Abstract

Traffic congestion is a significant issue in urban areas, largely exacerbated by inefficient fixed-time traffic signal controllers that do not adapt to real-time traffic conditions. This leads to unnecessary delays, increased fuel consumption, and heightened air pollution. This project presents the design and implementation of a prototype Smart Traffic Light Control System aimed at mitigating these issues. The system leverages computer vision and deep learning, specifically the YOLOv8 object detection model, to monitor traffic flow via a standard webcam. It identifies vehicles within predefined Regions of Interest (ROIs) corresponding to different lanes at an intersection. Real-time traffic density for each lane is calculated based on the area occupied by detected vehicles within their respective ROIs. A dynamic decision-making algorithm utilizes this density information to allocate green light time adaptively, prioritizing lanes with higher traffic volume and skipping empty lanes entirely. The system's status, including live lane densities and the current signal timer, is presented through a web-based dashboard built using Flask for the backend API and HTML/CSS/JavaScript for the frontend interface. This report details the system architecture, implementation specifics of each module, the technologies employed, observed results, challenges encountered, and potential avenues for future enhancement. The prototype demonstrates the feasibility of using real-time vision-based density analysis for more efficient traffic signal control.

Table of Contents

Chapter 1: Introduction

1.1. Background

Urban traffic congestion remains a persistent global challenge, impacting daily commutes, economic productivity, and environmental quality. A significant contributor to this inefficiency is the rudimentary control logic employed by many traditional traffic signal systems. These systems often operate on pre-programmed, fixed-time cycles, allocating green light intervals without considering the dynamic and often unpredictable nature of traffic flow at any given moment.

1.2. Motivation

The rigidity of fixed-time controllers frequently leads to scenarios where vehicles queue excessively on busy approaches while lanes with little or no traffic are unnecessarily given green signals. This mismatch between signal timing and actual demand results in increased travel times, driver frustration, wasted fuel, and consequently, higher emissions of greenhouse gases and pollutants. The motivation for this project stems from the clear need for more intelligent, adaptive traffic management systems that can perceive and respond to real-time traffic conditions.

1.3. Problem Statement

To design and implement a prototype traffic light control system that dynamically adjusts signal timings based on real-time, vision-based analysis of traffic density, aiming to reduce unnecessary waiting times and improve traffic flow efficiency compared to traditional fixed-time systems.

1.4. Proposed Solution

This project proposes an intelligent traffic management system prototype that utilizes computer vision for real-time traffic monitoring. The core components include:

- **Live Video Input:** Capturing video feed from a standard webcam overlooking an intersection.
- **Vehicle Detection:** Employing the YOLOv8 object detection model to identify and locate vehicles within the camera's view.
- **Lane Density Calculation:** Defining specific Regions of Interest (ROIs) for each lane and calculating traffic density based on the spatial occupancy of detected vehicles within these ROIs.
- **Adaptive Signal Control:** Implementing a decision-making algorithm that allocates green light time based on relative lane densities, prioritizing busier lanes and skipping empty ones.

- **Web-Based Monitoring:** Providing a dashboard accessible via a web browser to visualize the calculated densities and the current signal status (active green lane and remaining time) in real-time.

1.5. Objectives

- To integrate a real-time object detection model (YOLOv8) with live video input for vehicle identification.
- To develop a method for calculating lane-specific traffic density using ROIs and detected vehicle bounding boxes.
- To implement an adaptive algorithm that modifies traffic signal timings based on calculated densities.
- To create a web-based dashboard using Flask and standard web technologies (HTML, CSS, JS) to display real-time system data (densities, timer).
- To evaluate the feasibility and potential benefits of the proposed system compared to fixed-time approaches.

1.6. Scope and Limitations

- **Scope:** The project focuses on a single intersection scenario monitored by one camera. It includes vehicle detection, density calculation for up to four predefined lanes, adaptive green light timing logic, and data visualization on a web dashboard.
- **Limitations:**
    - The system relies on manual ROI configuration specific to the camera view.
    - Detection accuracy is dependent on the chosen YOLOv8 model variant, lighting conditions, weather, and potential occlusions.
    - The density metric is area-based and does not directly count vehicles or consider speed.
    - The decision logic is simplified and does not account for pedestrians, complex signal phases, or emergency vehicles.
    - The web dashboard displays data but not the live annotated video feed.
    - Performance is dependent on the processing hardware (CPU/GPU).

1.7. Report Organization

This report is structured as follows: Chapter 2 discusses background concepts and related work. Chapter 3 details the system's design and architecture. Chapter 4 elaborates on the implementation specifics of each module. Chapter 5 presents the results and discusses the system's performance and challenges. Finally, Chapter 6 concludes the report and suggests directions for future work.

Chapter 2: Background Study and Literature Review

2.1. Conventional Traffic Light Systems

Traditional traffic control relies predominantly on two types of systems:

- **Fixed-Time Controllers:** Operate on predetermined timing schedules that repeat cyclically. These schedules are typically based on historical traffic data but cannot adapt to short-term fluctuations or unexpected events.
- **Actuated Controllers:** Utilize sensors (typically inductive loops embedded in the pavement or radar) to detect the presence of vehicles at specific points. While they offer some level of responsiveness by extending green times or skipping phases with no detected demand, they often lack a comprehensive view of queue lengths or overall density.

2.2. Computer Vision in Traffic Management

Computer vision offers a powerful alternative or supplement to traditional sensors. Cameras provide a richer source of information, allowing for the analysis of entire scenes rather than just point detections. Applications include vehicle counting, speed estimation, queue length measurement, vehicle classification, and incident detection. The decreasing cost of cameras and advancements in processing power make vision-based systems increasingly viable.

2.3. Object Detection Models for Vehicle Recognition

Object detection is a core task in computer vision, aiming to identify and localize objects within an image or video frame. Deep learning models, particularly Convolutional Neural Networks (CNNs), have achieved state-of-the-art performance.

- **YOLO (You Only Look Once):** A family of real-time object detection models known for their speed and accuracy. YOLO processes the entire image at once, making it suitable for real-time video analysis.
- **YOLOv8:** The latest iteration (at the time of project inception) offers various model sizes (e.g., n-nano, s-small, m-medium) providing trade-offs between speed and accuracy. yolov8n.pt was selected for this project due to its lightweight nature, making it suitable for real-time processing on potentially resource-constrained hardware, while still providing reasonable accuracy for common vehicle types.

2.4. Web Technologies for Data Visualization

Presenting real-time system data requires effective visualization tools.

- **Backend Framework (Flask):** A lightweight Python microframework suitable for

building APIs. It allows the core Python processing logic to expose data easily over HTTP.

- **Frontend Technologies (HTML/CSS/JS):** The standard building blocks for web interfaces. HTML structures the content, CSS (enhanced by frameworks like Tailwind CSS for rapid styling) defines the presentation, and JavaScript handles dynamic updates by fetching data from the backend API and manipulating the Document Object Model (DOM).

Chapter 3: System Design and Architecture

3.1 Workflow Description:

1. **Frame Capture:** The backend continuously captures video frames from the webcam using OpenCV.
2. **Vehicle Detection:** Each frame is passed to the YOLOv8 model, which detects vehicles and outputs their bounding boxes, confidence scores, and class labels.
3. **ROI Filtering:** Detected vehicles are checked against the predefined ROIs for each lane. Only vehicles whose centers fall within an ROI are considered for that lane's density calculation.
4. **Density Calculation:** For each lane, the system calculates the percentage of the ROI area covered by the bounding boxes of vehicles assigned to it.
5. **Signal Decision Logic:** Based on the calculated densities (ignoring zero-density lanes), the algorithm determines which lane should receive the next green light and for how long (within defined minimum and maximum limits).
6. **Update Signal State:** The system updates the currently active green lane index and decrements the corresponding timer.
7. **Shared Data Store:** The latest calculated densities, active green lane index, and remaining timer value are stored in a shared Python dictionary, protected by a lock for thread safety.
8. **Flask API:** A Flask web server runs concurrently, providing an API endpoint (/api/traffic_data) that, when requested, reads the latest information from the shared data store and returns it as JSON.
9. **Web Dashboard:** The frontend HTML page uses JavaScript to periodically fetch data from the Flask API endpoint.
10. **DOM Update:** Upon receiving the JSON data, the JavaScript code updates the relevant parts of the HTML page (density values, signal status indicators, timer display) to reflect the real-time state.

3.2. Component Description

- **Video Input:** Standard webcam connected to the host machine.
- **Detection Module (detection.py):** Responsible for loading the YOLOv8 model and performing inference on input frames.
- **Utility Module (utils.py):** Contains helper functions for ROI definition, checking if points are within ROIs, and drawing functions (used primarily for visualization/debugging).
- **Density Module (density.py):** Implements the logic to calculate lane density based on bounding box areas within ROIs.
- **Decision Module (decision.py):** Contains the algorithm for adaptive signal

timing based on densities.

- **Backend Module (backend.py):** Orchestrates the main processing loop in a background thread, manages the shared data state, and runs the Flask web server providing the data API.
- **Frontend Module (templates/, static/):** Comprises the HTML structure, CSS styling (Tailwind), and JavaScript logic for the user-facing web dashboard.

3.3. Technology Stack and Justification

- **Python:** Chosen for its extensive libraries for computer vision (OpenCV), deep learning (PyTorch/Ultralytics), web development (Flask), and general-purpose programming. Its readability and large community support are also advantages.
- **OpenCV:** The standard library for computer vision tasks, used here for efficient video capture and potentially basic image pre-processing or drawing.
- **YOLOv8 (Ultralytics):** Selected for its state-of-the-art real-time object detection capabilities. The yolov8n variant offers a good balance between speed (essential for real-time video) and accuracy for common vehicle types on standard hardware.
- **Flask:** A lightweight and flexible Python web framework, ideal for creating the simple API needed to serve data to the frontend without the overhead of larger frameworks.
- **HTML/CSS/JavaScript:** The fundamental technologies for building web interfaces.
- **Tailwind CSS:** A utility-first CSS framework used to rapidly style the frontend dashboard directly within the HTML structure, reducing the need for extensive custom CSS.
- **NumPy:** Used for efficient numerical operations, particularly in handling bounding box data and density calculations.
- **Threading (Python threading module):** Used in the backend to run the computationally intensive video processing loop separately from the web server, preventing the processing from blocking web requests.

Chapter 4: Implementation Details

4.1. Development Environment Setup

The project was developed using Python 3.x. A virtual environment (venv) was utilized to manage project dependencies, ensuring isolation from system-wide packages. Key libraries installed include opencv-python, ultralytics, Flask, and numpy. Development was primarily done using the PyCharm IDE.

4.2. Detection Module (detection.py)

- **Model Loading:** The load_yolo_model function handles loading the specified YOLOv8 model file (defaulting to yolov8n.pt). It checks for CUDA availability to utilize GPU acceleration if possible, otherwise falling back to CPU. The function caches the loaded model in a global variable to avoid reloading on subsequent calls. It also extracts and stores the model's class names (model.names) and identifies the specific indices corresponding to the desired VEHICLE_CLASSES defined in utils.py. A warm-up inference is performed to potentially reduce latency on the first real detection task.
- **Vehicle Detection:** The detect_vehicles function takes a single frame (as a NumPy array) and the loaded model as input. It calls the model's inference method, passing parameters to filter by class_indices_to_detect and conf_threshold. The verbose=False argument suppresses detailed per-frame logging from the Ultralytics library. The function processes the results, extracts the bounding box data (.boxes.data), converts it to a NumPy array on the CPU, and returns it. Error handling is included for cases where the model is not loaded or inference fails.

4.3. Utility Module (utils.py)

- **ROI Definition:** The LANE_ROIS list is centrally defined here. **Crucially, these coordinates (x, y, width, height) must be manually calibrated based on the specific camera placement and perspective to accurately delineate the areas corresponding to each traffic lane.** Incorrect ROIs will lead to inaccurate density calculations.
- **Vehicle Classes:** The VEHICLE_CLASSES list defines which object types detected by YOLOv8 should be considered for density calculations.
- **Helper Functions:** Includes get_vehicle_class_indices (used by detection.py), is_within_roi (checks if a bounding box center falls within a given ROI rectangle), and drawing functions (draw_rois, draw_detections) primarily used for visual debugging (e.g., in the app_opencv.py version) but draw_detections also contains the logic to assign detections to specific lanes based on ROIs, which is used by

the backend even without drawing.

4.4. Density Calculation Module (density.py)

- The calculate_lane_density function takes the list of detections assigned to each lane (output from utils.draw_detections) and the list of ROI definitions.
- For each lane, it calculates the total area of its corresponding ROI (roi_area = roi_w * roi_h).
- It then iterates through the bounding boxes of vehicles detected within that lane, summing their individual areas (bbox_area = (x2 - x1) * (y2 - y1)).
- Density is computed as (total_vehicle_area / roi_area) * 100.
- The result is capped at 100% to handle cases where bounding boxes might overlap significantly.
- It returns a list containing the calculated density percentage for each lane.

4.5. Decision Logic Module (decision.py)

- **Constants:** Defines MIN_GREEN_TIME, MAX_GREEN_TIME, and thresholds like ZERO_DENSITY_THRESHOLD.
- **get_next_signal_state Function:** This is the core adaptive timing algorithm.
  - It filters out lanes with density below ZERO_DENSITY_THRESHOLD.
  - If all lanes are effectively empty, it cycles to the next lane sequentially with MIN_GREEN_TIME.
  - Otherwise, it identifies the lane with the highest density among the non-empty lanes.
  - It includes logic to potentially switch to the second-highest density lane if the current green lane still has the maximum density, preventing one lane from dominating unfairly when multiple lanes have traffic.
  - It calculates the green light duration for the selected lane, scaling it linearly between MIN_GREEN_TIME and MAX_GREEN_TIME based on the lane's density percentage.
  - It returns the index of the next lane to get the green light and its calculated duration.

4.6. Backend API Module (backend.py)

- **Flask Setup:** Initializes a Flask application, configuring it to serve static files (CSS, JS) from the static folder and render HTML templates from the templates folder.
- **Shared Data:** A global dictionary latest_data stores the system state (densities, timer, etc.). A threading.Lock (data_lock) ensures that updates and reads to this dictionary from different threads are synchronized. A threading.Event (stop_processing_flag) is used for clean shutdown.
- **Background Processing Thread:** The traffic_processing_loop function

encapsulates the entire pipeline (camera read -> detect -> density -> decision -> update state). This runs in a separate thread (daemon=True) initiated when the Flask app starts. It continuously updates the latest_data dictionary.

- **API Endpoint (/api/traffic_data):** A Flask route that, when accessed via a GET request, reads the current state from latest_data (using the lock) and returns it as a JSON response.
- **HTML Route (/):** Serves the main index.html page.
- **Execution & Shutdown:** The if __name__ == '__main__': block starts the background thread and then runs the Flask development server. It includes a finally block to set the stop_processing_flag and attempt to join the background thread upon server shutdown.

4.7. Frontend Dashboard (templates/, static/)

- **index.html:** Defines the HTML structure using semantic tags. It includes placeholders (elements with specific IDs like density-0, status-1, timer-0) for dynamic data. Tailwind CSS classes are used extensively for layout and styling (grid, cards, padding, text size, etc.). It links to the Tailwind CDN, the custom style.css, and the script.js.
- **style.css:** Contains minimal CSS, primarily defining .lane-green and .lane-red classes used by JavaScript to visually indicate the signal status on the lane cards (background/border color changes).
- **script.js:**
  - Defines the API_URL and UPDATE_INTERVAL.
  - Gets references to the necessary HTML elements using their IDs.
  - The fetchTrafficData function asynchronously fetches JSON data from the backend API endpoint.
  - The updateDashboard function takes the received JSON data and updates the textContent of the corresponding elements for densities and timers. It also updates the status text and dynamically adds/removes the .lane-green or .lane-red classes to the lane card elements to change their appearance based on the current_green_lane index.
  - Uses setInterval to call fetchTrafficData repeatedly at the specified interval, ensuring the dashboard stays updated.

Chapter 5: Results

5.1. System Functionality Demonstration

Upon execution, the system successfully initializes the webcam and loads the YOLOv8 model. The backend process begins capturing frames and performing detections. The web dashboard, accessed via a browser, connects to the backend API and starts displaying data.

- **Density Updates:** The density percentages displayed for each lane on the dashboard update in near real-time, reflecting changes in the number and size of vehicles detected within the predefined ROIs. Higher values are observed when more vehicles or larger vehicles occupy the ROI space.
- **Signal Status:** The dashboard correctly highlights the currently active green lane (e.g., changing the card background/border to green) and displays the corresponding countdown timer. Other lanes are marked as red.
- **Adaptive Timing:** The system demonstrates adaptive behavior. When a lane's density increases significantly, it is prioritized for the next green light phase. Lanes with zero or very low density are skipped in the green light cycle, preventing unnecessary waiting. The duration of the green light varies based on the density of the lane receiving it, within the configured minimum and maximum limits.

5.2. Performance Observations

- **Detection Speed:** The YOLOv8n model provides real-time performance, allowing the processing loop to run at a reasonable frame rate (e.g., 10-20 FPS, depending heavily on hardware).
- **CPU/GPU Usage:** Object detection is computationally intensive. CPU usage is significant when running on CPU only. If a compatible GPU is available and configured, performance improves considerably, and CPU usage decreases.
- **Dashboard Responsiveness:** The web dashboard updates smoothly due to the asynchronous JavaScript fetching data at regular intervals (e.g., every second). The load on the Flask backend per API request is minimal as it only reads the pre-calculated data.
- **Latency:** There is a small inherent latency between an event happening on the road, its capture, processing, and the data update appearing on the dashboard. This is typically within acceptable limits for this application type.

5.3. Analysis of Adaptive Control

The implemented system successfully demonstrates the core principle of adaptive traffic control based on vision. By prioritizing denser lanes and skipping empty ones,

the system shows a clear potential advantage over fixed-time controllers, especially in scenarios with fluctuating traffic patterns. The proportional green time allocation further attempts to balance demand. While the current density metric (area occupancy) and decision logic are relatively simple, they form a functional baseline for responsive signal timing.

5.4. Challenges Encountered

- **ROI Calibration:** Accurately defining the LANE_ROIS manually in utils.py proved to be crucial and requires careful measurement based on the specific camera view. Misaligned ROIs lead directly to incorrect density calculations.
- **Detection Variability:** YOLOv8n's performance can be affected by:
  - **Lighting:** Poor lighting (night, glare) reduces detection accuracy.
  - **Occlusion:** Vehicles hidden behind larger ones may not be detected.
  - **Weather:** Rain or fog can impair visibility.
  - **Vehicle Size/Distance:** Very small or distant vehicles might be missed.
- **Density Metric Limitations:** The area-based density doesn't distinguish between one large truck and several small cars occupying the same area. It also doesn't factor in vehicle speed or queue formation dynamics beyond the ROI.
- **Hardware Dependency:** Real-time performance is heavily reliant on the processing power (CPU/GPU) of the machine running the backend.
- **Threading Complexity:** Implementing the background processing thread in Flask required careful handling of shared data (using threading.Lock) and ensuring clean shutdown (threading.Event).

Chapter 6: Conclusion and Future Work

6.1. Conclusion

This project successfully developed and demonstrated a prototype system for smart traffic light control using real-time computer vision. By integrating the YOLOv8 object detection model with OpenCV for video processing, calculating lane-specific traffic density, and implementing an adaptive decision algorithm in Python, the system dynamically adjusts signal timings. A Flask-based backend API serves this real-time data to a responsive web dashboard built with HTML, CSS (Tailwind), and JavaScript. The prototype effectively prioritizes traffic flow based on measured density, showcasing its potential to alleviate congestion and reduce unnecessary waiting times compared to conventional fixed-time systems. Despite limitations in the current density metric and decision logic, the project establishes a viable framework for vision-based intelligent traffic management.

6.2. Future Enhancements

- **Live Video Streaming:** Integrate live, annotated video feed (showing ROIs and detections) into the web dashboard using technologies like MJPEG streaming or WebSockets for a more comprehensive monitoring view.
- **Advanced Density/Queue Metrics:** Implement more sophisticated methods like vehicle counting within ROIs, optical flow analysis, or tracking algorithms (e.g., DeepSORT) to estimate queue lengths or wait times more accurately.
- **Improved Decision Logic:** Develop a more complex algorithm considering factors like historical data, time-of-day patterns, pedestrian detection (requiring model retraining or a different model), adjacent intersection coordination, and potentially reinforcement learning approaches.
- **Model Optimization:** Experiment with larger YOLOv8 models (yolov8s, yolov8m) if hardware permits, or fine-tune the model on traffic-specific datasets to improve detection accuracy in challenging conditions.
- **Robustness & Error Handling:** Implement more robust error handling for camera disconnections, processing errors, and network issues. Add mechanisms for automatic recovery.
- **ROI Auto-Calibration:** Explore methods for semi-automatic or automatic ROI calibration based on lane line detection or user interaction.
- **Deployment:** Package the backend application using Docker for easier deployment and scalability. Utilize a production-grade WSGI server (e.g., Gunicorn, Waitress) instead of Flask's built-in development server.

7. References

(List any libraries, frameworks, papers, or significant online resources consulted)

- Ultralytics YOLOv8: https://github.com/ultralytics/ultralytics
- OpenCV Library: https://opencv.org/
- Flask Web Framework: https://flask.palletsprojects.com/
- Tailwind CSS: https://tailwindcss.com/
- NumPy: https://numpy.org/
- Research Paper: https://link.springer.com/article/10.1007/s44163-023-00087-z