Bauhaus-Universität Weimar
Faculty of Media
Degree Programme Computer Science and Media

Bauhaus-Universität Weimar

Autonomous Car Navigation Using Microcontroller Camera

MLOM-II Summer Semester 2023

Project Report

Darshan Dodamani (124753)
Purandhara Sai Santosh Lingala (124755)
Saikiran Pasupuleti (124775)

1. Referee: Benjamin Burse
2. Referee: Prof. Dr. Jan Oliver Ringert

Submission date: September 17, 2023

Declaration

Unless otherwise indicated in the text or references, this thesis is entirely the product of my own scholarly work.

Weimar, September 17, 2023

Darshan Dodamani (124753)Purandhara Sai Santosh Lingala (124755)Saikiran Pasupuleti (124775)

Abstract

In the era of autonomous vehicles and machine learning on micro-controllers, our focus is on implementing machine learning on the ESP32 CAM micro-controller. We aim to seamlessly integrate multiple machine learning algorithms, decision-making processes, and hardware optimization to create a self-driving car capable of navigating in a laboratory environment.

Our project involves an extensive evaluation of various machine learning models, including Principal Component Analysis (PCA), Convolutional Neural Networks (CNN), and MobileNetV2. We assess critical parameters such as accuracy, memory utilization, code optimization for the micro-controller, and parameter count.

Furthermore, we embark on a unique comparison between micro-controller-based systems and traditional computers, shedding light on the practicality of micro-controller technology for navigation tasks. Our project strives to advance the field of machine learning on micro-controllers.

Index Terms: Autonomous Vehicles, ESP32 CAM, Machine Learning, Principal Component Analysis, Convolutional Neural Networks, MobileNetV2, code optimization, memory utilization.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background and Motivation

In the rapidly evolving landscape of autonomous vehicles, the convergence of cutting-edge technologies has paved the way for transformative strides in navigation and perception. The goal of this project is to design a vehicle system that takes advantage of the capabilities of a micro-controller camera, notably the ESP32 CAM, to accomplish real-time visual perception and intelligent navigation. By synergizing the power of machine learning algorithms, sophisticated decision-making processes, and seamless hardware integration, our aim is to manifest a self-driving car that navigates fluidly within real-world scenarios, even within the controlled environment of a laboratory.

Adding a microcontroller camera as the main sensory input brings a level of complexity to the project. The system we propose focuses on creating a machine-learning algorithm that can analyze data and make navigation decisions, ultimately ensuring safe and precise path planning.

One crucial part of this project revolves around coordinating the hardware components. By using a microcontroller to control a DC motor we enable the car to execute navigation decisions, such, as changing speed adjusting direction, or following the selected path. This integration is vital as it ensures that the autonomous vehicle can actively adapt to its surroundings. The more challenging part was to make the machine learning optimize and fit into the space-constrained micro-controller and make it work as precise as possible in comparison with the working of the same algorithm in the computer [16, 13].

Finally, to validate our research hypothesis, we rigorously assess the performance of the machine learning models employed in the project, including PCA, CNN, and MobileNetV2. Our evaluation encompasses not only metrics such as accuracy, memory usage during code execution, and model compatibility for generating optimized code for the micro-controller but also considers the number of parameters. This comprehensive evaluation process allows us to identify the model that best aligns with our hypothesis - the creation of an autonomous navigation system utilizing the ESP32 CAM microcontroller effectively.

Moreover, a distinctive approach to this project involves a comparative analysis of the computational capabilities and performance between the micro-controller-based system and conventional computer setups. By undertaking this comparison, we aim to shed light on the practicality and feasibility of micro-controller technology for real-world navigation scenarios, thereby offering crucial insights into the potential of our hypothesis-driven approach.

1. **Hypothesis 1:** We hypothesize that the integration of machine learning algorithms, such as Convolutional Neural Networks (CNN) and Principal Component Analysis (PCA), with the ESP32 CAM micro-controller will enable real-time object recognition and obstacle avoidance, contributing to enhanced autonomous navigation capabilities.
2. **Hypothesis 2:** We posit that the optimization of machine learning models for the ESP32 CAM micro-controller, achieved through techniques like quantization and pruning, will result in a significant reduction in memory usage without compromising navigation accuracy.
3. **Hypothesis 3:** It is our hypothesis that the utilization of the ESP32 CAM microcontrollers low power consumption capabilities will lead to an energy-efficient autonomous vehicle system, making it suitable for prolonged operation in resource-constrained environments.
4. **Hypothesis 4:** We anticipate that the comparative analysis between microcontroller-based systems and traditional computers will reveal the ESP32 CAM microcontroller's suitability for cost-effective and efficient autonomous navigation, potentially revolutionizing the landscape of autonomous vehicle technology.

1.2 Project Objectives

The objectives of this project are:

Autonomous Car System with Micro-controller Camera (ESP32 CAM): Develop a sophisticated autonomous car (Donkey Car) system that effectively employs a microcontroller camera, specifically the ESP32 CAM, for seamless visual perception and navigation.

Machine Learning Algorithm for Navigation: Design and implement an advanced machine learning algorithm that capitalizes on the processed visual data. This algorithm will drive navigation decisions to ensure optimal path planning for the autonomous vehicle.

Integration of DC Motor and Micro-controller: Effectively integrate the DC motor with the ESP32 CAM micro-controller to facilitate precise control over the car's motions. This integration will enable the system to enact navigation decisions encompassing speed modulation, directional shifts, and steering adjustments.

Optimization of Computational Efficiency and Memory Usage: The overarching goal is to attain peak performance while abiding by the constraints intrinsic to microcontroller technology and establish a comparison point with conventional computer processing power.

1.3 Project Scope

Machine learning algorithm innovation: Designing the machine learning algorithm to process the real-time visual inputs that is images and make navigation choices and perform decision-making processes and hardware control.

ESP 32 micro-controller integration: The fusion of ESP 32 CAM microcontroller with the motor controller for the motor speed. Cumulating in a comprehensive system that captures visual data executes machine learning competitions and influences the vehicle's movement.

Comparison of machine learning approaches: Investigate different machine learning algorithms like Conventional Neural Networks, MobileNetV2, and Principal Component Analysis. Find out the suitable for the micro-controller and implement it.

Optimization of computational efficiency: Optimize the machine learning algorithm compatible with the ESP32 CAM micro-controller using the Edge Impulse.

Comparison with conventional computers: A unique aspect of this project involves comparing the computational capabilities and performance of the microcontroller-based systems with the conventional computer setups this comparison provides insights into the viability of microcontroller technology for real-world navigation scenarios.

2 Literature Review

The study conducted by [8] delves into a comprehensive overview of various machine learning algorithms, offering insights into the selection criteria relevant to their experiment. In the initial phases, the authors meticulously curated a dataset, partitioning it into distinct classes. Their dataset comprised images of leaves, which exhibited inherent noise. The authors meticulously executed a series of image preprocessing steps, including segmentation and classification. Notably, the paper elucidated a pivotal image processing task: the conversion from RGB to HSV color space. Employing K-means clustering with two cluster centers, the authors successfully performed image segmentation. The research then embarked on a rigorous comparison of classification accuracy across several machine learning models, encompassing Logistics Regression, K-Nearest Neighbors (KNN), Support Vector Machine (SVM), and Convolutional Neural Network (CNN). Ultimately, CNN emerged as the frontrunner, offering the highest accuracy, and leading the authors to delve deeper into the intricacies of CNN, unveiling the multi-layered architecture that underpins its superior performance.

In this research paper, as documented by Google scholars [3, 18], an innovative mobile architecture is presented, representing a substantial advancement in the performance of mobile models across diverse tasks and benchmarks. The paper primarily revolves around the comprehensive exploration of the MobileNetV2 network across varying model sizes, providing valuable insights into its capabilities. Notably, the authors conduct an in-depth analysis of its performance concerning tasks such as ImageNet classification, COCO object detection, and VOC image segmentation. The paper further delves into the intricate layers that constitute the network, shedding light on the critical aspect of hyperparameter trade-offs. An important highlight is the meticulous assessment of memory consumption across MobileNetV1, MobileNetV2, and ShuffleNet, ultimately concluding in favor of a lower memory consumption model as the dominant choice. This research is instrumental not only in understanding the MobileNetV2 architecture but also in its practical implementation [18].

Paper by [9] surveys Machine Learning in Embedded Systems and Mobile Device Optimization. It underscores the surge of ML in resource-constrained devices, tackling challenges of limited computing power and memory. The paper explores tailored optimization techniques for these environments and examines hardware platforms. It outlines challenges and strategies for overcoming limitations in both hardware and algorithms. Specific optimization methods (model pruning, data quantization, reduced precision, tiling) are highlighted for diverse ML algorithms.

The article authored by [2] examines and contrasts supervised and unsupervised learning models, while also delving into classification patterns. Its main emphasis lies in assessing the influence of machine learning on the performance of autonomous vehicles. The research involved training a DNN-powered autonomous car with a combination of real and synthetic images. The accuracy and potential errors of the algorithm were rigorously examined using the specialized tool DeepTest. This evaluation underscored the algorithm's resilience and its adaptability to various scenarios, contributing significantly to our understanding of the application of machine learning in autonomous driving.

Paper by [6] explores using ESP32 with a built-in camera for machine learning. ESP32, especially with camera-equipped boards like ESP32-CAM and M5CAMERA, shows promise. These boards offer sufficient resolution, though down-sampling on the ESP32 may be time-consuming. Setting the camera resolution to its lowest is advised. In summary, ESP32 with a camera can handle basic machine learning tasks and image preprocessing for more powerful processors. Future research may focus on its performance with different neural networks and utilizing both of its cores for calculations.

The work by [4] addresses the issue of image classification in the presence of extensive image datasets. It introduces an innovative method that integrates depth information, a component often overlooked in conventional techniques. This approach entails depth estimation from images, extracting features based on depth, employing PCA for dimensionality reduction, and applying SVM for classification. Experimental findings illustrate the effectiveness of this method in enhancing classification accuracy when compared to the typical RGB-based methods.

The paper by [1] explores the recent surge of Convolutional Neural Networks (CNNs) in the field of machine learning. It underscores their dominance in recognizing patterns from images when compared to earlier artificial intelligence models. Drawing inspiration from biological neural networks, CNNs present a streamlined yet potent approach to deep learning. The paper introduces CNNs, elucidates their fundamental principles, and layer arrangement, and offers guidance on utilizing them effectively for tasks related to image analysis.

3 Task Distribution

The table 3 illustrates the allocation of responsibilities among the team members for the various phases of the project. The project is divided into two phases. The first phase involves the implementation of optimized code using PlatformIO. Further, the second phase involves the edge impulse implementation.

Table 3.1: Project Stages and Contributions

Phases	Project Stages	Darshan Dodamani	Santosh Lingala	Saikiran Pasupuleti
Phase - 1	Project Ideas Presentation	✓	✓	✓
	Initial Literature	✓	✓	✓
	Define Project Objectives	✓		
	Project Setup	✓	✓	✓
	Dataset Collection 1		✓	✓
	Image Preprocessing	✓		✓
	Machine Learning Model	✓	✓	✓
	Initial deployment into ESP32		✓	✓
Phase - 2	Introduction of Edge Impulse	✓		
	Dataset Collection 2	✓	✓	✓
	Motor Logic		✓	
	Deployment into ESP32 CAM	✓	✓	✓
	Results	✓	✓	✓
	Report	✓	✓	✓

4 Methodology

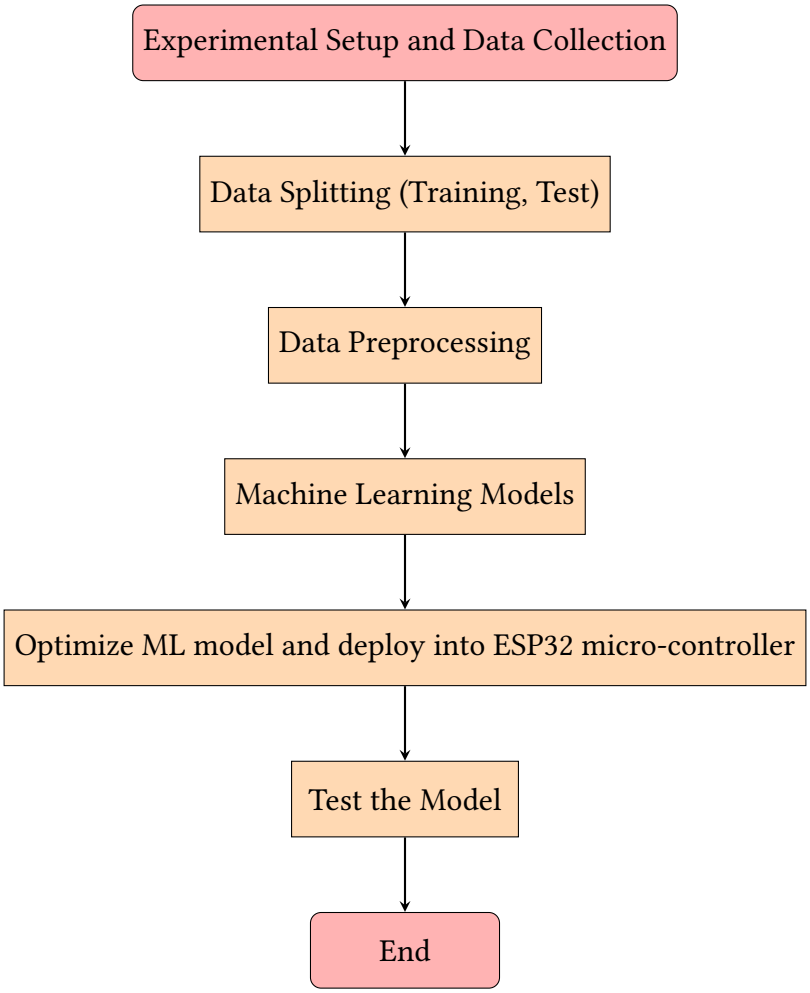


Figure 4.1: Workflow Diagram

4.1 Experimental Setup and Data Collection

In the initial phase of our project, we meticulously set up a controlled laboratory environment for data collection. This included carefully managing lighting conditions and camera angles to ensure the creation of a diverse and representative image dataset. The data collection process itself involved manual driving of the vehicle along a predefined path, exclusively utilizing the ESP32 camera module for image capture. This deliberate setup allowed us to gather data under consistent conditions, setting the stage for robust experimentation.

4.2 Data Splitting (Training, Test)

To ensure the validity and generalization capabilities of our machine learning model, we adopted a rigorous data splitting strategy [15]. This involved partitioning the collected dataset into three subsets: a training set, a validation set, and a test set. Specifically, 60% of the images were allocated for training, while 20% each were reserved for validation and testing. This balanced distribution aimed to minimize bias and enhance the model's ability to generalize to new data [11], [10].

4.3 Data Preprocessing: Image Resizing

Prior to feeding images into the machine learning model, preprocessing steps are undertaken to ensure and enhance model performance. In the data preprocessing phase, we standardized the resolutions of our images, transitioning from the initial 112 x 84 pixels to a uniform 32 x 28 pixels to align with our model's requirements. Given the grayscale nature of our images, additional normalization or complex segmentation was deemed unnecessary. Our primary focus during preprocessing was distinguishing between the foreground (white strips) and background (black areas), as accurate vehicle navigation relied on identifying the white strips [14]. Remarkably, our dataset inherently contained diverse image angles and rotations, rendering traditional data augmentation techniques, such as rotation and brightness adjustments, redundant for our project.

4.4 CNN architecture design

Designing an optimal convolution neural network architecture is the main aim of the project's success. When designing a CNN architecture, it is important to carefully consider factors such as the number and arrangement of convolutional layers, pooling layers, and fully connected layers. The overall architecture should be designed to effectively capture both local and global features present in the input data, while also maintaining a balance between model complexity and computational efficiency. Additionally, it is crucial to conduct comparative evaluations of different architecture variants using aggregated performance metrics such as accuracy, model complexity, and computation efficiency. These metrics provide valuable insights into the effectiveness of design decisions and help determine the best architecture for a given task.

4.5 Optimize ML model and Deployment into ESP32 micro-controller

The optimization and deployment of our machine learning model onto the ESP32 CAM microcontroller were streamlined through the utilization of the Edge Impulse platform. This platform played a pivotal role in converting the trained model into a resource-efficient format compatible with the microcontroller's computational capabilities and memory constraints. The process entailed generating optimized C++ code and seamlessly integrating it with the ESP32 CAM, enabling real-time predictions. Additionally, Edge Impulse simplified the integration with necessary libraries, effectively bridging the gap between the development and deployment phases of our machine learning model. The resulting optimized C++ code, along with the essential libraries, was seamlessly incorporated into the Arduino IDE for further development and deployment.

5 Overview of Autonomous Car Navigation

An autonomous vehicle is one that can function safely and effectively without the need for human control. This vehicle is made up of several interconnected systems that work together to navigate its environment. Humans can choose the destination but is not necessary to operate the vehicle's mechanical systems in autonomous cars, which can perceive their surroundings and drive independently [17]. Autonomous cars rely on a combination of hardware and software to navigate and operate safely. Choosing compatible hardware and software for the project is a challenging task. After a tedious research and study on the different components suitable hardware components are chosen.

5.1 Hardware components

1. **ESP32-CAM Micro-controller:** The board shown in Figure 5.3 is powered by an ESP32-S SoC from ***Espressif***, a powerful, programmable MCU with out-of-the-box WIFI and Bluetooth. It offers an onboard camera module, MicroSD card support, and 4MB PSRAM at the same time. More technical details about the micro-controller are found in

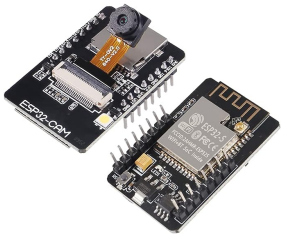


Figure 5.1: ESP32CAM micro-controller

2. **L289 Motor Driver:** This L298N Motor Driver Module shown in Figure 5.3 is a high-power motor driver module for driving DC and Stepper Motors. This module consists of an L298 motor driver IC and a 78M05 5V regulator. L298N Module can control up to 4 DC motors, or 2 DC motors with directional and speed control.

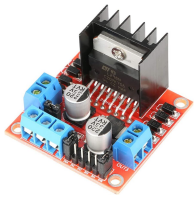


Figure 5.2: L289 Motor Driver

3. **Other hardware components:** Other hardware components refer to DC motors, wheels, chassis, and power source.

5.2 Software Tools

1. **PlatformIO:** A lightweight powerful cross-platform source code editor added ad extension in VS code editor for the development.
2. **VS Code and Arduino IDE:** These provide easy-to-write code and upload it to the board.
3. **Drivers:** To connect ESP32 board to computer and load the program necessary driver named **CH340C USB** has to be installed which is shown in figure 5.3

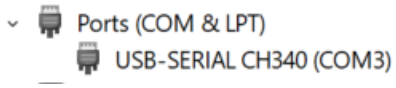


Figure 5.3: CH340 C Driver

4. **Libraries:** TensorFlow, OpenCV, PyTorch, and libraries were used in building and training machine learning models.
5. **Edge Impulse:** A platform used to train the machine learning model and export the optimized code for the embedded device.

5.3 Embedded Systems for Autonomous Vehicles

Embedded systems play a crucial role in the functioning of an autonomous car. They serve as the vehicle's brain, controlling and coordinating various components and functions to enable autonomous driving. The functioning of an embedded system in an autonomous car involves complex software, algorithms, and hardware integration.

Embedded systems in autonomous vehicles are equipped with powerful microcontrollers or microprocessors that process data from sensors like LiDAR, cameras, radar, and ultrasonic sensors. These sensors provide a constant stream of information about the vehicle's surroundings, including other vehicles, pedestrians, road signs, and obstacles. The embedded system's role is to analyze this data, extract meaningful insights, and make instantaneous decisions based on algorithms and machine learning models.

In the context of Donkey Car, many controllers like Raspberry Pi, Arduino, ESP8266, and ESP32 can be used. In this project, we have considered the ESP32-CAM microcontroller because of its versatile features and affordable price. In detail is elaborated in Section 5.3.1

5.3.1 ESP32CAM Micro-controller

ESP32-CAM is a dual-core 32-bit microcontroller module manufactured by Espressif Systems, a successor to the ESP8266. The brilliant feature of ESP32-CAM is that it has integrated Wi-Fi and Bluetooth. It is based on the Tensilica Xtensa LX6 CPU architecture and is highly versatile for a variety of applications [12]. The standout feature of the ESP32-CAM is its OV2640 camera module which can capture images at a maximum resolution of 1600 x 1200 pixels. However, it supports various resolutions such as SXGA (1280 x 1024), QVGA (320 x 240), VGA (640 x 480), SVGA (800 x 600), etc. as shown in 5.4.

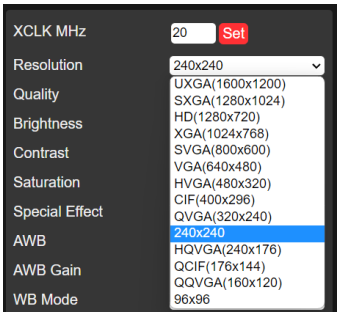


Figure 5.4: ESP32CAM Camera Resolutions

It has several GPIO pins, which allow us to interface with various sensors, displays, and other peripherals. The clock frequency is up to 240 MHz, has multiple power modes, and has 520 KB SRAM. It includes a built-in microSD card slot, which can be used for storing images and videos locally.

Because of its adaptable and potent attributes, the ESP32-CAM is a remarkable selection for machine learning applications. Its dual-core processor facilitates efficient and rapid processing of intricate machine learning algorithms, rendering it highly suitable for real-time tasks. An additional advantage of the ESP32-CAM is its minimal power consumption, making it an excellent fit for devices powered by batteries. Furthermore, the ESP32 can be effortlessly linked to external devices and networks, allowing seamless data transmission and control between them and the cloud.

6 Machine Learning Models

Machine Learning techniques equip self-driving cars with the capability to continuously refine their understanding of the environment, enhance their decision-making skills, and adapt to various driving scenarios [2]. As these algorithms accumulate more data and experience, their performance continually improves, making autonomous vehicles safer and more reliable [6]. There are many supervised, unsupervised, and reinforcement machine learning algorithms used in Autonomous vehicles. One of the major tasks of a machine learning algorithm is a continuous rendering of the surrounding environment and forecasting the changes that are possible to these surroundings.

Supervised machine learning algorithms such as SVM, Decision Trees, YOLO (You Only Look Once), and Convolutional Neural Networks (CNN) are generally used for computer vision tasks such as object detection, traffic sign recognition, and lane detection.

Convolutional Neural Networks (CNN) are mainly used for processing spatial information, such as images, and can be viewed as image feature extractors and universal non-linear function approximators [1], [9]. Before the rise of deep learning, computer vision systems used to be implemented based on handcrafted features, such as HAAR (Viola and Jones), Local Binary Patterns (LBP), or Histograms of Oriented Gradients (HoG). In comparison to these traditional handcrafted features, convolutional neural networks can automatically learn a representation of the feature space encoded in the training.

Unsupervised machine learning algorithms such as K-means clustering and Principal Component Analysis (PCA) are used for sensor data segmentation and dimensionality reduction for sensor data respectively.

Reinforcement Learning Technique such as Deep Q- Networks (DQN) is commonly used for decision-making in navigation and control tasks.

6.1 Principal Component Analysis

PCA is an unsupervised Machine Learning technique that is primarily used to simplify complex datasets while retaining trends and patterns in the data [4]. It achieves this by transforming the original features into a new set of uncorrelated features called principal components.

PCA reduces the number of variables (dimensions) in a dataset while minimizing information loss. It is particularly useful when dealing with high-dimensional data. It provides a way to assess how much variance in the data is explained by each principal component. This helps in deciding how many principal components to retain, as you can often reduce dimensionality while retaining most of the dataset's variability. It is highly beneficial when working with large datasets with many characteristics.

6.2 Convolution Neural Network

Convolutional Neural Networks (CNNs) are specialized for image processing tasks and have excelled in areas like object recognition in applications such as self-driving cars and facial recognition. They automatically learn patterns and features from image data without manual extraction. A CNN comprises an input layer, hidden layers (including Convolutional, Pooling, and Fully Connected Layers), and an output layer [1].

The input layer receives raw data, like images, and passes it on. Hidden layers process and transform data, learning complex patterns. Convolutional layers apply filters to detect features, pooling layers downsample to reduce complexity, and fully connected layers aid in classification. Activation functions add non-linearity, with ReLU, Tanh, and Sigmoid being common choices. The output layer produces classifications or predictions based on features learned.

6.3 MobileNetV2

MobileNetV2 [18] is a lightweight and efficient deep neural network architecture designed for mobile and embedded vision applications. It's a part of the mobile net family of models developed by Google research team [3] it improves upon the original mobile net architecture by introducing several key components to ensure both accuracy and efficiency.

The key features of the MobileNetV2 architecture are as follows:

- **Depth-wise Separable Convolution:** This technique splits a standard convolution into depth-wise and point-wise convolutions, reducing the number of parameters and computational requirements.
- **Bottleneck Design:** It incorporates 1x1 convolutions to reduce channel dimensions before applying depth-wise separable convolutions, conserving computational resources while maintaining feature learning capacity.
- **Inverted Residuals:** MobileNetV2 expands filter and projects channel dimensions, ensuring robust feature representation while maintaining model efficiency.
- **Global Average Pooling:** This component concludes the architecture by enabling fixed-size feature vectors, regardless of input image dimensions.

In detail implementation of these machine learning models in our project is detailed in the section 7.1.4

7 Experimental Setup

In our experimental setup (see Section 4), we followed the methodology described in the previous section. We followed two approaches to reach the objectives of the project (see Section 1.2).

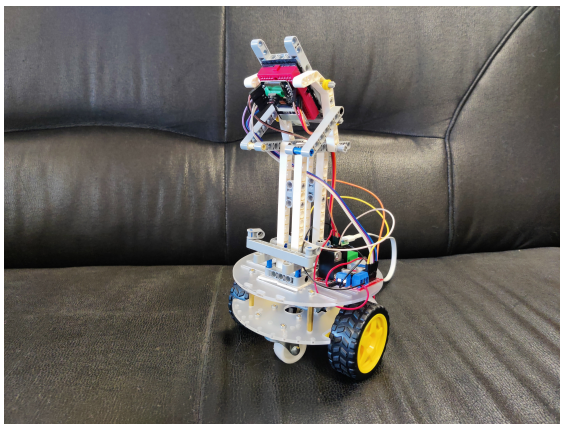


Figure 7.1: Autonomous Car

7.1 Phase 1

7.1.1 Data Collection

Beginning with collecting images for creating the dataset ESP32 camera had many resolutions refer to figure 5.4. To ensure a consistent region of interest (ROI) within the captured images, it was imperative to adopt a resolution of 112 x 84 pixels. This resolution was strategically chosen to optimize computational efficiency while preserving the necessary level of detail for subsequent image processing and analysis [8]. The ESP32 camera module also enabled us with the acquisition of images in grayscale. Grayscale images played a pivotal role in image preprocessing tasks.

We have captured and curated images to 1200, distributed among four different classes (Forward, Right, Left, Stop), shown below 7.2.

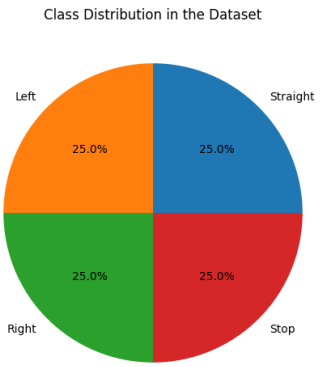


Figure 7.2: Distribution of Dataset

7.1.2 Data Splitting

Following a comprehensive study of the data splitting procedure for training the machine learning model, it was discovered that each class contributed to the data split, as shown in Figure 7.3. Specifically, 80.5% of the images were set aside for training, with the remaining 19.5% set out for testing. Notably, the data splitting approach has had not much effect on our training performance or model accuracy, demonstrating the model's tolerance to modifications in data distribution.

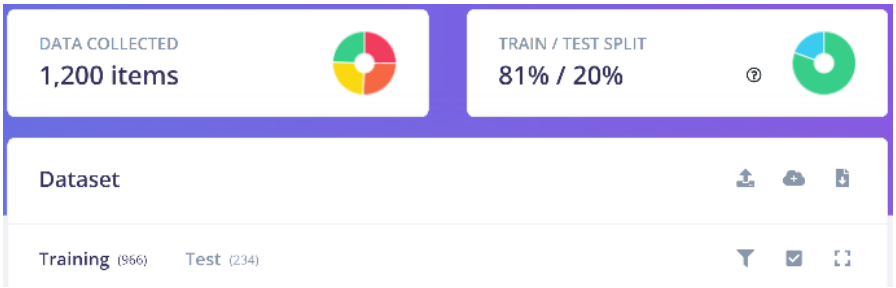


Figure 7.3: Datasplit overview

7.1.3 Image Preprocessing

The neural networks in the ML model will identify edges, colors, and shapes and then perform the classification task. Images are the matrices where individual cells of the matrix contain the pixel data. Each grid holds the pixel information. Many image preprocessing techniques can be implemented to make the training processing easy for the ML model like image resizing, grayscale conversion, and data augmentation.

Gray scale conversion

Grayscale images are the single channel as they represent the intensity information in the range of 0-1. As we don't need to deal with the color balancing in the images because of the only white and black data we didn't undergo grayscale conversion in the preprocessing. We didn't perform this image preprocessing because of reduced image size, simplified preprocessing, faster processing, lighting invariance, and reduced noise.

Image Resizing

Image resizing was implemented as preprocessing as shown in code snippet7.4. This was usually performed because the networks require the input data as explained in 7.1.4. Reducing computational complexity, memory constraints, and improving generalization, and compatibility are the main reasons for the resizing.

```
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # Resize the input image to 32x32
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485], std=[0.229]),
])
```

Figure 7.4: Code Snippet: Image Preprocessing (Resizing, Transformation, and Normalization)

7.1.4 Machine Learning Models PCA, MobileNetV2, and CNN

Principal Component Analysis:

We applied Principal Component Analysis (PCA) to our dataset with the aim of converting high-dimensional data into a more concise representation by generating principal components [4]. We specifically selected 100 components while considering categories such as forward, left, right, and stop. Since PCA serves as a dimensionality reduction technique rather than a classifier, we proceeded to utilize different classifiers.

These classifiers were trained using the dimension-reduced data, encompassing methods such as Support Vector Machines (SVM), Decision Trees, Logistic Regression, and simple neural networks, all geared towards the task of classification. These classifiers effectively learned patterns from the reduced dataset and subsequently made predictions.

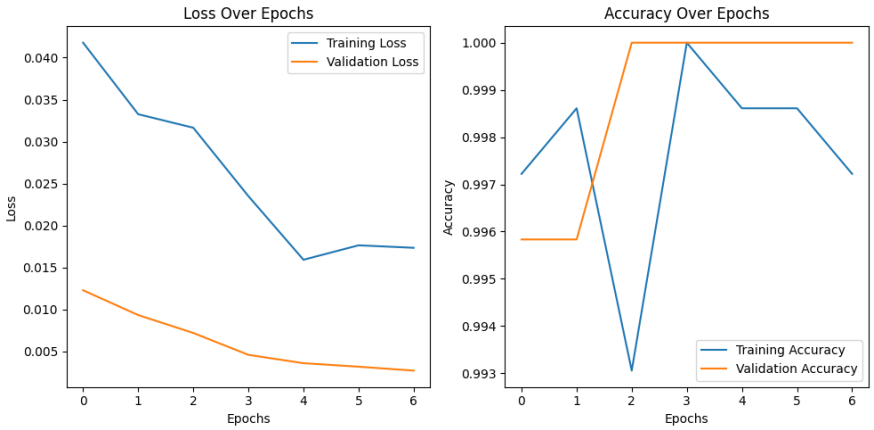


Figure 7.5: PCA: Validation and training loss and Accuracy (Neural Network as Classifier)

Both the validation and testing data were divided equally at 20% for each classifier. For the neural network, we utilized activation functions like **ReLU** and **SoftMax**, setting the number of epochs to 7. However, we encountered an issue with the achieved results: validation and testing accuracy of 100% across all classifiers. This outcome indicates overfitting, which is not desired as it suggests the model has likely memorized the training data and may not generalize well to new data instances. Code can be found in A.3.

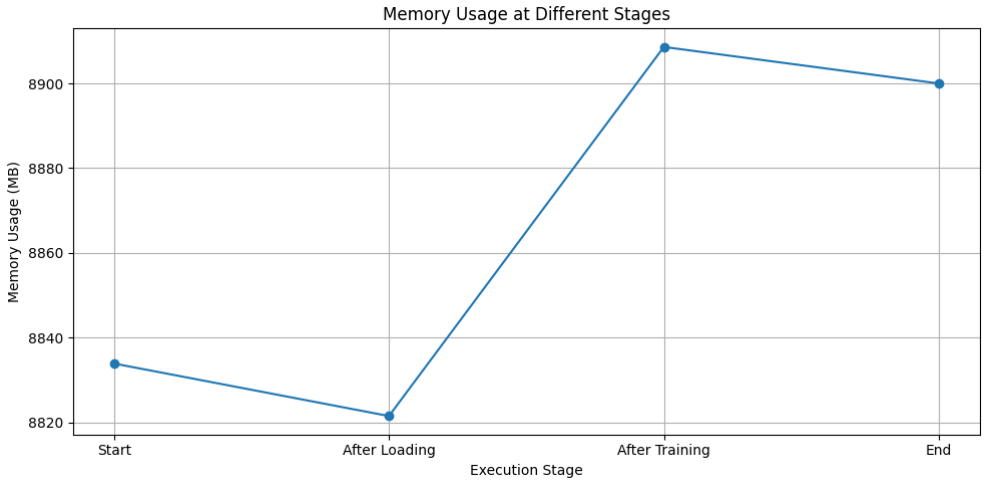


Figure 7.6: PCA Memory Usage at Different Stages

Taking into account elements such as validation accuracy, testing accuracy, and the seamless integration of machine learning algorithms with micro-controllers, PCA hasn't emerged as the optimal selection for this project.

MobileNet V2:

We are using the pre-trained MobileNetV2 [18] architecture defined by [8]. This was most widely used for embedded devices. We load the pre-trained part of the model and make it non-trainable, which means we only train the last layers for our specific task. Then add layers to adapt the model to meet the (objectives 1.2). In Figure 7.7 Global Average Pooling layer, a couple of dense layers, and dropout to prevent overfitting. The model is compiled with an appropriate loss function, optimizer, and accuracy metric Code can be found in A.3.

Figure 7.7: MobileNet V2 model Architecture

The model is trained for 10 epochs. During each epoch entire training dataset (1000 images) is divided into smaller batches. The batch size was 32, so each batch contained 32 images. So overall,

$$\frac{1000}{32} = 31.25 \text{ iterations per epoch}$$

After each epoch, we evaluated the model's performance using the testing dataset. Additionally, we deemed it crucial to monitor the memory usage of the script throughout its execution. This was done to assess and manage the script's memory requirements effectively, ensuring optimal performance and the results can be found in figure 7.8

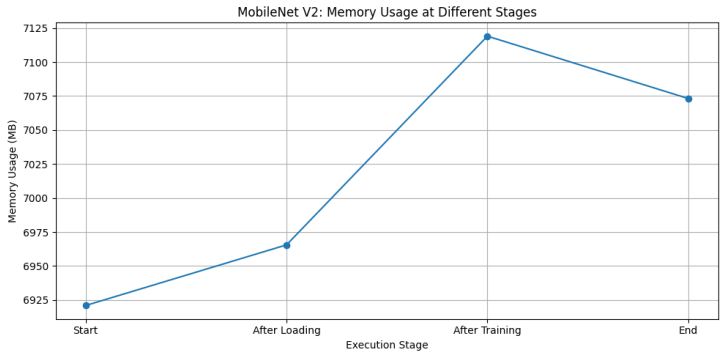


Figure 7.8: MobileNet V2: Memory Usage at Different Stages

The provided training results in Figure 7.9 suggest potential issues with the model's performance. The displayed accuracy and loss metrics for both training and validation sets indicate several problems. Firstly, the accuracy remains approximately constant at around 50%, which is similar to random chance. This suggests that the model is not effectively learning the underlying patterns in the data. Additionally, the validation loss increases over the epochs, which indicates overfitting—the model performs well on the training data but poorly on unseen data.

Towards the end, we converted our trained model into a TensorFlow Lite format, which is suitable for deployment into the ESP32 microcontroller. This step involves optimizations to reduce the model's size and improve its efficiency. Finally, we save the TensorFlow Lite model to a file for later use. The critical factor over here is the file size which was **6864KB** before the quantization by using the default tflite optimizer. After, quantization the model size was **5653 KB**.

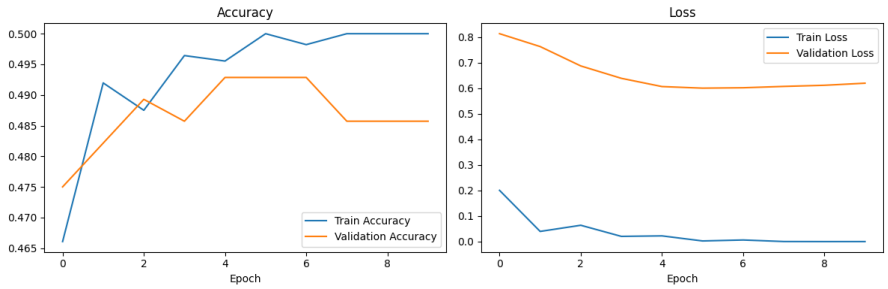


Figure 7.9: MobileNet V2: Validation and training loss and Accuracy

We initially thought the short duration of training epochs (e.g., Epoch 1/10) could be an issue. With such a limited number of epochs, the model may not have had sufficient opportunities to learn complex patterns in the data. Despite applying various optimization techniques, such as quantization and experimenting with different optimization settings, the resulting TensorFlow Lite (TFLite) file size remained larger than required (<350KB). The trade-off between model size and inference performance proved to be challenging to achieve in this project's objectives 1.2. In our quest to minimize the memory usage of the model for ESP32 deployment, we encountered challenges due to MobileNetV2's inherent complexity and the specific demands of our task. Despite our optimization efforts, the desired reductions in file size proved inadequate, potentially implementing by default conversion optimizations that could inflate the model size [9]. Striking the ideal balance between compactness and inference performance can be a doable task. As a result, we made the decision to explore alternative model architectures, setting aside the MobileNetV2 for the project.7.1.4.

Convolution Neural Network:

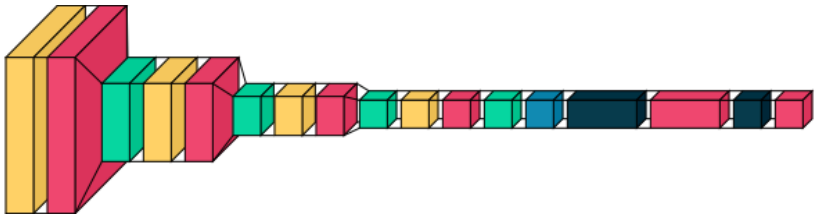


Figure 7.10: CNN Model Architecture

The Convolutional Neural Network (CNN) model deployed in phase 1 uses Keras with TensorFlow as its backend. This model architecture shown in figure 7.10 is based on the LeNet design, a well-known CNN structure. The LeNet model is constructed with multiple layers, including convolutional layers (Conv2D) for feature extraction, ReLU activation functions for non-linearity, and max-pooling layers (MaxPooling2D) to downsample feature maps. Subsequently, fully connected layers (Dense) are employed with ReLU activations. Lastly, there's an output layer employing the softmax activation function, which is particularly suitable for multi-class classification problems.

Images of multiple classes are imported from a specified directory and are resized to a 28x28 resolution (to match LeNet's input requirements) and normalized pixel values are to lie within the [0, 1] range. Class labels are derived from the image file paths and are mapped to numerical labels (0, 1, 2, 3) representing the classes 'forward,' 'right,' 'left,' and 'stop.' The data is further divided into training and testing sets for model evaluation, and class labels are encoded to facilitate multi-class classification Code can be found in A.3.

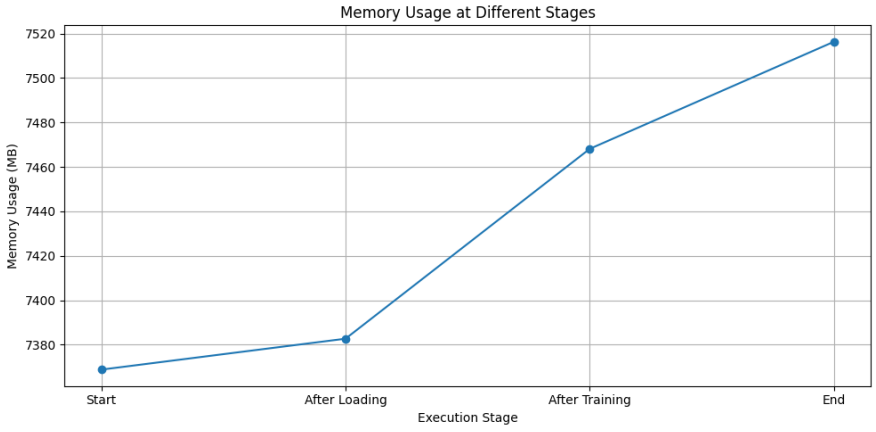


Figure 7.11: CNN: Memory Usage at Different Stages

Device memory usage at various stages of the CNN code execution can be observed in the figure 7.11. In this case, memory usage gradually increased from stage to stage. The training stage showed the most memory usage of any stage. This is expected as training a machine learning model requires a lot of computational power.

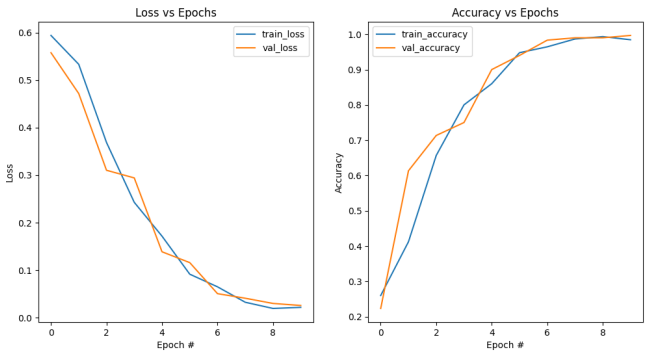


Figure 7.12: CNN: Validation and training loss and Accuracy

Both the graphs of Loss vs. Epochs and Accuracy vs. Epochs in figure 7.12 show that the model training yielded great results. This is because the loss reached a minimum and accuracy reached a maximum after the training. Another thing that can be observed in both graphs is the steady changes in the curve which indicates smooth progress in training.

A few parameters that had to be considered were the final model size, inference time (run-time for a single prediction), and accuracy. The ideal scenario of the built model is a small model size, faster inference time, and higher accuracy. Initial objectives considered for the model are high accuracy (>85%) and small size (<350 KB). To achieve good results, changes to the CNN network are carried out as mentioned in section 8.

7.2 Deployment: Platform IO

After the careful implementation of PCV, MobileNet V2, and CNN. We chose CNN for the deployment. Platform IO provides the in-build libraries to work on the ESP32 Microcontroller.

In this phase 1 (section 7.1) approach, a Platform IO project is created with two library dependencies (ESP32 Camera and TFmicro). ESP32 Camera library is used for controlling the microcontroller camera while the TFmicro library is used to predict the image by running the image data through a machine learning interpreter which uses the TensorFlow Lite model file.

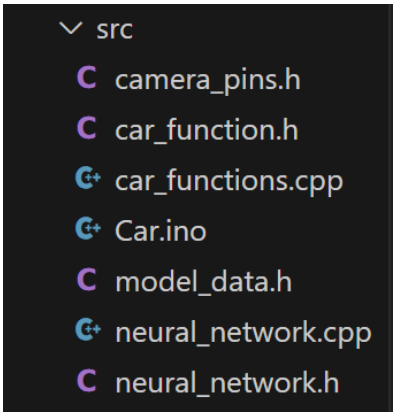


Figure 7.13: PlatformIo Source Code

Once the CPP array data file is generated, it is imported into the Platform IO project source code (Figure 7.13) as a header file which can later be accessed for predictions. Now that the trained model is imported into the project, a few more CPP files and Header files are required to complete the code. **camera_pins.h** file which comes along with Webserver code for ESP32 has all the microcontroller pin configurations defined in it for its camera to run. **car_functions.cpp** file is created to include car motor logic to run both motors. car_function.h header file is generated to call the **car_functions.cpp** after a prediction is made.

Coming to the neural network files there is one CPP file and one Header file **(neural_network.cpp and neural_network.h)**. **neural_network.h** file contains all the required headers and data to run the **neural_network.cpp** file. **neural_network.cpp** file loads in the **model_data.h** file and contains the CNN model architecture required to interpret the image. It uses **MicroMutableOpResolver** from the TFmicro library to define all the CNN layers. This step is necessary to ensure that the image is predicted in a comparable way to Python. Then an interpreter that takes in all the arguments required for the prediction such as the model, resolver, etc., is created. This interpreter is later used to pass the image through the CNN model to get the prediction. **NeuralNetwork::predict** function initializes the created interpreter and takes in the pre-processed image as input. This function provides the prediction by giving the output as a class label.

Now coming to the **Car.ino** file, this is where the image from ESP32 is captured and then resized to 28x28. Once the image is resized, it calls the **NeuralNetwork::predict** function with this resized image. When the prediction is returned from the above function, it calls the appropriate predefined car function for the car to move. This process happens continuously and updates the image and the car movement after each iteration.

7.2.1 Challenges Faced for Phase 1 approach

1. **Serial Monitor Issue:** Faced issues with the serial monitor in Platform IO as the output was not being displayed. After trying a few fixes such as updating the software and extensions, nothing seemed to fix this issue. Finally, one fix helped in resolving this issue by adding the below two lines to the platformio.ini file of the project. monitor_rts = 0 monitor_dtr = 0
2. **Model File Type:** Finding the right machine-learning file type for ESP32 proved to be quite a challenge. An obvious choice is the Tflite model. However, integrating this with C++ is quite difficult due to the limitations of the language in this domain. Finally, the CPP data array file is selected as the main model file as it contains all the information of the model from Tflite and also it can be interpreted by the default interpreter present in the TFmicro library which is imported to the project. But this came with its challenges as the TensorFlow lite model converted to **const unsigned char cpp array** increases its size significantly which is why the TensorFlow lite model size has been brought down significantly lower than the requirement.
3. **Image Processing in C++:** Images captured by the microcontroller need to be resized and processed to run the machine-learning algorithms on them with fewer resources. To perform image processing, python has multiple different libraries such as Open CV, Sci-kit Image, Pillow, etc., But coming to C++ it is not as straightforward as installing a library. It is much more complex and complicated to perform even small operations on the image. After trying multiple times to install Open CV for C++, it remained unsuccessful. So, the approach of manipulating the pixel data directly is implemented.
4. **Machine Learning in C++:** Similar to image processing, performing machine learning in C++ is also a challenge due to the lack or limited availability of existing libraries. TensorFlow lite is one of the few libraries which can be used for our project implementation. However, this came with a lot of drawbacks compared to its Python counterpart and was a challenge to use.

7.3 Discussion and Interpretation of Results

One major issue faced during this approach is that the prediction is the same every time. It is verified that the image being passed through the **NeuralNetwork::predict** function is updating after every iteration. Also, the neural network and preprocessing of the image is the same as its Python counterpart. Despite all the verifications, the outcome of predictions remained the same which made this approach not usable.

7.4 Phase 2

After completing Phase 1, we realized that our deployment through Phase 1 in 7.1 was not reliable. The next challenge on our path was deployment, making sure that the powerful machine learning models we would develop could be put to practical use in the real world. So, we introduced Phase 2, where we focused on deployment using a platform called Edge Impulse.

7.4.1 Edge Impulse

Edge Impulse along with Arduino is used instead of Platform IO. In Edge Impulse after a new project is created, two parameters should be selected (labeling method and target device) in the dashboard. For our requirement "one label per data item" is selected as the labeling method and "Espressif ESP-EYE (ESP32 240MHz)" is selected as the target device.

Then in the data acquisition tab shown in Figure 7.14, a dataset consisting of multiple images of different classes is uploaded to the edge impulse platform. Using the choose files option, images of a class are selected, and in the space provided below in the label section, the class label name is entered. This process is repeated until the data of every class is uploaded. "Automatically split between training and testing" is selected for random allocation of training and testing data.

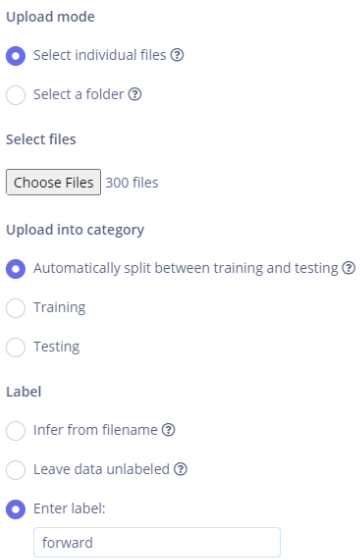


Figure 7.14: Data Upload Parameters

Coming to the impulse design tab shown in Figure 7.15, there are three sub-menus (Create Impulse, Image, Classifier) in total. In create impulse, in the image data block, resize image width and height are defined. To maintain the same ratio and avoid any preprocessing issues, resize width and height are set to “32x24”. The resize mode is set to “squash” which avoids cropping the edges of the image, unlike other available modes. In the second block which is the preprocessing block, “Image” is selected which in this case means the image is pre-processed and normalized. In the last block which is the learning block, “Classification” is selected as the learning method required in our case. This takes an image as input and gives any one of the four classes as output.

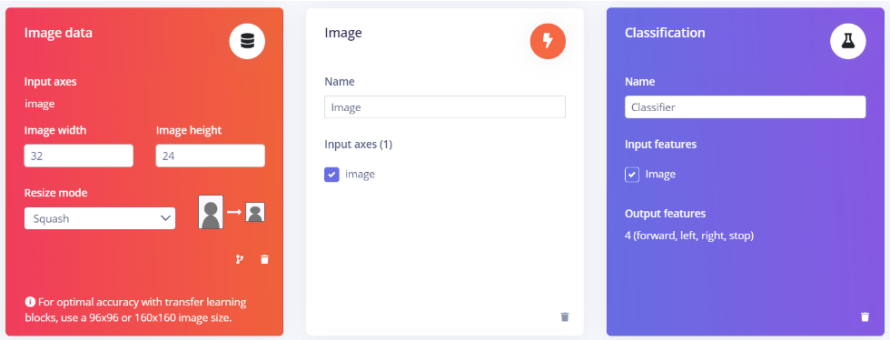


Figure 7.15: Impulse Design

In the next sub-menu Image, the color depth of the image is selected as “Grayscale” as the image still retains most of the information required when it is converted. In the last sub-menu Classifier, CNN parameters developed in the initial approach are used. Previous findings helped a lot to optimize the model in this phase.

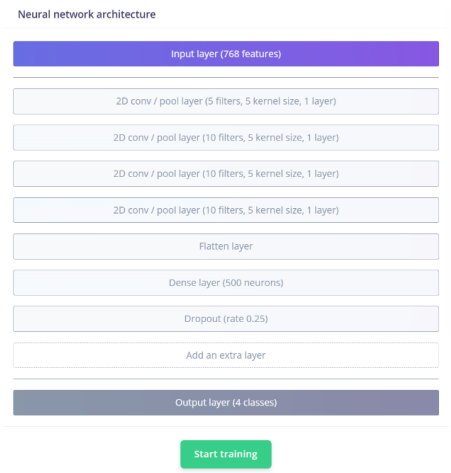


Figure 7.16: CNN Architecture

The model is then trained with a validation set size of 20%. The results of the training are shown in Figure 7.17, 7.18. Training has generated both quantized Figure 7.17 and unoptimized models Figure 7.18 which produced 100% accurate results while keeping the inference time, RAM usage, and flash usage as low as possible. This is possible due to the large size of the dataset (1200 images) and the optimized CNN parameters.

Figure 7.17: Quantized Model

Figure 7.18: Unoptimized Model

When comparing both the quantized and unoptimized models, the inference time, RAM usage, and flash usage are noticeably lower on the quantized model when compared to the unoptimized model. As ESP 32 is a small device with limited power, the quantized model is selected over the unoptimized model. In the data explorer graph, it can be visualized how individual classes are separated/classified.

Coming to the testing, the model is tested in the model testing tab, testing resulted in 100% accuracy.

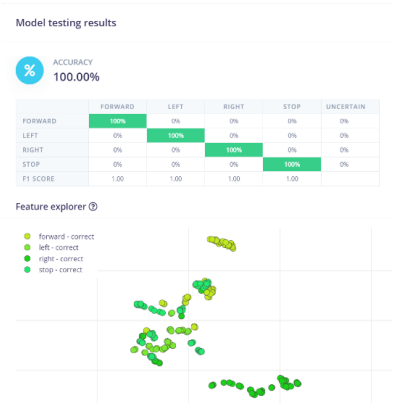


Figure 7.19: Model Test Results

Finally, in the deployment tab, the Arduino library is selected as the deployment method and then a quantized model is built. Building in this section downloads an Arduino library as a zip file containing all the required data such as the Tflite model, model parameters, and header files.

7.4.2 Integration of CNN Model with ESP32CAM Microcontroller

The downloaded Arduino library can be added to Arduino IDE which at the time is on version 2.1, through **Sketch** -> **Include Library** -> **Add .Zip** Library. Once the library is added, the required code file can be opened from File -> Examples -> Custom Library (Newly added library) -> esp32 -> esp32_camera. Before installing the library, it should be made sure that there are no existing copies of any older versions of this library or other similar edge impulse libraries. This ensures that there are no errors in the installation of the custom library. Required modifications are done to the esp32_camera.ino file to include motor pin configuration, PWM configuration, functions for car movement, and an addition in the void loop which reads the prediction and calls the function for car movement. Once these modifications are made, the code is uploaded to the ESP32 microcontroller.

Pin connections are defined as mentioned in the motor logic. Similarly, the car movement functions are defined. In the setup function, all the pins are initialized as output and the PWM pins are set up as referred from A.1. Initially stop function is called before the ESP32 starts capturing images. In the loop function, the resulting prediction is stored in the variable called "label". This stored prediction value is then used to call the respective car movement function. Each movement of the motors is defined to make the car stay on the correct path. Each label value corresponds to a particular class, 0 for forward, 1 for left, 2 for right, and 3 for stop.

Once these modifications are made, the code is uploaded to the ESP32 microcontroller. After the code is uploaded, the ESP32 starts taking images in a loop, predicts the image taken, and gives commands to the motors. Additionally, until the ESP32 stays connected to the laptop after the code is uploaded, it prints out some information on the predictions in real-time to the Serial Monitor in Arduino IDE. Some information that it prints out is the predicted class and inference time.

7.4.3 Challenges Faced

1. **Accuracy Choice:** Selecting accuracy proved to be a challenge as the theory and practical application of machine learning differ. In machine learning, 100% accuracy is considered overfitting [5], but in our project running a similar model with a slightly lower accuracy of 96.4%, resulted in the car performing very poorly. This is because when the car makes one wrong prediction it tends to completely go out of position/track. To avoid this problem, the model with 100% accuracy is considered for the final implementation.
2. **Movement Logic:** Direct implementation of commands such as forward, right, left, and stop resulted in an unrestrained movement of the car. Some tweaks to this logic needed to be made to make the car perform well in the path. An explanation of the changes made is as follows, if the label value for an image is predicted as 0, the car moves forward. If the label value is predicted as 3, the car stops for 1 second and then it moves forward again. This logic for stop helps in avoiding a deadlock situation whenever there is a false stop prediction made which gets stuck in a loop. Making the car move forward after a stop gives a new image for the microcontroller to predict. If the label value is predicted as 1, the car turns right for 200 milliseconds and then stops. Similarly, if the label value is predicted as 2, the car turns left for 200 milliseconds and then stops. Stopping the car after a slight turn ensures that the car does not turn too much.

3. **Motor Speed:** The motors when running at full speed are too fast to make predictions in real time. When the motors were tested at full speed, by the time the next prediction is made the car is covering a lot of distance. This meant that, often the car was going out of the path. To prevent this from happening, all motor functions except the stop function have PWM. This helps in controlling the speed of the vehicle when in motion.
4. **Upload Delay:** It usually takes about 2 to 5 minutes to upload the code for the first time and reduces slightly in the next uploads if there are no changes to the code. This delay was manageable but required careful consideration when iterating on the code.

7.4.4 Real-time Image Processing and Prediction

In this project, real-time image processing and prediction play a crucial role in the successful operation of the car. Here are some key aspects related to this,

1. **Inference Time:** The inference time, which is the time taken by the ESP32 microcontroller to process an image and make a prediction, is low at 60-75 milliseconds. This means the microcontroller can make approximately 13 to 16 predictions per second.
2. **Quick Adjustments:** The low inference time allows the microcontroller to make rapid, real-time adjustments to the car's movement. It ensures that the car can respond almost instantly to changes in its course.
3. **Path Following:** When the car follows the path, it makes micro-adjustments swiftly and accurately to stay on the correct course. These adjustments are vital for maintaining the car's trajectory and preventing it from deviating off the path.

7.4.5 Discussion and Interpretation of Results

The results of the car's performance are proof of the effectiveness of the implemented method. Here's an overview of the results achieved:

1. **Navigation:** During testing, the car consistently completed the path without errors. It exhibited a zig-zag pattern on certain straight paths due to the nature of the dataset and the implemented logic. Despite this unusual movement, it didn't affect the car's accuracy in a straight, but due to the nature of a zig-zag pattern, the time it took to travel the straight has increased. Coming to errors noticed while testing such as the car turning too much or going off the track did occur, especially at sharp corners. Notably, the car performed more accurately on curved turns, as the curve was more visible when the car was closer to it.
2. **Stop Sign Recognition:** At the end of the path, where a stop sign was placed, the car reliably came to a stop. This demonstrates the effectiveness of the model in recognizing and responding to stop signs, contributing to the overall effectiveness of the system.

In summary, the autonomous car successfully navigated the predefined path, made real-time adjustments, and responded to stop signs accurately. While some minor challenges were encountered and the car exhibited a zig-zag pattern on straight paths, the project achieved its primary objective of demonstrating autonomous navigation using machine learning and real-time image processing on a small, resource-constrained microcontroller.

8 Hyperparameter Tuning and Model Configuration

Hyperparameters are the configuration settings that are determined prior to training a machine learning model, and they play a pivotal role in shaping the model's architecture and influencing its training process. These settings encompass a wide range of choices, from the number of layers and neurons in a neural network to the learning rate, batch size, and more. In this section, we delve into the intricate details of the hyperparameters meticulously chosen for our project. We explore the rationale behind each selection and provide insights into how they impact the model's performance, ultimately guiding us toward an optimal and effective machine-learning solution.

8.1 Initial CNN parameters

- 2 Convolution Layers
- Filter = (20,50)
- 2 Dense Layers
- Dense Layer Units = 500
- Medium Sized Dataset
- Image Resized Size = 28x28
- Epochs = 15
- Learning Rate = 1e-3
- Batch Size = 32

By considering the parameter in the 8.1, we worked on changing the different layers to get the optimal accuracy of the model and also the TFLITE file size.

8.1.1 Effect of Convolution Layers (First Iteration)

The table 8.1 demonstrates the trade-off between model accuracy and size when varying the number of convolution layers. While increasing the number of layers improves accuracy, it also increases the model's size. The decision on the number of convolution layers depends on the specific deployment requirements. For applications with limited memory and storage, a model with fewer layers (e.g., 2 or 3) may be preferred, as it offers a reasonable compromise between accuracy and size. On the other hand, if higher accuracy is paramount, a deeper network (e.g., 3 convolution layers) can be chosen, provided that the available resources can accommodate the larger model size.

Table 8.1: Impact of Convolution Layers on Model Performance and Size

Convolution Layers	Accuracy	Keras Model Size (MB)
1	72%	22.6
2	81%	14.5
3	83%	3.87
4	68%	2.34

8.1.2 Effect of Dense Layers (First Iteration)

This table 8.2 provides valuable insights for choosing the appropriate number of dense layers in your neural network. While adding more layers may enhance model performance to some extent, it's essential to consider the trade-off in terms of increased model size, especially for ESP32.

Table 8.2: Impact of Dense Layers on Model Performance and Size

Dense Layers	Accuracy	Keras Model Size (MB)
2 (1 Fully Connected & 1 Softmax)	83%	3.87
3 (2 Fully Connected & 1 Softmax)	81%	6.77

8.1.3 Effect of Quantization (First Iteration)

Table 8.3 highlights the effectiveness of quantization in significantly reducing the model's memory footprint while maintaining its functionality [7]. It demonstrates the importance of choosing an appropriate model format for deployment. In this iteration Quantized model was not supported.

Table 8.3: Effect of Quantization on Model Size

Model	Size
Keras	3.87 MB
Tflite	1.21 MB
Quantized Tflite	317 KB

8.1.4 Effect of Pruning (First Iteration)

In the "Effect of Pruning" iteration, we delved into the impact of pruning techniques on both model accuracy and size. Initially, the Keras model achieved a commendable 83% accuracy with a relatively modest model size of 3.87 MB. However, when pruning strategies were applied to reduce the model's complexity, a surprising outcome emerged. While the pruned model achieved a higher accuracy of 85%, indicating improved predictive performance, it came at the cost of an expanded model size, now totaling 6.22 MB. This concluded in model size as shown in Table 8.4.

Table 8.4: Effect of Pruning on Model Size and Accuracy

Model	Accuracy	Model Size
Keras	83%	3.87 MB
Pruned	85%	6.22 MB

8.1.5 Effect of Dataset (First Iteration)

In our analysis of dataset size impact (Table 8.5), we observed notable trends. With a small dataset of 80 images, the model achieved 70% accuracy and had a Tflite size of 1.21 MB. Moving to a medium-sized dataset (192 images) improved accuracy to 85%, maintaining the same Tflite model size. The large dataset (500 images) delivered exceptional results with, 100% accuracy without altering the Tflite model size. This emphasizes how data volume influences accuracy without bloating the model size, reaffirming the efficiency of our chosen architecture (see Section 7.1.4).

Table 8.5: Effect of Dataset on Model Accuracy and Tflite Size

Images	Accuracy	Tflite Size
Small - 80 images	70%	1.21 MB
Medium - 192 images	85%	1.21 MB
Large - 500 images	100%	1.21 MB

8.1.6 Effect of Convolution Layers (Second Iteration)

Now, we carried out the same process for the second iteration. In this analysis (Table 8.6), we observed that moving from 3 to 4 convolution layers yielded a similar level of accuracy (100% and 99%, respectively) while reducing the Tflite model size from 1.21 MB to 706 KB.

Table 8.6: Effect of Convolution Layers (Second Time)

Convolution Layers	Accuracy	Tflite Size (KB)
3	100%	1,210
4	99%	706

8.1.7 Effect of Dense Layers (First Iteration)

In this case (Table 8.7), opting for no dense layer units resulted in reduced accuracy but a slightly smaller Tflite model size (99% accuracy and 706 KB vs. 93% accuracy and 619 KB).

Table 8.7: Effect of Dense Layer Units

Dense Layer Units	Accuracy	Tflite Size (KB)
500	99%	706
100	93%	619

8.1.8 Effect of Resized Image size

When considering different image sizes (Table 8.8), retaining the original size led to the highest accuracy (99%), while reducing the size to 30x30 resulted in a slight accuracy drop to 95%.

Table 8.8: Effect of Resized Image Size

Size	Accuracy	Tflite Size (KB)
28x28	99%	706
30x30	95%	706

8.1.9 Effect of Filters (First Iteration)

Analyzing filter choices (Table 8.9), we found that (5,10,10,10) filters provided the same accuracy as (5,15,15,15) filters (93%) while reducing the Tflite model size from 98 KB to 61 KB.

Table 8.9: Effect of Filters (First Time)

Filters	Accuracy	Tflite Size (KB)
(20, 50, 50, 50)	99%	706
(10, 25, 25, 25)	92%	215
(5, 15, 15, 15)	86%	98

8.1.10 Effect of Epochs

Investigating the impact of epochs (Table 8.10), we found that using 25 epochs improved accuracy (93%) while maintaining the Tflite model size at 98 KB.

Table 8.10: Effect of Epochs

Epochs	Accuracy	Tflite Size (KB)
15	86%	98
20	92%	98
25	93%	98

8.1.11 Effect of Filters (Second Iteration)

In this evaluation (Table 8.11), selecting (5,10,10,10) filters maintained accuracy at 93% while further reducing the Tflite model size to 61 KB.

Table 8.11: Effect of Filters (Second Time)

Filters	Accuracy	Tflite Size (KB)
(5, 15, 15, 15)	93%	98
(5, 10, 10, 10)	93%	61

8.1.12 Ineffective Changes

1. Changing the learning rate:
    - Decreased accuracy
    - Same model size
2. Changing the batch size:
    - Increased runtime
    - Same model size

Note: Despite the initial requirement of 85% accuracy, higher accuracy is considered to accommodate the losses caused by converting the Keras model to TensorFlow lite and then to CPP array data.

8.1.13 Outcomes of the optimization

TensorFlow Model Size - 61 KB

CPP Array File Size - 278 KB

After the CNN parameters are properly configured in Python, the model code can be executed which trains the network and generates all three model files (Keras, Tflite, and CPP array data).

9 Circuit Diagram

Power to the L298N motor driver is provided directly from the power bank. Power to the ESP32 microcontroller is provided with the help of the L298N 5V pin referred from A.1. Both motors are connected to the motor pins on either side of the motor driver. A common ground is provided for the microcontroller and motor driver. The pin connections between ESP32 and the L298N motor driver for motor control are as follows, GPIO pins 13,15,14,2 are connected to IN 1, IN 2, IN 3, and IN 4 respectively. For PWM, GPIO pin 12 is connected to EN A and EN B pins. Connecting the same GPIO pin to both the enable pins ensures the same speed of the motors. The circuit diagram of all the components is shown in the figure below.

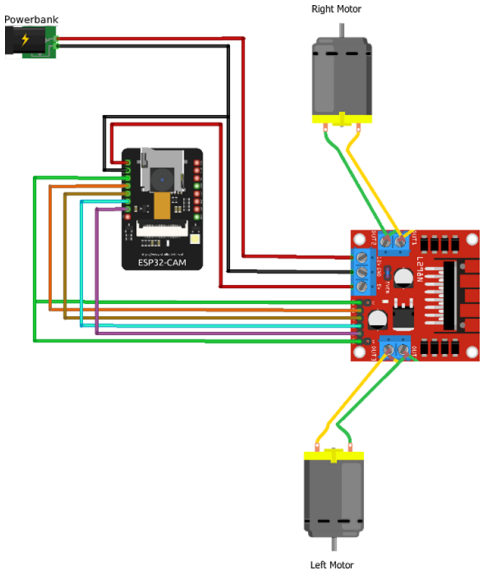


Figure 9.1: Circuit Diagram

10 Motor Logic

To control the car's movement as required, motor logic is implemented referred from A.1. This tells both the motors exactly how to move. Motor logic varies from setup to setup as the circuit connection is never the same between the DC motors, motor driver, and the microcontroller. Motor logic is comprised of two parameters, one is the state of the motor (on or off), second is its direction (clockwise or anti-clockwise) in reference to A.1. And since two motors are being used, a total of four parameters are to be considered. Each parameter has either HIGH or LOW as its value. To figure out the motor logic for our vehicle setup, a set of trails and errors for each of the four movements (forward, right, left, and stop) are executed. Following is the circuit connection followed throughout the trial-and-error process which ensures that the logic remains the same for any similar connection.

ESP32 to L298N Motor Connections:

- GPIO 13 to IN 1
- GPIO 15 to IN 2
- GPIO 14 to IN 3
- GPIO 2 to IN 4
- GPIO 12 to EN A & EN B (PWM-Pins)

10.1 Logic:

After several trials, the following motor logic was determined for the respective movements:

- Forward - HIGH LOW HIGH LOW
- Right - HIGH LOW LOW HIGH
- Left - LOW HIGH HIGH LOW
- Stop - LOW LOW LOW LOW

11 Real World Testing in Controlled Environments

After successfully implementing the Machine Learning model on the ESP32-CAM, we conducted tests with the Donkey Car on our custom-designed track shown in figure 11.1. This track included straight segments, right-angle edges, left-angle edges, and stop zones. The front-facing camera on the vehicle captured 15 frames per second.

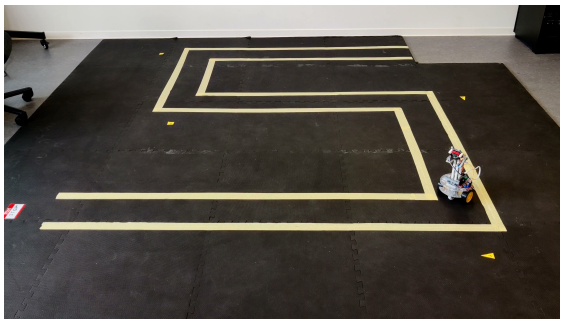


Figure 11.1: Testing Car in the lab

Upon supplying power and allowing the vehicle to operate in this environment, it performed admirably. It navigated the straight segments flawlessly, with occasional deviations when approaching the edges. To address this, we adjusted the motor speed by introducing pulse width modulation (PWM) and set it to 200.

Additionally, we pushed the boundaries of our solution by modifying the track layout. Replacing straight edges with curved sections did not affect the donkey car's performance; it continued to excel in maintaining the course and navigating the updated track with precision.

During 30 testing runs, the vehicle performed well in 20 instances, 8 instances occasionally strayed off the track, possibly due to model overfitting. Detailed statistical test runs are shown in figure 13.1. Subsequently, we modified the track by replacing curves with edges, and the vehicle continued to perform effectively.

12 Challenges faced and Solutions

During the project, we encountered several challenges related to the integration and functionality of the ESP32 camera module. Initially establishing a connection between the ESP 32 microcontroller and computer pauses the difficulties leading to intermittent data transfer issues. Drivers were non-functional and incompatible with the Windows updates. Wi-Fi connection with ESP 32 was a bit challenging. The microcontroller can store the captured images on the SD card but mounting the SD card into ESP 32 and detecting it on the laptop was a bit tedious.

Here are detailed lists of challenges:

12.1 ESP32 camera and storing the Images in SD card

The integration and functionality of the ESP 32 camera module was initially a bit challenging. As ESP 32 can be mounted with an external SD card it poses space constraints.

Solution: We changed the cameras frequently as it was a hardware issue. The external SD card should be up to 8GB only. From (ESP32-CAM Take Photo and Save to MicroSD Card, n.d.) website we implemented the SD card functionality as per the project.

12.2 Image Size

The size of the captured images presented challenges, particularly regarding the computational resources required for processing and training. Images with large dimensions demanded substantial memory and processing power, potentially affecting the efficiency of the deployed solution on the ESP32 microcontroller.

Solution: To address the image size challenge, we employed image preprocessing techniques such as resizing. By reducing the dimensions of captured images to a suitable resolution, we mitigated the computational load during preprocessing, training, and real-time predictions. This optimization facilitated smooth execution on the ESP32 microcontroller while maintaining an acceptable level of accuracy. We finally decided to capture images with 112 X 84 pixels.

12.3 Transition from Platform IO to Arduino IDE

We encountered compatibility and usability challenges while working with Platform IO IDE. Although initially chosen for its extensive features, we faced difficulties in achieving seamless integration with certain libraries and components. This led us to reconsider our choice of development environment.

Solution: In response, we transitioned to the Arduino IDE, which provided a more streamlined and compatible environment for our project needs. While this transition required adapting to a new interface and workflow, it significantly improved the stability of the development process and the integration of essential components. We mostly followed the (ESP32-CAM Object Detection with Edge Impulse, n.d.).

12.4 Machine Learning Model Selection Issue

Selecting the appropriate machine learning model for image classification presented a challenge due to the complexity of the task and the limited resources of the ESP32 microcontroller. Choosing a model that balanced accuracy and computational efficiency was crucial. It was very easy when the model was created and run on the laptop. However, due to micro-controller complexity, we were challenged to work with multiple machine Learning models 7.1.4.

Solution: To address this challenge, we conducted a comparative analysis of different machine learning approaches, including Convolutional Neural Networks (CNNs), PCA, and MobileNetV2, and Principal Component Analysis (PCA). By evaluating their performance in terms of accuracy, speed, and memory usage, we determined that CNNs were the most suitable choice for our project. This decision was guided by their ability to capture complex patterns in images while remaining feasible for deployment on the microcontroller.

12.5 Optimizing Code for ESP32CAM

The process of optimizing the machine learning model's code for deployment on the ESP32CAM micro-controller presented a significant challenge. Ensuring that the model's execution was efficient, responsive, and resource-friendly while accommodating the microcontroller's constraints required careful consideration. We found the difficulty until we gave a thought to Edge Impulse.

Solution: To tackle the optimization challenge, we leveraged platforms like Edge Impulse to generate optimized C++ code for the ESP32CAM microcontroller. This approach facilitated the integration of the model's inference logic into the micro-controller environment, optimizing performance while adhering to resource limitations.

13 Conclusion

13.1 Summary of Achievements

We chose the ESP32 microcontroller for its reliability, despite its longer processing times. To optimize performance, we set the camera resolution to the lowest level since machine learning algorithms typically work with low-resolution images.

Our achievements include successfully implementing advanced machine learning techniques, including Convolutional Neural Networks (CNN), Principal Component Analysis (PCA), and MobileNetV2. We successfully deployed the complex CNN algorithm on the compact ESP32-CAM microcontroller, harnessing the full potential of these advanced techniques.

Upon integration into the Donkey Car platform, our solution demonstrated exceptional performance. It navigated the designed track, including straight sections, right turns, left turns, and stopping points, with remarkable precision and minimal errors, as evident in Graph 13.1.

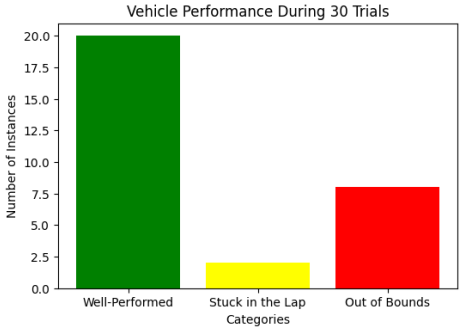


Figure 13.1: Testing Car in the lab

One of the standout features of our solution is its exceptional efficiency, not only in terms of its advanced algorithms but also in its minimal power consumption. The ESP32 device exhibited impressively low power usage, underscoring the overall efficiency of our implementation.

To summarize our project objectives:

1. **Autonomous Car System with Micro-controller Camera (ESP32 CAM):** We have successfully developed a sophisticated autonomous car system using the ESP32 CAM, enabling seamless visual perception and navigation and tested in lab condition as discussed in section 11.
2. **Machine Learning Algorithm for Navigation:** Section 7.1.4 describes the designed and implemented an advanced machine learning algorithm that effectively utilizes processed visual data for navigation, ensuring optimal path planning for the autonomous vehicle. A suitable Machine learning model is opted for by working on hyperparameters and choosing the best model. Results on these are explained in 8.
3. **Integration of DC Motor and Micro-controller:** We have seamlessly integrated the DC motor with the ESP32 CAM micro-controller, enabling precise control over the car's movements, including speed modulation, directional shifts, and steering adjustments detailed in section 10, 9.
4. **Optimization of Computational Efficiency and Memory Usage:** Our project has successfully prioritized computational efficiency and achieved impressive results within the constraints of micro-controller technology, including optimizing memory usage. Figure 13.2 provides a visual comparison of memory usage at different checkpoints for PCA, CNN, and MobileNetV2.

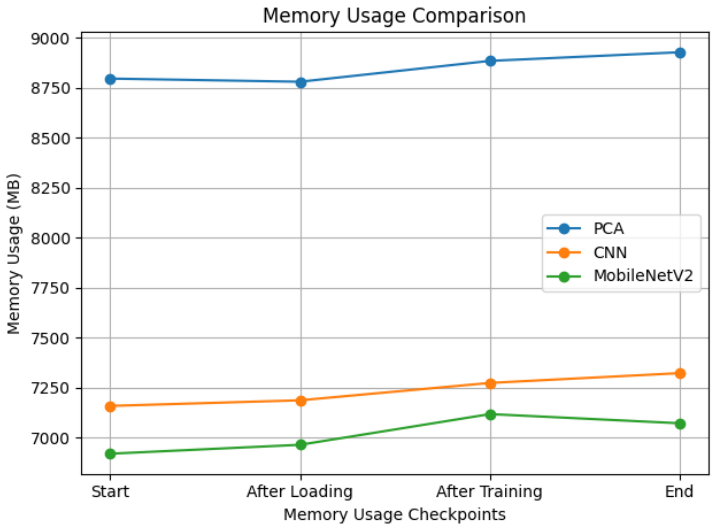


Figure 13.2: Memory usage comparison at different checkpoints for PCA, CNN, and MobileNetV2

To conclude, our project serves as a prime example of what micro-controller-based solutions can achieve in the field of autonomous vehicles. It demonstrates a seamless blend of sophisticated algorithms, streamlined hardware integration, and resource-efficient optimization.

13.2 Future Direction for Improvement

In our current project, we have successfully applied machine learning algorithms for path following. In future developments, we plan to extend this application to include obstacle detection and avoidance capabilities.

While we have currently integrated only Convolutional Neural Networks (CNN) with a microcontroller, our future goals involve integrating additional machine learning algorithms such as Principal Component Analysis (PCA), MobileNetV2, and others with the microcontroller.

We also aim to investigate the variations in accuracy when running machine learning code in Python versus using Edge Impulse and explore methods to optimize monitoring speed for enhanced performance.

Bibliography

[1] Keiron O'Shea and Ryan Nash. "An Introduction to Convolutional Neural Networks". In: *CoRR* abs/1511.08458 (2015). arXiv: 1511.08458. URL: http://arxiv.org/abs/1511.08458 (cit. on pp. 5, 14, 15).

[2] Kalyan Sudhakar and Sameer Mohammad. "Machine Learning -autonomous Vehicles". In: *SSRN Electronic Journal* 8 (July 2018), pp. 314–333 (cit. on pp. 5, 14).

[3] Mark Sandler et al. *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. 2019. arXiv: 1801.04381 [cs.CV] (cit. on pp. 4, 15).

[4] Yaoqi Sun et al. "Image classification base on PCA of multi-view deep representation". In: *Journal of Visual Communication and Image Representation* 62 (2019), pp. 253–258. DOI: https://doi.org/10.1016/j.jvcir.2019.05.016. URL: https://www.sciencedirect.com/science/article/pii/S1047320319301713 (cit. on pp. 5, 15, 19).

[5] Xue Ying. "An Overview of Overfitting and its Solutions". In: *Journal of Physics: Conference Series* 1168 (Feb. 2019), p. 022022. DOI: 10.1088/1742-6596/1168/2/022022 (cit. on p. 32).

[6] Kristian Dokic. "Microcontrollers on the Edge – Is ESP32 with Camera Ready for Machine Learning?" In: *Image and Signal Processing*. Ed. by Abderrahim El Moataz et al. Cham: Springer International Publishing, 2020, pp. 213–220 (cit. on pp. 5, 14).

[7] Kristian Dokic, Marko Martinovic, and Dubravka Mandusic. "Inference speed and quantisation of neural networks with TensorFlow Lite for Microcontrollers framework". In: *2020 5th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*. 2020, pp. 1–6. DOI: 10.1109/SEEDA-CECNSM49515.2020.9221846 (cit. on p. 37).

[8] Pushkara Sharma, Pankaj Hans, and Subhash Chand Gupta. "Classification Of Plant Leaf Diseases Using Machine Learning And Image Preprocessing Techniques". In: *2020 10th International Conference on Cloud Computing, Data Science and Engineering (Confluence)*. 2020, pp. 480–484. DOI: 10.1109/Confluence47617.2020.9057889 (cit. on pp. 4, 17, 21).

[9] Taiwo Samuel Ajani, Agbotiname Lucky Imoize, and Aderemi A. Atayero. "An Overview of Machine Learning within Embedded and Mobile Devices–Optimizations and Applications". In: *Sensors* 21.13 (2021). DOI: 10.3390/s21134412. URL: https://www.mdpi.com/1424-8220/21/13/4412 (cit. on pp. 5, 14, 22).

[10] Colby R. Banbury et al. "MLPerf Tiny Benchmark". In: *CoRR* abs/2106.07597 (2021). arXiv: 2106.07597. URL: https://arxiv.org/abs/2106.07597 (cit. on p. 8).

[11] Khalid M. Kahloot and Peter Ekler. "Algorithmic Splitting: A Method for Dataset Preparation". In: *IEEE Access* 9 (2021), pp. 125229–125237. DOI: 10.1109/ACCESS.2021.3110745 (cit. on p. 8).

[12] Md Ziaul Haque Zim. "TinyML: Analysis of Xtensa LX6 microprocessor for Neural Network Applications by ESP32 SoC". In: *CoRR* abs/2106.10652 (2021). arXiv: 2106.10652. URL: https://arxiv.org/abs/2106.10652 (cit. on p. 12).

[13] Hui Han and Julien Siebert. "TinyML: A Systematic Review and Synthesis of Existing Research". In: *2022 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*. 2022, pp. 269–274. DOI: 10.1109/ICAIIC54071.2022.9722636 (cit. on p. 1).

[14] Michail Moraitis, Konstantinos Vaiopoulos, and Athanasios T. Balafoutis. "Design and Implementation of an Urban Farming Robot". In: *Micromachines* 13.2 (2022). DOI: 10.3390/mi13020250. URL: https://www.mdpi.com/2072-666X/13/2/250 (cit. on p. 8).

[15] Ismail Muraina. "IDEAL DATASET SPLITTING RATIOS IN MACHINE LEARNING ALGORITHMS: GENERAL CONCERNS FOR DATA SCIENTISTS AND DATA ANALYSTS". In: Feb. 2022 (cit. on p. 8).

[16] Partha Pratim Ray. "A review on TinyML: State-of-the-art and prospects". In: *Journal of King Saud University - Computer and Information Sciences* 34.4 (2022), pp. 1595–1623. DOI: https://doi.org/10.1016/j.jksuci.2021.11.019. URL: https://www.sciencedirect.com/science/article/pii/S1319157821003335 (cit. on p. 1).

[17] Swapnil Sayan Saha, Sandeep Singh Sandha, and Mani Srivastava. "Machine Learning for Microcontroller-Class Hardware: A Review". In: *IEEE Sensors Journal* 22.22 (Nov. 2022), pp. 21362–21390. DOI: 10.1109/jsen.2022.3210773. URL: https://doi.org/10.1109%2Fjsen.2022.3210773 (cit. on p. 10).

[18] TensorFlow Models. *MobileNetV2 Source Code*. Accessed on Date. Year or Last Updated. URL: https://github.com/tensorflow/models/tree/master/research/slim/nets/mobilenet (cit. on pp. 4, 15, 21).

A Appendix

A.1 Useful Links and Resources

- **ESP Drivers** - https://shorturl.at/bstv5
- **ESP Basics** - https://rb.gy/60y67
- **Platform IO Basics** - https://shorturl.at/fsBO0
- **ESP Wi-Fi Connection** - https://shorturl.at/luNQ1
- **ESP Pinout Diagram** - https://rb.gy/vfpxr
- **Circuit Diagram** - https://shorturl.at/ltG17
- **Initial Approach Logic** - https://rb.gy/iosrs
- **TensorFlow Lite with Platform IO** - https://shorturl.at/euM05
- **PWM Connection and Configuration** - https://rb.gy/eo3jl
- **Edge Impulse Tutorial** - https://rb.gy/95xwb
- **Chassis Assembly Tutorial** - https://shorturl.at/pARV9

A.2 Data-set sample images

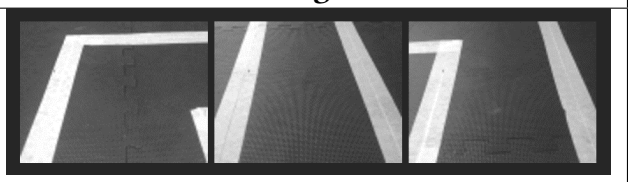
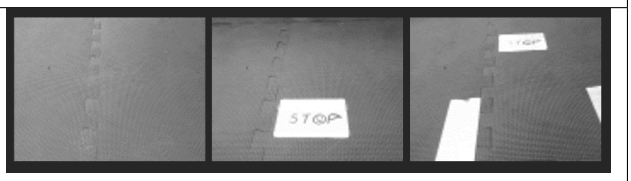
Classes	Images
Forward	
Right	
Left	
Stop	

Table A.1: Class Samples and Images

A.3 Machine Learning Model Codes

You can access the machine learning model codes for this project on GitHub by following this link:

https://shorturl.at/adlRU

A.4 Edge Impulse Code for Deployment

The code for the deployment of our project using Edge Impulse can be found at the following link:

https://studio.edgeimpulse.com/public/274389/latest main.tex