****

Faculty of Engineering

Final Year Project Report

CCE Department – Fifth year – English and French Sections

**Autonomous Vehicle**

**Authored By**

Ahmad Salemeh, 65036

Ali Kassas, 65650

**Supervised By**

Dr. Ali Abu-Melhem

**March 2024**

# Acknowledgement

The satisfaction and happiness that accompany the successful completion of any task would be incomplete without the mention of the people, who made it possible, whose constant guidance and encouragement aided us in the completion of our project.

We consider it our privilege to express voice of gratitude and respect to all those who guided and inspired us in the completion of this project.

We extend our gratitude to our project supervisor Dr. Ali abo-Melhem, who took keen interest in our project and guided us throughout the project by providing all the necessary knowledge and who without his support, the project would have been nowhere near completion.

We would never forget to thank the professors who taught us throughout our five years, you were all the reason behind us reaching here and we owe you a lot.

We would like to express our appreciation to our dean Prof. Ahmad Ghandour and the head of department Dr. Kassem Hamze for their support.

Finally, yet importantly, we are deeply in debts to our parents for their inspiration and encouragement, which enabled pursue our studies.

# TABLE OF CONTENTS

[Acknowledgement 2](#_Toc172978402)

[TABLE OF CONTENTS 3](#_Toc172978403)

[Table of Figures 5](#_Toc172978404)

[Abstract 6](#_Toc172978405)

[Requirements and Deliverables 7](#_Toc172978406)

[Requirements 7](#_Toc172978407)

[Deliverables 7](#_Toc172978408)

[Chapter 1: Introduction 8](#_Toc172978409)

[1.1 What is a self-driving car? 8](#_Toc172978410)

[1.2 How famous self-driving cars work 8](#_Toc172978411)

[1.2.1 Google's autonomous car: Waymo 8](#_Toc172978412)

[1.2.2 Tesla's autonomous car: Tesla Model S, X, Y & 3 9](#_Toc172978413)

[Which approach is better for our project? 9](#_Toc172978414)

[1.3 Technologies Behind Automation: 9](#_Toc172978415)

[1.3.1 Sensors 9](#_Toc172978416)

[1.3.2 Machine Learning and Deep Learning 10](#_Toc172978417)

[1.4 Pros and cons of autonomous cars 10](#_Toc172978418)

[1.4.1 Pros of autonomous cars: 10](#_Toc172978419)

[1.4.2 Cons of autonomous cars: 11](#_Toc172978420)

[Chapter 2: Literature Review 12](#_Toc172978421)

[2.1 Introduction to Machine Learning and Deep Learning 12](#_Toc172978422)

[2.2 Types of Machine Learning Models 12](#_Toc172978423)

[2.3 Deep Learning Models for Autonomous Driving 13](#_Toc172978424)

[2.3.1 Convolutional Neural Networks (CNNs) 13](#_Toc172978425)

[2.3.2 Recurrent Neural Networks (RNNs) and Long Short-Term Memory Networks (LSTMs) 18](#_Toc172978426)

[2.3.3 Generative Adversarial Networks (GANs) 18](#_Toc172978427)

[2.3.4 Which model best suits our project? 18](#_Toc172978428)

[2.4 Sensor Fusion and Perception 19](#_Toc172978429)

[2.5 State-of-the-Art Approaches and Technologies 20](#_Toc172978430)

[2.6 Challenges and Limitations 20](#_Toc172978431)

[2.7 Gap Identification and Future Directions 21](#_Toc172978432)

[Chapter 3: Methodology 23](#_Toc172978433)

[3.1 Introduction 23](#_Toc172978434)

[3.2 System Architecture 23](#_Toc172978435)

[3.3 Hardware Setup 24](#_Toc172978436)

[3.3.1 Hardware Components 24](#_Toc172978437)

[3.3.2 Hardware Implementation and connections 29](#_Toc172978438)

[3.4 Software Development 34](#_Toc172978439)

[3.4.1 Environmental setups and configuration 35](#_Toc172978440)

[3.4.2 Modules Used in the Project 41](#_Toc172978441)

[3.3.4 Training 51](#_Toc172978442)

[ **Dataset Gathering Process** 52](#_Toc172978443)

[ **Importance of Accuracy** 52](#_Toc172978444)

[ **Image Example** 53](#_Toc172978445)

[3.6 Model Evaluation 63](#_Toc172978446)

[3.7 Hardware and software Limitations 65](#_Toc172978447)

[3.7.1 Hardware Limitations 65](#_Toc172978448)

[3.7.2 Software Limitations 65](#_Toc172978449)

[3.8 Summary 66](#_Toc172978450)

[Chapter 4: Results and Discussions 67](#_Toc172978451)

[4.1 Model Architecture 67](#_Toc172978452)

[4.2 Training Process 68](#_Toc172978453)

[4.3 Performance metrics 69](#_Toc172978454)

[4.3.1 NVIDIA's Model 69](#_Toc172978455)

[4.3.2 Custom Model 69](#_Toc172978456)

[4.3.3 Interpretation 70](#_Toc172978457)

[4.3.4 Summary 70](#_Toc172978458)

[4.4 Discussion 70](#_Toc172978459)

[Chapter 5: Conclusion 72](#_Toc172978460)

[5.1 Summary of Achievements 72](#_Toc172978461)

[5.2 Limitations 72](#_Toc172978462)

[5.3 Assumptions and Impact 72](#_Toc172978463)

[5.4 Future Work 72](#_Toc172978464)

[5.5 Final Thoughts 73](#_Toc172978465)

# Table of Figures

[Figure 1- CNN model architecture example 13](#_Toc172978365)

[Figure 2- Car chassis 24](#_Toc172978366)

[Figure 3- motors & wheels 25](#_Toc172978367)

[Figure 4- Ultrasonic sensor 25](#_Toc172978368)

[Figure 5- raspberry pi camera v2 26](#_Toc172978369)

[Figure 6- Raspberry pi 26](#_Toc172978370)

[Figure 7- L298N motor driver 27](#_Toc172978371)

[Figure 8- Power Supplies and other components 28](#_Toc172978372)

[Figure 9- Road (Hard Papers with White Tapes) 28](#_Toc172978373)

[Figure 10- Component Connectivity Diagram 33](#_Toc172978374)

[Figure 11- car overview 34](#_Toc172978375)

[Figure 12- Data collection diagram 48](#_Toc172978376)

[Figure 13- Training diagram 51](#_Toc172978377)

[Figure 14- 'DataCollected' folder overview 53](file:///C:\Users\user\Downloads\FYP%20report%20(1)%20(1).docx#_Toc172978378)

[Figure 15- Images folder overview 53](#_Toc172978379)

[Figure 16- csv file overview 54](#_Toc172978380)

[Figure 17- Data import diagram 55](#_Toc172978381)

[Figure 18- Data balance and visualization 56](#_Toc172978382)

[Figure 19- preprocessing preparation diagram 57](#_Toc172978383)

[Figure 20- Data splitting diagram 58](#_Toc172978384)

[Figure 21- Data augmentation diagram 59](#_Toc172978385)

[Figure 22- Data preprocessing 59](#_Toc172978386)

[Figure 23- Model creation diagram 60](#_Toc172978387)

[Figure 24- training process overview 61](#_Toc172978388)

[Figure 25- Save model diagram 62](#_Toc172978389)

[Figure 26- Results plotting overview 62](#_Toc172978390)

[Figure 27- MSE equation 63](#_Toc172978391)

[Figure 28- MAE equation 64](#_Toc172978392)

[Figure 29- RMSE equation 64](#_Toc172978393)

[Figure 30- Workflow diagram 71](#_Toc172978394)

# Abstract

The self-driving robot car project aims to develop an autonomous navigation system using advanced deep learning algorithms and various sensors. This project focuses on integrating a camera to capture real-time images and ultrasonic sensors to measure distances and detect obstacles. The primary goal is to enable the robot car to navigate a predefined path accurately and safely, without human intervention, leveraging the latest advancements in artificial intelligence and sensor technology.

The system employs convolutional neural networks (CNNs) to process the camera images and predict the appropriate steering angles required for the car to stay on course. By analyzing the visual data, the model determines the best path forward, adjusting the steering to ensure the car remains within the designated lane. The ultrasonic sensors provide critical data for obstacle detection and avoidance, ensuring the car can navigate around obstacles smoothly. This dual-sensor approach enhances the reliability and robustness of the navigation system, allowing for real-time adjustments based on dynamic environmental conditions.

This report comprehensively covers the project's requirements, including the necessary hardware components such as cameras, ultrasonic sensors, and the robot chassis, as well as software tools like deep learning frameworks and programming languages. Detailed descriptions of the methodologies employed are provided, encompassing system design, hardware and software implementation, data collection, preprocessing, model training, and rigorous testing procedures.

The results section presents detailed performance metrics, showcasing the accuracy and effectiveness of the deep learning model in steering prediction and the obstacle avoidance system's reliability. Visualizations such as graphs, charts, and images illustrate the car’s performance during various test scenarios. The discussion section offers an in-depth analysis of these results, addressing technical challenges encountered during the project and proposing potential improvements for future iterations. The discussion also includes insights into the scalability and practical applications of the developed system.

In conclusion, this project demonstrates the successful integration of machine learning and sensor technologies in creating an autonomous self-driving robot car. The achievements highlight the potential for further advancements in the field of autonomous vehicles and intelligent transportation systems, contributing to safer and more efficient roadways. The implications of this work extend beyond the immediate application, offering a foundation for future research and development in autonomous navigation and related technologies.

# Requirements and Deliverables

Requirements**:**

* **Hardware:**
  + Camera: Essential for capturing road images.
  + Ultrasonic Sensor: Measures distance to detect and avoid obstacles.
  + Robot Chassis: Provides the structural framework.
  + Microcontroller/Processor: Executes the control algorithms.
* **Software:**
  + Deep Learning Frameworks: TensorFlow, Keras for model development.
  + Programming Languages: Python for coding the algorithms.
  + Data Processing Tools: Libraries for image and sensor data preprocessing.

Deliverables**:**

* A fully functional self-driving robot car capable of navigating a pre-defined path.
* Comprehensive documentation and source code for the entire system.
* Detailed testing and validation reports illustrating the car’s performance.

# Chapter 1: Introduction

Imagine a vehicle capable of autonomous navigation without human intervention. This project delves into the realm of self-driving cars, which hold the potential to enhance road safety, facilitate easier transportation, and alleviate traffic congestion. However, achieving flawless autonomous operation presents significant challenges. We must develop systems that enable these vehicles to accurately perceive their surroundings and make sound decisions. Furthermore, there are critical ethical and legal considerations to address, such as liability in case of malfunctions and public acceptance of autonomous vehicles. Although this project represents a small-scale prototype, it is an exciting step toward comprehending and potentially developing fully autonomous cars in the future.

## 1.1 What is a self-driving car?

A self-driving car (sometimes called an autonomous car or driverless car) is a vehicle that uses a combination of sensors, cameras, radar, and artificial intelligence (AI) to travel between destinations without a human operator. To qualify as fully autonomous, a vehicle must be able to navigate without human intervention to a predetermined destination over roads that have not been adapted for its use. These vehicles can navigate various environments, including city streets, highways, and rural roads, while adhering to traffic laws and safety regulations. Self-driving cars are typically categorized into different levels of automation, ranging from Level 0 (no automation) to Level 5 (full automation), based on the extent to which they can operate without human intervention.

## 1.2 How famous self-driving cars work

AI technologies power self-driving car systems. Developers of self-driving cars use vast amounts of data from image recognition systems, along with machine learning and neural networks, to build systems that can drive autonomously. The neural networks identify patterns in the data, which are fed to the machine learning algorithms. That data includes images from cameras on self-driving cars from which the neural network learns to identify traffic lights, trees, curbs, pedestrians, street signs, and other parts of any given driving environment.

### 1.2.1 Google's autonomous car: Waymo

Considering the two most famous autonomous cars as an example, Google's self-driving car project, called Waymo, uses a mix of sensors, LIDAR (light detection and ranging -- a technology similar to RADAR), radar, and cameras and combines all of the data those systems generate to identify everything around the vehicle and predict what those objects might do next. This happens in fractions of a second.

### 1.2.2 Tesla's autonomous car: Tesla Model S, X, Y & 3

In 2022, Tesla took the next step in Tesla Vision by removing ultrasonic sensors (USS) from Model 3 and Model Y for most global markets, followed by all Model S and Model X in 2023. This advanced approach relies on eight cameras and robust neural net processing to perceive the car's surroundings and provide Autopilot features. With today’s software, this approach gives Autopilot high-definition spatial positioning, longer range visibility, and the ability to identify and differentiate between objects.

## Which approach is better for our project?

After doing some research and watching some experimental videos regarding the efficiency of both cars, we found out that both cars are efficient and reliable at what they do, but the Waymo car seemed to be a lot more expensive to implement as a small hardware in our project due to the high cost of the radar and lidar sensors, which are considered the main components for the car to function properly. For that, we decided to take Tesla's approach since it only depends on cameras and camera vision to make decisions and controls.

## 1.3 Technologies Behind Automation:

### 1.3.1 Sensors

1. **LiDAR (Light Detection and Ranging)**

LiDAR sensors emit thousands of laser pulses per second and measure the time it takes for the light to return after hitting objects in the environment. This enables the creation of detailed 3D maps of the surroundings, including the distance, shape, and position of objects. LiDAR is particularly useful for detecting obstacles, mapping the terrain, and identifying lane boundaries.

1. **GPS (Global Positioning System)**

GPS receivers provide precise location and velocity information by receiving signals from satellites. While GPS is primarily used for navigation and determining the vehicle's position on a map, it can also be integrated with other sensors to enhance localization accuracy and provide additional context about the vehicle's surroundings.

1. **Ultrasonic Sensors**

Ultrasonic sensors emit high-frequency sound waves and measure the time it takes for the waves to bounce back after hitting objects. They are commonly used for short-range detection, such as detecting nearby vehicles, obstacles, or pedestrians during parking maneuvers or low-speed maneuvers in dense traffic.

1. **Cameras**

Cameras capture visual data from the vehicle's surroundings, including traffic signs, lane markings, traffic lights, and other vehicles. Advanced image processing algorithms analyze this data to identify and classify objects, recognize road signs, and interpret traffic signals.

1. **Radio Detection and Ranging (RADAR)**

RADARs are highly effective because they use radio waves instead of lasers to measure distances, so they work in any conditions. It’s important to understand that radars are noisy sensors. This means that even if the camera sees no obstacle, the radar will detect some obstacles. The RADAR data should be cleaned in order to make good decisions and predictions.

### 1.3.2 Machine Learning and Deep Learning

Advanced algorithms for processing sensor data, predictive modeling, and making autonomous driving decisions in self-driving cars. Machine learning is a subset of artificial intelligence that focuses on developing algorithms that allow computers to learn from and make predictions based on data. Deep learning, a further specialization within machine learning, involves neural networks with many layers that can analyze vast amounts of complex data, such as images and sensor readings, to make highly accurate predictions and decisions.

## 1.4 Pros and cons of autonomous cars

### 1.4.1 Pros of autonomous cars:

**1. Enhanced Safety:** Autonomous cars have the potential to significantly reduce accidents caused by human error, such as distracted driving, speeding, and impaired driving, leading to fewer injuries and fatalities on the roads.

**2. Increased Efficiency:** Autonomous cars can optimize routes, reduce traffic congestion, and improve fuel efficiency through smoother driving patterns, ultimately saving time and reducing environmental impact.

**3. Improved Accessibility:** Self-driving cars can provide increased mobility for people who are unable to drive due to age, disability, or other factors, offering greater independence and inclusivity.

**4. Enhanced Productivity**: With the ability to perform tasks while commuting, such as working, studying, or relaxing, passengers in autonomous cars can reclaim valuable time that would otherwise be spent solely on driving.

**5. Potential Cost Savings:** Autonomous cars may lead to reduced insurance premiums, lower fuel costs, and decreased need for car ownership through the proliferation of shared autonomous vehicle services.

### 1.4.2 Cons of autonomous cars:

**1. Technological Limitations:** Autonomous cars face challenges in reliably navigating complex and unpredictable environments, particularly in adverse weather conditions or unfamiliar situations.

**2. Ethical Dilemmas:** Self-driving cars raise ethical questions regarding decision-making in unavoidable accident scenarios, such as prioritizing the safety of occupants versus pedestrians, as well as issues related to data privacy and cybersecurity.

**3. Regulatory Hurdles:** The development and deployment of autonomous cars require comprehensive regulatory frameworks to address liability, safety standards, and infrastructure compatibility, which may lag behind technological advancements.

**4. Job Displacement:** The widespread adoption of autonomous vehicles could lead to job displacement in industries such as transportation, including taxi and truck drivers, potentially exacerbating socioeconomic inequalities.

**5. Public Acceptance and Trust:** Convincing the public to trust and embrace self-driving technology remains a significant challenge, as concerns regarding safety, reliability, and loss of control persist despite technological advancements and safety assurances.

# Chapter 2: Literature Review

## 2.1 Introduction to Machine Learning and Deep Learning

**Machine learning** (ML) is a branch of artificial intelligence (AI) focused on developing algorithms that allow computers to learn from and make decisions or predictions based on data, without explicit programming. ML encompasses various paradigms, including supervised learning, unsupervised learning, and reinforcement learning, each tailored to different types of learning tasks.

**Deep learning** (DL), a subset of ML, employs neural networks with multiple layers (deep networks) to learn hierarchical representations of data. DL has revolutionized AI by significantly improving the performance of tasks such as image and speech recognition, natural language processing, and autonomous driving. Its ability to automatically learn complex patterns from large amounts of data makes it particularly well-suited for sophisticated tasks in autonomous systems.

## 2.2 Types of Machine Learning Models

* **Supervised Learning**: Supervised learning algorithms learn from labeled data to make predictions or decisions. They involve training a model on input-output pairs and optimizing it to minimize prediction errors. In autonomous driving, supervised learning is used for tasks such as predicting steering angles based on labeled images or classifying objects in the environment.
* **Unsupervised Learning**: Unsupervised learning algorithms explore patterns in data without labeled outputs. They include clustering algorithms, anomaly detection methods, and dimensionality reduction techniques. In autonomous systems, unsupervised learning is crucial for tasks like grouping similar driving scenarios and identifying anomalies in sensor data.
* **Reinforcement Learning**: Reinforcement learning (RL) involves agents learning to make decisions through trial and error interactions with an environment. RL algorithms receive feedback in the form of rewards or penalties for their actions, enabling them to learn optimal strategies over time. In autonomous driving, RL is applied to learn driving policies and behaviors in complex and dynamic environments, optimizing tasks such as navigation and decision-making.

## 2.3 Deep Learning Models for Autonomous Driving

### 2.3.1 Convolutional Neural Networks (CNNs)

#### 2.3.1.1 Definition

Convolutional Neural Networks (CNNs) are a class of deep neural networks specifically designed for processing grid-like data, such as images and videos. They are characterized by their hierarchical architecture, consisting of convolutional layers, pooling layers, and fully connected layers. Each type of layer has a specific role and function, contributing to the overall effectiveness of CNNs in tasks like image recognition, object detection, and autonomous driving.

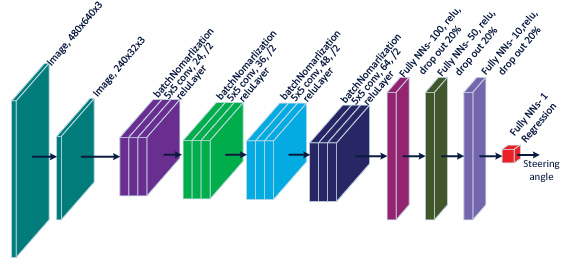


Figure 1- CNN model architecture example

**1.Convolutional Layers:**

Convolutional layers are the core building blocks of CNNs. These layers apply convolution operations to the input images, which involves sliding a filter or kernel over the input data to produce feature maps. Each filter is responsible for detecting specific features such as edges, textures, and patterns in the input image. By learning spatial hierarchies of features, convolutional layers enable CNNs to localize and recognize objects in complex scenes. This ability is essential for tasks like object detection, lane detection, and other image processing tasks in autonomous vehicles.

* **Role:** Extracting and learning features from the input data.
* **Summary:** A convolutional layer uses a set of learnable filters to scan the input image and produce feature maps highlighting different aspects of the image.

**2. Pooling Layers:**

Pooling layers, also known as subsampling or downsampling layers, follow convolutional layers and serve to reduce the spatial dimensions of the feature maps. This reduction is achieved through operations such as max pooling or average pooling. By decreasing the spatial resolution, pooling layers retain the most important information while discarding less relevant details. This process improves the model's robustness to variations in the input data, such as changes in position or scale, and reduces computational complexity, making the model more efficient.

* **Role:** Reducing spatial dimensions and computational complexity, while retaining essential information.
* **Summary:** A pooling layer performs a downsampling operation on the feature maps, typically using max pooling (taking the maximum value) or average pooling (taking the average value) within a defined window.

**3.Fully Connected Layers:**

Fully connected layers, also known as dense layers, are typically found towards the end of the CNN architecture. These layers take the high-level features extracted by the convolutional and pooling layers and integrate them to make final predictions or classifications. Each neuron in a fully connected layer is connected to every neuron in the previous layer, allowing for a comprehensive combination of features to form the final output. In autonomous driving applications, fully connected layers play a pivotal role in real-time processing of sensor data, enabling accurate perception and decision-making on the road.

* **Role:** Integrating features and making final predictions or classifications.
* **Summary:** A fully connected layer connects every neuron from the previous layer to every neuron in the current layer, combining all features to produce the final output.

#### 2.3.1.2 Expanded Overview of CNNs in Autonomous Driving

CNNs are instrumental in autonomous driving, where they process vast amounts of visual data from cameras and other sensors to understand the vehicle's environment. Here's how each layer contributes to this process:

**1. Convolutional Layers:** These layers detect and learn crucial features from road images, such as lane markings, road signs, vehicles, and pedestrians. By stacking multiple convolutional layers, the network can build a hierarchical understanding of the scene, from simple edges to complex objects.

**2. Pooling Layers:** After convolutional layers extract features, pooling layers reduce the dimensionality of these feature maps, preserving important information while discarding less critical details. This dimensionality reduction helps the model focus on the most significant features and improves its robustness to variations like shifts or distortions in the input data.

**3. Fully Connected Layers:** Finally, the fully connected layers take the high-level, abstracted features and use them to make predictions, such as determining the steering angle or recognizing traffic signs. These layers integrate all the information gathered by previous layers to form a comprehensive understanding of the scene, enabling the vehicle to make informed driving decisions.

#### 2.3.1.3 Introduction to Top CNN Models for Autonomous Driving Projects

In this section, we will introduce and analyze the top three convolutional neural network (CNN) models commonly employed in autonomous driving projects. We will provide a detailed definition of each model, compare their characteristics and performance, and explain the rationale behind our choice of the optimal model for predicting steering angles.

**1.NVIDIA's Model:** NVIDIA's model is a relatively simple yet effective convolutional neural network (CNN) designed for end-to-end autonomous driving. It consists of five convolutional layers followed by three fully connected layers. The model processes raw images directly from the vehicle's cameras and outputs steering angles. It is designed to generalize well across different driving environments with minimal preprocessing. The architecture is efficient, providing a good balance between complexity and performance, making it suitable for real-time applications.

**2.Comma.ai's Model:** Comma.ai's model is an advanced CNN architecture that aims to enhance robustness and accuracy. It includes additional convolutional layers and batch normalization layers. The extra layers allow the model to capture more intricate details from the images, improving its performance on complex road conditions. Batch normalization helps in stabilizing the training process and enables higher learning rates, leading to faster convergence and better generalization. This model is known for its high accuracy in predicting steering angles, particularly in challenging driving scenarios.

**3.Udacity's Model:** Udacity's model incorporates dropout layers to handle overfitting, a common issue in neural networks. The architecture includes multiple convolutional layers followed by fully connected layers, similar to NVIDIA's model but with added dropout layers between them. These dropout layers randomly deactivate a fraction of neurons during training, which helps in preventing the network from relying too much on any single feature. This results in better generalization to different road types and conditions. The model has been tested and proven to perform well in various driving environments, making it versatile and reliable.

#### 2.3.1.4 Comparison of Leading CNN Models

When evaluating the top CNN models for autonomous driving, each presents unique strengths tailored to different aspects of performance and robustness. **NVIDIA's model** stands out for its simplicity and efficiency, featuring a streamlined architecture that balances complexity and computational demands. This model excels in real-time processing with minimal computational overhead, making it highly suitable for practical implementation in autonomous vehicles. In contrast, **Comma.ai's model** introduces additional convolutional layers and batch normalization, enhancing its ability to capture intricate details and improving its robustness to varying road conditions. This increased complexity allows for greater accuracy, particularly in challenging driving scenarios. Meanwhile, **Udacity's model** incorporates dropout layers to combat overfitting, providing enhanced generalization across diverse driving environments. This feature ensures that the model performs reliably in various road types and conditions. While Comma.ai’s model offers superior accuracy and Udacity’s model addresses overfitting, NVIDIA’s model provides an optimal balance of efficiency and performance, making it a compelling choice for real-time autonomous driving applications.

#### 2.3.1.5 Conclusion: Why NVIDIA's Model is the Best for Predicting Steering Angle

While each of the discussed CNN models has its strengths, NVIDIA's model stands out as the optimal choice for predicting steering angles in autonomous driving. Its simplicity and efficiency ensure that it can be implemented in real-time systems with limited computational resources, which is crucial for practical applications. The model's architecture is designed to balance complexity and performance, allowing it to generalize well across different driving environments without the need for extensive preprocessing. This makes it highly suitable for end-to-end autonomous driving solutions, where quick and accurate perception and decision-making are paramount. Although Comma.ai's and Udacity's models offer enhancements in robustness and versatility, respectively, NVIDIA's model provides an optimal balance that meets the demands of real-time, on-road performance.

model = Sequential()  
model.add(Conv2D(24, (5, 5), (2, 2), input\_shape=(66, 200, 3), activation='elu'))  
model.add(Conv2D(36, (5, 5), strides=(2, 2), activation='elu'))  
model.add(Conv2D(48, (5, 5), strides=(2, 2), activation='elu'))  
model.add(Conv2D(64, (3, 3), activation='elu'))  
model.add(Conv2D(64, (3, 3), activation='elu'))  
model.add(Flatten())  
model.add(Dense(100, activation='elu'))  
model.add(Dense(50, activation='elu'))  
model.add(Dense(10, activation='elu'))  
model.add(Dense(1)) # Regression output for steering angle  
  
model.compile(optimizer=Adam(learning\_rate=0.0001), loss='mean\_squared\_error')  
return model

#### 2.3.1.6 Alternative custom model

**Characteristics:**

1. **Architecture**:
   * **Convolutional Layers**: Two convolutional layers with 8 filters each and a 3x3 kernel size. These layers help in detecting features such as edges and textures in the input images.
   * **MaxPooling**: Applied after each convolutional layer, pooling reduces the spatial dimensions, helping in feature extraction and computational efficiency.
   * **Dropout**: A dropout rate of 0.5 is used after the convolutional layers to prevent overfitting by randomly dropping units during training.
2. **Activation Functions**:
   * **ReLU**: The activation function used in convolutional and dense layers is ReLU, which introduces non-linearity and helps the network learn complex patterns.
3. **Dense Layers**:
   * **Dense Layer**: A dense layer with 50 units and ReLU activation follows the convolutional and pooling layers. This layer helps in making predictions based on the features extracted by the convolutional layers.
4. **Output Layer**:
   * **Dense Output**: A single dense unit is used for the regression task, which predicts the steering angle.
5. **Optimizer and Learning Rate**:
   * **Adam Optimizer**: The Adam optimizer with a learning rate of 0.001 is used. Adam adapts the learning rate based on the gradients, which often helps in achieving faster convergence.

**Advantages:**

1. **Simplicity**:
   * **Easier to Implement**: The model’s simplicity makes it easier to implement and understand, which is beneficial for prototyping and debugging.
   * **Less Computationally Intensive**: With fewer layers and parameters, this model is less computationally intensive and can be trained faster.
2. **Good Starting Point**:
   * **Baseline Model**: It serves as a good baseline model for steering angle prediction. You can use it to gauge initial performance before moving on to more complex models.
3. **Regularization**:
   * **Dropout**: The inclusion of dropout helps prevent overfitting, making the model more robust to variations in the training data.
4. **Efficient Learning**:
   * **ReLU Activation**: ReLU is computationally efficient and helps in faster convergence compared to other activation functions like sigmoid or tanh.
5. **Well-Suited for Simpler Data**:
   * **Effective for Basic Tasks**: This model can perform well on tasks with relatively simple features and smaller datasets.

model = models.Sequential()  
model.add(Conv2D(8, (3, 3), input\_shape=(66, 200, 3), activation='relu'))  
model.add(MaxPooling2D(pool\_size=(2, 2)))  
model.add(Conv2D(8, (3, 3), activation='relu'))  
model.add(MaxPooling2D(pool\_size=(2, 2)))  
model.add(Dropout(0.5))  
model.add(Flatten())  
model.add(Dense(50, activation='relu'))  
model.add(Dense(1))  
adam = Adam(learning\_rate=0.001)  
model.compile(loss='mean\_squared\_error', optimizer=adam)  
return model

### 2.3.2 Recurrent Neural Networks (RNNs) and Long Short-Term Memory Networks (LSTMs)

Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining an internal state (memory) that captures information about past observations. They are particularly suitable for tasks requiring context from previous inputs, such as predicting steering angles based on a sequence of images or predicting future states in dynamic environments.

* **Long Short-Term Memory Networks (LSTMs)**: LSTMs are a specialized variant of RNNs designed to address the vanishing gradient problem and capture long-term dependencies in sequential data. Their gated architecture allows them to retain information over extended time periods, making them effective for modeling temporal dynamics in autonomous driving scenarios.

In autonomous driving applications, RNNs and LSTMs enable the vehicle to learn from historical data, anticipate future actions, and adapt to changing environmental conditions, thereby enhancing navigation and decision-making capabilities.

### 2.3.3 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) are a class of deep learning models comprising two neural networks: a generator and a discriminator. GANs are used primarily for generating new data instances that resemble the training data, such as images or videos.

* **Generator**: The generator network learns to generate synthetic data by transforming random noise into outputs that resemble real data samples from the training set. It aims to produce realistic outputs that can deceive the discriminator.
* **Discriminator**: The discriminator network learns to distinguish between real data samples from the training set and fake samples generated by the generator. It provides feedback to the generator to improve the quality of generated samples over time.

### 2.3.4 Which model best suits our project?

We chose to use **Convolutional Neural Network NVIDIA’s** **Model** in our self-driving robot car project due to their proven effectiveness in handling visual data and performing complex tasks such as object detection, lane detection, and traffic sign recognition. CNNs are specifically designed to automatically learn hierarchical representations from pixel data, making them ideal for processing images captured by our onboard camera. By leveraging CNNs, we can accurately interpret and react to the visual information essential for safe navigation and decision-making on the road. Their ability to extract meaningful features from images in real-time allows our autonomous vehicle to identify and respond to dynamic traffic conditions swiftly and with precision, enhancing both the safety and efficiency of our self-driving system.

## 2.4 Sensor Fusion and Perception

Autonomous vehicles rely on a variety of sensors to perceive and understand their environment, enabling them to make informed decisions and navigate safely. Sensor fusion is the process of integrating data from multiple sensors to obtain a more accurate and comprehensive understanding of the surroundings. Key sensors used in autonomous driving include cameras, lidar (Light Detection and Ranging), radar (Radio Detection and Ranging), and ultrasonic sensors.

* **Camera Sensors**: Cameras capture visual data, providing rich information about the vehicle's surroundings, including lane markings, traffic signs, and other vehicles. Deep learning models, particularly CNNs, process camera data for tasks such as object detection, lane detection, and semantic segmentation.
* **Lidar Sensors**: Lidar sensors emit laser pulses to measure distances and create high-resolution 3D maps of the environment. They are effective in detecting and classifying objects, estimating distances, and identifying obstacles in various lighting conditions.
* **Radar Sensors**: Radar sensors use radio waves to detect the presence and location of objects, providing information about relative speed and direction. They are robust in adverse weather conditions and can penetrate fog, rain, and dust, making them essential for maintaining situational awareness in challenging environments.
* **Ultrasonic Sensors**: Ultrasonic sensors measure distances by emitting high-frequency sound waves and detecting their reflections. They are used for short-range detection of obstacles, parking assistance, and low-speed maneuvering.

**Sensor Fusion Techniques**: Integrating data from multiple sensors enhances the reliability and accuracy of perception in autonomous vehicles:

* **Data Fusion Algorithms**: Algorithms combine information from different sensors to generate a unified and coherent representation of the environment. Fusion techniques include sensor-level fusion (combining raw sensor data), feature-level fusion (combining extracted features), and decision-level fusion (combining decision outputs).
* **Multi-Sensor Calibration**: Ensuring accurate alignment and synchronization of sensor data is crucial for reliable perception. Calibration techniques adjust sensor measurements to a common coordinate system, minimizing errors and improving consistency in perception tasks.

In autonomous driving systems, effective sensor fusion and perception capabilities enable vehicles to perceive, interpret, and respond to dynamic environments with high accuracy and reliability, ensuring safe navigation and operation. **In this specific project, we had the chance to calibrate and integrate data from ultrasonic sensor for distance measuring and obstacle avoidance and a camera sensor for image recognition and processing.**

## 2.5 State-of-the-Art Approaches and Technologies

Recent advancements in deep learning have significantly enhanced the capabilities and performance of autonomous driving systems, addressing key challenges and pushing the boundaries of innovation:

* **Object Detection and Tracking**: Advanced object detection frameworks, such as Single Shot MultiBox Detector (SSD) and Region-Based Fully Convolutional Networks (R-FCN), enable real-time detection and tracking of objects in complex environments. These models leverage deep learning techniques to accurately localize and classify objects, improving situational awareness and safety.
* **Semantic Segmentation**: State-of-the-art semantic segmentation models, such as DeepLab and PSPNet (Pyramid Scene Parsing Network), provide pixel-level labeling of scenes, distinguishing between different objects and background elements. Semantic segmentation enhances understanding of the environment and supports decision-making tasks in autonomous driving scenarios.
* **Localization and Mapping**: Localization techniques, including Simultaneous Localization and Mapping (SLAM) algorithms and high-definition (HD) mapping technologies, enable precise positioning and mapping of the vehicle's surroundings. SLAM combines sensor data to create accurate maps, essential for navigating complex urban environments and dynamic road conditions.
* **Simulation and Virtual Testing**: Virtual simulation environments, powered by GANs and other simulation techniques, facilitate safe and cost-effective testing of autonomous driving algorithms. Simulation platforms replicate diverse driving scenarios, enabling developers to validate and optimize algorithms before deployment in real-world settings.

## 2.6 Challenges and Limitations

Despite significant progress, autonomous driving technology faces several challenges and limitations that require further research and development:

* **Complex Urban Environments**: Autonomous cars must navigate through intricate urban landscapes with varying road markings, pedestrian movements, cyclists, and diverse driving behaviors. Ensuring safe and efficient navigation in congested city streets remains a significant challenge.
* **Adverse Weather Conditions**: Heavy rain, snow, fog, and other adverse weather conditions can impair sensor performance and affect the vehicle's ability to perceive its surroundings accurately. Developing robust algorithms that can operate reliably in all-weather scenarios is crucial for autonomous driving systems.
* **Safety in Mixed Traffic**: Coexisting with human-driven vehicles introduces unpredictability and requires autonomous cars to anticipate and respond to the behavior of other drivers and pedestrians. Ensuring safe interactions and adherence to traffic rules in mixed traffic environments is a complex challenge.
* **Real-time Decision-making**: Autonomous cars must make split-second decisions based on rapidly changing information from sensors and surroundings. Achieving real-time processing and decision-making capabilities to handle unexpected events and emergencies is essential for safe operation.
* **Cybersecurity and Safety**: As vehicles become more connected and reliant on communication networks (V2X communication), cybersecurity threats such as hacking and malware pose risks to the safety and reliability of autonomous driving systems. Implementing robust cybersecurity measures to protect vehicle data and operations is critical.
* **Regulatory and Legal Frameworks**: Developing comprehensive regulatory frameworks and addressing legal implications related to liability, insurance, and compliance with traffic laws are crucial for the widespread deployment and acceptance of autonomous vehicles. Harmonizing regulations across jurisdictions presents a challenge for global adoption.
* **Ethical Dilemmas**: Autonomous cars may face ethical dilemmas in situations where decisions must be made that prioritize safety, such as determining actions during unavoidable accidents or emergencies. Addressing ethical considerations and developing ethical decision-making frameworks is an ongoing challenge in autonomous vehicle development.
* **Infrastructure Readiness**: The readiness of infrastructure, such as roads and signage, to support autonomous driving technologies varies across regions. Ensuring adequate infrastructure support, including high-definition mapping and communication networks, is essential for the effective deployment and operation of autonomous vehicles.

## 2.7 Gap Identification and Future Directions

Identifying gaps in current research and proposing future directions is crucial for advancing the capabilities and adoption of autonomous driving technologies:

* **Research Gaps**: Identifying areas where existing literature lacks coverage or where improvements can be made in deep learning applications for autonomous vehicles. Addressing challenges such as real-time decision-making, scalability of algorithms, and adaptation to dynamic environments requires interdisciplinary research and collaboration.
* **Emerging Technologies**: Exploration of emerging technologies, including edge computing, AI hardware accelerators, and quantum computing, and their potential impact on enhancing the capabilities of autonomous driving systems. Leveraging these technologies can improve efficiency, reduce latency, and support complex decision-making processes in autonomous vehicles.

# Chapter 3: Methodology

## 3.1 Introduction

The development of a self-driving robot car represents a significant technological endeavor, requiring extensive research, innovation, and meticulous engineering across hardware and software domains. This chapter details the methodology employed to design, develop, and implement an autonomous navigation system capable of perceiving its environment, making real-time decisions, and navigating safely through dynamic traffic conditions.

The hardware and software components of our self-driving robot car project necessitated substantial effort and dedication. From selecting and integrating complex sensors and actuators to developing sophisticated deep learning algorithms, every aspect of the project demanded meticulous attention to detail and iterative refinement. The synergy between hardware engineering and software development was crucial in achieving seamless integration and robust performance, underscoring the interdisciplinary nature of autonomous driving technology.

Our methodology systematically outlines the steps taken to overcome technical challenges, optimize system performance, and ensure the reliability and safety of autonomous operations. By documenting the rigorous processes involved in hardware setup, software development, data collection, model training, and integration testing, we provide a comprehensive framework for understanding the complexities and innovations driving the evolution of self-driving technology.

Throughout this chapter, we emphasize the hard work and dedication invested in realizing our vision of a self-driving robot car capable of navigating complex environments autonomously. The integration of cutting-edge technologies, coupled with rigorous testing and validation procedures, underscores our commitment to advancing the frontiers of autonomous driving research and development. By sharing our methodology, insights, and lessons learned, we contribute to the broader discourse on autonomous vehicle innovation and pave the way for future advancements in this transformative field.

## 3.2 System Architecture

The system architecture of our self-driving robot car project is structured to facilitate seamless integration and operation of various hardware and software components. At its core, the architecture comprises interconnected subsystems that collectively enable perception, decision-making, and control functionalities essential for autonomous driving.

Central to the architecture is the robot car chassis, selected for its adaptability to house sensors, computing devices, and actuators while ensuring structural integrity and weight distribution conducive to safe autonomous operation. The chassis serves as the physical foundation, supporting the mounting of components such as the Raspberry Pi microcontroller, cameras for visual perception, power bank and batteries for power supplying, and ultrasonic sensor for close-range obstacle detection. Each component's placement and wiring configuration are meticulously designed to optimize performance and reliability, minimizing interference and maximizing data throughput across the system.

A detailed block diagram visually illustrates the hierarchical arrangement of components within the self-driving robot car (Figure 1). This diagram serves as a blueprint, delineating the flow of data and control signals among sensors, microcontrollers, and computing units. It highlights the integration points where hardware subsystems converge to facilitate real-time data processing, decision-making, and motor control. By adopting a modular architecture, we ensure scalability and flexibility in accommodating future upgrades or enhancements to meet evolving technological requirements and performance benchmarks.

## 3.3 Hardware Setup

The hardware setup of our self-driving robot car involves meticulous selection, integration, and calibration of components tailored to meet specific functional requirements essential for autonomous navigation and perception tasks.

### 3.3.1 Hardware Components

A series of photographs (Figures 2, 3, 4, 5, 6, 7,8,9) show the components needed to build the robot car for our project. Each component plays a crucial role in the functionality and performance of the car:

1. **Chassis**: The robust chassis forms the structural foundation of our self-driving robot car, providing stability and durability essential for navigating varied terrains. Its design accommodates the integration of sensors, microcontrollers, and power systems, ensuring optimal weight distribution and mechanical resilience.

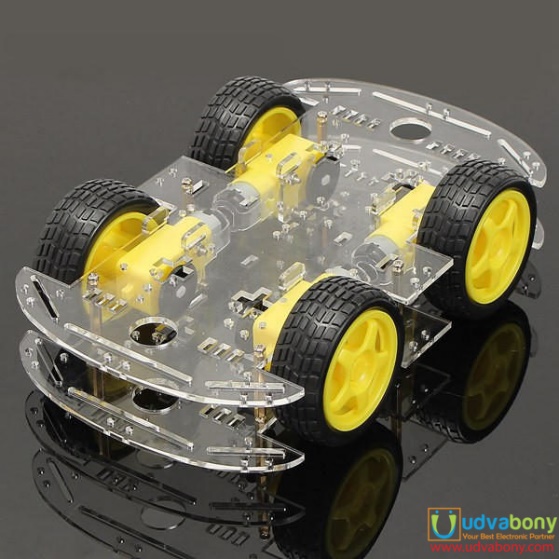


Figure 2- Car chassis

1. **Motors and Wheels**: Motors coupled with durable wheels enable precise movement and maneuverability, essential for autonomous navigation. The motors are equipped to handle varying speeds requirements with the ability to push the car in both forward and backward directions efficiently, supporting agile responses to environmental stimuli during driving operations.



Figure 3- motors & wheels

1. **Ultrasonic Sensor**: Integrated ultrasonic sensors facilitate accurate distance measurement and obstacle detection in real-time. Positioned strategically around the car, these sensors enable proactive collision avoidance and precise maneuvering in confined spaces via measuring distance from the surrounded targets, enhancing overall safety and operational efficiency.



Figure 4- Ultrasonic sensor

It works by emitting ultrasonic sound waves, and converts the reflected sound into an electrical signal. Ultrasonic waves travel faster than the speed of audible sound (i.e. the sound that humans can hear). Ultrasonic sensors have two main components: the transmitter (which emits the sound using piezoelectric crystals) and the receiver (which encounters the sound after it has travelled to and from the target).

In order to calculate the distance between the sensor and the object, the sensor measures the time it takes between the emission of the sound by the transmitter to its contact with the receiver. The formula for this calculation is D = ½ T x C (where D is the distance, T is the time, and C is the speed of sound ~ 343 meters/second)

1. **Raspberry Pi Camera Module 2**: The camera module captures high-resolution images of the car's surroundings, providing critical visual data for object detection and lane recognition. Advanced image processing algorithms leverage camera outputs to enhance environmental perception and decision-making capabilities in dynamic driving scenarios. See camera specifications here.

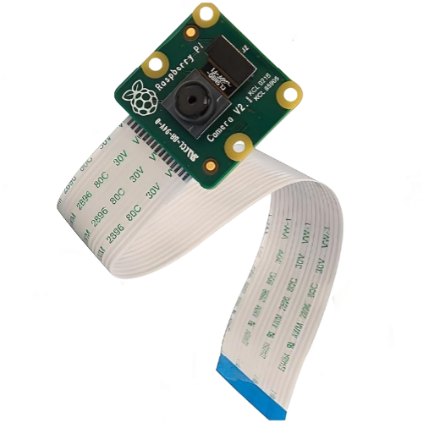


Figure 5- raspberry pi camera v2

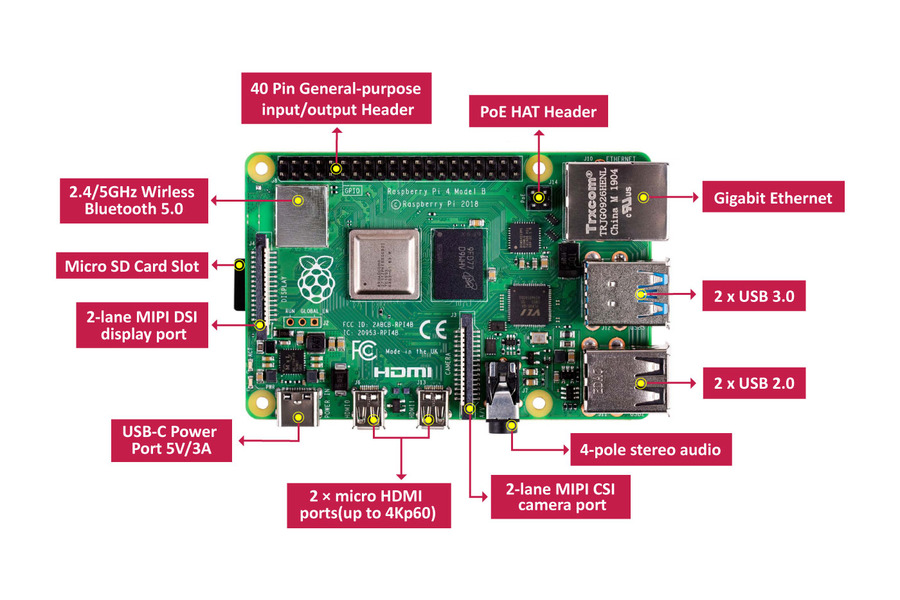
1. **Raspberry Pi 4 B+**: Serving as the central processing unit, the Raspberry Pi 4 B+ orchestrates sensor data fusion, deep learning model inference, and real-time control commands. Its computational power and versatility support complex algorithms for autonomous navigation, ensuring rapid data processing and seamless integration with peripheral devices. You can check out the list of the raspberry pi specifications used in this project here.

Figure 6- Raspberry pi

1. **L298N Motor Driver Module**: The L298N motor driver module plays a crucial role in regulating motor speed and direction, translating control signals from the Raspberry Pi into precise movements of the car. The module features multiple pins, each serving a specific function to enhance motor efficiency and responsiveness while optimizing power consumption. The key pins include:

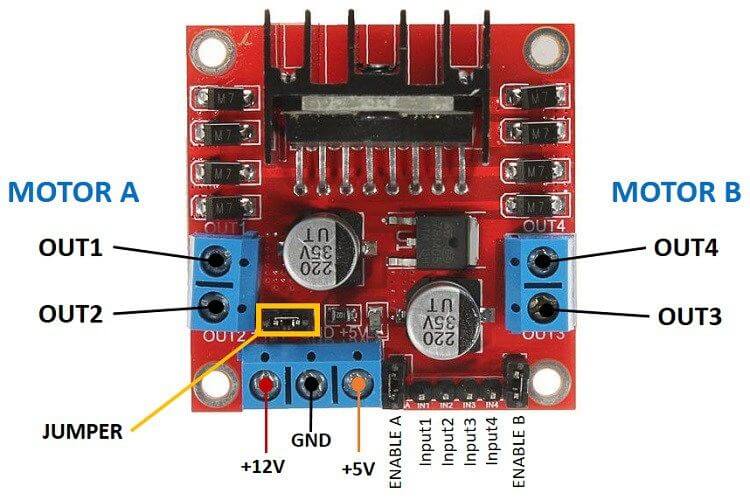


Figure 7- L298N motor driver

* **ENA and ENB Pins:** These pins enable or disable the motors connected to the module. By applying a PWM (Pulse Width Modulation) signal to these pins, the speed of the motors can be precisely controlled.
* **IN1, IN2, IN3, and IN4 Pins:** These input pins determine the direction of the motors, where IN1 and IN2 are for controlling MOTOR A (right side motor) whereas IN3 and IN4 are for controlling MOTOR B (left side motor). For example, setting IN1 to HIGH and IN2 to LOW will rotate Motor A backward, while setting IN1 to LOW and IN2 to HIGH will rotate it farward. The same logic applies to IN3 and IN4 for Motor B.
* **OUT1, OUT2, OUT3, and OUT4 Pins:** These output pins connect to the motors, transmitting the signals that control the motors' direction and speed.
* **Vcc Pin:** This pin provides the supply voltage to the motors, typically requiring a higher voltage (e.g., 12V) to power the motors efficiently.
* **GND Pins:** These ground pins complete the electrical circuit, ensuring stable operation of the module.
* **12V Pin:** This pin supplies power to the motor driver itself, separate from the Vcc pin that powers the motors.
* **5V Pin:** This pin can provide a regulated 5V output, useful for powering other components or sensors in the system.

1. **Power Supplies and other components**: A dedicated power bank supplies stable voltage to the Raspberry Pi, ensuring uninterrupted operation of computational tasks and sensor interfacing. Additionally, a series of 12V batteries (eight batteries in total) to power the motors, delivering sufficient current for dynamic driving maneuvers and extended operational endurance, furthermore a 9V battery connected to a small fan attached above the raspberry pi for cooling purposes, and a switch to power the fan on and off easily.



Figure 8- Power Supplies and other components

1. **Road (Hard Papers with White Tapes)**: The simulated road consists of hard papers lined with white tapes, providing a structured environment for testing and validating the self-driving capabilities of our robot car. The white tapes serve as lane markings, guiding the car's navigation algorithms to maintain trajectory and adhere to road rules.



Figure 9- Road (Hard Papers with White Tapes)

### 3.3.2 Hardware Implementation and connections

In an autonomous driving car project, it is essential to establish proper hardware connections to ensure seamless control and communication between various components.

This section outlines the connectivity of each component with a clear and detailed explanation to facilitate understanding and implementation. It is followed by a well-architected diagram that visually represents the connections.

Understanding these connections is crucial for the successful operation of the autonomous driving system.

#### 3.3.2.1 Connection between H-bridge and DC motors

For our robot car, we require four motors, each attached to a corresponding wheel, along with an H-Bridge L298N Driver and a Raspberry Pi 4 B+. The following steps outline the process for connecting the motors:

1. **Pairing Motors**:

* Connect two motors in parallel such that they move in the same direction. Repeat this process with the remaining two motors to create two pairs.

1. **Connecting to H-Bridge**:

* Connect the positive and negative terminals of the first motor pair to the OUT1 and OUT2 terminals of the H-Bridge, respectively.
* Connect the positive and negative terminals of the second motor pair to the OUT3 and OUT4 terminals of the H-Bridge, respectively.

By following these steps, the motors will be correctly interfaced with the H-Bridge, allowing for proper control via the Raspberry Pi.

#### 3.3.2.2 Connection between Raspberry Pi and H-Bridge

To control the H-Bridge L298N Driver with the Raspberry Pi 4 B+, follow these steps:

1. **Power Connections**:

* After setting up the 12V power supply to power up the H-Bridge, connect the 12V and GND pins of the H-Bridge to the positive and negative poles of the 12V power supply case respectively to power the driver and control it from the raspberry pi. Furthermore, connect the GND pin of the H-Bridge to a raspberry pi GND pin (pin 6 for example).

1. **Control Pins**:

Connect the IN1, IN2, IN3, and IN4 pins of the H-Bridge to the GPIO pins on the Raspberry Pi. For instance:

* + - Connect IN1 on the H-Bridge to GPIO17 (pin 11) on the Raspberry Pi.
    - Connect IN2 on the H-Bridge to GPIO18 (pin 12) on the Raspberry Pi.
    - Connect IN3 on the H-Bridge to GPIO22 (pin 15) on the Raspberry Pi.
    - Connect IN4 on the H-Bridge to GPIO23 (pin 16) on the Raspberry Pi.

1. **Enable Pins**:
   * If your H-Bridge has ENA and ENB pins for enabling the motor driver channels, connect them to additional GPIO pins on the Raspberry Pi. For example:
     + Connect ENA to GPIO24 (pin 18) for controlling the first motor pair.
     + Connect ENB to GPIO25 (pin 22) for controlling the second motor pair.

By completing these connections, the Raspberry Pi will be able to send control signals to the H-Bridge, thereby managing the operation of the connected DC motors.

#### 3.3.2.3 Connection between Raspberry Pi and ultrasonic sensor

To interface the ultrasonic sensor with the Raspberry Pi 4 B+, follow these steps, including the use of resistors to protect the GPIO pins:

1. **Power Connections**:

* Connect the VCC pin of the ultrasonic sensor to a 5V pin on the Raspberry Pi (e.g., pin 2 or pin 4).
* Connect the GND pin of the ultrasonic sensor to a GND pin on the Raspberry Pi (e.g., pin 6 or pin 9).

1. **Control Pins**:

* Connect the TRIG (trigger) pin of the ultrasonic sensor to a GPIO pin on the Raspberry Pi. For example, connect the TRIG pin to GPIO23 (pin 16).
* Connect the ECHO (echo) pin of the ultrasonic sensor to a voltage divider made of two resistors before connecting it to a GPIO pin on the Raspberry Pi. This is necessary to step down the 5V signal from the ECHO pin to a safe 3.3V for the Raspberry Pi GPIO.

1. **Resistor Connections**:

* Use a voltage divider with two resistors (e.g., 1 kΩ and 2 kΩ) to reduce the voltage from the ECHO pin.
* Connect the ECHO pin of the ultrasonic sensor to one end of the 1 kΩ resistor.
* Connect the other end of the 1 kΩ resistor to one end of the 2 kΩ resistor and also to the GPIO24 pin (pin 18) on the Raspberry Pi.
* Connect the other end of the 2 kΩ resistor to GND.

1. **Wiring Diagram**:

* **VCC** (Ultrasonic Sensor) → **5V** (Raspberry Pi, pin 2 or 4)
* **GND** (Ultrasonic Sensor) → **GND** (Raspberry Pi, pin 6 or 9)
* **TRIG** (Ultrasonic Sensor) → **GPIO23** (Raspberry Pi, pin 16)
* **ECHO** (Ultrasonic Sensor) → **Voltage Divider** → **GPIO24** (Raspberry Pi, pin 18)

With these connections and the use of resistors to create a voltage divider, the Raspberry Pi can safely interact with the ultrasonic sensor by sending a pulse to the TRIG pin to initiate a measurement and reading the pulse duration on the ECHO pin to determine the distance to an object.

#### 3.3.2.4 Connection between Raspberry Pi 4 and Camera Module V2

To connect the Raspberry Pi Camera Module V2 to the Raspberry Pi 4 B+, follow these steps:

1. **Preparation**:

* Ensure that the Raspberry Pi is powered off before connecting the camera to prevent any damage to the hardware.

1. **Locate the Camera Interface**:

* Find the camera connector (labeled "CAMERA") on the Raspberry Pi board. It is located near the HDMI ports.

1. **Insert the Camera Ribbon Cable**:

* Carefully lift the plastic latch on the camera connector.
* Insert the ribbon cable from the Camera Module V2 into the connector with the metal contacts facing away from the HDMI ports (toward the Raspberry Pi's board).
* Once the ribbon cable is in place, push down the plastic latch to secure the cable.

#### 3.3.2.5 Connecting Raspberry Pi to Buzzer, LED Lamps, and Switches

* **Buzzer**: triggered on car brake
* Positive Terminal → GPIO17 (pin 11)
* Negative Terminal → GND (pin 6 or pin 9)
* **LED Lamps**: lights up when the car brakes
* LED1 Anode → Resistor (e.g., 220Ω) → GPIO27 (pin 13)
* LED1 Cathode → GND (pin 6 or pin 9)
* **Switches**: used to easily power on/off the components
* Motor Switch:
  + Terminal 1 → 12V H-Bridge pin
  + Terminal 2 → 12V power supply
* Fan Switch:
  + Terminal 1 → 9V Battery positive poll
  + Terminal 2 → 9V Battery negative poll

#### 3.3.2.6 Component Connectivity Diagram

This section provides a comprehensive explanation of how each component is mounted and connected to ensure the car’s functionality.

First, securely attach the motors to the lower chassis. Then, connect the motors on each side of the car in parallel; that is, connect the right-side motors together and the left-side motors together, as illustrated in the diagram. In the third step, connect each pair of motors to the output ports of the H-bridge, ensuring correct polarity for both negative and positive poles. The fourth step involves connecting the positive terminal of the 12V power supply to the 12V pin on the H-bridge, along with the switch, and connecting the negative terminal of the power supply to the ground pin of the H-bridge, also along with the other terminal of the switch. All these connections are made on the bottom of the chassis, so please refer to the overview image before beginning.

Next, mount the power bank on the other side of the lower chassis as shown in the overview image, using ribbons to secure it. Connect the H-bridge to the Raspberry Pi by linking the IN1, IN2, IN3, IN4, EnA, and EnB pins to any GPIO pins of your choice. Ensure that the ground pin of the H-bridge is connected to a ground pin on the Raspberry Pi. Mount the Raspberry Pi on the second chassis inside its cover, along with the camera. At this point, the car should be able to move as required.

Proceed by connecting the camera by plugging the camera line into the camera port. If the camera does not function, this issue will be addressed later in the software chapter. Ensure that all connections match those shown in the diagram below. Next, connect the ultrasonic sensor, which has four pins: VCC (connect to a 5V Raspberry Pi pin), GND (connect to a Raspberry Pi ground pin), TRIG (connect to any GPIO pin), and ECHO. Connect the ECHO pin to a voltage divider made of two resistors before linking it to a GPIO pin on the Raspberry Pi. This step is necessary to reduce the 5V signal from the ECHO pin to a safe 3.3V for the Raspberry Pi GPIO.

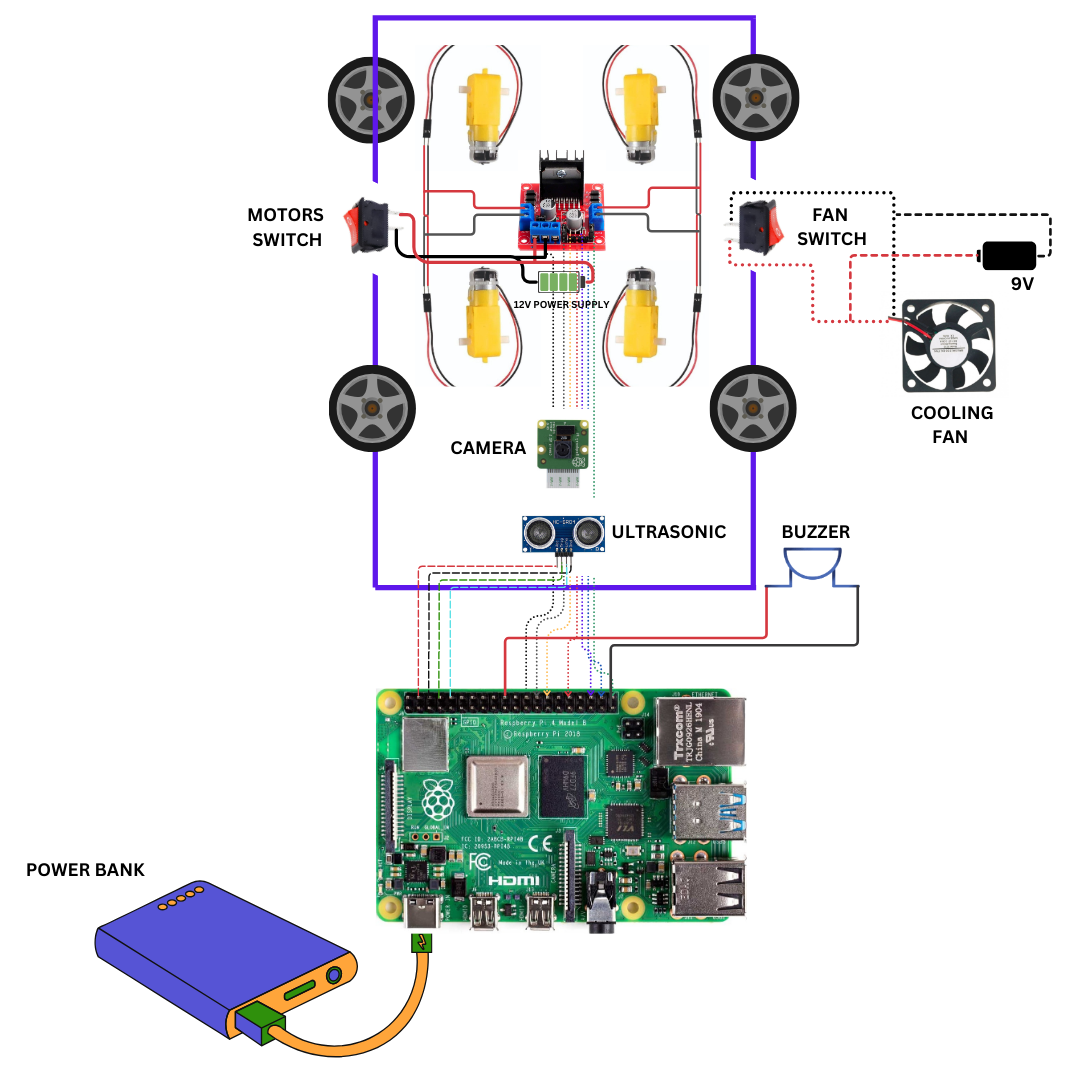
****Finally, connect the buzzer by connecting one pole to a ground Raspberry Pi pin and the other pole to a GPIO pin. For the Raspberry Pi fan and the 9V battery, connect each terminal to its corresponding pin, including the switch, as depicted in the diagram. Lastly, there are optional small LED lights that can be connected as desired.

Figure 10- Component Connectivity Diagram

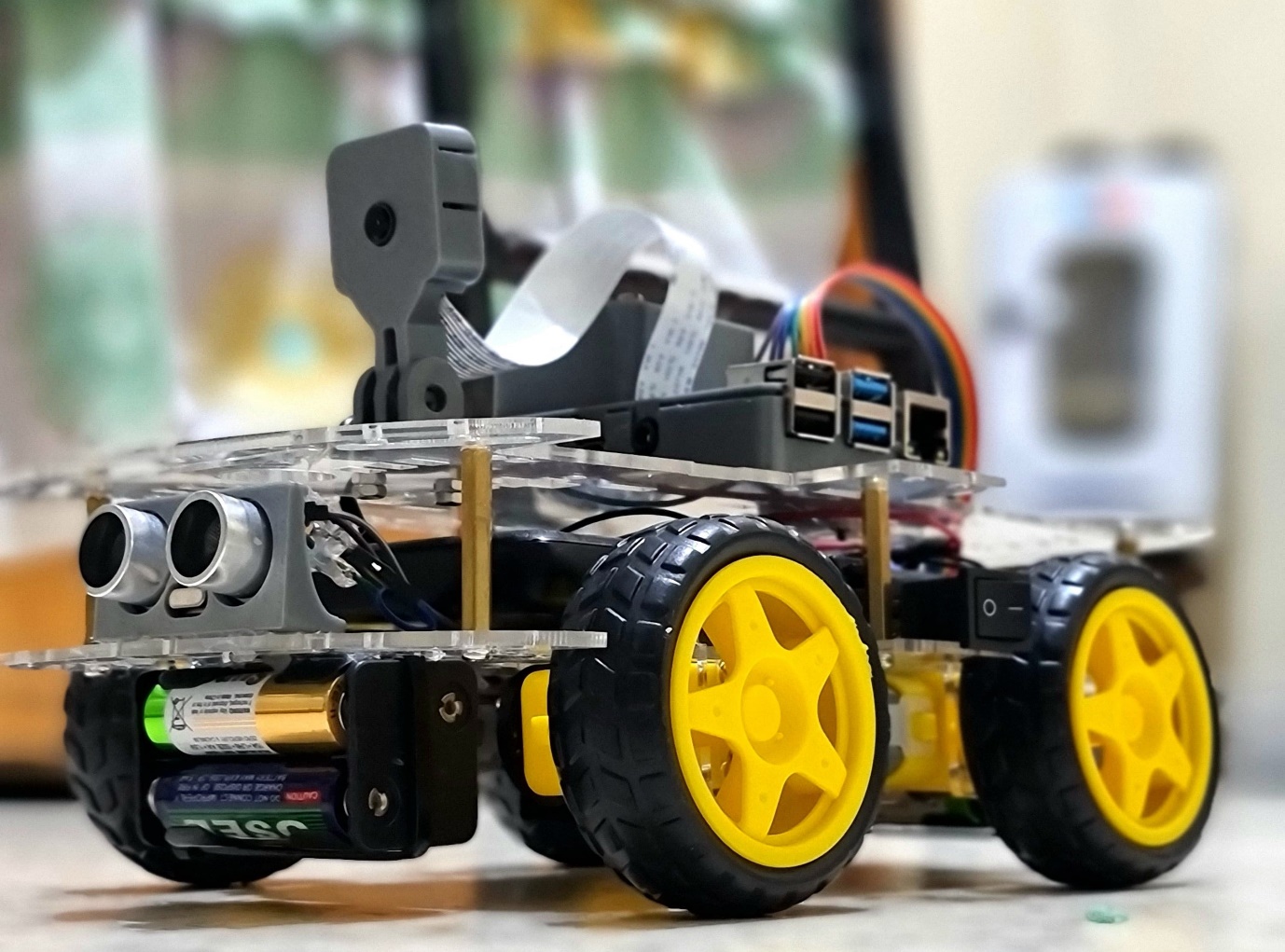


Figure 11- car overview

## 3.4 Software Development

Software development for our self-driving robot car project encompasses the creation and integration of diverse modules tailored to handle sensor data processing, deep learning model implementation, real-time decision-making algorithms, and interface functionalities. Implemented primarily in Python using libraries such as TensorFlow for machine learning and OpenCV for computer vision, these modules form the computational backbone of our autonomous driving system.

Central to our software architecture is the Image Processing Module, responsible for preprocessing raw sensor data captured by cameras and other sensors mounted on the vehicle. Preprocessing techniques include image normalization, cropping, resizing, and color space transformation, and noise reduction, aimed at enhancing the quality and consistency of input data for subsequent processing stages. These processed images serve as input to the Deep Learning Model, a Convolutional Neural Network (CNN) optimized for tasks such as object detection, lane detection, and traffic sign recognition.

The CNN architecture comprises multiple layers of convolutional, pooling, and fully connected nodes designed to automatically extract hierarchical features from input images, enabling the car to perceive and interpret its surroundings accurately. Model training involves iterative optimization of network parameters using labeled datasets collected during initial testing and calibration phases. Hyperparameters such as learning rate, batch size, and dropout rate are fine-tuned to maximize model performance and generalization capabilities across diverse driving scenarios.

Real-time Decision-Making Algorithms interface with the trained deep learning model to process sensor inputs, predict steering angles, adjust vehicle speed, and execute safe navigation commands in response to dynamic environmental conditions. These algorithms leverage feedback loops and control theory principles to ensure smooth trajectory planning, obstacle avoidance, and compliance with traffic regulations while prioritizing passenger safety and operational efficiency.

Detailed code snippets will be provided at the end to illustrate key aspects of our software implementation, showcasing algorithms for sensor data fusion, real-time object detection, and motor control integration. These snippets highlight the application of machine learning techniques and computer vision algorithms to achieve autonomous driving capabilities, providing transparency and reproducibility in our software development approach.

### 3.4.1 Environmental setups and configuration

Before diving deep into the project's software components, this section focuses on the crucial steps of setting up and testing the environment. Ensuring that the hardware and software environments are properly configured is essential for the smooth development and execution of the project. This involves a series of preparatory steps, including installing necessary software packages, configuring system settings, and verifying hardware functionality.

#### 3.4.1.1 Detailed Guide to Installing an Operating System on Raspberry Pi 4

Installing an operating system on a Raspberry Pi 4 can be done in several steps. Here's a detailed guide to help you through the process:

**Step 1**: Gather the Necessary Components

1. **Raspberry Pi 4**
2. **MicroSD Card (16GB or larger)**
3. **MicroSD Card Reader**
4. **Power Supply (5V 3A USB-C)**
5. **HDMI Cable (Micro HDMI to standard HDMI)**
6. **Monitor**
7. **Keyboard and Mouse**
8. **Internet Connection (Ethernet cable or Wi-Fi)**

**Step 2**: Download Raspberry Pi Imager

1. Go to the official Raspberry Pi website.
2. Download the Raspberry Pi Imager for your operating system (Windows, macOS, or Ubuntu).

**Step 3**: Install Raspberry Pi Imager

1. Run the downloaded installer and follow the on-screen instructions to install the Raspberry Pi Imager.

**Step 4**: Prepare the MicroSD Card

1. Insert the MicroSD card into your computer using the card reader.
2. Open the Raspberry Pi Imager software.
3. Click on “Choose OS” and select the desired operating system. For beginners, the recommended option is “Raspberry Pi OS (32-bit)”.
4. Click on “Choose SD Card” and select your MicroSD card from the list.
5. Apply some configurations (optional) and pay attention to the username and password that you create because they will be used to access the raspberry pi later.
6. Click “Write” to start the installation process. This will format the MicroSD card and write the OS image onto it.

**Step 5**: Insert the MicroSD Card into the Raspberry Pi

1. Once the image has been written, safely eject the MicroSD card from your computer.
2. Insert the MicroSD card into the MicroSD card slot on the Raspberry Pi 4.

**Step 6**: Connect Peripherals

1. Connect your monitor to the Raspberry Pi using the Micro HDMI cable.
2. Connect the keyboard and mouse to the USB ports on the Raspberry Pi.
3. If using a wired connection, connect the Ethernet cable.

**Step 7**: Power Up the Raspberry Pi

1. Connect the power supply to the Raspberry Pi and plug it into a power outlet.
2. The Raspberry Pi should start booting up. You will see the boot screen on your monitor.

**Step 8**: Initial Setup

In this part we are going to learn how to quickly setup your Raspberry pi for the first time. We will learn how we can remotely control Raspberry pi with windows or mac.

1. Follow the on-screen instructions to set up your Raspberry Pi. This includes setting up the language, timezone, and network connection.
2. The Raspberry Pi OS will also prompt you to update the software. It's a good idea to do this to ensure you have the latest security updates and features.
3. Go to **preferences** > **raspberry pi configurations**, and from there enable both **VNC** and **SSH** options in order to access the raspberry pi remotely.
4. Install **VNC viewer** along with **Angry IP scanner** on the device that you want to connect with your raspberry pi (ex. Your pc).
5. Open **Angry IP scanner** software and write the ip range of your network (ex. 192.168.1.0 to 192.168.1.255) and hit the start button, the ip address of your raspberry pi should be visible to you, copy it and paste it in the VNC viewer.
6. The VNC viewer will ask for the username and password that you already created when installing the os.

**Step 9**: preparing the environment

1. Make sure python is properly installed, open the raspberry pi terminal and write (python --version).
2. Try opening a python ide and test some code to see if everything is properly functioning.

#### 3.4.1.2 Setting up the camera

This is the part that took us a significant amount of time and hard work to complete. There are numerous unmentioned libraries and modules that must be installed, as well as intricate system configurations that need to be done to set up the camera to function properly. Our resources were limited regarding this issue, which added to the challenge. We had to research extensively and troubleshoot various problems to identify and install all the necessary components.

It is crucial to follow the instructions in this section step by step to properly configure the camera. Ensure that each mentioned library and module is downloaded and installed successfully, otherwise the camera will not function as intended. Pay close attention to the details provided, as missing even a single component can lead to operational issues. This meticulous process requires patience and precision, reflecting the considerable effort and dedication we put into overcoming the obstacles to achieve a fully functional setup.

we are going to Learn how to install opencv on Raspberry Pi. Usually this can be done with the python package manager pip but in most cases the pip install does not work properly with opencv on raspberry pi. Therefore, we have to install from source. In this section we will go through all the steps require to install opencv from source.

**Step 1: Update Raspbian to the latest version.**

Open the terminal and update the package list of raspberry pi:

sudo apt-get update && sudo apt-get upgrade && sudo rpi-update

**Step 2: Increase the swap-size**

The swap is a file on disk that serves as ‘overflow’ RAM space. By default, raspberry pi swap size is 100Mb, which is way too small as while installing OpenCV your process will crash unnecessarily. To avoid this we need to increase the swap size. To increase the swap size, open swapfile by:

sudo nano /etc/dphys-swapfile

and edit the variable CONF\_SWAPSIZE :

#CONF\_SWAPSIZE=100  
 CONF\_SWAPSIZE=2048

**Step 3: Install tools and libraries for openCV**

This step will take about 10 minutes.

sudo apt-get install build-essential cmake pkg-config  
sudo apt-get install libjpeg-dev libtiff5-dev libjasper-dev libpng12-dev  
sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev libv4l-dev  
sudo apt-get install libxvidcore-dev libx264-dev  
sudo apt-get install libgtk2.0-dev libgtk-3-dev  
sudo apt-get install libatlas-base-dev gfortran

**Step 4: Install Python3 and pip3**

If you do not have Python installed, you can install it by the following command:

sudo apt-get install python3-dev  
sudo apt-get install python3-pip

**Step 5: Get OpenCV 4.1.0 source code**

You need to download and unzip as follows:

wget -O opencv.zip <https://github.com/opencv/opencv/archive/4.10.0.zip>wget -O opencv\_contrib.zip <https://github.com/opencv/opencv_contrib/archive/4.10.0.zip>

unzip opencv.zip

unzip opencv\_contrib.zip

*Note: Change the version number in step 5 to the latest available version, the latest version at the time this project was done is 4.10.0.*

**Step 6: Install numpy:**

Numpy is used to perform array operations in Python.

sudo pip3 install numpy

**Step 7: Compile OpenCV**

For this step you need to create a build folder where all the files are created.

cd ~/opencv-4.1.0/  
mkdir build  
cd buildcmake -D CMAKE\_BUILD\_TYPE=RELEASE \  
 -D CMAKE\_INSTALL\_PREFIX=/usr/local \  
 -D INSTALL\_PYTHON\_EXAMPLES=ON \  
 -D OPENCV\_EXTRA\_MODULES\_PATH=~/opencv\_contrib-4.1.0/modules \  
 -D BUILD\_EXAMPLES=ON ..

If you are trying to install a different version of OpenCV, update the paths accordingly.

**Step 8: Build OpenCV**

This is the most crucial step and it may even take more than 3 hours. To use all four cores on the Raspberry Pi, type in the following:

make -j4

Make sure the raspberry pi completes this step without any errors.

**Step 9: Install OpenCV**

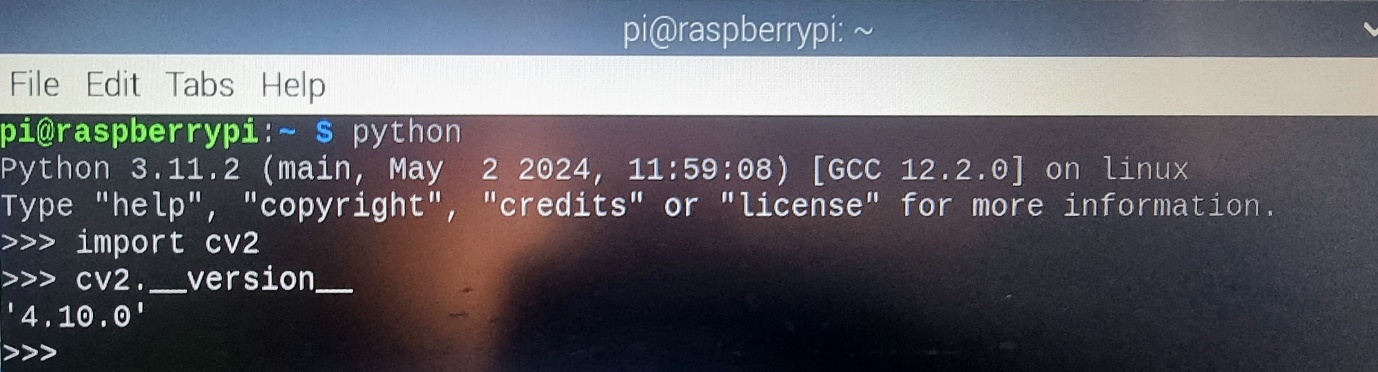
Now finally, you can install OpenCV.

sudo make install && sudo ldconfig

At this moment if everything is done without any errors then you can reboot your system :)

sudo reboot

**Step 10: Check for OpenCV**



**Step 11: install picamera2**

sudo apt install -y python3-picamera2 libcamera-apps

pip3 install picamera2

**Step 12: test code**

import cv2

from picamera2 import Picamera2

picam2 = Picamera2()

picam2.preview\_configuration.main.size = (1920,1080)

picam2.preview\_configuration.main.format = "RGB888"

picam2.preview\_configuration.align()

picam2.configure("preview")

picam2.start()

def getImg()

img= picam2.capture\_array()

cv2.imshow('IMG',img)

cv2.waitKey(1)

if \_\_name\_\_ == '\_\_main\_\_':

while True:

getImg()

**3.4.1.3 motors testing**

In this section we are going to learn how to run a DC motor with Raspberry pi using L298 motor driver. We will write a Motor class using Object Oriented programming so that it could scaled for bigger projects. In other words, you will be able run and manage lots of motors with just few lines of code.

import RPi.GPIO as GPIO

from time import sleep

GPIO.setmode(GPIO.BCM)

GPIO.setwarnings(False)

class Motor():

def \_\_init\_\_(self,EnA,In1,In2,In3,In4,EnB):

self.EnA= EnA

self.In1 = In1

self.In2 = In2

self.EnB= EnB

self.In3 = In3

self.In4 = In4

GPIO.setup(self.EnA,GPIO.OUT);GPIO.setup(self.In1,GPIO.OUT);GPIO.setup(self.In2,GPIO.OUT)

GPIO.setup(self.EnB,GPIO.OUT);GPIO.setup(self.In3,GPIO.OUT);GPIO.setup(self.In4,GPIO.OUT)

self.pwmA = GPIO.PWM(self.EnA, 100);

self.pwmB = GPIO.PWM(self.EnB, 100);

self.pwmA.start(0);

self.pwmB.start(0);

def moveForward(self,x=100,t=0):

self.pwmA.ChangeDutyCycle(x)

self.pwmB.ChangeDutyCycle(x)

GPIO.output(self.In1,GPIO.LOW)

GPIO.output(self.In2,GPIO.HIGH)

GPIO.output(self.In4,GPIO.LOW)

GPIO.output(self.In3,GPIO.HIGH)

sleep(t)

def moveBackward(self,x=100,t=0):

self.pwmA.ChangeDutyCycle(x)

self.pwmB.ChangeDutyCycle(x)

GPIO.output(self.In2,GPIO.LOW)

GPIO.output(self.In1,GPIO.HIGH)

GPIO.output(self.In3,GPIO.LOW)

GPIO.output(self.In4,GPIO.HIGH)

sleep(t)

def stop(self,t=0):

self.pwmA.ChangeDutyCycle(0);

self.pwmB.ChangeDutyCycle(0);

sleep(t)

if \_\_name\_\_==’\_\_main\_\_’:

motor = Motor(1,2,3,4,5,6) #### raspberry pi pins that are connected to the motors driver

motor.moveF(100,5) ######### move forward for 5 seconds

motor.moveB(100,5) ######### move backward for 5 seconds

motor.stop() ################ stop !!

With the foundational components successfully implemented and tested, we can confidently move forward to explore more advanced features and optimizations. This includes refining our sensor integration, enhancing our machine learning models, and incorporating more sophisticated data processing techniques. Our next steps will involve rigorous testing, iterative improvement, and the application of best practices to ensure the robustness and reliability of our system.

### 3.4.2 Modules Used in the Project

This project involves several key modules that handle various aspects of the autonomous driving car, including motor control, joystick input, and camera operations. Below is a detailed explanation of each module, highlighting its purpose and functionality within the project, these modules will be used for data collection and project testing purposes.

#### 3.4.2.1 Motor Module

The motor module is responsible for controlling the movement of the car. It uses the GPIO library to interface with the Raspberry Pi's GPIO pins, allowing the car to move forward, backward, and turn. This module includes methods for starting, stopping, and rotating the car, as well as an obstacle avoidance mechanism.

import RPi.GPIO as GPIO

from time import sleep

GPIO.setmode(GPIO.BCM)

GPIO.setwarnings(False)

class Motor():

def \_\_init\_\_(self, EnA, In1, In2, In3, In4, EnB):

self.EnA = EnA

self.In1 = In1

self.In2 = In2

self.EnB = EnB

self.In3 = In3

self.In4 = In4

GPIO.setup(self.EnA, GPIO.OUT)

GPIO.setup(self.In1, GPIO.OUT)

GPIO.setup(self.In2, GPIO.OUT)

GPIO.setup(self.EnB, GPIO.OUT)

GPIO.setup(self.In3, GPIO.OUT)

GPIO.setup(self.In4, GPIO.OUT)

self.pwmA = GPIO.PWM(self.EnA, 100)

self.pwmB = GPIO.PWM(self.EnB, 100)

self.pwmA.start(0)

self.pwmB.start(0)

def move(self, speed=0.5, turn=0, t=0):

speed \*= 100

turn \*= 70

leftSpeed = speed - turn

rightSpeed = speed + turn

GPIO.output(self.In1, GPIO.LOW)

GPIO.output(self.In2, GPIO.HIGH)

GPIO.output(self.In3, GPIO.HIGH)

GPIO.output(self.In4, GPIO.LOW)

self.pwmA.ChangeDutyCycle(abs(leftSpeed))

self.pwmB.ChangeDutyCycle(abs(rightSpeed))

if leftSpeed > 0:

GPIO.output(self.In1, GPIO.LOW)

GPIO.output(self.In2, GPIO.HIGH)

else:

GPIO.output(self.In1, GPIO.HIGH)

GPIO.output(self.In2, GPIO.LOW)

if rightSpeed > 0:

GPIO.output(self.In3, GPIO.HIGH)

GPIO.output(self.In4, GPIO.LOW)

else:

GPIO.output(self.In3, GPIO.LOW)

GPIO.output(self.In4, GPIO.HIGH)

sleep(t)

def stop(self, t=0):

self.pwmA.ChangeDutyCycle(0)

self.pwmB.ChangeDutyCycle(0)

sleep(t)

def fullRotate(self):

self.pwmA.ChangeDutyCycle(100)

self.pwmB.ChangeDutyCycle(100)

GPIO.output(self.In1, GPIO.HIGH)

GPIO.output(self.In2, GPIO.LOW)

GPIO.output(self.In3, GPIO.HIGH)

GPIO.output(self.In4, GPIO.LOW)

def avoid(self):

GPIO.output(self.In1, GPIO.HIGH)

GPIO.output(self.In2, GPIO.LOW)

GPIO.output(self.In3, GPIO.HIGH)

GPIO.output(self.In4, GPIO.LOW)

self.pwmA.ChangeDutyCycle(100)

self.pwmB.ChangeDutyCycle(100)

sleep(0.3)

GPIO.output(self.In1, GPIO.LOW)

GPIO.output(self.In2, GPIO.HIGH)

GPIO.output(self.In3, GPIO.HIGH)

sleep(0.65)

GPIO.output(self.In3, GPIO.LOW)

GPIO.output(self.In4, GPIO.HIGH)

sleep(0.6)

GPIO.output(self.In3, GPIO.HIGH)

GPIO.output(self.In4, GPIO.LOW)

sleep(0.65)

GPIO.output(self.In1, GPIO.HIGH)

GPIO.output(self.In2, GPIO.LOW)

GPIO.output(self.In3, GPIO.HIGH)

GPIO.output(self.In4, GPIO.LOW)

sleep(0.35)

This class is initialized with the GPIO pins used to control the motor driver. The move method adjusts the speed and direction of the motors based on the input parameters. The stop method halts the motors, and the fullRotate method sets the motors to rotate in place. The avoid method provides a basic obstacle avoidance routine.

**Description**: The motor module is responsible for controlling the car's movement. It uses the GPIO library to interface with the Raspberry Pi's GPIO pins, allowing the car to move forward, backward, and turn.

**Code Explanation**: The motor module is implemented through a class that initializes the GPIO pins and sets up PWM (Pulse Width Modulation) for controlling motor speed. Here's a detailed breakdown of the code:

1. **Initialization**:

* The \_\_init\_\_ method sets up the GPIO pins for motor control and initializes PWM for both motors.
* EnA and EnB are the enable pins for the motors, while In1, In2, In3, and In4 control the direction.

1. **Movement Control**:

* The move method adjusts the speed and direction of the motors based on input parameters.
* It calculates leftSpeed and rightSpeed to control the motors independently for turning.

1. **Stopping the Motors**:

* The stop method halts the motors by setting the PWM duty cycle to zero.

1. **Full Rotation**:

* The fullRotate method sets the motors to rotate the car in place by running both sides of the car in the opposite directions.

1. **Obstacle Avoidance**:

* The avoid method implements a basic obstacle avoidance routine, adjusting the motor directions and speeds to navigate around obstacles(ahead cars) on a straight road.

**Challenges and Solutions**: Implementing the motor control required precise timing and calibration to ensure smooth movement. One challenge was managing the PWM signals to avoid jittery movements, which was resolved by fine-tuning the PWM frequencies and duty cycles.

#### 3.4.2.2 Joystick Module

The joystick module uses the Pygame library to read inputs from a joystick, allowing manual control of the car. It captures button presses and analog stick movements, updating a dictionary that tracks the state of each input.

import pygame

from time import sleep

pygame.init()

controller = pygame.joystick.Joystick(0)

controller.init()

buttons = {

'x': 0, 'o': 0, 's': 0, 't': 0,

'share': 0, 'useless': 0, 'options': 0,

'axis1': 0.0, 'axis2': 0.0

}

def getJS(name=''):

global buttons

for event in pygame.event.get():

if event.type == pygame.JOYAXISMOTION:

if event.axis == 0:

buttons['axis1'] = round(event.value, 2)

elif event.axis == 1:

buttons['axis2'] = round(event.value, 2)

elif event.type == pygame.JOYBUTTONDOWN:

if event.button < len(buttons):

button\_name = list(buttons.keys())[event.button]

buttons[button\_name] = 1

elif event.type == pygame.JOYBUTTONUP:

if event.button < len(buttons):

button\_name = list(buttons.keys())[event.button]

buttons[button\_name] = 0

if name == '':

return buttons

else:

return buttons.get(name, None)

This module initializes the joystick and sets up a dictionary to store the states of various buttons and analog sticks. The getJS function updates this dictionary based on joystick events, allowing the program to respond to user inputs.

**Description**: The joystick module uses the Pygame library to read inputs from a joystick, allowing manual control of the car. It captures button presses and analog stick movements.

**Code Explanation**: The joystick module initializes Pygame and sets up a dictionary to store the states of various buttons and analog sticks.

1. **Initialization**:
   * The joystick is initialized using Pygame, and a dictionary named buttons stores the state of each button and analog stick axis.
2. **Updating Joystick State**:
   * The getJS function processes joystick events to update the buttons dictionary.
   * It handles different event types, such as axis motion (for analog sticks) and button presses/releases.
3. **Returning Button States**:
   * The getJS function returns the current state of all buttons or a specific button if a name is provided.

**Challenges and Solutions**: One challenge was ensuring responsive joystick input handling without lag. This was addressed by optimizing the event loop and ensuring that the joystick state is updated frequently.

#### 3.4.2.3 Webcam Module

The webcam module utilizes the OpenCV library and picamera2 to capture images from the Raspberry Pi camera. It configures the camera, captures images, and optionally displays them.

import cv2

from picamera2 import Picamera2

picam2 = Picamera2()

picam2.preview\_configuration.main.size = (1920, 1080)

picam2.preview\_configuration.main.format = "RGB888"

picam2.preview\_configuration.align()

picam2.configure("preview")

picam2.start()

def getImg(display=True, size=[240, 120]):

img = picam2.capture\_array()

img = cv2.resize(img, (size[0], size[1]))

if display:

cv2.imshow('IMG', img)

cv2.waitKey(1)

return img

This module sets up the camera with a specific resolution and format, starts the camera, and defines the getImg function to capture and resize images. If display is set to True, it displays the captured image using OpenCV's imshow function.

**Description**: The webcam module utilizes the OpenCV library and picamera2 to capture images from the Raspberry Pi camera. It configures the camera, captures images, and optionally displays them.

**Code Explanation**: The webcam module sets up the camera with specific configurations and defines a function to capture and display images.

1. **Camera Setup**:
   * The camera is configured with a resolution of 1920x1080 and RGB888 format using the picamera2 library.
2. **Image Capture**:
   * The getImg function captures an image, resizes it, and optionally displays it using OpenCV.
   * If display is set to True, the image is shown in a window.

**Challenges and Solutions**: Configuring the camera and handling image capture required ensuring compatibility between the picamera2 library and OpenCV. Any discrepancies in image format or resolution were resolved by carefully configuring the camera settings.

#### 3.4.2.4 Haar cascade classifier module

For the purpose of detecting stop signs and other objects, we chose to use a method called Haar cascade instead of traditional detection models due to its accuracy and ease of use. The Haar cascade classifier is a machine learning-based approach that has been extensively trained to recognize specific objects, such as stop signs and car plates, in images. This method offers several advantages over traditional detection methods, including enhanced robustness in varying lighting conditions and orientations, higher detection accuracy, and faster processing speeds.

Traditional detection methods, such as template matching or edge detection, often struggle with variations in object appearance and environmental conditions. In contrast, Haar cascades utilize a large set of positive and negative images during training to build a robust model capable of detecting objects with high precision. Additionally, Haar cascades are computationally efficient, enabling real-time object detection, which is crucial for applications like autonomous driving.

**Haar Cascades: Definition and Advantages**

Haar cascades are a machine learning-based approach used for object detection, introduced by Paul Viola and Michael Jones in their 2001 paper "Rapid Object Detection using a Boosted Cascade of Simple Features." The Haar cascade classifier is trained with a large number of positive images (images containing the objects to be detected) and negative images (images without the objects). The classifier uses features similar to Haar wavelets, which involve calculating the difference in intensity between adjacent rectangular regions in an image. Once trained, the classifier can efficiently detect objects in new images by scanning them at different scales and positions.

**Advantages of Haar cascades include:**

**1. Speed:** Haar cascades perform real-time object detection, making them suitable for applications such as video surveillance and autonomous vehicles.

**2. Accuracy:** With proper training, Haar cascades can accurately detect objects even under varying lighting conditions and orientations.

**3. Efficiency:** The method is computationally efficient due to the use of integral images, which allow for quick calculation of feature values.

Compared to traditional sign detection methods, which might rely on template matching or edge detection, Haar cascades are more robust and adaptable to variations in scale, orientation, and partial occlusion of objects.

**Stop sign and plate detection code Functionality Overview**

The codes are designed to detect stop signs and plates in a video feed using the OpenCV library, where both codes function exactly the same way but differs in step 1 that is mentioned below, where each detected object (stop sign and plate) has their own haar cascade classifier. The flow of the code can be summarized as follows:

**1. Loading the Classifier**: The code starts by loading a pre-trained Haar cascade classifier specifically designed for stop sign or plate detection. These classifiers have been trained using images of the object to be detected to recognize their unique features.

**2. Capturing Video Feed:** The code initiates video capture from the default camera. This enables the system to process live video frames for real-time detection.

The code continuously processes each frame from the video feed. Each frame is converted to grayscale because the Haar cascade classifier operates on single-channel images for efficiency.

**4. Detecting Stop Signs:** The classifier is applied to the grayscale frame to detect the required object. It scans the image at multiple scales to identify potential stop signs of various sizes.

**5. Annotating Detections:** For each detected object, the code draws a rectangle around it on the original frame. Additionally, it labels the detected region with the text "Stop Sign" or “plate” to visually indicate the detection.

**6. returning the labeled image along with the number of detected objects:** The processed frame, now annotated with rectangles and labels for detected stop signs, is displayed in a window. This provides real-time feedback on the detection process.

**7. taking actions:** as for the actions taken after detection, the car stops for a predefined time interval when detecting a stop sign and then continues moving or it remains still until the stop sign is removed, and as for the plate detection the coordinates of the plate are interpreted so that the car moves according to these coordinates, for example if the plate is locates on the left side of the image then the car will move left until the plate is set to the center of the image, and the same concept applies if the plate is detected on the right side of the image.

This process demonstrates how the combination of a pre-trained Haar cascade classifier and OpenCV's video processing capabilities can be used to implement a real-time stop sign detection system, which is essential for applications like autonomous driving where timely and accurate object detection is critical.

#### 3.4.2.5 Integration

* **Interfacing Modules**: The integration of these modules is crucial for the autonomous driving car's functionality. The motor module controls the car's movement, the joystick module allows for manual input, and the webcam module captures images for visual processing.
* **Overall Functionality**: Together, these modules enable the car to navigate autonomously or under manual control, detect and avoid obstacles, and capture visual data for further processing.
* **IMPORTANT NOTE**: due to the low processing speed of our car, the concept ofplate tracking is canceled because it turns out that we need a higher processing power to detect the changes of plate direction in real time.
* **Summary**: The motor, joystick, and webcam modules are essential components of the autonomous driving car project. Each module was carefully implemented and integrated to ensure smooth operation and responsiveness. Challenges were addressed through precise configuration and optimization, resulting in a functional and robust system.

#### 3.4.3 Data Collection

Data collection for our self-driving robot car project involved capturing images of the road environment in both forward and backward directions, totaling 8 sessions for each direction. Each session comprised 600 images, providing a comprehensive dataset for training and validating our deep learning models.

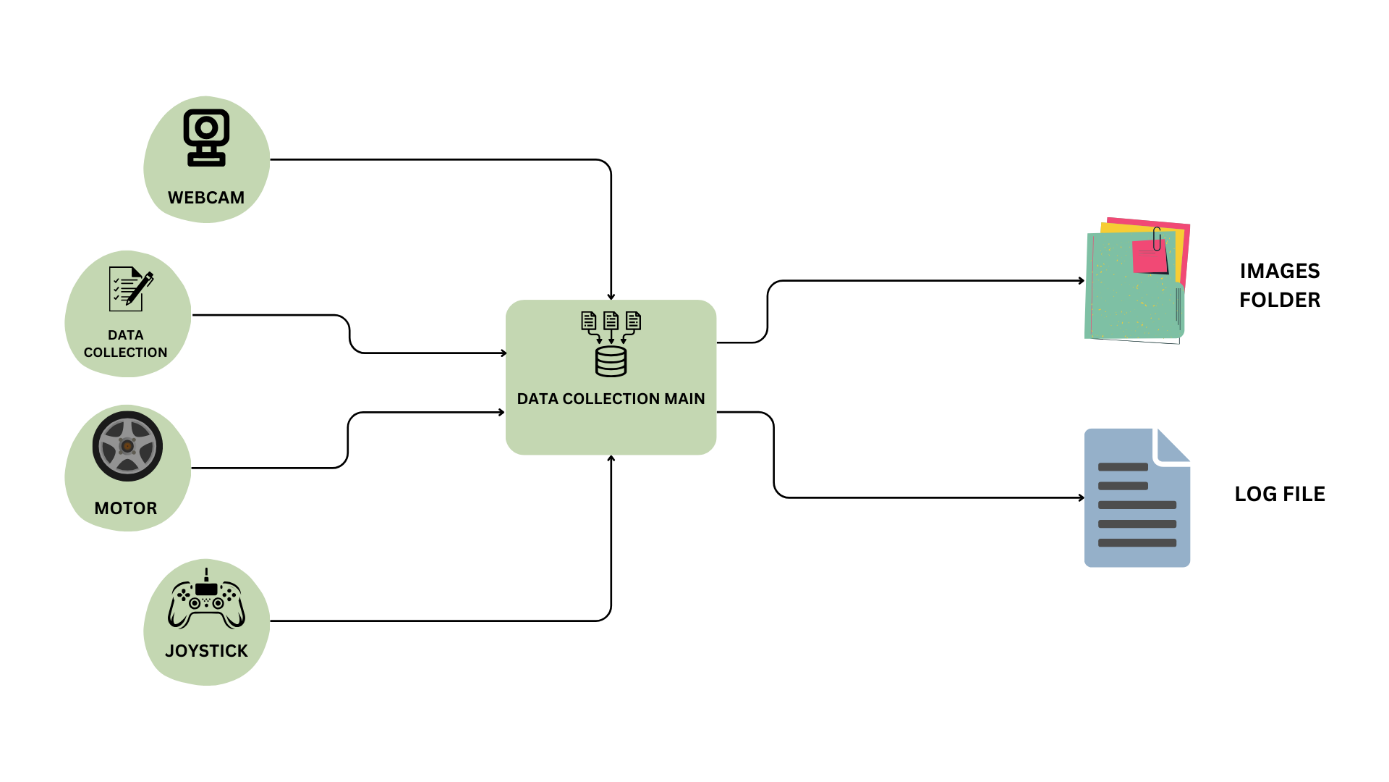
The road environment was simulated using hard papers lined with white tapes, creating lane markings that guided the car's navigation algorithms. By capturing images from the car's mounted Raspberry Pi Camera Module 2, we obtained high-resolution visual data essential for tasks such as lane detection, object recognition, and traffic sign classification.

Figure 12- Data collection diagram

The provided diagram illustrates the data collection process for an autonomous driving car project, emphasizing the interaction between various hardware components and the central data collection system. At the core of the system is the "Data Collection Main" module, which serves as the hub for integrating data from multiple sources.

1. **Webcam**: The webcam is responsible for capturing visual data, providing real-time images of the road. This visual input is crucial for tasks such as path detection and stop sign recognition. The captured images are directed to the Data Collection Main module, where they are stored in an "Images Folder" for further analysis and model training using the Data Collection file.
2. **Joystick**: The joystick module allows for manual control and input, enabling the user to interact with the car directly. This input is essential for tasks like calibrating the system, performing manual overrides, or testing the car's responsiveness. Data from the joystick is fed into the Data Collection Main module, which records the inputs and uses them to adjust the car's movement accordingly, the main use of this module is gathering steering values corresponding to each image whenever t euser press the “**share**” joystick button.
3. **Motor**: The motor module controls the car's physical movement, including acceleration, braking, and turning. It receives commands from the Data Collection Main module, which processes inputs from both the webcam and joystick to determine the appropriate motor actions. The motor module ensures that the car moves accurately according to the planned path or manual inputs.
4. **Data Collection**: This aspect involves gathering various data points from the car's operation, including images and steering angles. The collected data is logged in a "Log File" for record-keeping and future reference.

Overall, the diagram highlights the interplay between the webcam, joystick, motor, and data collection modules, all centralized through the Data Collection Main module. This integrated approach ensures that the autonomous driving car operates smoothly, with all relevant data being captured, processed, and stored efficiently. This setup not only facilitates real-time control and monitoring but also aids in the continuous improvement of the autonomous system through data-driven insights.

#### 3.4.3.1 Data collection module

*"""  
- This module saves images and a log file.  
- Images are saved in a folder.  
- Folder should be created manually with the name "DataCollected"  
- The name of the image and the steering angle is logged  
in the log file.  
- Call the saveData function to start.  
- Call the saveLog function to end.  
- If runs independent, will save ten images as a demo.  
"""*import pandas as pd  
import os  
import cv2  
from datetime import datetime  
  
  
global imgList, steeringList  
countFolder = 0  
count = 0  
imgList = []  
steeringList = []  
  
#GET CURRENT DIRECTORY PATH  
myDirectory = os.path.join(os.getcwd(), 'DataCollected')  
# print(myDirectory)  
  
# CREATE A NEW FOLDER BASED ON THE PREVIOUS FOLDER COUNT  
while os.path.exists(os.path.join(myDirectory,f'IMG{str(countFolder)}')):  
 countFolder += 1  
newPath = myDirectory +"/IMG"+str(countFolder)  
os.makedirs(newPath)  
print('folder created: ',newPath)  
  
# SAVE IMAGES IN THE FOLDER  
def saveData(img,steering):  
 global imgList, steeringList  
 now = datetime.now()  
 timestamp = str(datetime.timestamp(now)).replace('.', '')  
 #print("timestamp =", timestamp)  
 fileName = os.path.join(newPath,f'Image\_{timestamp}.jpg')  
 cv2.imwrite(fileName, img)  
 imgList.append(fileName)  
 steeringList.append(steering)  
  
  
# SAVE LOG FILE WHEN THE SESSION ENDS  
def saveLog():  
 global imgList, steeringList  
  
 rawData = {'Image': imgList,  
 'Steering': steeringList}  
 df = pd.DataFrame(rawData)  
 df.to\_csv(os.path.join(myDirectory,f'log\_{str(countFolder)}.csv'), index=False, header=False)  
 print('Log Saved')  
 print('Total Images: ',len(imgList))

This Python script is designed to save images and corresponding steering angles for autonomous driving data collection. It first creates a new folder named "IMG{countFolder}" within a manually created "DataCollected" directory, where `{countFolder}` is a unique number assigned to avoid overwriting existing folders. The `saveData` function saves an image and its steering angle to this folder, naming the image file with a timestamp to ensure uniqueness. The image file paths and steering angles are recorded in global lists. When the data collection session ends, the `saveLog` function generates a CSV log file containing these paths and angles, and saves it in the "DataCollected" directory with a unique log file name.

#### 3.4.3.2 Data collection main

import WebcamModule as wM  
import DataCollectionModule as dcM  
import JoyStickModule as jsM  
import MotorModule as mM  
import cv2  
from time import sleep  
  
maxThrottle = 0.25  
motor = mM.Motor(7,1,12,16,20,21)  
record = 0  
  
while True:  
 joyVal = jsM.getJS()  
 #print(joyVal)  
 steering = joyVal['axis1']  
 throttle = joyVal['o']\*maxThrottle  
 if joyVal['share'] == 1:  
 if record ==0: print('Recording Started ...')  
 record +=1  
 sleep(0.300)  
 if record == 1:  
 img = wM.getImg(True,size=[240,120])  
 dcM.saveData(img,steering)  
 elif record == 2:  
 dcM.saveLog()  
 record = 0  
 motor.move(throttle,steering)  
 cv2.waitKey(1)

This script integrates various modules to control an autonomous driving setup with real-time data recording. It continuously reads joystick inputs to adjust steering and throttle. If the 'share' button on the joystick is pressed, it toggles recording mode. When recording starts, the script captures images from a webcam, logs them with their steering angles using the `DataCollectionModule`, and saves them to a specified folder. After the recording session, indicated by a second press of the 'share' button, it saves the log file and resets the recording status. Meanwhile, it controls the motor to adjust the vehicle's movement based on the throttle and steering values derived from the joystick.

### 3.3.4 Training

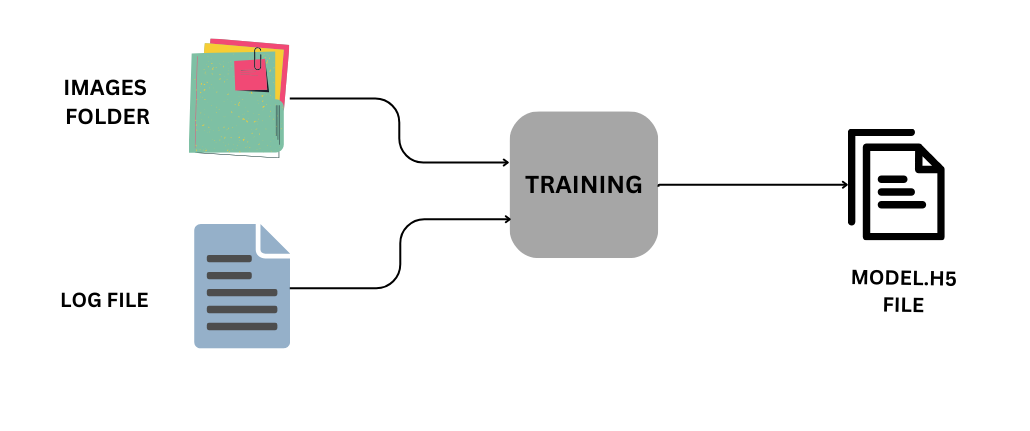


Figure 13- Training diagram

The training of deep learning models lies at the heart of enabling our self-driving robot car to perceive and interpret its environment accurately. Leveraging TensorFlow framework and GPU-accelerated computing resources, our methodology focuses on optimizing model architectures, selecting appropriate loss functions, and fine-tuning hyperparameters to achieve high prediction accuracy and robust performance in real-world driving scenarios.

Model training commences with the initialization of a pre-designed Convolutional Neural Network (CNN) architecture tailored for tasks such as predicting the steering angle based on the fed image. The CNN architecture consists of multiple layers of convolutional, pooling, and fully connected nodes, designed to automatically extract hierarchical features from input images while preserving spatial relationships and semantic context critical for scene understanding.

#### 3.3.4.1 Dataset

My dataset for training the autonomous driving model is composed of a collection of images captured through a webcam, organized within a dedicated folder structure. Accompanying these images is a meticulously maintained log file that maps each image to its corresponding steering angle. This log file plays a crucial role in ensuring the model learns the association between visual inputs and the required steering actions. The dataset is structured to enable the model to understand and predict the appropriate steering angle for any given image it processes. This association is fundamental for developing a robust autonomous driving system that can accurately navigate based on real-time visual data. By training on this dataset, the model is expected to gain the capability to interpret new images captured by the webcam and determine the correct steering angle, thereby facilitating smooth and safe driving decisions.

The process of gathering a high-quality and accurate dataset is fundamental to the success of training an autonomous driving model. For my project, the dataset was collected meticulously using a webcam to capture images in various driving conditions. The accuracy of the data collection process ensures that the model can generalize well to real-world scenarios.

* **Dataset Gathering Process**

1. **Setup and Calibration**: The webcam was carefully positioned to mimic the driver's perspective, ensuring that the captured images accurately represent the view a human driver would have. This setup was crucial to provide the model with relevant visual inputs.
2. **Diverse Conditions**: Images were captured under various lighting conditions, weather scenarios, and different types of roads. This diversity helps the model learn to handle a wide range of driving environments, making it more robust and versatile.
3. **Real-time Data Logging**: As each image was captured, it was immediately logged with the corresponding steering angle. This real-time logging ensured that the association between the visual input and the steering command was precise and accurate.
4. **Consistent Frame Rate**: The images were captured at a consistent frame rate, providing a steady stream of data that mimics the continuous nature of driving. This consistency helps the model understand the flow of visual information over time.

* **Importance of Accuracy**

1. **Model Performance**: Accurate data collection directly impacts the model's ability to learn and generalize. High-quality images paired with precise steering angles enable the model to make reliable predictions.
2. **Safety**: In the context of autonomous driving, safety is paramount. An accurate dataset ensures that the model can predict steering angles correctly, reducing the risk of errors that could lead to accidents.
3. **Real-world Applicability**: By capturing a wide range of driving conditions and scenarios, the dataset ensures that the model can handle the complexities of real-world driving. This applicability is essential for deploying the model in practical autonomous driving applications.

* **Image Example**

An example image from the dataset gathering process will illustrate the setup and the type of data collected. The image shows the camera's viewpoint, capturing the road ahead with a clear timestamp and associated steering angle logged. This precise and methodical approach to data collection forms the backbone of the training process, leading to a reliable and effective autonomous driving model.

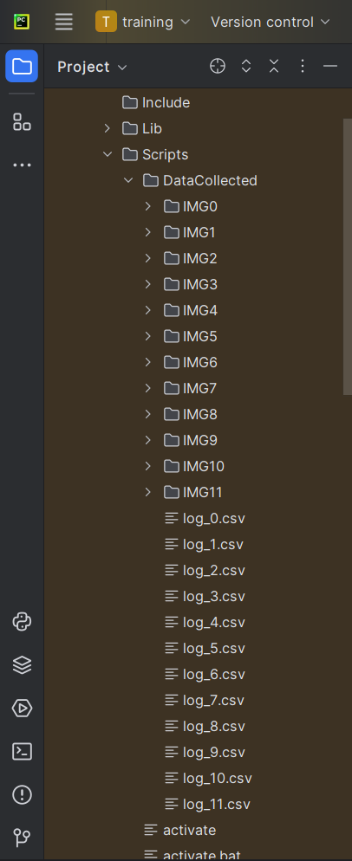


Figure 14- 'DataCollected' folder overview

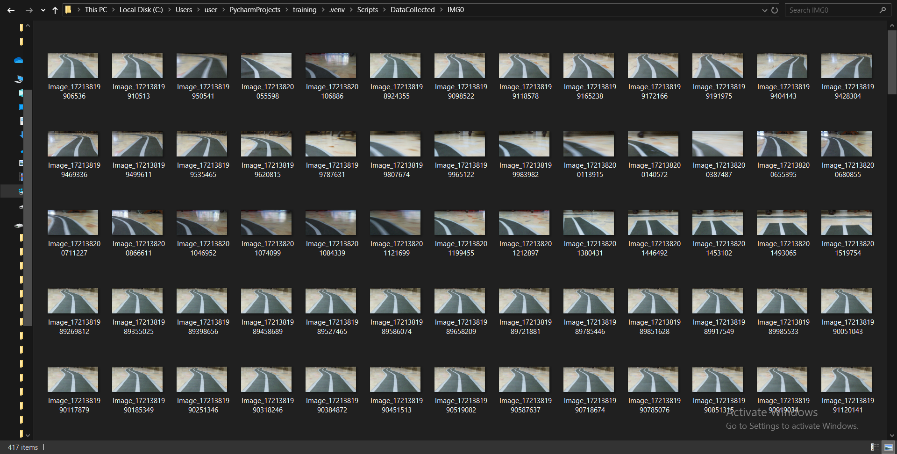


Figure 15- Images folder overview

* The first image shows the “DataCollected” folder that contains both images folders and their corresponding log.csv file, each images folder contains about 450 to 500 different road images, each with its corresponding steering angle saved in the log file in order to be used later for training our model.
* The dataset is organized into two distinct categories: one containing images of the vehicle moving forward on the road, and the other containing images of the vehicle moving in the opposite direction. Each category comprises five folders, thereby providing a comprehensive collection of driving scenarios. This structured approach ensures that the model is exposed to a wide range of road curvatures and driving directions, facilitating the learning process. By training on this diverse dataset, the autonomous driving system can develop a more robust understanding of various steering maneuvers, enhancing its ability to navigate different road conditions effectively.
* The second image is an overview of folder “IMG0” showing an overview of how the dataset looks like and how images where collected.
* And the below image is an overview of the log\_0.csv file that shows each image followed by it steering angle value, negative values means a left turn and positive value indicates a right turn, finally zero value indicates a straight forward path with no steering.

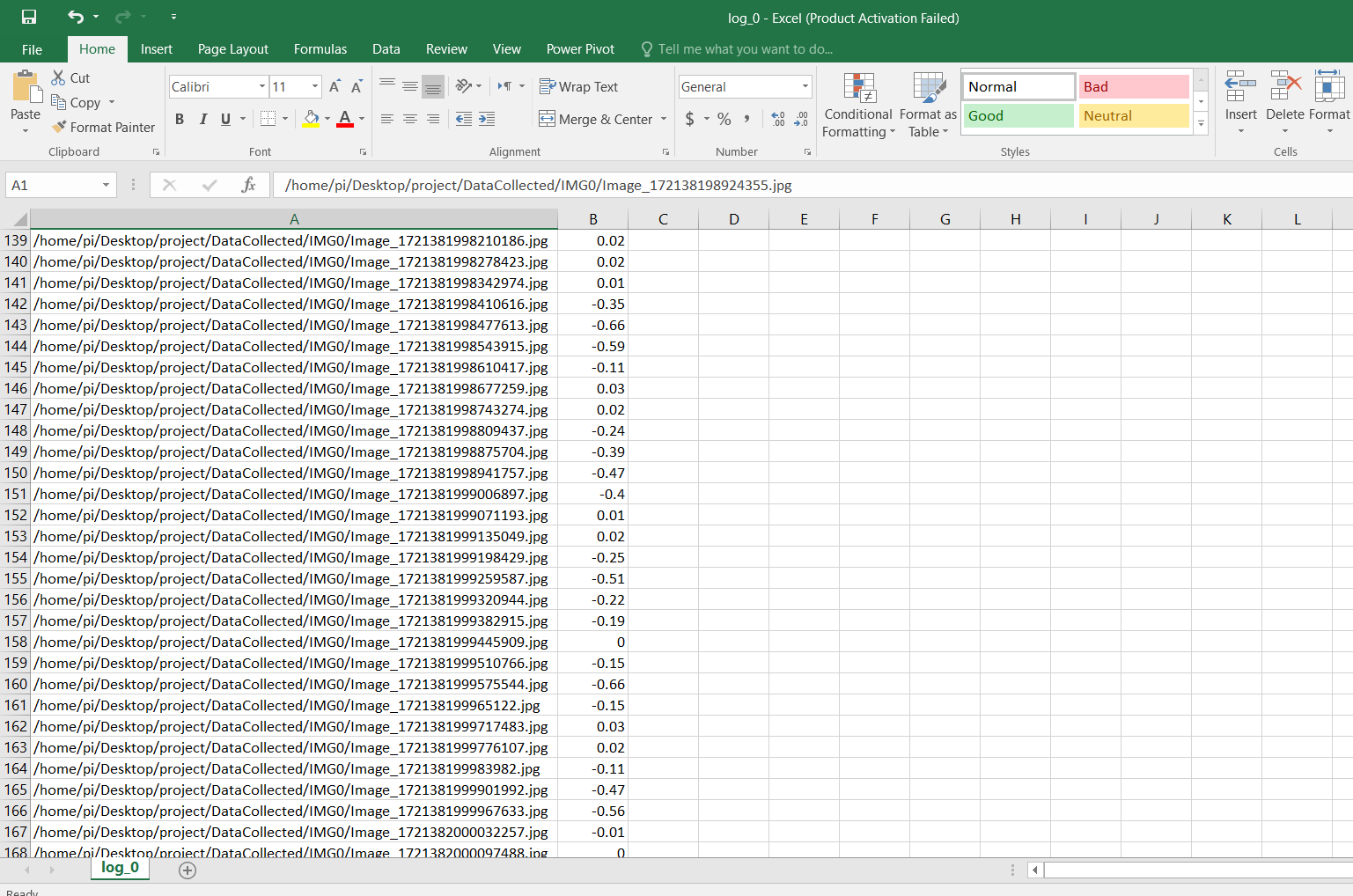


Figure 16- csv file overview

In summary, the careful and accurate gathering of the dataset is a cornerstone of developing a robust autonomous driving system. Each step in the data collection process is designed to ensure that the model receives high-quality, relevant information, enabling it to make safe and reliable driving decisions.

#### 3.3.4.2 Training overflow

The training workflow for developing my autonomous driving model involves several crucial steps, each contributing to the overall accuracy and robustness of the system. It begins with data preprocessing, where the captured images are resized, normalized, and augmented to enhance the model's ability to generalize across various driving conditions. Following preprocessing, the data is split into training and validation sets to ensure the model's performance is evaluated fairly. The core of the workflow is the model training phase, where a deep learning model is trained using the processed images and corresponding steering angles. During this phase, various techniques such as regularization and dropout are employed to prevent overfitting and improve generalization. The training process involves iterative optimization using backpropagation and gradient descent to minimize the prediction error. Throughout training, the model's performance is continuously monitored on the validation set, allowing for hyperparameter tuning and adjustments to the model architecture as needed. Finally, the trained model is evaluated on a test set to assess its real-world applicability, ensuring it can reliably predict steering angles based on new images. This comprehensive training workflow is designed to create a robust autonomous driving model capable of navigating diverse driving environments safely and efficiently.

The training process for my autonomous driving model is meticulously organized into 10 distinct sections, each designed to address specific aspects of the development workflow. These sections are detailed below to provide a comprehensive understanding of the entire process:

1. **Import data info**

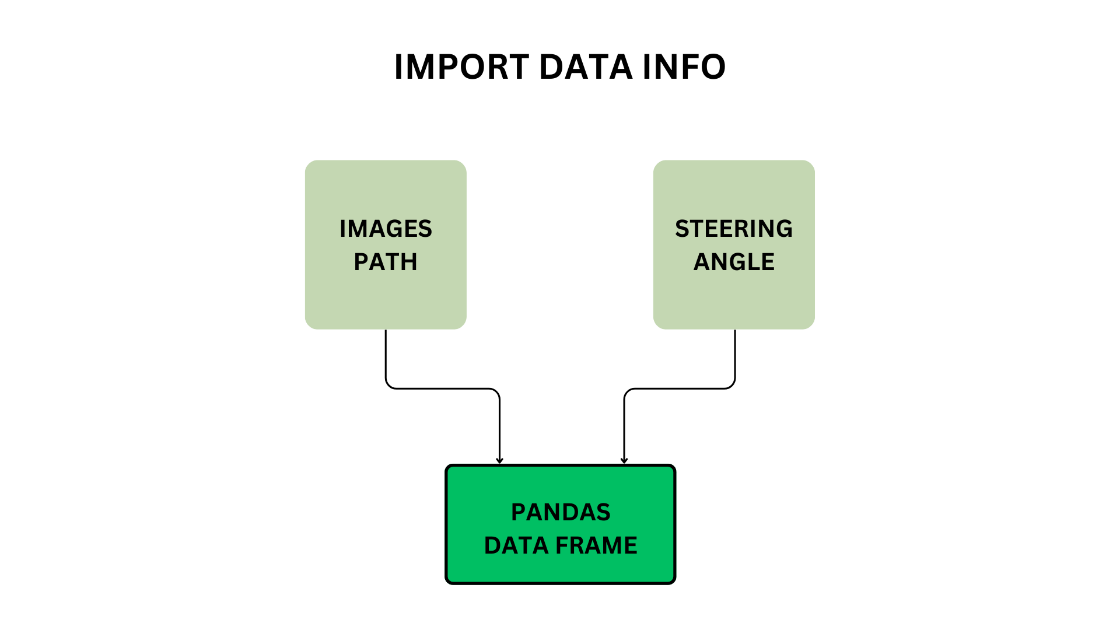
****

Figure 17- Data import diagram

As illustrated here, the initial step in our process involves importing our dataset, which comprises the images and their corresponding steering angles. These images were captured using a webcam during the data collection phase, while the steering angles were logged in real-time. To manage and manipulate this data effectively, we will utilize the Pandas library, which provides powerful data handling capabilities.

The primary goal at this stage is to ensure that our dataset is balanced, meaning that we have a well-distributed representation of different steering angles. An imbalanced dataset, where certain steering angles are overrepresented, can lead to a biased model that performs poorly in real-world scenarios. To achieve this balance, we will load the images and steering angles into a Pandas DataFrame, a tabular data structure that facilitates easy data manipulation and analysis.

By converting our dataset into a Pandas DataFrame, we can leverage various techniques to balance the data. For instance, we can visualize the distribution of steering angles and apply methods such as undersampling, oversampling, or augmentation to ensure that each steering angle is adequately represented. This balanced DataFrame will then serve as the foundation for training our model, ensuring that it learns to make accurate predictions across a wide range of driving conditions.

In summary, importing our dataset and converting it into a balanced Pandas DataFrame is a crucial step in preparing our data for effective training. This approach not only organizes our data in a structured format but also ensures that our model is trained on a representative and unbiased dataset.

1. **Data balance and visualization**

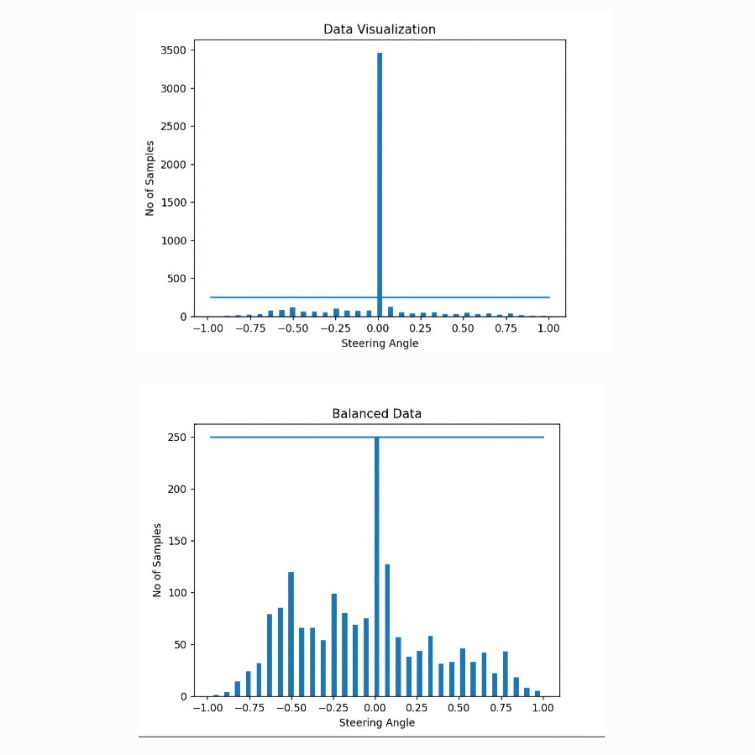
Next, we will proceed to visualize the dataset to gain insights into the distribution of the steering angles. Visualization is a crucial step as it allows us to understand the data better and identify any potential imbalances that could affect the training process.

Figure 18- Data balance and visualization

To begin, we will plot the steering angles to observe their distribution. This can be done using visualization libraries such as Matplotlib or Seaborn, which provide intuitive and easy-to-interpret plots. By creating a histogram of the steering angles, we can see how frequently each angle occurs in our dataset. This visual representation will help us identify any overrepresented or underrepresented angles.

Once we have visualized the steering angles, we will take steps to balance the dataset. If the histogram reveals that certain steering angles are disproportionately represented, we will apply techniques to address this imbalance. These techniques may include undersampling overrepresented angles, oversampling underrepresented ones, or using data augmentation methods to create additional samples for underrepresented angles.

Balancing the dataset is essential for ensuring that our model does not become biased towards certain steering angles and can generalize well to various driving conditions. By addressing any imbalances at this stage, we can improve the model's performance and reliability.

1. **Prepare for preprocessing**

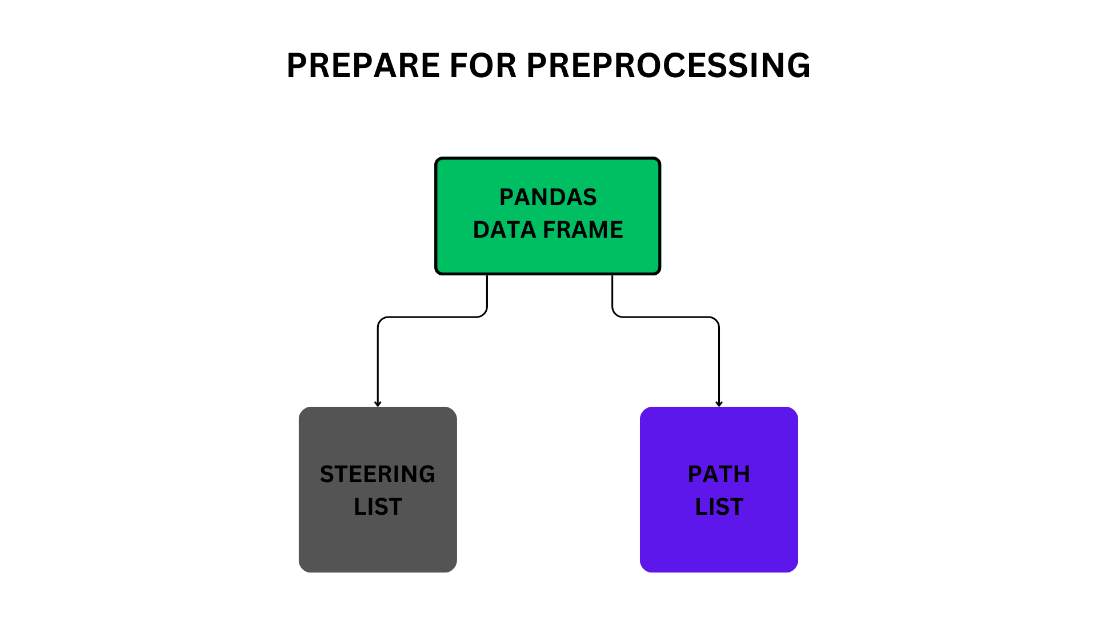


Figure 19- preprocessing preparation diagram

Next, we will prepare the dataset for processing. This involves converting the data from the Pandas DataFrame format into a list format, making it easier to work with during the training phase.

To begin, we will extract the images and their corresponding steering angles from the DataFrame and store them in separate lists. This conversion is straightforward and allows us to use the data in a more flexible manner. The list format is particularly useful because it simplifies the process of iterating through the dataset and feeding it into the training pipeline.

By extracting the images and steering angles into lists, we can leverage various Python functions and libraries more efficiently. For instance, the print function can be used to quickly inspect the data, and other built-in Python functionalities can be employed to manipulate and process the data as needed.

1. **Data splitting**

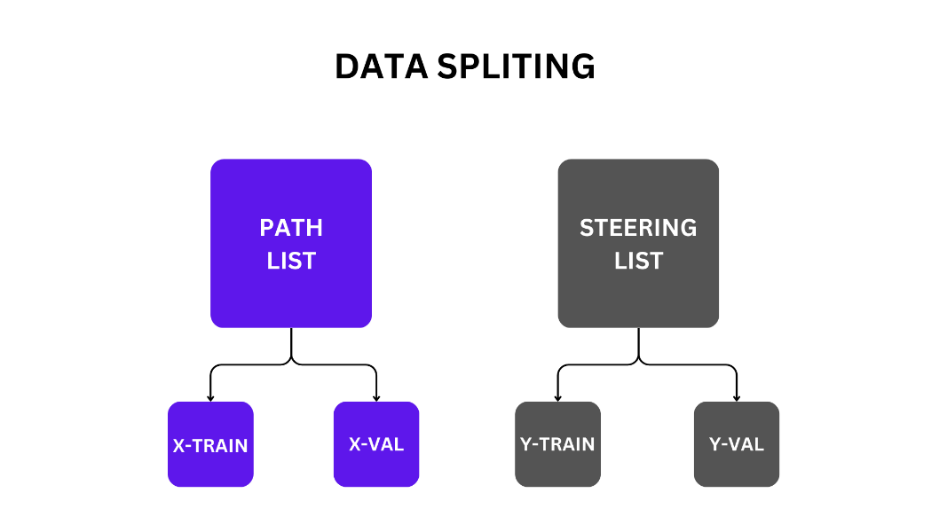
****

Figure 20- Data splitting diagram

After converting the data into lists, we will use the split method to divide the dataset into training and validation sets, ensuring that the model's performance can be accurately evaluated. The validation set, denoted as `x\_val` for validation images and `y\_val` for their corresponding steering angles, will help us assess how well the model generalizes to unseen data. The training set, labeled `x\_train` for training images and `y\_train` for the associated steering angles, is used to train the model. By splitting the data this way, we ensure that each image in the training set (`x\_train`) has a corresponding steering angle (`y\_train`), and similarly, each image in the validation set (`x\_val`) has a corresponding steering angle (`y\_val`). This partitioning is crucial as it allows us to monitor the model's performance on a separate validation set, ensuring it does not overfit to the training data and maintains accuracy when exposed to new, unseen images. And the most recommended data splitting ratio, as suggested by many machine learning practitioners and researchers, is 80% for training data and 20% for validation data. This approach involves using **80%** of the dataset to train the model, allowing it to learn patterns and relationships within the data. The remaining **20%** is used to validate the model, providing an independent dataset to evaluate the model's performance and generalization capabilities. This split, advocated by experts in the field, ensures that the model is exposed to a substantial amount of data during training while also being rigorously tested on unseen data to prevent overfitting and ensure robust performance.

* Total images: 4937
* Removed Images: 3214
* Remaining Images: 1723 (after balancing)
* Total Training Images: 1378
* Total Validation Images: 345

1. **Data augmentation**

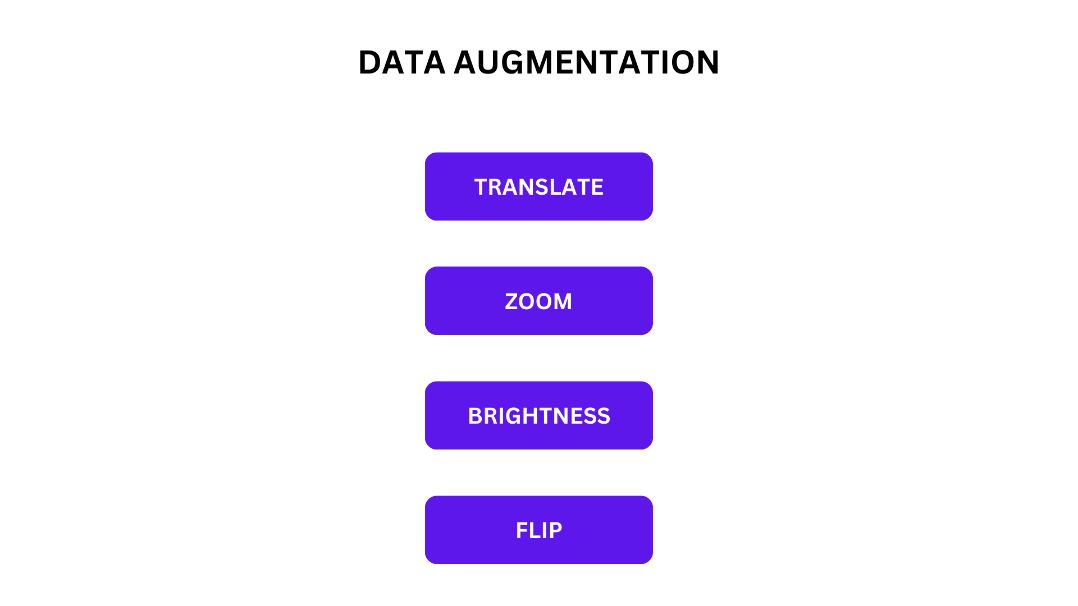
****

Figure 21- Data augmentation diagram

Given the limited size of our dataset, we will employ data augmentation techniques to enhance its diversity and improve the model's ability to generalize. Specifically, we will apply several augmentation methods: translating images to shift them left or right, zooming in and out to vary the scale, and adjusting the brightness to simulate different lighting conditions. Additionally, we may flip the images horizontally to create a mirrored effect. When flipping images, it is crucial to also flip the corresponding steering angles to maintain the correct relationship between the image and the driving direction. For instance, if the road in the original image appears to curve right, flipping the image would make it curve left, so the steering angle must be adjusted accordingly to reflect this change. These augmentation techniques will help create a more robust and versatile dataset, allowing the model to perform better in various driving scenarios.

1. **Data preprocessing**

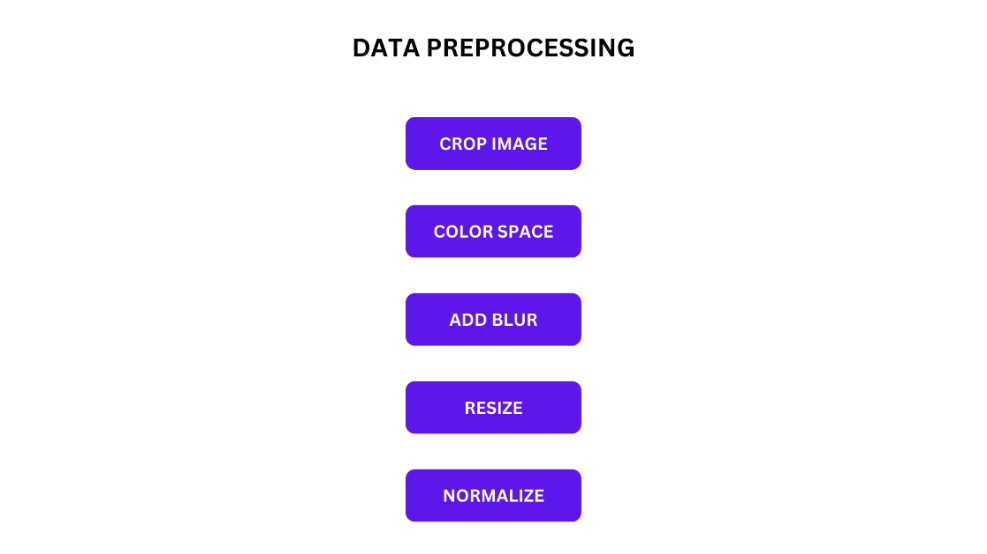


Figure 22- Data preprocessing

For preprocessing, we will follow a series of steps to ensure that our images are optimized for training. First, we will crop the images to focus solely on the relevant region of interest—the area where the road is visible—while removing extraneous parts that do not contribute to the driving task. This cropping helps in reducing the computational load and improving the model's focus on pertinent features.

Next, we will convert the images to a suitable color space. Depending on the model's requirements, this could involve transforming the images from RGB to YUV or another color space that enhances feature extraction and simplifies processing.

To further refine the images, we will apply a blur effect, which can help in reducing noise and improving the model's robustness by smoothing out minor variations. This step is particularly useful for enhancing the model's ability to generalize from the data.

Following blurring, we will resize the images to a consistent resolution. This standardization is important for ensuring that all input images have the same dimensions, facilitating more effective training and model performance.

Finally, we will normalize the pixel values of the images. Normalization adjusts the pixel values to a common scale, typically between 0 and 1, which helps in stabilizing the training process and improving convergence. By applying these preprocessing steps, we ensure that the images are well-prepared for training, leading to a more accurate and efficient model.

1. **Model creation**

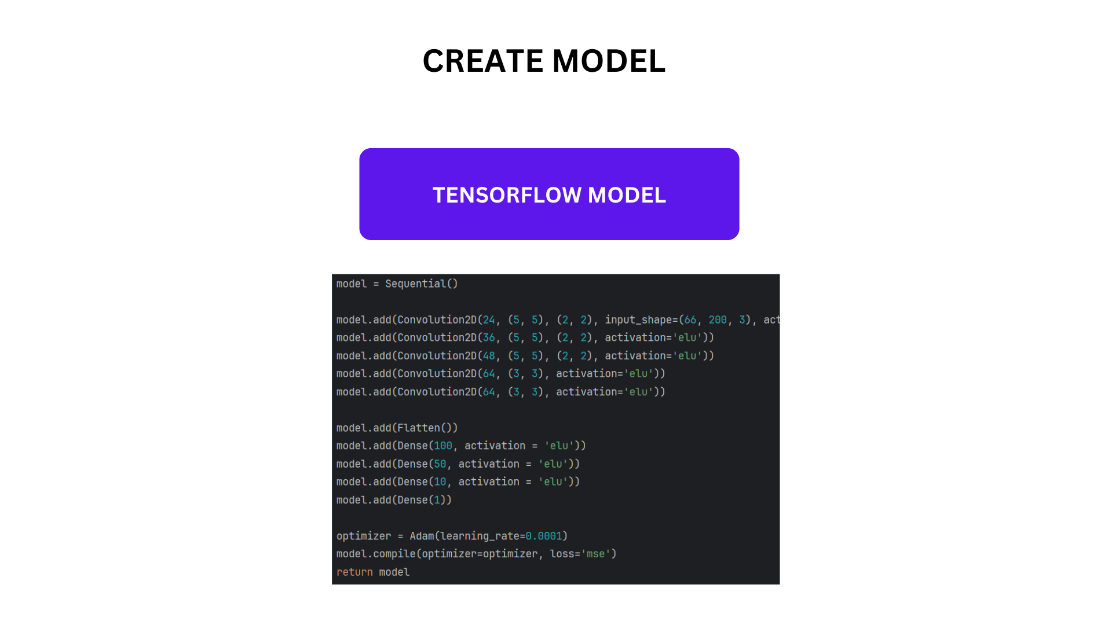
****

Figure 23- Model creation diagram

After preprocessing the images, we will convert them into tensors suitable for input into our neural network model. This model will include several key layers designed to process and learn from the image data effectively. Convolutional layers will apply filters to the input images to automatically extract essential features such as edges and textures, capturing the spatial hierarchies within the images. Activation functions like ReLU will introduce non-linearity, enabling the model to learn complex patterns. Pooling layers will reduce the spatial dimensions of the feature maps, simplifying the data while preserving crucial information, which helps in minimizing computational complexity and preventing overfitting. Dense layers, located towards the end of the network, will combine the features extracted by the convolutional layers to make final predictions. Additionally, the model may include other layers such as dropout for regularization, normalization layers to stabilize training, and fully connected layers for generating outputs. By incorporating these layers, the neural network will be equipped to effectively process the tensor data, learn from it, and make accurate predictions regarding steering angles.

1. **Training the model**

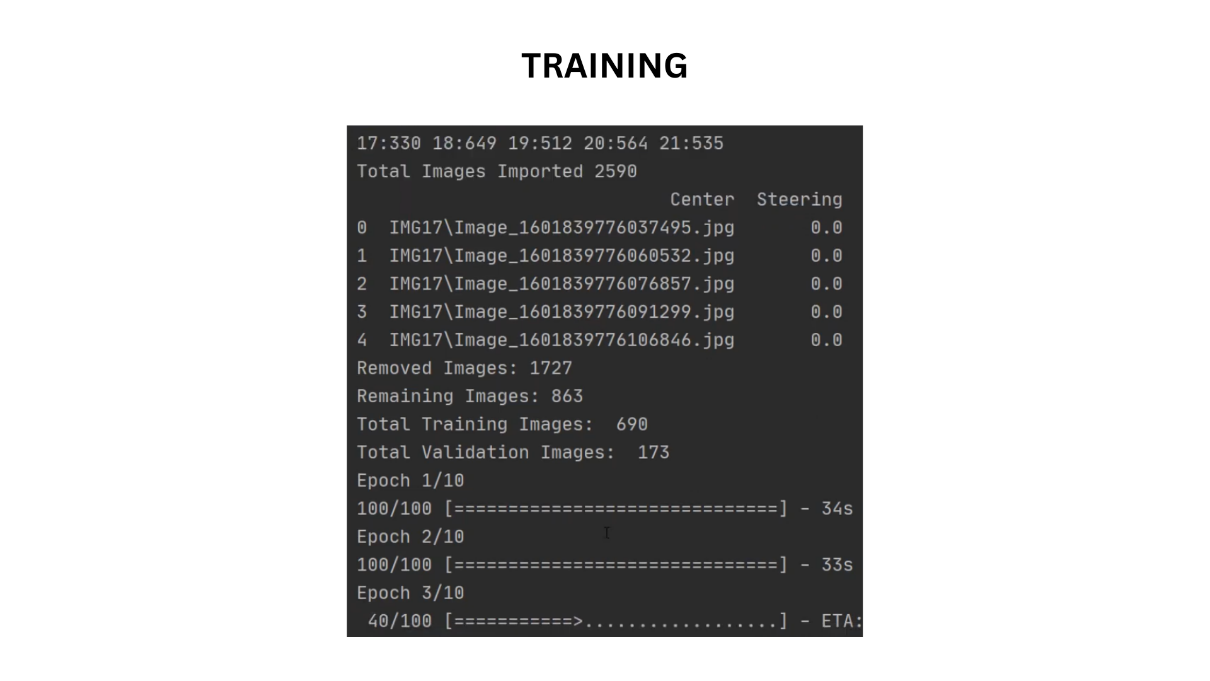


Figure 24- training process overview

After setting up the model and preparing the data, we will proceed to the training phase. During this stage, the model will be trained for thirty epochs. Training involves passing the entire dataset through the network multiple times to refine its parameters and enhance its performance. Specifically, we use the `model.fit` method, where `dataGen(xTrain, yTrain, 100, 1)` generates batches of training data with a batch size of 100, and `steps\_per\_epoch=100` indicates how many batches are processed in each epoch. We train the model for 20 epochs, during which it iterates through the training data to minimize the loss and improve accuracy. Additionally, we validate the model’s performance on unseen data using `dataGen(xVal, yVal, 50, 0)`, with `validation\_steps=50` to evaluate its generalization ability. This comprehensive training process allows the model to effectively learn and predict steering angles from images.

# Define early stopping callback  
early\_stopping = EarlyStopping(monitor='val\_loss', patience=2, restore\_best\_weights=True)  
  
#### STEP 8 - TRAIN MODEL  
history = model.fit(dataGen(xTrain, yTrain, 100, 1),

steps\_per\_epoch=1200,

epochs=5,  
 validation\_data=dataGen(xVal, yVal, 50, 0),  
 callbacks=[early\_stopping],  
 validation\_steps=50,

verbose=1)

1. **Saving the model**

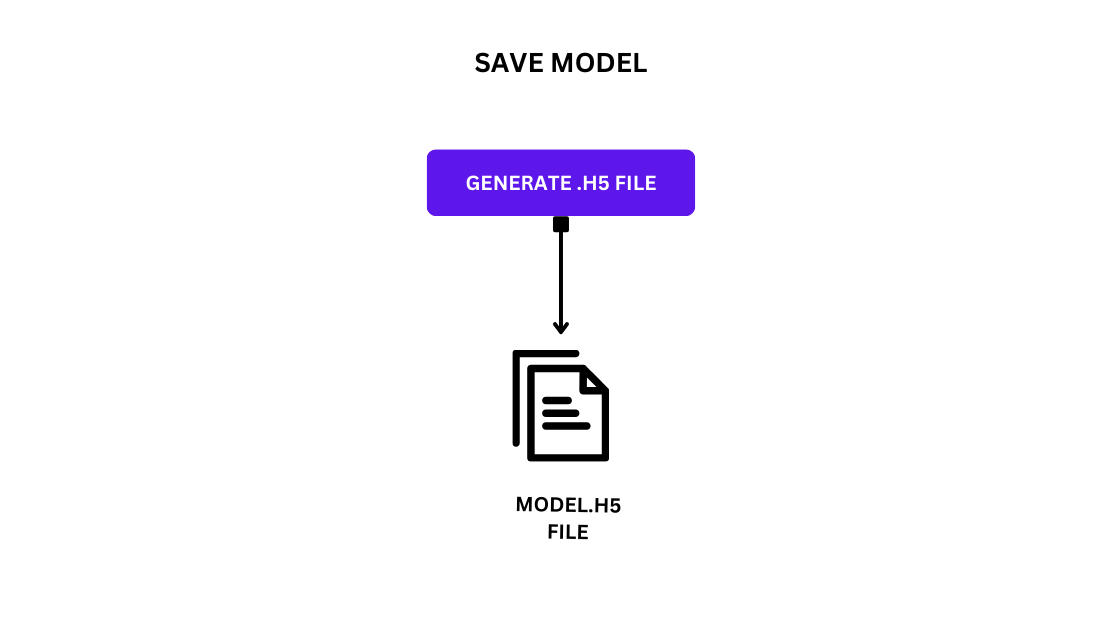


Figure 25- Save model diagram

After the training process is completed, we will save the trained model. This step involves storing the model's architecture, weights, and training configuration to a file, which in this case will be named `model.h5`. Saving the model ensures that it can be easily reloaded and used for future predictions or further training without the need to retrain from scratch.

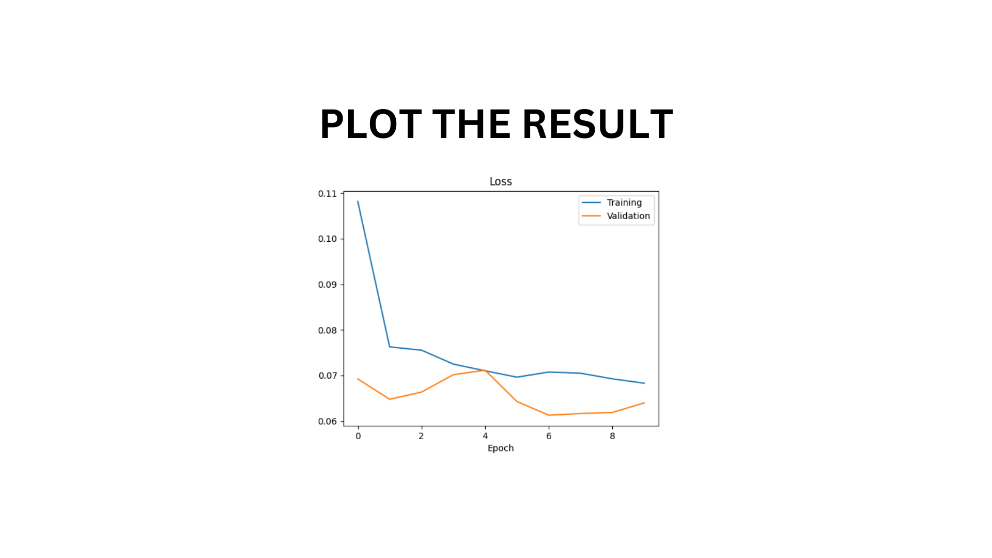
1. **Results plotting**

Figure 26- Results plotting overview

Finally, we will visualize the training results by plotting the loss curves for both the training and validation datasets. This step involves using Matplotlib to create a graph that displays how the loss evolves over each epoch for both training and validation. By plotting these curves, we can assess how well the model has learned and whether it is overfitting or underfitting. The plot will show the training loss decreasing as the model improves its performance on the training data, while the validation loss will indicate how well the model generalizes to unseen data. This visualization helps in understanding the effectiveness of the training process and in making any necessary adjustments to the model or training procedure.

## 3.6 Model Evaluation

Upon completing the training phase, it is essential to evaluate the autonomous driving model using several key metrics to ensure its effectiveness and reliability. These metrics include accuracy, precision, recall, and F1 score, MSE, MAE, RMSE. Each providing different insights into the model’s performance. But metrics like accuracy, F1-score, or precision are more suited for classification problems rather than regression tasks like predicting steering angles. For our model, focusing on MSE, MAE, and possibly RMSE would be most appropriate.

Below is a detailed explanation of each metric, including their equations and code for calculation:

**1.** **Mean Squared Error (MSE)**

**Definition:** Mean Squared Error (MSE) is a metric used to evaluate the performance of a predictive model by measuring the average squared difference between the predicted values and the actual values. In the context of predicting steering angles, MSE calculates the average of the squares of the differences between the predicted steering angles and the true steering angles.

**Mathematical Expression:**

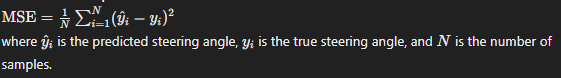


Figure 27- MSE equation

**Advantages:**

**Penalizes Larger Errors:** Since MSE squares the differences, larger errors are penalized more than smaller ones. This can be useful if large errors are particularly undesirable in your application.

**Mathematical Convenience:** Squaring the errors and then averaging makes MSE mathematically convenient for optimization algorithms, especially in gradient-based methods.

Disadvantages:

Sensitive to Outliers: Because errors are squared, MSE is highly sensitive to outliers or extreme values, which can disproportionately affect the error metric.

**2. Mean Absolute Error (MAE)**

**Definition:** Mean Absolute Error (MAE) measures the average magnitude of errors in a set of predictions, without considering their direction. It calculates the average of the absolute differences between predicted and actual values.

**Mathematical Expression:**

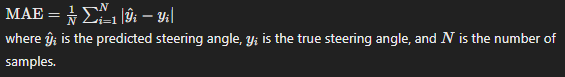


Figure 28- MAE equation

**Advantages:**

Robust to Outliers: Unlike MSE, MAE does not square the errors, making it less sensitive to outliers. This means it provides a more robust measure of central tendency for errors.

Interpretability: The units of MAE are the same as those of the steering angle, making it easy to interpret in terms of actual prediction errors.

**Disadvantages:**

Less Sensitive to Larger Errors: MAE treats all errors equally, so it does not penalize large errors as heavily as MSE, which might be a disadvantage if large errors are particularly problematic in your model.

**3. Root Mean Squared Error (RMSE)**

Definition: Root Mean Squared Error (RMSE) is the square root of the Mean Squared Error. It measures the average magnitude of errors, similar to MSE, but in the same unit as the predicted values, which can be more interpretable.

**Mathematical Expression:**

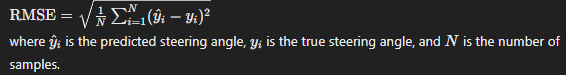


Figure 29- RMSE equation

**Advantages:**

Interpretability: RMSE provides an error measure in the same unit as the steering angle, which makes it easier to understand the magnitude of errors.

Penalizes Larger Errors: Like MSE, RMSE penalizes larger errors more heavily, which can be useful if you want to emphasize reducing large deviations.

**Disadvantages:**

Sensitive to Outliers: RMSE shares the same sensitivity to outliers as MSE due to the squaring of errors, which can skew the metric if there are significant outliers.

Together, these metrics and their associated terms provide a comprehensive evaluation of the model, ensuring it performs effectively across different scenarios and maintains a high level of accuracy and reliability.

## 3.7 Hardware and software Limitations

In the course of developing and testing our autonomous vehicle, we encountered several limitations that impacted both the hardware and software aspects of the project. These constraints presented challenges that affected the overall performance and functionality of the vehicle. Understanding these limitations is crucial for identifying areas of improvement and refining future iterations of the project. The following sections provide a detailed examination of the specific hardware and software limitations we faced, outlining their effects on our project's progress and outcomes.

### 3.7.1 Hardware Limitations

Throughout this project, we encountered several hardware limitations that impacted our progress and the overall performance of our autonomous vehicle. The wheels of our robot car were not designed to rotate, leading to slow and imprecise movements, especially in cases of hard road turns, significantly affecting the car's maneuverability and responsiveness. Additionally, the camera employed for visual data acquisition had a limited lens view and suboptimal image quality, reducing the accuracy of the data collected, thus affecting the performance of the model in predicting the steering angle under dimmed light. Moreover, the weight of the car, exacerbated by the mounted components, especially the power bank supply, added to the overall burden on the system, further hindering the car’s movement performance. Additionally, it is important to note the significant raspberry pi and motors battery consumption observed during the training and testing phases of the project. The motors, which were integral to the vehicle's movement, exhibited a higher power draw than initially anticipated, leading to rapid depletion of the battery supply.

### 3.7.2 Software Limitations

On the software side, we faced challenges related to the processing capabilities of the Raspberry Pi, which served as the central processing unit for our system. The Raspberry Pi's processing speed proved to be insufficient for real-time decision-making and response times, affecting the overall efficiency of the autonomous vehicle. This limitation hindered our ability to implement more complex algorithms and achieve the desired level of performance. Specifically, the low processing speed forced us to cancel the idea of plate tracking, where the car was supposed to follow a specific plate wherever it moves, even though this functionality had already been implemented and was ready for use. Despite these software constraints, the project provided us with valuable insights and allowed us to make meaningful strides in advancing our understanding of autonomous vehicle technology.

## 3.8 Summary

In summary, the methodology chapter outlines a systematic approach to designing and implementing a self-driving robot car, integrating hardware components, developing software modules, collecting and preprocessing data, training deep learning models, and conducting rigorous testing. Each section provides detailed explanations, supported by visual aids where appropriate, to elucidate the processes and methodologies employed in achieving the project's objectives. By following this structured approach, we ensure a comprehensive understanding of our methodology and its implications for autonomous driving technology.

# Chapter 4: Results and Discussions

This chapter presents a detailed analysis of the performance and evaluation of the models employed in this study. Specifically, it focuses on the comparative results of NVIDIA’s model and the custom model developed as part of the research.

In the preceding chapters, we outlined the methodology and the theoretical framework guiding the model development. Now, we transition to an empirical examination of these models, scrutinizing their effectiveness in predicting steering angles for autonomous driving applications. This chapter provides a comprehensive overview of the evaluation metrics, performance outcomes, and insights derived from the experiments conducted.

By juxtaposing the results of NVIDIA’s model with those of the custom model, we aim to elucidate their relative strengths, limitations, and suitability for the task at hand. This comparative analysis will facilitate a deeper understanding of each model's performance, guiding future improvements and refinements. Through this discussion, we will also explore the implications of our findings and their potential impact on the advancement of autonomous driving technologies.

## 4.1 Model Architecture

**1.NVIDIA's Model:**

model = Sequential()  
model.add(Conv2D(24, (5, 5), (2, 2), input\_shape=(66, 200, 3), activation='elu'))  
model.add(Conv2D(36, (5, 5), strides=(2, 2), activation='elu'))  
model.add(Conv2D(48, (5, 5), strides=(2, 2), activation='elu'))  
model.add(Conv2D(64, (3, 3), activation='elu'))  
model.add(Conv2D(64, (3, 3), activation='elu'))  
model.add(Flatten())  
model.add(Dense(100, activation='elu'))  
model.add(Dense(50, activation='elu'))  
model.add(Dense(10, activation='elu'))  
model.add(Dense(1))   
model.compile(optimizer=Adam(learning\_rate=0.0001), loss='mean\_squared\_error')  
return model

* **Layers:**
  + **Convolutional Layers:** 24 filters with 5×5 kernels, followed by 36 and 48 filters with 5×5 kernels, and 64 filters with 3×3 kernels.
  + **Activation Functions:** ELU (Exponential Linear Unit) for all convolutional and dense layers.
  + **Dense Layers:** 100, 50, and 10 neurons before the output layer.
  + **Output Layer:** Dense layer with 1 neuron for regression.
* **Key Features:**
  + **Deep Architecture:** Utilizes multiple convolutional layers with varying filter sizes and strides, which can capture more complex features.
  + **ELU Activation:** Generally, improves learning speed and accuracy by preventing dying neurons and offering smooth gradient flow.

**2.Custom Model:**

model = models.Sequential()  
model.add(Conv2D(8, (3, 3), input\_shape=(66, 200, 3), activation='relu'))  
model.add(MaxPooling2D(pool\_size=(2, 2)))  
model.add(Conv2D(8, (3, 3), activation='relu'))  
model.add(MaxPooling2D(pool\_size=(2, 2)))  
model.add(Dropout(0.5))  
model.add(Flatten())  
model.add(Dense(50, activation='relu'))  
model.add(Dense(1))  
adam = Adam(learning\_rate=0.001)  
model.compile(loss='mean\_squared\_error', optimizer=adam)  
return model

* **Layers:**
  + **Convolutional Layers:** 8 filters with 3×3 kernels, followed by another 8 filters with 3×3 kernels.
  + **MaxPooling Layers:** Reduces spatial dimensions, which helps in reducing computation and overfitting.
  + **Dropout Layer:** 0.5 dropout rate to prevent overfitting by randomly dropping neurons during training.
  + **Dense Layers:** 50 neurons in the fully connected layer.
  + **Output Layer:** Dense layer with 1 neuron for regression.
* **Key Features:**
  + **Simpler Architecture:** Fewer filters and layers, which may lead to quicker training but potentially less ability to capture complex features.
  + **ReLU Activation:** Commonly used activation function that can speed up convergence but may suffer from dying neuron issues.
  + **Dropout:** Aims to improve generalization by preventing overfitting.

## 4.2 Training Process

* **NVIDIA's Model:**
  + **Learning Rate:** 0.0001
  + **Optimizer:** Adam, which is known for its adaptive learning rate capabilities.
  + **Loss Function:** Mean Squared Error (MSE), which is standard for regression tasks.
* **Custom Model:**
  + **Learning Rate:** 0.001, slightly higher, which may result in faster convergence.
  + **Optimizer:** Adam, similar to NVIDIA’s model.
  + **Loss Function:** Mean Squared Error (MSE).

## 4.3 Performance metrics

In this section, we evaluate the performance of the models using key metrics that provide insight into their accuracy and reliability. Performance metrics are crucial for understanding how well a model predicts steering angles, which is essential for ensuring the effectiveness of autonomous driving systems.

We will focus on three primary metrics: Mean Squared Error (MSE), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE). These metrics offer different perspectives on the model's prediction errors:

**Mean Squared Error (MSE):** Measures the average of the squared differences between predicted and actual values. It highlights larger errors more significantly due to the squaring, providing an indication of overall prediction accuracy.

**Mean Absolute Error (MAE):** Calculates the average of the absolute differences between predicted and actual values. It provides a straightforward measure of prediction accuracy in the same units as the steering angle, making it easy to interpret.

**Root Mean Squared Error (RMSE):** The square root of MSE, RMSE represents the average magnitude of the prediction errors in the same units as the steering angle, which helps in understanding the scale of errors.

By analyzing these metrics, we can assess the strengths and limitations of each model, enabling us to make informed decisions about their suitability for real-world applications in autonomous driving. This evaluation also facilitates the comparison of different models, guiding future enhancements and ensuring that the chosen model meets the necessary performance criteria.

### 4.3.1 NVIDIA's Model

* **Mean Squared Error (MSE): 0.054084538889009486**
* **Mean Absolute Error (MAE): 0.19842662997916338**
* **Root Mean Squared Error (RMSE): 0.2325608283632682**

### 4.3.2 Custom Model

* **Mean Squared Error (MSE): 0.05239475629114165**
* **Mean Absolute Error (MAE): 0.19571502373648295**
* **Root Mean Squared Error (RMSE): 0.22889900893438062**

### 4.3.3 Interpretation

**1.NVIDIA:** The MSE of 0.0541 indicates that the NVIDIA model has a relatively low average squared difference between predicted and actual steering angles. The MAE of 0.1984 shows that, on average, the model's predictions deviate by approximately 0.1984 units from the true values. The RMSE of 0.2326, which gives a sense of the model's prediction error, further supports the idea that the NVIDIA model performs well, though there are occasional higher deviations in predictions.

**2.Custom:** With an MSE of 0.0524, the custom model demonstrates slightly better accuracy compared to the NVIDIA model in terms of average squared error. The MAE of 0.1957 indicates that the custom model's average deviation from the actual values is marginally lower than that of the NVIDIA model. Additionally, the RMSE of 0.2289 shows that the custom model has a slightly smaller average deviation from actual values compared to the NVIDIA model.

### 4.3.4 Summary

Both models exhibit strong performance in predicting steering angles, with metrics that are close to each other. The custom model shows a marginal improvement over the NVIDIA model in all key metrics, indicating a slight edge in accuracy. The lower MSE and RMSE values for the custom model suggest it is slightly more reliable in minimizing prediction errors. However, the differences between the models are relatively small, implying that both are quite effective for the task at hand. Depending on specific project requirements or constraints, either model could be suitable, but the custom model’s marginally better performance might make it a preferable choice for applications where precision is crucial.

|  |  |  |
| --- | --- | --- |
| metrics / models | **NVIDIA** | **CUSTOM** |
| **MSE** | 0541 | 0.0524 |
| **MAE** | 0.1984 | 0.1957 |
| **RMSE** | 0.2326 | 0.2289 |

Table 1- models metrics comparison

## 4.4 Discussion

The comparative performance of NVIDIA’s model and the custom model reveals that both are highly effective in predicting steering angles for autonomous driving applications. The custom model slightly outperforms the NVIDIA model in all metrics, including Mean Squared Error (MSE), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE). These improvements, though modest, are indicative of the custom model’s ability to more accurately predict steering angles with less deviation from actual values. This suggests that the custom model may have benefited from specific optimizations or architectural adjustments tailored to the dataset and task, which might not be present in the pre-built NVIDIA model.

While the differences in performance metrics are not drastic, they highlight the importance of model tuning and customization. In practice, even small enhancements in prediction accuracy can significantly impact the overall performance of an autonomous driving system, particularly in real-world scenarios where precision is critical. The marginally better results of the custom model could be attributed to more effective feature extraction, data preprocessing, or hyperparameter tuning.

However, it's important to consider other factors such as computational efficiency, model complexity, and ease of integration when choosing between models. The NVIDIA model, being a pre-built solution, might offer advantages in terms of deployment and consistency, while the custom model’s superior performance might justify its development and maintenance costs. Ultimately, the choice between models should align with project goals, resource availability, and specific application requirements.

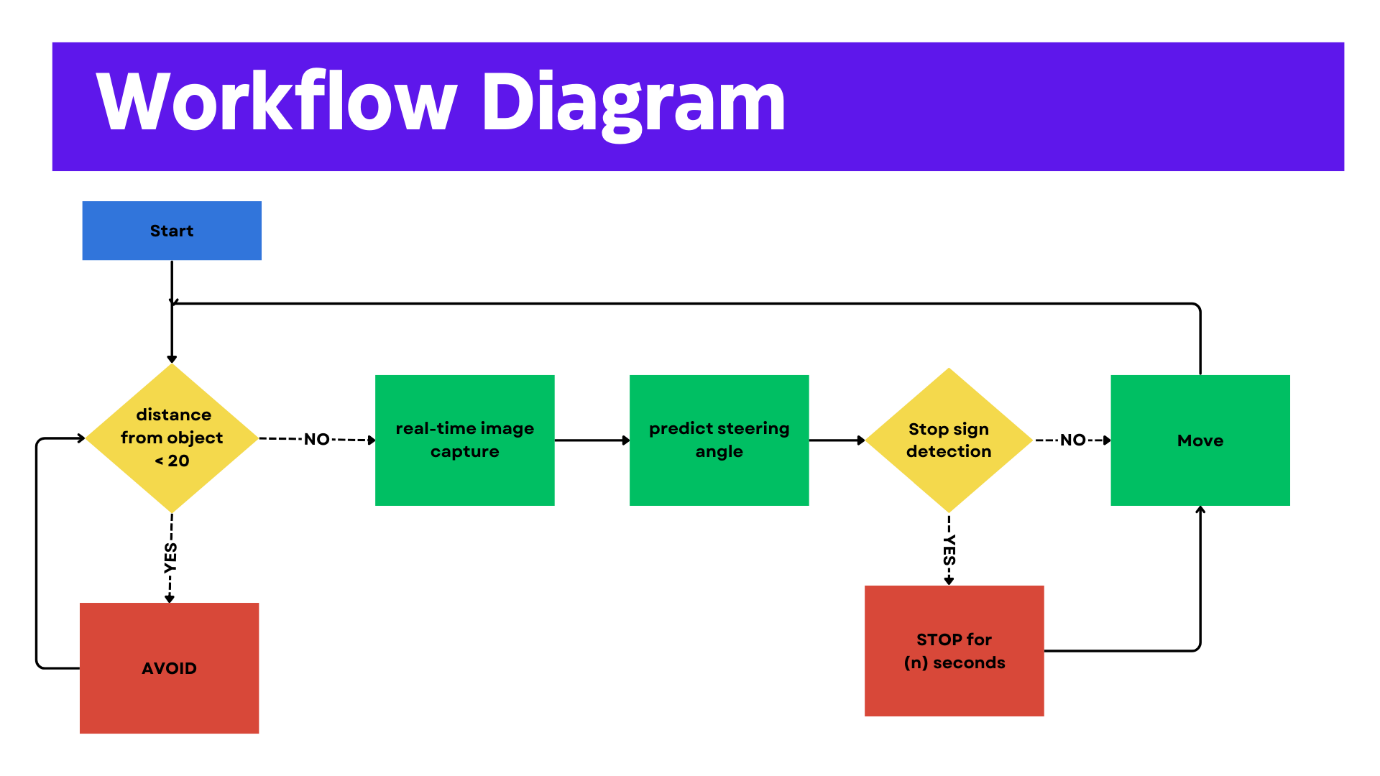


Figure 30- Workflow diagram

The workflow diagram illustrates the operational sequence of an autonomous robot car. The process begins with the car's system initializing. The first decision point evaluates if the distance from an object is less than 20 cm. If the distance is below this threshold, the car initiates an avoidance maneuver. If the distance is adequate, the system proceeds to capture real-time images. These images are then processed to predict the steering angle, which ensures the car navigates the path correctly. Subsequently, the system checks for the presence of stop signs. If a stop sign is detected, the car stops for a specified number of seconds before resuming movement. If no stop sign is detected, the car continues to move forward, iterating through this sequence continuously to navigate autonomously.

# Chapter 5: Conclusion

This project aimed to explore the development of an autonomous vehicle capable of navigating and making decisions without human intervention. Despite the inherent challenges, significant progress was made, resulting in valuable insights into both the potential and limitations of autonomous vehicle technology.

## 5.1 Summary of Achievements

Throughout the project, we successfully integrated various hardware components, including ultrasonic sensors and cameras, to enable the vehicle to perceive its environment. Advanced data processing techniques were employed to interpret sensor data, allowing the vehicle to follow designated paths and recognize stop signs. The implementation of a Convolutional Neural Network (CNN) for steering angle prediction further enhanced the vehicle's ability to navigate accurately.

## 5.2 Limitations

However, the project faced several limitations that impacted both hardware and software aspects. The non-rotating wheels of the robot car resulted in slow and imprecise movements, hampering maneuverability. The camera, with its limited lens view and suboptimal image quality, constrained the accuracy of visual data acquisition. The vehicle's weight, exacerbated by the power bank supply and other mounted components, added to the overall burden on the system, affecting performance. On the software side, the processing capabilities of the Raspberry Pi posed considerable constraints, with insufficient processing speed hindering real-time decision-making and response times. This limitation forced us to abandon the plate tracking system, despite it being fully implemented and ready for use. Additionally, the high battery consumption by the motors during training and testing phases led to frequent battery replacements and recharges, complicating the project's operational efficiency.

## 5.3 Assumptions and Impact

Several assumptions underpinned our project's development, including the expectation of minimal environmental interference and adequate accuracy from the camera and sensors. While these assumptions were necessary for guiding the project, they also highlighted areas for potential improvement. The project's achievements, despite the limitations, underscore the importance of addressing these assumptions in future iterations.

## 5.4 Future Work

Looking ahead, several areas warrant further exploration and refinement. Addressing hardware limitations, such as enhancing the mobility of the wheels and improving camera quality, will be crucial for achieving more precise navigation. Upgrading the processing unit to a more powerful alternative than the Raspberry Pi could enable the implementation of more complex algorithms and improve real-time decision-making capabilities. Optimizing power consumption and exploring energy-efficient components will also be essential for enhancing the vehicle's operational efficiency.

## 5.5 Final Thoughts

In conclusion, this project has provided a comprehensive understanding of the challenges and potential solutions in the development of autonomous vehicles. Despite the hardware and software limitations encountered, the progress made offers a solid foundation for future advancements. The insights gained from this project will undoubtedly contribute to the ongoing efforts to develop fully autonomous vehicles, paving the way for safer, more efficient, and technologically advanced transportation systems.