Technical Report: Real Time Traffic Monitoring System

Introduction

This project implements a real time traffic monitoring system using a YOLO based object detection model (YOLOv8s) in ONNX format combined with ByteTrack for multi-object tracking. The system provides a GUI for live video feed inference, object counts, FPS plotting, and system resource monitoring.

Model and Algorithm Justification (Chosen Detection Model YOLOv8s ONNX format)

Reasoning: YOLO provides high detection speed with reasonable accuracy making it suitable for real time traffic monitoring. As for why I chose YOLOv8s instead of other YOLO variants such as YOLOv9 or YOLO26s or YOLO26n is because, YOLOv9 model's size is much larger and inference time is higher than YOLOv8s but it gives more accuracy but YOLOv8s gives close to the same amount of accuracy while its inference time is lower and even though YOLO26s and YOLO26n were not chosen is because after training I saw that their inference speed is fast but their accuracy is bad thus I chose YOLOv8s for its reasonable accuracy and inference speed balance.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed T4 TensorRT10 (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	**80.4**	**1.47**	3.2	8.7
YOLOv8s	640	44.9	128.4	2.66	11.2	28.6
YOLOv8m	640	50.2	234.7	5.86	25.9	78.9
YOLOv8l	640	52.9	375.2	9.06	43.7	165.2
YOLOv8x	640	53.9	479.1	14.37	68.2	257.8
YOLOv9t	640	**38.3**	-	2.3	**2.0**	**7.7**
YOLOv9s	640	**46.8**	-	3.54	**7.1**	**26.4**
YOLOv9m	640	**51.4**	-	6.43	**20.0**	**76.3**
YOLOv9c	640	53.0	-	7.16	25.3	102.1
YOLOv9e	640	**55.6**	-	16.77	57.3	189.0

This is taken from the official ultralytics page here is the link.

ONNX format: Using ONNX allows compatibility across frameworks and enables GPU acceleration through ONNX runtime. But my cuDNN in my pc was having some issues so the system kept using the CPU instead that is why in the demonstration video the CPU usage reached very high.

Tracking Algorithm (ByteTrack): ByteTrack is state of the art tracking algorithm optimized for speed and robustness. It maintains object identities across frames, reduces ID switching, efficient and works well with YOLO detection models (ultralytics module has built-in ByteTrack tracker).

Optimization Techniques

GPU Acceleration: PyTorch with CUDA for YOLO inference and ONNX runtime with CUDAExecutionProvider for ONNX model execution (Defaulted to CPUExecutionProvider later due to my GPU issues during system demonstration).

Frame Skipping and FPS Averaging: Processed every frame but updated FPS metrics using a rolling average to reduce fluctuations.

Efficient Drawing: Used OpenCV for frame annotation besides that, optimized text and rectangle rendering with precomputed colors. I used streamlit first for the GUI but the web UI caused lagging a little bit.

Resource Monitoring: Only updates system metrics every 3 frames to reduce CPU and GPU load.

Performance Analysis

Table 1: Global Metrics (All Classes)

Metric	Value
Precision	0.807
Recall	0.539
mAP@50	0.662
mAP@50-95	0.385

We can see that, the model shows high precision which indicates that most detections are correct but a moderate recall shows that the model misses some objects, especially small or occluded vehicles. The mAP@50 shows good detection accuracy and the mAP@50-95 shows stricter IoU thresholds and dataset difficulty.
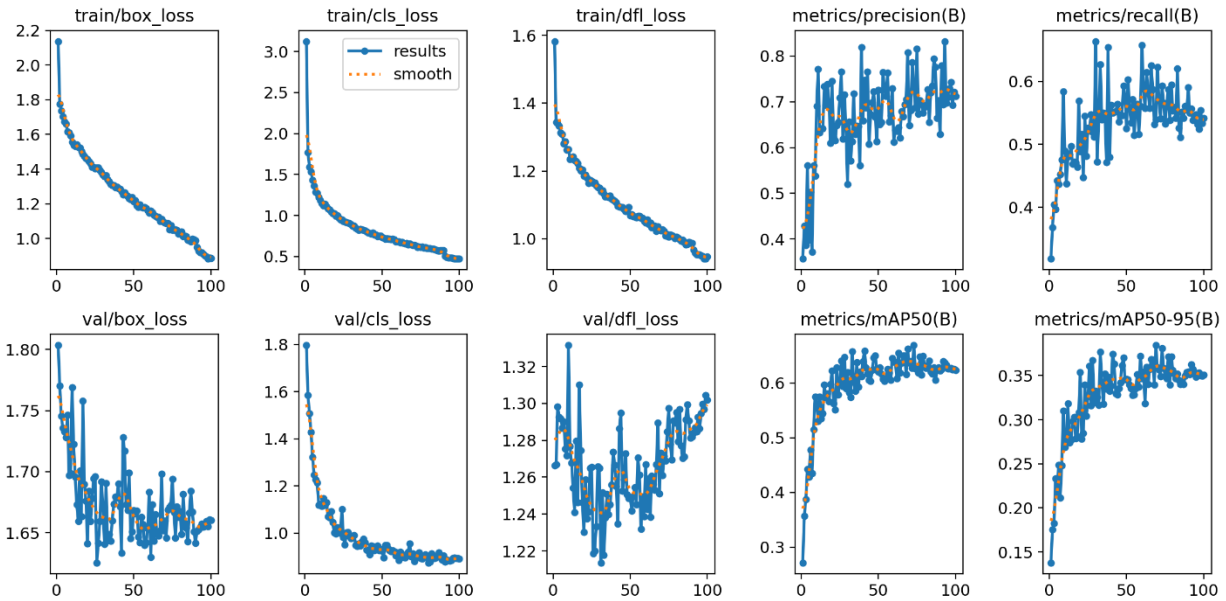

Figure 1: Model Results

System Analysis During Real Time Inference

The system shows the average FPS around 9 FPS and due to the GPU issues the GPU was not used and the model used the CPU instead, which caused the CPU usage to spike around 90% (Other software and other system usage included so around 30%-40% CPU usage). But there were some FPS spikes and CPU usage spikes when there were too many objects to detect but the model was able to run smoothly during the real time inference and the Start, Stop, Pause and Resume worked perfectly without any issues. But with more data the model can be improved and with some minor fixes the GPU usage can be utilized so that the CPU does not have to bear all of the load of the model.

The system maintained stable performance with moderate resource consumption.

Setup Instructions

First clone the repository using

git clone https://github.com/Platinum-Pluto/Traffic_Monitoring_Object_Tracking.git

Then navigate to the folder using

cd Traffic_Monitoring_Object_Tracking

After that create a virtual environment using

Python -m venv venv

Venv\Scripts\activate

Then install the dependencies using

pip install -r requirements.txt

But I used uv project manager as it is fast, efficient and less error prone. Using uv the whole setup can be done with a few commands easily after cloning the repository.

Here is the link to installing uv official documentation:

https://docs.astral.sh/uv/getting-started/installation/

After installing uv in your OS run the command below inside the project folder

uv sync

This is use the pyproject.toml file and install all the dependencies and setup the environment and replicates the entire environment.

Then finally use the command below to run the project code in the uv environment

uv run main.py

This will run the project code in the uv environment with all the dependencies installed.