# TRAFFIC MANAGEMENT SYSTEM USING MACHINE LEARNING

**A PROJECT REPORT**

***Submitted by***

| | |
|---|---|
| **MADESH G** | **211420105057** |
| **SANJAI S** | **211420105088** |
| **SANJAY KUMAR M** | **211420105092** |
| **SANJAY RAJAN J** | **211420105093** |

***in partial fulfillment for the award of the degree***

***of***

**BACHELOR OF ENGINEERING**

**in**

**ELECTRICAL AND ELECTRONICS ENGINEERING**

**PANIMALAR ENGINEERING COLLEGE**

**(An Autonomous Institution, Affiliated to Anna University, Chennai)**

**APRIL 2024**

# PANIMALAR ENGINEERING COLLEGE

**(An Autonomous Institution, Affiliated to Anna University, Chennai)**

## BONAFIDE CERTIFICATE

Certified that this project report **"TRAFFIC MANAGEMENT SYSTEM USING MACHINE LEARNING"** is the bonafide work of **"MADESH.G (211420105057), SANJAI.S (211420105088), SANJAY KUMAR.M (211420105092), SANJAY RAJAN J (211420105093)"** who carried out the project work under my supervision.

| | |
|---|---|
| **SIGNATURE** | **SIGNATURE** |
| **Dr. S. SELVI, M.E, Ph.D.** | **Mr. R. SAKTHIVEL, M.E,** |
| **HEAD OF THE DEPARTMENT** | **ASSISTANT** |
| **PROFESSOR** | **PROFESSOR** |
| Department of Electrical and | Department of Electrical and |
| Electronics Engineering, | Electronics engineering |
| Panimalar Engineering College. | Panimalar Engineering College, |
| Chennai - 600 123 | Chennai - 600 123 |

Submitted for End Semester Project Viva Voce held on …………………… at Panimalar Engineering College, Chennai.

**INTERNAL EXAMINER** **EXTERNAL EXAMINER**

# ACKNOWLEDGEMENT

Our sincere thanks to our Honourable Founder and Chairman, Dr. JEPPIAAR, M.A., B.L., Ph.D., for his sincere endeavor in educating us in his premier institution.

We would like to express our deep gratitude to our beloved Secretary and correspondent, Dr. P. CHINNADURAI, M.A., M.Phil., Ph.D., for his enthusiastic motivation which inspired us a lot in completing this project and our sincere thanks to our Directors Mrs. C. VIJAYA RAJESWARI, Dr. C. SAKTHIKUMAR, M.E., Ph.D., and Dr. SARANYASREE SAKTHI KUMAR, B.E., M.B.A., Ph.D., for providing us with necessary facilities for the completion of this project.

We would like to express thanks to our Principal, Dr. K. MANI, M.E., Ph.D., for extending his guidance and cooperation.

We would also like to thank our Head of the Department, Dr. S. SELVI, M.E., Ph.D., Professor and Head, Department of Electrical and Electronics Engineering for her encouragement. Personally, we thank our guide Mr. R. SAKTHIVEL, M.E, Assistant Professor in the Department of Electrical and Electronics Engineering for the persistent motivation and support for this project, who at all times was the mentor of the germination of this project from a small idea.

We express our sincere thanks to the project coordinators Dr. S. DEEPA, M.E., Ph.D., & Mr. R. SAKTHIVEL, M.E, in the Department of Electrical and Electronics Engineering for the valuable suggestions from time to time at every stage of our project.

Finally, we would like to take this opportunity to thank our family members, faculty, and non-teaching staff members of our department, friends, and well-wishers who have helped us for the successful completion of our project.

# ABSTRACT

The proposed project aims to develop a real-time traffic management system utilizing YOLO (You Only Look Once) machine learning for object detection and ESP32 microcontroller coupled with LED lights for hardware control. Traditional traffic management systems often rely on predefined timing patterns, which may not efficiently adapt to dynamically changing traffic conditions. By integrating machine learning with hardware control, our system can dynamically adjust traffic signals based on real-time detection of vehicles, pedestrians, and other relevant objects.

The system leverages pre-trained YOLO models for object detection, customized to recognize traffic-related entities such as vehicles, pedestrians, traffic lights, and signs. Communication between the YOLO model and the ESP32 microcontroller enables real-time data processing, facilitating prompt decision-making for traffic signal adjustments.

ESP32 serves as the central controller, receiving object detection data via MQTT messages and orchestrating the operation of LED lights to emulate real-world traffic signals. The system's intelligence lies in its ability to analyze incoming data streams, prioritize traffic flow, and dynamically adjust signal timings based on detected objects. Key components of the project include configuring the YOLO model for object detection, integrating ESP32 with MQTT for communication, and implementing algorithms for real-time traffic signal adjustments.

By leveraging MQTT for efficient communication and combining machine learning with hardware control, the proposed traffic management system offers a smart, adaptable solution to optimize traffic flow, enhance safety, and reduce congestion in urban and suburban environments.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| IOT | Internet of Things |
| YOLO | You Only Look Once Model |
| ESP | Espressif Systems |
| MQTT | Message Queuing Telemetry Transport |
| CNN | Convolutional Neural Network |
| RCNN | Region-Based Convolutional Neural Network |
| NMS | Non-Max Suppression |
| IOU | Intersection Over Unions |
| TCP | Transmission Control Protocol |
| AWS | Amazon Web Services |
| MISRA | Motor Industry Software Reliability Association |
| IDE | Integrated development environment |
| CPU | Central processing unit |
| GPU | Graphics processing unit |
| OASIS | Organization for the Advancement of Structured Information Standards |
| OTA | Over-the-air |
| TLS | Transport Layer Security |
| SSL | Secure Sockets Layer |
| PyPI | Python Package Index |
| HALs | Hardware abstraction layers |
| QoS | Quality of Service |

# CHAPTER - 1

# INTRODUCTION

Traffic management is a critical aspect of urban infrastructure, influencing the efficiency of transportation systems, public safety, and overall quality of life in cities worldwide. Traditional traffic management systems often rely on fixed timing patterns for traffic signals, which may not effectively adapt to fluctuating traffic conditions, leading to congestion, delays, and safety hazards. To address these challenges, the proposed project aims to develop an innovative traffic management system that leverages advanced technologies such as machine learning and IoT (Internet of Things) for real-time, adaptive control of traffic signals.

The project integrates two main components: YOLO (You Only Look Once) machine learning for object detection and ESP32 microcontroller with MQTT (Message Queuing Telemetry Transport) protocol for hardware control. YOLO is a state-of-the-art object detection algorithm known for its speed and accuracy, making it well-suited for real-time applications such as traffic monitoring. By training YOLO models to recognize various traffic-related entities such as vehicles, pedestrians, traffic lights, and signs, the system can effectively analyze traffic conditions in real-time.

The ESP32 microcontroller, equipped with built-in Wi-Fi and Bluetooth capabilities, serves as the central control unit for the traffic management system. It receives object detection data from the YOLO model via MQTT messages, enabling seamless communication and rapid decision-making for traffic signal adjustments. ESP32 then orchestrates the operation of LED lights installed at intersections to emulate conventional traffic signals, dynamically optimizing signal timings based on detected traffic patterns.

The proposed system offers several advantages over traditional traffic management approaches. Firstly, its adaptive nature allows it to respond in real-time to changing traffic conditions, improving overall traffic flow and reducing congestion. Secondly, by incorporating machine learning, the system can continuously learn and adapt to evolving traffic patterns, enhancing its effectiveness over time. Additionally, the use of IoT technology enables remote monitoring and control of traffic signals, facilitating efficient maintenance and management of the system.

In this project, we will outline the design, development, and implementation of the traffic management system, including the configuration of YOLO for object detection, integration of ESP32 with MQTT protocol, and deployment of LED lights for hardware control. We will also discuss the testing and validation procedures conducted to ensure the reliability and effectiveness of the system in simulated traffic scenarios. Ultimately, the proposed system aims to provide a smart, adaptable solution for optimizing traffic flow, enhancing safety, and improving the overall urban mobility experience.

# CHAPTER - 2

# LITERATURE SURVEY

R. Zhang, Y. Zhang, W. Shi, and H. Guo conducted a review of traffic congestion prediction and management techniques. The review highlights the limitations of existing systems and emphasizes the need for adaptive, real-time solutions to address dynamically changing traffic conditions.

S. Alomari, M. A. Kadir, M. Z. Khan, and M. K. Hasan conducted a survey on traffic management using machine learning techniques. The survey explores various machine learning algorithms employed in traffic management systems and evaluates their effectiveness in improving traffic efficiency and safety.

J. Redmon, S. Divvala, R. Girshick, and A. Farhadi introduced the YOLO (You Only Look Once) algorithm for real-time object detection. The paper discusses the architecture and training process of YOLO models and their applications in traffic monitoring and other domains.

N. Saxena, S. Chaudhary, N. Kumar, and S. Gupta conducted a comprehensive review of IoT-based smart traffic management systems. The review discusses the architecture, components, and applications of IoT technologies in enabling real-time data collection, analysis, and control of traffic signals.

L. Ma, K. Ota, M. Dong, X. Yang, and H. Zhu presented an overview of the ESP32 microcontroller as a low-cost IoT platform. The paper discusses the architecture, features, and programming options of the ESP32, highlighting its suitability for wireless communication and sensor integration in IoT projects.

A. Banerjee, S. Nandi, and S. Nandi conducted a technical review of the MQTT protocol. The review explores the lightweight, publish-subscribe messaging model of MQTT and its application in IoT systems, making it suitable for communication between devices in traffic management systems.

# CHAPTER - 3

# EXISTING SYSTEM

Existing traffic management systems typically rely on fixed timing patterns for traffic signals, often predetermined based on historical traffic data or manually adjusted by traffic engineers. While these systems have been effective to some extent, they suffer from several limitations and flaws:

- Static Timing Patterns: Traditional traffic signal systems operate based on fixed timing patterns that do not adapt to real-time traffic conditions. This lack of adaptability can lead to inefficient traffic flow, especially during peak hours or in response to unexpected events such as accidents or road closures.
- Limited Sensor Integration: Many existing traffic management systems rely on rudimentary sensors such as induction loops or cameras at intersections to detect vehicles and trigger signal changes. These sensors may be prone to inaccuracies or limited in their ability to capture real-time traffic dynamics comprehensively.
- Inefficient Signal Coordination: Coordination between adjacent traffic signals is often suboptimal in traditional systems, leading to traffic congestion and delays, particularly along major arterial routes. Without real-time coordination and synchronization, traffic signals may operate independently, exacerbating congestion and increasing travel times.
- Manual Adjustment: Traffic signal timing adjustments in conventional systems are often performed manually by traffic engineers based on historical data or periodic observations. This manual intervention can be time-consuming and reactive, limiting the system's ability to respond dynamically to changing traffic patterns.
- Limited Data Analysis: Traditional traffic management systems may lack sophisticated data analysis capabilities, hindering their ability to derive

actionable insights from real-time traffic data. Without comprehensive data analysis, these systems may struggle to optimize signal timings effectively and proactively manage traffic flow.

- Maintenance Challenges: Maintenance of traditional traffic signal systems can be costly and disruptive, particularly for systems with outdated infrastructure or components. Regular maintenance and upgrades are necessary to ensure optimal system performance and reliability, which can pose logistical challenges and incur additional expenses.
- Vulnerability to Failures: Traditional traffic management systems may be susceptible to single points of failure, such as hardware malfunctions or communication breakdowns. These failures can lead to disruptions in traffic flow, safety hazards, and delays in resolving issues, highlighting the need for robust and resilient system design.

**Fig. 3.1 Existing Traffic lights**

# CHAPTER - 4

# PROPOSED SYSTEM

## 4.1 BLOCK DIAGRAM

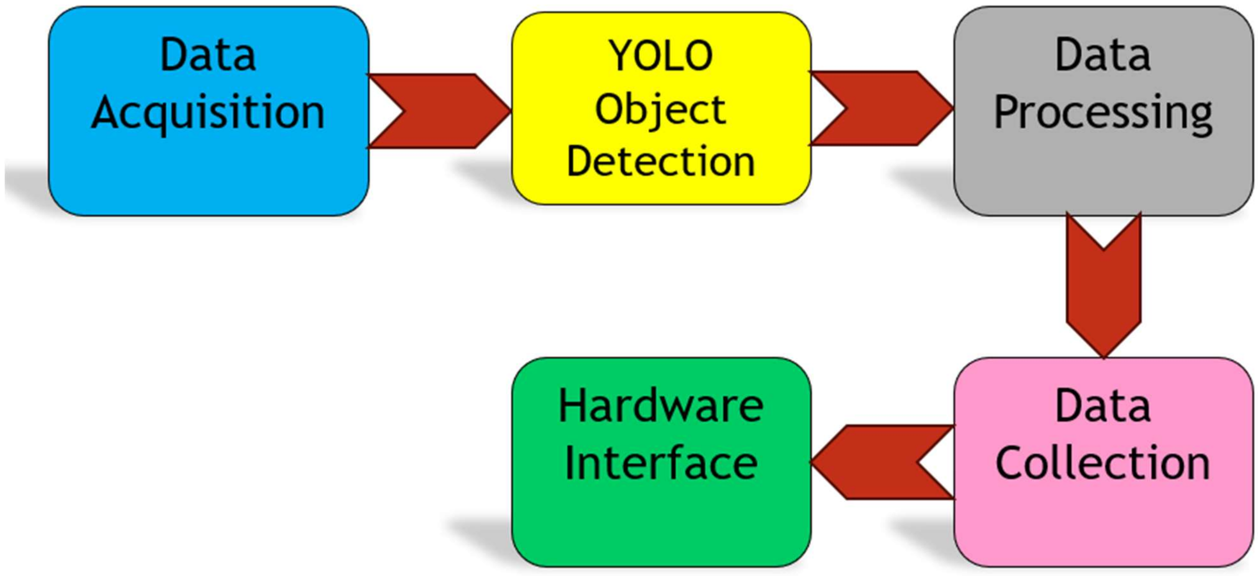


**Fig.4.1 Block Diagram of proposed system**

## 4.2 DATA ACQUISITION

Data acquisition process would primarily involve collecting and annotating images relevant to traffic scenarios. a diverse set of images representing various traffic scenarios. This may include:

- Images of intersections with different traffic signal configurations.
- Images of roads with different types of vehicles (cars, trucks, bicycles) and pedestrians.
- Images captured during different times of the day and under various weather conditions.
- Images with different lighting conditions (daytime, nighttime, low visibility).

This meticulous image collection process ensures a well-rounded dataset that adequately represents the intricacies of real-world traffic scenarios

**Fig.4.2 Sample Image 1**

**Fig.4.3 Sample Image 2**

**Fig.4.4 Sample Image 3**

## 4.3 YOLO ALGORITHM

You Only Look Once (YOLO) is a state-of-the-art, real-time object detection algorithm introduced in 2015 by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi in their famous research paper "You Only Look Once: Unified, Real-Time Object Detection".

The authors frame the object detection problem as a regression problem instead of a classification task by spatially separating bounding boxes and associating probabilities to each of the detected images using a single convolutional neural network (CNN).

You Only Look Once (YOLO) proposes using an end-to-end neural network that makes predictions of bounding boxes and class probabilities all at once. It differs from the approach taken by previous object detection algorithms, which repurposed classifiers to perform detection. Following a fundamentally different approach to object detection, YOLO achieved state-of-the-art results, beating other real-time object detection algorithms by a large margin.

While algorithms like Faster RCCN work by detecting possible regions of interest using the Region Proposal Network and then performing recognition on those regions separately, YOLO performs all of its predictions with the help of a single fully connected layer.

Methods that use Region Proposal Networks perform multiple iterations for the same image, while YOLO gets away with a single iteration.

Some of the reasons why YOLO is leading the competition include its:

1. **Speed**

   YOLO is extremely fast because it does not deal with complex pipelines. It can process images at 45 Frames Per Second (FPS). In addition, YOLO reaches more than twice the mean Average Precision (mAP) compared to other real-time systems, which makes it a great candidate for real-time processing.

From the graphic below, we observe that YOLO is far beyond the other object detectors with 91 FPS.

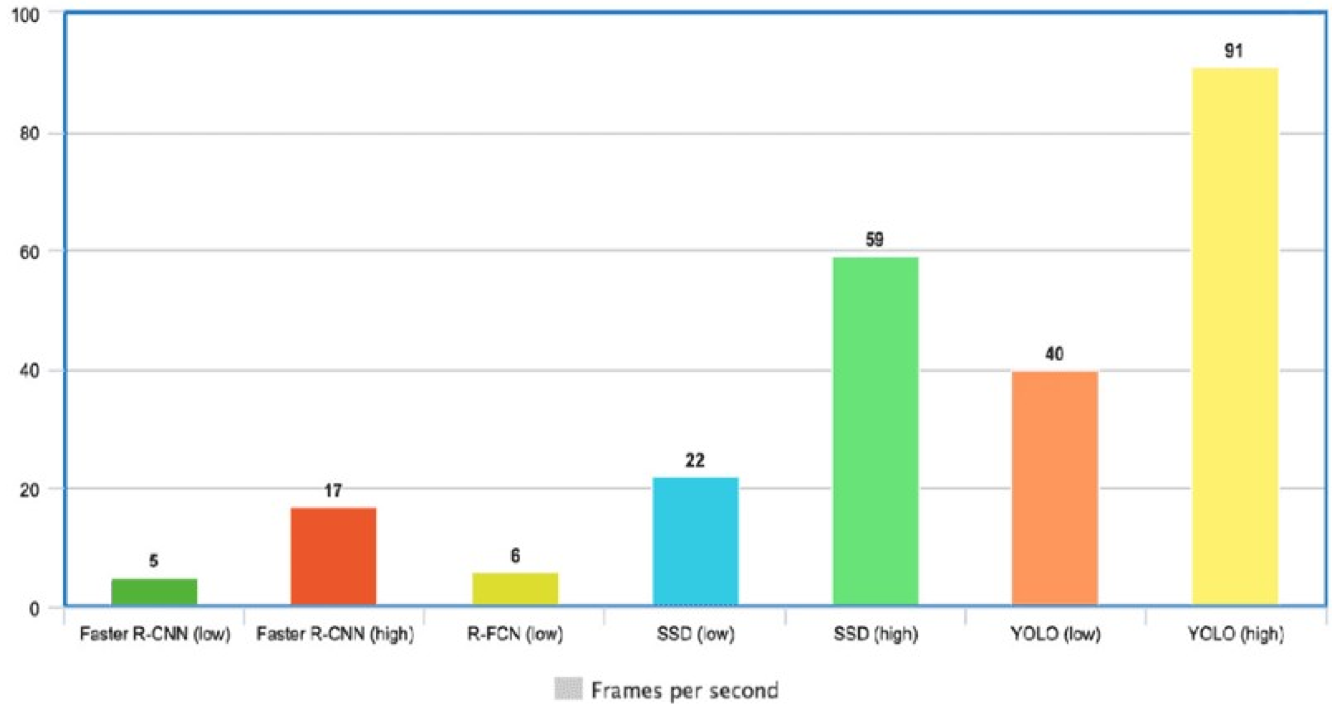


**Fig.4.5 YOLO Speed compared to other state-of-the-art object detectors**

2. **High detection accuracy**

YOLO is far beyond other state-of-the-art models in accuracy with very few background errors.

3. **Better generalization**

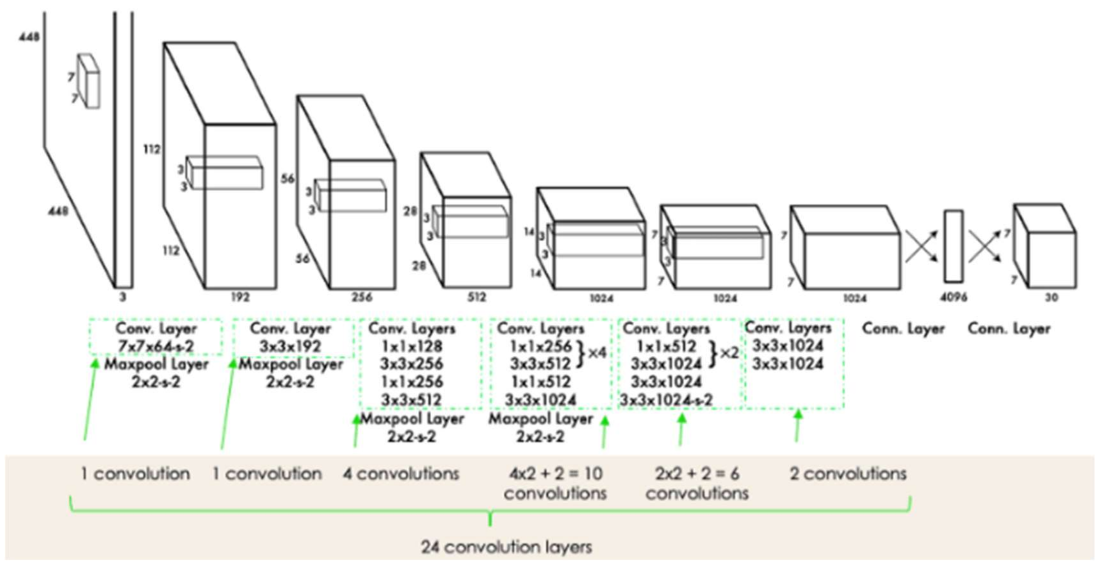
This is especially true for the new versions of YOLO, which will be discussed later in the article. With those advancements, YOLO pushed a little further by providing a better generalization for new domains, which makes it great for applications relying on fast and robust object detection.

For instance the Automatic Detection of Melanoma with Yolo Deep Convolutional Neural Networks paper shows that the first version YOLOv1 has the lowest mean average precision for the automatic detection of melanoma disease, compared to YOLOv2 and YOLOv3.

4. **Open source**

Making YOLO open-source led the community to constantly improve the model. This is one of the reasons why YOLO has made so many improvements in such a limited time.

### 4.3.1 YOLO ARCHITECTURE



Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 }x4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 }x2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer
1 convolution
1 convolution
4 convolutions
4x2 + 2 = 10 convolutions
2x2 + 2 = 6 convolutions
2 convolutions
24 convolution layers

**Fig.4.6 YOLO Architecture**

The architecture works as follows:

- Resizes the input image into 448x448 before going through the convolutional network.
- A 1x1 convolution is first applied to reduce the number of channels, which is then followed by a 3x3 convolution to generate a cuboidal output.

- The activation function under the hood is ReLU, except for the final layer, which uses a linear activation function.
- Some additional techniques, such as batch normalization and dropout, respectively regularize the model and prevent it from overfitting.

### 4.3.2 WORKING OF YOLO OBJECT DETECTION

The algorithm works based on the following four approaches:

- Residual blocks
- Bounding box regression
- Intersection Over Unions or IOU for short
- Non-Maximum Suppression.

Let's have a closer look at each one of them.

#### 1.Residual blocks

This first step starts by dividing the original image (A) into NxN grid cells of equal shape, where N in our case is 4 shown on the image on the right. Each cell in the grid is responsible for localizing and predicting the class of the object that it covers, along with the probability/confidence value.

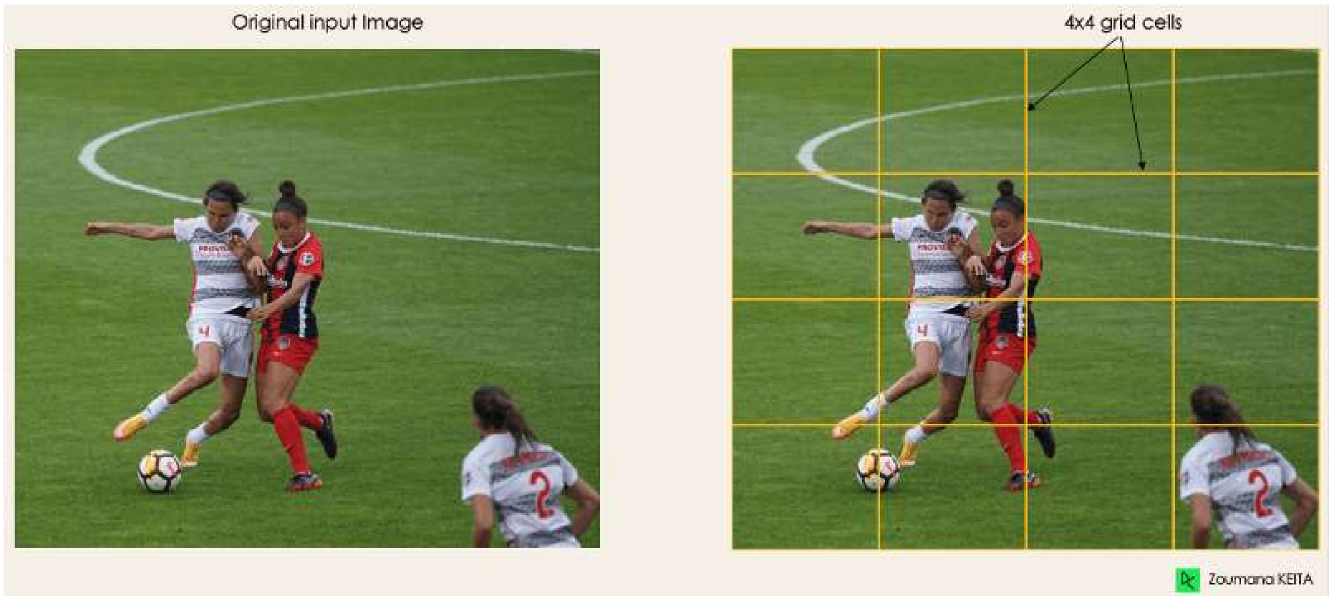


**Fig.4.7 Residual Blocks**

2. **Bounding box regression**

The next step is to determine the bounding boxes which correspond to rectangles highlighting all the objects in the image. We can have as many bounding boxes as there are objects within a given image.

YOLO determines the attributes of these bounding boxes using a single regression module in the following format, where Y is the final vector representation for each bounding box.

Y = [pc, bx, by, bh, bw, c1, c2]

This is especially important during the training phase of the model.

- pc corresponds to the probability score of the grid containing an object. For instance, all the grids in red will have a probability score higher than zero. The image on the right is the simplified version since the probability of each yellow cell is zero (insignificant).

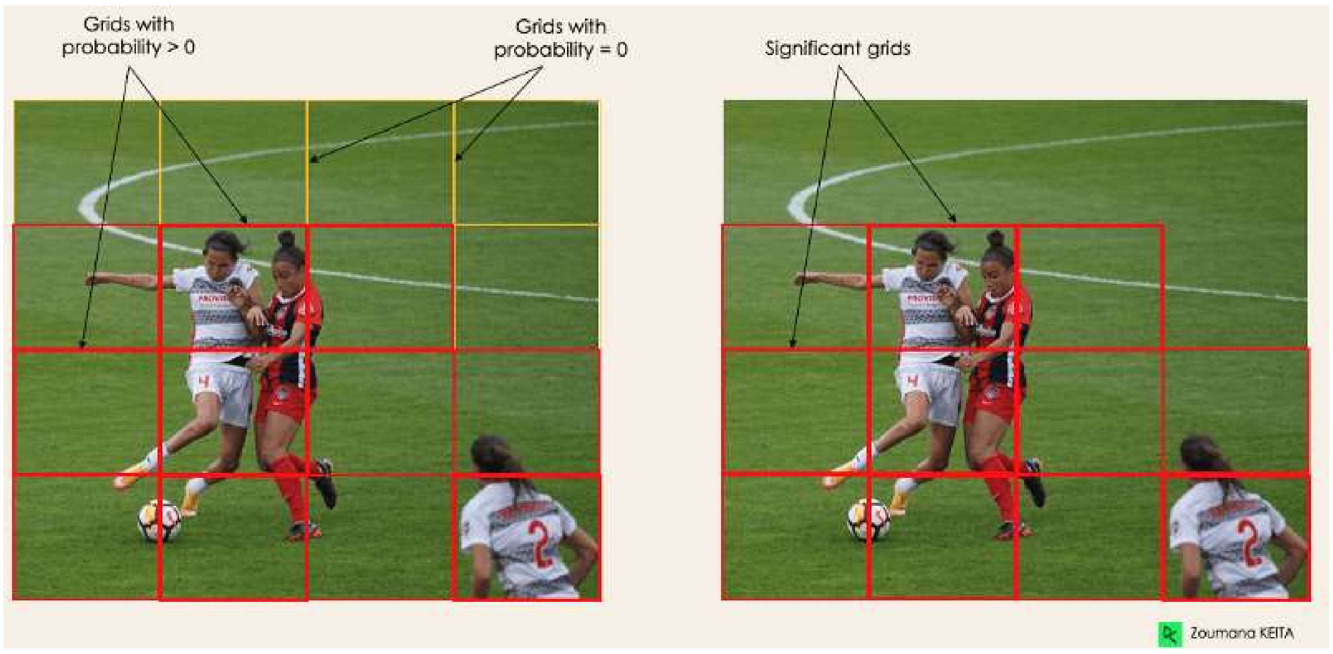
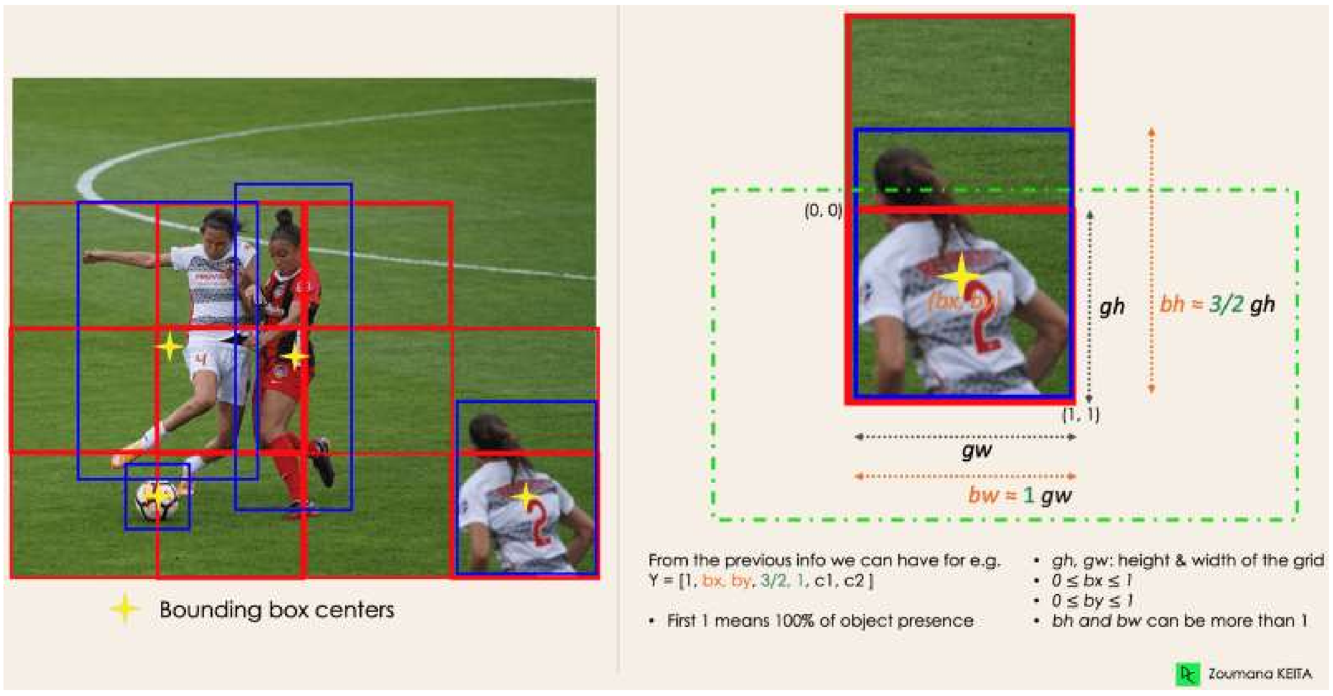


**Fig.4.8 Bounding Boxes 1**

- bx, by are the x and y coordinates of the center of the bounding box with respect to the enveloping grid cell.

- bh, bw correspond to the height and the width of the bounding box with respect to the enveloping grid cell.

- c1 and c2 correspond to the two classes Player and Ball. We can have as many classes as your use case requires.



Bounding box centers
(0, 0)
(bx, by)
gh
bh ≈ 3/2 gh
(1, 1)
gw
bw ≈ 1 gw
From the previous info we can have for e.g.
Y = [1, bx, by, 3/2, 1, c1, c2]
First 1 means 100% of object presence
gh, gw: height & width of the grid
0 ≤ bx ≤ 1
0 ≤ by ≤ 1
bh and bw can be more than 1
Zoumana KEITA

**Fig.4.9 Bounding Boxes 2**

**3. Intersection Over Unions or IOU**

Most of the time, a single object in an image can have multiple grid box candidates for prediction, even though not all of them are relevant. The goal of the IOU (a value between 0 and 1) is to discard such grid boxes to only keep those that are relevant. Here is the logic behind it:

- The user defines its IOU selection threshold, which can be, for instance, 0.5.
- Then YOLO computes the IOU of each grid cell which is the Intersection area divided by the Union Area.
- Finally, it ignores the prediction of the grid cells having an IOU ≤ threshold and considers those with an IOU > threshold.

Below is an illustration of applying the grid selection process to the bottom left object. We can observe that the object originally had two grid candidates, then only "Grid 2" was selected at the end.

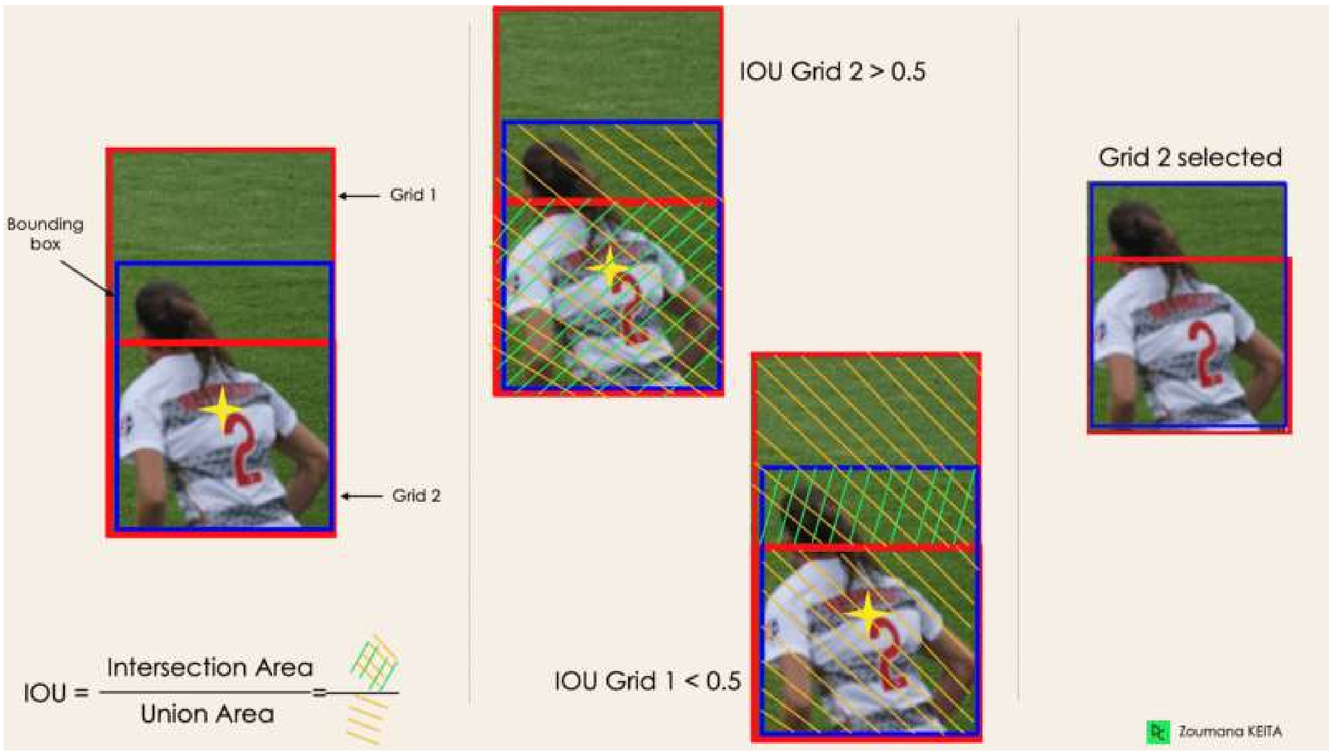


**Fig.4.10 Intersection over Unions**

**4. Non-Max Suppression or NMS**

Setting a threshold for the IOU is not always enough because an object can have multiple boxes with IOU beyond the threshold, and leaving all those boxes might include noise. Here is where we can use NMS to keep only the boxes with the highest probability score of detection.

### 4.3.3 YOLO APPLICATIONS

YOLO object detection has different applications in our day-to-day life. In this section, we will cover some of them in the following domains: healthcare, agriculture, security surveillance, and self-driving cars.

**1.Application in industries**

Object detection has been introduced in many practical industries such as healthcare and agriculture. Let’s understand each one with specific examples.

Healthcare: Specifically in surgery, it can be challenging to localize organs in real-time, due to biological diversity from one patient to another. Kidney Recognition in CT used YOLOv3 to facilitate localizing kidneys in 2D and 3D from computerized tomography (CT) scans.

The Biomedical Image Analysis in Python course can help you learn the fundamentals of exploring, manipulating, and measuring biomedical image data using Python.

Agriculture: Artificial Intelligence and robotics are playing a major role in modern agriculture. Harvesting robots are vision-based robots that were introduced to replace the manual picking of fruits and vegetables. One of the best models in this field uses YOLO. In Tomato detection based on modified YOLOv3 framework, the authors describe how they used YOLO to identify the types of fruits and vegetables for efficient harvest.

**2.Security surveillance**

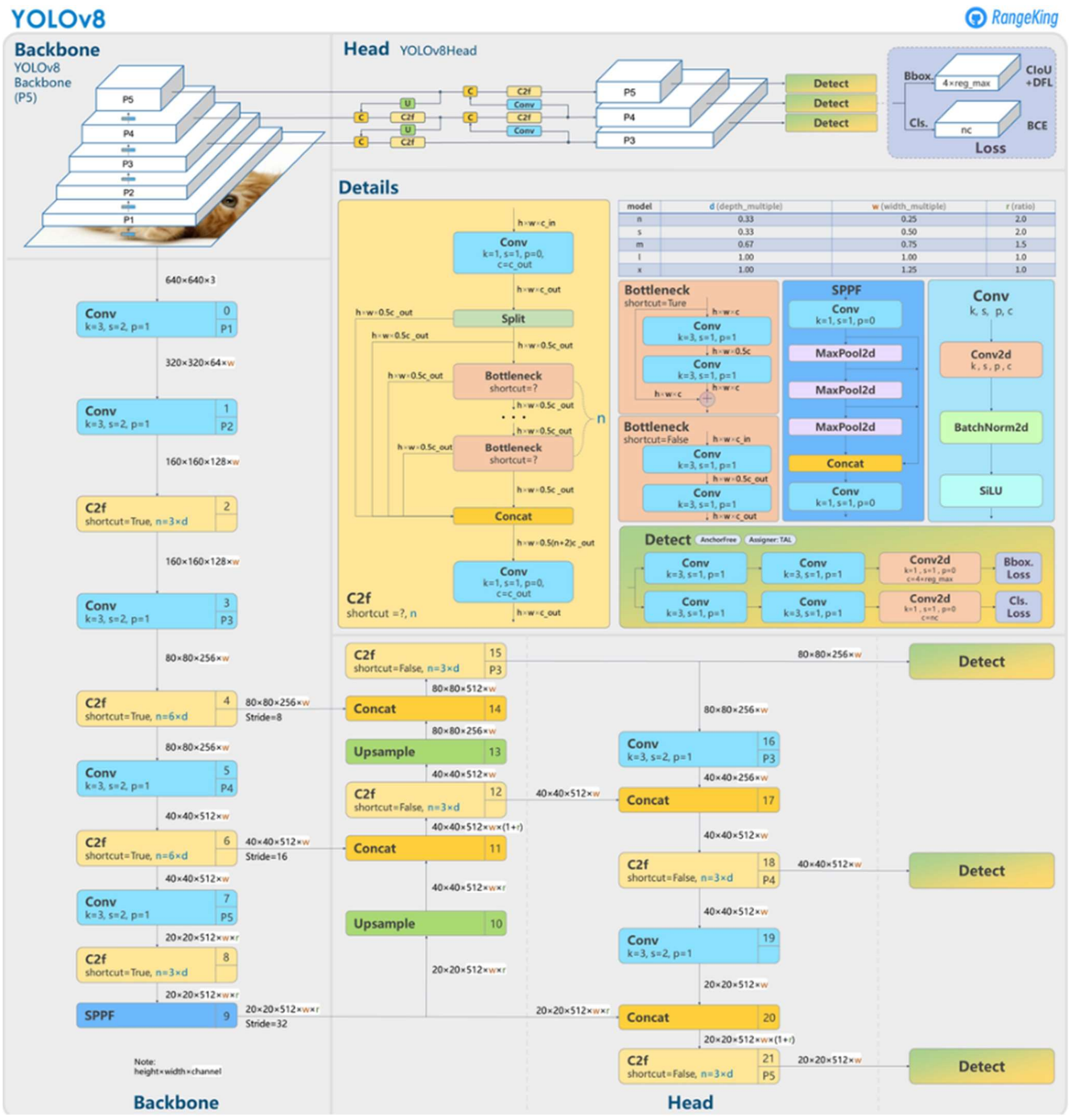
Even though object detection is mostly used in security surveillance, this is not the only application. YOLOv3 has been used during covid19 pandemic to estimate social distance violations between people.

**3. Self-driving cars**

Real-time object detection is part of the DNA of autonomous vehicle systems. This integration is vital for autonomous vehicles because they need to properly identify the correct lanes and all the surrounding objects and pedestrians to increase road safety. The real-time aspect of YOLO makes it a better candidate compared to simple image segmentation approaches.

## 4.3.4 YOLO V8

The **YOLO (You Only Look Once)** series of models has become famous in the computer vision world. YOLO's fame is attributable to its considerable accuracy while maintaining a small model size. YOLO models can be trained on a single GPU, which makes it accessible to a wide range of developers. Machine learning practitioners can deploy it for low cost on edge hardware or in the cloud.



| model | d (depth_multiple) | w (width_multiple) | r (ratio) |
|---|---|---|---|
| n | 0.33 | 0.25 | 2.0 |
| s | 0.33 | 0.50 | 2.0 |
| m | 0.67 | 0.75 | 1.5 |
| l | 1.00 | 1.00 | 1.0 |
| x | 1.00 | 1.25 | 1.0 |

**Fig 4.11 YOLO v8 Architecture**

The task of retrieving high-level information from input photos falls under the purview of the backbone network. Yolo V8 uses an enhanced version of the CSPDarknet53 architecture, which has been demonstrated to be efficient in the recording of accurate location data.

This architecture was developed by Yolo.The fusion of scale-invariant features is the responsibility of the neck network. Path Aggregation Network, more commonly referred to as PANet, serves as the primary backbone network for Yolo V8.

PANet provides a more accurate representation of the characteristics by combining the data that is gathered from multiple layers of the underlying network.

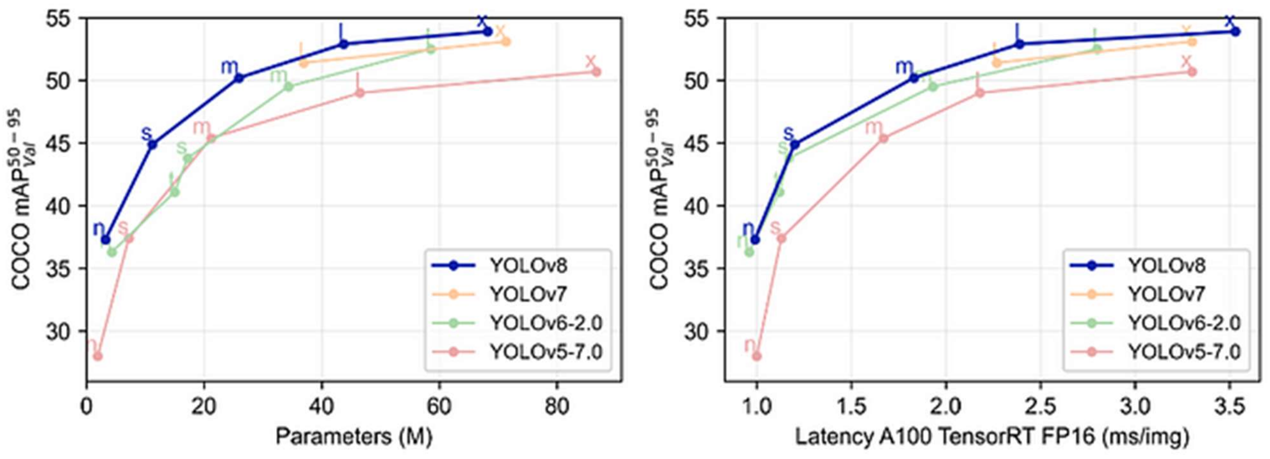


**Fig 4.12 YOLO v8 Accuracy**

YOLOv8 is an anchor-free model. This means it predicts directly the center of an object instead of the offset from a known anchor box.

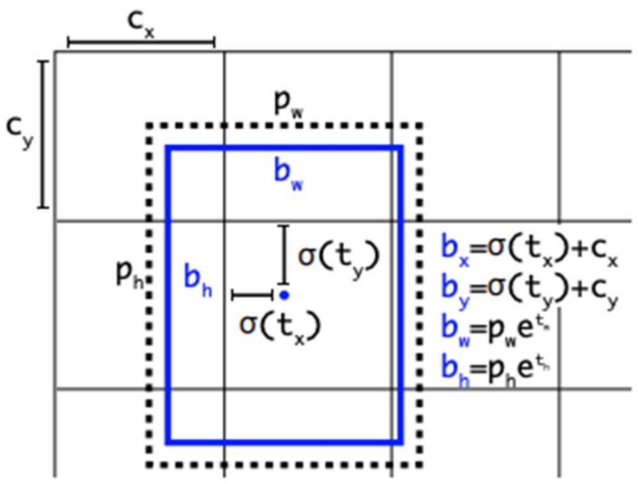


**Fig 4.13 Visualization of an anchor box in YOLO**

Anchor boxes were a notoriously tricky part of earlier YOLO models, since they may represent the distribution of the target benchmark's boxes but not the distribution of the custom dataset.

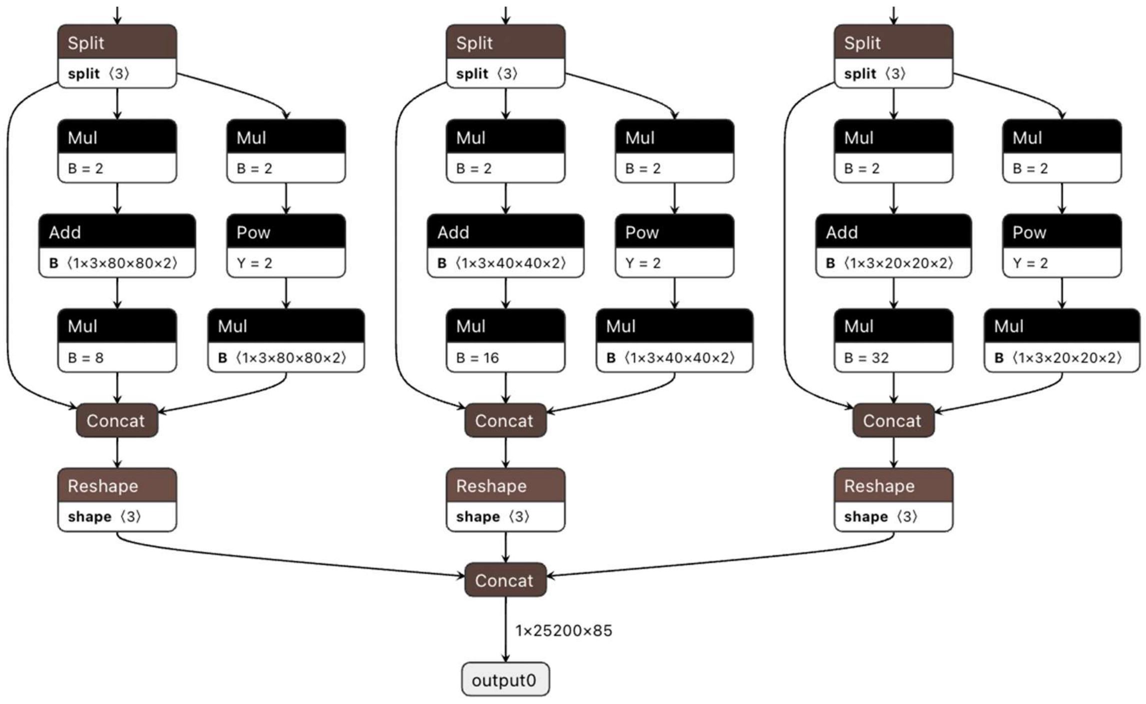


**Fig 4.14 The detection head of YOLOv5**

Anchor free detection reduces the number of box predictions, which speeds up Non-Maximum Suppression (NMS), a complicated post processing step that sifts through candidate detections after inference.

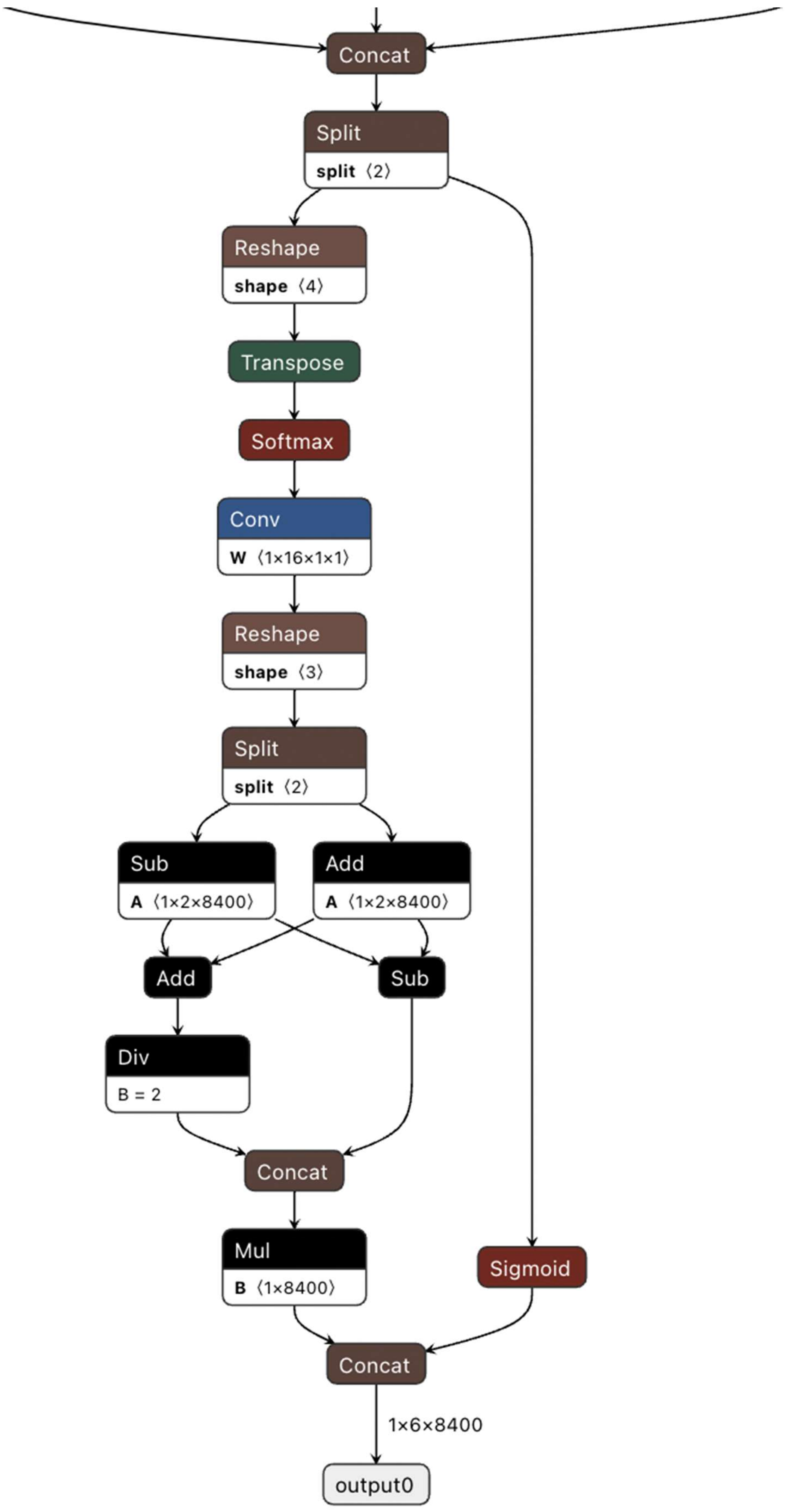


**Fig 4.15 The detection head of YOLOv8**

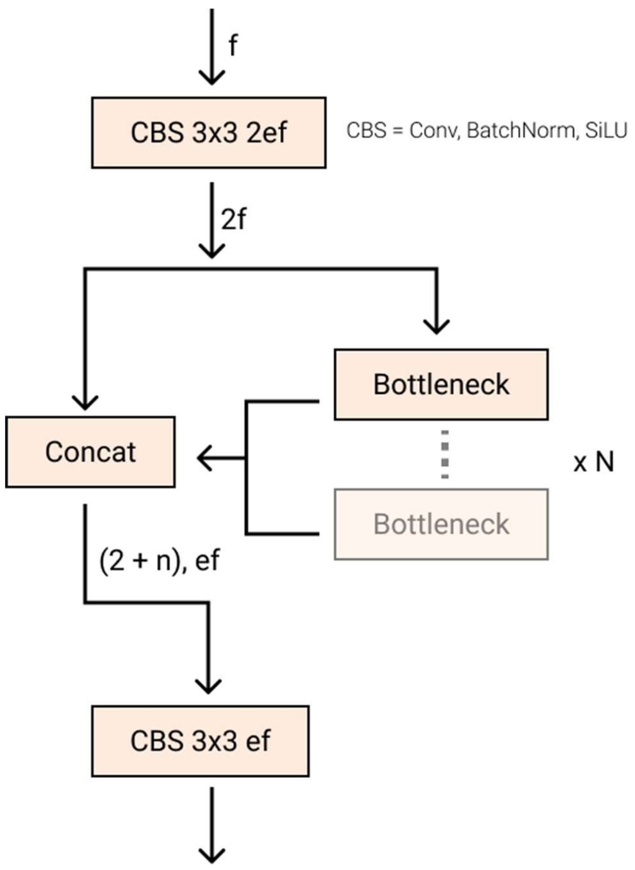


Fig 4.16 New YOLOv8 c2f module

## 4.4 MQTT Protocol

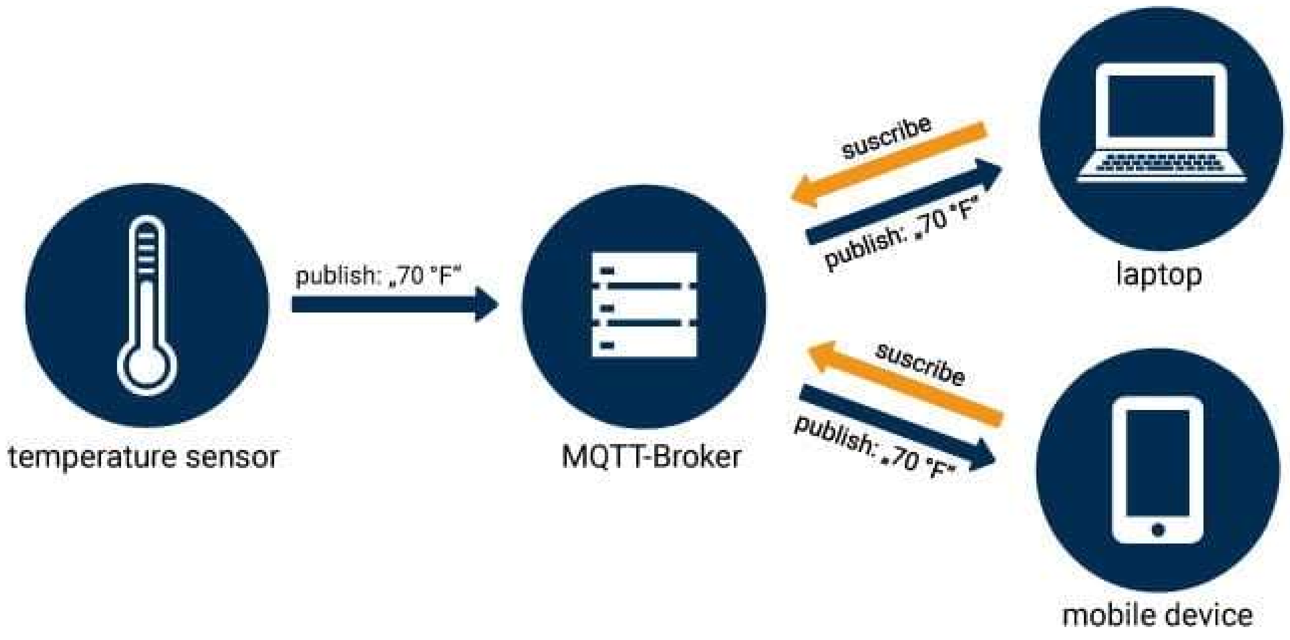


Fig 4.19 Overview of MQTT

The MQTT protocol was invented in 1999 for use in the oil and gas industry. Engineers needed a protocol for minimal bandwidth and minimal battery loss to monitor oil pipelines via satellite. Initially, the protocol was known as Message Queuing Telemetry Transport due to the IBM product MQ Series that first supported its initial phase. In 2010, IBM released MQTT 3.1 as a free and open protocol for anyone to implement, which was then submitted, in 2013, to Organization for the Advancement of Structured Information Standards (OASIS) specification body for maintenance. In 2019, an upgraded MQTT version 5 was released by OASIS. Now MQTT is no longer an acronym but is considered to be the official name of the protocol. MQTT is the most commonly used messaging protocol for the Internet of Things (IoT). MQTT stands for MQ Telemetry Transport. The protocol is a set of rules that defines how IoT devices can publish and subscribe to data over the Internet. MQTT is used for messaging and data exchange between IoT and industrial IoT (IIoT) devices, such as embedded devices, sensors, industrial PLCs, etc. The protocol is event driven and connects devices using the publish / subscribe (Pub/Sub) pattern. The sender (Publisher) and the receiver (Subscriber) communicate via Topics and are decoupled from each other. The connection between them is handled by the MQTT broker. The MQTT broker filters all incoming messages and distributes them correctly to the Subscribers.

MQTT has emerged as one of the best IoT protocols due to its unique features and capabilities tailored to the specific needs of IoT systems. Some of the key reasons include:

- Lightweight: IoT devices are often constrained in terms of processing power, memory, and energy consumption. MQTT's minimal overhead and small packet size make it ideal for these devices, as it consumes fewer resources, enabling efficient communication even with limited capabilities.
- Reliable: IoT networks can experience high latency or unstable connections. MQTT's support for different QoS levels, session awareness,

and persistent connections ensures reliable message delivery even in challenging conditions, making it well-suited for IoT applications.

- Secure communications: Security is crucial in IoT networks as they often transmit sensitive data. MQTT supports Transport Layer Security (TLS) and Secure Sockets Layer (SSL) encryption, ensuring data confidentiality during transmission. Additionally, it provides authentication and authorization mechanisms through username/password credentials or client certificates, safeguarding access to the network and its resources.
- Bi-directionality: MQTT's publish-subscribe model allows for seamless bi-directional communication between devices. Clients can both publish messages to topics and subscribe to receive messages on specific topics, enabling effective data exchange in diverse IoT ecosystems without direct coupling between devices. This model also simplifies the integration of new devices, ensuring easy scalability.
- Continuous, stateful sessions: MQTT allows clients to maintain stateful sessions with the broker, enabling the system to remember subscriptions and undelivered messages even after disconnection. Clients can also specify a keep-alive interval during connection, which prompts the broker to periodically check the connection status. If the connection is lost, the broker stores undelivered messages (depending on the QoS level) and attempts to deliver them when the client reconnects. This feature ensures reliable communication and reduces the risk of data loss due to intermittent connectivity.
- Large-scale IoT device support: IoT systems often involve a large number of devices, requiring a protocol that can handle massive-scale deployments. MQTT's lightweight nature, low bandwidth consumption, and efficient use of resources make it well-suited for large-scale IoT applications. The publish-subscribe pattern allows MQTT to scale effectively, as it decouples sender and receiver, reducing network traffic and resource usage. Furthermore, the protocol's support for different QoS levels allows

customization of message delivery based on the application's requirements, ensuring optimal performance in various scenarios.

- Language support: IoT systems often include devices and applications developed using various programming languages. MQTT's broad language support enables easy integration with multiple platforms and technologies, fostering seamless communication and interoperability in diverse IoT ecosystems. You can visit our MQTT Client Programming blog series to learn how to use MQTT in PHP, Node.js, Python, Golang, Node.js, and other programming languages

MQTT implements the publish/subscribe model by defining clients and brokers as below.

## 1. MQTT client

An MQTT client is any device from a server to a microcontroller that runs an MQTT library. If the client is sending messages, it acts as a publisher, and if it is receiving messages, it acts as a receiver. Basically, any device that communicates using MQTT over a network can be called an MQTT client device.

## 2. MQTT broker

The MQTT broker is the backend system which coordinates messages between the different clients. Responsibilities of the broker include receiving and filtering messages, identifying clients subscribed to each message, and sending them the messages. It is also responsible for other tasks such as:

- Authorizing and authenticating MQTT clients
- Passing messages to other systems for further analysis
- Handling missed messages and client sessions

## 3. MQTT connection

Clients and brokers begin communicating by using an MQTT connection. Clients initiate the connection by sending a CONNECT message to the MQTT broker. The broker confirms that a connection has been established by responding with a CONNACK message. Both the MQTT client and the broker require a TCP/IP stack to communicate. Clients never connect with each other, only with the broker.

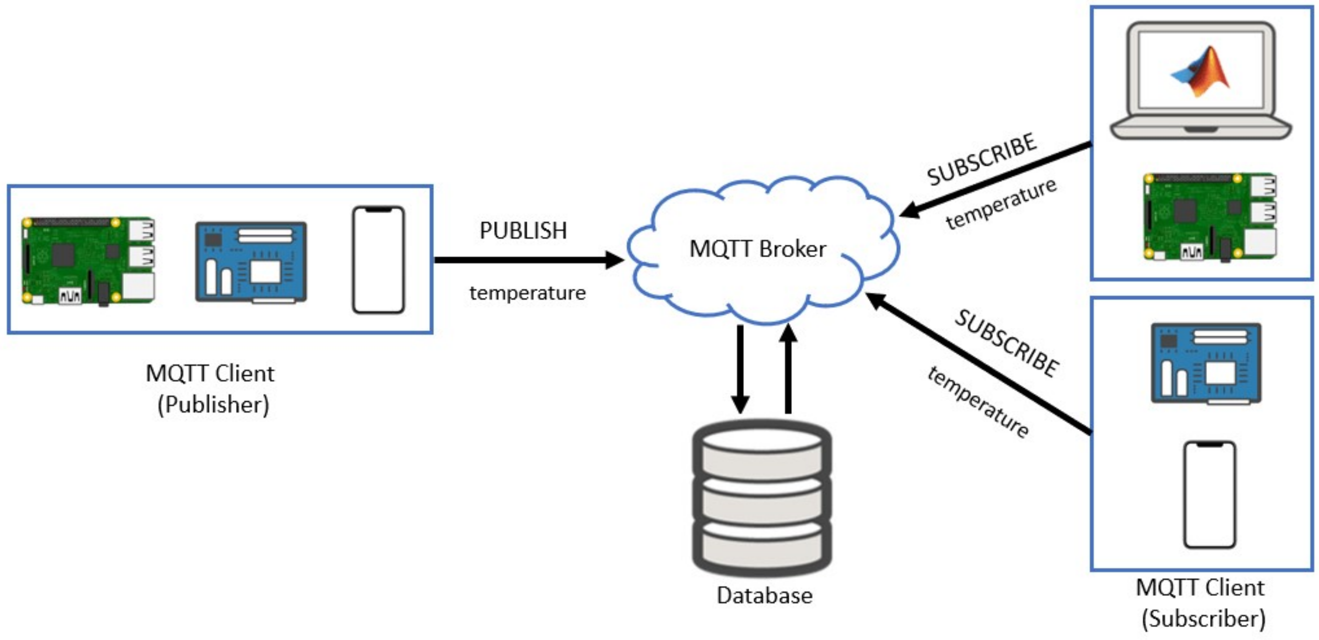


Fig. 4.20 Working of MQTT

# CHAPTER – 5

# SOFTWARES AND LIBRARIES USED

## 5.1 PYTHON

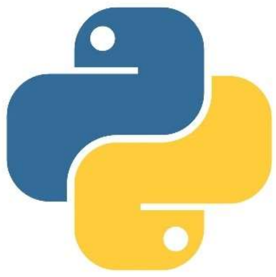


**Fig 5.1 Python**

Python is a high-level, interpreted programming language known for its simplicity, readability, and versatility. Created by Guido van Rossum and first released in 1991, Python has since become one of the most popular and widely used programming languages in the world. Here's an overview of Python:

- Simplicity and Readability: Python's syntax is designed to be clear and concise, resembling plain English, which makes it easy to learn and understand. Its indentation-based formatting enforces code readability and encourages developers to write clean and maintainable code.
- Interpreted and Interactive: Python is an interpreted language, meaning that code is executed line by line by an interpreter without the need for compilation. This allows for rapid development and testing through interactive sessions and immediate feedback.
- Rich Standard Library: Python comes with a comprehensive standard library that provides modules and packages for a wide range of tasks,

including file I/O, networking, web development, data manipulation, and more. This extensive library reduces the need for external dependencies and facilitates rapid development.

- Versatility: Python is a general-purpose programming language that supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Its versatility enables developers to tackle a wide range of tasks, from scripting and automation to complex software development and scientific computing.
- Cross-Platform Compatibility: Python is cross-platform, meaning that code written in Python can run on different operating systems without modification. This portability allows developers to write code once and deploy it across multiple platforms, including Windows, macOS, and Linux.
- Community and Ecosystem: Python boasts a vibrant and active community of developers who contribute to its ecosystem by creating and maintaining libraries, frameworks, and tools. The Python Package Index (PyPI) hosts thousands of third-party packages that extend Python's functionality for various purposes, making it easy to find solutions to diverse problems.
- Support for Data Science and Machine Learning: Python has emerged as a leading language for data science and machine learning due to its rich ecosystem of libraries and frameworks, such as NumPy, pandas, scikit-learn, TensorFlow, and PyTorch. These tools provide powerful capabilities for data manipulation, analysis, visualization, and machine learning model development.
- Web Development: Python is widely used for web development, with frameworks such as Django and Flask offering robust solutions for building web applications, APIs, and microservices. Python's simplicity and versatility make it an attractive choice for developing both backend and frontend components of web applications.
- Education and Community Outreach: Python's beginner-friendly syntax and extensive learning resources make it an ideal language for teaching

programming concepts to beginners. Python is often used in educational settings, coding bootcamps, and online courses to introduce students to programming and computational thinking.

In summary, Python's simplicity, readability, versatility, rich ecosystem, and community support make it a powerful and widely adopted programming language suitable for a wide range of applications and industries, from web development and data science to scientific computing and artificial intelligence. Its popularity and ease of use make it an excellent choice for both beginners and experienced developers alike.

## 5.2 EMBEDDED C

**Fig 5.2 Embedded C**

Embedded C is a programming language variant specifically designed for programming embedded systems. These systems typically have limited computational resources, such as microcontrollers or microprocessors with constrained memory and processing capabilities. Embedded C is tailored to meet the unique requirements and constraints of embedded systems, offering features and optimizations suited for efficient and low-level programming. Here's an overview of Embedded C:

- Low-Level Programming: Embedded C allows developers to interact directly with hardware peripherals and memory-mapped registers, enabling precise control over system behavior and resource utilization. This low-level programming approach is essential for writing device drivers, handling interrupts, and implementing real-time functionality in embedded systems.
- Efficient Memory Management: Embedded C emphasizes efficient memory management, as embedded systems often have limited RAM and ROM resources. Developers must carefully manage memory allocation, deallocation, and optimization to minimize memory usage and maximize system performance. Techniques such as static allocation, stack-based memory management, and memory pooling are commonly used in Embedded C programming.
- Hardware Abstraction: Embedded C supports hardware abstraction through the use of register definitions, macros, and inline assembly language. By abstracting hardware access and functionality, developers can write portable code that can be easily adapted to different microcontroller architectures and configurations. Hardware abstraction layers (HALs) and device-specific libraries further simplify hardware interaction and code reuse in Embedded C development.
- Real-Time Operation: Many embedded systems require real-time operation, where tasks must be executed within strict timing constraints. Embedded C provides features for implementing real-time scheduling, task prioritization, and interrupt handling to ensure timely response to external events and stimuli. Techniques such as polling, interrupts, and scheduling algorithms (e.g., round-robin, priority-based) are commonly used in real-time embedded systems programming.

- Cross-Platform Compatibility: Embedded C code is often written to be cross-platform compatible, allowing it to run on different microcontroller architectures and development environments. Standardized compiler directives, preprocessor macros, and hardware abstraction layers facilitate portability and code reuse across embedded systems with varying configurations and requirements.
- Development Tools and Environments: Embedded C development typically involves using specialized integrated development environments (IDEs), compilers, and debugging tools tailored for embedded systems programming. These tools provide features such as code editing, compilation, debugging, simulation, and emulation to streamline the development process and ensure code correctness and reliability.
- Embedded C Standards: The development of Embedded C follows industry standards and guidelines, including the MISRA C (Motor Industry Software Reliability Association) coding standard, which defines rules and best practices for writing safe and reliable embedded software. Adhering to standards helps ensure code quality, maintainability, and compliance with industry regulations and requirements.

In summary, Embedded C is a specialized programming language optimized for writing software for embedded systems. Its focus on low-level programming, efficient memory management, hardware abstraction, real-time operation, cross-platform compatibility, and adherence to industry standards makes it well-suited for developing embedded software solutions across a wide range of applications and industries.

## 5.3 ULTRALYTICS LIBRARY

**Fig.5.3 Ultralytics**

The Ultralytics library is a computer vision library primarily designed for object detection and image classification tasks. It provides easy-to-use interfaces for training, evaluating, and deploying deep learning models, particularly focusing on state-of-the-art architectures such as YOLO (You Only Look Once) and EfficientDet.

Here's an overview of the Ultralytics library:

- Object Detection: Ultralytics supports object detection tasks using YOLO (v3 and v5) and EfficientDet architectures. These models are capable of detecting and localizing multiple objects within an image, making them suitable for various applications such as surveillance, autonomous driving, and inventory management.
- Image Classification: In addition to object detection, Ultralytics offers support for image classification tasks using popular architectures like ResNet and EfficientNet. These models classify input images into predefined categories, enabling tasks such as content-based image retrieval, medical diagnosis, and product recognition.

- Training and Evaluation: The Ultralytics library provides user-friendly APIs for training deep learning models on custom datasets. It includes features such as data augmentation, transfer learning, and hyperparameter optimization to streamline the training process. Additionally, Ultralytics offers built-in evaluation tools to assess model performance on validation datasets, including metrics such as precision, recall, and mean average precision (mAP).
- Model Deployment: Once trained, models can be deployed for inference using Ultralytics' lightweight runtime engine. This engine enables efficient deployment of deep learning models on various platforms, including edge devices, cloud servers, and mobile applications. Models can be deployed as standalone applications or integrated into existing software systems using common deployment frameworks such as TensorFlow Serving or ONNX Runtime.
- Performance Optimization: Ultralytics prioritizes performance optimization to ensure fast and efficient inference on resource-constrained devices. This includes techniques such as model quantization, pruning, and acceleration using hardware accelerators like GPUs or TPUs. By optimizing model performance, Ultralytics enables real-time inference in latency-sensitive applications such as video analytics and robotics.
- Community and Support: Ultralytics is actively maintained by a community of developers and researchers, ensuring ongoing support and updates. The library is open source, allowing users to contribute improvements, report issues, and collaborate with other developers. Additionally, Ultralytics provides comprehensive documentation, tutorials, and examples to facilitate learning and usage of the library.

Overall, the Ultralytics library is a powerful tool for computer vision tasks, offering state-of-the-art models, intuitive APIs, and performance optimization features. Whether you're a researcher, developer, or hobbyist, Ultralytics simplifies the process of building, training, and deploying deep learning models for object detection and image classification applications.

## 5.4 PAHO MQTT LIBRARY

**Fig 5.4 Paho MQTT**

The Paho MQTT libraries are open-source client libraries that facilitate MQTT (Message Queuing Telemetry Transport) protocol communication in various programming languages. MQTT is a lightweight and efficient messaging protocol designed for constrained devices and unreliable networks, making it ideal for IoT (Internet of Things) applications, real-time messaging, and machine-to-machine (M2M) communication.

Here's an overview of the Paho MQTT libraries:

- Paho MQTT Clients: The Paho MQTT libraries provide client implementations in several programming languages, including Python, Java, C/C++, JavaScript, and more. Each client library offers consistent APIs and functionality across different platforms, allowing developers to seamlessly integrate MQTT communication into their applications regardless of the programming language used.
- MQTT Protocol Support: The Paho MQTT libraries support the full MQTT protocol specification, including features such as publishing and

subscribing to topics, QoS (Quality of Service) levels, message persistence, retained messages, and last will and testament messages. This comprehensive support enables robust and reliable communication between MQTT clients and brokers.

- Asynchronous Communication: Paho MQTT clients support asynchronous communication, allowing applications to send and receive messages without blocking the main execution thread. Asynchronous operation is essential for handling large volumes of messages efficiently and maintaining responsiveness in real-time applications.
- TLS/SSL Encryption: The Paho MQTT libraries support TLS (Transport Layer Security) and SSL (Secure Sockets Layer) encryption for secure communication between MQTT clients and brokers. This ensures that data exchanged over the network is encrypted and protected from eavesdropping and tampering, enhancing the security of IoT deployments and sensitive applications.
- Integration with MQTT Brokers: Paho MQTT clients seamlessly integrate with MQTT brokers, which are responsible for routing messages between clients. Popular MQTT brokers such as Eclipse Mosquitto, HiveMQ, and AWS IoT Core are fully compatible with the Paho MQTT libraries, enabling easy deployment and scalability of MQTT-based systems.
- Cross-Platform Compatibility: The Paho MQTT libraries are designed to be cross-platform, supporting deployment on a wide range of operating systems and hardware platforms. Whether running on desktop computers, embedded devices, or cloud servers, Paho MQTT clients can communicate with MQTT brokers using a consistent interface, simplifying development and deployment.
- Community and Support: The Paho MQTT project is maintained by the Eclipse Foundation, an open-source community dedicated to fostering collaboration and innovation in software development. As a result, the Paho MQTT libraries benefit from active community support, ongoing

development, and regular updates to address bugs, add new features, and improve performance.

Overall, the Paho MQTT libraries provide a reliable and efficient solution for implementing MQTT communication in a variety of applications and environments. Whether you're building IoT solutions, real-time messaging systems, or M2M communication protocols, Paho MQTT clients offer the flexibility, scalability, and security required for modern distributed applications.

## 5.5 GOOGLE COLAB

**Fig 5.5 Google Colab**

Google Colab, short for Google Colaboratory, is a cloud-based platform provided by Google that offers free access to Jupyter notebooks and computational resources. It allows users to write and execute Python code in a browser-based environment, eliminating the need for local installation of software or hardware resources.

Here are some key features and benefits of Google Colab:

- Free Access: Google Colab is available for free to anyone with a Google account. It provides access to computing resources, including CPU, GPU, and TPU (Tensor Processing Unit), without the need for additional setup or configuration.

- Jupyter Notebooks: Google Colab supports Jupyter notebooks, which provide an interactive and visual interface for writing and running Python code. Notebooks allow users to combine code, text, images, and multimedia content in a single document, making it easy to document and share code workflows and analyses.
- Collaboration: Google Colab enables real-time collaboration between multiple users on the same notebook. Users can share notebooks with collaborators, view changes made by others, and communicate via comments and chat, facilitating teamwork and knowledge sharing.
- Access to Google Services: Google Colab integrates seamlessly with other Google services, including Google Drive, Google Sheets, and Google Cloud Storage. Users can import and export data from these services directly within Colab notebooks, making it easy to access and manipulate data stored in the cloud.
- Hardware Acceleration: Google Colab provides access to GPU and TPU resources, allowing users to accelerate computation-intensive tasks such as deep learning model training and inference. Users can enable hardware acceleration with a single click, significantly reducing the time required to execute compute-intensive code.
- Pre-installed Libraries and Packages: Google Colab comes with many popular Python libraries and packages pre-installed, including TensorFlow, PyTorch, scikit-learn, pandas, and NumPy. This eliminates the need for users to install these libraries manually, allowing them to focus on writing code and performing analyses.
- Version Control and History: Google Colab automatically saves the history of notebook edits and provides version control features, allowing users to revert to previous versions of the notebook if needed. This ensures that changes are tracked and can be easily undone, providing peace of mind when experimenting with code.

- Integration with GitHub: Users can easily import notebooks from GitHub repositories into Google Colab or save notebooks directly to GitHub. This integration facilitates code sharing, collaboration, and reproducibility, allowing users to leverage the full power of version control and collaborative development workflows.

Overall, Google Colab is a powerful and user-friendly platform for writing, running, and sharing Python code in a collaborative and cloud-based environment. Its seamless integration with Jupyter notebooks, access to computing resources, collaboration features, and integration with Google services make it an invaluable tool for data scientists, machine learning engineers, researchers, educators, and anyone else looking to harness the power of Python in a cloud-based environment.

## 5.6 ARDUINO IDE

**Fig 5.6 Arduino IDE**

The Arduino Integrated Development Environment (IDE) is an open-source software tool used for writing, compiling, and uploading code to Arduino microcontroller boards. It provides a user-friendly interface for programming

Arduino devices, making it accessible to both beginners and experienced developers. Here's an overview of the Arduino IDE:

- Cross-Platform Compatibility: The Arduino IDE is available for multiple operating systems, including Windows, macOS, and Linux, making it accessible to a wide range of users across different platforms.
- Simple and Intuitive Interface: The Arduino IDE features a simple and intuitive interface that allows users to write, edit, and manage code easily. It provides syntax highlighting, auto-completion, and code suggestion features to assist users in writing code efficiently.
- Integrated Text Editor: The IDE includes a built-in text editor for writing Arduino sketches (programs). The editor supports multiple tabs, allowing users to work on multiple sketches simultaneously. It also provides basic editing features such as cut, copy, paste, and find/replace.
- Library Management: The Arduino IDE includes a library manager that allows users to easily install and manage libraries (collections of pre-written code) from the Arduino Library Manager or from third-party sources. Libraries provide additional functionality and support for various sensors, actuators, and communication protocols.
- Board Manager: The IDE includes a board manager that allows users to select the Arduino board they are using and install the necessary board definitions and drivers. This enables users to program a wide range of Arduino-compatible microcontroller boards, including the popular Arduino Uno, Arduino Mega, and Arduino Nano.
- Serial Monitor: The IDE includes a serial monitor tool that allows users to communicate with their Arduino board via the serial port. This tool enables users to send and receive data to and from the Arduino board, making it useful for debugging and troubleshooting purposes.
- Compilation and Upload: The Arduino IDE compiles Arduino sketches into machine-readable binary files (HEX files) and uploads them to the Arduino board via a USB connection. The IDE handles the compilation and

uploading process automatically, making it easy for users to test and deploy their code.

- Open Source and Extensible: The Arduino IDE is open-source software, meaning that its source code is freely available for modification and redistribution. Additionally, the IDE is extensible, allowing users to create custom plugins and extensions to enhance its functionality and tailor it to their specific needs.

Overall, the Arduino IDE provides a beginner-friendly and versatile environment for programming Arduino microcontroller boards. Its simplicity, ease of use, and extensive features make it a popular choice among hobbyists, students, educators, and professionals for prototyping, experimentation, and developing projects involving embedded systems and electronics.

# CHAPTER – 6
# HARDWARE COMPONENTS

## 6.1 ESP32 MICROCONTROLLER

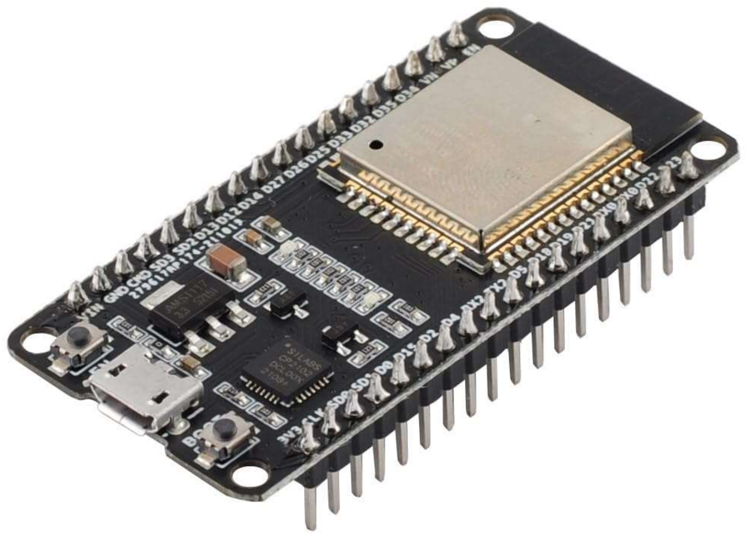


**Fig 6.1 ESP32 Microcontroller**

The ESP32 is a versatile microcontroller chip developed by Espressif Systems, which has gained widespread popularity in the IoT (Internet of Things) and embedded systems communities due to its powerful features and low cost. Here's an overview of the ESP32:

- Dual-Core Processor: The ESP32 features a dual-core Xtensa LX6 microprocessor, which enables multitasking and parallel processing. This allows the chip to handle multiple tasks simultaneously, making it suitable for applications that require real-time processing and communication.
- Wi-Fi and Bluetooth Connectivity: One of the key features of the ESP32 is its built-in Wi-Fi and Bluetooth connectivity. It supports both 2.4 GHz Wi-Fi (802.11b/g/n) and Bluetooth (Bluetooth Classic and BLE), allowing

devices to connect to Wi-Fi networks, communicate with other devices over Bluetooth, and create mesh networks.

- Low Power Consumption: The ESP32 is designed to operate with low power consumption, making it suitable for battery-powered and energy-efficient applications. It includes various power-saving features, such as multiple sleep modes, dynamic power scaling, and wake-up sources, to minimize power consumption and prolong battery life.
- Rich Peripheral Set: The ESP32 features a rich set of peripheral interfaces, including GPIO (General Purpose Input/Output) pins, UART (Universal Asynchronous Receiver/Transmitter), SPI (Serial Peripheral Interface), I2C (Inter-Integrated Circuit), ADC (Analog-to-Digital Converter), DAC (Digital-to-Analog Converter), PWM (Pulse Width Modulation), and more. These peripherals enable interfacing with sensors, actuators, displays, and other external devices.
- Integrated Security Features: Security is a priority in IoT applications, and the ESP32 includes built-in security features to protect data and communications. It supports hardware-accelerated cryptographic algorithms, secure boot, secure storage, and secure connections over Wi-Fi and Bluetooth.
- Rich Development Ecosystem: The ESP32 has a vibrant development ecosystem with comprehensive documentation, tutorials, and community support. Espressif provides the ESP-IDF (ESP32 IoT Development Framework), which is the official development framework for programming the ESP32 in C/C++. Additionally, there are numerous third-party libraries, frameworks, and development boards available for the ESP32, making it easy to get started and build projects.
- Arduino Compatibility: The ESP32 is compatible with the Arduino IDE, allowing users to program it using the familiar Arduino programming language and development environment. This simplifies the development process for those already familiar with Arduino and opens up a vast ecosystem of existing libraries and projects.

- Customizable and Extensible: The ESP32 is highly customizable and extensible, allowing developers to tailor it to their specific requirements. It supports over-the-air (OTA) updates, allowing firmware updates to be delivered remotely. Additionally, the chip's modular architecture enables the integration of custom firmware and applications.

Overall, the ESP32 is a powerful and versatile microcontroller chip with built-in Wi-Fi and Bluetooth connectivity, low power consumption, rich peripheral interfaces, integrated security features, and a vibrant development ecosystem. It is well-suited for a wide range of IoT, connected devices, and embedded systems applications, including home automation, smart wearables, industrial automation, and more.

Specifications:

- Integrated Crystal− 40 MHz
- Module Interfaces− UART, SPI, I2C, PWM, ADC, DAC, GPIO, pulse
- counter, capacitive touch sensor
- Integrated SPI flash− 4 MB
- ROM− 448 KB (for booting and core functions)
- SRAM− 520 KB
- Integrated Connectivity Protocols− WiFi, Bluetooth, BLE
- On−chip sensor− Hall sensor
- Operating temperature range− −40 − 85 degrees Celsius
- Operating Voltage− 3.3V
- Operating Current− 80 mA (average)

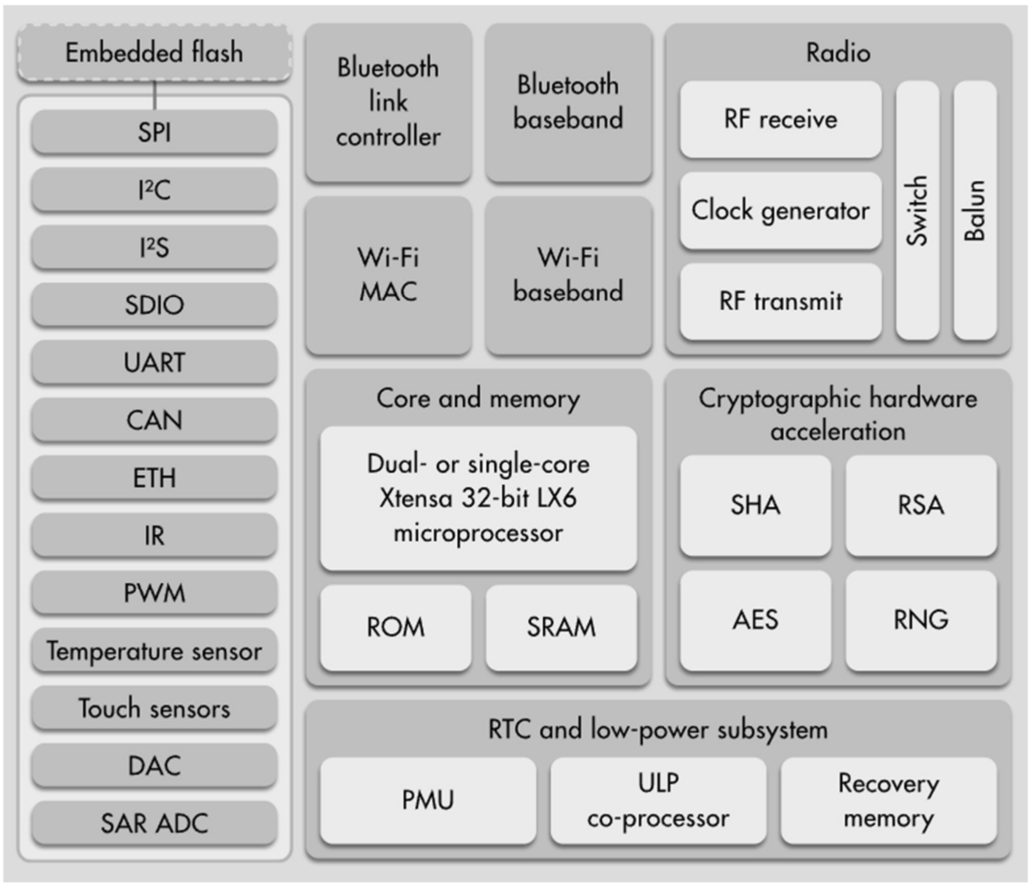


**Fig 6.2 ESP32 Functional block diagram**

## 6.2 POWER SUPPLY MODULE

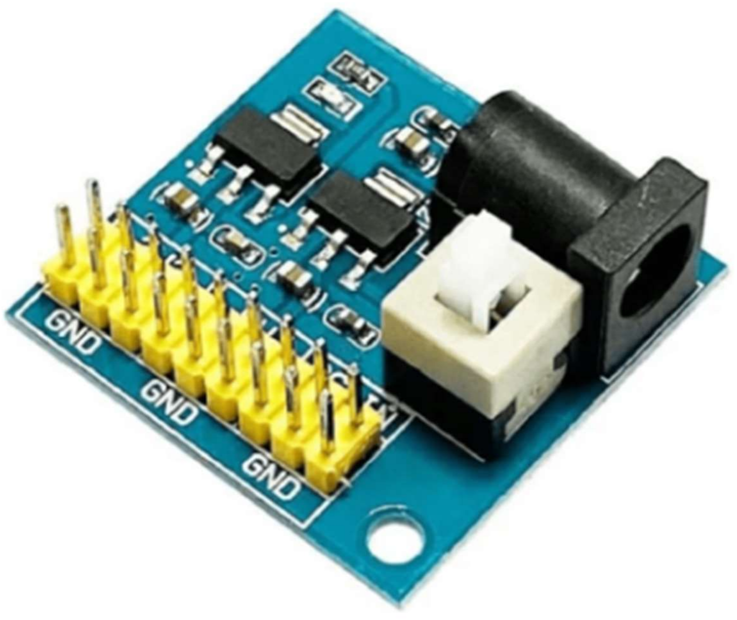


**Fig 6.3 Power Supply Module**

A power supply module with pins,is a compact electronic component designed to provide a stable output voltage or current to power other electronic devices or circuits. These modules typically come with pins or connectors that allow for easy integration into electronic projects without the need for soldering. Here's an overview of the features and components commonly found in a power supply module with pins:

- Input Voltage: Power supply modules usually accept input voltage from an external power source, such as a wall adapter, battery, or USB port. The input voltage range varies depending on the specific module but is typically specified in the module's datasheet.
- Voltage Regulation: Power supply modules incorporate voltage regulation circuitry to maintain a stable output voltage regardless of changes in the input voltage or load conditions. This ensures that the connected devices receive a consistent and reliable power supply.
- Output Voltage: The output voltage of the power supply module is the voltage provided to the load or the circuit being powered. Common output voltage options include fixed voltages (e.g., 3.3V, 5V, 12V) or adjustable voltages that can be set within a specified range using onboard potentiometers or jumpers.
- Output Current: The maximum output current of the power supply module determines the amount of current that can be drawn from the module to power the connected devices or circuits. It is important to ensure that the output current rating of the module meets the requirements of the load.
- Protection Features: Many power supply modules include built-in protection features to safeguard against overvoltage, overcurrent, short circuits, and overheating. These protection mechanisms help prevent damage to the module and connected devices in case of unexpected events or faults.
- Connectors or Pins: Power supply modules often come with pins or connectors that simplify the integration of the module into electronic

projects. These connectors may include header pins, screw terminals, or other types of connectors that allow for easy connection to external circuits or devices.

- Mounting Holes: Some power supply modules include mounting holes or mounting tabs that enable secure attachment to a PCB (printed circuit board) or enclosure. This ensures mechanical stability and ease of installation in electronic projects.
- LED Indicators: Some power supply modules feature LED indicators to provide visual feedback on the status of the module, such as power on/off, output voltage, or fault conditions. These indicators help users monitor the operation of the module and diagnose potential issues

## 6.3 LED LIGHTS

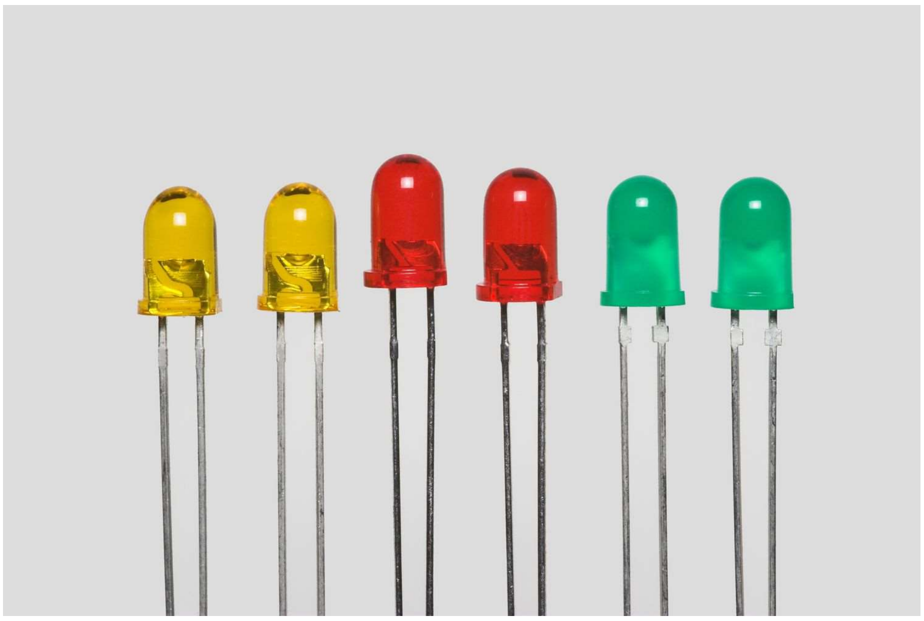


**Fig 6.4 LED**

LED stands for Light Emitting Diode. It is a semiconductor device that emits light when an electric current passes through it. LEDs are widely used in various applications due to their energy efficiency, longevity, compact size, and durability.

## 6.4 CIRCUIT DIAGRAM

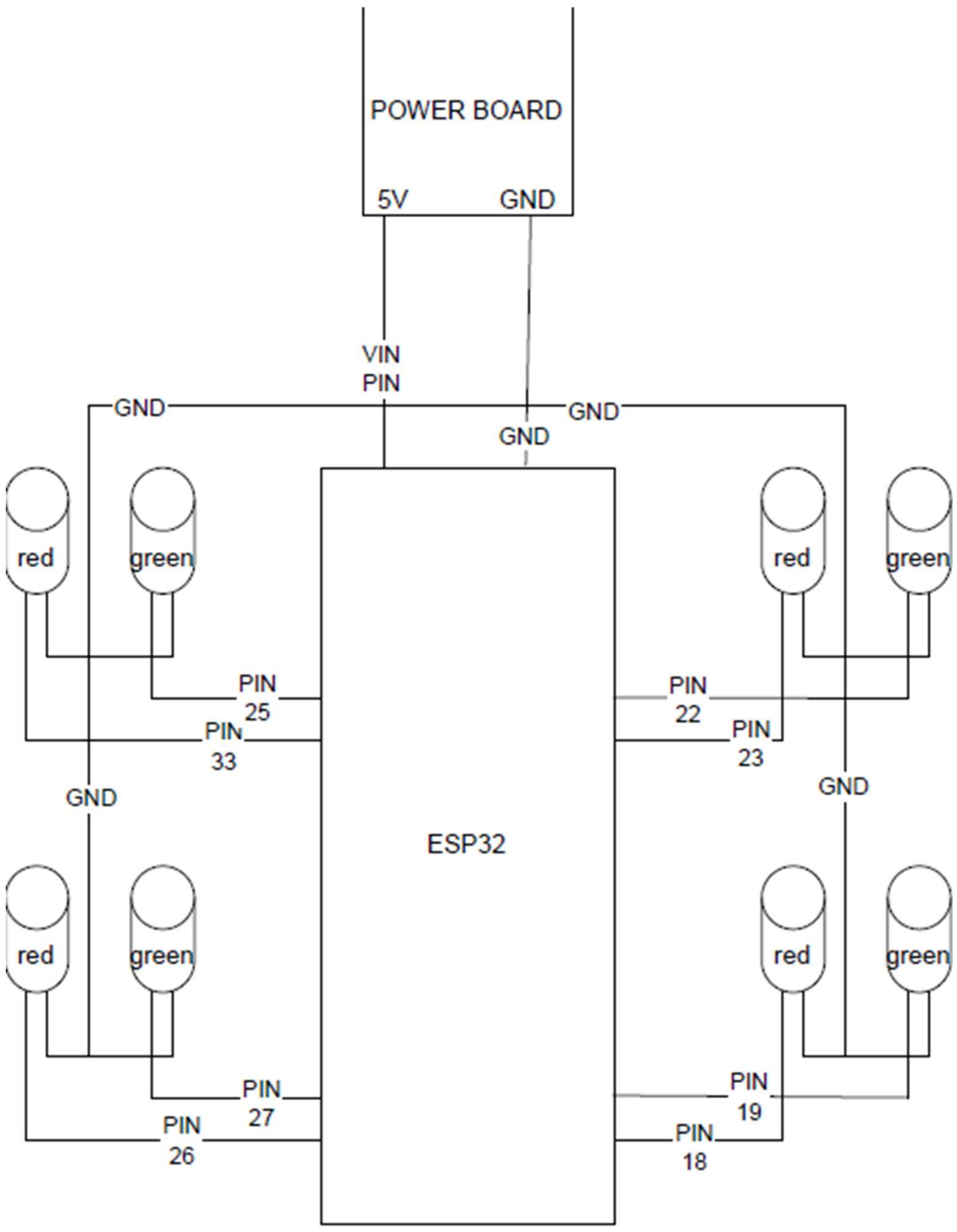


**Fig 6.5 Circuit Diagram**

# CHAPTER – 7

# RESULTS AND DESCRIPTION

Model deployment is the process of making your trained machine learning model accessible for use in real-world applications. In Traffic management system utilizing YOLO object detection, model deployment involves making the trained YOLO model available to process incoming traffic data and provide real-time object detection outputs.

Our model can identify any person and various types of vehicles and can be used for finding vehicle or person count in the respective lane.

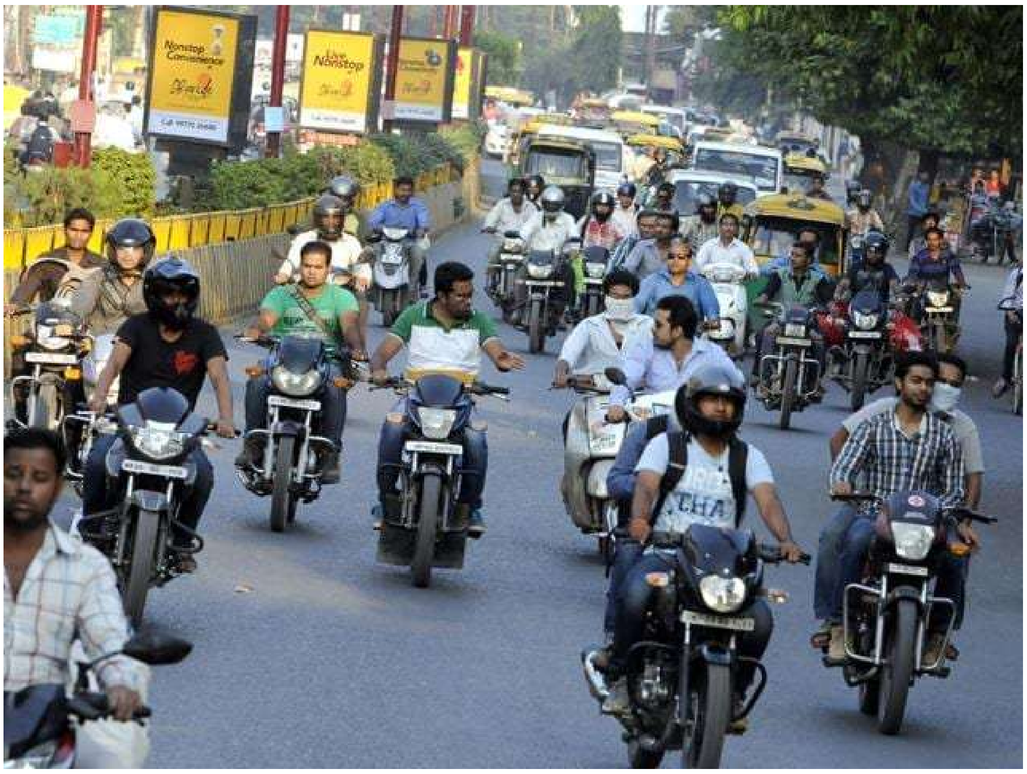


**Fig 7.1 Sample Image**

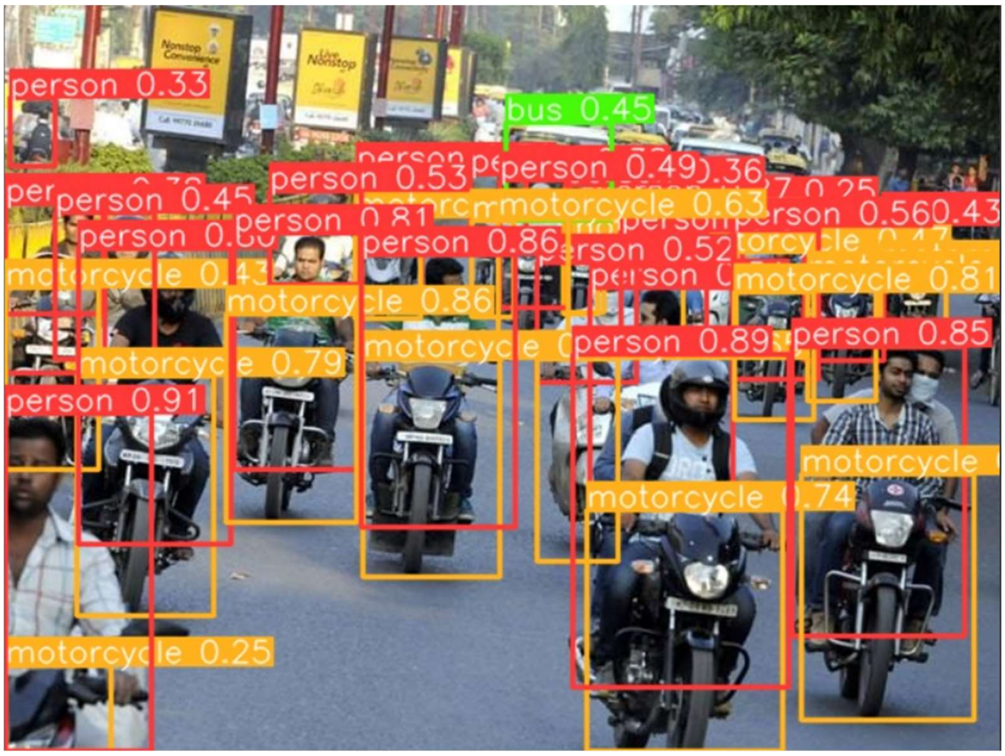
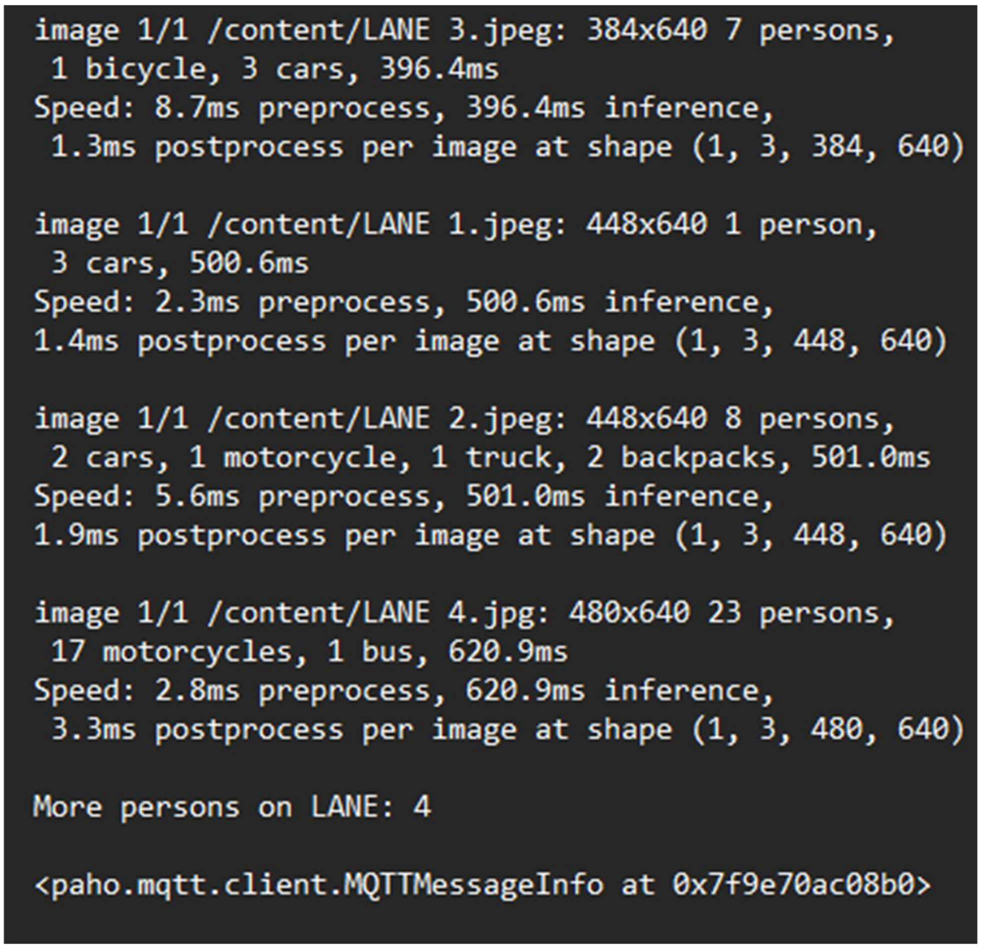


**Fig 7.2 Model Deployment**



```
image 1/1 /content/LANE 3.jpeg: 384x640 7 persons,
 1 bicycle, 3 cars, 396.4ms
Speed: 8.7ms preprocess, 396.4ms inference,
 1.3ms postprocess per image at shape (1, 3, 384, 640)

image 1/1 /content/LANE 1.jpeg: 448x640 1 person,
 3 cars, 500.6ms
Speed: 2.3ms preprocess, 500.6ms inference,
1.4ms postprocess per image at shape (1, 3, 448, 640)

image 1/1 /content/LANE 2.jpeg: 448x640 8 persons,
 2 cars, 1 motorcycle, 1 truck, 2 backpacks, 501.0ms
Speed: 5.6ms preprocess, 501.0ms inference,
1.9ms postprocess per image at shape (1, 3, 448, 640)

image 1/1 /content/LANE 4.jpg: 480x640 23 persons,
 17 motorcycles, 1 bus, 620.9ms
Speed: 2.8ms preprocess, 620.9ms inference,
 3.3ms postprocess per image at shape (1, 3, 480, 640)

More persons on LANE: 4

<paho.mqtt.client.MQTTMessageInfo at 0x7f9e70ac08b0>
```

**Fig 7.3 Model output**

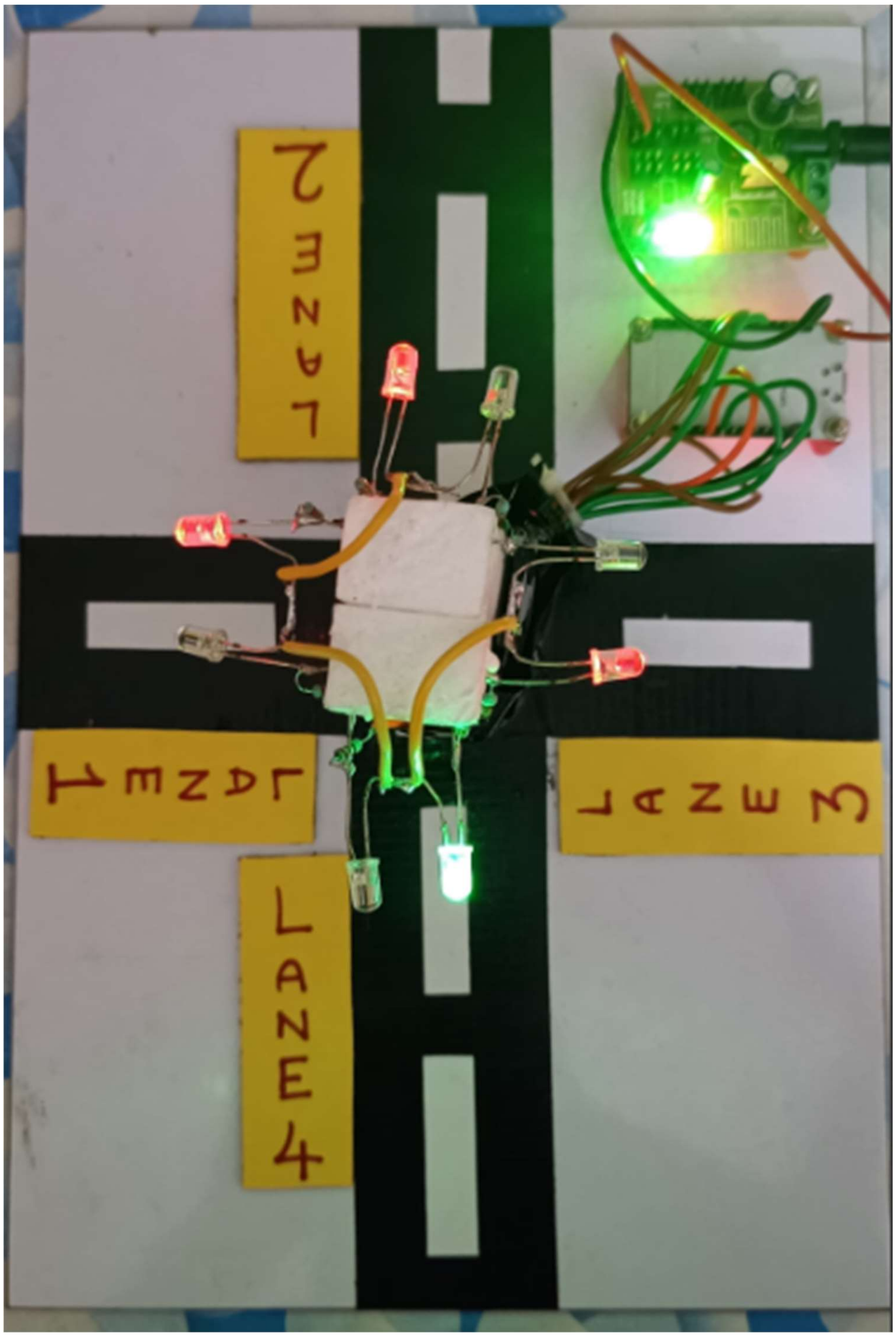


**Fig 7.4 Hardware Output**

## CONCLUSION

In conclusion, the integration of YOLO machine learning with ESP32 microcontrollers and LED lights offers a promising approach to developing a traffic management system. By leveraging YOLO's real-time object detection capabilities, the system can accurately identify vehicles, pedestrians, and other objects in the traffic environment. The ESP32's capabilities allow for seamless integration with the YOLO model and enable the control of LED lights to communicate traffic signals effectively.

This system has the potential to enhance road safety and efficiency by providing timely alerts and signals to drivers and pedestrians. Whether it's signaling drivers to stop for pedestrians, alerting to red-light violations, or displaying traditional traffic signals, the combination of machine learning and IoT hardware offers a versatile and adaptable solution.

However, it's important to acknowledge that the development of such a system requires careful consideration of factors such as model accuracy, real-time processing requirements, hardware limitations, and deployment challenges. Testing, optimization, and ongoing maintenance are crucial to ensure the system's reliability and effectiveness in real-world traffic scenarios.

Overall, the integration of YOLO machine learning with ESP32 microcontrollers and LED lights holds great promise for creating innovative and impactful traffic management solutions that prioritize safety and efficiency on the roads.

## REFERENCES

1. Sabeen Javaid; Ali Sufian; Saima Pervaiz; Mehak Tanveer Smart traffic management system using Internet of Things. 10.23919/ICACT.2018.8323770
2. Smart Traffic Management System Jay Mehta, Pratik Chavatekar, Snehasish Das, Kaveri Sawant Vol 7
3. Rachana K P, Aravind R, Ranjitha M, Spoorthi Jwanita, Soumya K, 2021, IOT Based Smart Traffic Management System
4. M. Sarrab, S.Pulparmbil, N. Kraiem, M. Al-Badawi Real-time traffic monitoring systems based on magnetic sensor integration International Conference on Smart City and Informatization (2019).
5. A. Kadar Muhammad Masum, M. Kalim Amzad Chy, I. Rahman, M. Nazim Uddin and K. Islam Azam, "An Internet of Things (IoT) based Smart Traffic Management System: A Context of Bangladesh," 2018 International Conference on Innovations in Science, Engineering and Technology (ICISET), 2018.
6. Dipak K Dash, TNN May 31, 2012. ―India loses Rs 60,000 crore due to traffic congestion: Study‖. Times Of India. http://articles.timesofindia.indiatimes.com/2012-05-31/india/31920307_1_toll-plazas-road-space-stoppage
7. Zantalis, F., Koulouras, G., Karabetsos, S., & Kandris, D. (2019). A review of machine learning and IoT in smart transportation. Future Internet, 11(4), 94.
8. Sarmiento, J. M., Gogineni, A., Bernstein, J. N., Lee, C., Lineen, E. B., Pust, G. D., & Byers, P. M. (2020).Alcohol/illicit substance use in fatal motorcycle crashes. Journal of surgical research, 256, 243-250.
9. Alsrehin, N. O., Klaib, A. F., & Magableh, A. (2019). Intelligent transportation and control systems using data mining and machine learning techniques: A comprehensive study. IEEE Access, 7, 49830-49857.

10. Brown, M. E., Rizzuto, T., & Singh, P. (2019). Strategic compatibility, collaboration and collective impact for community change. Leadership & Organization Development Journal.
11. Haghighat, A. K., Ravichandra-Mouli, V., Chakraborty, P., Esfandiari, Y., Arabi, S., & Sharma, A. (2020). Applications of deep learning in intelligent transportation systems. Journal of Big Data Analytics in Transportation, 2, 115-145.
12. Sprague-Jones, J., Singh, P., Rousseau, M., Counts, J., & Firman, C. (2020). The Protective Factors Survey: Establishing validity and reliability of a self-report measure of protective factors against child maltreatment. Children and Youth Services Review, 111, 104868.
13. Gangwani, D., & Gangwani, P. (2021). Applications of machine learning and artificial intelligence in intelligent transportation system: A review.
14. Applications of Artificial Intelligence and Machine Learning: Select Proceedings of ICAAAIML 2020, 203-216.
15. Reddy Sadashiva Reddy, R., Reis, I. M., & Kwon, D. (2020). ABCMETAapp: R Shiny Application for Simulation-based Estimation of Mean and Standard Deviation for Meta-analysis via Approximate Bayesian Computation (ABC). arXiv e-prints, arXiv-2004.

## APPENDIX

## HARDWARE CODE:

```
#include <WiFi.h>
#include <PubSubClient.h>


const char* ssid = "project";
const char* password = "project@123";
const char* mqtt_server = "103.217.220.20";
const int mqtt_port = 1883;
const char* mqtt_topic = "traffic/lane";


int lane = 0; // Default lane


const int redPins[] = {33, 26, 23, 19};   // Red LED pins for lanes
const int greenPins[] = {25, 27, 22, 18};  // Green LED pins for lanes


WiFiClient espClient;
PubSubClient client(espClient);


void setup_wifi() {
  delay(10);
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived in topic: ");
  Serial.println(topic);
  Serial.print("Message:");

  for (int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.println();
  // Convert payload to string and update the lane
  String message;
  for (int i = 0; i < length; i++) {
    message += (char)payload[i];
  }
  lane = message.toInt();
}

void reconnect() {
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    if (client.connect("ESP32Client")) {
      Serial.println("connected");
      client.subscribe(mqtt_topic);
    } else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      delay(5000);
    }
  }
}

void setup() {
  for (int i = 0; i < 4; i++) {
    pinMode(redPins[i], OUTPUT);
    pinMode(greenPins[i], OUTPUT);
    // Initially, all lanes have red LEDs on
    digitalWrite(redPins[i], HIGH);
    digitalWrite(greenPins[i], LOW);
  }

  Serial.begin(115200);
  setup_wifi();
  client.setServer(mqtt_server, mqtt_port);
  client.setCallback(callback);
}

void loop() {
  if (!client.connected()) {
    reconnect();
  }
  client.loop();

  // Check which lane is selected and update the LEDs accordingly
  switch (lane) {
    case 1:
      updateLights(greenPins[0], redPins[0]);
      break;

    case 2:
      updateLights(greenPins[1], redPins[1]);
      break;

    case 3:
      updateLights(greenPins[2], redPins[2]);
      break;

    case 4:
      updateLights(greenPins[3], redPins[3]);
      break;

    default:
      // All LEDs are red if no lane is selected
      updateLights(LOW, HIGH);
  }
}

// Function to update the LEDs based on the selected lane
void updateLights(int greenPin, int redPin) {
  for (int i = 0; i < 4; i++) {
    digitalWrite(greenPins[i], LOW);
    digitalWrite(redPins[i], HIGH);
  }

  digitalWrite(greenPin, HIGH);
  digitalWrite(redPin, LOW);
}
```

**SOFTWARE CODE**

```
!pip install ultralytics

!pip install paho.mqtt==1.6

from ultralytics import YOLO


# Build a YOLOv9c model from pretrained weight

model = YOLO('yolov8s.pt')


# Display model information (optional)

model.info()

import os


# Run inference with the YOLOv8 model on the '.jpg' image

# Note: Replace '/content/LANE 4.jpg' with the path to your image

results = model('/content/LANE 4.jpg')
```

```
# Create a directory to store the prediction results if it doesn't exist
if not os.path.exists('pred'):
    os.mkdir('pred')


# Save the inference results (bounding boxes, labels, etc.) to an image file
results[0].save(filename='pred/result.jpg')


# Display the saved image using IPython display
from IPython.display import display, Image
display(Image(filename='pred/result.jpg'))
import paho.mqtt.client as mqtt


# Specify the MQTT broker address
broker_address = "103.217.220.20"


# Create an MQTT client instance
client = mqtt.Client("MyClient")


# Connect to the MQTT broker
```

```
client.connect(broker_address, 1883)

1# Import necessary libraries

from imutils import paths


# Use imutils to list all image paths in the current directory

image_paths = list(paths.list_images('.'))


# Initialize variables

count_init = 0


# Iterate over each image path

for image in image_paths:

    if not 'result.jpg' in image:

        # Perform object detection using the 'model' (assumed to be a pre-trained
model)

        results = model(image)


        # Extract class labels from the results

        my_list = results[0].boxes.cls.tolist()
        # Count the occurrences of class label '0.0' (assuming it represents a specific
class, e.g., lane)

        count = my_list.count(0.0)


          # Update the maximum count and corresponding image path if a higher
count is found

        if count > count_init:

            count_init = count

            img = image

# Extract the lane information from the image path (assuming a specific naming
convention)

lane = img.split()[-1][0]


print(f'\nMore persons on LANE: {lane}\n')

# Publish the lane information to an MQTT topic named "traffic/lane"

client.publish("traffic/lane", str(lane))
```