Real-Time Vehicle Speed Monitoring via YOLOv11 and ByteTrack

Presentation slides added in this file at the end

An Edge-Cloud Hybrid System for Intelligent Traffic Analysis

Utpal Anand 20231271

Electronics Project

Course Code: PH3144

Group Member: Mohammad Hammad and Nikunj

Abstract

This report details the design and implementation of a traffic monitoring system that detects vehicles, tracks their movement, and estimates their speed in real-time. By combining a low-cost ESP32-CAM for video streaming with a powerful laptop for AI processing, the system achieves high accuracy without expensive industrial hardware. The project utilizes the YOLOv11m model for detection and the ByteTrack algorithm for tracking.

Contents

1 Introduction **2**

2 System Architecture **5**

3 Methodology **7**

4 Implementation Details **10**

4.1 Python Processing Pipeline 11

5 Computational Barriers in ML **13**

References **20**

A Hardware Setup Guide **21**

B Software Installation **22**

B.1 Prerequisites 22

B.2 Installation Commands 22

Chapter 1

Introduction

Overview

Traffic management and campus security are growing challenges in modern infrastructure. Traditional solutions, such as industrial speed guns and dedicated radar systems, are prohibitively expensive and often limited to single-task operations. However, the democratization of Artificial Intelligence (AI) and Computer Vision has opened new avenues for surveillance. We can now leverage standard optical sensors to monitor complex environments, detect anomalies, and identify specific objects automatically.

This project proposes a Hybrid surveillance ecosystem. Rather than deploying expensive, high-compute hardware on every street pole, we utilize cost-effective devicesscapture visual data. This data is streamed over Wi-Fi to a centralized, powerful server (Cloud or Local Edge) responsible for the heavy lifting: running deep learning models like YOLOv11. This architecture ensures the system is scalable, affordable, and modular.

Context and Scalability

The relevance of this project extends beyond simple traffic monitoring. In the modern world, the ability to deploy cheap, "set-and-forget" visual sensors allows for massive scalability.

- **Modularity:** The same hardware setup used to track cars can be retrained to monitor crop health in agriculture, track wildlife, or ensure safety compliance in factories.

- **Cost-Efficiency:** By offloading processing to a central server, the per-unit cost of deployment drops significantly, allowing institutions to cover larger areas with limited budgets.

Local Context: The IISER Pune Implementation

While the technology has global applications, this research is grounded in solving specific logistical and safety problems faced within the IISER Pune campus. The adaptability of the system addresses two distinct local challenges:

Restricted Area Monitoring

The area directly in front of the Main Building is a designated restricted zone for unauthorized vehicles. Despite regulations, delivery motorcyclists frequently breach this perimeter. By the time security guards are notified, the violators have often moved to different locations, making interception difficult. This project aims to automate the detection of these specific violators and log their entry instantly.

Laboratory Corridor Safety

The corridors housing experimental laboratories require strictly controlled environments. Stray dogs wandering into these areas pose a significant safety risk to both the animals and sensitive equipment. The proposed model is trained to identify non-human entities (specifically dogs) and trigger immediate alerts to security personnel, ensuring lab safety is maintained without constant human patrol.

Problem Statement

Developing this hybrid system addresses three main categories of problems:

1. **The Hardware Constraint:** Running State-of-the-Art (SOTA) models like YOLOv11 requires computation that microcontrollers (like the ESP32) cannot handle. A robust pipeline is needed to bridge low-power capture with high-power processing.

2. **The Geometry of Speed:** Estimating the real-world speed of a 3D object moving through a 2D video frame involves complex perspective transformation and calibration, which is mathematically non-trivial.

3. **Operational Latency:** In scenarios like the IISER main building breach or a dog entering a lab, notification must be near-instantaneous. Manual monitoring is prone to fatigue and error.

Broader Impact: Democratizing Tech in Science

Finally, this project serves as a case study for the accessibility of modern technology. Historically, implementing such a system required deep expertise in embedded systems and computer science. However, with the advent of open-source libraries (OpenCV, PyTorch) and the assistance of Large Language Models (LLMs) in code generation, the barrier to entry has lowered significantly. This work demonstrates how researchers across various departments—physics, biology, or chemistry—can now leverage low-cost electronics and AI to automate data collection and monitoring in their own experiments.

Chapter 2

System Architecture

High-Level Design

The system is divided into two main parts: the **Edge Tier** (the camera) and the **Cloud Tier** (the processing computer).

The Edge Tier is responsible for capturing the visual data. The Cloud Tier is responsible for understanding it. This separation allows us to keep the hardware on the street cheap and disposable, while the intelligence resides on a more powerful, centralized machine.

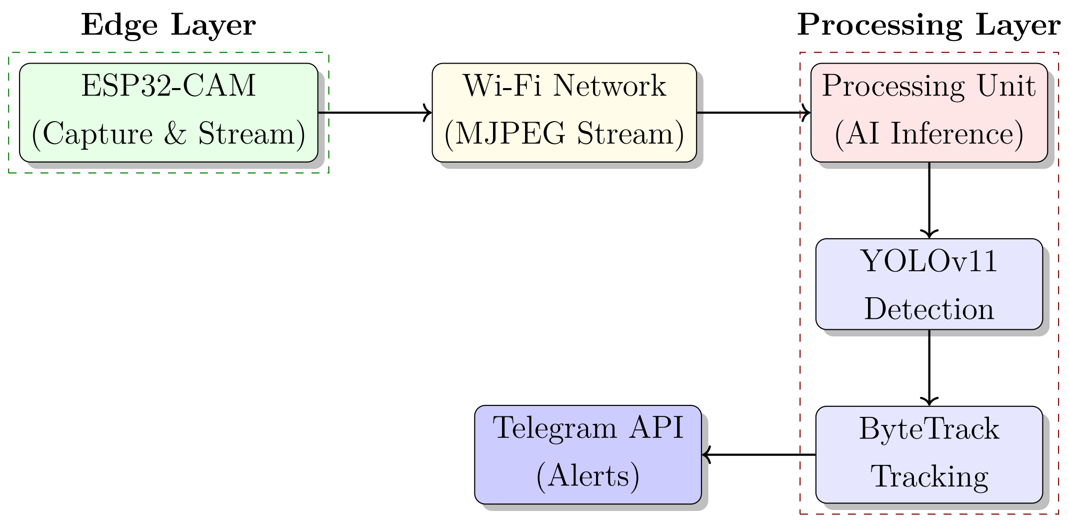


Figure 2.1: System Data Flow Diagram

Hardware Components

ESP32-CAM

The visual sensor is a small development board containing an ESP32 chip and an OV2640 camera sensor.

- **Cost:** Approx. $10 USD.
- **Role:** Connects to Wi-Fi and hosts a web server that streams video frames as JPEGs.
- **Limitation:** It generates heat and has limited Wi-Fi range, but provides sufficient resolution for prototyping and proof-of-concept.

Processing Unit (Workstation)

The heavy lifting is performed by a centralized computer. While the YOLOv11 architecture is optimized enough to run on standard CPUs (Central Processing Units), real-time tracking of high-speed vehicles benefits significantly from hardware acceleration.

For this implementation, we utilized a GPU-enabled setup to maximize frame rates, though the system remains compatible with CPU-only environments for less time-critical monitoring.

Implementation Specs: Inference on an image : 256x320 2 bus-l-s, 19 cars, 3 truck-l-s, 98.4ms Speed: 0.8ms preprocess, 98.4ms inference, 1.0ms postprocess per image at shape (1, 3, 256, 320)

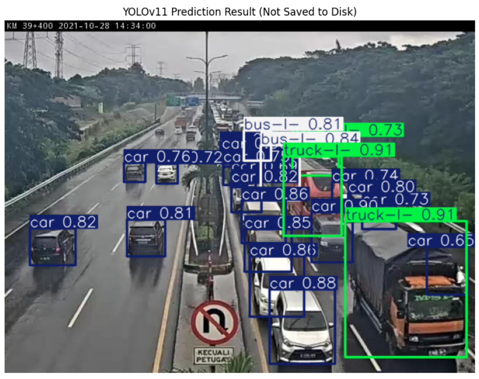


Figure 2.2: Inference Output using our fine tuned model

Chapter 3

Methodology

Object Detection with YOLOv11

YOLO stands for "You Only Look Once." It is a popular AI model because it is very fast. Unlike older models that scan an image multiple times, YOLO looks at the whole image once and predicts where objects are.

We selected the **YOLOv11m (Medium)** version.

- **Nano/Small versions** were too inaccurate for distant cars.
- **Large/X-Large versions** were too slow for real-time video.

Object Tracking with ByteTrack

Detecting a car in one frame is easy. Understanding that the car in Frame 1 is the *same* car in Frame 2 is hard. This is called Tracking.

We used ByteTrack. Most trackers throw away detections if the AI isn't 100% sure (low confidence). ByteTrack keeps these low-confidence detections and tries to match them to existing tracks. This is crucial for situations where a car is temporarily hidden behind a truck or bus.

Tracking Logic: The ByteTrack Algorithm

To calculate speed, we first need to track the same object across multiple video frames. We utilize ByteTrack, a state-of-the-art Multi-Object Tracking (MOT) algorithm.

Unlike traditional trackers (like DeepSORT) that discard low-confidence detections to avoid errors, ByteTrack attempts to associate *every* detection box. It operates on the principle that low-confidence detections are not necessarily background noise; they are often simply distinct objects that are currently occluded or blurred.

The algorithm proceeds in the following steps:

1. Detection and Partitioning

In every frame, the YOLO detector outputs a set of bounding boxes along with a confidence score. ByteTrack separates these boxes into two categories based on a high threshold (e.g., 0.6):

- **High-Confidence Detections ($\mathcal{D}_{high}$):** Objects the model is sure about (e.g., a clearly visible car).
- **Low-Confidence Detections ($\mathcal{D}_{low}$):** Objects with lower scores, often due to occlusion or motion blur.

2. First Association (High Confidence)

First, the algorithm attempts to match the high-confidence boxes ($\mathcal{D}_{high}$) to the existing Tracklets (objects currently being tracked) using the Kalman Filter's predicted location.

$$\text{Match}(\mathcal{D}_{high}, \text{Tracks}) \rightarrow \text{Matched Tracks}$$

Using Intersection over Union (IoU), we confirm that the new box effectively overlaps with where we expected the object to be.

3. Second Association (Low Confidence Recovery)

This is the core innovation of ByteTrack. Objects that were not matched in the first step are usually occluded (e.g., a delivery bike passing behind a pillar). Instead of deleting the unmatched tracks, the algorithm tries to match them against the Low-Confidence boxes ($\mathcal{D}_{low}$).

$$\text{Match}(\mathcal{D}_{low}, \text{Unmatched Tracks}) \rightarrow \text{Recovered Tracks}$$

This step allows us to "recover" objects that temporarily fade from clear view, maintaining their ID and ensuring the speed calculation remains continuous rather than resetting.

Formula used:

$$\text{Real Distance (m)} = \text{Pixel Distance} \times \text{Ratio)}$$

Ratio depends on the placement of camera and angle

$$\text{Speed (km/h)} = \frac{\text{Real Distance}}{\text{Time elapsed}} \times 3.6$$

Chapter 4

Implementation Details

The ESP32 Firmware

The code below runs on the camera. It sets up a web server and streams JPEGs efficiently.

```c
#include "esp_camera.h"
#include "esp_http_server.h"

void startCameraServer() {
    httpd_config_t config = HTTPD_DEFAULT_CONFIG();
    config.server_port = 80;

    httpd_uri_t stream_uri = {
        .uri       = "/stream",
        .method    = HTTP_GET,
        .handler   = stream_handler,
        .user_ctx  = NULL
    };

    if (httpd_start(&stream, &config) == ESP_OK) {
        httpd_register_uri_handler(stream, &stream_uri);
    }
}
// Note: Full setup involves WiFi connection logic
```

Listing 4.1: ESP32-CAM Streaming Setup

4.1 Python Processing Pipeline

The Python script on the laptop connects to the stream and processes it frame-by-frame.

```python
from ultralytics import YOLO
import cv2
import time

# Load the model
model = YOLO('yolo11m.pt')

# Connect to ESP32 stream
stream_url = "http://192.168.1.100:81/stream"
cap = cv2.VideoCapture(stream_url)

# Calibration Constant (Meters per Pixel)
PPM = 0.045

while True:
    ret, frame = cap.read()
    if not ret: break

    # 1. Detect
    results = model(frame, verbose=False)

    # 2. Update Tracker (ByteTrack)
    # (Tracking logic implemented in separate class)
    tracks = tracker.update(results)

    for track in tracks:
        track_id = track.id
        # 3. Calculate Speed
        # Measure distance moved since last frame
        pixel_dist = calculate_distance(track.prev_pos, track.curr_pos)
        real_dist = pixel_dist * PPM
        speed_mps = real_dist / time_elapsed
        speed_kmh = speed_mps * 3.6

        # 4. Alert
        if speed_kmh > 100:
            send_telegram_alert(frame, speed_kmh)

    cv2.imshow("Monitor", frame)
    if cv2.waitKey(1) == ord('q'): break
```

Listing 4.2: Main Processing Loop

Alert System

We use the Telegram Bot API. It is free and instant. When a vehicle exceeds the speed limit (e.g., 100 km/h), the system captures the current frame, draws a red box around the car, and uploads the photo to a specific chat ID.

Chapter 5

Computational Barriers in ML

The Cost of Research

While this project was successful, it highlighted a major problem in modern science: **Computer Science is becoming expensive.**

To train the YOLO model for this project, we needed a powerful GPU. We used one A30 GPU, but training from scratch would have taken weeks. Large research labs (like Google or OpenAI) use clusters of thousands of GPUs costing millions of dollars.

The "Tiered" System

This creates a split in the research community:

- **Tier 1:** Tech Giants and Ivy League Universities. They have unlimited compute power. They can try thousands of ideas.
- **Tier 2:** Smaller Universities and Independent Researchers. We have limited resources. We have to use pre-trained models because we cannot afford to train our own from scratch.

Why This Matters

If only rich institutions can afford to do AI research, then AI will only solve problems that rich institutions care about. Local problems—like traffic in specific developing nations, or regional agricultural diseases—might get ignored because local researchers don't have the hardware to build the tools they need.

Strategies like Transfer Learning (taking a big model and teaching it a small new trick) are essential for democratizing AI.

YOLOv11 Model Training Results

Performance Curves

These plots illustrate the model's performance metrics over the training epochs.

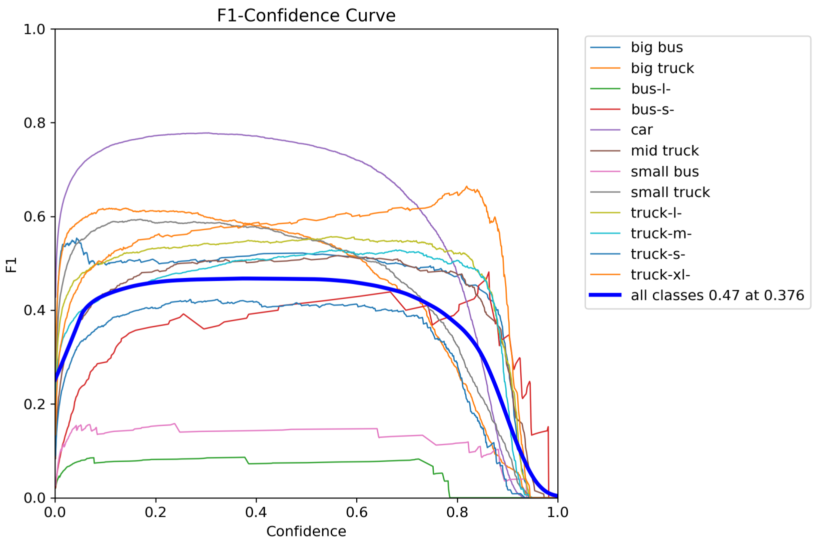


Figure 5.1: Box F1 Curve. Measures the harmonic mean of precision and recall for bounding box localization over training epochs.

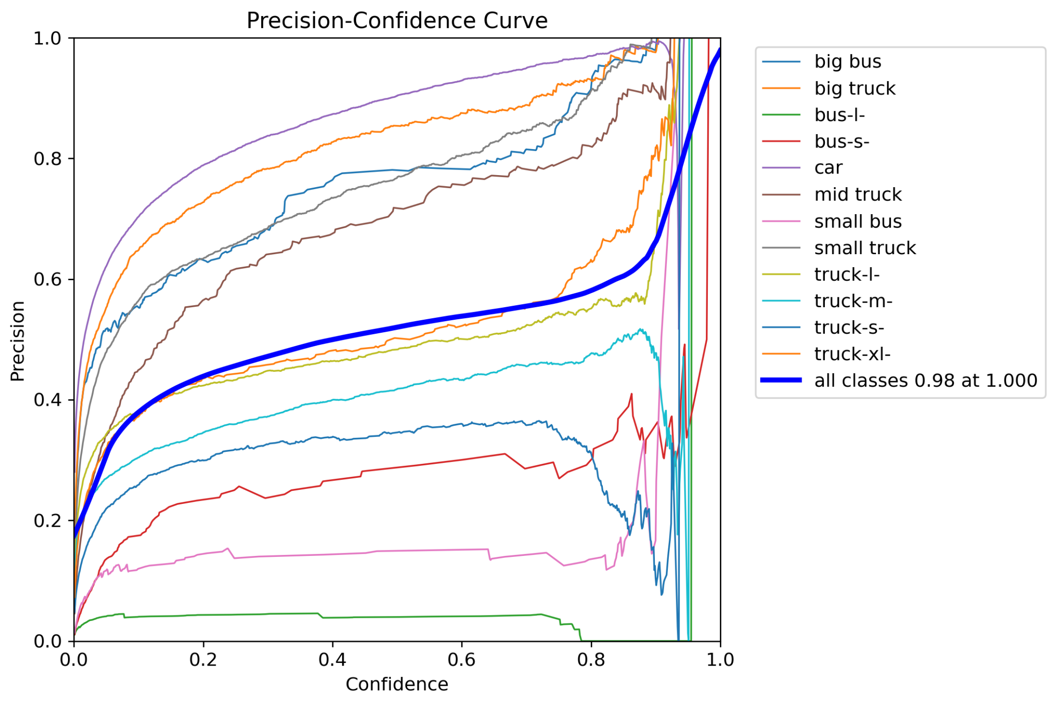


Figure 5.2: Box Precision (P) Curve. Shows the accuracy of positive predictions (True Positives / All Positives) over training epochs.

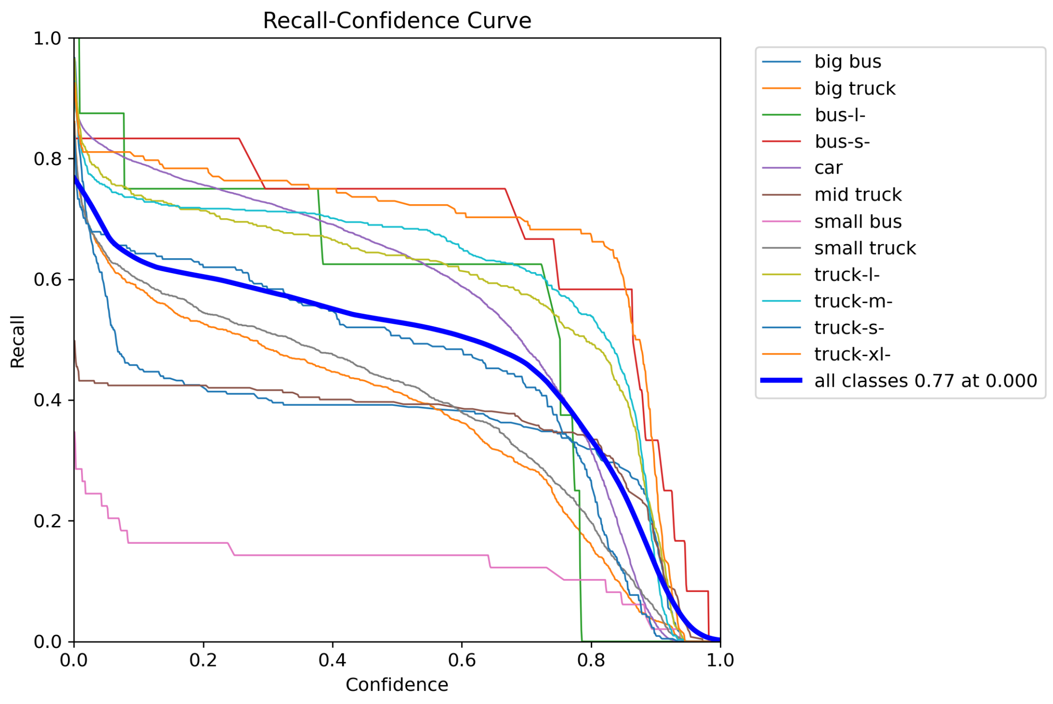


Figure 5.3: Box Recall (R) Curve. Shows the model's ability to find all positive samples (True Positives / All Real Positives) over training epochs.

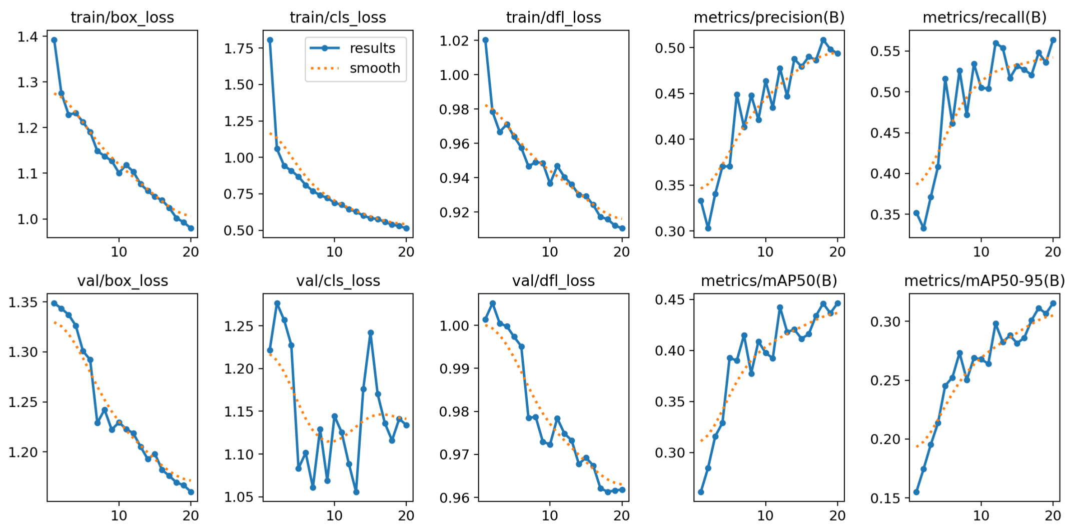


Figure 5.4: Training Summary Plot. A comprehensive overview of all training and validation metrics (loss, precision, recall, mAP) across all epochs.

Model Validation and Visualization

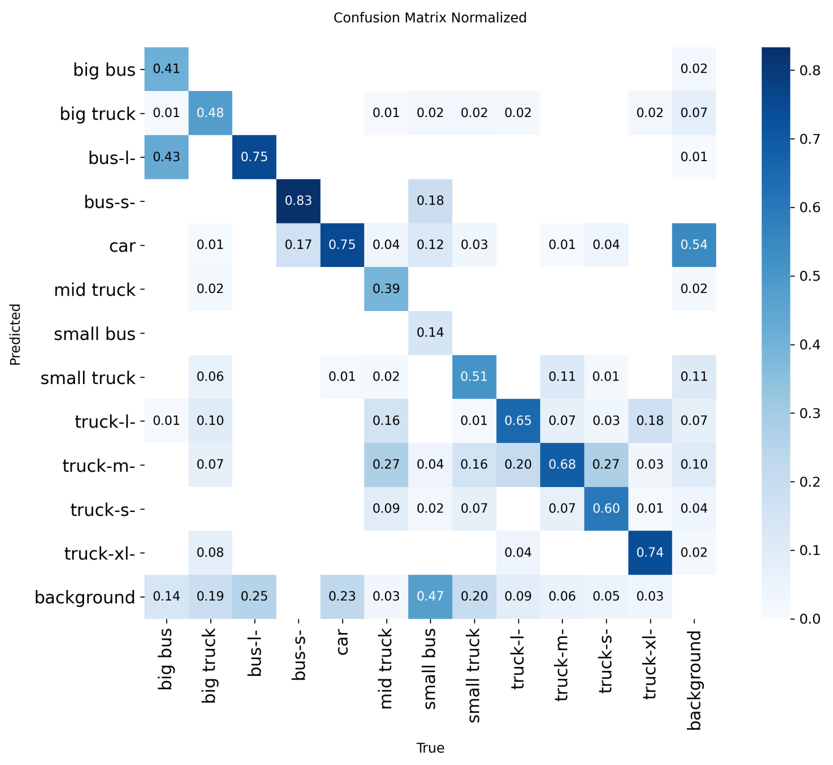
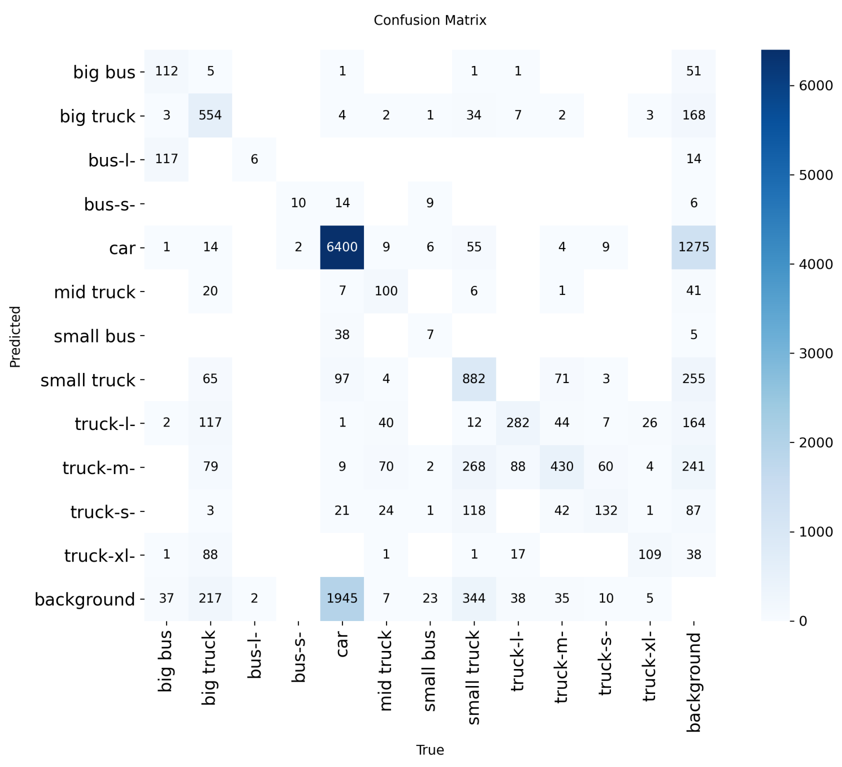


Figure 5.5: Normalized Confusion Matrix. Illustrates the percentage distribution of true and predicted classes, vital for analyzing class imbalance and misclassification rates.



Predicted \ True	big bus	big truck	bus-l-	bus-s-	car	mid truck	small bus	small truck	truck-l-	truck-m-	truck-s-	truck-xl-	background
big bus	112	5			1			1	1				51
big truck	3	554			4	2	1	34	7	2		3	168
bus-l-	117		6										14
bus-s-				10	14		9						6
car	1	14		2	6400	9	6	55		4	9		1275
mid truck		20			7	100		6		1			41
small bus					38		7						5
small truck		65			97	4		882		71	3		255
truck-l-	2	117			1	40		12	282	44	7	26	164
truck-m-		79			9	70	2	268	88	430	60	4	241
truck-s-		3			21	24	1	118		42	132	1	87
truck-xl-	1	88				1		1	17			109	38
background	37	217	2		1945	7	23	344	38	35	10	5	

Figure 5.6: Raw Confusion Matrix. Shows the absolute counts of true and predicted classes for the validation set.

Validation Sample Predictions

The following figures show examples from the validation dataset, comparing the ground truth labels against the model's predictions.

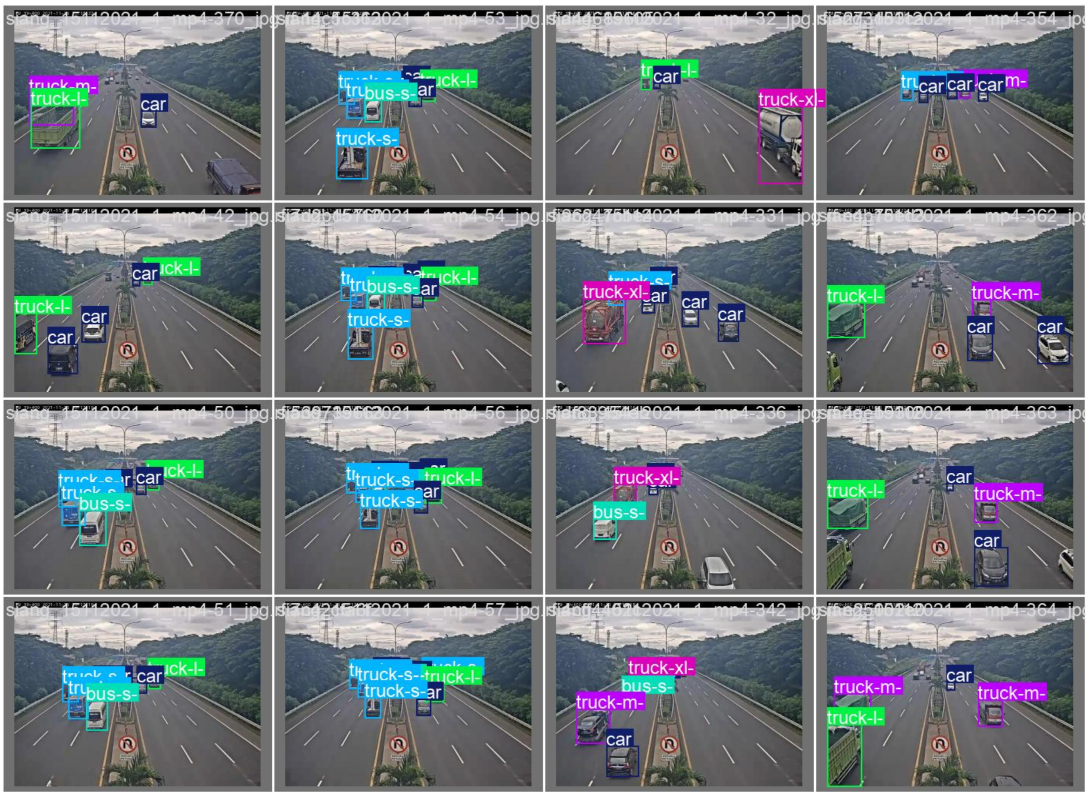


Figure 5.7: Validation Batch 0: Ground Truth Labels.

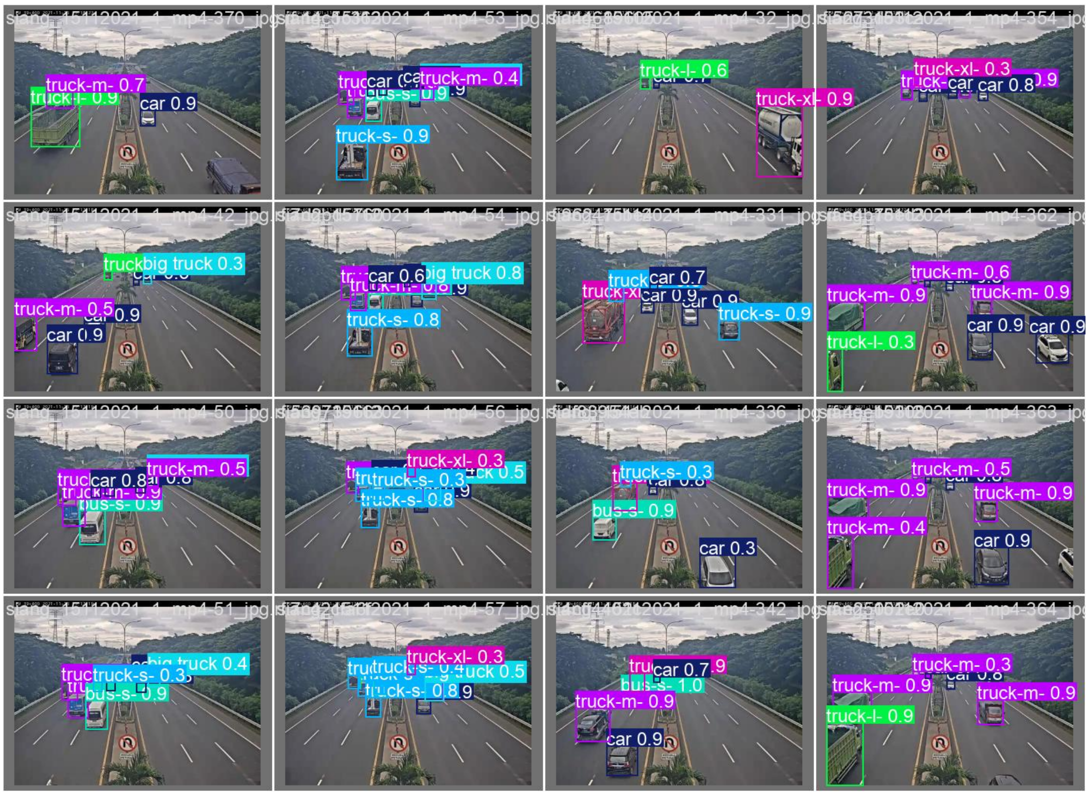


Figure 5.8: Validation Batch 0: Model Predictions.

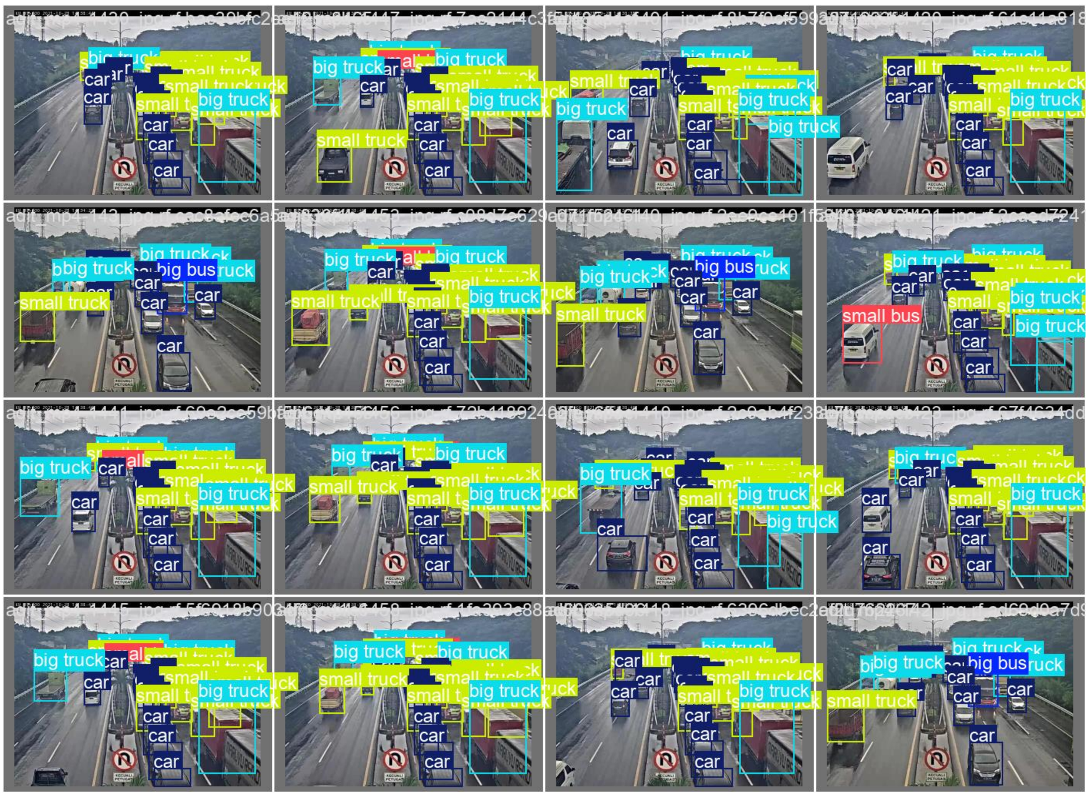


Figure 5.9: Validation Batch 1: Ground Truth Labels.

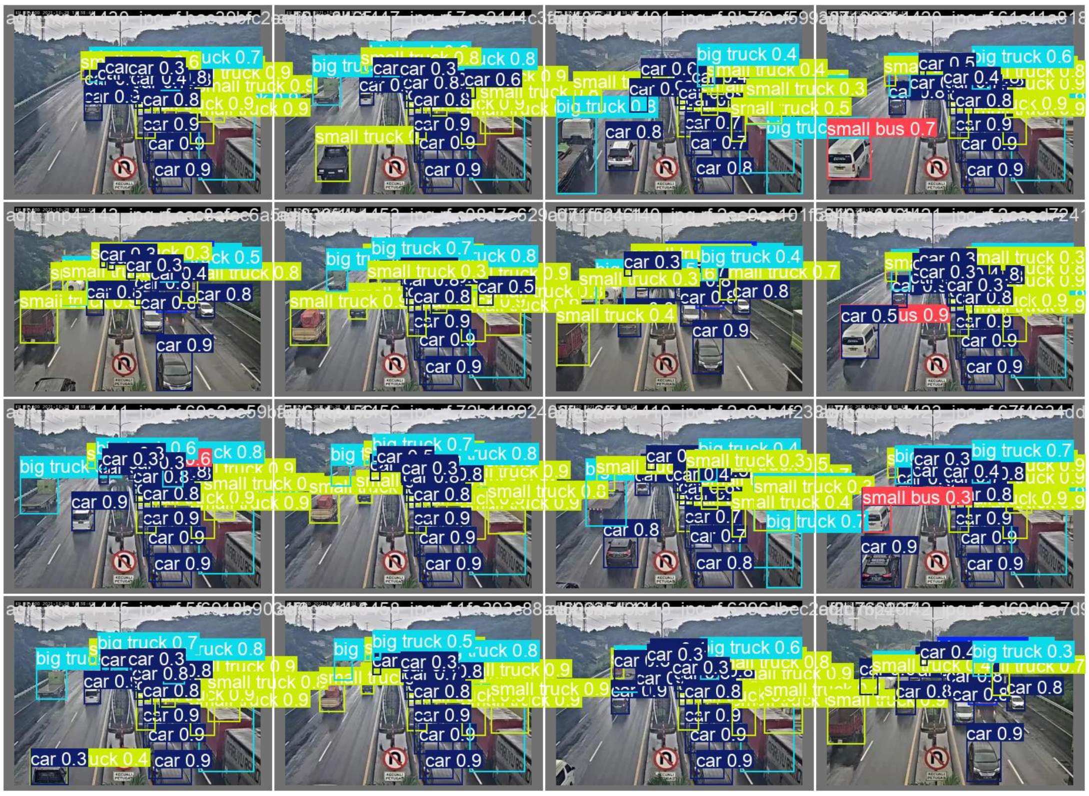


Figure 5.10: Validation Batch 1: Model Predictions.

References

1. Jocher, G., et al. (2023). *Ultralytics YOLOv8 and v11.* GitHub.

2. Zhang, Y., et al. (2022). *ByteTrack: Multi-Object Tracking by Associating Every Detection Box.* ECCV.

3. Redmon, J., et al. (2016). *You Only Look Once: Unified, Real-Time Object Detection.* CVPR.

4. Espressif Systems. (2023). *ESP32-CAM Datasheet.*

5. Telegram API Documentation. (2024). *Bots: An introduction for developers.*

Appendix A

Hardware Setup Guide

Wiring the ESP32-CAM

To program the ESP32-CAM, you need an FTDI adapter (USB-to-Serial converter).

1. Connect **5V** to **5V**.
2. Connect **GND** to **GND**.
3. Connect **U0T** to **RX**.
4. Connect **U0R** to **TX**.
5. **Important:** Connect IO0 to GND to put the board in "flashing mode."
6. Or, just use MB Board for connections.

Appendix B

Software Installation

B.1 Prerequisites

You need Python 3.8+ and an NVIDIA GPU with CUDA installed.

B.2 Installation Commands

Run the following in your terminal:

```
# 1. Create a virtual environment
python -m venv traffic_env
source traffic_env/bin/activate  # Windows: traffic_env\Scripts\
    activate

# 2. Install PyTorch (Ensure CUDA version matches)
pip install torch torchvision --index-url https://download.pytorch.org/
    whl/cu118

# 3. Install YOLO and OpenCV
pip install ultralytics opencv-python numpy

# 4. Install Telegram support
pip install python-telegram-bot
```

Listing B.1: Installation Script