.join("captures", "capture.jpg")
102               cv2.imwrite(out_path, frame)
103               print(f"Saved capture to {out_path}")
104
105               # print probabilities for each detected face
106               if not results:
107                   print("No face detected.")
108               else:
109                   for idx, (label, conf, probs, box) in enumerate(results):
110                       print(f"Face {idx+1}: {label} ({conf*100:.1f}%)")
111                       print("Probabilities:")
112                       for cls, p in zip(CLASSES, probs):
113                           print(f"  {cls:15s}: {p*100:.1f}%")
114
115           elif key == ord('q'):
116               break
117
118       cap.release()
119       cv2.destroyAllWindows()
120
121   if __name__ == "__main__":
122       main()
123
```

final_detect_and_classify.py

```
1  from common import *
2  from tensorflow.keras.applications.mobilenet import preprocess_input
3
4
5  trained_model = f"{image_path}/data/saved_model.h5"
6
7  IMG_SIZE = (TARGET, TARGET)          # trained at 224x224
8  CONF_THRESH = 0.5                # face detection threshold
9
10 # load models
11 net = cv2.dnn.readNetFromCaffe(dep_prototxt, caffe_model)
12 model = load_model(trained_model)
13
14 def face_mask_detector(frame_bgr):
15     h, w = frame_bgr.shape[:2]
16
17     # OpenCV DNN face detection (expects BGR)
18     blob = cv2.dnn.blobFromImage(frame_bgr, 1.0, (300, 300), (104, 177, 123), swapRB=False,
   crop=False)
19     net.setInput(blob)
20     detections = net.forward()
21
22     for i in range(detections.shape[2]):
23         conf = float(detections[0, 0, i, 2])
24         if conf < CONF_THRESH:
25             continue
26
27         # box coords
28         box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
29         x1, y1, x2, y2 = box.astype(int)
30         x1, y1 = max(0, x1), max(0, y1)
31         x2, y2 = min(w - 1, x2), min(h - 1, y2)
32         if x2 <= x1 or y2 <= y1:
33             continue
34
35         # crop face, convert to RGB, resize to training size, scale to [0,1]
36         face_bgr = frame_bgr[y1:y2, x1:x2]
37         face_rgb = cv2.cvtColor(face_bgr, cv2.COLOR_BGR2RGB)
38         face_rgb = cv2.resize(face_rgb, IMG_SIZE).astype("float32") / 255.0
39         x = np.expand_dims(face_rgb, 0)  # (1, 224, 224, 3)
40
41         # classify
42         probs = model.predict(x, verbose=0)[0]
43         idx = int(np.argmax(probs))
44         label = CLASSES[idx]
45         conf_txt = f"{label} {probs[idx]*100:.1f}%"
46
47         # draw
```

```
48          color = COLORS.get(label, (0, 255, 0))
49          cv2.rectangle(frame_bgr, (x1, y1), (x2, y2), color, 2)
50          y_txt = y1 - 8 if y1 - 8 > 10 else y1 + 20
51          cv2.putText(frame_bgr, conf_txt, (x1, y_txt), cv2.FONT_HERSHEY_SIMPLEX, 0.6, color, 2,
    cv2.LINE_AA)
52
53      return frame_bgr
54
55  def main():
56      cap = cv2.VideoCapture(0)  # default camera
57      if not cap.isOpened():
58          raise RuntimeError("Cannot open webcam")
59
60      print("Press 'q' to quit.")
61      while True:
62          ok, frame = cap.read()
63          if not ok:
64              break
65          out = face_mask_detector(frame)
66          cv2.imshow("Mask Detector", out)
67          if (cv2.waitKey(1) & 0xFF) == ord('q'):
68              break
69
70      cap.release()
71      cv2.destroyAllWindows()
72
73  if __name__ == "__main__":
74      main()
75
```

webui_source\app.py

```python
from fastapi import FastAPI, UploadFile, File, Request
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates

import uvicorn
import numpy as np
import cv2
import time
from tensorflow.keras.models import load_model

from common import *   # gives TARGET, CLASSES, COLORS, caffe_model, dep_prototxt, image_path

app = FastAPI(docs_url="/docs")

# ---------- Frontend setup ----------
app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")

@app.get("/", response_class=HTMLResponse)
async def home(request: Request):
    return templates.TemplateResponse("index.html", {"request": request})

# ---------- Load models once ----------
IMG_SIZE = (TARGET, TARGET)
CONF_THRESH = 0.5                 # face detection threshold

net = cv2.dnn.readNetFromCaffe(dep_prototxt, caffe_model)
model = load_model(f"{image_path}/data/saved_model.h5")


# ---------- Detector function ----------
def face_mask_detector(frame_bgr):
    h, w = frame_bgr.shape[:2]

    # OpenCV DNN face detection (expects BGR)
    blob = cv2.dnn.blobFromImage(
        frame_bgr, 1.0, (300, 300),
        (104, 177, 123), swapRB=False, crop=False
    )
    net.setInput(blob)
    detections = net.forward()

    for i in range(detections.shape[2]):
        conf = float(detections[0, 0, i, 2])
        if conf < CONF_THRESH:
            continue

```

```
49            # box coords
50            box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
51            x1, y1, x2, y2 = box.astype(int)
52            x1, y1 = max(0, x1), max(0, y1)
53            x2, y2 = min(w - 1, x2), min(h - 1, y2)
54            if x2 <= x1 or y2 <= y1:
55                continue
56
57            # crop face, convert to RGB, resize to training size, scale to [0,1]
58            face_bgr = frame_bgr[y1:y2, x1:x2]
59            face_rgb = cv2.cvtColor(face_bgr, cv2.COLOR_BGR2RGB)
60            face_rgb = cv2.resize(face_rgb, IMG_SIZE).astype("float32") / 255.0
61            x = np.expand_dims(face_rgb, 0)  # (1, 224, 224, 3)
62
63            # classify
64            probs = model.predict(x, verbose=0)[0]
65            idx = int(np.argmax(probs))
66            label = CLASSES[idx]
67            conf_txt = f"{label} {probs[idx]*100:.1f}%"
68
69            # draw
70            color = COLORS.get(label, (0, 255, 0))
71            cv2.rectangle(frame_bgr, (x1, y1), (x2, y2), color, 2)
72            y_txt = y1 - 8 if y1 - 8 > 10 else y1 + 20
73            cv2.putText(
74                frame_bgr, conf_txt, (x1, y_txt),
75                cv2.FONT_HERSHEY_SIMPLEX, 0.6, color, 2, cv2.LINE_AA
76            )
77
78        return frame_bgr
79
80
81    # ---------- API endpoint ----------
82    @app.post("/predict")
83    async def predict(file: UploadFile = File(...)):
84        start = time.perf_counter()  # start timing
85
86        data = await file.read()
87        np_img = np.frombuffer(data, np.uint8)
88        frame = cv2.imdecode(np_img, cv2.IMREAD_COLOR)
89
90        out_frame = face_mask_detector(frame)
91
92        # encode result image
93        _, jpeg = cv2.imencode(".jpg", out_frame)
94
95        elapsed_ms = (time.perf_counter() - start) * 1000.0
96        print(f"/predict processing time: {elapsed_ms:.1f} ms")
97
98        # if you don't care about returning timing to frontend, remove "rtt_ms"
```

```
99      return {
100         "image": jpeg.tobytes().hex(),
101         "rtt_ms": elapsed_ms
102     }
103
104
105 if __name__ == "__main__":
106     uvicorn.run("app:app", host="127.0.0.1", port=9000, reload=True)
107
```

webui_source\static\app.js

```
const statusEl = document.getElementById("status");
const video = document.getElementById("video");
const canvas = document.getElementById("canvas");
const ctx = canvas.getContext("2d");

const startBtn = document.getElementById("startBtn");
const stopBtn  = document.getElementById("stopBtn");

let stream = null;       // current MediaStream
let detectTimer = null;  // interval ID for sendFrame loop

console.log("app.js loaded");

// ---- Start camera and detection ----
async function startCamera() {
    try {
        // If already running, stop first
        await stopCamera(false);

        stream = await navigator.mediaDevices.getUserMedia({
            video: { width: 640, height: 480 }
        });
        video.srcObject = stream;

        statusEl.textContent = "Webcam started. Running detection...";
        startBtn.disabled = true;
        stopBtn.disabled = false;

        // Start detection loop (every 500 ms)
        detectTimer = setInterval(sendFrame, 500);

    } catch (err) {
        console.error("Error accessing webcam:", err);
        statusEl.textContent = "Error accessing webcam. Check permissions.";
    }
}

// ---- Stop camera and detection ----
async function stopCamera(updateStatus = true) {
    // Stop detection timer
    if (detectTimer !== null) {
        clearInterval(detectTimer);
        detectTimer = null;
    }

    // Stop video tracks
    if (stream) {
        stream.getTracks().forEach(t => t.stop());
```

```
49            stream = null;
50        }
51
52        // Clear canvas
53        ctx.clearRect(0, 0, canvas.width, canvas.height);
54
55        if (updateStatus) {
56            statusEl.textContent = "Camera stopped. Click 'Start Camera' to run again.";
57        }
58        startBtn.disabled = false;
59        stopBtn.disabled = true;
60    }
61
62    // ---- Send frame to backend ----
63    async function sendFrame() {
64        if (!video.videoWidth || !video.videoHeight) {
65            // video not ready
66            return;
67        }
68
69        canvas.width = video.videoWidth;
70        canvas.height = video.videoHeight;
71        ctx.drawImage(video, 0, 0);
72
73        canvas.toBlob(async (blob) => {
74            if (!blob) return;
75
76            const formData = new FormData();
77            formData.append("file", blob, "frame.jpg");
78
79            try {
80                const res = await fetch("/predict", {
81                    method: "POST",
82                    body: formData
83                });
84
85                if (!res.ok) {
86                    console.error("Non-200 from /predict:", res.status);
87                    statusEl.textContent = "Server error from /predict";
88                    return;
89                }
90
91                const data = await res.json();
92
93                const bytes = new Uint8Array(
94                    data.image.match(/.{1,2}/g).map(b => parseInt(b, 16))
95                );
96                const img = new Image();
97                img.src = URL.createObjectURL(new Blob([bytes], { type: "image/jpeg" }));
98
```

```
99              img.onload = () => {
100                 ctx.drawImage(img, 0, 0);
101             };
102         } catch (e) {
103             console.error("Error calling /predict:", e);
104             statusEl.textContent = "Error calling /predict. See console.";
105         }
106     }, "image/jpeg", 0.8);
107 }
108 
109 // ---- Wire up buttons ----
110 startBtn.addEventListener("click", () => {
111     startCamera();
112 });
113 
114 stopBtn.addEventListener("click", () => {
115     stopCamera();
116 });
```

webui_source\templates\index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Mask Detection Web UI</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 20px;
            background: #111;
            color: #eee;
        }
        h1 {
            margin-bottom: 10px;
        }
        #controls {
            margin-bottom: 10px;
        }
        button {
            margin-right: 10px;
            padding: 6px 12px;
            font-size: 14px;
        }
        #status {
            margin-top: 5px;
            margin-bottom: 10px;
            color: #0f0;
        }
        #video, #canvas {
            border: 2px solid #555;
            display: block;
            margin-top: 10px;
        }
        #video {
            display: none;  /* hide raw webcam */
        }
    </style>
</head>
<body>
    <h1>Mask Detection – WebCam Demo</h1>

    <div id="controls">
        <button id="startBtn">Start Camera</button>
        <button id="stopBtn" disabled>Stop Camera</button>
    </div>
    <div id="status">Click "Start Camera" to begin.</div>

```

```
49      <video id="video" autoplay playsinline></video>
50      <canvas id="canvas"></canvas>
51
52      <script src="/static/app.js"></script>
53  </body>
54  </html>
55
```