International Scientific Journal of Engineering and Management (ISJEM) ISSN: 2583-6129
Volume: 05 Issue: 04 | April – 2026 DOI:10.55041/ISJEM06937
An International Scholarly || Multidisciplinary || Open Access || Indexing in all major Database & Metadata

Smart City Traffic Flow Optimization using Reinforcement Learning

Dr Sanjiv Shukla[1], **Mr.Darshan Rajendra Ahire**[2], **Bandi Ramteja**[3], **Banala Nikhil Kumar Reddy**[4],**Choppari Shoba Raju**[5], **Gujjula Shiva Kumar Reddy**[6]

[1]Professor (HOD), Computer Science & Engineering, Sandip University, Nashik, Maharashtra, India
[2]Professor (Guide), Computer Science & Engineering, Sandip University, Nashik, Maharashtra, India
[3456]Scholar, Sandip University, Nashik, Maharashtra, India

Abstract

Urban traffic congestion is one of the most pressing challenges in modern smart city development. Conventional fixed-timer traffic signals fail to adapt to real-world traffic dynamics, causing unnecessary delays, increased fuel consumption, and elevated carbon emissions. This research paper presents a Smart Traffic Signal Control System that leverages computer vision, deep learning (YOLOv8 object detection), and reinforcement learning (Q-Learning) to dynamically allocate green-signal durations and optimize lane scheduling based on real-time vehicle density and type across four independent lanes. The system integrates a FastAPI backend with OpenCV-based video processing, a React.js real-time dashboard, a region-of-interest (ROI) based lane management module, and a Q-Learning agent for intelligent lane priority scheduling.

Vehicle classes including cars, motorcycles, buses, trucks, and bicycles are weighted differently to compute proportional green-light timers, while the Q-Learning agent learns optimal lane activation sequences to minimize overall intersection waiting time. Experimental evaluations demonstrate average vehicle detection accuracy above 92% and green-time allocation efficiency improvements of up to 38% compared to fixed-cycle systems. The RL agent converges within 200 episodes and reduces average waiting time by an additional 15-22% over the baseline weighted model. The platform supports both live camera feeds and user-uploaded images, making it versatile for deployment in diverse real-world intersections.

Keywords: YOLOv8, Computer Vision, OpenCV, Smart Traffic Signal, Vehicle Detection, FastAPI, React.js, Adaptive Traffic Control, Deep Learning, ROI Detection, Reinforcement Learning, Q-Learning, Smart City.

1. Introduction

The rapid urbanisation of cities worldwide has resulted in exponential growth in vehicle populations, placing enormous strain on existing road infrastructure. According to the World Health Organization, traffic congestion costs the global economy an estimated $1 trillion annually through lost productivity, wasted fuel, and environmental degradation. Traditional traffic management systems rely on fixed pre-programmed signal cycles that remain oblivious to actual traffic conditions. This leads to wasteful green phases for empty roads and insufficient green time at high-density intersections.

Smart traffic signal systems powered by artificial intelligence offer a transformative alternative by continuously analysing live traffic data and dynamically adapting signal timings in real time. Computer vision, enabled by advances in convolutional neural networks (CNNs) and object detection algorithms, has made it feasible to automatically detect, classify, and count vehicles from standard CCTV footage. The YOLO (You Only Look Once) family of detectors, particularly YOLOv8, achieves state-of-the-art accuracy and inference speed, making it ideal for embedded and server-side traffic monitoring applications.

This project, the Advanced Smart Traffic Management System, implements a complete end-to-end pipeline: from video ingestion and vehicle detection through OpenCV, to adaptive signal timing via a FastAPI server, and finally to a React.js operator dashboard that visualises per-lane statistics in real time. Unlike existing solutions that focus solely on vehicle counting, our system incorporates vehicle-type-aware weighting, polygon-based region-of-interest filtering, and asynchronous multi-lane coordination.

1.1 Motivation

Indian cities such as Nashik, Pune, and Mumbai experience severe traffic congestion during peak hours, where fixed-timer signals create cascading delays across intersections. Emergency vehicles are often stuck in gridlocks because existing systems cannot prioritize urgent traffic. The motivation behind this project is to develop an affordable, camera-based intelligent traffic control system that can be retrofitted onto existing signal infrastructure without requiring expensive inductive loop sensors or radar equipment.

1.2 Problem Statement

Existing fixed-cycle traffic signal systems allocate equal green-time durations to all lanes irrespective of real-time traffic conditions. This results in: (a) excessive waiting times on low-traffic lanes, (b) insufficient clearance times on congested lanes, (c) increased fuel consumption and emissions from idling vehicles, and (d) inability to respond to sudden traffic surges or emergency situations. There is a pressing need for an adaptive system that uses real-time visual data to dynamically compute optimal green-signal durations for each lane.

1.3 Objectives

The primary objectives of this research are: (1) to design and implement a weighted vehicle-counting algorithm that assigns different traffic-load weights to different vehicle classes; (2) to develop a configurable region-of-interest (ROI) polygon tool that restricts detection to the relevant lane zone; (3) to build a fully asynchronous backend capable of serving simultaneous MJPEG video streams and REST API calls; (4) to create an interactive React.js dashboard for real-time traffic monitoring; and (5) to evaluate the system's performance against fixed-cycle baselines under various traffic scenarios.

1.4 Scope of the Study

This study focuses on a four-lane intersection model with independent camera feeds for each lane. The scope includes vehicle detection and classification using pre-trained COCO weights, adaptive green-time computation, real-time video streaming, and operator dashboard development. Pedestrian detection, vehicle tracking across frames, and multi-intersection coordination are outside the current scope but are identified as future enhancements.

2. Literature Review

Significant research has been conducted on intelligent traffic signal control over the past two decades. Roess et al. [1] provided foundational models of signalised intersection analysis used in conventional controllers. Their work established the mathematical basis for cycle-length optimization but assumed static traffic patterns. Redmon et al. [2] introduced the original YOLO architecture, revolutionizing real-time object detection by framing it as a single regression problem, achieving 45 FPS on standard hardware while maintaining competitive accuracy.

Wang et al. [3] demonstrated that CNN-based vehicle detectors outperform classical background-subtraction methods under varying illumination and occlusion conditions. Their comparative study showed 15-20% accuracy improvement over traditional methods. Abdulhai et al. [4] explored reinforcement-learning-based traffic signal controllers that adapt to time-of-day patterns but required expensive simulation environments and extensive training data. Liang et al. [5] combined YOLO detection with SORT tracking to estimate queue lengths at intersections, achieving real-time performance on GPU hardware with 91% accuracy.

More recent works have integrated edge-computing devices such as NVIDIA Jetson with YOLO detectors for on-device inference, eliminating the need for cloud connectivity [6]. Buch et al. [7] reviewed video-based traffic monitoring comprehensively and identified that polygon-based ROI masking significantly reduces false positives in multi-lane scenarios. Zhang et al. [8] proposed a density-aware signal control algorithm that weighted vehicles by type, similar to our approach, but relied on custom-trained models rather than transfer learning from COCO weights.

However, most academic prototypes lack complete software stacks including operator dashboards, REST APIs, and persistent data logging. The present system addresses these gaps by providing a production-grade full-stack architecture combining YOLOv8 detection with OpenCV preprocessing, FastAPI middleware, and a React.js frontend.

Table 1: Comparison of Related Works

Author(s)	Year	Method	Accuracy	Real-Time	Full Stack
Roess et al.	2004	Fixed-cycle model	N/A	No	No
Redmon et al.	2016	YOLOv1	63.4% mAP	Yes	No
Wang et al.	2023	YOLOv7 CNN	89.2%	Yes	No
Abdulhai et al.	2003	Reinforcement Learning	78%	No	No
Liang et al.	2019	YOLO + SORT	91%	Yes	No
Proposed System	2025	YOLOv8 + OpenCV + FastAPI	92.5%	Yes	Yes

3. System Architecture

The system follows a three-tier architecture comprising: a computer-vision processing layer, a REST API middleware layer, and a web-based presentation layer. Each tier is independently deployable and communicates through well-defined interfaces, enabling horizontal scaling and technology upgrades without system-wide disruption.

3.1 Computer Vision Layer (OpenCV + YOLOv8)

OpenCV (cv2) manages all video I/O operations. The VideoStream class opens a webcam or video file via cv2.VideoCapture, reads frames in a dedicated executor thread, and exposes them as NumPy arrays. Frames are passed to the YOLOv8 nano model (yolov8n.pt) for object detection. Only COCO class IDs corresponding to vehicles are retained: bicycle (1), car (2), motorcycle (3), bus (5), and truck (7). Each detected bounding box is filtered by verifying that its centroid lies within the lane's ROI polygon using a binary mask generated with cv2.fillPoly. Surviving detections are weighted and aggregated into a traffic-load score.

The vision layer also performs frame preprocessing including resizing to 640x480 for consistent inference, color space normalization, and optional histogram equalization for low-light conditions. Annotated frames with bounding boxes, class labels, confidence scores, and ROI overlays are encoded as JPEG and streamed via MJPEG to the frontend.

3.2 Backend Middleware (FastAPI)

The FastAPI application exposes a RESTful API and an MJPEG streaming endpoint. Key routes include: /api/status (traffic state), /api/start and /api/stop (controller lifecycle), /api/roi/{lane_id} (save/load ROI polygons), /api/stream/{lane_id} (live annotated video), /api/capture/{lane_id} (single frame analysis), /api/upload-image/{lane_id} (uploaded image analysis), and /api/reset/{lane_id} (reset lane).

The TrafficController runs an asynchronous cyclic loop that iterates through lanes 1-4, analyses each lane, sets it active, counts down the computed green time, and moves to the next lane. FastAPI's async capabilities ensure that API requests are served concurrently with the control loop, preventing blocking during long green phases. CORS middleware is configured to allow cross-origin requests from the React frontend development server.

3.3 Frontend Dashboard (React.js + Vite)

The operator dashboard is built with React.js and bundled with Vite for fast hot-module replacement during development. It displays four LaneCard components, each showing: vehicle count, traffic weight, computed green time, current signal state (green/red), and a live MJPEG video feed. An ROI Setup page allows operators to click polygon vertices on a canvas snapshot to define the detection zone for each lane. The dashboard auto-refreshes status data every 500ms via polling.

3.4 Database and Logging Layer

All traffic events, detection counts, and signal transitions are logged to a SQLite database via SQLAlchemy ORM. The logging layer records timestamps, lane IDs, vehicle counts by class, computed green times, and actual signal durations. This historical data enables post-hoc analysis of traffic patterns and system performance evaluation. Session-based authentication ensures only authorized operators can modify ROI configurations or start/stop the controller.

3.5 Communication Protocol

Inter-component communication follows a hybrid REST + streaming model. Status queries and control commands use standard HTTP REST calls with JSON payloads. Video feeds use server-sent MJPEG streams over HTTP, which are natively supported by HTML img elements without requiring WebSocket connections. ROI polygon data is serialized as JSON arrays of [x, y] coordinate pairs and persisted to disk as JSON files for each lane.

4. OpenCV Features and Functions Used in This Project

OpenCV is the primary image-processing backbone of the system. The following OpenCV features and functions are extensively utilised throughout the detection and streaming pipeline:

OpenCV Function / Feature	Purpose in This Project
cv2.VideoCapture	Opens webcam (index) or video file; manages frame buffering and release.
cap.read()	Reads the next frame from the video source as a BGR NumPy array.
cap.set(cv2.CAP_PROP_POS_FRAMES, 0)	Resets video file to frame 0 for seamless looping.
cv2.imencode('.jpg', frame)	Encodes a NumPy frame to JPEG bytes for MJPEG streaming.
cv2.imdecode(buf, cv2.IMREAD_COLOR)	Decodes uploaded JPEG/PNG bytes back to a NumPy array.
cv2.fillPoly(mask, [pts], 255)	Fills a polygon region on a binary mask for ROI definition.
cv2.polylines(frame, [pts], ...)	Draws the ROI polygon outline on the annotated frame.
cv2.rectangle(frame, pt1, pt2, color, th)	Draws bounding boxes around detected vehicles.
cv2.putText(frame, label, org, font, ...)	Overlays vehicle class and confidence labels.
np.zeros(shape, dtype=np.uint8)	Creates blank binary mask used for polygon ROI filtering.
mask[cy, cx]	Pixel-level centroid-in-ROI check for lane assignment.
cv2.resize(frame, (w, h))	Resizes frames to standard 640x480 for consistent inference.
cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)	Converts color space for model input compatibility.
cv2.FONT_HERSHEY_SIMPLEX	Font constant used for bounding-box annotation text rendering.
cv2.IMWRITE_JPEG_QUALITY	Controls JPEG compression quality (75) for streaming bandwidth.

Beyond the core functions listed above, OpenCV's contour detection (cv2.findContours) and morphological operations (cv2.dilate, cv2.erode) are available as fallback methods for scenarios where deep learning inference is too slow on resource-constrained hardware. The system architecture allows hot-swapping between YOLO-based detection and classical OpenCV background subtraction methods through a configuration flag, providing flexibility across different deployment environments.

5. Methodology

The adaptive green-time calculation follows a weighted vehicle-density model. Each detected vehicle class is assigned a traffic-load weight reflecting its physical footprint, manoeuvre time at intersections, and contribution to overall congestion:

Table 3: Vehicle Class Weights

Vehicle Class	COCO Class ID	Traffic Weight	Justification
Bicycle	1	0.8	Small footprint, fast clearance
Car	2	1.0	Standard reference vehicle
Motorcycle	3	0.8	Small, manoeuvrable
Bus	5	2.0	Large footprint, slow acceleration
Truck	7	2.5	Largest footprint, slowest clearance

5.1 Green-Time Computation Algorithm

The total weight for a lane is computed as W = sum(weight_i) for all detected vehicles i within the ROI. The green-signal duration T is then calculated as: T = clamp(W / 2, T_min, T_max), where T_min = 5 seconds and T_max = 35 seconds. The divisor of 2 was empirically selected to balance throughput and fairness across lanes. This proportional model requires no training data, executes in O(N) time where N equals the number of detections, and produces intuitively understandable outputs for traffic operators.

5.2 Cyclic Lane Controller

The cyclic controller iterates through lanes 1 -> 2 -> 3 -> 4 indefinitely. Before activating each lane's green phase, a fresh frame is captured and analysed to reflect current conditions. The algorithm proceeds as follows: (Step 1) Capture frame from lane camera; (Step 2) Run YOLOv8 inference; (Step 3) Filter detections by ROI polygon; (Step 4) Compute weighted traffic load; (Step 5) Calculate green duration; (Step 6) Set lane signal to GREEN; (Step 7) Countdown timer with async sleep; (Step 8) Set lane signal to RED; (Step 9) Move to next lane. Asyncio tasks ensure non-blocking countdown timers and concurrent API serving during green phases.

5.3 ROI-Based Lane Isolation

Each lane's detection zone is defined by a user-configurable polygon. The operator draws the polygon vertices on a live frame snapshot via the dashboard's canvas interface. The polygon coordinates are saved as JSON and loaded at detection time. A binary mask is created using cv2.fillPoly, and only detections whose centroid pixel falls within the white region of the mask are counted for that lane. This approach eliminates cross-lane contamination and background false positives, which are common in multi-lane intersection scenarios.

5.4 Confidence Threshold Tuning

The YOLOv8 confidence threshold is set to 0.35 based on empirical evaluation. Lower thresholds (0.20-0.30) introduced excessive false positives from shadows, road markings, and parked vehicles. Higher thresholds (0.50+) caused missed detections of partially occluded motorcycles and bicycles. The chosen threshold of 0.35 provides the optimal precision-recall trade-off for the target deployment environment.

6. Reinforcement Learning for Optimal Lane Scheduling

While the weighted vehicle-density model described in Section 5 effectively computes green-signal durations, the fixed cyclic lane order (1 to 2 to 3 to 4) may not be optimal under asymmetric traffic conditions. To address this limitation, a Reinforcement Learning (RL) module based on Q-Learning is integrated into the system. The RL agent learns which lane should receive the green signal next, while the existing weighted formula continues to determine how long each green phase lasts. This hybrid approach combines the interpretability of rule-based timing with the adaptability of learned scheduling policies.

6.1 Q-Learning Theory

Q-Learning is a model-free, off-policy reinforcement learning algorithm that learns an optimal action-selection policy by iteratively updating a Q-table. The Q-table maps state-action pairs to expected cumulative rewards. The core update rule is: Q(s, a) <- Q(s, a) + alpha * [r + gamma * max(Q(s', a')) - Q(s, a)], where alpha is the learning rate (0.1), gamma is the discount factor (0.9), r is the immediate reward, s is the current state, a is the action taken, and s' is the resulting next state. Over many episodes, the Q-values converge to optimal values, enabling the agent to select the best action (lane) for any given traffic state.

6.2 State Representation

The traffic state is represented as a tuple of discretized weight bins for all four lanes. Each lane's continuous weight sum is mapped to one of five discrete bins: empty (0-2), light (2-5), moderate (5-10), heavy (10-20), and very heavy (20+). This discretization reduces the state space from continuous to approximately 5^4 = 625 possible states, enabling efficient Q-table storage and faster convergence. The state is encoded as a string key (e.g., '2-0-3-1') for hash-table lookup in the Q-table.

Table: RL State Discretization Bins

Bin Index	Weight Range	Traffic Level	Example
0	0 - 2	Empty / Very Light	0-2 bicycles
1	2 - 5	Light	2-5 cars
2	5 - 10	Moderate	5-8 mixed vehicles
3	10 - 20	Heavy	10+ vehicles
4	20+	Very Heavy / Congested	15+ mixed with buses/trucks

6.3 Action Space

The action space consists of four discrete actions corresponding to the four lane IDs (1, 2, 3, 4). At each decision point (after a green phase completes), the agent selects which lane should receive the next green signal. A small penalty is applied for re-selecting the same lane consecutively to ensure fairness and prevent starvation of low-traffic lanes.

6.4 Reward Function Design

The reward function is designed to encourage serving high-traffic lanes while penalizing excessive waiting on other lanes. The reward for activating lane k is: R = W_k - 0.3 * sum(W_j for j != k), where W_k is the weight sum of the activated lane and W_j are the weight sums of the waiting lanes. This formulation balances throughput maximization (serving congested lanes yields positive reward) with fairness (making other lanes wait incurs a penalty proportional to their congestion level).

6.5 Exploration vs Exploitation

The agent uses an epsilon-greedy exploration strategy. Initially, epsilon is set to 1.0 (pure exploration), and it decays by a factor of 0.995 per episode until reaching a minimum of 0.05. During exploration, the agent randomly selects a lane (excluding the currently active lane when possible). During exploitation, the agent selects the lane with the highest Q-value for the current state. This decay schedule allows the agent to explore diverse traffic scenarios early in training while gradually shifting to optimal learned behavior.

6.6 Integration with Existing System

The RL module is designed as a drop-in enhancement that operates alongside the existing system without modifying any core detection or timing logic. When RL mode is enabled via the API (/api/rl/toggle), the controller replaces the fixed cyclic order with the Q-Learning agent's lane selection. When disabled, the system reverts to the default 1-2-3-4 cyclic order. The Q-table is automatically persisted to disk as a JSON file after every 20 episodes and upon controller shutdown, ensuring learned policies survive system restarts.

7. Dataset and Evaluation Setup

The system was evaluated on a workstation with an Intel Core i7-12700H CPU, 16 GB RAM, and an NVIDIA RTX 3060 GPU running Ubuntu 22.04. Two datasets were used for evaluation:

(a) Custom Intersection Video Corpus: 2 hours of video recorded at a four-way junction in Nashik, Maharashtra, during morning peak (8:00-9:00 AM) and evening peak (5:30-6:30 PM) hours. The video was captured using 1080p IP cameras mounted at 6-meter height with a 45-degree downward angle. Ground truth annotations were manually created for 500 randomly sampled frames, labeling all visible vehicles with bounding boxes and class labels.

(b) UA-DETRAC Benchmark: A public vehicle detection benchmark containing over 140,000 frames from 24 different locations with varying weather and lighting conditions. This dataset was used to validate the generalizability of the detection pipeline beyond our custom deployment scenario.

Table 4: Dataset Summary

Dataset	Frames	Vehicles Annotated	Classes	Resolution
Custom Nashik	500 (sampled)	3,847	5	1920x1080
UA-DETRAC	140,000+	1.2M+	4	960x540

8. Experimental Results and Performance Evaluation

8.1 Detection Accuracy

The YOLOv8n model was used with pre-trained COCO weights at a confidence threshold of 0.35. Detection performance was evaluated per vehicle class using precision, recall, and F1-score metrics computed against the manually annotated ground truth frames:

Table 5: Detection Performance by Vehicle Class

Vehicle Class	Precision (%)	Recall (%)	F1-Score (%)	Avg. Confidence
Car	94.2	93.8	94.0	0.81
Motorcycle	91.5	89.3	90.4	0.76
Bus	96.1	95.4	95.7	0.88
Truck	95.8	94.7	95.2	0.87
Bicycle	88.3	86.1	87.2	0.71
Overall	93.2	91.9	92.5	0.81

The overall F1-score of 92.5% demonstrates strong detection capability across all vehicle classes. Buses and trucks achieve the highest scores due to their larger size and distinct visual features. Bicycles show the lowest accuracy due to their small size and frequent occlusion by larger vehicles, particularly in dense traffic conditions.

8.2 Green-Time Allocation Efficiency

The adaptive green-time allocation was compared against a standard 30-second fixed-cycle system across four traffic density scenarios:

Table 6: Green-Time Allocation - Adaptive vs Fixed-Cycle

Scenario	Fixed Green (s)	Adaptive Green (s)	Improvement
Light traffic (<=3 vehicles)	30	8	+73.3% less wait
Moderate traffic (4-8 vehicles)	30	18	+40.0% less wait
Heavy traffic (9-15 vehicles)	30	28	Comparable
Mixed-vehicle heavy	30	35	Extended as needed

8.3 System Throughput and Latency

Metric	Value
YOLOv8n inference time (GPU)	~18 ms / frame
YOLOv8n inference time (CPU)	~110 ms / frame
MJPEG stream frame rate	15 FPS
REST API response time	< 25 ms
ROI polygon save/load latency	< 5 ms
End-to-end detection latency	~130 ms (CPU)
Dashboard refresh rate	500 ms polling
Concurrent stream capacity	4 lanes simultaneous

8.4 Accuracy Under Varying Conditions

The system was tested under six environmental conditions to evaluate robustness:

Condition	Detection Accuracy (%)	Green-Time Error (s)
Daylight (clear)	94.8	+-1.2
Daylight (cloudy)	93.1	+-1.5
Dusk / Dawn	89.7	+-2.1
Night (street-lit)	85.3	+-3.4
Rain (moderate)	87.2	+-2.8
Fog (light)	82.6	+-4.1

The system maintains above 85% accuracy in most conditions but shows degradation during fog and nighttime due to reduced contrast and increased noise in camera feeds. Future integration of infrared cameras could mitigate these limitations.

8.5 Reinforcement Learning Performance

The Q-Learning agent was evaluated over 500 training episodes using the custom Nashik intersection dataset. Performance metrics were recorded at regular intervals:

Episodes	Epsilon	Avg Reward	Q-Table Size	Wait Time Reduction vs Cyclic
0-50	1.0 - 0.78	-2.1	45	— (exploring)
50-100	0.78 - 0.61	0.8	128	5%
100-200	0.61 - 0.37	2.4	285	12%
200-300	0.37 - 0.22	3.1	380	18%
300-500	0.22 - 0.05	3.6	425	22%

The RL agent converges around 200 episodes, achieving stable positive rewards and consistent lane scheduling decisions. After convergence, the agent reduces average waiting time by 22% compared to the fixed cyclic order, primarily by prioritizing congested lanes during peak hours and avoiding unnecessary green phases for empty lanes during off-peak periods.

9. Technologies and Tools

Component	Technology / Library	Version
Object Detection	YOLOv8 (Ultralytics)	>= 8.2.18
Computer Vision	OpenCV	>= 4.9.0
Deep Learning	PyTorch	>= 2.5.0
Reinforcement Learning	Q-Learning (custom)	—
Array Processing	NumPy	>= 2.1.0
Backend Framework	FastAPI	>= 0.111.0
ASGI Server	Uvicorn	>= 0.30.1
Frontend Framework	React.js + Vite	Latest
Language	Python 3.11 / JS ES2022	—
Data Analytics	Pandas	>= 2.2.2
Vision Inference	TorchVision	>= 0.20.0
Database	SQLite + SQLAlchemy	>= 2.0
Authentication	JWT + bcrypt	—

10. Implementation Challenges

Several technical challenges were encountered during the development and deployment of the system. Understanding these challenges provides valuable insights for researchers working on similar projects.

10.1 Video Stream Synchronization

Managing four simultaneous video streams while maintaining real-time detection performance required careful thread management. OpenCV's VideoCapture is inherently blocking, so each camera feed was wrapped in a dedicated asyncio executor thread. Buffer management was critical to prevent memory leaks when frames accumulated faster than they could be processed. A circular buffer pattern with a maximum depth of 3 frames was implemented to ensure the system always processes the most recent frame rather than stale queued frames.

10.2 ROI Polygon Accuracy

Defining precise ROI polygons proved challenging due to perspective distortion in camera views. Vehicles at the far end of a lane appear significantly smaller than those near the camera. The polygon must encompass the entire lane region while excluding adjacent lanes and sidewalks. An iterative calibration process was developed where operators can adjust polygon vertices while viewing live detection results, enabling real-time feedback on polygon accuracy.

10.3 Model Inference Optimization

On CPU-only deployments, YOLOv8n inference takes approximately 110ms per frame, limiting the processing rate to about 9 FPS. To maintain acceptable performance, frame skipping was implemented where only every third frame is processed for detection while intermediate frames are streamed without annotation. This reduces the effective detection rate to 5 FPS while maintaining smooth 15 FPS video streaming to the dashboard.

11. Comparative Analysis

To contextualize the performance of the proposed system, a comparative analysis was conducted against three baseline approaches: (a) fixed-cycle signals, (b) inductive loop-based adaptive signals, and (c) a simpler background-subtraction-based computer vision system.

Feature	Fixed-Cycle	Inductive Loop	Background Sub.	Proposed System
Detection Method	None	Magnetic	Frame diff.	YOLOv8 CNN
Vehicle Classification	No	Limited	No	5 classes
Accuracy	N/A	~85%	~72%	92.5%
Installation Cost	Low	Very High	Low	Low
Maintenance	Minimal	High	Low	Low
Weather Robust	N/A	Yes	Poor	Good
Real-time Dashboard	No	No	No	Yes
Scalability	Low	Low	Medium	High

The proposed system outperforms all baselines in detection accuracy and feature completeness while maintaining low installation and maintenance costs. Unlike inductive loop sensors that require road surface cutting and periodic recalibration, our camera-based approach requires only mounting a standard IP camera at each lane and connecting it to the processing server. The full-stack architecture with a web dashboard is a unique differentiator not offered by any baseline system.

12. Discussion

The experimental results confirm that adaptive green-time allocation based on real-time vehicle detection delivers measurable efficiency gains over fixed-cycle systems, particularly in low-to-moderate traffic conditions where fixed systems over-allocate green time. The weighted vehicle model demonstrates that accounting for vehicle type produces

fairer signal allocation than simple vehicle counts, as heavy vehicles such as buses and trucks contribute more to congestion than bicycles or motorcycles.

The ROI polygon feature is critical for deployment accuracy. Without ROI masking, vehicles in adjacent lanes or on pavements contaminate the count. OpenCV's fillPoly-based mask efficiently restricts detection to only the relevant lane zone. The system's dual-mode operation, supporting both live camera feeds and image upload, makes it practical for real-time deployments as well as offline analysis or demonstration purposes.

The system's modular architecture facilitates independent upgrades of individual components. For example, replacing YOLOv8n with a larger model like YOLOv8m would improve accuracy at the cost of inference speed, but this change requires modification of only the detection module without affecting the frontend or API layers. Similarly, the React dashboard can be enhanced with historical analytics charts without modifying the backend logic.

A limitation of the current approach is the use of a fixed cyclic lane order (1 to 2 to 3 to 4), which may not be optimal under highly asymmetric traffic loads. Additionally, the system does not currently handle pedestrian crossings, emergency vehicle pre-emption, or multi-intersection coordination, which are essential features for production-grade traffic management.

13. Future Scope

Several enhancements are planned for future iterations of the system to improve its capabilities and real-world applicability:

(1) Priority-Queue Scheduling: Replace the fixed cyclic lane order with a priority-queue model where the lane with the highest accumulated weight score is served first. This would significantly improve performance under asymmetric traffic loads.

(2) Pedestrian Detection: Integrate pedestrian detection using YOLO's person class to implement safe pedestrian crossing phases. The system would automatically insert pedestrian green phases when pedestrians are detected waiting at crosswalk zones.

(3) Emergency Vehicle Pre-emption: Detect emergency vehicles (ambulances, fire trucks, police cars) through visual features (flashing lights, distinctive colors) and automatically grant them priority green signals regardless of the current cycle position.

(4) Edge Deployment: Port the detection pipeline to NVIDIA Jetson Nano or Raspberry Pi 5 with neural compute sticks for fully on-device inference, eliminating the need for a centralized server. TensorRT optimization could reduce inference time to under 10ms on edge hardware.

(5) Multi-Intersection Coordination: Extend the system to coordinate signals across multiple connected intersections using a graph-based optimization model. This would enable green-wave corridors for arterial roads, further reducing overall network congestion.

(6) Vehicle Tracking: Integrate SORT or DeepSORT tracking algorithms to track individual vehicles across frames. This would enable speed estimation, queue length measurement, and more accurate traffic flow analysis.

14. Conclusion

This paper presented the Advanced Smart Traffic Management System, a full-stack AI-powered solution for adaptive traffic signal control. By combining YOLOv8's high-speed vehicle detection with OpenCV's rich image-processing capabilities and a modern FastAPI + React.js architecture, the system achieves over 92% detection accuracy and reduces unnecessary waiting time by up to 73% in light-traffic scenarios.

The ROI polygon mechanism, weighted vehicle scoring, and asynchronous controller loop together form a robust, scalable pipeline suitable for real-world intersection management. The comprehensive evaluation across multiple traffic scenarios, lighting conditions, and weather conditions demonstrates the system's practical applicability. The comparative analysis confirms that the proposed approach outperforms traditional fixed-cycle systems, inductive loop sensors, and simpler computer vision methods in both accuracy and feature completeness.

The open-source, modular design allows straightforward extension to multi-camera networks, edge-device deployment, and integration with existing city-level traffic management infrastructure. Future work will focus on priority-queue scheduling, pedestrian detection, emergency vehicle pre-emption, and multi-intersection coordination to further enhance the system's capabilities for smart city deployment.

References

[1] Roess, R. P., Prassas, E. S., & McShane, W. R. (2004). Traffic Engineering (3rd ed.). Prentice Hall.

[2] Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. CVPR 2016.

[3] Wang, C. Y., Bochkovskiy, A., & Liao, H. Y. M. (2023). YOLOv7: Trainable Bag-of-Freebies Sets New State-of-the-Art. CVPR 2023.

[4] Abdulhai, B., Pringle, R., & Karakoulas, G. J. (2003). Reinforcement Learning for True Adaptive Traffic Signal Control. J. Transportation Engineering, 129(3).

[5] Liang, X., Du, X., Wang, G., & Han, J. (2019). Towards Real-Time Deep Vehicle Detection for Adaptive Traffic Signal Control. IEEE Trans. ITS.

[6] Ultralytics. (2023). YOLOv8 Documentation. https://docs.ultralytics.com

[7] Buch, N., Velastin, S. A., & Orwell, J. (2011). A Review of Computer Vision Techniques for the Analysis of Urban Traffic. IEEE Trans. ITS, 12(3).

[8] Zhang, Y., Wang, H., & Li, F. (2022). Density-Aware Adaptive Traffic Signal Control Using Deep Learning. IEEE Access, 10.

[9] OpenCV Developers. (2024). OpenCV 4.x Documentation. https://docs.opencv.org

[10] FastAPI Documentation. (2024). FastAPI Framework. https://fastapi.tiangolo.com

[11] Jocher, G., Chaurasia, A., & Qiu, J. (2023). Ultralytics YOLOv8. https://github.com/ultralytics/ultralytics

[12] Bewley, A., Ge, Z., Ott, L., Ramos, F., & Upcroft, B. (2016). Simple Online and Realtime Tracking. ICIP 2016.

[13] Lin, T. Y., et al. (2014). Microsoft COCO: Common Objects in Context. ECCV 2014.

[14] Wen, L., et al. (2020). UA-DETRAC: A Benchmark Suite for Multi-Object Detection and Tracking. arXiv:1511.04136.

[15] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. CVPR 2016.