

Content

1	Rezumat în limba română.....	3
1.1	Stadiul Actual.....	3
1.2	Fundamentare Teoretică.....	3
1.2.1	Raspberry Pi	4
1.2.2	Modul Semafor cu 3 LED-uri.....	4
1.2.3	Camere video USB și hub USB alimentat extern	4
1.2.4	Python.....	4
1.2.5	Modelul YOLOv11 pentru detecția mașinilor	5
1.2.6	Arbori Decizionali	5
1.3	Implementarea soluției adoptate	6
1.3.1	Detecția și numărarea mașinilor	6
1.3.2	Antrenarea modelului ML și implementarea cu modulele de semafor.....	7
1.3.3	Aplicația Web pentru monitorizare.....	7
1.4	Rezultate Experimentale	8
2	Work planning.....	11
3	State of the Art	12
3.1	Current context of urban traffic.....	12
3.2	Intelligent traffic light systems currently in use.....	12
3.3	Examples of implemented ATCS.....	12
3.4	Advantages and drawbacks of ATCS systems.....	13
3.5	Vehicle Detection Using YOLOv11 and the ncnn Framework	13
3.6	Decision Models for Traffic Light Control.....	13
3.7	Conclusion	14
4	Theoretical Fundamentals	15
4.1	Hardware Technologies Used	16
4.1.1	Raspberry Pi: Classification, Architectural details, and Role in the Project	16
4.1.2	LED Traffic Light Module: Specifications, Compatibility and Role in this Project	18
4.1.3	USB Video Cameras and Powered USB Hub	19
4.2	Software Technologies Used	21
4.2.1	Python.....	21
4.2.2	YOLOv11 Model for vehicle detection.....	22
4.2.3	Decision Trees	24
5	Implementation.....	27
5.1	Block Diagram.....	27
5.2	Raspberry Pi Configuration and VS Code Remote Development (SSH)	28

5.2.1	Raspberry Pi Configuration	28
5.2.2	VS Code Remote Development (SSH).....	29
5.3	Vehicle Detection and Counting using YOLOv11	30
5.3.1	Camera Configuration	30
5.3.2	Vehicle Detection and Counting using YOLOv11n in NCNN format	31
5.4	ML Model Training and Implementation with Traffic Light Module using GPIO	34
5.4.1	Training the ML Decision Tree Model.....	34
5.4.2	Connecting Traffic Light Modules to the Board	37
5.4.3	Implementing the Trained Model in the Project.....	38
5.5	Web Application (Flask Dashboard).....	41
5.5.1	Web Application Architecture	41
5.5.2	Backend Implementation (“web_server.py”)	41
5.5.3	Frontend Implementation (HTML/CSS/JavaScript).....	42
6	Experimental Results.....	44
6.1	Physical System Implementation	44
6.2	Evaluation of vehicle detection and counting using YOLOv11n	45
6.3	Evaluation of the Decision Tree Model	46
6.3.1	Decisional Tree Graphic Representation	46
6.3.2	Confusion Matrix.....	49
6.3.3	Cross-Validation Score (5-Fold) vs Training Accuracy	50
6.3.4	Feature Importance	50
6.4	System Functionality in Various Scenarios	51
6.4.1	Scenario 1: Equal Number of Cars on Both Axes (NS = EW).....	51
6.4.2	Scenario 2: Heavier traffic on East-West Axis (EW > SN).....	52
6.4.3	Scenario 3: Heavier traffic on South-North Axis (SN > EW)	53
6.4.4	Scenario 4: Manually set Emergency Lights (Blinking Yellow).....	54
6.4.5	Scenario 5: Camera Error	55
7	Conclusion.....	56
8	References	57
9	Appendix	60
10	Curriculum Vitae.....	82

1 Rezumat în limba română

1.1 Stadiul Actual

Urbanizarea rapidă și creșterea numărului de mașini au suprasolicitat intersecțiile semaforizate, deoarece semafoarele cu timpi fix nu reușesc să se adapteze traficului dinamic, provocând congestie și poluare. Rapoartele INRIX 2024 și TomTom 2023 arată că șoferii pierd anual până la 144 de ore în trafic [1][2]. Vehiculele aflate în staționare emit între 10 și 30 de grame de CO₂ pe minut, contribuind cu 47% la emisiile unei călătorii urbane [3][4].

Din cauza acestui volum mare de trafic, autoritățile și cercetătorii au început să dezvolte diferite sisteme de semaforizare inteligentă. Aceste sisteme se bazează pe senzori și diferiți algoritmi de procesare pentru a adapta durata semnalelor luminoase în timp real, în funcție de condițiile de trafic.

Aceste sisteme oferă multiple avantaje precum reducerea întârzierilor în trafic, scăderea emisiilor poluante și îmbunătățirea siguranței rutiere. Totuși, există și anumite provocări care includ costuri ridicate de implementare, dependența de senzori care necesită întreținere frecventă și complexitate operațională [10].

1.2 Fundamentare Teoretică

Sistemele de control al traficului sunt esențiale pentru infrastructura urbană modernă, ajutând la fluidizarea traficului, creșterea siguranței rutiere și reducerea emisiilor. Semaforizarea clasică, bazată pe cicluri fixe este în general inefficientă în fața variațiilor din trafic. Ca răspuns, au fost dezvoltate sisteme adaptive (ATCS) care ajustează timpii semafoarelor în timp real în funcție de condițiile din trafic, folosind senzori și algoritmi, pentru a optimiza fluxurile de trafic și a reduce poluarea.

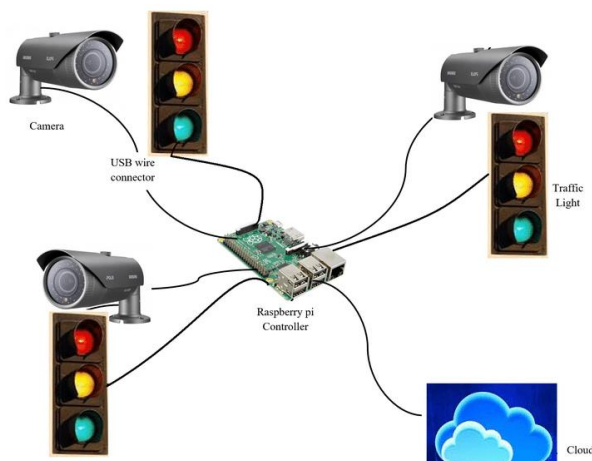


Figura 1. Sistem adaptiv de control al traficului (ATCS) [18]

Această lucrare prezintă un astfel de sistem (Figura 1.) implementat la scară redusă, utilizând algoritmul YOLOv11 pentru detecția mașinilor și un arbore decizional care determină timpii necesari a culorii verde la semafoare, toate acestea fiind implementate pe un Raspberry Pi 4. Portabilitatea și independența sistemului îl fac o soluție practică și flexibilă pentru a gestiona corect traficul.

1.2.1 Raspberry Pi

Raspberry Pi este o serie de microcomputere dezvoltată de Fundația Raspberry Pi. Lansat inițial în 2012, seria a evoluat de-a lungul timpului oferind diferite modele cu performanțe și funcționalități îmbunătățite, devenind una dintre cele mai populare platforme pentru proiecte din domeniul roboticii, automatizării sau IoT. Modelele se împart în general în două tipuri: Model B care este cel mai puternic și care dispune de un port Ethernet și Model A care este mai accesibil, cu mai puține porturi și o memorie RAM mai mică.

Pentru acest proiect, a fost ales modelul Raspberry Pi 4 Model B de 4 GB RAM. Acesta are rolul de nucleu hardware al sistemului adaptiv de control al traficului, integrând atât funcționalități de detectare a vehiculelor, cât și de control al semaforului.

Pinii GPIO (General Purpose Input/Output) ai Raspberry Pi reprezintă o componentă cheie, permițând plăcii să interacționeze cu componente electrice externe. Acești pini sunt versatili și pot fi programați pentru a transmite sau primi semnale digitale. În cadrul acestui sistem, pinii GPIO sunt esențiali pentru controlul direct al luminilor roșii, galbene și verzi ale modulelor semaforului.

1.2.2 Modul Semafor cu 3 LED-uri

Pentru a simula sistemul de semaforizare au fost folosite patru module de semafor, fiecare cu câte 3 LED-uri. Aceste module sunt ușor de folosit având patru pini: câte unul pentru fiecare culoare și un pin pentru împământare. Conectarea acestora la sistem se face prin pinii GPIO a Raspberry Pi, fiind ușor de controlat prin comenzi software. În plus, un alt avantaj îl constituie dimensiunea redusă a acestora, fiind ideale pentru prototipuri la scară redusă.

În cadrul proiectului au un rol crucial, fiind folosite ca interfață vizuală pentru sistemul adaptiv de control al traficului. Prin afișarea precisă a luminilor semafoarelor, modulul contribuie direct la gestionarea adaptivă a fluxului de trafic.

1.2.3 Camere video USB și hub USB alimentat extern

Camerele video sunt componente esențiale în cadrul proiectului, acestea fiind folosite pentru a monitoriza în timp real traficul. În cadrul proiectului, au fost folosite 4 camere (câte una pentru fiecare direcție din intersecție) cu ajutorul cărora sunt furnizate date esențiale pentru modelul YOLO, asigurând o detecție și numărare precisă și exactă a vehiculelor din intersecție.

Raspberry Pi-ul are o limitare fizică importantă în ceea ce privește curentul furnizat de porturile USB (aproximativ 1.2 A). Deoarece fiecare cameră consumă circa 400 mA [23], folosirea a patru camere necesită 1.6 A, depășind capacitatea plăcii. Această situație duce la instabilitatea sistemului și la reporniri frecvente. Pentru a evita această problemă și a avea o funcționalitate stabilă, s-a utilizat un hub USB cu alimentare externă.

1.2.4 Python

Ca limbaj principal de programare pentru proiectul acesta a fost ales Python datorită versatilității sale, fiind de altfel limbajul standard folosit pentru Raspberry Pi. Dezvoltat inițial de Guido van Rossum în 1991, Python este un limbaj orientat pe obiecte, cunoscut pentru simplitatea sintaxei. Este frecvent utilizat în domeniul inteligenței artificiale, analiză de date și sisteme embedded, fiind potrivit pentru dispozitive cu resurse limitate. În plus, acesta este gratuit, open-source și sprijină constant dezvoltarea teoretică [25]. Principalele librării folosite în cadrul proiectului sunt: ultralytics, cv2 (OpenCV), RPi.GPIO, Flask.

În contextul sistemului, Python este folosit ca o legătură între analiza imaginilor și controlul semnalelor luminoase, permițând detectarea traficului și calculul timpilor de semaforizare.

1.2.5 Modelul YOLOv11 pentru detecția mașinilor

YOLO este o familie de algoritmi care detectează obiecte într-o singură trecere a imaginii, asigurând viteză și acuratețe. Lansat în 2014 de Ultralytics, folosește o rețea neuronală convoluțională optimizată și împarte imaginea în grilă pentru a prezice simultan clasele și coordonatele obiectelor. Antrenat pe seturi mari de date (ex. COCO), acest algoritm este ideal pentru recunoașterea vehiculelor în timp real [27].

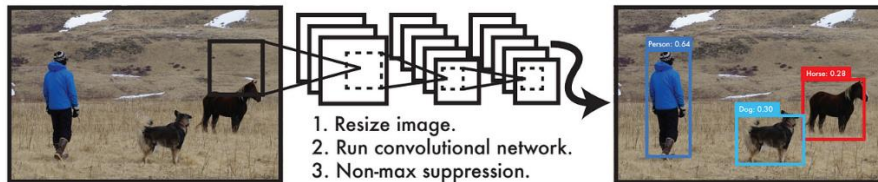


Figura 2. Fluxul de procesare al modelului YOLO [11]

YOLO funcționează prin redimensionarea imaginii la o dimensiune standard, care este apoi procesată printr-o rețea neuronală într-o singură trecere. Aceasta împarte imaginea într-o grilă, iar fiecare celulă din grilă prezice simultan zonele în care se află obiecte, alături de scoruri de încredere. Ulterior, predicțiile cu scoruri sub un anumit prag sunt eliminate pentru a reduce zgomotul, iar prin aplicarea tehnicii Non-Maximum Suppression se elimină suprapunerile, fiind păstrate cele mai precise detecții [11].

1.2.6 Arbori Decizionali

Arborii Decizionali sunt un model de învățare automată folosit, în cadrul proiectului, pentru estimarea duratelor optime ale fazelor semaforului, bazat pe analiza traficului. Alegerea lor se datorează interpretabilității ridicate și cerințelor reduse de resurse, ideale pentru sisteme cu performanța limitată. Modelul se bazează pe structurarea procesului decizional printr-o ierarhie de noduri și ramuri, unde fiecare nod reprezintă o condiție bazată pe date de intrare (ex. densitatea traficului). Bazat pe principii matematice, arborele poate fi folosit atât pentru regresie, cât și pentru clasificare. [36]

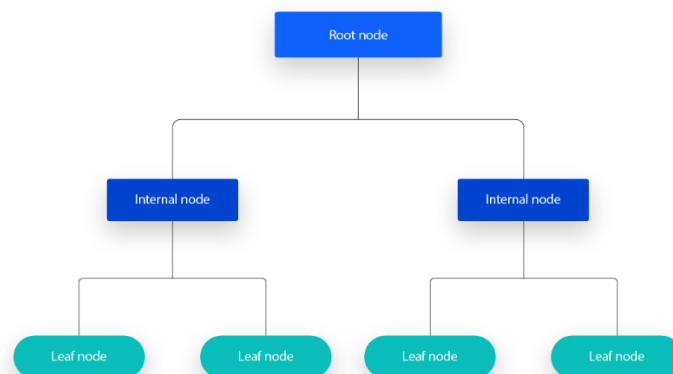


Figura 3. Structura unui Arbore Decizional [37]

Există mai mulți algoritmi cunoscuți pentru construirea arborilor decizionali, fiecare cu particularități diferite: ID3, C4.5, CART. Biblioteca Scikit-learn, folosită pentru acest proiect, utilizează o versiune optimizată a algoritmului CART, care construiește arbori binari și poate fi folosit atât pentru regresie, cât și pentru clasificare.

1.3 Implementarea soluției adoptate

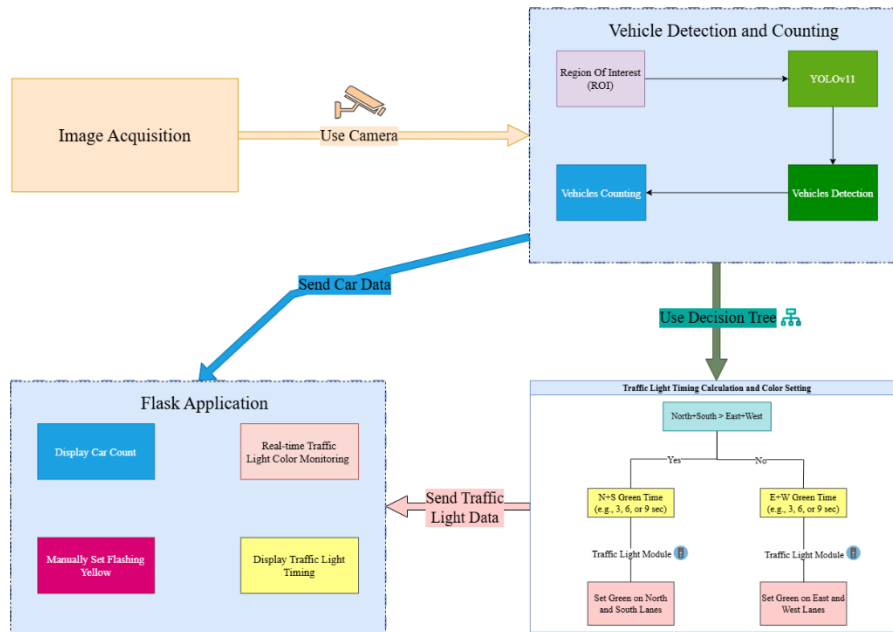


Figura 4. Diagrama Bloc a Sistemului de Trafic Inteligent

Diagrama bloc de mai sus prezintă arhitectura sistemului implementat, care integrează achiziția de imagini, detecția și numărarea mașinilor, decizia privind temporizarea semafoarelor și o aplicație web pentru a monitoriza starea sistemului.

Implementarea sistemului poate fi împărțită în trei pași principali: detecția și numărarea mașinilor, antrenarea modelului ML și implementarea acestuia împreună cu modulele de semafor și dezvoltarea aplicației web.

1.3.1 Detecția și numărarea mașinilor

Primul pas necesar pentru detecția și numărarea mașinilor a fost configurarea camerelor. Acest lucru a presupus identificarea și utilizarea unor căi USB persistente pentru fiecare cameră conectată la Raspberry Pi, fiecare fiind asociată unei direcții de trafic specifice. Aceste asocieri au fost esențiale pentru a corela corect fluxurile video cu benzile de circulație. Ulterior, căile au fost integrate în codul de detecție, astfel încât modelul YOLOv11 să proceseze fluxul video corespunzător fiecărei direcții.

Camerele au fost conectate prin porturile USB ale Raspberry Pi. O provocare importantă a fost lățimea de bandă limitată a porturilor USB, care a afectat performanța în cazul folosirii simultane a celor patru camere. Pentru a rezolva această problemă, camerele au fost folosite secvențial, la un interval de comutare de trei secunde.

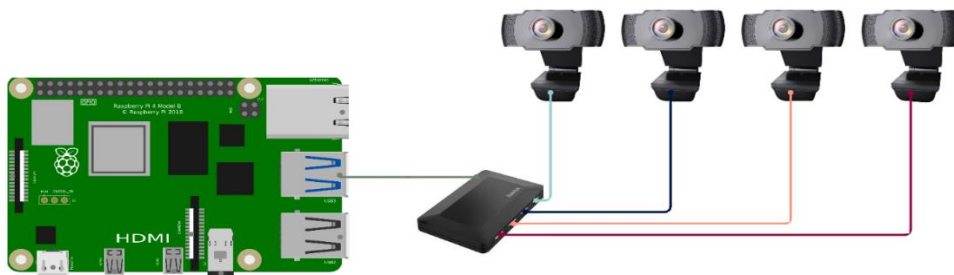


Figura 5. Conexiunea camerelor cu Raspberry Pi

1.3.2 Antrenarea modelului ML și implementarea cu modulele de semafor

Arborele de decizie a fost antrenat folosind calculatorul personal pentru a depăși limitările plăcii Raspberry Pi. Un set de date sintetic, care simulează numărul de vehicule pe fiecare direcție, a fost generat în Python. Acest set de date a fost folosit pentru a antrena modelul. Au fost folosite valori numerice pentru a simplifica modelul și a asigura consistența datelor de intrare.

Modulele de semafor au fost conectate la Raspberry Pi cu ajutorul pinilor GPIO, fiecare modul fiind atribuit pentru o anumită direcție (Nord, Sud, Est sau Vest). Fiecare modul are patru pini: câte unul pentru fiecare culoare a semaforului și un pin pentru împământare, iar toate conexiunile de masă au împărțit o cale comună către pinul 6 de pe Raspberry.

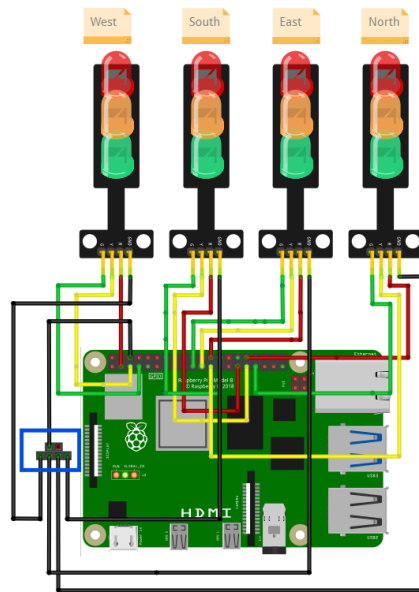


Figura 6. Conexiunea modulelor la Raspberry Pi

În final, modelul antrenat de arbore decizional a fost încărcat pe Raspberry Pi utilizând fișierul .plk și implementat în sistemul adaptiv pentru a controla stările pinilor GPIO astfel încât să gestioneze luminile semafoarelor conform predicțiilor modelului.

1.3.3 Aplicația Web pentru monitorizare

Aplicația web, dezvoltată folosind Flask, funcționează ca o interfață de monitorizare și control în timp real pentru sistemul adaptiv de semaforizare. Aceasta oferă multiple funcționalități precum: oferă vizualizare live a stărilor semafoarelor, afișează date importante despre sistem cum ar fi numărul de vehicule pe fiecare direcție și timpul necesar pentru culoarea verde pe direcțiile prioritare, posibile erori ale camerelor indicând care camera are probleme, un buton care trece semafoarele pe modul „Galben Intermitent” în cazul unor situații de urgență.



Figura 7. Aplicația Web pentru monitorizarea sistemului

1.4 Rezultate Experimentale

Sistemul a fost implementat fizic pe o placă de lemn, reprezentând o intersecție cu patru direcții, fiecare având o singură bandă pe sens. Camerele și modulele LED au fost montate cu suporturi printate 3D pentru a asigura detecția precisă a vehiculelor, în timp ce Raspberry Pi 4 a fost plasată sub placă.

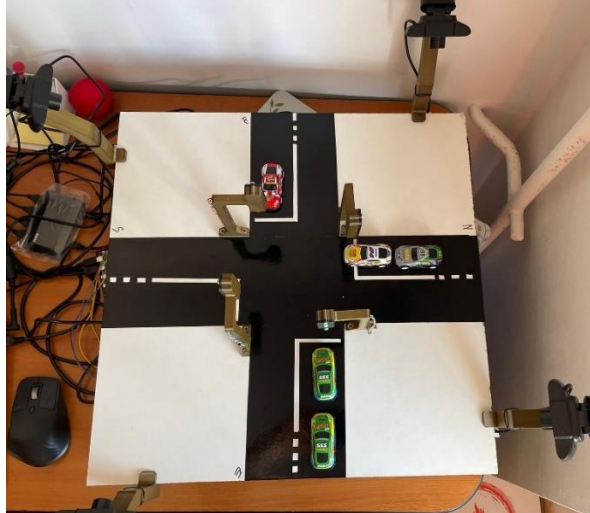


Figura 8. Implementarea Fizică a Sistemului

Modelul YOLOv11n a detectat și numărat cu succes vehicule aflate în cadrul unor Regiuni de Interes (ROI) definite.

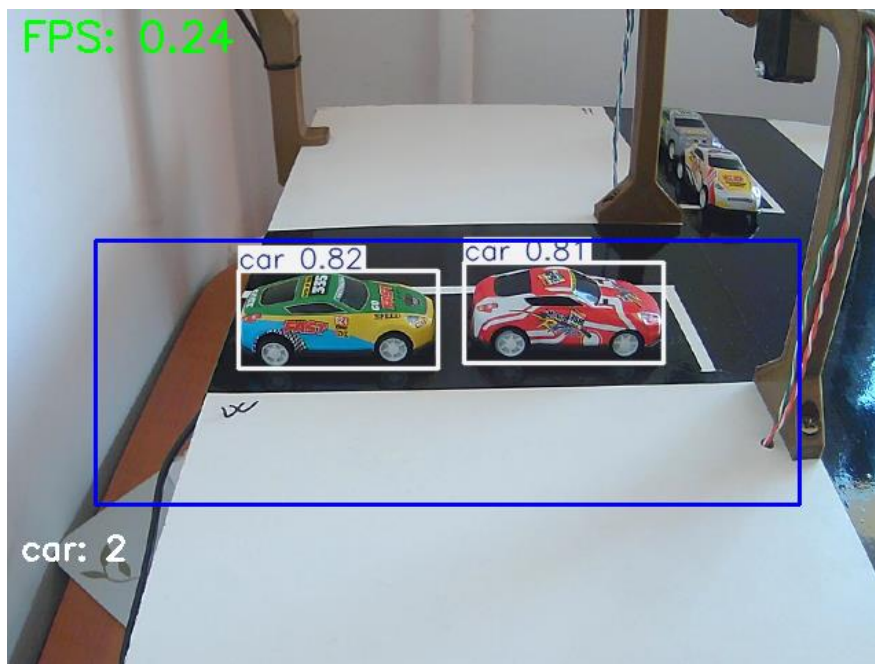


Figura 9. Detecția a doua mașini din intersecție

Performanța modelului ML de arbore decizional a fost evaluat unor diferite modalități. O matrice de confuzie a expus o acuratețe ridicată, cu clasificări greșite minime și fără confuzie între timpii „Short” și „Long”. Validarea încrucișată a arătat o acuratețe medie de 89%, cu rezultate consistente indicând lipsa efectului de „overfitting”. În plus, importanța caracteristicilor a evidențiat direcția Nord-Sud ca fiind un factor principal în luarea deciziilor a modelului, datorită frecvenței ridicate în setul de date de antrenament.

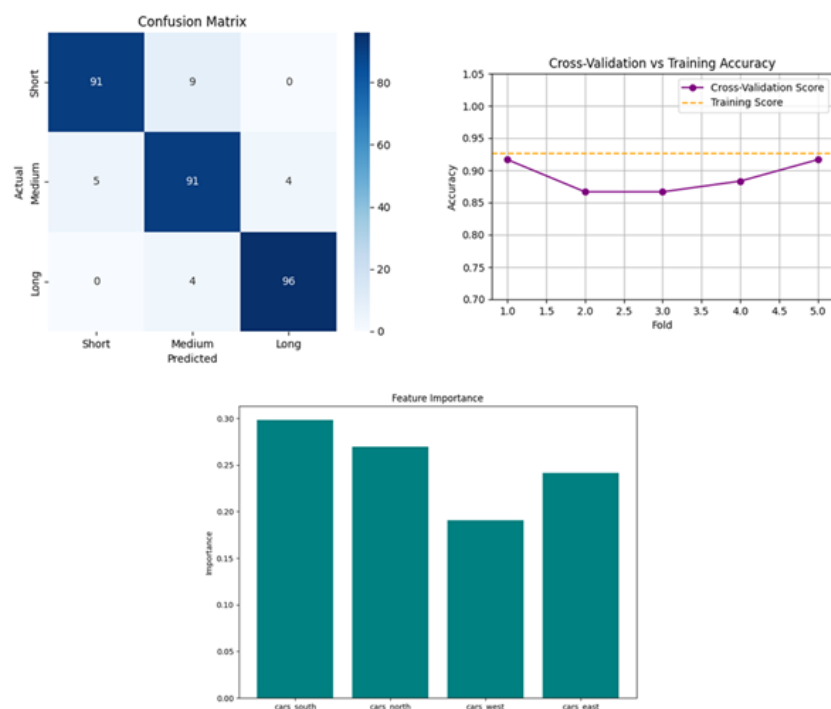


Figura 10. Principalele instrumente de analiză a modelului ML

În final, sistemul a fost testat în diferite condiții, demonstrând adaptabilitatea și funcționarea corectă a acestuia. Cu un număr egal de mașini atât pe direcția Nord-Sud (NS), cât și pe Est-Vest (EV) (trei mașini), a fost atribuit o durată medie de 6 secunde pentru NS demonstrând prioritatea dată de setul de date de intrare (Scenariu 1). În situația în care traficul e mai intens pe EV (patru mașini) comparativ cu NS (două) a declanșat o durată lungă de 9 secunde (Scenariu 2). Dacă pe direcția NS se află mai multe mașini decât pe EV (o mașină, respectiv niciuna) a fost calculat un timp scurt de 3 secunde (Scenariu 3).

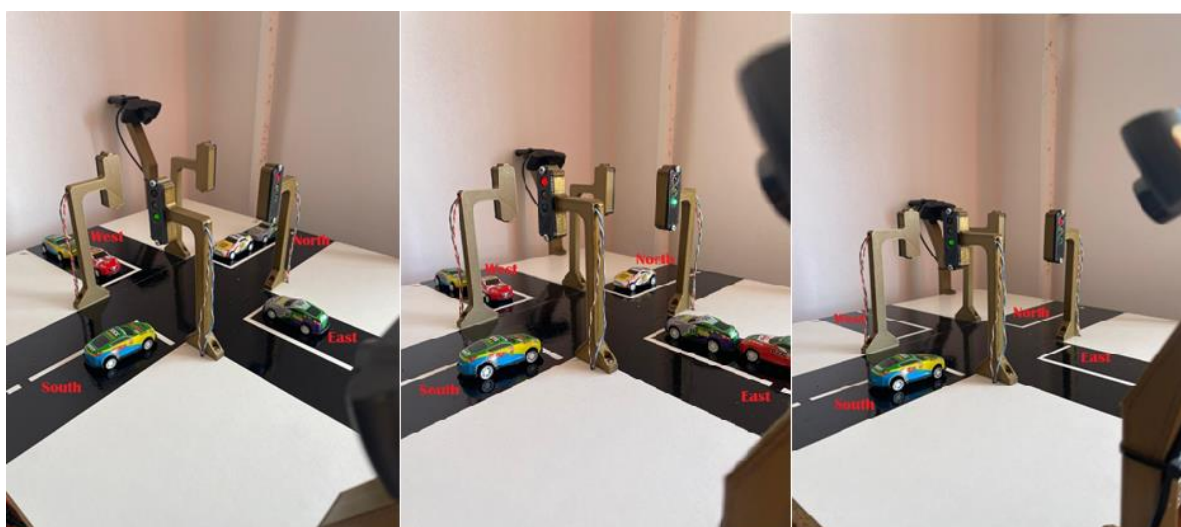


Figura 11. Testarea Sistemului (de la stânga la dreapta: Scenariu 1 - Scenariu 3)

În plus au fost testate și sistemele de siguranță reprezentate prin activarea manuală a luminilor în modul „galben intermitent” și comutarea automată a acestui mod în cazul unei defecțiuni a camerelor.



Figura 12. Funcționalitatea sistemelor de siguranță în aplicația web

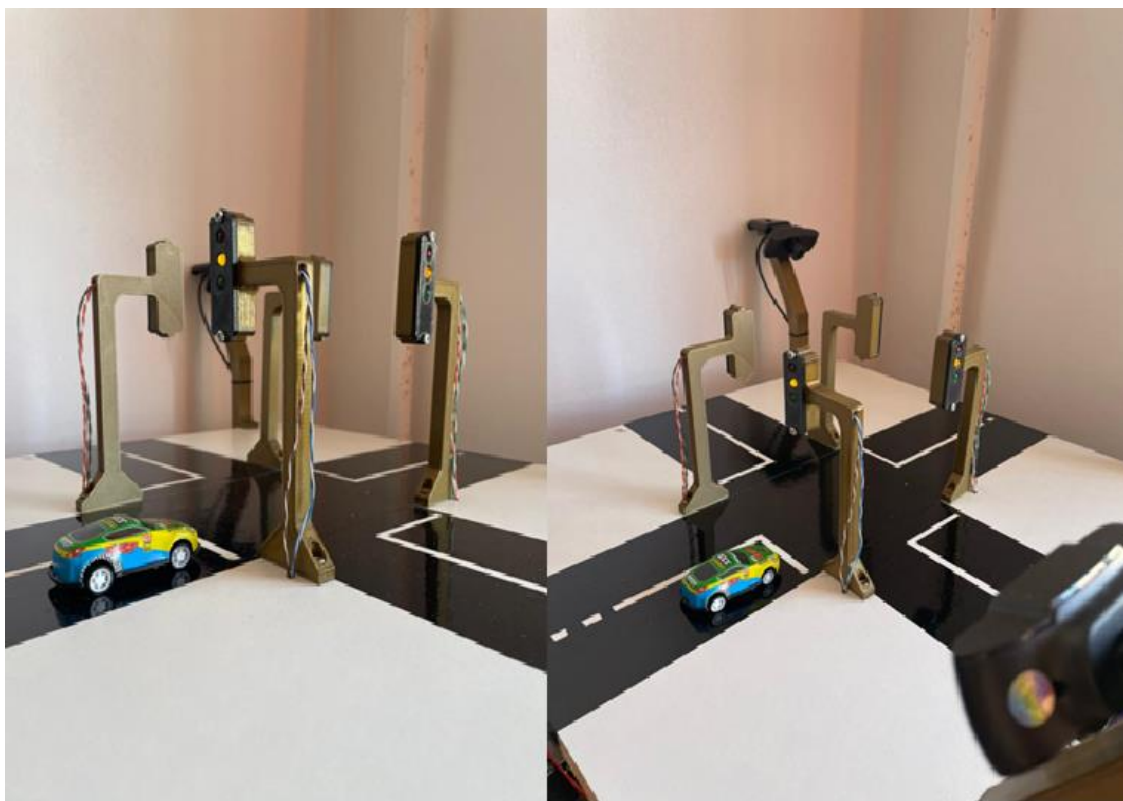


Figura 13. Testarea sistemelor de siguranță în modelul fizic al sistemului

2 Work planning

Task No.	Task Description	Duration	Start Date	End Date
1	Choosing the Bachelor's Thesis Topic	5 days	03.10.2024	07.10.2024
2	Theoretical Research on the Subject	28 days	08.10.2024	04.11.2024
3	Analysis of Necessary Hardware Components	14 days	05.11.2024	18.11.2024
4	Detection Algorithm Selection	14 days	19.11.2024	02.12.2024
5	Choosing the Machine Learning Algorithm	14 days	03.12.2024	16.12.2024
6	Building the Physical Scaled-Intersection	14 days	17.12.2024	30.12.2024
7	Connecting Cameras to Raspberry Pi	6 days	03.01.2025	08.01.2025
8	Testing the YOLOv11n Model for Detection	21 days	09.01.2025	29.01.2025
9	Training the Decision Tree	21 days	30.01.2025	19.02.2025
10	Verified ML Model Performance	7 days	20.02.2025	26.02.2025
11	Connecting and Testing Traffic Light Modules	14 days	27.02.2025	12.03.2025
12	Deploy and Implement the ML Model into the System	21 days	13.03.2025	02.04.2025
13	Creating the Web Application	21 days	03.04.2025	23.04.2025
14	Testing the Functionality of the Entire System	14 days	24.04.2025	07.05.2025
15	Adjustments and refinements	14 days	08.05.2025	21.05.2025
16	Documentation Writing	40 days	22.05.2025	30.06.2025

3 State of the Art

3.1 Current context of urban traffic

Due to rapid urbanization and the constant increase in vehicle numbers in recent years, urban road infrastructure, especially signalized intersections, has become significantly overburdened. Conventional fixed-time traffic light systems are not capable of adapting to the dynamic nature of 21st century traffic demands. This results in traffic congestion, extended travel-times, and increased pollution levels in major urban areas.

According to the INRIX Global Traffic Scorecard 2024, drivers in congested cities lose between 50 and 100 hours annually in traffic, with most of this time spent at signalized intersections [1]. In Romania, TomTom Traffic Index 2023 indicates that drivers in Bucharest lose an average of over 144 hours per year due to congested traffic [2].

Beyond the time lost, the environmental impact is significant. Recent studies show that a vehicle idling produces between 10 and 30 grams of carbon dioxide per minute, depending on the engine type [3], those values decrease when the vehicles are at constant speed. Emissions during idling account for approximately 47% of the total emissions associated with a typical urban journey [4].

For these reasons, research and development of intelligent traffic control systems have begun, aiming to reduce waiting times and the environmental impact of road transport.

3.2 Intelligent traffic light systems currently in use

Given the increasing volume of urban traffic, authorities and transportation researchers have begun implementing intelligent traffic light systems (ATCS – Adaptive traffic control systems) capable of dynamically managing traffic flow. Unlike traditional fixed-time traffic light systems, modern ATCS relies on detection sensors (radars, video cameras), processing algorithms and inter-system communication to adapt the duration of light signals in real time based on traffic conditions.

3.3 Examples of implemented ATCS

SCOOT (Split Cycle Offset Optimization Technique) is an adaptive traffic light control system whose development has begun in the United Kingdom in 1979 and continues to evolve, with the last version updated by the Transport Research Laboratory (TLR) in 2021[5]. It is primarily used in the UK, but also in several other countries [5]. SCOOT automatically adjusts the duration of green, yellow and red signals based on real-time data received from sensors [6].

Surtrac (Scalable Urban Traffic Control) is a system developed in the United States that uses artificial intelligence algorithms to autonomously and cooperatively manage traffic lights at intersections [7]. It was first tested in 2012 in Pittsburgh, Pennsylvania, where it reduced traffic delays by 40% [8]. A successful follow-up test occurred in October 2013, and starting from 2015, the technology began to be commercialized.

SCATS (Sydney Coordinated Adaptive Traffic System) is another ATCS system, initially developed in Australia and later expanded globally. SCATS enables dynamic coordination of groups of intersections to optimize traffic flow. The system is used multiple cities such as Singapore, Tehran, Shanghai and Dublin [9].

3.4 Advantages and drawbacks of ATCS systems

ATCS systems offer a high-performance alternative to conventional traffic light systems, but they also pose certain technical and financial challenges [10].

Advantages:

- Reduction in traffic delays
- Decrease in pollutant emissions
- Improve road safety
- Scalability
- Potential for integration with other smart mobility technologies

Drawbacks:

- High implementation costs
- Dependence on invasive physical sensors which require frequent maintenance
- Operational complexity

3.5 Vehicle Detection Using YOLOv11 and the ncnn Framework

In modern adaptive traffic light systems, accurate vehicle detection forms the foundation for dynamically adjusting traffic signal phases. Traditional detection methods (inductive loops, radars or infrared sensors) involve high installation and maintenance costs and offer limited flexibility in recognizing positions in complex scenarios. In this context, computer vision, supported by artificial intelligence algorithms, has emerged as an efficient and scalable alternative.

One of the most effective and modern methods is YOLO (You Only Look Once) algorithm, initially introduced by Joseph Redmon in 2016 [11]. Since its first version, YOLO has received numerous improvements in terms of accuracy, speed and size. The latest version, YOLOv11, supports model export and execution on edge platforms (e.g., Raspberry Pi) while maintaining very good performance.

The YOLOv11 model used in this thesis is an optimized version compatible with the ncnn framework, a library developed by Tencent for better performance on ARM and x86 architecture [12]. Using this framework enables real-time execution of the YOLO model resource-constrained hardware, making it suitable for embedded applications. According to Mujadded Al Rabbani Alif, from Huddersfield University, the accuracy and speed of YOLOv11 models has increased significantly compared to its predecessors (YOLOv8 and YOLOv10) with a precision of about 80%, demonstrating their viability for adaptive traffic light control systems [13].

3.6 Decision Models for Traffic Light Control

Following vehicle detection in each lane, a decision must be made regarding the duration of the green light phase. The literature highlights several paradigms for automated decision making:

- Decision Trees: Symbolic models based on logical rules, for instance separation, easily interpretably and compatible with embedded environments [14]
- Fuzzy Logic: Suitable for uncertain contexts where decisions cannot be made solely on numerical thresholds but use linguistic terms [15]
- Neural Networks and Reinforcement Learning Algorithms: Employed in simulated environments or for multi-objective control in complex systems [16]

The approach proposed in this study falls within the category of interpretable symbolic systems, implemented through a decision tree trained on a manually labeled dataset that classifies intersection situations based on vehicle distribution across lanes. This method is justified by its practical advantages:

- Very short training time,
- Interpretable and transparent decisions,
- Small model size, compatible with embedded system architectures.

According to Breiman (1984), decision trees provide an effective solution in contexts where a balance between prediction accuracy and decision interpretability is required [14]. Unlike “black box” models (e.g., convolutional neural networks), decision trees generate logical structures that are easy to analyze, visualize and understand by human operators. This characteristic is crucial in sensitive domains such as traffic control, where justifying algorithmic decisions from a logical perspective is often necessary.

Recent literature highlights the utility of decision trees in adaptive traffic control systems (ATCS), particularly in applications with limited infrastructure and when you need a small size model [17]. In such situations, where hardware resources are constrained and decision traceability is a key criterion, symbolic models prove to be the best choice. Additionally, those models can be trained on small, manually labeled datasets, making them ideal for experimental projects.

3.7 Conclusion

The analysis of specialized literature highlights a clear trend toward adopting machine learning algorithms and artificial intelligence models in adaptive traffic control systems (ATCS).

Each technology used in ATCS presents advantages and limitations, influenced by factors such as accuracy, hardware requirements and implementation costs. As previously mentioned, systems based on physical sensors, such as inductive loops or radars, offer good accuracy but involve invasive installation and high costs. In contrast, computer vision-based solutions are more flexible and easier to install but can be affected by lighting or weather conditions.

Regarding decision-making for traffic light phase durations, symbolic models like decision trees enable interpretable control. Current research trends focus on integrating symbolic algorithms with embedded visual detection systems, which can operate efficiently in intersections with limited infrastructures.

The solution proposed in this thesis combines symbolic model (decision tree) with visual detection system (YOLOv11), running on an embedded platform in a scaled intersection. This approach meets the requirements for interpretability and flexibility, making it suitable in the current context of ATCS development.

4 Theoretical Fundamentals

Traffic control systems play a crucial role in modern urban infrastructure, helping to ease up the traffic flows, increase road safety, and reduce pollutant emissions. These systems manage the interactions between different traffic entities-vehicles, pedestrians- through strict traffic rules or signaling. Among the key components of these systems are traffic lights, which ensure the controlled alternation of traffic flows and help prevent collisions at intersections. Conventional traffic lights, operating on fixed cycles, follow predefined sequences regardless of the actual traffic volume or unexpected variations on the road. In today's bustling urban environments, this rigid approach frequently leads to several significant drawbacks. For instance, it often results in traffic jams, long waiting times for commuters, and excessive fuel consumption. These inefficiencies highlight a critical need for more responsive solutions.

Consequently, over the past few decades, there has been a significant shift in focus towards developing intelligent and adaptive traffic light systems, commonly referred to as ATCS (Adaptive Traffic Control Systems). These advanced systems aim to overcome the limitations of their traditional counterparts by dynamically adjusting the times of the traffic lights to real-time traffic conditions. They utilize sensors, video cameras, communication networks, and processing algorithms to interpret different situations and react quickly and efficiently. This approach leads to shorter waiting times, reduces pollution, and an overall increase in the efficiency of the road infrastructure.

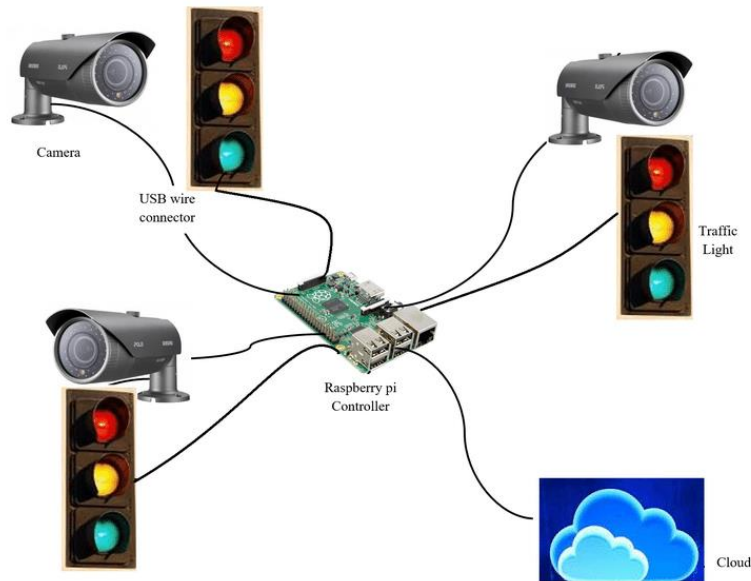


Figure 14. Adaptive Traffic Control System [18]

This paper proposed the development and testing of an adaptive traffic control system (Figure. 1 [18]), implemented at a reduced scale intersection. This system utilizes the YOLOv11 algorithm for real-time vehicle detection. The crucial decision regarding the duration of the green light signal for each approach is made through a machine learning model- specifically, a decision tree. All processing occurs directly on an embedded Raspberry Pi 4 device, eliminating the reliance on external computing resources. This design provides the system with significant portability and independence. This self-sufficiency makes it an ideal solution for deployment in diverse locations, offering a robust and flexible approach to modern traffic management.

4.1 Hardware Technologies Used

4.1.1 Raspberry Pi: Classification, Architectural details, and Role in the Project

The Raspberry Pi is a family of microcomputers developed by the Raspberry Pi Foundation, a UK-based educational organization. It was initially launched in 2012 with the goal of promoting computer science education. Over time, the series has evolved, offering different models with improved performance and functionalities, becoming one of the most popular platforms for projects in robotics, automation, and Internet of Things (IoT).

Table 1. Raspberry Pi Model B Evolution [19]

Model	SoC	Memory	GPIO	Connectivity
Raspberry Pi Model B	BCM2835	256MB, 512MB	26-pin GPIO header	HDMI, 2 × USB 2.0, CSI camera port, DSI display port, 3.5mm audio jack, RCA composite video, Ethernet (100Mb/s)
Raspberry Pi Model B+	BCM2835	512MB	40-pin GPIO header	HDMI, 4 × USB 2.0, CSI camera port, DSI display port, 3.5mm AV jack, Ethernet (100Mb/s)
Raspberry Pi 2 Model B	BCM2836/ BCM2837	1 GB	40-pin GPIO header	HDMI, 4 × USB 2.0, CSI camera port, DSI display port, 3.5mm AV jack, Ethernet (100Mb/s)
Raspberry Pi 3 Model B	BCM2837	1 GB	40-pin GPIO header	HDMI, 4 × USB 2.0, CSI camera port, DSI display port, 3.5mm AV jack, Ethernet (100Mb/s), 2.4GHz single-band 802.11n Wi-Fi (35Mb/s), Bluetooth 4.1
Raspberry Pi 3 Model B+	BCM2837b0	1 GB	40-pin GPIO header	HDMI, 4 × USB 2.0, CSI camera port, DSI display port, 3.5mm AV jack, PoE-capable Ethernet (300Mb/s), 2.4/5GHz dual-band 802.11ac Wi-Fi (100Mb/s), Bluetooth 4.2
Raspberry Pi 4 Model B	BCM2711	1GB, 2GB, 4GB, 8GB	40-pin GPIO header	2 × micro-HDMI, 2 × USB 2.0, 2 × USB 3.0, CSI camera port, DSI display port, 3.5mm AV jack, PoE-capable Gigabit Ethernet (1Gb/s), 2.4/5GHz dual-band 802.11ac Wi-Fi (120Mb/s), Bluetooth 5
Raspberry Pi 5	BCM2712D0	2GB, 4GB, 8GB	40-pin GPIO header	2 × micro HDMI, 2 × USB 2.0, 2 × USB 3.0, 2 × CSI camera/DSI display ports, single-lane PCIe FFC connector , UART connector , RTC battery connector, four-pin JST-SH PWM fan connector , PoE+-capable Gigabit Ethernet (1Gb/s), 2.4/5GHz dual-band 802.11ac Wi-Fi 5 (300Mb/s), Bluetooth 5

Since its debut, the Raspberry Pi has gone through several generations, with each one bringing significant improvements. Table 1. above, presents an evolution of the Model B series, which is more powerful and has an Ethernet port, in contrast to Model A which is most affordable, has fewer USB ports, and less RAM.

For this project, a Raspberry Pi 4 Model B, configured with 4 GB of RAM, was chosen. This particular model serves as the hardware core of the adaptive traffic control system, integrating both traffic detection and traffic light control functionalities. It stands out due to its superior performance compared to older models. Furthermore, the inclusion of USB 3.0 ports is a significant advantage, especially considering the limited bandwidth of older models and the system's reliance on multiple USB cameras [20].

Its design is built upon a System-on-Chip (SoC) architecture. This compact design seamlessly integrates all the fundamental components of a conventional computer. Utilizing an ARM architecture, this system is finely tuned to achieve an optimal balance between processing power and efficient resource utilization. The Raspberry Pi 4 is particularly notable for its flexibility, ease of portability, and its robust capability to execute Linux operating systems specifically compiled for ARM platforms.

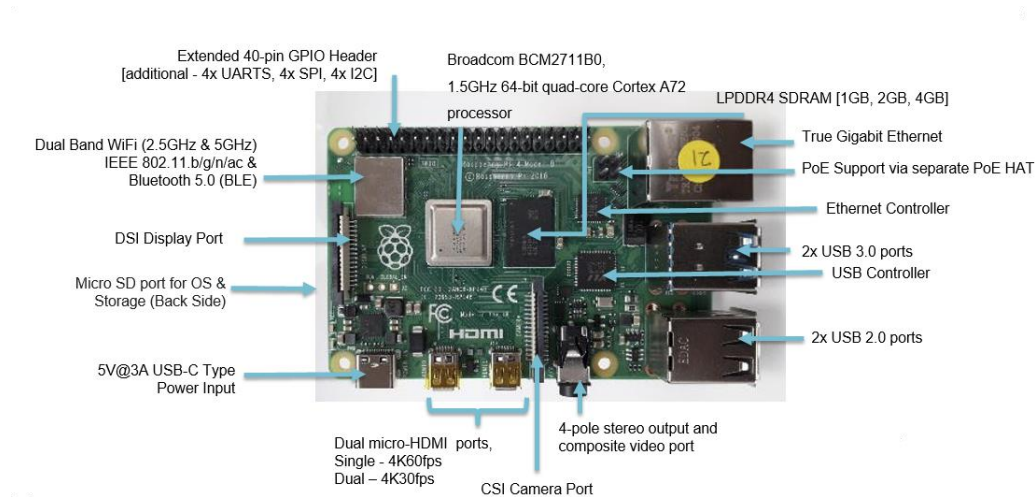


Figure 15. Raspberry Pi 4 Components [21]

The specifications of the core components (Figure 2. [21]) [21]:

- Broadcom BCM2711B0, 1.5GHz 64-bit quad-core Cortex-A72 processor: This is the central processing unit, responsible for all computational tasks.
- LPDDR4 SDRAM 4GB: The main memory (RAM) used for temporary data storage and running applications.
- True Gigabit Ethernet: Provides high-speed wired network connectivity.
- Ethernet Controller: Manages communication over the Ethernet port.
- 2x USB 3.0 ports: High-speed USB ports for connecting peripherals, including the cameras in your project.
- 2x USB 2.0 ports: Standard USB ports for connecting various devices.
- USB Controller: Manages data transfer through the USB ports.
- Dual micro-HDMI ports, Single - 4K60fps, Dual - 4K30fps: Two small HDMI ports allowing for connection to one or two displays, supporting 4K resolution at different frame rates.
- Extended 40-pin GPIO Header [additional - 4x UARTS, 4x SPI, 4x I2C]: A set of general-purpose input/output pins that allow the Raspberry Pi to interface with other electronic components and modules, including digital interfaces like UART, SPI, and I2C.

A key component of the Raspberry Pi is its GPIO (General Purpose Input/Output) pins (Figure 3. [20]), which serve as an essential interface between the microcomputer and the external world. These pins are versatile and can be programmatically configured to either receive or send digital signals, allowing the Raspberry Pi to interact with its physical environment and provide direct control over a variety of electronic components.

GPIO pins are essential when it comes to automation projects and embedded systems. By enabling connectivity with sensors, actuators, LEDs, and other modules, the Raspberry Pi is transformed from just a tiny computer into a powerful physical computing platform. For this adaptive traffic control system, the GPIO pins are crucial for controlling the red, yellow and green lights of the traffic signal modules.

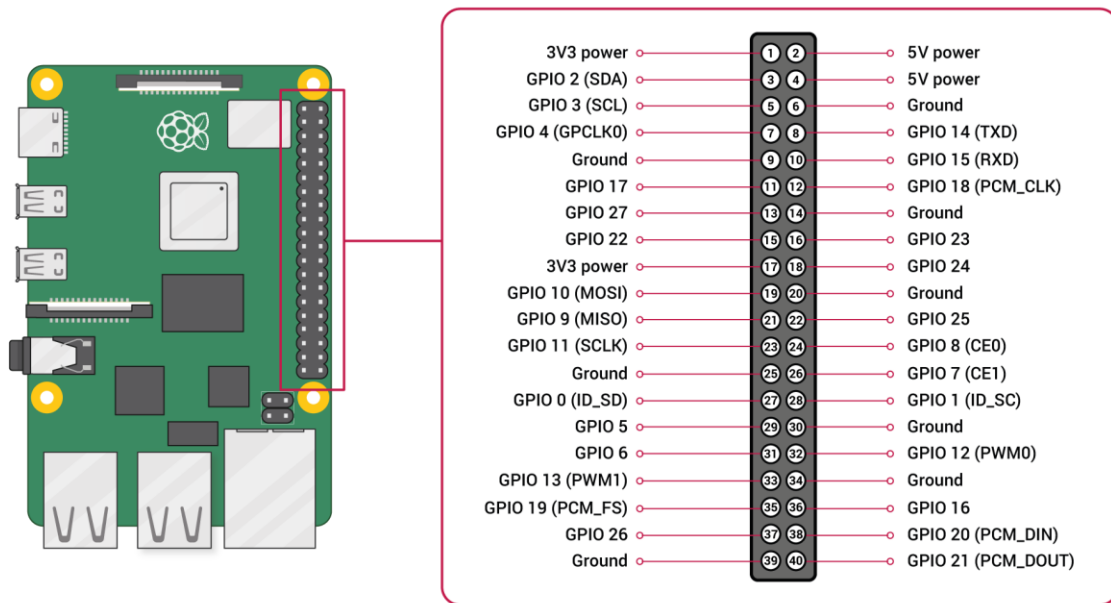


Figure 16. GPIO Pinout for Raspberry Pi 4 [20]

In this project, the Raspberry Pi serves as the central processing unit (CPU), autonomously managing all traffic control operations. It's responsible for receiving video streams from the USB cameras, applying YOLO detection to identify vehicles and count them, and then sending data to a decision-making model. Based on this analysis, the Raspberry Pi controls the traffic light LEDs by emitting digital signals via its GPIO pins. Crucially, all these operations occur locally, entirely self-contained without any interaction with external servers or cloud services. This local processing ensures fast response times, enhanced security, and complete independence from internet connectivity, making the system robust and reliable.

4.1.2 LED Traffic Light Module: Specifications, Compatibility and Role in this Project

To simulate traffic light behavior, for this project, four compact modules with three LEDs were used (Figure 4. [22]). These are controlled via four pins: one for each color (R, Y, G) and a common ground pin (GND). By selectively activating the R, Y, or G pins, the corresponding traffic light colors are illuminated. Connecting this to the Raspberry Pi is done via GPIO pins, with the sequential control of the lights achieved through synchronized software commands.

The advantage of this module lies in its simple connections and the flexibility it offers in educational or experimental scenarios. Additionally, the small size of the traffic light makes it ideal for physical prototypes of scaled intersections.



Figure 17. LED Traffic Light Module [22]

LED Traffic Light Module specification [22]:

- Supply Voltage: 3.3V or 5V DC
- LED Colors: Red, Yellow, Green
- LED Type: Common cathode
- Input: Digital
- LED Size: 8mm
- Dimensions: 56 x 21 x 11 mm

Given its specifications, this LED Traffic Light Module is an ideal choice for the adaptive traffic control system, especially due to its compatibility with the Raspberry Pi. Since it operates at 3.3V DC, the module integrates perfectly with the Raspberry Pi's 3.3V GPIO pins, eliminating the need for complex voltage adaptation circuits. Control is straightforward thanks to its digital input, allowing the red, yellow, or green LEDs to turn on or off directly using simple HIGH/LOW signals. Additionally, its common cathode configuration further simplifies wiring. With 8mm LEDs and a compact size (56 x 21 x 11 mm), the module offers adequate visibility in a discreet and lightweight format, making it a perfect choice for both prototyping and small-scale implementations.

Within the project, this module serves as the crucial visual output interface for the adaptive traffic management system. It translates the real-time decisions made by Raspberry-based on vehicle detection and traffic flow optimization-into clear, actionable traffic signals. By accurately displaying the red, yellow, or green light, the module directly facilitates the adaptive control of traffic flow, ensuring the system effectiveness in reducing congestion and improving road safety.

4.1.3 USB Video Cameras and Powered USB Hub

USB cameras are a crucial component of the adaptive traffic light system, used for real-time traffic detection by capturing images of the intersection. These cameras are integrated with the Raspberry Pi and play a key role in providing essential data for the YOLO model, ensuring accurate and precise vehicle detection and counting.



Figure 18. USB Camera [23]

In this project, four standard USB cameras are used (Figure 5. [23]), identified by persistent paths. These cameras are classified as V4L2 (Video4Linux2) devices, ensuring compatibility with Raspberry Pi OS.

Technical Details of the USB Cameras [23]:

- Resolution: 1080p
- Interface: USB 2.0
- Frame Rate: 30 FPS
- Sensor: CMOS
- Current Consumption: 400 mA

The Raspberry Pi, while incredible versatile, comes with a notable physical limitation regarding the current it can supply through its onboard USB ports. This limit is typically around 1.2 Amperes (A) for all connected peripherals combined. This seemingly minor detail becomes a critical consideration when integrating multiple USB cameras.

As the technical specifications for our chosen USB cameras indicate each camera consumes around 400 mA. When multiplying this by the four cameras required for our adaptive traffic system ($400 \text{ mA/camera} * 4 \text{ cameras} = 1600 \text{ mA}$ or 1.6 A), it becomes immediately apparent that the total current draw significantly exceeds the Raspberry Pi's 1.2 A USB power budget.

Attempting to power all four cameras directly from the Raspberry Pi in such a scenario would inevitably lead to a range of undesirable and disruptive issues, such as:

- System Instabilities: Insufficient power can cause the cameras to malfunction intermittently, leading to corrupted data streams or unexpected behavior
- Frequent System Restarts: When the power demand fluctuates and briefly dips below the required threshold, the Raspberry Pi's power management unit might detect this as a critical event, triggering an automatic reboot to protect the system.

To effectively mitigate these challenges and ensure the robust, stable operation of the multi-camera setup, a straightforward yet effective solution was implemented: the integration of a powered USB hub (Figure 6. [24]). Unlike passive USB hubs that simply split the existing power from the host device, a powered USB hub has its own external power supply.



Figure 19. HAMA Powered USB Hub [24]

Powered USB Hub Specifications [24]:

- Number of ports: 4
- Interface: USB 2.0
- Transfer Speed: Up to 480 Mbps
- Power Supply: External 5V/2A, exceeding the Raspberry Pi's limit

By utilizing this powered USB hub, each of the four cameras receives its necessary current directly from the hub's dedicated power supply, rather than relying solely on Raspberry Pi's limited USB power output.

In essence, the powered USB hub acts as a vital intermediary, bridging the gap between the cameras power requirements and the Raspberry Pi's limitations, thereby guaranteeing a much better functioning of the visual data acquisition subsystem.

4.2 Software Technologies Used

4.2.1 Python

Python serves as the central pillar of the project's software approach. It was chosen for its versatility and its ability to support a rich ecosystem of theoretical tools. Furthermore, it is the primary programming language used for Raspberry Pi, making it ideal for this project.

Initially developed by Guido van Rossum and released in 1991, Python has evolved into an interpreted, object-oriented programming language renowned for its intuitive syntax and vast community that contributes to its development. It's widely used in fields like artificial intelligence, data analysis, and embedded systems due to its modularity and design, which favors rapid prototyping. Its continuous development always introduces new improvements in memory management and performance optimizations, making it suitable for applications on resource-constrained devices. The choice of this language is based on its free, open-source availability and its support for a theoretical development environment, which facilitates concept exploration [25].



Figure 20. Python Logo [26]

The project relies on a set of essential libraries, each extending Python's capabilities in a specific and crucial way for the system overall functionality. These libraries aren't just isolated tools; they form a coherent structure, facilitating the integration of object detection, data processing, and hardware system management. The main libraries used are:

- **ultralytics:** This library provides a robust framework for implementing object detection models, especially those based on the popular YOLO (You Only Look Once) family. It's designed to process images and accurately identify various entities such as vehicles, pedestrians, or other objects of interest, transforming raw visual data into structured and actionable information. [27]
- **cv2 (OpenCV):** Representing an extensive suite tool for image processing and computer vision, cv2 is essential for visual data analysis. Its capabilities range from defining regions of interest (ROI) and applying filters, to complex tasks like object tracking and facial recognition. [28]
- **RPi.GPIO:** For interacting with the physical world, RPi.GPIO is indispensable. It provides an intuitive model for interacting with the general input/output (GPIO) pins of the Raspberry Pi. Through this library, the system can conceptually control signaling, allowing for the activation of actuators, reading of sensors, or management of other hardware components. [29]
- **Flask:** To provide a user interface and enable remote control, Flask acts as a lightweight web development micro-framework. It facilitates the creation of fast and simple web interfaces, allowing for the conceptual integration of the system with remote monitoring and control via an HTTP-based interface. This way, users can interact with the system remotely, either to view data or control it. [30]

Python's role in this project is to act as a binder between image analysis processing and signaling control. It facilitates the exploration of traffic detection methods and the prediction of optimal signaling phases durations, providing a flexible framework for adapting to various traffic scenarios. Its versatility enables the development of optimization strategies, paving the way for the practical implementation of the solution.

4.2.2 YOLOv11 Model for vehicle detection

YOLO (You Only Look Once) is a family of computer vision algorithms that detect objects in a single pass of an image, offering an efficient compromise between speed and accuracy. YOLOv11, a recent version, developed by Ultralytics and released in 2024, is based on an optimized convolutional neural network (CNN). It divides the image into a grid and simultaneously predicts classes (e.g., car) and bounding box coordinates. The model is trained on extensive datasets, such as COCO, and adapted for vehicle recognition, providing a scalable solution for traffic analysis. Its main advantage lies in its ability to process complex images in a short time, making it suitable for dynamic systems [27].

Object detection with YOLO is a powerful technique that can find and identify various objects in an image. It starts by taking an input image and processing it in a very efficient way (Figure 8. [11]).

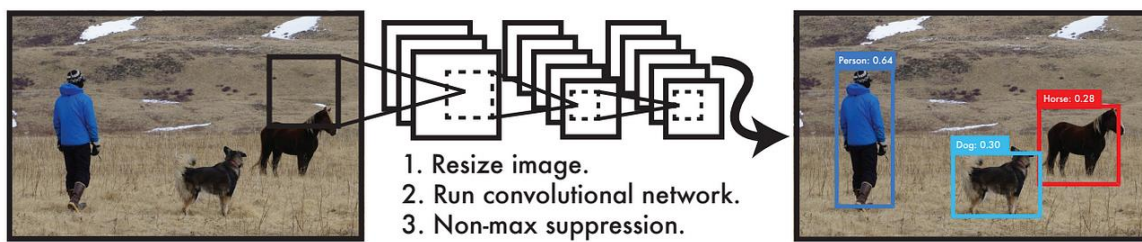


Figure 21. Processing flow of the YOLO model for object detection [11]

How it works [11]:

1. Image Preparation: First the image is prepared by resizing it to a standard dimension (e.g., 448 x 448 pixels). This standardized size is fed into the detection system.
2. Single Pass Detection: The resized image then goes through a powerful convolutional neural network in a single phase. During this single “look”, the network effectively divides the image into an $S \times S$ grid and, for each cell of this grid, simultaneously predicts:
 - a. Bounding Boxes and Confidence: Each cell predicts a fixed number of potential bounding boxes that might contain an object. For each predicted box, it also outputs a confidence score. This score reflects the likelihood that an object is present within that box and how accurate the box's position is.
 - b. Class Probabilities: Each cell also predicts class probabilities for the object it believes is present. This is visualized as the “Class probability map” in Figure 9., where different colors might represent the likelihood of different object classes (like dog or a bicycle) being in the grid cell.

All these predictions from the entire grid are encoded into a comprehensive output, often described as a tensor, ready for the next stages.

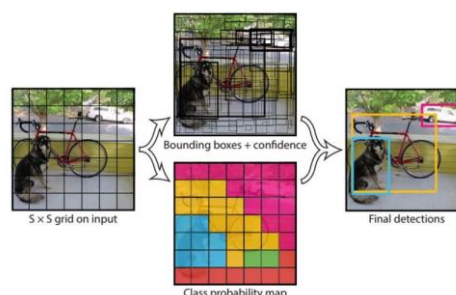


Figure 22. Bounding Boxes and Class Predictions Map [11]

3. **Initial Filtering by Confidence:** After the network outputs all these predictions, there will be many bounding boxes, some of them with very low confidence scores. To clean this up and focus on promising detections, any predicted bounding boxes with a confidence score below a certain predefined threshold are simply discarded. This helps to remove noise and less reliable detections.
4. **Refining Detections with Non-Maximum Suppression:** Even after filtering by confidence, it's common to have multiple, slightly overlapping bounding boxes predicting the same object. This is where Non-Maximum Suppression (NMS) comes in. NMS intelligently identifies and removes these redundant boxes, keeping only the most confident and best-fitting box for each object. This ensures that each object is detected only once, providing a clean set of results.

Like any object detector, YOLOv11 utilizes sophisticated architecture typically composed of three main components: Backbone, Neck, and Head [31] (as seen in Figure 10. [31]).

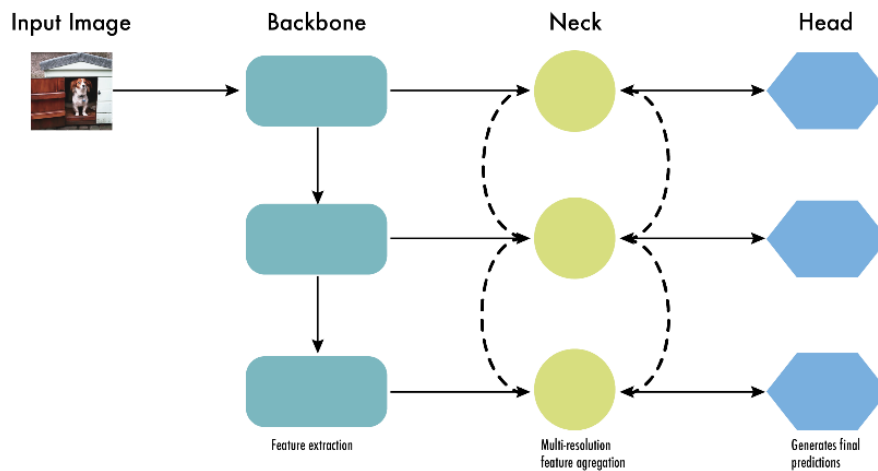


Figure 23. YOLO Architecture [31]

- **Backbone:** This initial part of the network is responsible for extracting basic features from the image. It processes the raw pixel data to capture fundamental visual details such as edges, contours, textures, forming the foundation for detection [31].
- **Neck:** Following the Backbone, the Neck component combines and refines these extracted features. It uses techniques like the Feature Pyramid Network (FPN) to aggregate features from different scales [31]. This is crucial for car detection as it allows the model to effectively recognize cars regardless of size, colors and from different positions within the image.
- **Head:** The final component, the Head, takes the refined features from the Neck and directly makes predictions. For each cell in the $S \times S$ grid, it outputs, as described above:
 - The potential location and size of the objects (Bounding Boxes)
 - The class of the object. (e.g., identifying it specifically as a “car”)
 - A confidence score, indicating how likely it is that an object exists within that box and how accurate the box’s position is, based on a minimum confidence threshold

When it comes to specific car detection, the process focuses heavily on recognizing distinctive vehicle features such as the body shape, headlights, general dimensions, and unique patterns. This is achieved through extensive specialized training on diverse car datasets covering various models, angles, lighting, and environmental conditions.

For car detection, the algorithm can be strategically oriented towards regions of interest, using OpenCV, such as specific traffic lanes on the road. This focuses on the detection efforts where they are most relevant, improving efficiency and accuracy for traffic monitoring.

4.2.3 Decision Trees

Decision trees are a machine learning model used to estimate optimal signaling phase durations, based on traffic density analysis. They were selected as the optimal method due to their high interpretability and minimal computational resource requirements, making them suitable for applications on platforms with limited performance, such as those used in urban traffic control systems. This choice is based on their ability to structure decisions in a hierarchical and intuitive manner, providing an efficient solution for modeling discrete data, such as the number of vehicles in various directions. Unlike deep neural networks (DNNs) [32], which require complex infrastructures, specialized hardware, and large volumes of data for training, decision trees operate efficiently with smaller datasets and do not impose high computational demands. Additionally, compared to support vector machines (SVMs) [33], which can be slower and less flexible with discretely varying data, decision trees allow for the definition of simple logic thresholds, such as “number of cars above a certain level”, without requiring elaborate data transformations.

Another relevant comparison is with logistic regression, which assumes linear relationships between variables and requires data normalization to function optimally [34]. This can be a disadvantage in contexts with unstructured or nonlinear data, typical of urban traffic. In contrast, decision trees can capture nonlinear relationships and are more adaptable to traffic fluctuations, such as sudden congestion or seasonal variations. This flexibility, combined with their ability to be easily interpreted by non-specialized users, makes them a strategic choice for systems requiring rapid decisions and dynamic adjustment, laying the groundwork for integration with other traffic analysis methods [35].

Decision trees are a machine learning model that structures the decision-making process as a hierarchy of nodes and branches, drawing inspiration from graphical representations of logical processes. Each node in the tree represents a condition based on an input variable, such as traffic density in a specific direction, while the branches indicate possible responses to this condition. The leaves of the tree, located at the end of the branches, contain the final outcomes, for example, the estimated duration of the traffic light’s green phase. This model is based on mathematical principles like information maximization (measured by entropy) or Gini impurity minimization, which guide the tree’s construction to separate data efficiently.

Decision trees are versatile, used for both classification (e.g., assigning a class to an object) and regression (predicting a continuous value, such as duration). Their main advantage lies in their ability to model complex relationships between variables through a simple structure (Figure 11. [37]), without requiring prior assumptions about data distribution. This property makes them suitable for traffic analysis, where factors like vehicle count and road conditions can vary significantly, providing a conceptual basis for adaptive systems [36].

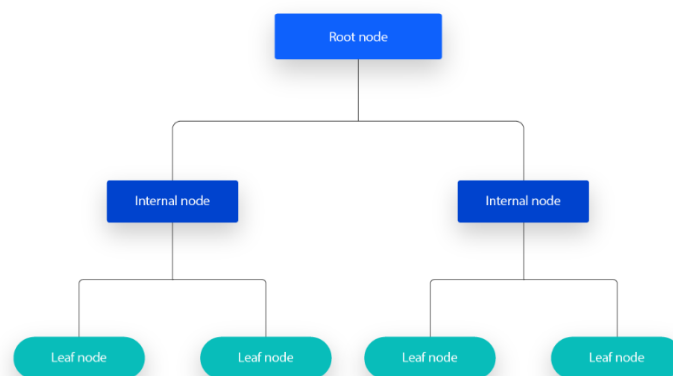


Figure 24. General Structure of a Decision Tree [37]

The decision tree model analyzes a set of input variables, such as traffic density in the North, South, East, and West directions, to estimate the optimal duration of the traffic light's green phase. The decision-making process relies on a hierarchical structure of conditions, where each node evaluates a specific condition. The branches stemming from these nodes lead to other conditions or to leaves, where the final duration is determined.

The tree construction is based on a conceptual dataset that correlates traffic densities with optimal durations, using criteria like Entropy, Information Gain, and Gini impurity to identify the most informative splits between nodes.

Entropy is a measure of uncertainty or disorder within a dataset, derived from information theory. Its mathematical formula is:

$$H = - \sum_k p_{mk} \log(p_{mk}) \quad (1)$$

where p_{mk} represents the probability of class k in a node m . This formula calculates the weighted average of uncertainty, with each class's contribution weighted by its probability. For example, if a node contains 50% observations from class A and 50% from class B, the entropy will be maximal ($H = 1$), indicating a complete mix. If all observations belong to the same class (e.g., 100% class A), the entropy will be 0, reflecting complete purity. In tree construction, the algorithm selects splits that minimize entropy, thereby reducing uncertainty and improving class separation [35].

Gini Impurity measures the degree of heterogeneity of classes within a node, serving as another criterion used to guide divisions. Its mathematical formula is:

$$G = 1 - \sum_k p_{mk}^2 \quad (2)$$

Where p_{mk} is the proportion of each class k in a node m . The Gini value ranges from 0 and $1-1/n$. For example, a node with 100% observations from the same class has $Gini = 0$ (completely pure), and a node with an equal distribution between two classes (e.g., 50% class A, 50% class B) has $Gini = 0.5$, indicating a significant mix. Unlike entropy, which penalizes imbalanced distribution more strongly, Gini is more sensitive to class balance and favors splits that lead to nodes with a single dominant class, providing a fast and efficient alternative for tree construction [35].

Information Gain (IG) measures the reduction in uncertainty achieved by splitting a node, defined as the difference between the parent node's entropy and the weighted average of the child nodes' entropies. Its mathematical formula is:

$$IG = H(\text{parent}) - \sum_j \frac{|S_j|}{|S|} H(S_j) \quad (3)$$

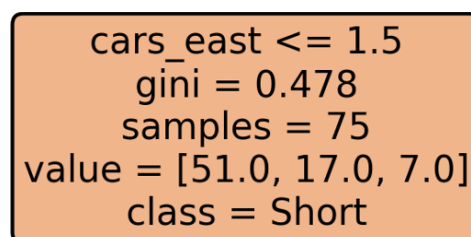
where $H(\text{parent})$ is the entropy of the initial node, S_j represents the subset of data in child node j , $|S_j|$ is the number of observations in the subset, $|S|$ is the total number of observations in the parent node, and $H(S_j)$ is the entropy of the subset. For example, if a parent node has an entropy of 1 (equal mix) and the split separates it into two children with an entropy of 0 (each node pure), the IG will be 1, reflecting a maximum reduction in uncertainty. Information Gain is used to select the most informative splitting feature, preferring divisions that maximize this value.

Several well-known algorithms exist for constructing a decision tree, each with a specific characteristic:

- ID3 (Iterative Dichotomiser 3): Introduced by Ross Quinlan in 1986, this algorithm builds a multi-way tree. It identifies the categorical feature that provides the greatest information gain at each node in a greedy fashion for categorical targets.
- C4.5: This algorithm, succeeded ID3, overcomes the limitation of requiring only categorical features. It can dynamically define discrete attributes from numerical variables, partitioning continuous values into district intervals. C4.5 converts the resulting trees into a collection of “if-then” rules.
- CART (Classification and Regression Trees): Highly similar to C4.5, CART distinguishes itself by supporting numerical target variables, enabling its use for regression tasks, and it does not produce explicit rule sets. CART builds binary trees by selecting the features and threshold that yield the greatest information gain at each node.

The popular Scikit-learn library employs an optimized version of CART algorithm. However, its current implementation does not directly support categorical variables, requiring numerical encoding for such features.

In Scikit-learn’s decision tree representation, each node includes additional information such as Samples, Value and Gini (Figure 12.). Samples indicate the total number of observations associated with that node, providing a picture of the volume of data analyzed. Value represents the distribution of the number of observations for each class or possible outcome within that node.



```
cars_east <= 1.5
gini = 0.478
samples = 75
value = [51.0, 17.0, 7.0]
class = Short
```

Figure 25. Example of a Decisional Tree Node (Scikit-learn representation)

For example, a node like the one shown in the figure above, with `samples = 75` and `value = [51.0, 17.0, 7.0]`, indicates that there are 75 observations in this node, where 51 of them belong to the first class (Short), 17 to the second class (Medium), and 7 for the third (Long). The `class = Short` indicates the majority class for the specific node. These indicators help to understand the tree’s structure and evaluate the quality of the splits.

5 Implementation

This chapter describes the implementation process of an adaptive traffic light system based on car detection and counting using machine learning. This system utilizes four video cameras, a YOLOv11 model, a decision tree for traffic light timing calculations, and a Flask application for monitoring. The implementation combines hardware and software components, integrating a Raspberry Pi that controls a traffic light module in a scaled intersection and a web interface for real-time monitoring the system.

5.1 Block Diagram

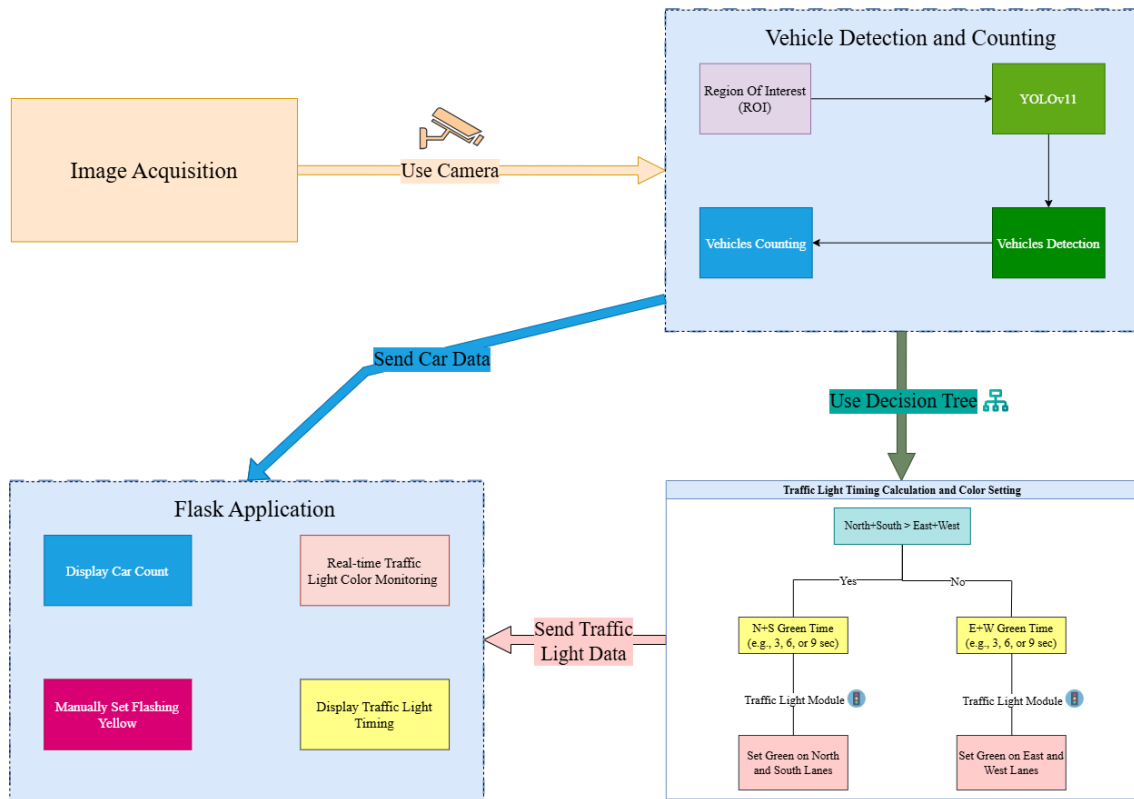


Figure 26. Block Diagram of the Traffic Light Control System

The block diagram above (Figure 13.) illustrates the functional architecture of the implemented system, which integrates image acquisition, vehicles detection and counting, traffic light timing decision-making, and the web application for system status monitoring.

1. Image Acquisition:

The system begins with image acquisition from the USB video cameras mounted around the intersection. These cameras capture real-time sequences, which are then transmitted to the visual analysis module.

2. Vehicle Detection and Counting:

Captured images are processed by the YOLOv11 model, which detects vehicles within the Region of Interest (ROI) for each camera. The detections are then quantified, resulting in the number of vehicles for each traffic direction.

- ROI restricts processing only the relevant areas in the image, reducing errors
- YOLOv11 performs object detection with high accuracy
- Vehicle Counting extracts the numerical data needed for the classifier

3. Traffic Light Timing Calculation and Setting using Decisional Tree:

The collected data (number of vehicles per direction) is fed into a symbolic model – a decision tree – which classifies the situation into one of the green light timing categories (3, 6, or 9 seconds). The decision is based on simple, interpretable rules, set after training the model on randomly generated datasets. Based on the decision made, the durations of the green light phases are set for the North(N)-South(S) or East(E)-West(W) directions. These are then applied to the GPIO logic for the corresponding activation of the traffic lights.

- If the N+S flow is higher, it receives priority timing.
- In the alternative, E+W has priority.

4. Flask Application:

The system includes a Flask application that provides a graphical interface for monitoring and control:

- Display Car Count – shows the number of detected vehicles.
- Real-time Monitoring – tracks the current traffic light colors.
- Display Timing – shows the duration of the green light phase set by the model.
- Manual Yellow Flashing – allows manual activation of an emergency phase.

5.2 Raspberry Pi Configuration and VS Code Remote Development (SSH)

This section describes configuring the Raspberry Pi 4 Model B as the system's central platform and using Visual Studio Code with Remote SSH for remote code development and management, optimizing the implementation process.

5.2.1 Raspberry Pi Configuration

Raspberry Pi 4 Model B was selected as the core of the system due to its robust GPIO management capabilities and excellent support for Linux-based operation systems like Raspberry Pi OS. This setup process began by acquiring the board, along with a 64 GB microSD card and a stable 5.1V/3A power supply.

Raspberry Pi OS was then installed using the Raspberry Pi Imager tool (Figure 14.). During this process, the Wi-Fi network was configured with specific credentials (SSID and password), and a distinct hostname ("trafficlight") was assigned to facilitate remote access. The installation involved formatting the microSD card, writing the OS image, and crucially, enabling SSH for secure remote connections.

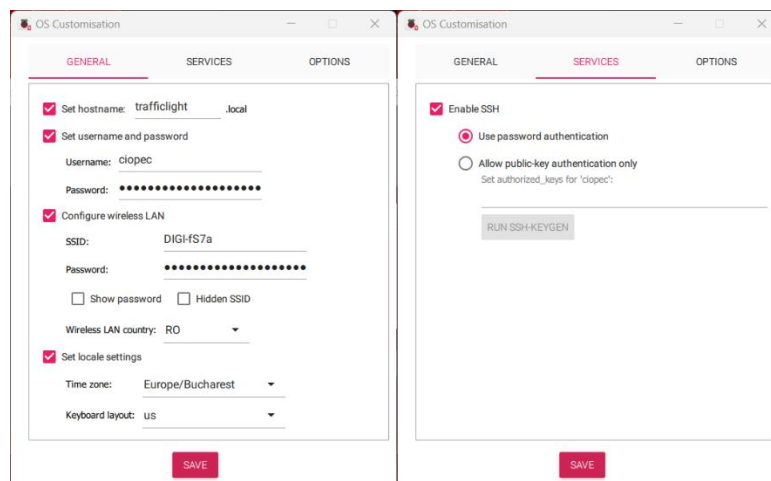
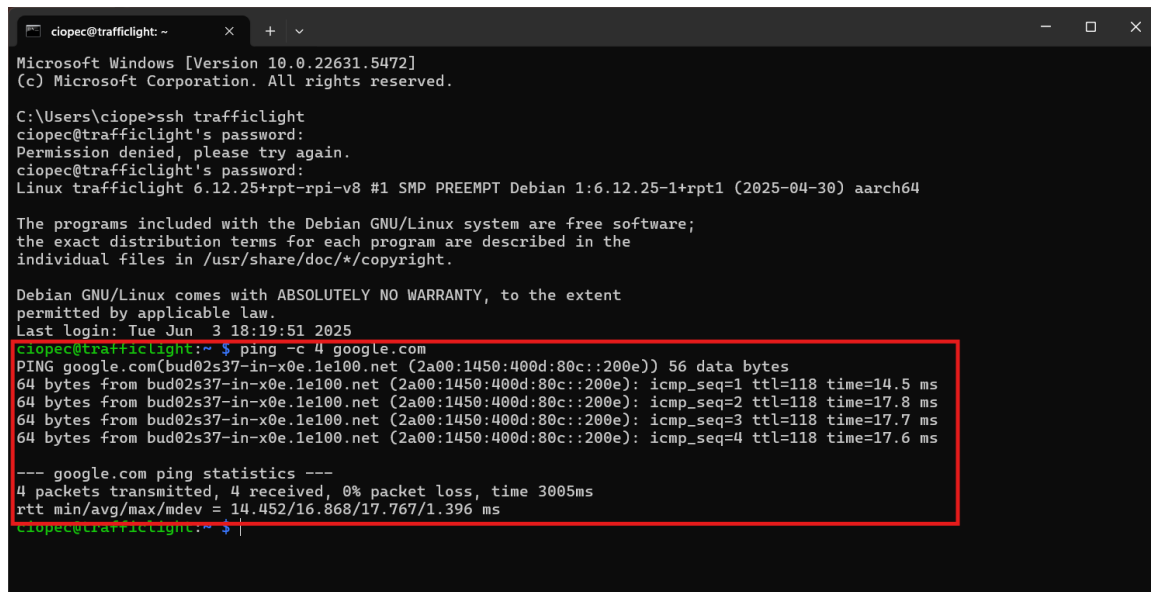


Figure 27. Raspberry Pi Imager configuration window

After the installation was complete, the board's network connection was checked to make sure it was working correctly. This step involved executing a ping command, specifically ping -c 4 google.com, from the terminal. A successful response from google.com confirmed that the board was correctly connected to the internet and could resolve domain names, indicating a stable and functional network link (red outline, Figure 15.).

A terminal window titled 'ciopec@traffilight: ~' showing a Windows command prompt session. The user runs 'ssh traffilight', enters a password, and is prompted for a second password. The terminal then shows the Debian GNU/Linux login banner. The user runs 'ping -c 4 google.com', and the output shows four successful ping requests to google.com with response times around 14-18 ms. The statistics at the bottom show 4 packets transmitted, 4 received, 0% packet loss, and a time of 3005ms. The entire ping output is highlighted with a red rectangle.

```
Microsoft Windows [Version 10.0.22631.5472]
(c) Microsoft Corporation. All rights reserved.

C:\Users\ciopec>ssh traffilight
ciopec@traffilight's password:
Permission denied, please try again.
ciopec@traffilight's password:
Linux traffilight 6.12.25+rpt-rpi-v8 #1 SMP PREEMPT Debian 1:6.12.25-1+rpt1 (2025-04-30) aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Jun 3 18:19:51 2025
ciopec@traffilight:~$ ping -c 4 google.com
PING google.com(bud02s37-in-x0e.1e100.net (2a00:1450:400d:80c::200e)) 56 data bytes
64 bytes from bud02s37-in-x0e.1e100.net (2a00:1450:400d:80c::200e): icmp_seq=1 ttl=118 time=14.5 ms
64 bytes from bud02s37-in-x0e.1e100.net (2a00:1450:400d:80c::200e): icmp_seq=2 ttl=118 time=17.8 ms
64 bytes from bud02s37-in-x0e.1e100.net (2a00:1450:400d:80c::200e): icmp_seq=3 ttl=118 time=17.7 ms
64 bytes from bud02s37-in-x0e.1e100.net (2a00:1450:400d:80c::200e): icmp_seq=4 ttl=118 time=17.6 ms

--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 14.452/16.868/17.767/1.396 ms
ciopec@traffilight:~$
```

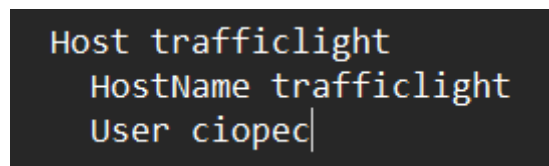
Figure 28. Checking Internet Connection

5.2.2 VS Code Remote Development (SSH)

For developing and debugging the code, Visual Studio Code (VS Code) was used with the Remote SSH extension. This setup allowed for a direct connection to the Raspberry Pi via an Ethernet cable, which provided a stable and fast link for development. The SSH connection was secured using the password configured during the initial installation of the Raspberry Pi OS (Figure 14.).

Below is a detailed description of the configuration and connection process, explaining how the development environment was set step-by-step:

- Connecting via Visual Studio Code:
 - Extension Installation: First, the Remote – SSH extension was installed directly from the Visual Studio Code marketplace.
 - SSH Configuration File: Next, the ~/.ssh/config file (Figure 16.) was set up on the local computer. This file contains the necessary details for the connection:

A screenshot of the SSH config file content, showing the host 'traffilight' with hostname 'traffilight' and user 'ciopec'.

```
Host traffilight
  HostName traffilight
  User ciopec
```

Figure 29. Config File

- Establishing the connection: To connect, “Remote Host – SSH: Connect to Host” was chosen from VS Code command palette (accessed with Ctrl+Shift+P). The “traffilight” option was then selected, then the pre-configured password was entered, which established a secure session.

Once connected, VS Code provided full access and control over the Raspberry Pi, allowing direct code manipulation, file transfer, and command execution in terminal, which significantly streamlined the development and debugging process (Figure 17.).

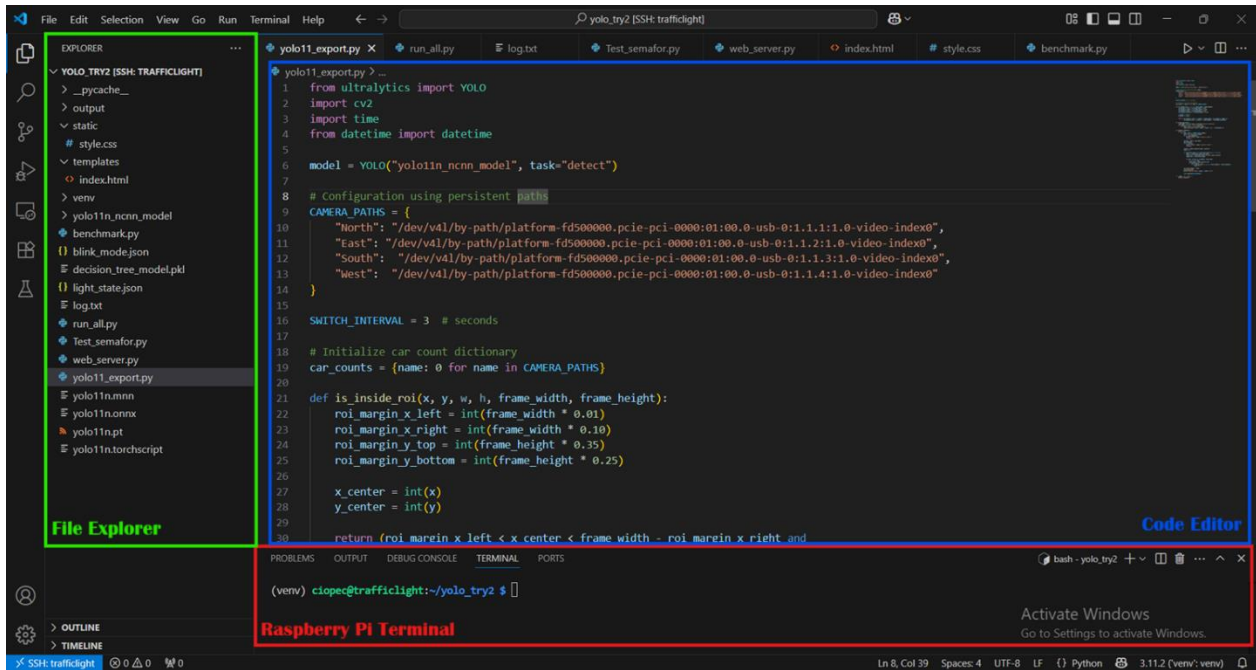


Figure 30. VS Code Interface for Remote Development on Raspberry Pi

5.3 Vehicle Detection and Counting using YOLOv11

5.3.1 Camera Configuration

The configuration of the video cameras involved identifying and utilizing persistent paths for each camera connected to the Raspberry Pi. This approach ensures that the system consistently recognizes each camera regardless of the order in which they are connected or if the system reboots.

These persistent device paths were discovered by exploring the `/dev/v4l/by-path/` directory using the command `"ls /dev/v4l/by-path/"` in the Raspberry terminal to list all available video devices along with the unique, persistent symbolic links (Figure 18.).

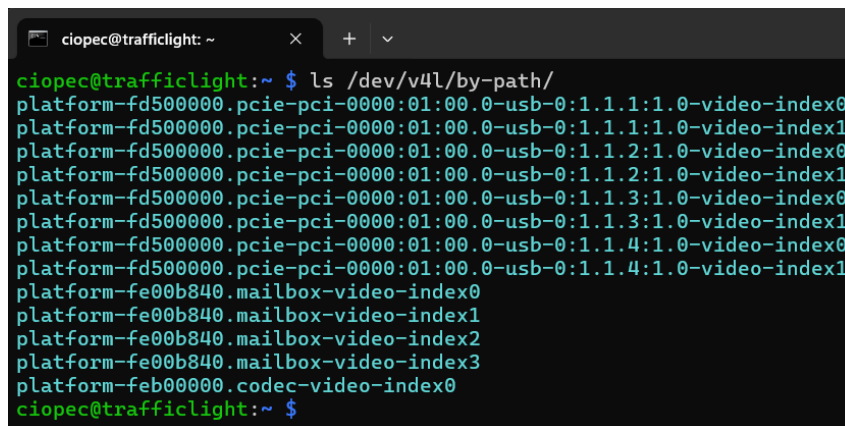


Figure 31. List of Persistent Camera Paths on Raspberry Pi

Each specific path was then explicitly associated with a particular traffic lane direction (North, East, West, South) relevant to the intersection being monitored. This assignment was crucial for accurately mapping video feeds to their respective traffic lanes. Finally, these identified and assigned paths were integrated into the detection code, ensuring that the YOLOv11 model processed the correct video stream for each direction, as shown in the green outline below:

```

9  CAMERA_PATHS = {
10     "North": "/dev/v4l/by-path/platform-fd500000.pcie-pci-0000:01:00.0-usb-0:1.1.1:1.0-video-index0",
11     "East":  "/dev/v4l/by-path/platform-fd500000.pcie-pci-0000:01:00.0-usb-0:1.1.2:1.0-video-index0",
12     "South": "/dev/v4l/by-path/platform-fd500000.pcie-pci-0000:01:00.0-usb-0:1.1.3:1.0-video-index0",
13     "West":  "/dev/v4l/by-path/platform-fd500000.pcie-pci-0000:01:00.0-usb-0:1.1.4:1.0-video-index0"
14 }
15
16 SWITCH_INTERVAL = 3 # seconds

```

Figure 32. Configuration of camera paths in the detection script

The connection of the cameras was established via Raspberry Pi's USB ports, utilizing standard USB cables to ensure physical stability. A significant challenge encountered was the limited bandwidth of the Raspberry Pi's USB ports, which affected performance when simultaneously processing video streams from all four cameras. This issue was mitigated by processing the cameras sequentially, with a switching interval of 3 seconds, defined as `SWITCH_INTERVAL = 3` (as seen in the red outline in Figure 19.). This approach allowed for alternative access to each camera without overstraining the system's resources.

Another problem identified was the insufficient power supply to the cameras, which led to intermittent disconnections. The adopted solution involved using an external USB hub with its own dedicated power supply (Figure 6.), connected to the Raspberry Pi. This setup (Figure 20.) ensures a stable and adequate power supply for all four cameras, resolving the connectivity issues.

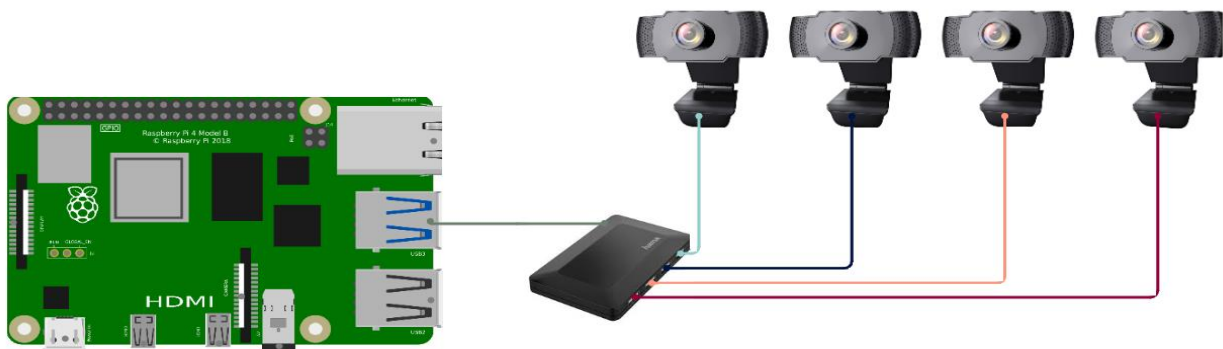


Figure 33. Camera connection with Raspberry Pi

5.3.2 Vehicle Detection and Counting using YOLOv11n in NCNN format

Car detection and counting were implemented using YOLOv11 nano model, optimized with the NCNN format. This specific format was chosen due to its inherent efficiency on resource-constrained devices, such as the Raspberry Pi, making it ideal for edge computing applications where computational power is limited.

The installation of the necessary framework was carried out by installing the Ultralytics library via the “pip install ultralytics” command. Following the library installation, the pre-trained YOLOv11n model was exported into a format compatible with NCNN. This conversion is crucial for optimizing the model for deployment on platforms like Raspberry Pi. The export process was performed using the following command: “yolo export model=yolov11n.pt format=ncnn”. This process generated the “yolo11n_ncnn_model” folder, which contains the necessary files (e.g., .param and .bin) for using the model (Figure 21.).

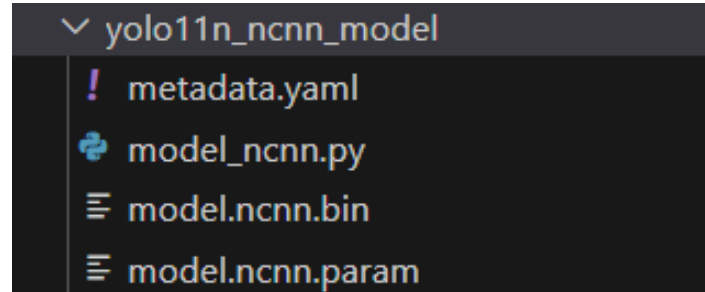


Figure 34. yolov11_ncnn_model Folder

The detection and counting process systematically analyzes video streams from each camera to determine the number of vehicles in respective traffic directions. This sequential approach is vital for efficient resource management on Raspberry Pi. The core steps are as follows:

1. Model Initialization:

The process begins by loading the pre-trained YOLOv11n model, optimized for the NCNN format (Figure 22.). This step prepares the neural network for object detection tasks. The model is initialized by specifying the path to its optimized files (Figure 21.):

```
5
6  model = YOLO("yolo11n_ncnn_model", task="detect")
7
```

Figure 35. YOLOv11n Initialization

2. Region of Interest (ROI) Definition:

To ensure that only relevant detections are counted and to filter out peripheral noise, a specific Region of Interest (ROI) is defined for each camera frame (Figure 23.). This is implemented as a function that checks if the center of a detected object falls within the defined margins. These margins are set as percentages of the frame’s width and height (1% from the left, 10% from the right, 35% from the top, and 25% from the bottom).

```
20
21 def is_inside_roi(x, y, w, h, frame_width, frame_height):
22     roi_margin_x_left = int(frame_width * 0.01)
23     roi_margin_x_right = int(frame_width * 0.10)
24     roi_margin_y_top = int(frame_height * 0.35)
25     roi_margin_y_bottom = int(frame_height * 0.25)
26
27     x_center = int(x)
28     y_center = int(y)
29
30     return (roi_margin_x_left < x_center < frame_width - roi_margin_x_right and
31             roi_margin_y_top < y_center < frame_height - roi_margin_y_bottom)
32
```

Figure 36. ROI Definition

3. Sequentially Frame Processing and Object Detection:

The system continuously iterates through each configured camera. For each camera, its video stream is accessed, a single frame is read, and the connection is then released to conserve resources. Object detection is performed on this acquired frame using the loaded YOLOv11n model, with a confidence threshold set to 0.5 to filter out weaker detections. Error handling is included to manage cases where a camera fails to open or read (Figure 24.).

```
41         for name, path in CAMERA_PATHS.items():
42             cap = cv2.VideoCapture(path)
43             if not cap.isOpened():
44                 print(f"Camera {name} failed to open.")
45                 continue
46
47             success, frame = cap.read()
48             cap.release()
49             if not success:
50                 print(f"Camera {name} failed to read.")
51                 continue
52
53             results = model.predict(frame, conf=0.5)
```

Figure 37. Frame Processing and Object Detection (part of “monitor_cameras” function)

4. Car Counting within ROI:

After detection, the system iterates through all identified bounding boxes. For each detected object, its class ID is checked to confirm it is a “car”. If it is identified as a car, its center coordinates are then passed to the “is_inside_roi” function (Figure 23.), these ensures that only the cars whose centers falls within the defined ROI are incremented in the count for the current camera (Figure 25.).

```
54         count = 0
55
56         if results[0].boxes and results[0].boxes.cls is not None:
57             boxes = results[0].boxes.xywh.cpu()
58             class_ids = results[0].boxes.cls.int().cpu().tolist()
59             names_list = results[0].names
60
61             for box, cls_id in zip(boxes, class_ids):
62                 x, y, w, h = box
63                 class_name = names_list[cls_id]
64                 if class_name == "car":
65                     if is_inside_roi(x, y, w, h, frame.shape[1], frame.shape[0]):
66                         count += 1
```

Figure 38. Car counting within the defined ROI (part of “monitor_cameras” function)

5. Data Update and Logging:

Once the car count for a camera is finalized, the “car_counts” dictionary is updated with this new value. Subsequently, the data is recorded into a log file for later use in the traffic light timing calculations and color adjustments. Concurrently, the updated count is printed to the console, serving as immediate real-time feedback and a valuable aid during the debugging process (Figure 26.).

```
67
68         car_counts[name] = count
69         write_log_file()
70         print(f"Updated Camera {name}: {count} cars")
71
```

Figure 39. Updating car counts and logging the data (part of “monitor_cameras” function)

6. Sequentially Switching Interval:

To manage the limited USB bandwidth of the Raspberry Pi and ensure stable operation, a crucial “SWITCH_INTERVAL” of 3 seconds is implemented (Figure 19.). After processing each camera, the system pauses for this duration before proceeding to the next camera in the sequence. This sequential processing prevents resource overload and ensures that each camera gets sufficient time for data acquisition and processing.

```
71  
72     time.sleep(SWITCH_INTERVAL)  
73
```

Figure 40. Sequential camera switching (part of "monitor_cameras function)

5.4 ML Model Training and Implementation with Traffic Light Module using GPIO

5.4.1 Training the ML Decision Tree Model

The Decision Tree Model was trained on a personal computer (PC) due to the hardware limitation of the Raspberry Pi. The choice of a PC was motivated by the availability of superior resources (CPU and RAM). The process begins with the generation of a synthetic dataset using Python, specifically designed to simulate car counts on each lane (North, South, East, and West). Numerical sets were preferred to simplify the model and ensure consistency of the input data.

The data generation process involved the following steps:

Step 1. Initialization: Constants are defined, and a random seed is set to ensure reproducibility of the dataset generation process (Figure 28.).

```
7  SAMPLES_PER_BUCKET_PER_PHASE = 50 # 50 for each of 3 buckets, for both SN and  
8  MAX_CARS = 3  
9  random.seed(42)  
10  
11  data = []  
12  bucket_counts = {  
13      'SN': defaultdict(int),  
14      'EW': defaultdict(int)  
15  }  
16  
17  tie_counter = 0 # To alternate tie assignments
```

Figure 41. Initialization constants and random.seed()

Step 2. Data Generation: Random car counts are generated for each lane, and the total number of cars for the South- North (SN) and East-West (EW) directions are calculated (Figure 29.).

```
32  cars_south = random.randint(0, MAX_CARS)  
33  cars_north = random.randint(0, MAX_CARS)  
34  cars_west = random.randint(0, MAX_CARS)  
35  cars_east = random.randint(0, MAX_CARS)  
36  
37  sn_total = cars_south + cars_north  
38  ew_total = cars_east + cars_west  
39
```

Figure 42. Random generation of cars counts for each traffic lane

Step 3. Phase Assignment: The dominant traffic phase (SN or EW) is determined based on which direction has a higher total car count. In cases of a tie in counts, the phase alternates to ensure fairness and prevent bias (Figure 30.).

```

41     if sn_total > ew_total:
42         light_phase = 'SN'
43         active_cars = sn_total
44     elif ew_total > sn_total:
45         light_phase = 'EW'
46         active_cars = ew_total
47     else:
48         # Alternate tie between SN and EW
49         light_phase = 'SN' if tie_counter % 2 == 0 else 'EW'
50         active_cars = sn_total # same as ew_total
51         tie_counter += 1

```

Figure 43. The logic for assigning the dominant traffic light phase

Step 4. Bucketing Green Time: The calculated active car total is classified into discrete intervals or “buckets”, which correspond to different green light durations. This function maps a range of car counts to a specific time bucket (Figure 31.).

```

19 def assign_bucket(total_cars):
20     if total_cars <= 2:
21         return 0 # Short
22     elif total_cars <= 4:
23         return 1 # Medium
24     else:
25         return 2 # Long
26

```

Figure 44. Function used to classify total cars into predefined green time duration

Step 5. Dataset Balancing: Samples are added to the dataset only if the current bucket for a given light phase (SN or EW) has fewer than the predefined number of samples (SAMPLES_PER_BUCKET_PER_PHASE = 50) (Figure. 28). This is crucial for balancing the dataset and preventing the model from being biased towards over-represented scenarios.

```

56     if bucket_counts[light_phase][green_time_bucket] < SAMPLES_PER_BUCKET_PER_PHASE:
57         data.append({
58             'cars_south': cars_south,
59             'cars_north': cars_north,
60             'cars_west': cars_west,
61             'cars_east': cars_east,
62             'light_phase': light_phase,
63             'green_time_bucket': green_time_bucket
64         })
65         bucket_counts[light_phase][green_time_bucket] += 1

```

Figure 45. Dataset Balancing Mechanism

Step 6. Data Saving: Finally, the generated synthetic data is converted into Pandas DataFrame and saved as a CSV file. This file, named “traffic_timing_balanced.csv”, serves as the input for the subsequent model training phase.

```

77 df = pd.DataFrame(data)
78 output_path = r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\data\traffic_timing_balanced.csv"
79 df.to_csv(output_path, index=False)
80 print(f"\nsaved to: {output_path}")

```

Figure 46. Data Converting and Saving

The necessary libraries from Scikit-learn, along with Pandas and Matplotlib for data handling and visualization, and Joblib for model serialization, were installed on the PC using “pip install scikit-learn pandas matplotlib joblib”. The actual model training process was performed as follows:

1. Data Loading and Preparation: The initial phase of model training involves loading the previously generated synthetic traffic data and preparing it for the machine learning algorithm. The CSV file containing this data is loaded into a Pandas DataFrame. Afterwards, this dataset is divided into two distinct components: the features (X), which consist of car counts for each traffic direction, and the target variable (y), which represents the corresponding assigned green time bucket (figure 34.).

```
18 df = pd.read_csv(DATA_PATH)
19 x = df[['cars_south', 'cars_north', 'cars_west', 'cars_east']]
20 y = df['green_time_bucket']
```

Figure 47. Script for Data Loading and Preparation

2. Data Splitting: Following data preparation, the dataset is systematically divided into training and testing sets. A “test_size” of 20% is allocated, meaning 80% of the data is used for training the model. Crucially, stratification is applied during this split. This ensure that the proportional representation of each “green_time_bucket” is maintained across both the training and testing subsets, which is vital for robust model evaluation and preventing bias. A “random_state” is set for reproducibility of the split (Figure 35.).

```
24
25 x_train, x_test, y_train, y_test = train_test_split(
26     x, y, stratify=y, test_size=0.2, random_state=42
27 )
28
```

Figure 48. Data Splitting

3. Model Training and Saving: With the data loaded, prepared, and split into training and testing sets, the next step is the actual model training. A decisional Tree Classifier is initialized and configured for this task. To prevent overfitting and ensure the model generalizes well the unseen data, a “max_depth = 5” is set. A “random_state = 42” is also specified for reproducibility of the training process. The classifier is then trained by fitting it to the “X_train” (training features) and “y_train” (target values) (red outline in Figure 36.).

Upon successful training, the trained model (represented by “clf”) is serialized. This process converts the model into a binary format, which is then saved to a “.pkl” file, at a designated “MODEL_PATH”, using the “joblib” library (green outline Figure 36.).

```
29
30 clf = DecisionTreeClassifier(max_depth=5, random_state=42)
31 clf.fit(x_train, y_train)
32
33 MODEL_PATH = os.path.join(BASE_DIR, "models", "decision_tree_model.pkl")
34 joblib.dump(clf, MODEL_PATH)
35
```

Figure 49. Model Training and Saving

5.4.2 Connecting Traffic Light Modules to the Board

The configuration of the traffic light modules on the Raspberry Pi board involved the physical connection of four modules, each representing a specific traffic direction (North, South, East, and West), via GPIO pins (Figure 37.). Each traffic light module is designed with four pins: red (R), green (G), yellow (Y), and ground (GND). These modules feature internal resistors, which means they do not require additional external ones when connected.

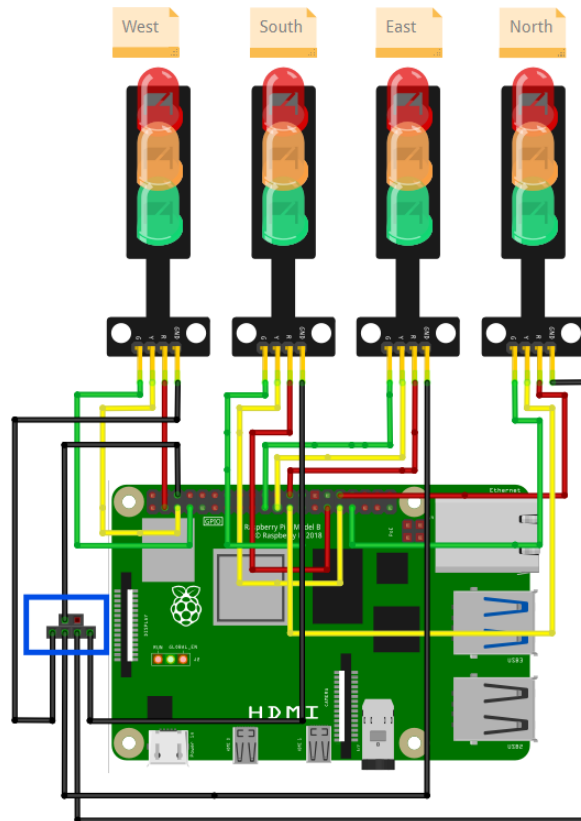


Figure 50. Modules configuration with Raspberry Pi

The connections were established using jumping wires running from the Raspberry Pi's GPIO pins to the corresponding pins on the traffic light modules. A common ground connection was made by connecting the GND pin of each module to Pin 6 (GND) of Raspberry Pi (blue outline in Figure 37.). This approach simplifies the overall circuit wiring by using a single shared return path for current from all modules, rather than requiring individual ground wires for each. This design choice contributes to a cleaner and more manageable setup, reducing complexity while ensuring proper functionality.

The allocation of GPIO pins to each module's respective colors and direction was defined within the system's Python code, ensuring precise control over each segment of the traffic light. This code, which links the logical control to physical pins, is presented in the figure below:

```
7 TRAFFIC_LIGHTS = {
8     "West": {"red": 2, "yellow": 3, "green": 4},
9     "South": {"red": 5, "yellow": 6, "green": 7},
10    "East": {"red": 8, "yellow": 9, "green": 10},
11    "North": {"red": 12, "yellow": 11, "green": 13},
12 }
```

Figure 51. Allocation of GPIO pins to the modules in Python

5.4.3 Implementing the Trained Model in the Project

The trained model was integrated into the project by loading its .pkl file onto the Raspberry Pi. The connection with the traffic light modules was made through the control of GPIO pins, configuring the states of the lights (red, yellow, green) based on the model predictions. The following steps illustrate the process of this implementation:

1. System Initialization and Configuration: The initial step involves importing necessary libraries, defining global constants for GPIO pin mapping, model paths, and timing parameters. The trained Decision Tree model is loaded at startup using “joblib.load()”. The “setup()” function then configures the Raspberry Pi’s GPIO pins, setting the numbering mode, disabling warnings, and initializing all traffic light outputs pins to LOW (off) state.

```
1 import RPi.GPIO as GPIO
2 import json
3 import time
4 import threading
5 import joblib
6
7 TRAFFIC_LIGHTS = {
8     "West": {"red": 2, "yellow": 3, "green": 4},
9     "South": {"red": 5, "yellow": 6, "green": 7},
10    "East": {"red": 8, "yellow": 9, "green": 10},
11    "North": {"red": 12, "yellow": 11, "green": 13},
12 }
13
14 MODEL_PATH = "/home/ciopec/yolo_try2/decision_tree_model.pkl"
15 LIGHT_STATE_FILE = "light_state.json"
16 LOG_FILE = "log.txt"
17 BLINK_FILE = "blink_mode.json"
18
19 # Timing constants (seconds)
20 YELLOW_TIME = 4.0 # Exactly 2 seconds for yellow phase
21 BLINK_COUNT = 3
22 BLINK_ON_TIME = 0.5
23 BLINK_OFF_TIME = 0.5
24
25 model = joblib.load(MODEL_PATH)
26 current_duration = 3 # fallback duration
27 last_green_dirs = None
28
29 def get_blink_state():
30     try:
31         with open(BLINK_FILE, "r") as f:
32             return json.load(f).get("blink", False)
33     except:
34         return False
35
36 def setup():
37     GPIO.setmode(GPIO.BCM)
38     GPIO.setwarnings(False)
39     for pins in TRAFFIC_LIGHTS.values():
40         for pin in [pins["red"], pins["yellow"], pins["green"]]:
41             GPIO.setup(pin, GPIO.OUT)
42             GPIO.output(pin, GPIO.LOW)
```

Figure 52. System Initialization, Global Constants, and GPIO Setup

2. Traffic Light Control Utility Functions: A set of functions is defined to manage the state of individual traffic lights and groups of lights. The “set_light()” function is fundamental, responsible for turning off all lights for a specific direction before activating the designated color.

```
44 def set_light(direction, color):
45     pins = TRAFFIC_LIGHTS[direction]
46     for c in ["red", "yellow", "green"]:
47         GPIO.output(pins[c], GPIO.LOW)
48     GPIO.output(pins[color], GPIO.HIGH)
49     update_state(direction, color)
50     print(f"Set {direction} to {color}")
51     time.sleep(0.1) # Short delay to ensure GPIO stability
```

Figure 53. set_light () Function

Helper functions like “all_red()”, “green_solid()”, “yellow_on()”, and “green_off()” support “set_light()” to control multiple light simultaneously or transition specific states. The “update_state()” function ensures the current light states are saved to a JSON file for external monitoring.

```

53 def update_state(name, color):
54     try:
55         with open(LIGHT_STATE_FILE, "r") as f:
56             state = json.load(f)
57     except:
58         state = {}
59         state[name] = color
60     with open(LIGHT_STATE_FILE, "w") as f:
61         json.dump(state, f)
62
63 def all_red(directions):
64     for d in directions:
65         set_light(d, "red")
66
67 def green_solid(directions):
68     for d in directions:
69         set_light(d, "green")
70
71 def yellow_on(directions):
72     for d in directions:
73         set_light(d, "yellow")
74
75 def green_off(directions):
76     for d in directions:
77         pins = TRAFFIC_LIGHTS[d]
78         GPIO.output(pins["green"], GPIO.LOW)
79         update_state(d, "off")
80         print(f"Green off for {d}")
81         time.sleep(0.1) # Short delay for GPIO stability

```

Figure 54. Helper Function to Control Multiple Traffic Light

3. Adaptive Traffic Light Logic (Data Acquisition and Prediction): The “predict_and_update()” function constitutes the main operational loop of the system. It continuously executes, reading real-time traffic data, making predictions, and managing light states. The loop begins by checking if the manual blink mode is active, pausing its adaptive logic if so. It then acquires the latest car counts for each direction by parsing the “log.txt” file generated by the detection module (Figure 26.). These counts are then used as input for the trained Decision Tree model to predict the optimal green light duration bucket.

```

97 def predict_and_update():
98     global current_duration, last_green_dirs
99
100     while True:
101         if get_blink_state():
102             print("Blink mode active, skipping prediction")
103             time.sleep(0.5)
104             continue
105
106         try:
107             with open(LOG_FILE, "r") as f:
108                 lines = f.readlines()
109
110                 counts = {'North': 0, 'East': 0, 'South': 0, 'West': 0}
111                 for line in lines:
112                     if line.startswith("Camera"):
113                         parts = line.strip().split()
114                         direction = parts[1].strip(":")
115                         count = int(parts[2])
116                         if direction in counts:
117                             counts[direction] = count
118
119                 X = [[counts['South'], counts['North'], counts['West'], counts['East']]]
120                 bucket = model.predict(X)[0]
121                 bucket_to_seconds = {0: 3, 1: 6, 2: 9}
122                 total_green_time = bucket_to_seconds.get(bucket, 3)

```

Figure 55. Read Car Counts and Generate Model Prediction

4. Adaptive Traffic Light Logic (Direction Determination and Transition Management): Following the prediction of the green time, this part of the “predict_and_update()” function determines which traffic directions should receive the green light based on the current car totals (SN vs EW). It then implements a sophisticated transition logic. If the newly predicted green directions are the same as the previously active ones, the green light remains solid. However, if a switch is required, a sequence of blinking green lights, followed by a yellow light phase, is executed for the outgoing directions before new directions are granted a solid green light.

```

124     sn_total = counts['South'] + counts['North']
125     ew_total = counts['East'] + counts['West']
126
127     if sn_total >= ew_total:
128         green_dirs = ['South', 'North']
129         red_dirs = ['East', 'West']
130     else:
131         green_dirs = ['East', 'West']
132         red_dirs = ['South', 'North']
133
134     print(f"Counts: {counts}, Bucket: {bucket}, Green time: {total_green_time}s, Green: {green_dirs}, Red: {red_dirs}")
135
136     if last_green_dirs == green_dirs:
137         # Same green group: solid green full time
138         print(f"Same green directions {green_dirs}, solid green for {total_green_time}s")
139         green_solid(green_dirs)
140         all_red(red_dirs)
141         time.sleep(total_green_time)
142     else:
143         if last_green_dirs:
144             # Blink green exactly BLINK_COUNT times
145             print(f"Blinking green on old green directions {last_green_dirs} {BLINK_COUNT} times")
146             for _ in range(BLINK_COUNT):
147                 green_off(last_green_dirs)
148                 time.sleep(BLINK_OFF_TIME)
149                 green_solid(last_green_dirs)
150                 time.sleep(BLINK_ON_TIME)
151
152             # Yellow on old green directions for exactly 2 seconds
153             print(f"Yellow on old green directions {last_green_dirs} for {YELLOW_TIME}s")
154             yellow_on(last_green_dirs)
155             all_red(red_dirs)
156             time.sleep(YELLOW_TIME)
157
158             # Turn old green directions red
159             print(f"Setting old green directions {last_green_dirs} to red")
160             all_red(last_green_dirs)
161
162             time.sleep(0.5) # Short pause before new green
163
164             # Ensure red on red_dirs
165             all_red(red_dirs)
166
167             # New green directions solid green
168             print(f"New green directions {green_dirs}, solid green for {total_green_time}s")
169             green_solid(green_dirs)
170             time.sleep(total_green_time)
171
172             last_green_dirs = green_dirs
173
174     except Exception as e:
175         print(f"Prediction error: {e}")
176         time.sleep(3)

```

Figure 56. Determine Active Lanes and Manage Light Transition

5. System Execution Flow: The “run()” function act as the main entry point managing the system. It first calls “setup()” function for initialization, then in parallel starts “blink_thread()” for manual blink mod and finally launches “predict_and_update()” to drive the adaptive traffic logic. The “if __name__ == “__main__”:

```

178 def run():
179     setup()
180     threading.Thread(target=blink_thread, daemon=True).start()
181     predict_and_update()
182
183 if __name__ == "__main__":
184     run()

```

Figure 57. Main System Execution and Thread Management

5.5 Web Application (Flask Dashboard)

The web application, developed using Flask framework, serves as a real-time monitoring and control interface for the adaptive traffic light system. It provides a live visualization of traffic light states, important data about the system, and displays when camera errors appear.

5.5.1 Web Application Architecture

The Flask application interacts with other system modules primarily through JSON and log files. This file-based approach simplifies communication between processes running on the Raspberry Pi:

- “light_state.json”: Store the current state of each traffic light.
- “log.txt”: Contains the number of vehicles detected by each camera.
- “traffic_times.json”: Records the green light duration calculated by the ML model
- “blink_mode.json”: A flag file used to enable/disable the emergency blink mode.
- “camera_status.json”: Stores information about camera errors.
- “users.json”: A simple file for storing authentication credentials.

5.5.2 Backend Implementation (“web_server.py”)

The backend component of the web application is implemented in Python using Flask. It manages web routes, reading and writing state files, user authentication, and sending data to the frontend interface.

1. Initialization and Data Access: The application configures Flask and defines constants for file paths. Utility functions, like “get_blink_state()” simplify the process of reading dynamic system data from these files, ensuring robust data retrieval.

```
1 from flask import Flask, render_template, redirect, jsonify, request, session, flash
2 import json
3
4 app = Flask(__name__)
5
6 STATE_FILE = "light_state.json"
7 BLINK_FILE = "blink_mode.json"
8 LOG_FILE = "log.txt"
9 # ... (other file constants)
10 def get_blink_state():
11     try:
12         with open(BLINK_FILE, "r") as f: return json.load(f).get("blink", False)
13     except: return False
14 # ... (other read_ and get_ functions for car_counts, traffic_times, camera_status, users)
```

Figure 58. Key Flask setup and a data reading utility

2. Dashboard and Update Routes: The “/” route renders the main page after user authentication, populating with current system states and data. The “/update” API route provides real-time data to the frontend via JSON, enabling dynamic updates without page reloads.

```
4 @app.route( / )
5 def index():
6     if not session.get("logged_in"): return redirect("/login")
7     # ... (read all data: states, car_counts, times, camera_status)
8     return render_template("index.html", lights=states, blinking=camera_status.get("blink", False), # ... (other data) )
9 @app.route("/update")
10 def update():
11     if not session.get("logged_in"): return jsonify({"error": "Unauthorized"}), 401
12     # ... (read all latest data: states, car_counts, times, camera_status)
13     blink = camera_status.get("blink", False) or get_blink_state()
14     return jsonify({"lights": states, "car_counts": car_counts, "times": times, "blinking": blink, "camera_error": camera_status.get("message")})
```

Figure 59. Flask Routes

3. Authentication Routes: The “/login” route handles user authentication by validating credentials against “users.json”, setting a session variable on success. The “/logout” route clears the session.

```
120 @app.route("/login", methods=["GET", "POST"])
121 def login():
122     if request.method == "POST":
123         username = request.form["username"]
124         password = request.form["password"]
125         users = get_users()
126         if users.get(username) == password:
127             session["logged_in"] = True
128             return redirect("/")
129         else:
130             flash("Invalid credentials", "error")
131             return redirect("/login")
132     return render_template("login.html")
133
134 @app.route("/logout")
135 def logout():
136     session.pop("logged_in", None)
137     return redirect("/login")
```

Figure 60. Flask Routes for authentication

5.5.3 Frontend Implementation (HTML/CSS/JavaScript)

The frontend provides the user interface for monitoring the traffic light system, build using HTML for structure, CSS for styling, and JavaScript for dynamic interactions.

1. Admin Login Page: The system access is secured via an administrative login page. This page features input fields for username and password, along with a login button. Upon successful authentication, the user is directed to the main dashboard.

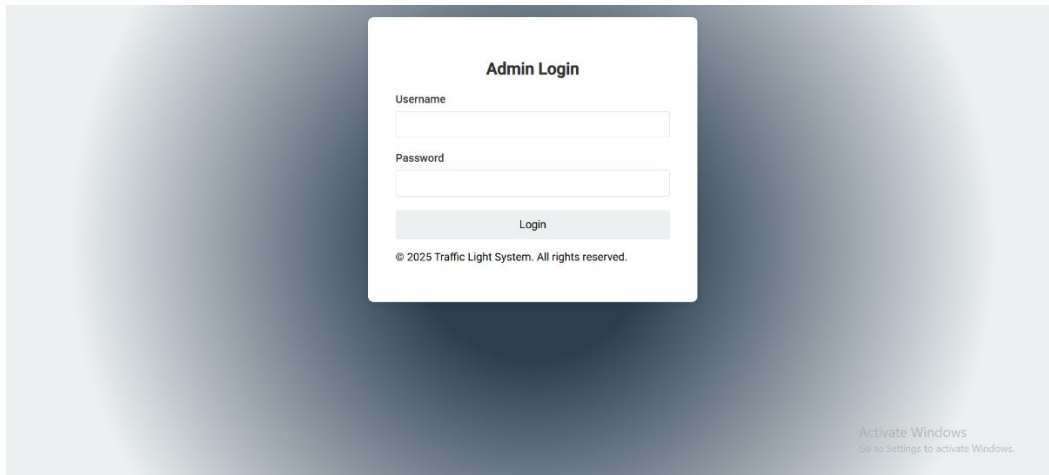
The image shows a web browser window with a light blue background. In the center is a white rectangular box titled "Admin Login". Inside the box, there are two input fields: "Username" and "Password". Below these fields is a grey button labeled "Login". At the bottom of the box, there is a small copyright notice: "© 2025 Traffic Light System. All rights reserved." In the bottom right corner of the browser window, there is a watermark that says "Activate Windows" and "Go to Settings to activate Windows."

Figure 61. Admin Login Page

2. Traffic Light Dashboard: After successfully logging in, administrators gain access to the main dashboard, which serves as the central monitoring hub for the adaptive traffic light system (Figure 49.). Its key components are:
 - Traffic Light Statuses: Individual blocks for each direction, dynamically changing color to reflect the real-time state of the physical traffic lights.
 - Car Count: Display the current number of vehicles detected by cameras at each lane of the intersection, providing insight into traffic density.
 - Traffic Times: Shows the calculated green light durations for each direction, demonstrating the adaptive logic provided by the ML model at work.

- **Manual Control:** A dedicated button allows administrators to put the system into an emergency “blinking yellow” mode, overriding the adaptive logic when necessary.
- **Camera Status:** An integrated error message area that alerts the user to any camera malfunctioning detected by the system, which also automatically triggers the blinking yellow mode.

The dashboard relies on JavaScript to periodically fetch updated data from the backend and refresh the displayed information without requiring a full page reload. This ensures a fluid and real-time user experience.

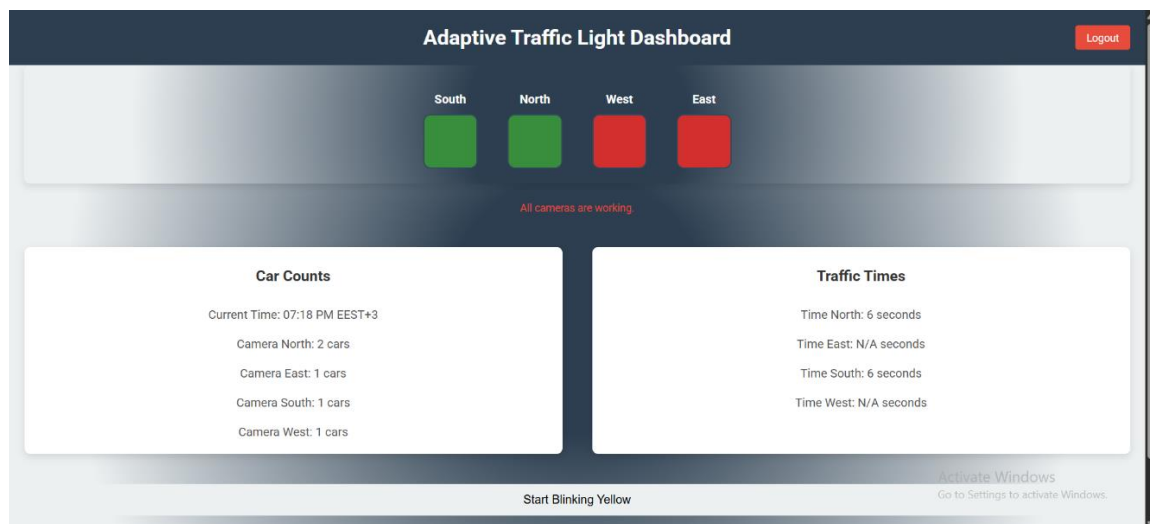


Figure 62. Dashboard Page for Adaptive Traffic Light System

6 Experimental Results

6.1 Physical System Implementation

The adaptive traffic light system was physically implemented using a wooden board (chipboard) on which a four-way intersection (one lane per direction of travel) was represented. This board forms the base of the system, upon which the cameras and traffic lights were mounted. The cameras are strategically positioned using 3D-printed supports, ensuring that camera detection and vehicle counting are as accurate as possible, which is crucial for the system's correct functionality. The supports for the traffic light modules were also made using a 3D printer and were positioned so that the complete image of the system closely resembles a real intersection. The Raspberry Pi was placed underneath the chipboard to be concealed, and to make the wiring organized as neatly as possible.



Figure 63. Physical Implementation (Side View)

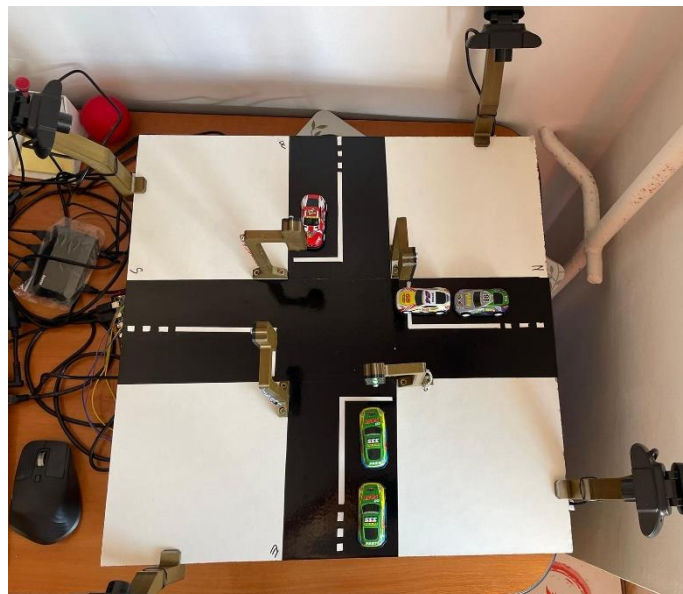


Figure 64. Physical Implementation (Top View)

6.2 Evaluation of vehicle detection and counting using YOLOv11n

Vehicle detection and counting were evaluated across all three possible scenarios (one, two, or three cars). The figures below illustrate the Region of Interest (ROI), the vehicle count displayed in the bottom left corner and the frames per second (FPS).

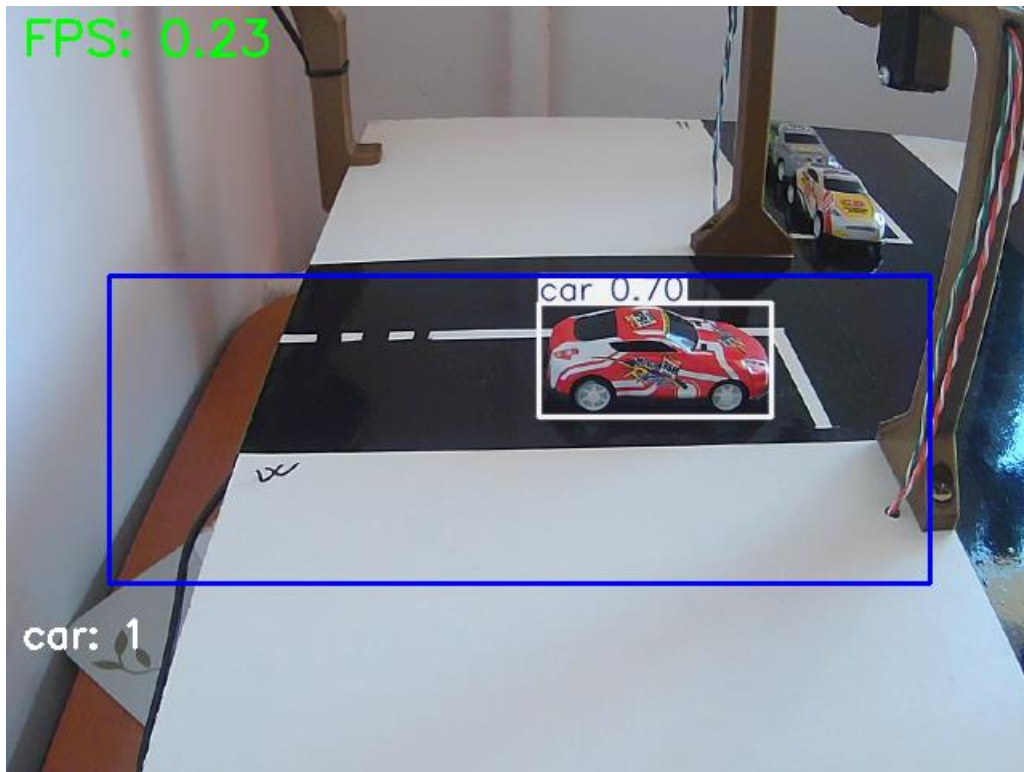


Figure 65. One car detected

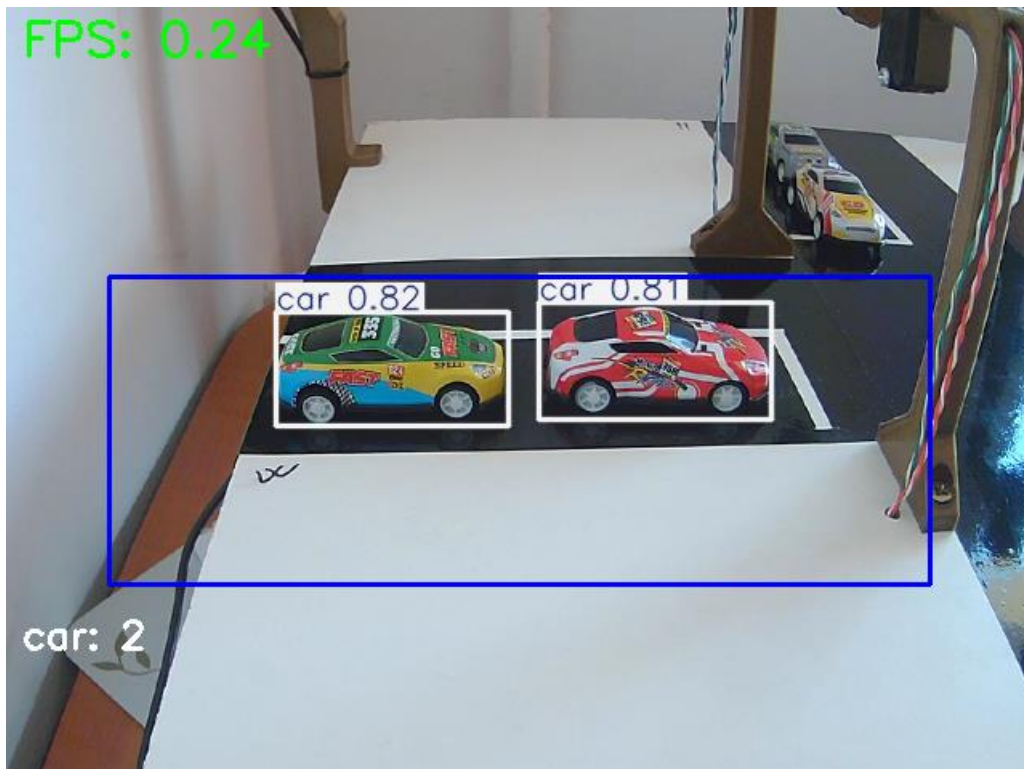


Figure 66. Two cars detected

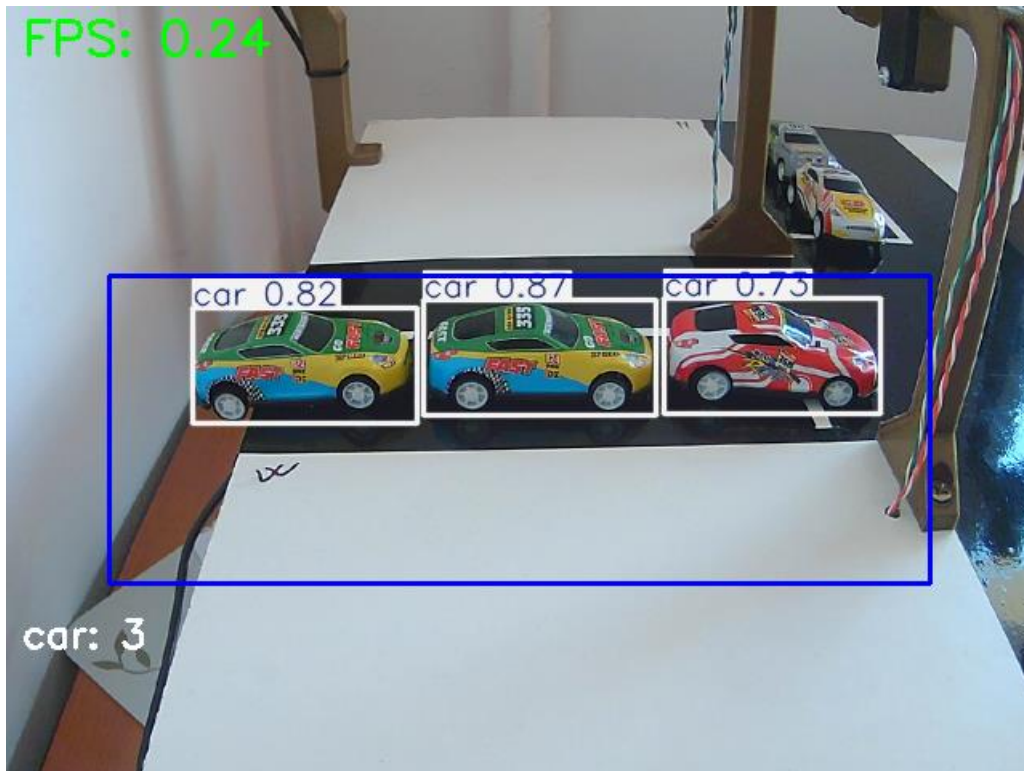


Figure 67. Three cars detected

6.3 Evaluation of the Decision Tree Model

To evaluate the machine learning model used (Decision Tree), it is necessary to highlight several important performance metrics useful for analyzing the accuracy, robustness, and efficiency of the predictions. These metrics were implemented and visualized using dedicated scripts, providing an overview of the model's ability to classify traffic light duration (Short, Medium, or Long) based on the number of cars detected. The metrics used for evaluation are: confusion matrix, cross-validation accuracy, feature importance.

6.3.1 Decisional Tree Graphic Representation

An essential aspect of evaluating the performance of the machine learning model is visual analysis of the decision tree, which serves as a graphic representation of the rules and internal logic used to predict the traffic light green durations (Short, Medium, and Long). This visualization is achieved by exporting the tree structure into a graphical format using Graphviz library, allowing for a clearer understanding of how the model makes decisions based on the input data, represented by the number of cars in each direction of the intersection.

```

11
12 dot_data = export_graphviz(
13     clf,
14     out_file=None,
15     feature_names=['cars_south', 'cars_north', 'cars_west', 'cars_east'],
16     class_names=["Short", "Medium", "Long"],
17     filled=True,
18     rounded=True,
19     special_characters=True
20 )
21
22 graph = graphviz.Source(dot_data)
23

```

Figure 68. Script Used for Plotting the Tree

Figure 69. Decision Tree

The evaluation method for the tree can be divided into the analysis of three main components: the Root Node and the two branches originated from it (True and False Branch).

The root node (Figure 57.) expresses the first branching condition of the model ($\text{cars_south} \leq 0.5$). This condition divides the data into two subsets: the true branch, if the condition is met, and the false branch otherwise. The node representation also displays certain indicators: Gini = 0.667 represents the maximum possible Gini coefficient value in a three-class system, signaling that the model cannot make an informed decision at this point. Samples = 240 represents the total number of observations in the root node, and Value = [80,80,80] indicates that each class has an equal number of samples, meaning that in the absence of a dominant class, an arbitrary one (Short) will be chosen.

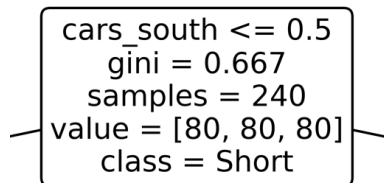


Figure 70. Root Class (Yellow Outline in Figure 56.)

The True Branch ($\text{cars_south} \leq 0.5$) manages cases where traffic from the south is low. The model analyzes the remaining directions to determine the traffic light duration. As can be observed in Figure 58., the dominant predictions are “Short” and “Medium”, as lower traffic across multiple directions justifies a shorter green time. The impurity consistently decreases through additional conditions, demonstrating that the model learns clear rules in this context. Furthermore, confusion between classes appears in only a few nodes, but future branching resolves this issue.

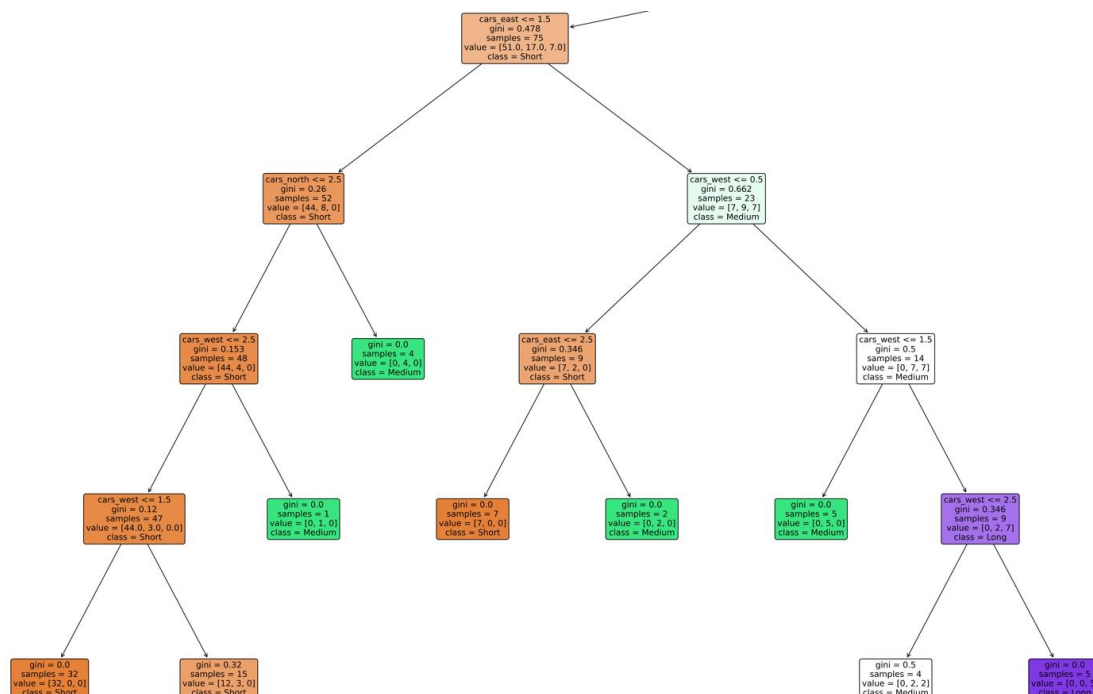


Figure 71. True Branch (Red Outline in Figure 56.)

The False Branch handles cases where there is more intense traffic from the South, which may justify longer traffic light durations. In this scenario, predictions tend toward “Medium” and “Long” classes, reflecting a necessity for a greater green light time. A notable diversity of decisions appears, with branches where classification is balanced or even ambiguous.

This leads to additional branching to help clarify the final decision (Figure 59.):

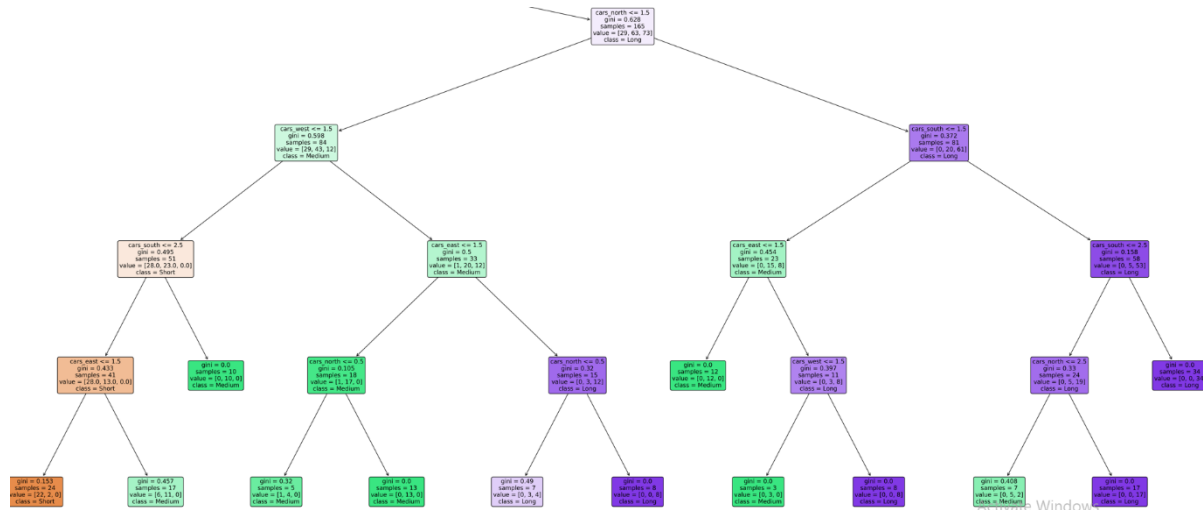


Figure 72. False Branch (Blue Outline Figure 57.)

6.3.2 Confusion Matrix

To evaluate the model accuracy as well as the possible types of errors, the confusion matrix is an essential tool. Its structure is as follows: the rows represent the actual number of observations for each class, the columns represent the number of observations predicted by the model, the main diagonal shows the number of correct predictions (True Positives) for each class, and the elements outside of this diagonal indicate the number of misclassifications.

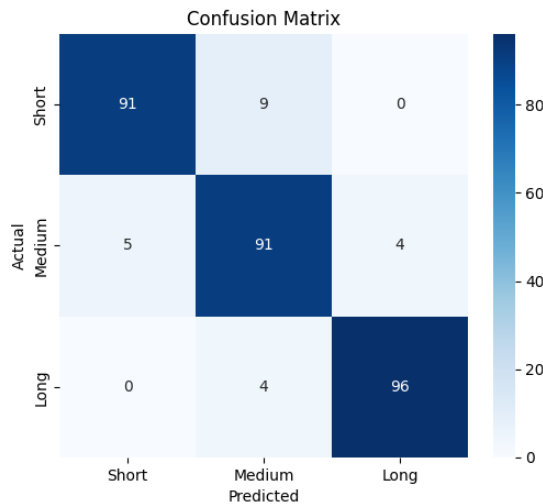


Figure 73. Confusion Matrix

Analyzing Figure 60., several conclusions can be drawn. Given the large number on the main diagonal, the model correctly classifies most observations. The number of errors is small compared to the correct predictions, suggesting a high efficiency of the model. Slight tendencies can be observed to confuse “Short” with “Medium” (9 cases) and “Medium” to “Short” or “Long” (5 and 4 cases), and “Long” is occasionally confused with “Medium” (4 cases). These errors most likely occur when traffic is on the boundary between these classes. An important aspect is that the model never confuses “Short” with “Long” or reciprocally (0 cases), which is an excellent aspect, as such errors would have a big negative impact on traffic flow.

6.3.3 Cross-Validation Score (5-Fold) vs Training Accuracy

To evaluate the model's ability to generalize to new data, a technique called Cross-Validation was used. The 5-Fold method involves splitting the dataset into five equal subsets, with the model being trained five times; each time using a different subset for testing and the remaining four are used for training.

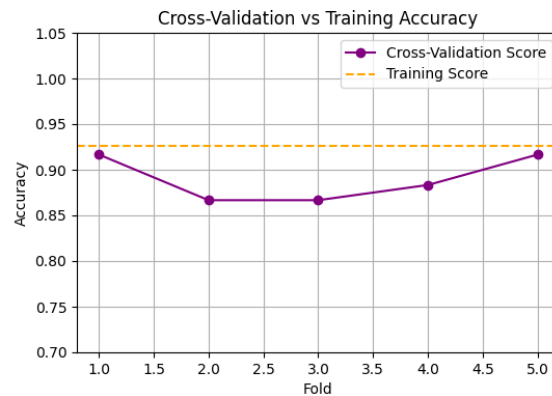


Figure 74. Cross-Validation Score

As can be observed in Figure 61., a graph displays cross-validation score alongside training score. Cross-validation accuracy across the five subsets remains consistent, indicating that the model is robust. Training accuracy appears very high, with cross-validation accuracy generally aligning closely with it, suggesting no overfitting and efficient generalization on new data. Average accuracy performance across the five folds reaches approximately 89%, confirming the model's capability to deliver precise predictions under dynamic conditions.

6.3.4 Feature Importance

Another important metric for evaluating the model is Feature Importance. This measures the contribution of each input variable to the model's predictions, indicating how each direction (cars number) influences the decisions regarding the green light duration.

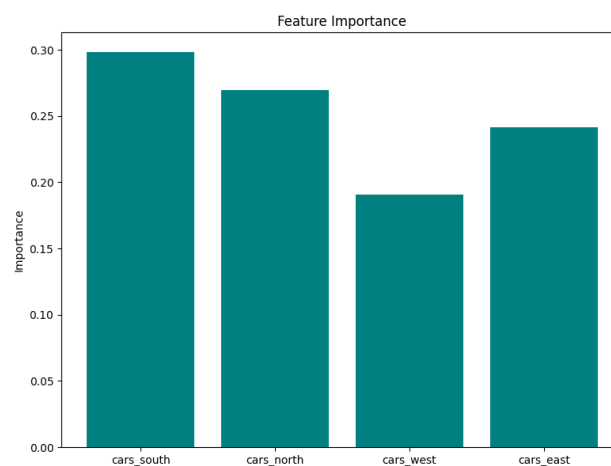


Figure 75. Feature Importance

As shown in the graph above, the model relies more on traffic from North-South axis when making decisions. This indicates that, in the training dataset, traffic on these directions is more intense, requiring greater attention. Consequently, the model prioritizes information from the most relevant directions to adapt the traffic light durations accordingly.

6.4 System Functionality in Various Scenarios

6.4.1 Scenario 1: Equal Number of Cars on Both Axes (NS = EW)

In this scenario, an equal number of cars is detected on both main axes of the intersection, North-South (NS) and East-West (EW). Since there is no dominant direction, the system will prioritize the NS direction due to its greater importance in the training dataset (Figure 62.). Then, based on the number of cars detected, the model will calculate the necessary green light duration. For example, the test was done with three cars placed on both NS and EW directions. After detecting the number of cars, and selecting the green light for the NS direction, the model calculated a duration of 6 seconds (“Medium”) (Figure 64.).

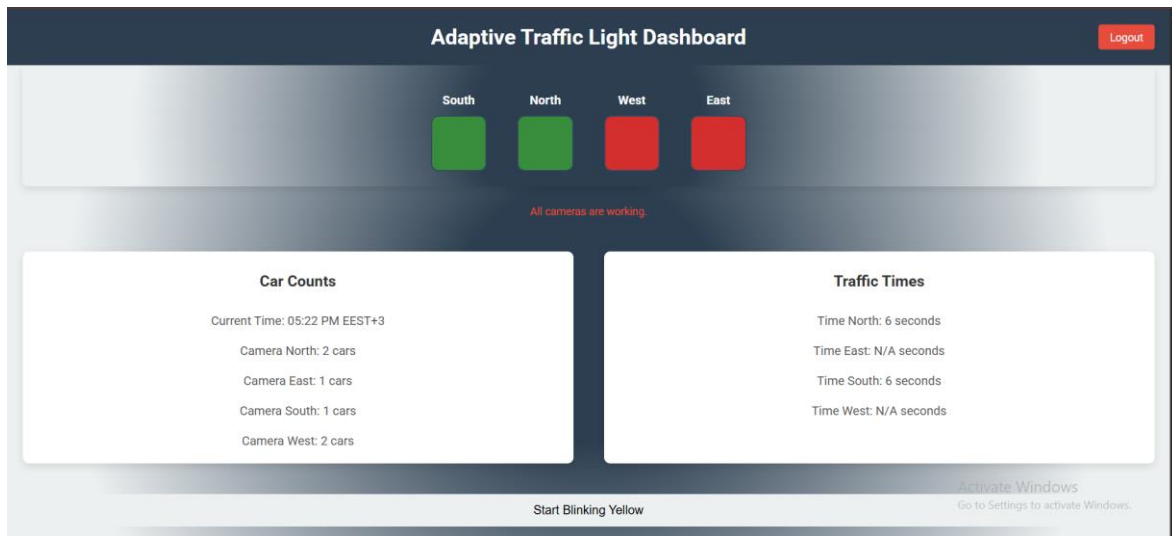


Figure 76. Dashboard in Scenario 1

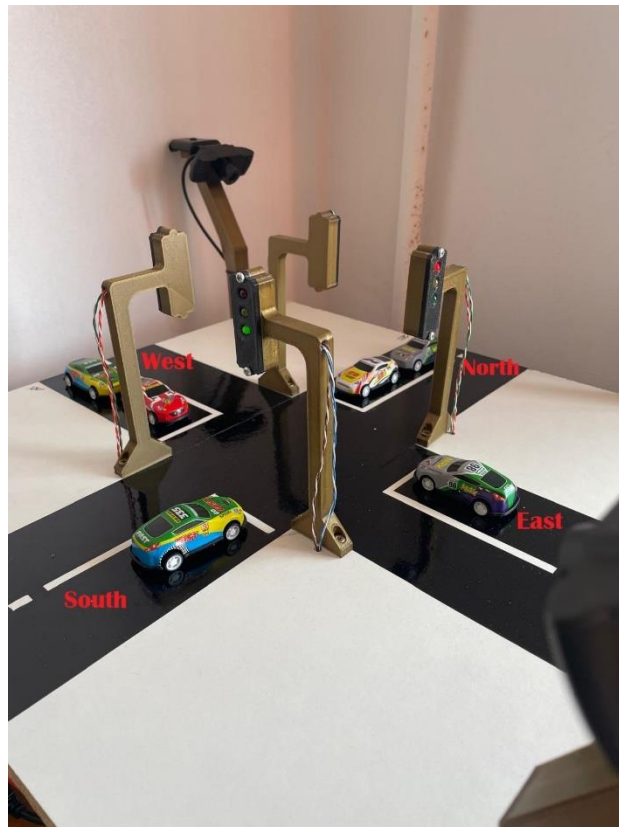


Figure 77. Intersection Model in Scenario 1

6.4.2 Scenario 2: Heavier traffic on East-West Axis (EW > SN)

In this case where a higher number of cars is detected on the East-West axis compared to the North-South, the system will consider EW as dominant, thus giving it priority. After this decision, the model will determine the necessary green time based on the total number of cars in EW direction. This can be seen in the test conducted below (Figure 66), where there are four cars on the EW direction and only two on the NS direction. Given this high number of cars on EW, the model decided on a duration of 9 seconds (“Long”).



Figure 78. Dashboard in Scenario 2



Figure 79. Intersection Model in Scenario 2

6.4.3 Scenario 3: Heavier traffic on South-North Axis (SN > EW)

If a higher number of cars is detected in the South-North direction compared to the East-West direction, the system will consider NS as dominant and will prioritize it. Subsequently, the model will determine the necessary green light duration based on the total car number on the NS axis. A relevant example was tested where the North-South direction has one car, and the East-West direction has none. Given the low number of cars in the intersection, the model allocated a short duration of 3 seconds for green light.



Figure 80. Dashboard in Scenario 3

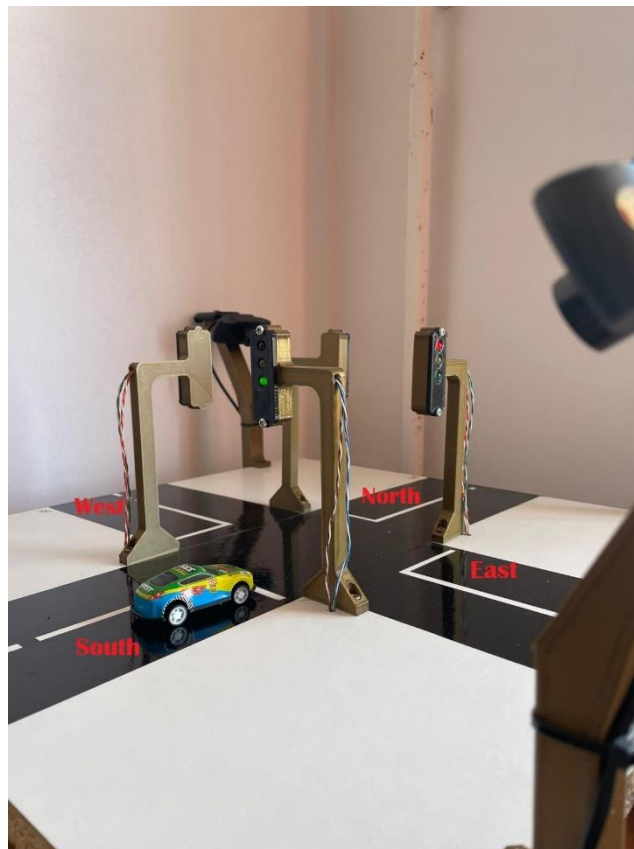


Figure 81. Intersection Model in Scenario 3

6.4.4 Scenario 4: Manually set Emergency Lights (Blinking Yellow)

In cases where the system needs to be stopped for various reasons, such as emergency or maintenance, to avoid disrupting traffic in the intersection or causing accidents due to potentially incorrect traffic light signals, a “Start Blinking Yellow” button was implemented in the Dashboard. When this button is activated, the yellow blinking mode should start both in the dashboard interface and on the physical traffic lights in the intersection. The functionality of this feature can be observed in the figures below.

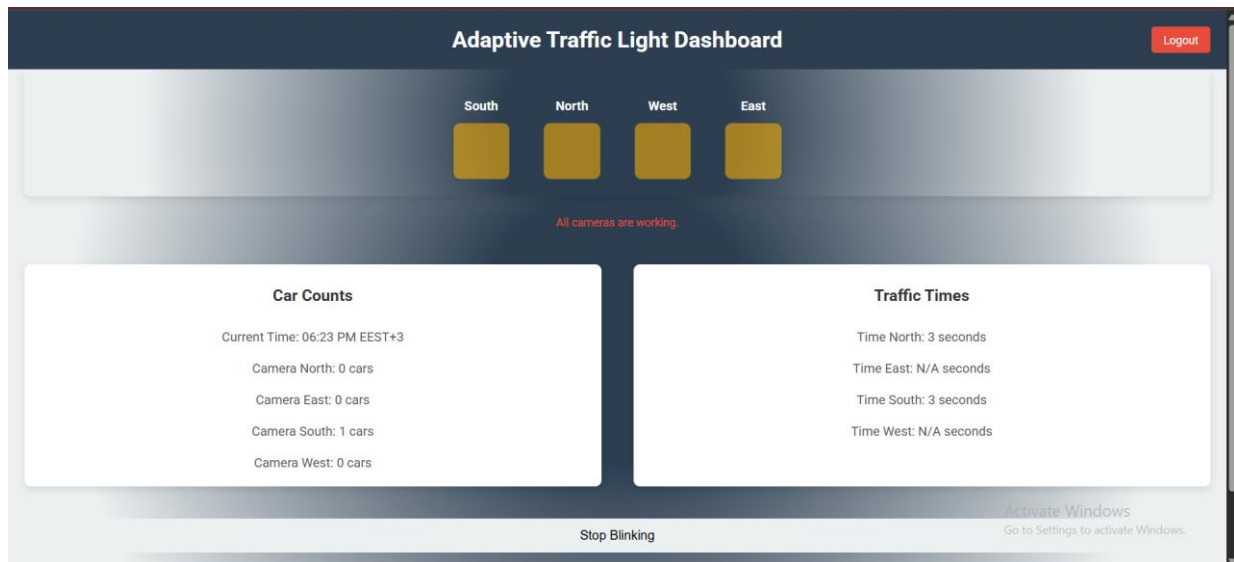


Figure 82. Start Blinking Yellow functionality in Dashboard

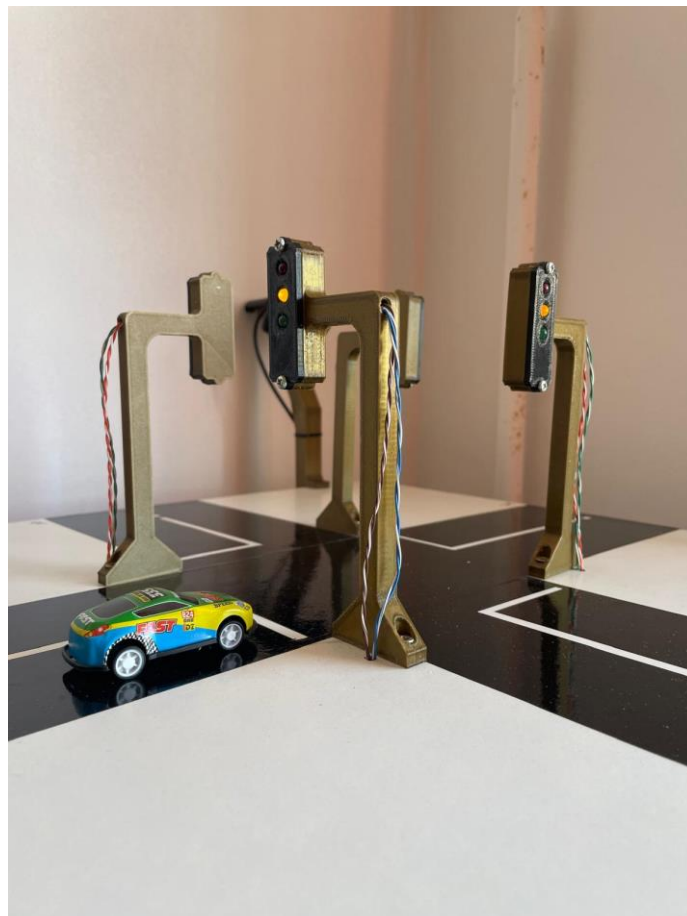


Figure 83. Blinking Yellow in the Intersection Model

6.4.5 Scenario 5: Camera Error

In the unfortunate scenario where one or more cameras stop working, to avoid any potential issues in the intersection, the system has built-in feature that automatically switches the traffic lights in all directions to emergency mode (Blinking Yellow). Additionally, an error message will appear on the dashboard, indicating the last camera where the error occurred. The functionality of this safety feature can be seen in the following figures.

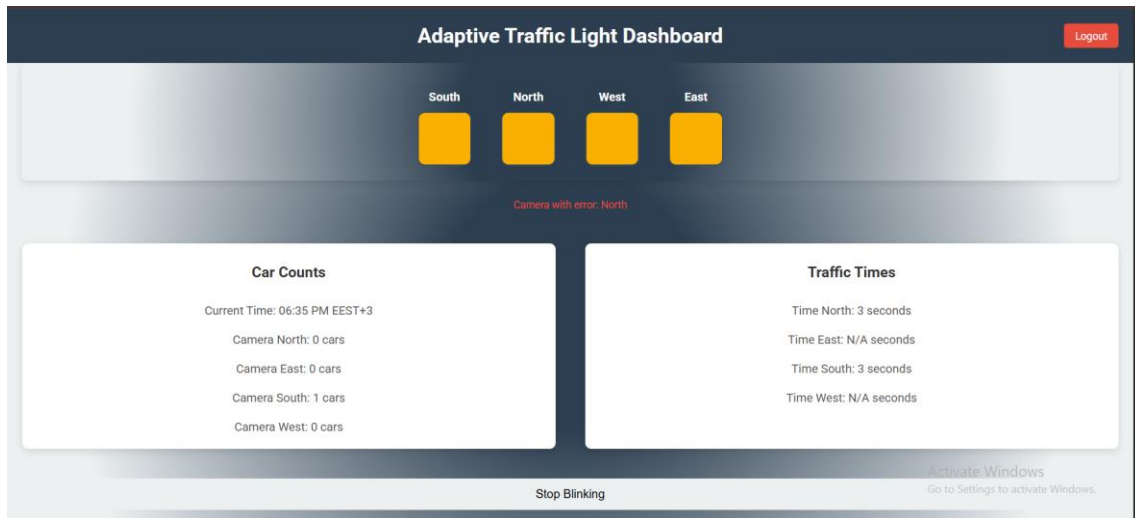


Figure 84. Error on North Camera in Dashboard

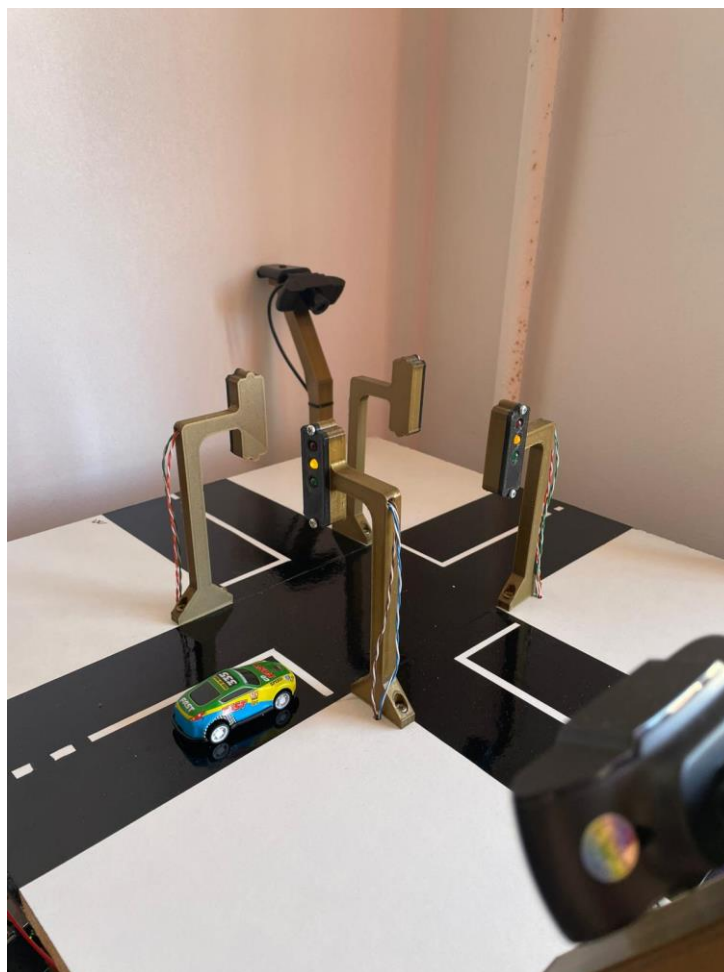


Figure 85. Blinking Yellow in the Intersection Model in Case of Camera Error

7 Conclusions

This project successfully explored and implemented an adaptive traffic light system designed to optimize traffic flow at an intersection, utilizing advanced machine learning and accessible hardware. The system incorporates the YOLOv11 model for vehicle detection, along with a decision tree-based machine learning model for predicting traffic light timings. Physical control of traffic lights is managed through Raspberry Pi GPIO using traffic light modules. This setup enables dynamic adjustment of traffic phases based on real-time conditions.

Experimental validation, conducted on a physical model and through various simulated scenarios (balanced traffic, asymmetric traffic, and camera errors), confirmed the proposed solution's robustness and efficiency. Machine learning metrics demonstrated high performance, including a cross-validation accuracy of approximately 89%. The confusion matrix showed predominantly correct predictions, and feature importance analysis prioritized the North-South axis. An integrated Flask application provides efficient monitoring capabilities. Additionally, camera error management and being able to start the "emergency" mode ensures a safe system operation.

The writing and development of this paper were supported using artificial intelligence tools, such as: ChatGPT and Gemini. Those tools were used for rephrasing parts of the text to make it clearer and precise, also helped translate important information accurately. This support played an important role in improving the overall clarity and coherence of the final document.

The implementation highlighted significant advantages, such as adaptability to variable traffic and the potential reduction of congestion. However, it also reveals some limitations, including its reliance on correctly functioning cameras and the prediction update interval. Future improvements involve optimizing the machine-learning model by training it on more diverse datasets, integrating additional sensors for more robust detection, and even expanding the system to multiple interconnected intersections.

8 References

- [1] INRIX, "Global Traffic Scorecard 2024", <https://inrix.com/scorecard>
- [2] TomTom Traffic Index 2023, "Bucharest – congestion levels", <https://www.tomtom.com>
- [3] Idling Action Research - Review of Emissions Data, Tim Barlow and Olivia Cairns <https://doi.org/10.58446/csjk8557>
- [4] H. K. Kim et al., "The Environmental Benefits of an Automatic Idling Control System in Urban Traffic", Scientific Reports, vol. 14, article no. 1032, 2024.
- [5] Carr Dave. "SCOOT: Basic Principles", Institute of Highway Engineers.
- [6] ["Urban traffic control systems: Evidence on performance"](#). Institute for Transport Studies, University of Leeds.
- [7] Xiao-Feng Xie, S. Smith, G. Barlow. [Smart and Scalable Urban Signal Networks: Methods and Systems for Adaptive Traffic Signal Control](#)
- [8] Stephen F. Smith, Gregory J. Barlow, Xiao-Feng Xie, Zachary B. Rubinstein. [Smart urban signal networks: Initial application of the SURTRAC adaptive traffic signal control system](#). International Conference on Automated Planning and Scheduling (ICAPS). Rome, Italy, 2013.
- [9] (2022) NSW Government website, https://www.transport.nsw.gov.au/system/files/media/documents/2022/SCATS-Core-brochure-Final-web-spreads_0.pdf
- [10] Exploring the Advantages and Disadvantages of Intelligent Traffic Systems, Eastgate Software website, <https://eastgate-software.com/exploring-the-advantages-and-disadvantages-of-intelligent-traffic-systems/>
- [11] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 779-788, doi: 10.1109/CVPR.2016.91.
- [12] Tencent, "ncnn: High-performance neural network inference framework", GitHub, 2022.
- [13] Al Rabbani Alif, Mujadded. (2024). YOLOv11 for Vehicle Detection: Advancements, Performance, and Applications in Intelligent Transportation Systems. 10.48550/arXiv.2410.22898.
- [14] Breiman, L., Friedman, J., Olshen, R. and Stone, C. (1984) Classification and Regression Trees. Chapman and Hall, Wadsworth, New York.
- [15] Zadeh, L. A., "Fuzzy sets", Information and Control, vol. 8, no. 3, pp. 338–353, 1965.
- [16] W. Zhao et al., "Vision-Based Adaptive Traffic Signal Control Using CNNs", Applied Sciences, vol. 10, no. 7, 2020.

- [17] N, Rithesh & R., Vignesh & M R, Anala. (2018). Autonomous Traffic Signal Control using Decision Tree. International Journal of Electrical and Computer Engineering (IJECE). 8. 1522. 10.11591/ijece.v8i3.pp1522-1529.
- [18] Ajayi, Olasupo & Bagula, Antoine & Isafiade, Omowunmi & Noutouglo, Ayodele. (2019). Effective Management of Delays at Road Intersections using Smart Traffic Light System.
- [19] (2024) Official Raspberry Pi website, <https://www.raspberrypi.com/news/raspberry-pi-product-series-explained/>
- [20] Official Raspberry Pi website, Raspberry Pi 4 Tech Specs, <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/>
- [21] electronics-lab.com website, Raspberry Pi 4 – A Look Under the Hood and How to Make most of it, <https://www.electronics-lab.com/project/raspberry-pi-4-look-hood-make/>
- [22] “LED Traffic Light Module datasheet”, https://cdn.shopify.com/s/files/1/1509/1638/files/LED_Ampel_Modul_Datenblatt_AZ-Delivery_Vertriebs_GmbH.pdf?v=1607630369
- [23] Official Wansview website, https://www.wansview.com/webcam_101
- [24] Official Hama Website, <https://ae.hama.com/00200122/hama-usb-hub-4-ports-usb-2-0-480-mbit-s-incl-cable-and-power-supply-unit>
- [25] Python Official Documentation, <https://docs.python.org/>
- [26] Python Logo, <https://www.python.org/community/logos/>
- [27] Ultralytics YOLO Docs, <http://docs.ultralytics.com/>
- [28] OpenCV Documentation, <https://docs.opencv.org/4.x/index.html>
- [29] RPi.GPIO Documentation, <https://sourceforge.net/p/raspberry-gpio-python/wiki/Home/>
- [30] Flask Documentation, <https://flask.palletsprojects.com/en/stable/>
- [31] Terven, J., Córdova-Esparza, D. M., & Romero-González, J. A. (2023). A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas. *Machine learning and knowledge extraction*, 5(4), 1680-1716.
- [32] LeCun, Yann; Bengio, Yoshua; Hinton, Geoffrey (2015). "Deep Learning" (PDF). *Nature*. 521 (7553): 436–444. Bibcode:2015Natur.521..436L. doi:10.1038/nature14539. PMID 26017442. S2CID 3074096.
- [33] Cortes, C., Vapnik, V. Support-vector networks. *Mach Learn* 20, 273–297 (1995). <https://doi.org/10.1007/BF00994018>
- [34] Official IBM Website, <https://www.ibm.com/think/topics/logistic-regression>

- [35] Scikit-learn Documentation, "Decision Trees", <https://scikit-learn.org/stable/modules/tree.html>
- [36] Hastie, Trevor & Tibshirani, Robert & Friedman, Jerome & Franklin, James. (2004). The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Math. Intell.. 27. 83-85. 10.1007/BF02985802.
- [37] Official IBM Website, <https://www.ibm.com/think/topics/decision-trees>

9 Appendix

Car Detection Code:

```
from ultralytics import YOLO
import cv2
import time
import json
from datetime import datetime

model = YOLO("yolo11n_ncnn_model", task="detect")

CAMERA_PATHS = {
    "North": "/dev/v4l/by-path/platform-fd500000.pcie-pci-0000:01:00.0-usb-0:1.1.1:1.0-
video-index0",
    "East": "/dev/v4l/by-path/platform-fd500000.pcie-pci-0000:01:00.0-usb-0:1.1.2:1.0-video-
index0",
    "South": "/dev/v4l/by-path/platform-fd500000.pcie-pci-0000:01:00.0-usb-0:1.1.3:1.0-
video-index0",
    "West": "/dev/v4l/by-path/platform-fd500000.pcie-pci-0000:01:00.0-usb-0:1.1.4:1.0-
video-index0"
}

SWITCH_INTERVAL = 3 # seconds

car_counts = {name: 0 for name in CAMERA_PATHS}

def is_inside_roi(x, y, w, h, frame_width, frame_height):
    roi_margin_x_left = int(frame_width * 0.01)
    roi_margin_x_right = int(frame_width * 0.10)
    roi_margin_y_top = int(frame_height * 0.35)
    roi_margin_y_bottom = int(frame_height * 0.25)

    x_center = int(x)
    y_center = int(y)

    return (roi_margin_x_left < x_center < frame_width - roi_margin_x_right and
            roi_margin_y_top < y_center < frame_height - roi_margin_y_bottom)

def write_log_file():
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    with open("log.txt", "w") as log_file:
        for name, count in car_counts.items():
            log_file.write(f"Camera {name}: {count} cars - {timestamp}\n")

def update_camera_status(error_name=None):
    status_file = "camera_status.json"
    try:
        with open(status_file, "r") as f:
            status = json.load(f)
    except (FileNotFoundError, json.JSONDecodeError):
```

```

status = {"errors": [], "last_error": None}

if "errors" not in status:
    status["errors"] = []
if "last_error" not in status:
    status["last_error"] = None

all_working = True
current_error = None
for name in CAMERA_PATHS:
    cap = cv2.VideoCapture(CAMERA_PATHS[name])
    if not cap.isOpened():
        all_working = False
        current_error = name
    cap.release()

if not all_working and current_error:
    if current_error not in status["errors"]:
        status["errors"].append(current_error)
    status["last_error"] = current_error
elif all_working:
    status["errors"] = []
    status["last_error"] = None

status["blink"] = bool(status["errors"])
status["message"] = f"Camera with error: {status['last_error']}" if status["last_error"] else
"All cameras are working."
status["error"] = bool(status["last_error"])

with open(status_file, "w") as f:
    json.dump(status, f)

def monitor_cameras():
    while True:
        for name, path in CAMERA_PATHS.items():
            cap = cv2.VideoCapture(path)
            if not cap.isOpened():
                print(f"Camera {name} failed to open.")
                update_camera_status(name)
                continue

            success, frame = cap.read()
            cap.release()
            if not success:
                print(f"Camera {name} failed to read.")
                continue

            results = model.predict(frame, conf=0.5)
            count = 0

            if results[0].boxes and results[0].boxes.cls is not None:
                boxes = results[0].boxes.xywh.cpu()

```



```

class_ids = results[0].boxes.cls.int().cpu().tolist()
names_list = results[0].names

for box, cls_id in zip(boxes, class_ids):
    x, y, w, h = box
    class_name = names_list[cls_id]
    if class_name == "car":
        if is_inside_roi(x, y, w, h, frame.shape[1], frame.shape[0]):
            count += 1

car_counts[name] = count
write_log_file()
print(f'Updated Camera {name}: {count} cars')
update_camera_status()

time.sleep(SWITCH_INTERVAL)

if __name__ == '__main__':
    monitor_cameras()

```

Generate Synthetic Dataset Code:

```

import pandas as pd
import random
import os
from collections import defaultdict

SAMPLES_PER_BUCKET_PER_PHASE = 50 # 50 for each of 3 buckets, for both SN and
EW
MAX_CARS = 3
random.seed(42)

data = []
bucket_counts = {
    'SN': defaultdict(int),
    'EW': defaultdict(int)
}

tie_counter = 0

def assign_bucket(total_cars):
    if total_cars <= 2:
        return 0 # Short
    elif total_cars <= 4:
        return 1 # Medium
    else:
        return 2 # Long

while any(bucket_counts[phase][bucket] < SAMPLES_PER_BUCKET_PER_PHASE
          for phase in ['SN', 'EW'] for bucket in range(3)):

```

```

cars_south = random.randint(0, MAX_CARS)
cars_north = random.randint(0, MAX_CARS)
cars_west = random.randint(0, MAX_CARS)
cars_east = random.randint(0, MAX_CARS)

sn_total = cars_south + cars_north
ew_total = cars_east + cars_west

if sn_total > ew_total:
    light_phase = 'SN'
    active_cars = sn_total
elif ew_total > sn_total:
    light_phase = 'EW'
    active_cars = ew_total
else:
    light_phase = 'SN' if tie_counter % 2 == 0 else 'EW'
    active_cars = sn_total
    tie_counter += 1

green_time_bucket = assign_bucket(active_cars)

if bucket_counts[light_phase][green_time_bucket] <
SAMPLES_PER_BUCKET_PER_PHASE:
    data.append({
        'cars_south': cars_south,
        'cars_north': cars_north,
        'cars_west': cars_west,
        'cars_east': cars_east,
        'light_phase': light_phase,
        'green_time_bucket': green_time_bucket
    })
    bucket_counts[light_phase][green_time_bucket] += 1

df = pd.DataFrame(data)

print("Dataset size:", len(df))
print("\nGreen phase distribution:")
print(df['light_phase'].value_counts())
print("\nGreen time bucket distribution per phase:")
print(df.groupby(['light_phase', 'green_time_bucket']).size())

output_path =
r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\data\traffic_timing_balanced.csv"
df.to_csv(output_path, index=False)
print(f"\nSaved to: {output_path}")

```

Train Decision Tree Code:

```
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import joblib
import os

BASE_DIR = r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML"
DATA_PATH = os.path.join(BASE_DIR, "data", "traffic_timing_balanced.csv")
MODEL_PATH = os.path.join(BASE_DIR, "models", "decision_tree_model.pkl")
PLOT_PATH = os.path.join(BASE_DIR, "outputs", "decision_tree_plot.png")

df = pd.read_csv(DATA_PATH)

X = df[['cars_south', 'cars_north', 'cars_west', 'cars_east']]
y = df['green_time_bucket']

X_train, X_test, y_train, y_test = train_test_split(
    X, y, stratify=y, test_size=0.2, random_state=42
)

clf = DecisionTreeClassifier(max_depth=5, random_state=42)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
print("==== Classification Report ====")
print(classification_report(y_test, y_pred))

print("==== Confusion Matrix ====")
print(confusion_matrix(y_test, y_pred))

scores = cross_val_score(clf, X, y, cv=5)
print("==== Cross-validation ====")
print("Scores:", scores)
print("Mean accuracy: {:.3f} ± {:.3f}".format(scores.mean(), scores.std()))

joblib.dump(clf, MODEL_PATH)
print(f"\nModel saved to: {MODEL_PATH}")

plt.figure(figsize=(10, 6))
plot_tree(clf, feature_names=X.columns, class_names=["Short", "Medium", "Long"],
filled=True)
plt.title("Decision Tree for Green Time Buckets")
plt.savefig(PLOT_PATH)
print(f"Tree plot saved to: {PLOT_PATH}")
```

Plot Confusion Matrix Code:

```
import joblib
import pandas as pd
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

clf =
joblib.load(r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\models\decision_tree_
model.pkl")
df =
pd.read_csv(r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\data\traffic_timing_b
alanced.csv")

X = df[['cars_south', 'cars_north', 'cars_west', 'cars_east']]
y = df['green_time_bucket']
y_pred = clf.predict(X)

cm = confusion_matrix(y, y_pred)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=["Short", "Medium",
"Long"], yticklabels=["Short", "Medium", "Long"])
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix")

output_path =
r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\outputs\confusion_matrix.png"
plt.savefig(output_path)
print(f'Saved to: {output_path}')
```

Plot Feature Importance Code:

```
import joblib
import matplotlib.pyplot as plt

model_path =
r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\models\decision_tree_model.pkl"
clf = joblib.load(model_path)

importances = clf.feature_importances_
features = ['cars_south', 'cars_north', 'cars_west', 'cars_east']
plt.figure(figsize=(8, 6))
plt.bar(features, importances, color='teal')
plt.title("Feature Importance")
plt.ylabel("Importance")
plt.tight_layout()
output_path =
r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\outputs\feature_importance.png"
plt.savefig(output_path)
print(f'Saved to: {output_path}')
```

Plot Cross-Validation Score (5-Fold) Code:

```
import joblib
import pandas as pd
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

clf =
joblib.load(r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\models\decision_tree_
model.pkl")
df =
pd.read_csv(r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\data\traffic_timing_b
alanced.csv")
X = df[['cars_south', 'cars_north', 'cars_west', 'cars_east']]
y = df['green_time_bucket']

scores = cross_val_score(clf, X, y, cv=5)

train_preds = clf.predict(X)
train_score = accuracy_score(y, train_preds)

plt.figure(figsize=(6, 4))
plt.plot(range(1, 6), scores, marker='o', color='purple', label='Cross-Validation Score')
plt.axhline(train_score, color='orange', linestyle='--', label='Training Score')
plt.ylim(0.7, 1.05)
plt.title("Cross-Validation vs Training Accuracy")
plt.xlabel("Fold")
plt.ylabel("Accuracy")
plt.legend()
plt.grid(True)

output_path =
r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\outputs\crossvalvstrain_scores.pn
g"
plt.savefig(output_path)
print(f'Saved to: {output_path}')
```

Plot Decision Tree:

```
import joblib
from sklearn.tree import export_graphviz
import graphviz
import os

model_path =
r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\models\decision_tree_model.pkl"
clf = joblib.load(model_path)

dot_data = export_graphviz(
    clf,
    out_file=None,
```

```

feature_names=['cars_south', 'cars_north', 'cars_west', 'cars_east'],
class_names=["Short", "Medium", "Long"],
filled=True,
rounded=True,
special_characters=True
)

graph = graphviz.Source(dot_data)

output_dir = r"C:\Users\ciope\Desktop\Licenta\prototypes\Train_times_ML\outputs"
graph.render(filename="decision_tree_graphviz", directory=output_dir, format="pdf",
cleanup=True)
print(f'Saved Graphviz tree to: {os.path.join(output_dir, 'decision_tree_graphviz.pdf')}')

```

ML model implemented with Traffic Light Modules:

```

import RPi.GPIO as GPIO
import json
import time
import threading
import joblib

TRAFFIC_LIGHTS = {
    "West": {"red": 2, "yellow": 3, "green": 4},
    "South": {"red": 5, "yellow": 6, "green": 7},
    "East": {"red": 8, "yellow": 9, "green": 10},
    "North": {"red": 12, "yellow": 11, "green": 13},
}

MODEL_PATH = "/home/ciopec/yolo_try2/decision_tree_model.pkl"
LIGHT_STATE_FILE = "light_state.json"
LOG_FILE = "log.txt"
BLINK_FILE = "blink_mode.json"
TIMES_FILE = "traffic_times.json"
CAMERA_STATUS_FILE = "camera_status.json"

YELLOW_TIME = 4.0
BLINK_COUNT = 3
BLINK_ON_TIME = 0.5
BLINK_OFF_TIME = 0.5

model = joblib.load(MODEL_PATH)
current_duration = 3
last_green_dirs = None

def get_blink_state():
    try:
        with open(BLINK_FILE, "r") as f:
            return json.load(f).get("blink", False)
    except:
        return False

```

```

def get_camera_blink_state():
    try:
        with open(CAMERA_STATUS_FILE, "r") as f:
            return json.load(f).get("blink", False)
    except:
        return False

def setup():
    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)
    for pins in TRAFFIC_LIGHTS.values():
        for pin in [pins["red"], pins["yellow"], pins["green"]]:
            GPIO.setup(pin, GPIO.OUT)
            GPIO.output(pin, GPIO.LOW)

def set_light(direction, color):
    pins = TRAFFIC_LIGHTS[direction]
    for c in ["red", "yellow", "green"]:
        GPIO.output(pins[c], GPIO.LOW)
    GPIO.output(pins[color], GPIO.HIGH)
    update_state(direction, color)
    print(f"Set {direction} to {color}")
    time.sleep(0.1)

def update_state(name, color):
    try:
        with open(LIGHT_STATE_FILE, "r") as f:
            state = json.load(f)
    except:
        state = {}
    state[name] = color
    with open(LIGHT_STATE_FILE, "w") as f:
        json.dump(state, f)

def all_red(directions):
    for d in directions:
        set_light(d, "red")

def green_solid(directions):
    for d in directions:
        set_light(d, "green")

def yellow_on(directions):
    for d in directions:
        set_light(d, "yellow")

def green_off(directions):
    for d in directions:
        pins = TRAFFIC_LIGHTS[d]
        GPIO.output(pins["green"], GPIO.LOW)
        update_state(d, "off")
        print(f"Green off for {d}")

```



```

time.sleep(0.1)

def blink_thread():
    while True:
        manual_blink = get_blink_state() # Blinking manual
        camera_blink = get_camera_blink_state() # Blinking din erori
        if manual_blink or camera_blink:
            for tl in TRAFFIC_LIGHTS.values():
                GPIO.output(tl["red"], GPIO.LOW)
                GPIO.output(tl["green"], GPIO.LOW)
                GPIO.output(tl["yellow"], GPIO.HIGH)
            time.sleep(BLINK_ON_TIME)
            for tl in TRAFFIC_LIGHTS.values():
                GPIO.output(tl["yellow"], GPIO.LOW)
            time.sleep(BLINK_OFF_TIME)
        else:
            time.sleep(0.1)

def predict_and_update():
    global current_duration, last_green_dirs

    while True:
        if get_blink_state() or get_camera_blink_state(): # Verifică ambele stări pentru a
suspenda predicția
            print("Blink mode active, skipping prediction")
            time.sleep(0.5)
            continue

        try:
            with open(LOG_FILE, "r") as f:
                lines = f.readlines()

            counts = {'North': 0, 'East': 0, 'South': 0, 'West': 0}
            for line in lines:
                if line.startswith("Camera"):
                    parts = line.strip().split()
                    direction = parts[1].strip(":")
                    count = int(parts[2])
                    if direction in counts:
                        counts[direction] = count

            X = [[counts['South'], counts['North'], counts['West'], counts['East']]]
            bucket = model.predict(X)[0]
            bucket_to_seconds = {0: 3, 1: 6, 2: 9}
            total_green_time = bucket_to_seconds.get(bucket, 3)

            sn_total = counts['South'] + counts['North']
            ew_total = counts['East'] + counts['West']

            if sn_total >= ew_total:
                green_dirs = ['South', 'North']
                red_dirs = ['East', 'West']

```

```

else:
    green_dirs = ['East', 'West']
    red_dirs = ['South', 'North']

    print(f'Counts: {counts}, Bucket: {bucket}, Green time: {total_green_time}s, Green:
{green_dirs}, Red: {red_dirs}')

    traffic_times = {
        "total_green_time": total_green_time,
        "green_dirs": green_dirs
    }
    with open(TIMES_FILE, "w") as f:
        json.dump(traffic_times, f)

    if last_green_dirs == green_dirs:
        print(f'Same green directions {green_dirs}, solid green for {total_green_time}s')
        green_solid(green_dirs)
        all_red(red_dirs)
        time.sleep(total_green_time)
    else:
        if last_green_dirs:
            print(f'Blinking green on old green directions {last_green_dirs}
{BLINK_COUNT} times")
            for _ in range(BLINK_COUNT):
                green_off(last_green_dirs)
                time.sleep(BLINK_OFF_TIME)
                green_solid(last_green_dirs)
                time.sleep(BLINK_ON_TIME)

            print(f'Yellow on old green directions {last_green_dirs} for
{YELLOW_TIME}s")
            yellow_on(last_green_dirs)
            all_red(red_dirs)
            time.sleep(YELLOW_TIME)

            print(f'Setting old green directions {last_green_dirs} to red")
            all_red(last_green_dirs)

            time.sleep(0.5)

            all_red(red_dirs)
            print(f'New green directions {green_dirs}, solid green for {total_green_time}s")
            green_solid(green_dirs)
            time.sleep(total_green_time)

            last_green_dirs = green_dirs

except Exception as e:
    print(f'Prediction error: {e}')
    time.sleep(3)

def run():

```

```

setup()
threading.Thread(target=blink_thread, daemon=True).start()
predict_and_update()

if __name__ == "__main__":
    run()

```

Login Page Template:

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Admin Login</title>
    <link
href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;500&display=swap"
rel="stylesheet">
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
    <div class="login-container">
        <h2>Admin Login</h2>
        {% if error %}
            <div class="error-message">
                <p>{{ error }}</p>
            </div>
        {% endif %}
        <form action="{{ url_for('login') }}" method="POST">
            <div class="form-group">
                <label for="username">Username</label>
                <input type="text" id="username" name="username" required>
            </div>
            <div class="form-group">
                <label for="password">Password</label>
                <input type="password" id="password" name="password" required>
            </div>
            <button type="submit" class="btn">Login</button>
        </form>
        <div class="footer">
            <p>© 2025 Traffic Light System. All rights reserved.</p>
        </div>
    </div>
</body>
</html>

```

Dashboard Template:

```
<!DOCTYPE html>
<html>
<head>
  <title>Traffic Light Dashboard</title>
  <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
  <link
href="https://fonts.googleapis.com/css2?family=Roboto:wght@400;500;700&display=swap"
rel="stylesheet">
</head>
<body>
  <header>
    <h1>Adaptive Traffic Light Dashboard</h1>
    <a href="/logout" class="logout-btn">Logout</a>
  </header>
  <div class="traffic-lights-container">
    {% for direction, color in lights.items() %}
    <div class="light">
      <h3>{{ direction }}</h3>
      <div class="block" id="light_{{ direction }}" class="{{ color }}"></div>
    </div>
    {% endfor %}
  </div>
  {% if camera_error %}
  <div class="error-message" style="text-align: center; margin: 10px 0;">
    <p>{{ camera_error }}</p>
  </div>
  {% endif %}
  <div class="info-container">
    <div class="info-panel car-count">
      <h2>Car Counts</h2>
      <p>Current Time: <span id="current-time">07:01 PM EEST</span></p>
      {% for direction in ['North', 'East', 'South', 'West'] %}
      <p id="car_{{ direction }}">Camera {{ direction }}: <span id="car_count_{{
direction }}">{{ car_counts[direction] if direction in car_counts else 0 }}</span> cars</p>
      {% endfor %}
    </div>
    <div class="info-panel traffic-times">
      <h2>Traffic Times</h2>
      {% for direction in ['North', 'East', 'South', 'West'] %}
      <p id="time_{{ direction }}">Time {{ direction }}: <span id="time_value_{{
direction }}">{{ times[direction] if direction in times else 'N/A' }}</span> seconds</p>
      {% endfor %}
    </div>
  </div>
  <form action="/toggle" method="get" class="toggle-form">
    <button type="submit" class="btn">
      {{ "Stop Blinking" if blinking else "Start Blinking Yellow" }}
    </button>
  </form>
  <footer class="footer">
```

<p>© 2025 Traffic Light System. All rights reserved.</p>
</footer>

```
<script>
function updateDashboard() {
  fetch('/update')
    .then(response => response.json())
    .then(data => {
      for (let direction in data.lights) {
        let lightElement = document.getElementById(`light_${direction}`);
        if (lightElement) {
          if (data.blinking) {
            lightElement.className = 'block yellow blink';
          } else {
            lightElement.className = 'block ' + data.lights[direction];
            lightElement.classList.remove('blink');
          }
        }
      }
      for (let direction in data.car_counts) {
        let countElement = document.getElementById(`car_count_${direction}`);
        if (countElement) {
          countElement.textContent = data.car_counts[direction] || 0;
        }
      }
      for (let direction in data.times) {
        let timeElement = document.getElementById(`time_value_${direction}`);
        if (timeElement) {
          timeElement.textContent = data.times[direction] || 'N/A';
        }
      }
      let toggleBtn = document.querySelector('.btn');
      if (toggleBtn) {
        toggleBtn.textContent = data.blinking ? "Stop Blinking" : "Start Blinking
Yellow";
      }
      if (data.camera_error) {
        document.querySelector('.error-message p').textContent = data.camera_error;
        document.querySelector('.error-message').style.display = 'block';
      } else {
        document.querySelector('.error-message').style.display = 'none';
      }
    })
    .catch(error => console.error('Eroare la actualizare:', error));
}

function updateTime() {
  const now = new Date();
  const options = { hour: '2-digit', minute: '2-digit', hour12: true, timeZoneName: 'short'
};
  const timeString = now.toLocaleTimeString('en-US', options).replace('GMT',
'EEST');
```

```

        document.getElementById('current-time').textContent = timeString;
    }

    setInterval(updateDashboard, 2000);
    setInterval(updateTime, 1000);
    updateDashboard();
    updateTime();
</script>
</body>
</html>

```

style.css

```

body {
    font-family: 'Roboto', sans-serif;
    background: radial-gradient(circle, #2C3E50 20%, #ECF0F1 80%);
    margin: 0;
    padding: 0;
    min-height: 100vh;
    display: flex;
    flex-direction: column;
    justify-content: space-between;
}

header {
    background-color: #2C3E50;
    color: white;
    text-align: center;
    padding: 20px 0;
    position: relative;
    width: 100%;
}

header h1 {
    margin: 0;
    font-size: 28px;
    font-weight: 700;
}

.logout-btn {
    position: absolute;
    right: 20px;
    top: 50%;
    transform: translateY(-50%);
    padding: 8px 15px;
    background-color: #e74c3c;
    color: white;
    text-decoration: none;
    border-radius: 4px;
    font-size: 14px;
}

```

```

.logout-btn:hover {
  background-color: #c0392b;
}

footer {
  text-align: center;
  padding: 20px 0;
  color: white;
  background-color: #2C3E50;
}

.login-container {
  background: #fff;
  padding: 40px;
  border-radius: 8px;
  box-shadow: 0 4px 10px rgba(0, 0, 0, 0.1);
  width: 100%;
  max-width: 400px;
  margin: 20px auto;
}

.login-container h2 {
  text-align: center;
  color: #333;
  margin-bottom: 20px;
}

.form-group {
  margin-bottom: 20px;
}

label {
  display: block;
  font-weight: 500;
  margin-bottom: 8px;
  color: #333;
}

input[type="text"], input[type="password"] {
  width: 100%;
  padding: 10px;
  border: 1px solid #D0E6F0;
  border-radius: 4px;
  box-sizing: border-box;
  font-size: 14px;
}

input[type="text"]:focus, input[type="password"]:focus {
  border-color: #2C3E50;
  outline: none;
}

```

```

.btn {
  width: 100%;
  padding: 12px;
  background-color: #ECF0F1;
  border: none;
  color: #000;
  font-size: 16px;
  border-radius: 4px;
  cursor: pointer;
  transition: background-color 0.3s ease;
}

.btn:hover {
  background-color: #4C5D70;
  color: #D0E6F0;
}

.error-message {
  color: #e74a3b;
  font-size: 14px;
  margin-top: 10px;
  text-align: center;
}

.traffic-lights-container {
  display: flex;
  justify-content: center;
  padding: 20px;
  background: radial-gradient(circle, #2C3E50 20%, #ECF0F1 80%);
  margin: 0 20px;
  border-radius: 8px;
  box-shadow: 0 4px 10px rgba(0, 0, 0, 0.1);
}

.light {
  margin: 0 20px;
  text-align: center;
}

.light h3 {
  margin-bottom: 10px;
  font-size: 16px;
  font-weight: 800;
  color: #fff;
}

.block {
  width: 70px;
  height: 70px;
  margin: 0 auto;
  border-radius: 10px;
  border: 2px solid #34495e;
}

```



```

    transition: opacity 0.5s;
}

.block.red {
    background: #d32f2f;
}

.block.green {
    background: #388e3c;
}

.block.yellow {
    background: #ffb300;
}

/* Efect de clipire */
.block.blink {
    animation: blink 1s infinite;
}

@keyframes blink {
    50% {
        opacity: 0;
    }
}

.info-container {
    display: flex;
    justify-content: center;
    padding: 20px;
    max-width: 1200px;
    margin: 0 auto 20px;
    gap: 40px;
}

.info-panel {
    flex: 1;
    min-width: 700px;
    background: #fff;
    padding: 25px;
    border-radius: 8px;
    box-shadow: 0 4px 10px rgba(0, 0, 0, 0.1);
}

.info-panel h2 {
    text-align: center;
    color: #333;
    margin-bottom: 20px;
    font-size: 20px;
    font-weight: 700;
}

```

```

.info-panel p {
  margin: 5px 0;
  font-size: 16px;
  color: #555;
}

.traffic-times, .car-count{
  display: flex;
  flex-direction: column;
  align-items: center;
  padding: 10px;
}

.time{
  font-size: 18px;
  font-weight: 600;
  color: #ECF0F1;
  text-align: center;
}

.car-count p, .traffic-times p {
  margin: 10px 0;
}

.toggle-form {
  text-align: center;
  margin-bottom: 20px;
}

.toggle-btn {
  padding: 12px 25px;
  font-size: 16px;
  background-color: #ECF0F1;
  border: none;
  color: #000;
  border-radius: 4px;
  cursor: pointer;
  transition: background-color 0.3s ease;
  font-family: 'Roboto', sans-serif;
  font-weight: 500;
}

.toggle-btn:hover {
  background-color: #4C5D70;
  color: #D0E6F0;
}

@media (max-width: 768px) {
  .traffic-lights-container {
    flex-direction: column;
    align-items: center;
    padding: 10px;
  }
}

```

```

    }
    .light {
        margin: 10px 0;
    }
    .info-container {
        flex-direction: column;
        padding: 10px;
    }
    .info-panel {
        width: 100%;
        margin-bottom: 20px;
    }
    .login-container {
        width: 90%;
        margin: 0 auto;
    }
}

```

Flask Application Code:

```

from flask import Flask, render_template, redirect, jsonify, request, session, flash
import json
import os

```

```

app = Flask(__name__)
STATE_FILE = "light_state.json"
BLINK_FILE = "blink_mode.json"
LOG_FILE = "log.txt"
TIMES_FILE = "traffic_times.json"
USERS_FILE = "users.json"
CAMERA_STATUS_FILE = "camera_status.json"

```

```

def get_blink_state():
    try:
        with open(BLINK_FILE, "r") as f:
            return json.load(f).get("blink", False)
    except:
        return False
def set_blink_state(state: bool):
    with open(BLINK_FILE, "w") as f:
        json.dump({"blink": state}, f)
def read_car_counts():
    car_counts = {}
    try:
        with open(LOG_FILE, "r") as f:
            for line in f:
                if line.startswith("Camera"):
                    parts = line.strip().split()
                    if len(parts) >= 3 and parts[2].isdigit():
                        direction = parts[1].strip(":")
                        count = int(parts[2])
                        car_counts[direction] = count

```

```

except:
    pass
return car_counts

def read_traffic_times():
    times = {}
    try:
        with open(TIMES_FILE, "r") as f:
            data = json.load(f)
            total_green_time = data.get("total_green_time", 3)
            green_dirs = data.get("green_dirs", [])
            for direction in TRAFFIC_LIGHTS.keys():
                times[direction] = total_green_time if direction in green_dirs else 0
    except:
        for direction in TRAFFIC_LIGHTS.keys():
            times[direction] = 0
    return times

def get_users():
    try:
        with open USERS_FILE, "r") as f:
            return json.load(f)
    except:
        return {}

def get_camera_status():
    try:
        with open(CAMERA_STATUS_FILE, "r") as f:
            return json.load(f)
    except (FileNotFoundError, json.JSONDecodeError):
        return {"errors": [], "blink": False, "message": "All cameras are working.", "error":
False}

TRAFFIC_LIGHTS = {
    "West": {"red": 2, "yellow": 3, "green": 4},
    "South": {"red": 5, "yellow": 6, "green": 7},
    "East": {"red": 8, "yellow": 9, "green": 10},
    "North": {"red": 12, "yellow": 11, "green": 13},
}
@app.route("/")
def index():
    if not session.get("logged_in"):
        return redirect("/login")
    try:
        with open(STATE_FILE, "r") as f:
            states = json.load(f)
    except:
        states = {}
    car_counts = read_car_counts()
    times = read_traffic_times()
    camera_status = get_camera_status()
    manual_blink = get_blink_state()

```

```

        camera_blink = camera_status.get("blink", False)
        return render_template("index.html", lights=states, blinking>manual_blink or
camera_blink,
                                car_counts=car_counts, times=times,
camera_error=camera_status.get("message"))
@app.route("/toggle")
def toggle():
    if not session.get("logged_in"):
        return redirect("/login")
    current_blink = not get_blink_state()
    set_blink_state(current_blink)
    return redirect("/")
@app.route("/update")
def update():
    if not session.get("logged_in"):
        return jsonify({"error": "Unauthorized"}), 401
    try:
        with open(STATE_FILE, "r") as f:
            states = json.load(f)
    except:
        states = {}
    car_counts = read_car_counts()
    times = read_traffic_times()
    camera_status = get_camera_status()
    manual_blink = get_blink_state()
    camera_blink = camera_status.get("blink", False)
    return jsonify({
        "lights": states,
        "car_counts": car_counts,
        "times": times,
        "blinking": manual_blink or camera_blink,
        "camera_error": camera_status.get("message")
    })
@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]
        users = get_users()
        if users.get(username) == password:
            session["logged_in"] = True
            return redirect("/")
        else:
            flash("Invalid credentials", "error")
            return redirect("/login")
    return render_template("login.html")
@app.route("/logout")
def logout():
    session.pop("logged_in", None)
    return redirect("/login")
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)

```

10 Curriculum Vitae

PERSONAL INFORMATION



Andrei Sorin CIOPEC

📍 Strada Observatorului, nr.107, Cluj-Napoca

☎ 0742 297 557

✉ ciopecandrei2002@gmail.com

🗣 LinkedIn Profile: <http://linkedin.com/in/andrei-sorin-ciopec-8834b3217>

Sex M | Date of birth 08/07/2002 | Nationality Romanian

WORK EXPERIENCE

June 2019

Volunteer during Rosia Montana Marathon

- Guiding and organizing participants on the entire duration of the marathon
- Coordinating the entire team of volunteers, assuring the good organization of the entire event

Rosia Montana, Alba County

July 2022-August 2022

DevOps Evozon Summer Internship 2022

- Learning DevOps practices and tools like: Azure, Docker, GCP, Nginx, Linux/Windows Server, and MySQL
- Create a WordPress basic page and an infrastructure on Azure that contain: two Linux Server VM's that contained Nginx, PHP-FPM, WordPress, Shared Storage
- Migrate the infrastructure from Azure to GCP

Cluj-Napoca, Cluj County

EDUCATION AND TRAINING

October 2021 – present time

Technical University of Cluj-Napoca, Faculty of Electronics, Telecommunications, and Information Technology

4th year student

- Main topic covered: Computer Programming (C/C++, Java), Electronic Devices, Applied Informatics (Microsoft Office), Fundamental Electronic Circuits, Computer Aided Graphics (MATLAB), Computer Aided Design (OrCAD), Language Classes, Web Programming (HTML, CSS/SCSS, JavaScript), Microprocessors Architecture (ASM x86), Microcontrollers (ASM, Embedded C), Systems with digital integrated circuits (Programming FPGA using VHDL), Digital Image Processing(Python).

September 2017– June 2021

National College Avram-Iancu, Câmpeni, MII, Computer Science Section

- Main topic covered: Computer Programming, Mathematics and English language.
- Baccalaureate Degree (Baccalaureate average grade: 8.11)

PERSONAL SKILLS

Native language(s): Romanian

Other language(s):

	UNDERSTANDING		SPEAKING		WRITING
	Listening	Reading	Spoken interaction	Spoken production	
English	C1	C1	B2	C1	C1
French	A1	B1	A1	A1	A2

Levels: A1/2: Basic user - B1/2: Independent user - C1/2 Proficient user
Common European Framework of Reference for Languages

Communication skills:

- particularly good problem-solving skills
- dedicated to providing high-quality work and launching a career built on professionalism and
- enthusiastic and highly motivated to learn

Organisational / managerial skills:

- good decision-making skills, taking initiatives during volunteering projects and organizing teamwork.
- good planning skills acquired while working on team projects.
- time management skills acquired while prioritizing exams and projects during college.

Job-related skills:

- good knowledge of HTML, CSS
- good/basic knowledge of Python and JavaScript
- good knowledge of OOP, data structures and algorithms
- basic familiarity with Angular
- interest in UI/UX, performance optimization, and modern web standards
- enthusiastic, initiative-taking, and curious
- adaptability to new languages, platforms and frameworks
- team orientation and collaboration mindset
- good abstract thinking

Computer skills:

- good command of Microsoft Office™ tools
- basic knowledge of different DevOps tools: Nginx, Docker, Azure, GCP, Linux/Windows Server, acquired when working for Evozon DevOps Internship
- basic/good knowledge of different programming languages: C/C++, PHP, HTML, CSS/SCSS, SQL, Java, MATLAB, VHDL, ASM, Python acquired when working on different personal projects during high-school and college years

Other skills:

- good adaptability in diverse types of environments
- fast learner, eager to develop both on a personal and professional level.
- passionate about technology
- understanding of the latest trends, standards, and developments

Driving licence:

- B

ADDITIONAL INFORMATION

Projects:

- Web Programming Certificate (September 2020-June 2021): Creating of an online car shop using HTML, CSS, PHP, and SQL.
- An infrastructure to deploy a basic WordPress post using Azure/GCP (August 2022): Creating the servers, install all the tools (Nginx, PHP-FPM etc.) and use the SSL Certificates to make the website work only on HTTPS (redirects traffic from HTTP to HTTPS)
- Interactive eye tracker (November 2023- January 2024): Utilized Python libraries such as OpenCV and dlib for facial detection and landmark prediction. Developed algorithms for isolating eye regions and estimating iris positions. Implement some functionalities if the user looks in different directions: open a browser tab, drawing using Turtle module, roll dice game (a basic script in Python)
- Website for my personal portfolio (March 2024-April 2024): Developed a static portfolio website using HTML, CSS, and JavaScript. The site includes three sections: "About Me" with an email link for contact, "My Projects" displaying detailed documentation of previous work, and "My Resume" with an option to download the CV. Designed for smooth navigation and a clean, professional presentation of skills and accomplishments.
- Automated Book Page Cropping (October 2024-December 2024): Developed a Python-based application to enhance scanned book pages by cropping unwanted borders and aligning content. Utilized libraries like OpenCV for edge detection and perspective correction, PyPDF2 for PDF handling, and Tkinter for a user-friendly GUI. The tool improved document readability and presentation by automating the trimming and reassembly of pages into high-quality PDFs.
- Digital Thermometer with Dual Unit Display (December 2024 - January 2025): Developed a digital thermometer using an AVR microcontroller (ATmega328P) and a 16x2 LCD interfaced via I2C. Implemented ADC functionality to read temperature from a 10k NTC thermistor. Designed logic to dynamically display temperature in Celsius or Fahrenheit based on a potentiometer's position. Ensured smooth and accurate temperature updates with two-decimal precision. Utilized low-level programming in Atmel Studio with AVR libraries to control peripherals and optimize system performance.

Competitions:

- Third place in the county competition of information and communication technology.
- First place in the national competition "PC-Între util si plăcut" making a PowerPoint presentation about the impact of recent technologies on health.

