

MASTER'S PROGRAM IN ELECTRONIC ENGINEERING



VNIVERSITAT
DE VALÈNCIA

MASTER'S THESIS

**RELIABLE IMAGE RECOGNITION
IN AUTONOMOUS DRIVING:
EVALUATION OF EDGE DEVICES
FOR DEEP LEARNING SOFTWARE**

**AUTHOR:
YANZHANG SONG**

**SUPERVISORS:
VALERO LAPARRA PÉREZ-MUELAS
VICENT GIRBÉS JUAN**

DECEMBER 2023

Declaration of Academic Integrity

I hereby confirm that the contents of this master thesis are original, and that no sources other than the ones mentioned in the text and listed in the references have been used. This thesis has not been previously presented to another examination board and has not been published.

A handwritten signature in black ink that reads "Yanzhang Song". The signature is written in a cursive, flowing style.

Espoo, Finland, 26th of November, 2023

Yanzhang Song

Acknowledgements

It's been a long and winding road. First of all, I would like to acknowledge and give my warmest thanks to my supervisors Valero Laparra Pérez-Muelas and Vicent Girbés Juan for their continuous guidance and patience while I carried through all the stages of my project.

I would also like to thank the University of Valencia for giving me the fantastic opportunity to increase my knowledge and thus, paving the road towards my professional goals. I am deeply grateful for the kindness shown by my classmates, particularly Camilo, Joaquín, Lucía and Jorge. You have encouraged and motivated me along the way. Because of you, I now have many wonderful and funny memories about my time in Valencia and I will cherish them forever.

My parents have always been there for me, supported me and made it possible for me to pursue my dreams and study in Europe. Xie xie nimen.

Finally, thank you Erika for your endless love and support, and for constantly pushing me to keep on going, even when I thought it was not possible.

Abstract

In recent years, deep learning has enjoyed great success in a variety of applications. Experimental results indicate that the performance of using deep learning is superior compared to traditional machine learning methods in areas such as image processing, computer vision, speech recognition and robotics.

In the automotive sector, many companies are actively working to develop autonomous driving technologies with the goal of eliminating the human factor in traffic accidents and urban road congestion and making the travel more comfortable. However, the technologies currently used are still in relatively early stages of development and further research is needed. Some of the biggest problems related to self-driving cars include image recognition inaccuracy which involves image segmentation.

This work is a technical project in which the accuracy of image recognition in traffic context is evaluated by comparing the performance of different edge devices. While numerous hardware and software can be utilized for this purpose, the comparison has been limited to three devices: Raspberry Pi 4, Jetson Nano and Intel Neural Compute Stick 2.

Deep learning models were implemented on the edge devices in order to perform traffic sign classification and lane detection. The learning process and the performance of the selected edge devices is evaluated for their capacity to process artificial intelligence in recognizing images from a moving vehicle through the means of lane detection and traffic sign training. For each of these devices, advantages and limitations will be discussed in the context of deep learning in autonomous driving.

Keywords: *self-driving car, autonomous driving, traffic signs, image recognition, edge devices, lane detection, deep learning, computer vision, Raspberry Pi*

Resumen

En los últimos años, el aprendizaje profundo ha cosechado un gran éxito en diversas aplicaciones. Los resultados experimentales indican que el rendimiento del uso del aprendizaje profundo es superior en comparación con los métodos tradicionales de aprendizaje automático en áreas como el procesamiento de imágenes, la visión por ordenador, el reconocimiento del habla y la robótica.

En el sector de la automoción, muchas empresas trabajan activamente para desarrollar tecnologías de conducción autónoma con el objetivo de eliminar el factor humano en los accidentes de tráfico y la congestión de las vías urbanas y hacer más cómodos los desplazamientos. Sin embargo, las tecnologías utilizadas actualmente se encuentran aún en fases relativamente tempranas de desarrollo y es necesario seguir investigando. Algunos de los mayores problemas relacionados con los coches autoconducidos son la imprecisión en el reconocimiento de imágenes, que implica la segmentación de las mismas.

Este trabajo es un proyecto técnico en el que se evalúa la precisión del reconocimiento de imágenes en el contexto del tráfico comparando el rendimiento de distintos dispositivos de borde. Aunque pueden utilizarse numerosos hardware y software para este fin, la comparación se ha limitado a tres dispositivos: Raspberry Pi 4, Jetson Nano e Intel Neural Compute Stick 2.

Se implementaron modelos de aprendizaje profundo en los dispositivos de borde para realizar la clasificación de señales de tráfico y la detección de carriles. Se evalúa el proceso de aprendizaje y el rendimiento de los dispositivos de borde seleccionados por su capacidad para procesar inteligencia artificial en el reconocimiento de imágenes de un vehículo en movimiento a través de los medios de detección de carriles y formación de señales de tráfico. Para cada uno de estos dispositivos, se discutirán las ventajas y limitaciones en el contexto del aprendizaje profundo en la conducción autónoma.

Palabras clave: coche autoconducido, conducción autónoma, señales de tráfico, reconocimiento de imágenes, dispositivos de borde, detección de carriles, aprendizaje profundo, visión por computador, Raspberry Pi.

Resum

En els darrers anys, l'aprenentatge profund ha aconseguit un gran èxit en diverses aplicacions. Els resultats experimentals indiquen que el rendiment de l'ús de l'aprenentatge profund és superior en comparació amb els mètodes tradicionals d'aprenentatge automàtic en àrees com el processament d'imatges, la visió per ordinador, el reconeixement de la parla i la robòtica.

Al sector de l'automoció, moltes empreses treballen activament per desenvolupar tecnologies de conducció autònoma amb l'objectiu d'eliminar el factor humà en els accidents de trànsit i la congestió de les vies urbanes i fer més còmodes els desplaçaments. No obstant això, les tecnologies utilitzades actualment es troben encara en fases relativament primerenques de desenvolupament i cal continuar investigant. Alguns dels problemes més importants relacionats amb els cotxes autoconduïts són la imprecisió en el reconeixement d'imatges, que implica la segmentació de les mateixes.

Aquest treball és un projecte tècnic on s'avalua la precisió del reconeixement d'imatges en el context del trànsit comparant el rendiment de diferents dispositius de vora. Tot i que es poden utilitzar nombrosos maquinari i programari per a aquesta finalitat, la comparació s'ha limitat a tres dispositius: Raspberry Pi 4, Jetson Nano i Intel Neural Compute Stick 2.

Es van implementar models d'aprenentatge profund als dispositius de vora per fer la classificació de senyals de trànsit i la detecció de carrils. S'avalua el procés d'aprenentatge i el rendiment dels dispositius de vora seleccionats per la seva capacitat per processar intel·ligència artificial en el reconeixement d'imatges d'un vehicle en moviment a través dels mitjans de detecció de carrils i formació de senyals de trànsit. Per a cadascun d'aquests dispositius, es discutiran els avantatges i les limitacions en el context de l'aprenentatge profund en la conducció autònoma.

Paraules clau: cotxe autoconduït, conducció autònoma, senyals de trànsit, reconeixement d'imatges, dispositius de vora, detecció de carrils, aprenentatge profund, visió per ordinador, Raspberry Pi.

Index

1	<i>Introduction.....</i>	14
1.1	Objective	14
2	<i>State of the Art Analysis.....</i>	16
2.1	Current activity in the field of autonomous driving	17
2.2	Current challenges	19
3	<i>Theoretical Framework of Deep Learning</i>	21
3.1	Deep learning.....	21
3.1.1	Measuring deep learning performance	22
3.1.2	Deep learning in autonomous driving	23
3.2	Edge computing	23
3.3	Computer vision and image recognition.....	25
4	<i>Software development.....</i>	27
4.1	Python	27
4.2	OpenCV	27
4.3	TensorFlow and Convolutional Neural Network.....	28
5	<i>Programming Content.....</i>	29
5.1	Lane detection	29
5.1.1	lane_detection	29
5.1.2	lane_detection_main.....	37
5.2	Identification of traffic signs.....	37
5.2.1	traffic_sign_training	38
5.2.2	traffic_sign_module	43
5.2.3	get_traffic_sign	46
6	<i>Hardware Implementation.....</i>	47
6.1	Raspberry Pi 4.....	47
6.1.1	Interface of Raspberry Pi 4.....	47
6.2	Jetson Nano	48
6.2.1	Interface of Jetson Nano.....	48
6.3	Neural Compute Stick 2.....	49

6.3.1 Connecting Diagram	50
6.4 Hardware configuration comparison.....	51
6.5 Installing the operating system.....	53
6.5.1 Installation on Raspberry Pi 4	53
6.5.2 Problems on Raspberry Pi 4 and their solutions.....	55
6.5.3 Installation on Jetson Nano	57
6.5.4 Problems on Jetson Nano and their solutions.....	59
6.5.5 Installation of Neural Computer Stick 2.....	60
7 Results and evaluation.....	62
7.1 Comparison on running the training programming.....	62
7.2 Observations on Raspberry Pi 4	62
7.3 Observations on Jetson Nano	63
7.4 Observations on Neural Compute Stick 2	64
7.5 Edge device comparison for image processing speeds	64
7.6 Donkey car.....	70
7.6.1 The donkey car setup	73
7.6.2 Results on the donkey car.....	74
8 Conclusion	76
9 References.....	78

Index of Figures

Figure 1: Autonomous test mileage in different companies [8].....	19
Figure 2: Artificial intelligence, machine learning and deep learning distinguished	21
Figure 3: Performance indicators.....	23
Figure 4: The LeNet5 architecture	28
Figure 5: Programming for pre-processing input images	29
Figure 6: Graying the image.....	30
Figure 7: Blurring the image	30
Figure 8: Cannying the image	31
Figure 9: The area of interest.....	31
Figure 10: Programming for image warping.....	32
Figure 11: Warping area of interest	32
Figure 12: Programming for histogram to mark lanes	33
Figure 13: Road line recognition	34
Figure 14: Programming for parameter return values	35
Figure 15: Programming for orientation and position calculations	35
Figure 16: Programming for lane line integration.....	36
Figure 17: Running functions for verification	37
Figure 18: Programming for path selection.....	38
Figure 19: Programming for image import.....	39
Figure 20: Programming for splitting image data and preprocessing image.....	39
Figure 21: Programming for preprocessing and augmenting images	40
Figure 22: Programming for convolution neural network model.....	41
Figure 23: Programming for training.....	41
Figure 24: Programming for specified color removal.....	43
Figure 25: Color removal for better traffic sign readability	44
Figure 26: Programming for contour detection	44
Figure 27: Programming for image preprocessing and traffic sign number listing..	45
Figure 28: Programming for traffic sign number definition	46
Figure 29: The Raspberry Pi 4 hardware	48
Figure 30: The Jetson Nano interface	49
Figure 31: The Neural Compute Stick 2 Interface	50
Figure 32: The donkey car connection diagram.....	51
Figure 33: The Raspberry Pi installer	54

Figure 34: Raspberry Pi OS selection.....	54
Figure 35: Unrecognized command line	56
Figure 36: Files requiring modifications	56
Figure 37: Numpy.core.multiarray failed to import	57
Figure 38: Cannot find cv2.....	57
Figure 39: Showing the cv2 version.....	57
Figure 40: Zipped image selection	58
Figure 41: SD card selection.....	58
Figure 42: The camera code from Jetson Nano.....	59
Figure 43: The camera code from Raspberry Pi 4.....	59
Figure 44: No module named 'numpy.testing.nosetester'	60
Figure 45: Cannot determine CPU frequency	60
Figure 46: ARM64 does not support NUMA	60
Figure 47: The macOS version used.....	61
Figure 48: Notification message when installing the driver on macOS	61
Figure 49: Notification message when installing the driver on Jetson Nano.....	61
Figure 50: Programming for time calculation.....	62
Figure 51: Total processing time of Raspberry Pi 4	63
Figure 52: GL-Z software monitoring the GPU	63
Figure 53: Total processing time on Jetson Nano	64
Figure 54: Jtop software monitoring the GPU.....	64
Figure 55: Total processing time on Raspberry Pi 4 + NCS2.....	64
Figure 56: Edge devices + camera.....	65
Figure 57: Programming for time calculation.....	66
Figure 58: Comparison graph of traffic sign recognition speeds.....	70
Figure 59: The donkey car control schematic	71
Figure 60: The schematic diagram	72
Figure 61: The donkey car setup.....	73
Figure 62: L298N Motor Driver Module Pinout	74
Figure 63: Testing a traffic scenario	75

Index of Tables

Table 1: Levels of autonomy in self-driving cars.....	17
Table 2: Comparison of key features in cloud and edge computing.....	25

Table 3: Comparison of hardware configuration	52
Table 4: Lane detection comparison	69
Table 5: Summary of differences between edge devices	77

1 Introduction

Deep learning as a subset of machine learning has gone through dramatic development in the recent years, and as a result, state-of-the-art deep learning techniques are being widely applied across industries. Particularly in the automotive domain, a wealth of potential use cases for deep learning techniques can be found in such areas as autonomous driving, advanced driving assistance systems and predictive maintenance of vehicles. These applications typically require storing and processing massive volumes of unstructured data, which is facilitated by the large neural networks utilized in deep learning.

Despite the rapid development, the deep technologies are still in relatively early stages of development and further research is needed. One of the biggest problems related to self-driving cars is the inaccuracy of image classification and prediction, which lays the foundation for the topic of this final project.

This work is a technical project in which the accuracy of image recognition in traffic is evaluated by comparing the performance of different edge devices. Deep learning methodologies serve here as the basis for classification and segmentation of images of driving paths and traffic signs. After testing the programming on edge devices, the results are analyzed in order to highlight the differences between each of them.

1.1 Objective

The main objective of this project is to compare the performance of different edge devices in the context of image recognition. While infinite opportunities for implementation exist, the specific area of interest here is the self-driving car technology, which is currently undergoing significant development and resulting in higher levels of automation in vehicles.

In order to perform tests for comparison, a system is developed with the ability to read and interpret simplified traffic scenarios. Deep learning methodologies serve here as the basis for classification and segmentation of images of driving paths and traffic signs. A number of libraries are used here to train the traffic sign classification and lane detection models.

The programming consists of a traffic sign training model and a lane detection algorithm, which are run on the selected edge devices. A camera is used to capture images of traffic signs and traffic lanes. Using Jetson Nano, Raspberry Pi 4 alone

and Raspberry Pi 4 together with Neural Compute Stick 2 as an accelerator, these images are then processed based on the trained models. The obtained results will be used to highlight the differences between the edge devices.

2 State of the Art Analysis

The concept of edge learning and reliable image recognition in autonomous driving offers an opportunity to develop technologies that improve people's daily activities, particularly through the advancement in the application of technologies such as artificial intelligence, machine learning and deep learning.

In a broad sense, cars with driving assistance functions can be referred to as "self-driving cars". Other terms that have been used to refer to self-driving cars include "autonomous vehicles", "robocars", "smart transportation robots (STR)" or "driverless cars" [1].

In recent years, a wealth of literature on autonomous driving studies has emerged from a wide range of academic disciplines. While autonomous driving is a classic feature in countless sci-fi movies, it is now becoming a reality and seen as the new modality, which will enable easier transportation in the future. One of the reasons that has sparked interest in the concept of unmanned vehicles is their potential in helping to avoid car accidents, which are typically caused by "human operation errors". However, it is safe to say that their existence would improve the outlook of our lives in numerous ways.

While autonomous driving has only recently been regarded as a viable solution to facilitate transportation, the idea is not exactly novel. Already nearly 80 years ago, the concept of an autonomous car was presented at the 1939 New York World's Fair in the form of General Motor's Futurama [1]. According to [2], the cornerstones of autonomous driving systems consist of contemporary developments in communication networks and wireless connectivity, the advent of precise and robust sensors that continuously miniaturize in size and cost, alongside with artificial intelligence.

To better understand the current state of the art, it is important to first have an overview to what the level the development of autonomous vehicles is today. As demonstrated in Table 1, various levels of autonomy can be identified in self-driving cars. While altogether six different levels ranging from 0 to 5 have been defined, the current technology has not been able to create a fully automated car yet. One of the most advanced vehicles developed so far is the Chinese electric passenger carmaker Arcfox, whose new Alpha S model published in April 2021 is said to possess the best capabilities in self-driving to date, placing it at Level 4 [3].

Level	Defining Characteristics
Level 0: No automation	The driver is responsible for all core driving tasks. However, Level 0 vehicles may still include features like automatic emergency braking, blind-spot warnings, and lane-departure warnings.
Level 1: Driver assistance	Vehicle navigation is controlled by the driver, but driving-assist features like lane centering or adaptive cruise control are included.
Level 2: Partial automation	Core vehicle is still controlled by the driver, but the vehicle is capable of using assisted-driving features like lane centering and adaptive cruise control simultaneously.
Level 3: Conditional automation	Driver is still required but is not needed to navigate or monitor the environment if certain criteria are met. However, the driver must remain ready to resume control of the vehicle once the conditions permitting an autonomous driving system are no longer met.
Level 4: High automation	The vehicle can carry out all driving functions and does not require that the driver remain ready to take control of navigation. However, the quality of the autonomous driving system navigation may decline under certain conditions such as off-road driving or other types of abnormal or hazardous situations. The driver may have the option to control the vehicle.
Level 5: Full automation	The autonomous driving system is advanced enough that the vehicle can carry out all driving functions no matter the conditions. The driver may have the option to control the vehicle.

*Table 1: Levels of autonomy in self-driving cars
Adapted from [4]*

In relation to the problem of how a self-driving car can navigate its way from Point A to Point B safely, a human-like observation and decision-making is required, which is made possible by deep learning. In order for a vehicle to make such observations and smart decisions about driving, at least Level 4 or ideally Level 5 of autonomy would be required.

2.1 Current activity in the field of autonomous driving

Autonomous driving is seen as having a potentially revolutionary impact on future traffic and social patterns across countries, which is why it has become the focus of attention on a global scale. Through countless development projects

worldwide, particularly in the United States and China, promising prototypes of self-driving vehicles have been developed in recent years.

Also, governments and other decision-making units across the world are prepared to welcome the arrival of the era of autonomous driving. Driverless mobility is discussed in the “On the road to automated mobility: An EU strategy for mobility of the future” [5], where it is estimated that fully autonomous driving could become commonplace by 2030. Since its publication, the Commission has been actively supporting research on cooperative, connected and automated mobility. Similarly, according to a report issued by the Japan Cabinet Office and the Japan Automobile Manufacturers Association, autonomous driving could be performed on national and local roads as soon as 2025 [6].

It should be pointed out that while self-driving cars are still under development, unmanned transportation has been in existence for more than a decade [7]. The example of train transportation is mentioned, in which self-driving technology has been successfully used for years. The SkyTrain in Vancouver, Canada and Yurikamome in Tokyo, Japan are listed as some of the examples of autonomous rail systems which effortlessly transport massive crowds of passengers on a daily basis. However, the introduction of self-driving cars in public roads is much higher in complexity due to the need to constantly interact with other vehicles and pedestrians and react appropriately to frequently changing traffic circumstances.

While still under development, there are currently taxis, minibuses and cars on roads that have already been established with this type of technology. [8] brings up the Google spin-off project Waymo as a well-known example which is regarded as the world leader in unmanned driving technology and specifically fully autonomous ride services. Its ride-hailing service Waymo One currently operates in the suburbs of Phoenix, USA with a fleet of several hundred autonomous vehicles [9]. The robotaxi can be ordered through a mobile app, after which the vehicle will drive to your location and transport you to the desired destination just like an ordinary taxi — except this one has no driver. For safety reasons, only the back seat is currently open for passengers [8].

Waymo is certainly not the only major project in the field of autonomous transportation. Many manufacturers internationally have been developing and testing autonomous technologies in their vehicles, including General Motors and Pony.AI. The extensive efforts put into their development can be seen in the published number of kilometers traveled with autonomous cars. According to [8], Waymo logged 628,000 miles (about 1 million kilometers) in the year 2020, while GM (General Motors) Cruise reached 770,000 miles (about 1.23 million kilometers), and the Chinese Pony.AI logged the third highest mileage with 225,000 miles (about

360,000 kilometers).

As demonstrated in Figure 1, most of the leading manufacturers in autonomous car advancements are of Chinese or American origin. According to [8], approximately 10% of new cars sold in China in the first half of 2020 incorporated Tier 2 automation technology. Furthermore, the Chinese automotive companies' plans to build Tier 2 and Tier 3 autonomous driving technology vehicles by 2025 account for 50% of all new car sales, which is estimated to reach as high as 70% by 2030 [10].

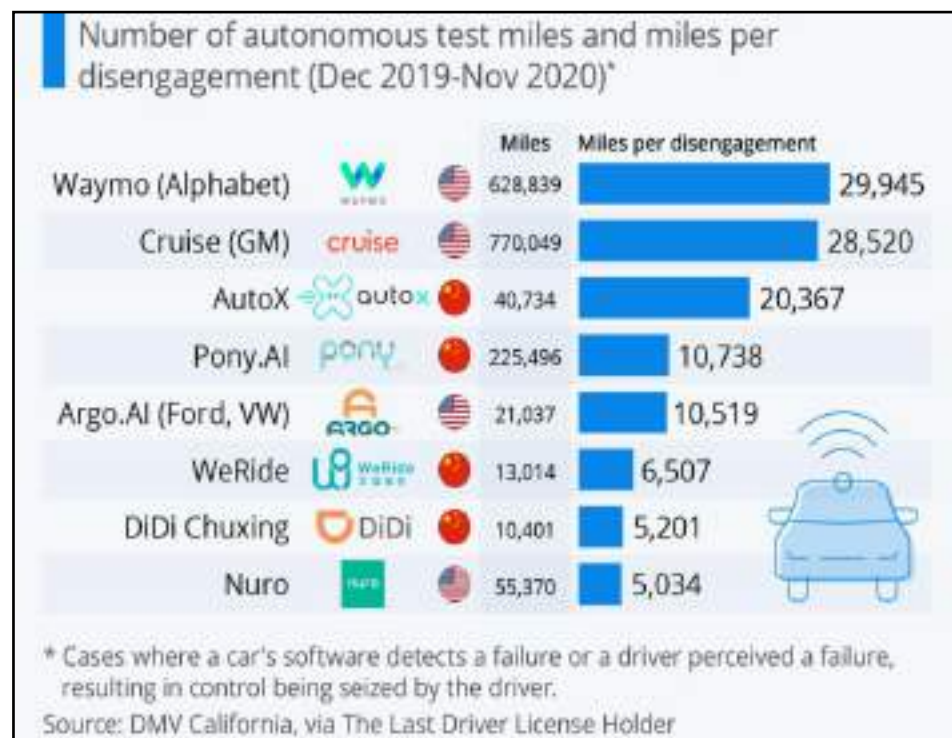


Figure 1: Autonomous test mileage in different companies [8]

2.2 Current challenges

Alongside the obvious benefits such as reduction of traffic congestion and accidents as well as facilitating our lives in general, there are many challenges and uncertainties that autonomous driving must overcome to be fully accepted globally.

Most of the autonomous vehicles developed until now can guarantee autonomous driving only to a certain extent. However, to ensure for a car to achieve fully autonomous driving and the ability to adapt to all road conditions, it must have the ability to self-learn, a complete environmental perception system, a central decision-making system, high-precision positioning system and so on, all of which still require further research and development. The perception of vehicles relies on

sensory input devices such as cameras, radar, and lasers to allow it to perceive the world around it, in a way creating a digital map to follow.

Firstly, a significant challenge at this point relates to the safety concerns and particularly the vehicle's ability to react correctly to unexpected situations and constantly changing situations. Diverse road types, complex traffic scenarios and extreme weather environments pose enormous challenges to the perception, decision-making and control systems of autonomous driving. At the same time, because the responsibility for safety is transferred to the vehicle, high requirements are imposed on the reliability of the automated driving system. The maturity of existing technologies, such as environmental perception, planning and decision-making, and cable control execution, is currently insufficient to support high-level autonomous driving mass production applications [10].

Secondly, infrastructural challenges can also be identified. Autonomous driving not only involves the product of the car itself, but also requires the coordinated development of vehicles, networks, roads, and clouds. It also requires the construction of various infrastructures such as smart roads, wireless communication networks and high-precision location services.

[6] points out that the development of transport infrastructure faces problems such as long investment cycles and large investment quotas, which have affected the progress of construction. As an example, China has been one of the forerunners in introducing self-driving cars to its roads. [10] predicts that autonomous vehicles are becoming a big business in China, but also points out that the lack of standardization of traffic signs and traffic lights hinders the development. Because Chinese drivers tend to ignore traffic rules, programming and training autonomous vehicles increases uncertainty. It is therefore necessary to optimize the decision-making algorithm for Chinese roads, which may require further effort and training.

Finally, the development of autonomous vehicles has a direct impact on data security. The self-driving car itself is a powerful information-gathering device. In the process of autonomous driving, geographic information, vehicle information and passenger information will be collected and recorded, and much of the information will be uploaded to the cloud for storage. Without strict management regulations, the leakage of a lot of sensitive information could cause national strategic security issues. If information security protection is lacking, vehicle data can be leaked or controlled remotely, leading to serious security risks that are difficult to control without sufficient policies and regulations on car data security [6].

3 Theoretical Framework of Deep Learning

This section provides a broad overview of the concept of deep learning. Additionally, some of the key applications used in deep learning in the context of autonomous driving as well as in this thesis are defined and described more in depth.

3.1 Deep learning

Deep learning is a relatively new concept which emerged around a decade ago. It can be classified as a very advanced machine learning technique which imitates the functions of the human brain. Deep learning is a crucial element of data science, including statistics and predictive models. It has proven to be beneficial for data scientists who are tasked with collecting, analyzing, and interpreting enormous amounts of data, as deep learning makes this process much faster and easier [11]. Figure 2 illustrates how deep learning is related to machine learning and artificial intelligence in a broader sense.

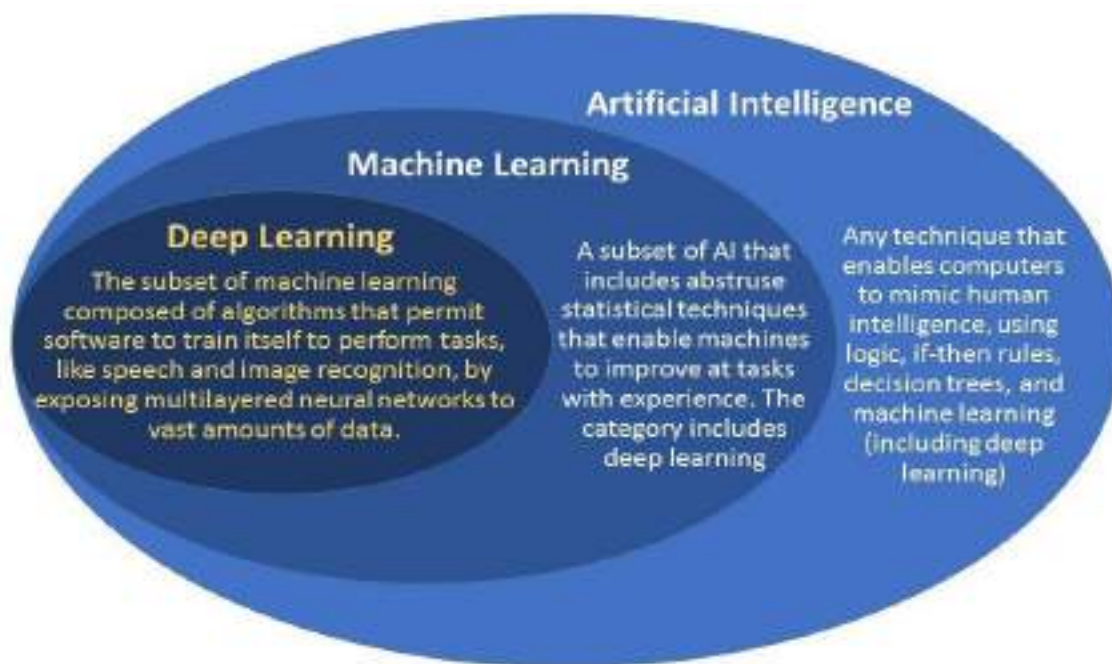


Figure 2: Artificial intelligence, machine learning and deep learning distinguished [12]

Deep learning processes data and creates patterns, which can be used to make decisions exactly how a human being would [11]. In deep learning, the device can be trained with example data from the learning process, and it continues to learn more and more based on the collected information. The science behind deep

learning is rather complex, as it describes a family of learning algorithms rather than a single method that can be used to learn complex prediction models, such as multi-layer neural networks with many hidden units [13]. With the powerful capability of automatic feature extraction, deep learning has achieved surprisingly excellent performance in applications that machine learning has not been able to overcome, enabling such functions that computers could not perform in the past [11].

Deep learning can be applied to countless purposes. Some examples of this include automatic machine translations, voice recognition, visual translations and cybersecurity, which are technologies that most people have already adopted to daily use. Over the past decade, numerous deep learning applications have emerged. In addition to solving classification and regression problems of traditional machine learning, they can also be applied to dimensionality reduction and even allow the computer to automatically generate text and images [14].

Enabling higher accuracy, there is significant potential for deep learning particularly in technologies involving image recognition. Among the most promising fields is the automotive industry and specifically self-driving vehicles, which is expected to be revolutionized by deep learning applications in the future.

3.1.1 Measuring deep learning performance

Deep learning can be used for both supervised and unsupervised learning. The success metric depends on the specific application area in which deep learning is applied. [15] suggests that one effective way to measure accuracy in object detection is by Mean Average Accuracy (mAP), which measures the degree of overlap between the predicted position of the object and the actual position of the terrain, and the average value across various categories of objects.

In machine translation, accuracy can be measured by the bilingual assessment index, which compares candidates' translations with several basic authentic reference translations. Other general indicators of system performance which are not related to the application include performance, latency, and power, as listed in Figure 3.

Performance metrics
Latency (s)
Energy (mW, J)
Concurrent requests served (#)
Network bandwidth (Mbps)
Accuracy (application-specific)

Figure 3: Performance indicators [15]

3.1.2 Deep learning in autonomous driving

Autonomous driving involves extremely complex multi-sectoral integration. In addition to traditional vehicle manufacturing, it also involves many emerging technologies, such as artificial intelligence and the Internet of Things. Because it is difficult for traditional manufacturers to form relevant technology research and development capabilities in a short time, this has given the industry several disadvantages in the production of autonomous vehicles. Technology companies related to engineering and technology have an excellent opportunity to enter this massive emerging market [16].

In order to make the self-driving cars more intelligent, they need to be equipped with smart sensors and analytics tools that collect and analyze heterogeneous data related to passengers on-board, pedestrians, and the environment in real-time, in which artificial intelligence plays a significant role [17]. It is applied to the three main components of self-driving car technology, namely cameras, radar and lidar, which give the car a clear understanding of the environment so it can navigate safely [18].

Through the application of deep learning, the ultimate goal is to reduce the number of traffic accidents caused by human error, as well as to increase the general convenience of transportation. Although further development and research is still required before self-driving cars based on deep learning technologies can be mass-produced, the technology is developing rapidly. As a result, more and more services and products are being created for future needs.

3.2 Edge computing

The exponential use of deep learning in a variety of applications has led to cloud servers processing larger volumes of data than ever before. As stated by [19], while

deep learning algorithms are continuously being refined to deliver better results, centralized cloud computing as the preferred computing model is struggling to meet the bandwidth and latency requirements. Such potential latency constraints, when not fulfilled, could lead to catastrophic consequences particularly in such applications as autonomous driving.

In the context of autonomous driving, [10] pinpoints intelligent sensing and perception as the most critical issues. As vehicles must first collect the information from sensors such as cameras and radars, and then conduct an intelligent perception and decision, relying entirely on vehicle-based and cloud-based solutions may not meet the computational capacity requirements for computational capacity, real-time feedback and security.

For this purpose, edge computing is seen as a viable solution which offers sufficient capacity to reduce the server overload and guarantees low latency. This is particularly beneficial in applications where short response times are critical, such as in autonomous vehicles as well as wearables and other devices based on IoT.

According to [20], the easiest way to use an edge server is to transfer all the computation from the terminal device to the edge server. In this case, the terminal device sends its data to a nearby edge server and receives the corresponding result after it has been processed by the server. Since edge computing involves delegating data processing tasks to devices at the edge of the network, as close as possible to data sources, this enables real-time data processing at a very high speed, which is a must for complex IoT solutions with machine learning capabilities. On top of that, it mitigates network limitations, reduces power consumption, increases security, and improves data privacy [21].

To distinguish the key differences between cloud computing and edge computing, some of the different features are compared in Table 2 as follows.

Features	Cloud servers	Edge servers and devices
Security	Moderate. As sensitive data is shared in a public cloud, security and privacy may be compromised.	High. Less sensitive data transferred improves privacy and security.
Computing power	High. Computationally powerful, but shared by many users.	Moderate. Less powerful, but shared only within a limited area.

Features	Cloud servers	Edge servers and devices
Latency	High. As large volumes of data is exchanged in public networks, the latency can be high.	Low. The proximity of edge servers/devices to data sources requires less bandwidth, therefore improving the latency.
Storage capacity	High. Large volumes of data can be processed and scaled accordingly.	Low. Limited availability of resources.
Management capability	High. Easier to manage due to centralization.	Low. Harder to manage due to distribution.
Advantages	Accessibility Scalability Easier to manage	High responsiveness Low latency Security

*Table 2: Comparison of key features in cloud and edge computing
Own elaboration based on [19]*

Edge computing devices are becoming increasingly efficient in their computational capabilities. Because of growing demand, numerous manufacturers have released their own versions of edge computing devices to act as nodes in the edge computing environment. Among the most common edge hardware are NVIDIA Jetson Nano, Raspberry Pi 4, Intel Neural Compute Stick, ASUS Tinker Edge R and Google Coral Dev Board. In this thesis, the three first-mentioned edge devices will be analyzed in more detail.

3.3 Computer vision and image recognition

Deep learning is regarded as the latest technology in image classification and target detection, which form a crucial part of autonomous driving. According to [16], image classification and object detection are basic computer vision tasks that are required in specific fields, such as video surveillance, object counting, and vehicle detection. This data comes naturally from cameras located at the edge of the network, and some commercial cameras have built-in deep learning capabilities. Real-time reasoning in computer vision is usually measured by the frame rate which can reach the camera's frame rate, usually 30 to 60 frames per second.

The need for edge computing in computer vision tasks is stimulated by various potential issues related to the technology. [16] points out that uploading camera data to the cloud involves certain privacy issues, particularly if the camera frame contains sensitive information, such as human faces or private documents. Scalability is mentioned as another reason why edge computing is useful in computer vision tasks — if many cameras load large streams of video, the upstream bandwidth to the cloud server can become a bottleneck. To ensure traffic safety relying increasingly on self-driving technology, such risks must be minimized before autonomous driving can be introduced on a larger scale.

4 Software development

This section focuses on the software used for the project. The different libraries and functions present in the used software are described in detail as follows.

4.1 Python

Python is a general-purpose, high-level computer programming language which was created based on a dynamic type of system and an emphasis on readability and rapid prototyping [22]. It combines remarkable power with clear syntax and provides interfaces to many system calls and libraries as well as to various window systems, and is extensible in C or C++ or for applications requiring a programmable interface [22].

As a programming language, Python is highly versatile, user-friendly and applicable to many different classes of problems. Some of the areas covered by its extensive standard library include string processing (regular expressions, Unicode, calculating differences between files), internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI programming), software engineering (unit testing, logging, profiling, parsing Python code), and operating system interfaces (system calls, filesystems, TCP/IP sockets) [22].

4.2 OpenCV

OpenCV (Open-Source Computer Vision Library) is an open-source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications. The library consists of more than 2,500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms.

These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality [23].

4.3 TensorFlow and Convolutional Neural Network

TensorFlow is an end-to-end open-source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that allows researchers to push the state-of-the-art in ML and developers easily build and deploy ML powered applications [24].

Convolutional Neural Network, also known as CNN, is a common deep learning method often used in complex computer vision and image recognition applications. It can be run as a part of TensorFlow for the purpose of processing and classifying visual images in a highly accurate manner [25].

Before the advent of CNN, image processing was a very tedious process with a large amount of data. An image is made up of pixels, each of which is made up of colors. To demonstrate the complexity of processing a singular image of about 500×500 pixels, as many as $500 \times 500 \times 3$ parameters can be found, because every pixel requires parameters to indicate color information [25]. CNN has the ability to pack large parameters into a compact form, and at the same time preserve the features of an image. It should be noted that a typical CNN is not just a three-layer structure as mentioned above, but a multi-layer structure such as the LeNet-5 network illustrated in Figure 4.

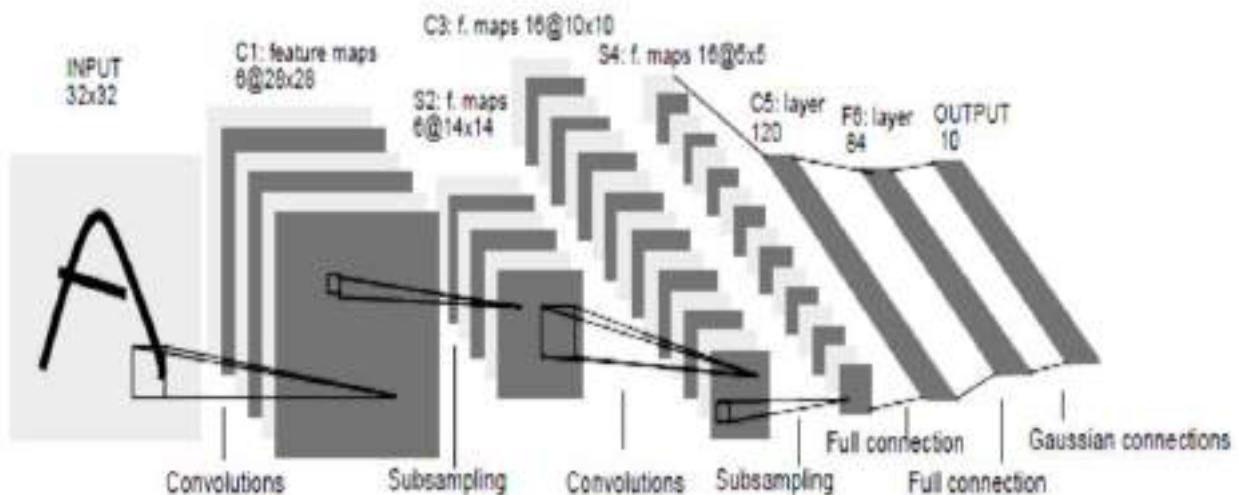


Figure 4: The LeNet5 architecture [26]

5 Programming Content

While driving on the road, self-driving vehicles use image recognition to identify various objects, such as pedestrians, vehicles, roads, and traffic signs. In this project, two of these parts, driving paths and traffic signs, are taken to observe how Python accomplishes the recognition process through the captured images. The procedure followed for this alongside the coding are presented in the following sections.

5.1 Lane detection

The lane detection function is incorporated in two files. The first one, named “lane_detection”, contains the function files required in the program. The other file, named “lane_detection_main”, is the file used to run the program. The steps of programming the lane detection function is visualized as follows.

5.1.1 lane_detection

The code for preprocessing the input images is presented in Figure 5, followed by a more detailed description of the steps taken.

```
1 import cv2
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 def process_image(image):
6     gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
7     blur_image = cv2.GaussianBlur(gray_image, (3, 3), 0)
8     Canny_image = cv2.Canny(blur_image, 10, 100)
9     return Canny_image
10
11 def mask_image(image):
12     bottom_padding = 10 # front bumper compression
13     height = image.shape[0]
14     width = image.shape[1]
15     bottom_left = (0, height-bottom_padding)
16     bottom_right = (width, height-bottom_padding)
17     top_right = (width//5, height//5)
18     top_left = (width//5, height*2/5)
19
20     vertices = np.array([bottom_left, bottom_right, top_right, top_left], dtype=np.int32)
21     mask = np.zeros_like(image)
22     cv2.fillPoly(mask, vertices, 255)
23     # cv2.imshow('mask', mask)
24     masked_image = cv2.bitwise_and(image, mask)
25     return masked_image
```

Figure 5: Programming for pre-processing input images

1. Importing libraries. As Python does not contain all the libraries, a third-party library must be imported first. This includes NumPy and matplotlib(.pyplot).
2. Pre-processing of input images. This procedure consists of graying, blurring and cannying the imported image as demonstrated in Figures 6, 7 and 8.

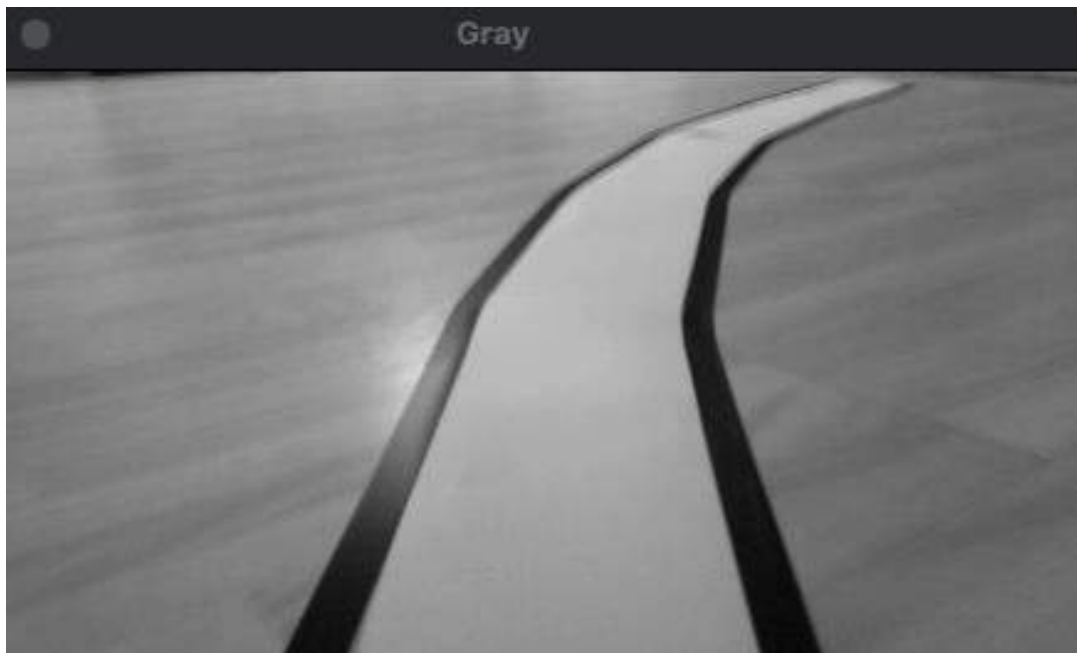


Figure 6: Graying the image

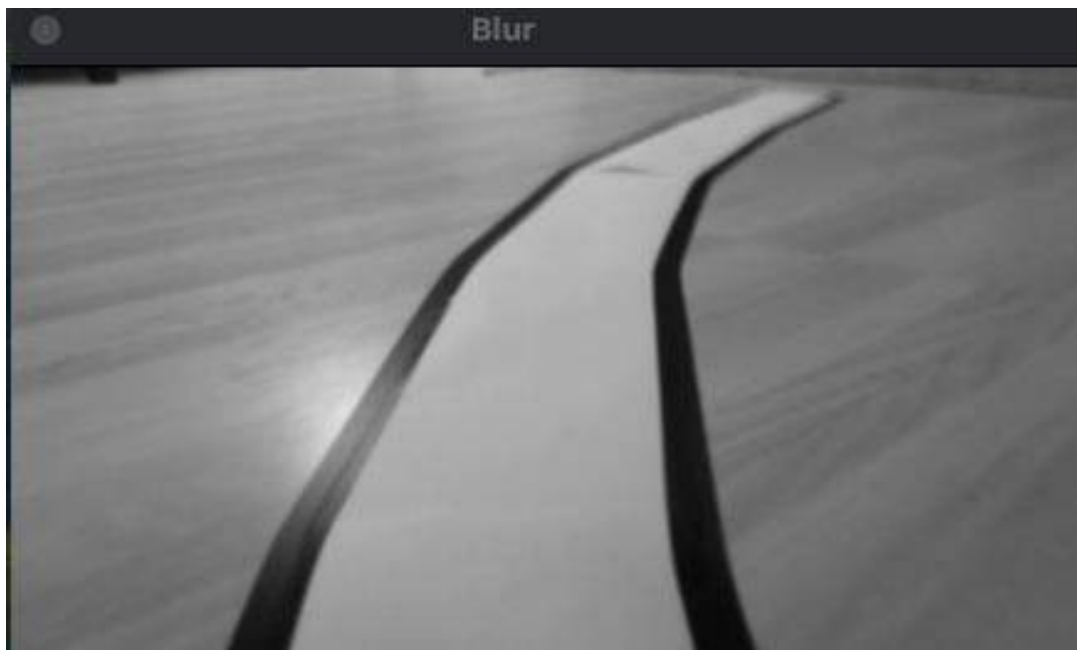


Figure 7: Blurring the image

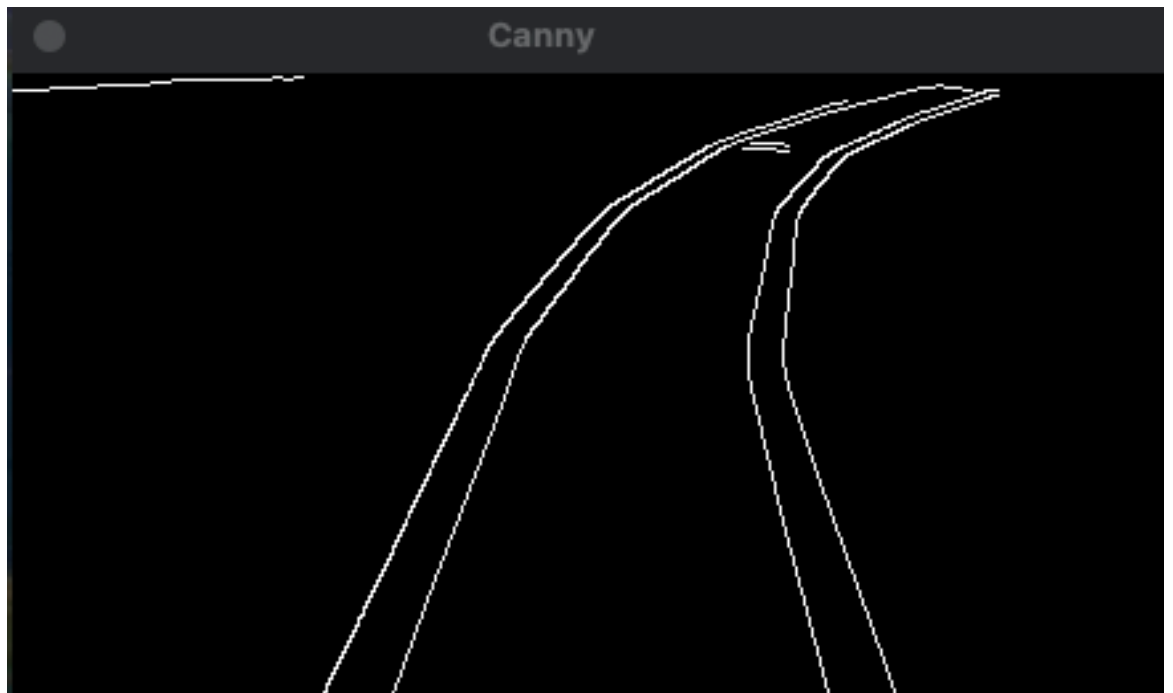


Figure 8: Cannying the image

3. Overwriting unwanted areas and keeping the areas of interest in image analysis.

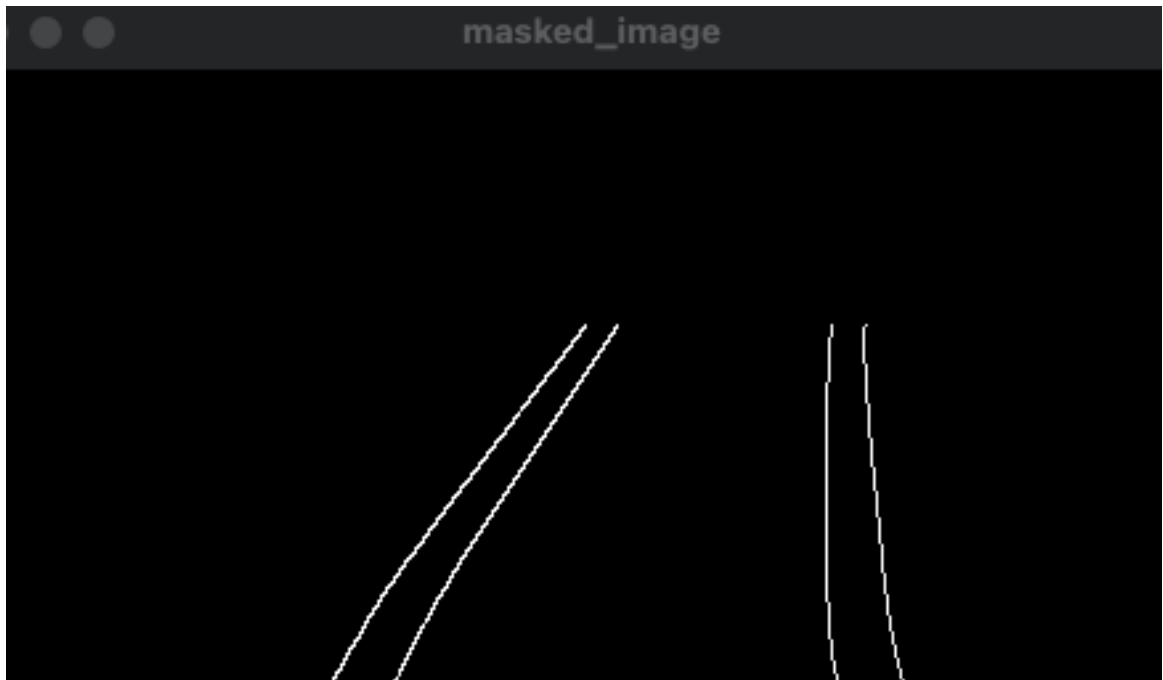


Figure 9: The area of interest

Based on the results shown in Figure 9, it can be observed that the road is deformed. This is not conducive to analyzing the results, so the resulting image needs to be processed further.

As the next step, image warping is performed through programming, the code of which is demonstrated in Figure 10.

```
def birdspoint_transform(img):
    leftupperpoint = (100, 80)
    leftlowerpoint = (20, 240)
    rightupperpoint = (100, 80)
    rightlowerpoint = (400, 240)

    src = np.float32([leftupperpoint, leftlowerpoint, rightupperpoint, rightlowerpoint])
    dst = np.float32([100, 0], [100, 300], [300, 0], [400, 300])

    img_size = img.shape[:2]
    M = cv2.getPerspectiveTransform(src, dst)
    M_inv = cv2.getPerspectiveTransform(dst, src)
    warped = cv2.warpPerspective(img, M, img_size, flags=cv2.INTER_LINEAR)

    return M, M_inv, warped
```

Figure 10: Programming for image warping

4. The area of interest in the image is warped as elaborated in Figure 11 below.



Figure 11: Warping area of interest
Own elaboration based on [26]

Warping the image is crucial for the next step, because when the vehicle is driving on the road, the low position of the camera results in the lane lines appearing to be intersecting at a distance when in reality, the lane lines are indeed parallel. The resulting image must therefore be processed in order to generate a “bird’s-eye view” of the image.


```

def find_line_pixels(image):
    # Take a histogram of the bottom half of the image
    histogram = np.sum(image[image.shape[0] // 2 : , :], axis=0)

    out_img = cv.cvtColor(image, image=image) + 255, cvtType='uint8')
    window_img = cv.cvtColor(out_img)

    # Find the peak in the histogram
    window = np.sum(histogram.shape[0] // 2)
    lefts_peak = np.argmax(histogram[window:])
    rights_peak = np.argmax(histogram[:window]) + window

    # Find the line parameters
    window_height = np.int(image.shape[0] * window)
    nonzero = image.nonzero()
    nonzero_y = np.array(nonzero[0])
    nonzero_x = np.array(nonzero[1])
    lefts_current = lefts_peak
    rights_current = rights_peak
    margin = 10
    width = 10
    left_lane_inds = []
    right_lane_inds = []

    for window in range(window_height):
        # Identify window boundaries in x and y (left and right)
        win_y_low = image.shape[0] - (window + 1) * window_height
        win_y_high = image.shape[0] - window * window_height
        win_xleft_low = lefts_current - margin
        win_xleft_high = lefts_current + margin
        win_xright_low = rights_current - margin
        win_xright_high = rights_current + margin
        cv.rectangle(out_img, (win_xleft_low, win_y_low), (win_xleft_high, win_y_high), (0, 255, 0), 2)
        cv.rectangle(out_img, (win_xright_low, win_y_low), (win_xright_high, win_y_high), (0, 255, 0), 2)
        # Identify the nonzero pixels in x and y within the window
        good_left_inds = ((nonzero_y >= win_y_low) & (nonzero_y < win_y_high) &
                          (nonzero_x >= win_xleft_low) & (nonzero_x < win_xleft_high)).nonzero()[0]
        good_right_inds = ((nonzero_y >= win_y_low) & (nonzero_y < win_y_high) &
                          (nonzero_x >= win_xright_low) & (nonzero_x < win_xright_high)).nonzero()[0]
        # Append these indices to the lists
        left_lane_inds.append(good_left_inds)
        right_lane_inds.append(good_right_inds)
        # If you have = margin pixels, center your window on their mean position
        if len(good_left_inds) > width:
            lefts_current = np.int(np.mean(nonzero_x[good_left_inds]))
        if len(good_right_inds) > width:
            rights_current = np.int(np.mean(nonzero_x[good_right_inds]))

    left_lane_inds = np.concatenate(left_lane_inds)
    right_lane_inds = np.concatenate(right_lane_inds)

    lefts = nonzero_x[left_lane_inds]
    lefts = nonzero_y[left_lane_inds]
    rights = nonzero_x[right_lane_inds]
    rights = nonzero_y[right_lane_inds]

    left_fit = np.polyfit(lefts, lefts, 2)
    right_fit = np.polyfit(rights, rights, 2)

    ploty = np.linspace(0, image.shape[0]-1, image.shape[0])
    out_img[nonzero_y[left_lane_inds], nonzero_x[left_lane_inds]] = 255, 0, 0
    out_img[nonzero_y[right_lane_inds], nonzero_x[right_lane_inds]] = 0, 255, 0
    left_fitx = left_fit[0]*ploty+left_fit[1]*ploty+left_fit[2]
    right_fitx = right_fit[0]*ploty+right_fit[1]*ploty+right_fit[2]

    left_line_window1 = np.array([np.transpose(np.vstack([left_fitx - margin, ploty]))],
                                np.array([np.transpose(np.vstack([left_fitx + margin, ploty]))])])
    left_line_pts = np.hstack([left_line_window1, left_line_window2])
    right_line_window1 = np.array([np.transpose(np.vstack([right_fitx - margin, ploty]))],
                                np.array([np.transpose(np.vstack([right_fitx + margin, ploty]))])])
    right_line_pts = np.hstack([right_line_window1, right_line_window2])

    cv.fillPoly(window_img, np.int_([left_line_pts]), (0, 255, 0))
    cv.fillPoly(window_img, np.int_([right_line_pts]), (0, 255, 0))
    result = cv2.addWeighted(out_img, 1, window_img, 0.5, 0)

```

Figure 12: Programming for histogram to mark lanes

5. Determining the location of the lane by using a histogram of colors to find the starting points of the left, right and center lanes within the captured image. In order to ensure that the lane markings are detected accurately, the first step is to determine the maximum area of probability. This can be done through the use of bird's-eye view histogram in which the peaks for the right and left lanes are clearly visible. Firstly, four points from the source image and four points from the target image are specified, and then the function `cv2.getPerspectiveTransform()` is performed. When transforming an image with the `cv2.warpPerspective()` function, a 3x3 transformation matrix is computed.

A neural network can be taught detect multiple objects within an image by using a computational technique called sliding windows. As the name implies, the sliding window algorithm slides a window over some input array and applies an operation to the content under the window. The network then relies on a metric called intersection-over-union to pick the best box and non-max suppression to discard boxes that are less accurate. [29]

Through the use of the sliding window search approach, different areas of the frame can be detected, pinpointing the regions with the highest pixel density and thus resulting in the correct identification of lanes. As shown in Figure 13, the starting point of pixels shows the most likely location of a specific lane line. Based on this, sliding windows running over the pixel coordinates are used to determine the position and direction of lanes. The coefficients of the line curves on both right and left lanes are computed by using the Numpy Polyfit method.

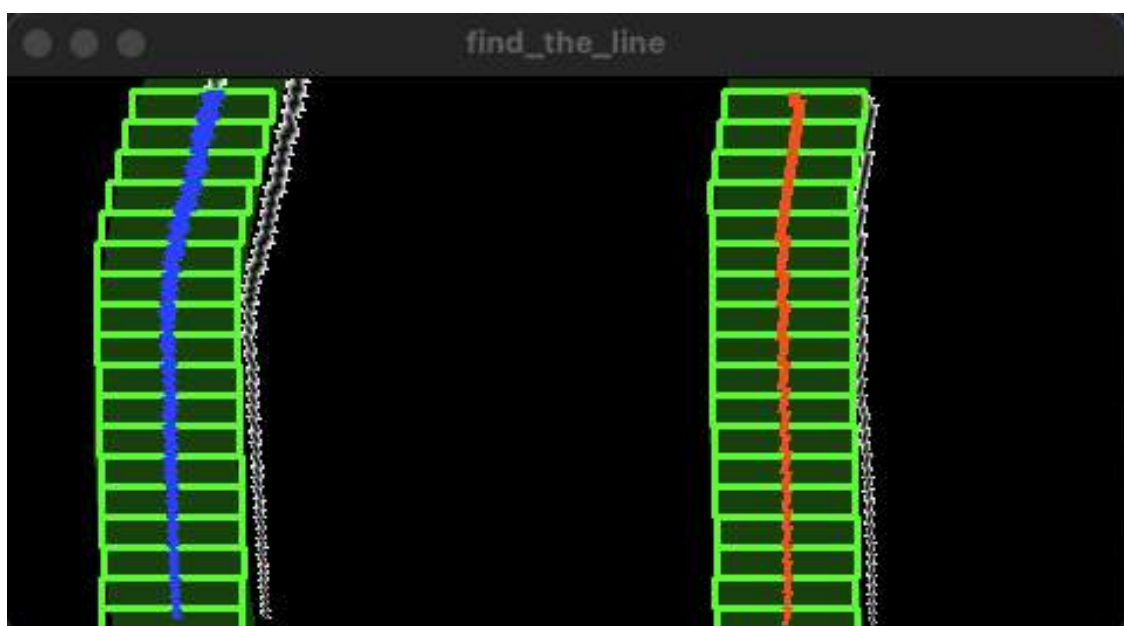


Figure 13: Road line recognition [27]

6. As demonstrated in Figure 14, the pixel value corresponding to the column with the lane line is larger, and two peaks appear. These can be used to locate the center of the lane line in the X-direction, which then serves as the starting point for finding the location of the lane line in the image through the sliding window.

```

13 plt.plot(left_fit, xdata, color='yellow')
14 plt.plot(right_fit, xdata, color='yellow')
15
16 ret = {}
17 ret['left_fit'] = left_fit
18 ret['right_fit'] = right_fit
19 ret['center'] = centerx
20 ret['center'] = centerx
21 ret['xdata'] = xdata
22 ret['left_lane_line'] = left_lane_line
23 ret['right_lane_line'] = right_lane_line
24
25 return result, ret

```

Figure 14: Programming for parameter return values

7. Establishing the return values of several required parameters in the form of a dictionary, as shown in Figure 15.

```

1 def calculate_parameters(left_fit, right_fit, xdata):
2     xdata = np.linspace(0, 1000, num=1000, endpoint=True)
3     left_fit = left_fit[0:1000]
4     right_fit = right_fit[0:1000]
5     centerx = 0
6     centerx = 0
7     centerx = 0
8     left_fit = np.polyfit(xdata, left_fit, 2)
9     right_fit = np.polyfit(xdata, right_fit, 2)
10    left_lane_line = (1 + (left_fit[0]*xdata + left_fit[1])) / 2
11    right_lane_line = (1 + (right_fit[0]*xdata + right_fit[1])) / 2
12
13    centerx = (left_lane_line + right_lane_line) / 2
14    lane_width = np.abs(left_lane_line - right_lane_line)
15    lane_center_line = (1 + (left_lane_line + right_lane_line)) / 2
16    left_lane_line = (1 + (left_lane_line - right_lane_line)) / 2
17    right_lane_line = (1 + (right_lane_line - left_lane_line)) / 2
18
19    return centerx, lane_width, lane_center_line, left_lane_line, right_lane_line

```

Figure 15: Programming for orientation and position calculations

8. Calculating the orientation and position of the vehicle relative to the center of the lane, as demonstrated in Figure 16.

```

109 def draw_lane(image, binary_warped, fitv, left_fit, right_fit):
110     ploty = np.linspace(0, binary_warped.shape[0]-1, binary_warped.shape[0])
111     left_fitx = left_fit[0]*ploty**2 + left_fit[1]*ploty + left_fit[2]
112     right_fitx = right_fit[0]*ploty**2 + right_fit[1]*ploty + right_fit[2]
113     warp_zero = np.zeros_like(binary_warped).astype(np.uint8)
114     color_warp = np.dstack((warp_zero, warp_zero, warp_zero))
115
116     pts_left = np.array([np.transpose(np.vstack([left_fitx, ploty]))])
117     pts_right = np.array([np.flipud(np.transpose(np.vstack([right_fitx, ploty])))])
118     pts = np.hstack((pts_left, pts_right))
119
120     cv2.fillPoly(color_warp, np.int_([pts]), (0,255, 0))
121
122     newwarp = cv2.warpPerspective(image, fitv, (image.shape[1], image.shape[0]))
123     result = cv2.addWeighted(image, 1, newwarp, 0.5, 0)
124     return result

```

Figure 16: Programming for lane line integration

9. Integrating the calculated lane lines into the original image.

5.1.2 lane_detection_main

All the functions must be run separately and then together to ensure correct lane detection. The programming used is displayed in Figure 17.

[illegible]

Figure 17: Running functions for verification

1. Importing the required libraries.
2. Running each function individually to check that they deliver the desired results, and then running all the functions together to make sure that the lanes are detected correctly.

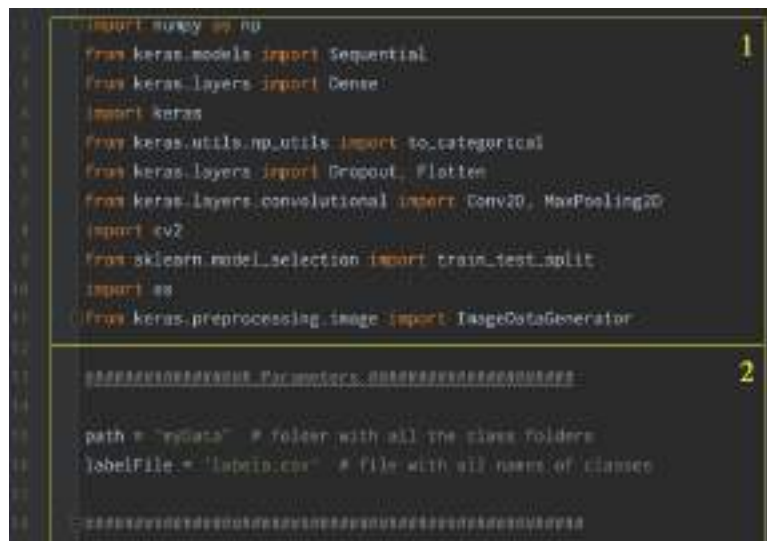
5.2 Identification of traffic signs

This part of the program is divided into three files. Firstly, “traffic_sign_training” is used for training the traffic sign data, and throughout the training process, the program is trained, verified, and tested by using pictures of the corresponding traffic signs. To simulate the versatility of real-life traffic scenarios, the training file used for the traffic sign identification function consists of 43 traffic signs commonly used in

traffic. Among others, these include “30 km/h”, “120 km/h”, “road works” and “vehicles over 3.5 metric tons prohibited”. Secondly, “traffic_sign_module” includes functions which are needed in the program. Finally, get_traffic_sign is the executable file.

5.2.1 traffic_sign_training

The programming used for traffic sign training is presented in Figure 18, followed by a brief explanation of the steps taken.



```
1 import numpy as np
2 from keras.models import Sequential
3 from keras.layers import Dense
4 import keras
5 from keras.utils.np_utils import to_categorical
6 from keras.layers import Dropout, Flatten
7 from keras.layers.convolutional import Conv2D, MaxPooling2D
8 import cv2
9 from sklearn.model_selection import train_test_split
10 import os
11 from keras.preprocessing.image import ImageDataGenerator
12
13 ##### Parameters #####
14
15 path = "xyData" # folder with all the class folders
16 labelFile = "labels.csv" # file with all names of classes
17
18 #####
```

Figure 18: Programming for path selection

1. Importing the required libraries.
2. Training traffic signs. The training part consists of two kinds of files. The first one includes the training images for each of the traffic signs. The other file, called ‘labels.csv’, includes the names and numbers of the traffic signs. These variables can be used to read and access data and labels from the specified directory and file within the code.

```

count = 0
images = []
classNo = []
myList = os.listdir(path)
print(os.listdir(path))
print("Total Classes Detected:" + len(myList))
noOfClasses = len(myList)
print("Importing Classes.....")
for x in range(0, 43):
    myPicList = os.listdir(path + "/" + str(count))
    for y in myPicList:
        curImg = cv2.imread(path + "/" + str(count) + "/" + y)
        images.append(curImg)
        classNo.append(count)
    print(count, end=" ")
    count += 1
print(" ")
images = np.array(images)
classNo = np.array(classNo)
data = np.array(images)
data = np.array(data).reshape(-1, 32, 32, 3)

```

Figure 19: Programming for image import

3. Importing all the traffic signs images used as shown in Figure 19. This part prepares images and labels for learning steps.

```

X_train, X_test, y_train, y_test = train_test_split(images, classNo, test_size=0.2,
                                                    Y_testing_test=1)
X_train, X_validation, y_train, y_validation = train_test_split(X_train, y_train, test_size=0.1)

print("Data Shape")
print("Train", X_train.shape, y_train.shape)
print("Validation", X_validation.shape, y_validation.shape)
print("Test", X_test.shape, y_test.shape)

def grayscale(img):
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    return img

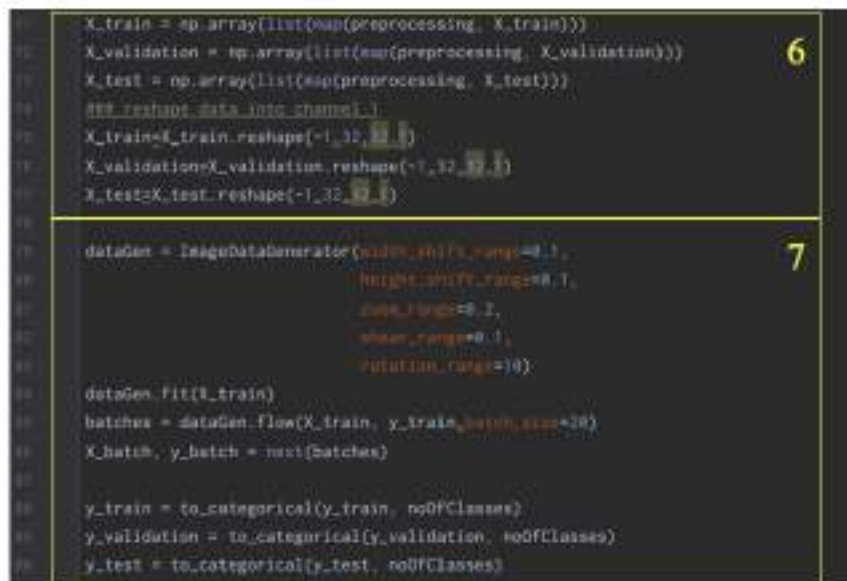
def equalizing(img):
    img = cv2.equalizeHist(img)
    return img

def preprocessing(img):
    img = grayscale(img)
    img = equalizing(img)
    img = img / 255.0 # image normalization
    return img

```

Figure 20: Programming for splitting image data and preprocessing image

4. Splitting all the image data into three parts used for training, validation, and the rest for testing, as demonstrated in Figure 20.
5. Preprocessing the image, including grayscale performance and processing equalization. These preprocessing functions can be used to improve the quality and consistency of images. In particular, they help to improve the quality and readability of the images, thus improving model performance.



```

1 X_train = np.array(list(map(preprocessing, X_train)))
2 X_validation = np.array(list(map(preprocessing, X_validation)))
3 X_test = np.array(list(map(preprocessing, X_test)))
4 ## reshape data into channel 1
5 X_train=X_train.reshape(-1,32,32,3)
6 X_validation=X_validation.reshape(-1,32,32,3)
7 X_test=X_test.reshape(-1,32,32,3)
8
9
10 dataGen = ImageDataGenerator(rotation_range=40,
11                             height_shift_range=0.1,
12                             zoom_range=0.2,
13                             shear_range=0.1,
14                             width_shift_range=0.1,
15                             validation_split=0.0)
16
17 dataGen.fit(X_train)
18 batches = dataGen.flow(X_train, y_train, batch_size=32)
19 X_batch, y_batch = next(batches)
20
21 y_train = to_categorical(y_train, noOfClasses)
22 y_validation = to_categorical(y_validation, noOfClasses)
23 y_test = to_categorical(y_test, noOfClasses)

```

Figure 21: Programming for preprocessing and augmenting images

6. Preprocessing all the images, iterating them, and then changing reshaping images into a 4D array. For now, the format of data is suitable for input into a model that expects 32x32 grayscale images. This preprocessing is typically performed to ensure the consistency and compatibility of the input data with the model's architecture and requirements.
7. Augmenting all the images to make them more generic. Since there are too few traffic sign samples of certain categories in the training set, and to increase the diversity of training samples so that the network learns more stable and essential features, some common enhancements to the training images must be performed. This can be done by using ImageDataGenerator.


```

11 def seq_model():
12     no_of_filters = 10
13     size_of_filter = (3, 3)
14     size_of_filter2 = (4, 4)
15     size_of_pool = (2, 2)
16     no_of_nodes = 500
17     model = Sequential()
18     model.add(Conv2D(no_of_filters, size_of_filter, input_shape=(12, 12, 3),
19         activation='relu'))
20     model.add(Conv2D(no_of_filters, size_of_filter, activation='relu'))
21     model.add(MaxPooling2D(pool_size=size_of_pool))
22
23     model.add(Conv2D(no_of_filters // 2, size_of_filter2, activation='relu'))
24     model.add(Conv2D(no_of_filters // 2, size_of_filter2, activation='relu'))
25     model.add(MaxPooling2D(pool_size=size_of_pool))
26     model.add(Dropout(0.5))
27
28     model.add(Flatten())
29     model.add(Dense(no_of_nodes, activation='relu'))
30     model.add(Dropout(0.5))
31     model.add(Dense(no_of_classes, activation='softmax'))
32     opt = keras.optimizers.Adam(lr=0.001)
33     model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
34     return model
35
36 model = seq_model()
37 print(model.summary())

```

Figure 22: Programming for convolution neural network model

8. Implementing a convolutional neural network model, as shown in Figure 22. The model has two convolutional blocks, and each with two convolutional layers followed by max-pooling, which helps in learning hierarchical features from the input images. The fully connected layers at the end make the final predictions for the classes. The dropout layers are used for regularization to prevent overfitting.

```

12 batch_size_val=50
13 steps_per_epoch_val=400
14 epochs_val=10
15 #train the model
16 history = model.fit_generator(dataGen.flow(X_train, y_train, batch_size=batch_size_val),
17     steps_per_epoch=steps_per_epoch_val, epochs=epochs_val,
18     validation_data=(X_validation, y_validation), shuffle=1)
19
20 score = model.evaluate(X_test, y_test, verbose=0)
21 print('Test Score: ', score[0])
22 print('Test Accuracy: ', score[1])

```

Figure 23: Programming for training

9. Conducting the training process by using epochs as 10 and giving 50 samples at a time, as demonstrated in Figure 23. The code trains the model for the specified number of epochs while using the training data generated by the data generator. The validation data is used to monitor the model's performance during training. The purpose of the training process is to optimize the model parameters.

10. Printing the training result and saving the result as “traffic_sign_model”. By evaluating the model on the test dataset, it can be assessed how well it generalizes to new, unseen data. Saving the model permits reusing it in various applications, such as later controlling of the donkey car.

5.2.2 traffic_sign_module

The image is preprocessed for specified color removal, in order to facilitate the reading of the traffic signs. The programming is presented in Figure 24.

```
import numpy as np
import cv2

def preprocess_img(imgBGR, erode_dilate=True):
    rows, cols, _ = imgBGR.shape
    imgHSV = cv2.cvtColor(imgBGR, cv2.COLOR_BGR2HSV)
    Rmin = np.array([10, 43, 46])
    Rmax = np.array([124, 255, 255])
    img_Rbin = cv2.inRange(imgHSV, Rmin, Rmax)

    Bmin1 = np.array([0, 43, 46])
    Bmax1 = np.array([5, 255, 255])
    img_Rbin1 = cv2.inRange(imgHSV, Bmin1, Bmax1)

    Bmin2 = np.array([134, 43, 46])
    Bmax2 = np.array([144, 255, 255])
    img_Rbin2 = cv2.inRange(imgHSV, Bmin2, Bmax2)
    img_Rbin = np.maximum(img_Rbin1, img_Rbin2)
    img_bin = np.maximum(img_Rbin, img_Rbin)
    if erode_dilate is True:
        kernelErosion = np.ones((3, 3), np.uint8)
        kernelDilation = np.ones((5, 5), np.uint8)
        img_bin = cv2.erode(img_bin, kernelErosion, iterations=2)
        img_bin = cv2.dilate(img_bin, kernelDilation, iterations=2)
    return img_bin
```

Figure 24: Programming for specified color removal

1. Importing libraries.
2. This preprocessing function is tailored for detecting blue and red signs in images, and removing the specified color to read the traffic sign more clearly. The results obtained are exemplified by “STOP” and “60km/h” as shown in Figure 25.

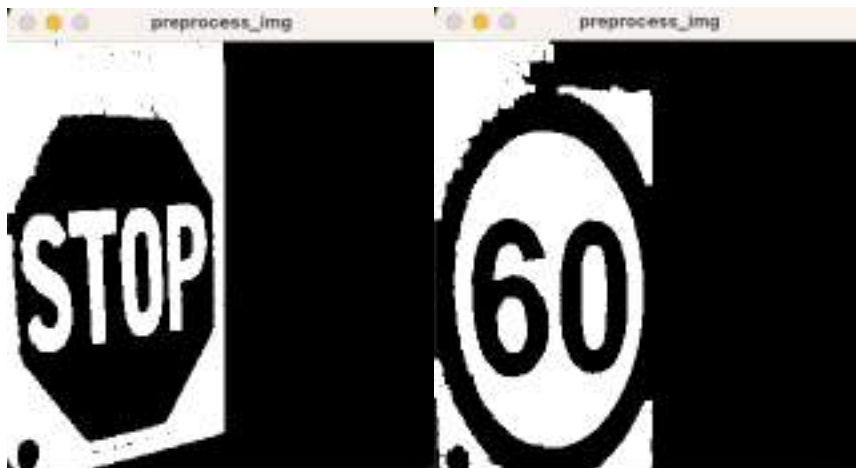


Figure 25: Color removal for better traffic sign readability

```

def contour_detection(img_bin, min_area, max_area, min_x, min_y, max_x, max_y):
    rects = []
    contours = cv2.findContours(img_bin.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if len(contours) == 0:
        return rects

    max_area = img_bin.shape[0] * img_bin.shape[1] if max_area < 0 else max_area
    for contour in contours:
        area = cv2.contourArea(contour)
        if area > min_area and area < max_area:
            x, y, w, h = cv2.boundingRect(contour)
            if 1 < w / h < 40 and 1 < h / w < 40:
                rects.append((x, y, w, h))
    return rects

```

Figure 26: Programming for contour detection

3. This step is used to identify and extract specific regions of interest. Detecting the contour and returning the contour as a rectangular box, as displayed by Figure 26.

```

def preprocessing(img):
    img = grayscale(img)
    img = equalize(img)
    img = img / 255
    return img

def getCalssName(classNo):
    if classNo == 0:
        return 'Speed Limit 20 km/h'
    elif classNo == 1:
        return 'Speed Limit 30 km/h'
    elif classNo == 2:
        return 'Speed Limit 50 km/h'
    elif classNo == 3:
        return 'Speed Limit 60 km/h'
    elif classNo == 4:
        return 'Speed Limit 70 km/h'
    elif classNo == 5:
        return 'Speed Limit 80 km/h'
    elif classNo == 6:

```

Figure 27: Programming for image preprocessing and traffic sign number listing

4. Preprocessing the image as demonstrated in Figure 27, including the running of grayscale and equalizing functions. The overall effect of these preprocessing steps is to convert the input image into a preprocessed grayscale image with enhanced contrast and normalized pixel values. This preprocessed image is often more suitable for tasks like object detection and classification.
5. Listing the traffic sign numbers based on the feedback value. The step turns the image into the corresponding labeling, which is fed back into the result.

5.2.3 get_traffic_sign

```

def get_traffic_sign(img):
    # Load the image
    img = cv2.imread(img_path)
    # Convert the image to grayscale
    img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Blur the image
    img = cv2.GaussianBlur(img, (5, 5), 0)
    # Threshold the image
    img = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY)[1]
    # Find contours
    contours, _ = cv2.findContours(img, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
    # Sort contours by area
    contours = sorted(contours, key=cv2.contourArea, reverse=True)
    # Get the largest contour
    cnt = contours[0]
    # Get the bounding box of the largest contour
    x, y, w, h = cv2.boundingRect(cnt)
    # Crop the image
    img = img[y:y+h, x:x+w]
    # Resize the image
    img = cv2.resize(img, (32, 32))
    # Predict the sign type
    prob = model.predict(img)
    # Get the sign type
    sign_type = prob.argmax()
    # Get the sign name
    sign_name = sign_names[sign_type]
    # Return the sign name
    return sign_name

```

Figure 28: Programming for traffic sign number definition

1. Determining the number of the traffic sign from the image and returning this value, as shown in Figure 28.
2. Predicting the type of traffic signs in the image based on the training results and adding text to the result if it is greater than the set percentage.

6 Hardware Implementation

For the implementation part of this thesis, a donkey car equipped with a camera for image recognition was used. To conduct a comparison of the image recognition performance in program execution, three types of hardware were selected: Raspberry Pi 4, Jetson Nano and Intel Neural Compute Stick 2. Prior to analysing the performance, these three selected hardware types alongside the donkey car setup are briefly introduced.

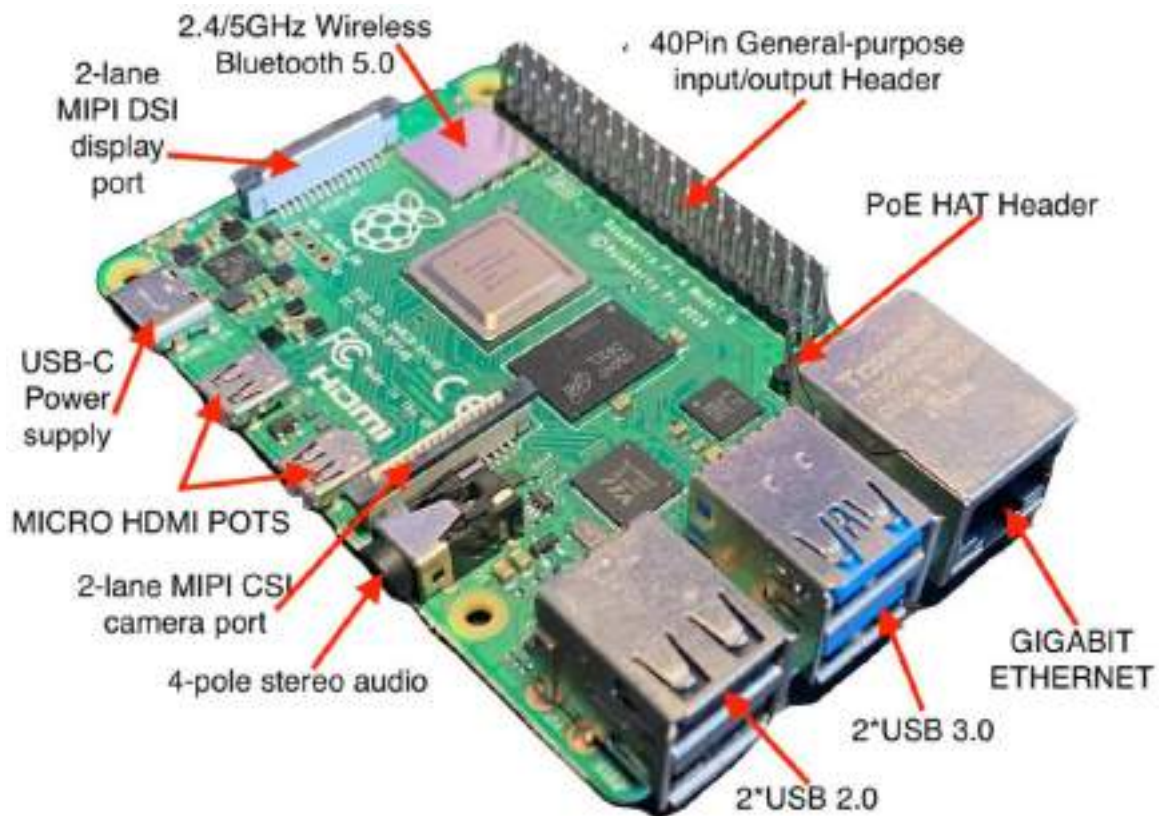
6.1 Raspberry Pi 4

Raspberry Pi, launched in 2012, is a series of small single-board computers (SBCs). It was developed in the United Kingdom by the Raspberry Pi Foundation in association with Broadcom [28]. It was originally created so that more people could afford to use computers.

The Raspberry Pi 4B is a "plug and play" type single board computer, which also serves as a great starting point for learning different kinds of AI projects. Based on the Linux operating system, the Raspberry Pi has access to a wide range of free software and tools for Linux [28]. Its user-friendliness makes it suitable also for people with less experience to learn programming languages such as Python. It can also do everything one would expect a desktop computer to do, from browsing the web and playing HD video to creating spreadsheets, word processing and playing games.

6.1.1 Interface of Raspberry Pi 4

Raspberry Pi 4 has numerous interfaces which can be used together with external devices, such as the keyboard and the monitor. The different parts of the Raspberry Pi 4 hardware are illustrated in more detail in Figure 29.



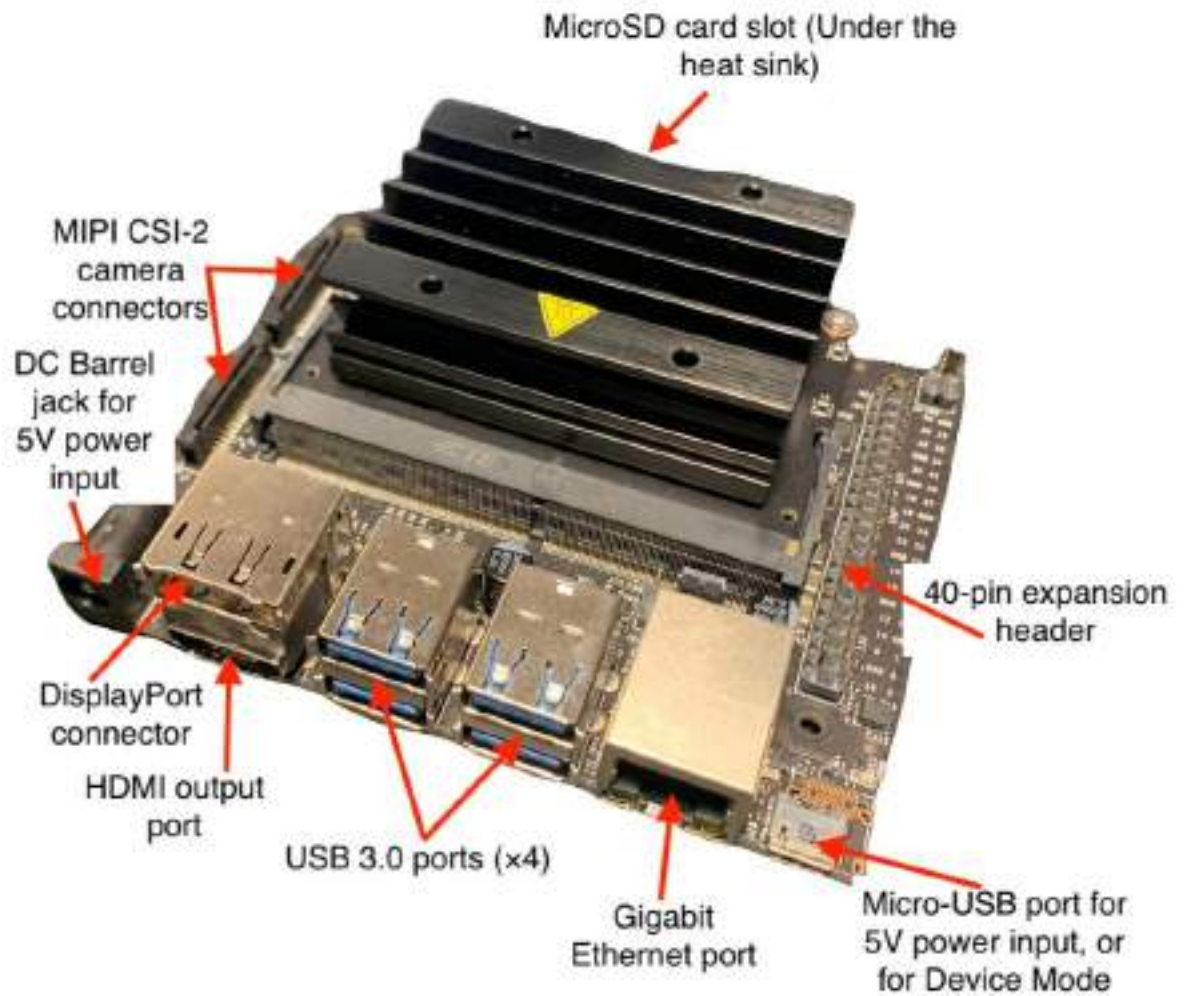
*Figure 29: The Raspberry Pi 4 hardware
Own elaboration*

6.2 Jetson Nano

The Jetson Nano development board is also a powerful small AI computer, which can be booted by simply inserting a microSD card with a system image. It has a built-in SOC system-on-chip that can parallelize neural networks, such as TensorFlow, PyTorch, Caffe/Caffe2, Keras and MXNet. These neural networks can be used for image classification, target detection, speech segmentation and intelligent analysis, as well as to build autonomous robots and complex AI systems [29].

6.2.1 Interface of Jetson Nano

Jetson Nano contains numerous interfaces. Just like in the case of Raspberry Pi, these can be used together with external devices, such as the keyboard and the monitor. The different parts of the Jetson Nano hardware are described in more detail in Figure 30 below.



*Figure 30: The Jetson Nano interface
Own elaboration*

6.3 Neural Compute Stick 2

Neural Compute Stick 2, or NCS2, is a small neural network training device in the shape of a USB stick, which can be plugged into the host computer through the USB port of other devices. The composition of NCS2 is demonstrated in Figure 31 below.



*Figure 31: The Neural Compute Stick 2 Interface
Own elaboration*

This device is often used in artificial programming at the edge where processing large amounts of data can become very labor-intensive. NCS2 functions as an accelerator, supporting the edge device CPU in running deep learning models in order to improve efficiency [30].

The NCS2 device is compatible with a variety of operating systems. It contains a built-in Intel Movidius Myriad X VPU vision processor dedicated to accelerated computing of neural networks [33]. The device supports TensorFlow, Caffe and other development frameworks.

6.3.1 Connecting Diagram

In this project, the camera, Raspberry Pi4 and a dual motor driver called L298N are linked to each other. Here, the L298N acts as a power module between the controller and the DC motor. It is used to simultaneously control both the speed and direction of the four DC motors in the autonomous vehicle constructed for the project.

In order to control the speed of the vehicle, the input voltage is altered through the use of a technique called Pulse Width Modulation (PWM). This technique allows for the average value of the input voltage to be changed according to the width of the pulses, so lower average voltage applied to the DC motor slows down the motor speed, whereas higher average voltage amplifies the motor speed. Besides speed, also the spinning of the DC motors can be controlled in the autonomous vehicle by switching the polarity of the input voltage. When the specific switches are closed

simultaneously, the polarity of the voltage is reversed, and as a result, the spinning direction of the DC motor changes.

After the camera captures the image, the Raspberry Pi4 processes it, and then transmits the result to the L298N to control the operation of the motor. The parts are connected to each other as illustrated in Figure 32.

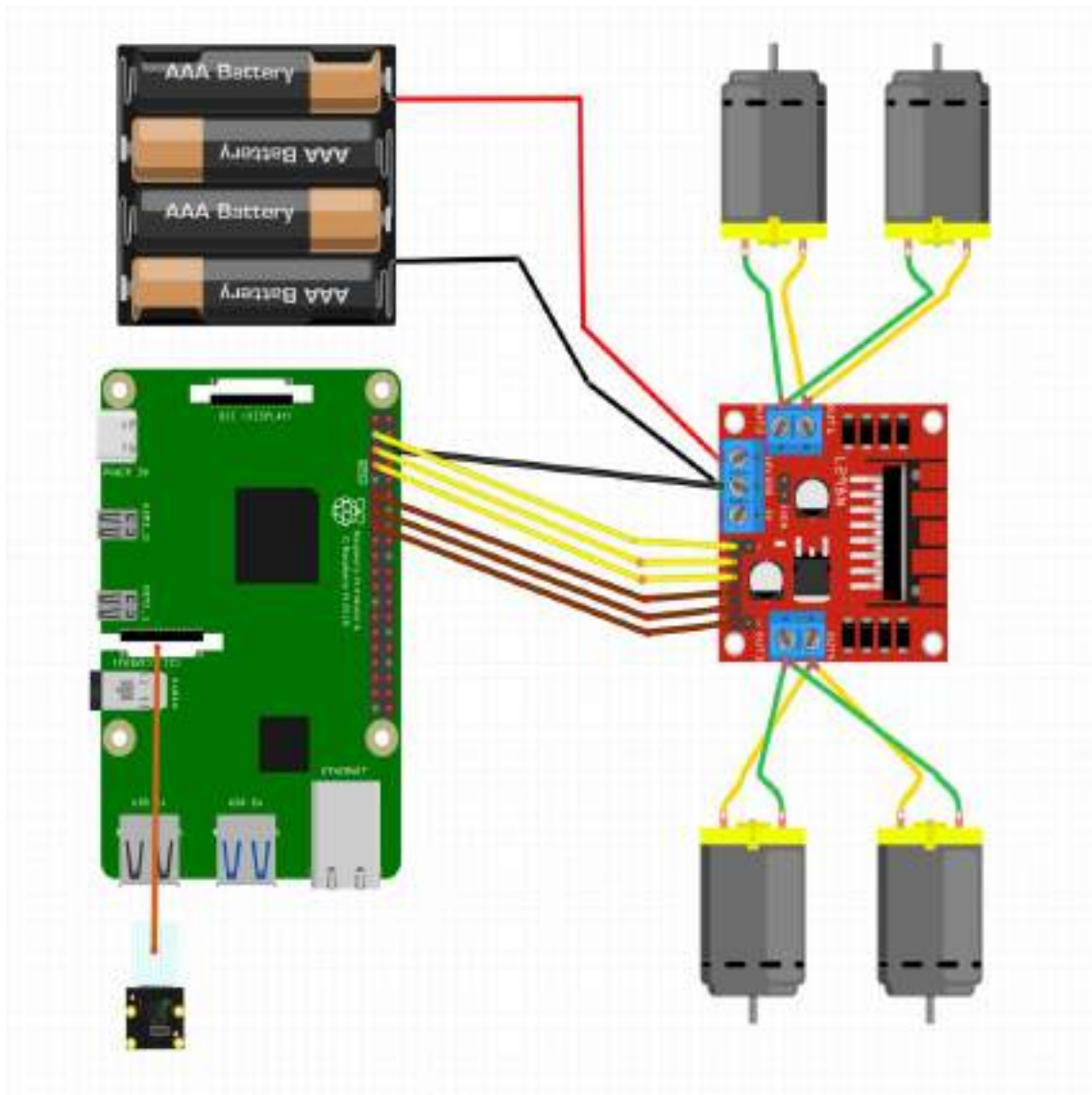


Figure 32: The donkey car connection diagram
Own elaboration

6.4 Hardware configuration comparison

Both Raspberry Pi and Jetson Nano are relatively scalable as hardware devices. The devices provide a wealth of interfaces, allowing users to connect many practical

devices, such as monitors, cameras, mice and keyboards and other external devices. This enables users to feel like using a computer in the traditional sense.

However, some differences can still be found in the design between the two hardware. Firstly, a significant difference in the hardware part is that Raspberry Pi 4 includes a wireless network module, whereas Jetson Nano does not. If the user wants to have a wireless network module in the Jetson Nano device, additional modules must be purchased, which will undoubtedly increase the total cost. If one does not want to buy additional wireless module function, the connection must be established through the network cable, greatly limiting the use of the location.

Some of the key differences in terms of features are gathered in Table 3 below. On a separate note, Neural Computer Stick 2 is not included in this table, because it is used as an accelerator device.

	Raspberry Pi 4	NVIDIA Jetson Nano
CPU	Quad-core ARM Cortex-A72 64-bit @1.5Ghz	Quad-core ARM Cortex-A72 64-bit @1.42Ghz
GPU	Broadcom VideoCore VI (32-bit)	NVIDA Maxwell w/ 128 CUDA cores @921Mhz
Memory	4GB LPDDR4	4GB LPDDR4
Networking	Gigabit Ethernet/ Wifi 802.11ac	Gigabit Ethernet/ M.2 Key E (for Wifi support)
Display	2* mirco-HDMI (up to 4Kp60)	HDMI 2.0 and eDP 1.4
USB	2* USB 3.0, 2*USB 2.0	2* USB 3.0, USB 2.0 Miceo-B
Other	40-PIN GPIO	40-PIN GPIO
Video Encode	H264(1080p30)	H.264/H.265(4Kp30)
Video Decode	H.265 (4Kp60), H.264 (1080p60)	H.264/H.265 (4Kp60, 2* 4Kp30)
Camera	MIPI CSI port	MIPI CSI port
Storage	Micro-SD	Micro-SD
Price	€51.86 Euro	€93.36 Euro

*Table 3: Comparison of hardware configuration
Own elaboration*

6.5 Installing the operating system

As there are significant differences in the processes of installing the operating system for the different edge devices, this section provides a more in-depth description and comparison for each of them.

6.5.1 Installation on Raspberry Pi 4

There are two ways to install the Raspberry Pi 4 system. It is officially recommended to use the Raspberry Pi imager, but an alternative option is to burn the image directly, which may be easier. Nevertheless, in order to avoid errors, for first-time use it is officially recommended to use an installation tool for Raspberry Pi 4, which is why this method was chosen for this thesis for the installation of the system. This was done as described in the following steps:

1. Download the Raspberry Pi imager from the official website. There are two versions available: one for Windows and one for MacOS. When opened, the software interface looks as shown in Figure 33.

In this software, four different operating systems are included in total, two of which do not include desktop environment while the other two do. One of them is the full version of Raspberry Pi OS, which includes Python 3.9 and in which some office software and programming applications are pre-installed. Another one is the Raspberry Pi Lite version, which includes Python 3.7. In this thesis, the Raspberry Pi Lite version is used, and the specific reasons for this decision will be described further on.



Figure 33: The Raspberry Pi installer

2. Select the correct operating system. In this case, Raspberry Pi OS Full (32-bit) was selected as when compared to Raspberry Pi OS Lite, this version includes more software functions, such as office software and education.



Figure 34: Raspberry Pi OS selection

3. Selecting storage SD card, after which the software installs the operating system automatically.
4. After installing the system, insert the SD card to the Raspberry Pi device. After following simple steps to setup, Raspberry Pi 4 is ready to use.

6.5.2 Problems on Raspberry Pi 4 and their solutions

As previously mentioned, Raspberry Pi includes two versions of Python: Version 3.7 and Version 3.9. Users must decide which version to install according to different application scenarios. In this thesis, OpenCV and TensorFlow have been used as application programming interfaces (API) for installation, but the installation process is not equal in the two Python versions.

Python 3.7 seemed to be compatible with both OpenCV and TensorFlow, and no major issues occurred during the steps of installation. However, when using Python 3.9, two major problems could be identified. Firstly, it turned out that installing TensorFlow was impossible for this version of Python. Secondly, when installing the OpenCV, constant issues occurred during the installation process. These issues as well as the solutions are presented as follows.

1. An error message appears during `sudo make -j1`: `c++: error: unrecognized command-line option '--param=ipcp-unit-growth=100000'; did you mean '--param=ipa-cp-unit-growth='?`.

Solution: Attempt to modify `build.make` and `flags.make` in the prompted directory, replacing all the `ipcp-unit-growth` inside with `ipa-cp-unit-growth`.


```
pi@raspberrypi: ~/opencv-4.0.0/build
File Edit Tabs Help
make[1]: *** [CMakeFiles/Makefile2:3237: 3rdparty/carotene/hal/carotene/CMakeFiles/carotene_objs.dir/all] Error 2
make: *** [Makefile:166: all] Error 2
pi@raspberrypi:~/opencv-4.0.0/build $ sudo make -j4
[ 2%] Built target libtiff
[ 4%] Built target libjasper
[ 7%] Built target libjpeg-turbo
[ 14%] Built target libwebp
[ 14%] Built target quirc
[ 14%] Building CXX object 3rdparty/carotene/hal/carotene/CMakeFiles/carotene_objs.dir/src/absdiff.cpp.o
c++: error: unrecognized command-line option '--param=ipcp-unit-growth=100000';
did you mean '--param=ipa-cp-unit-growth='?
make[2]: *** [3rdparty/carotene/hal/carotene/CMakeFiles/carotene_objs.dir/build.make:76: 3rdparty/carotene/hal/carotene/CMakeFiles/carotene_objs.dir/src/absdiff.cpp.o] Error 1
make[1]: *** [CMakeFiles/Makefile2:3237: 3rdparty/carotene/hal/carotene/CMakeFiles/carotene_objs.dir/all] Error 2
make[1]: *** Waiting for unfinished jobs....
[ 18%] Built target IlmImf
[ 19%] Built target ade
[ 24%] Built target libprotobuf
make: *** [Makefile:166: all] Error 2
pi@raspberrypi:~/opencv-4.0.0/build $
```

Figure 35: Unrecognized command line

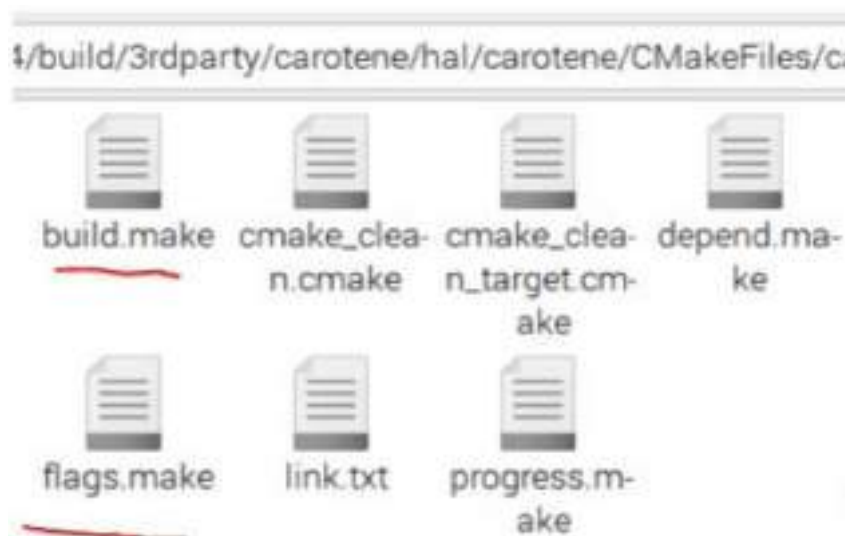


Figure 36: Files requiring modifications

2. Import Error: Numpy.core.multiarray failed to import.

Solution: `pip install -U numpy`.


```

pi@raspberrypi:~/opencv-4.0.0/build $ python3
Python 3.9.2 (default, Mar 12 2021, 04:06:34)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cv2
RuntimeError: module compiled against API version 0xe but this version of numpy
is 0xd
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.9/dist-packages/cv2/__init__.py", line 8, in <mod
ule>
    from .cv2 import *
ImportError: numpy.core.multiarray failed to import
>>>

```

Figure 37: Numpy.core.multiarray failed to import

3. Import Error: Cannot show CV2.

Solution: Run `sudo apt-get install python3-opencv` to fix it.

```

pi@raspberrypi:~/opencv-4.0.0/build $ python
Python 2.7.18 (default, Jul 14 2021, 08:11:37)
[GCC 10.2.1 20210110] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import cv2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named cv2
>>>

```

Figure 38: Cannot find cv2

```

pi@raspberrypi:~/opencv-4.0.0/build $ python3
Python 3.9.2 (default, Mar 12 2021, 04:06:34)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cv2
>>> cv2.__version__
  File "<stdin>", line 1
    cv2.__version__
    ^
SyntaxError: invalid syntax
>>> cv2.__version__
'4.5.4'

```

Figure 39: Showing the cv2 version

6.5.3 Installation on Jetson Nano

For Jetson Nano, the installation was performed by burning the image directly. It is possible to install the system into the Jetson Nano in the same way. The detailed

installation process of the system is described in the steps below, supported by Figures 40 and 41.

1. Install BalenaEtcher on the computer.
2. Download operating systems from the official Jetson Nano website.
3. Insert the SD memory card, using the SD Memory Card Formatter to format two microSD cards to FAT (both FAT16 and FAT32) file system.
4. Click “Flash from file” and choose the previously downloaded zipped image, then select SD card to install systems.

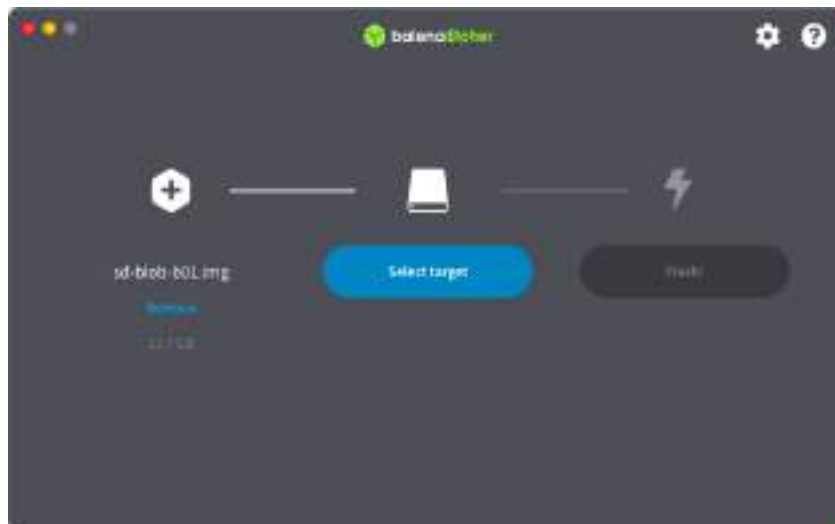


Figure 40: Zipped image selection

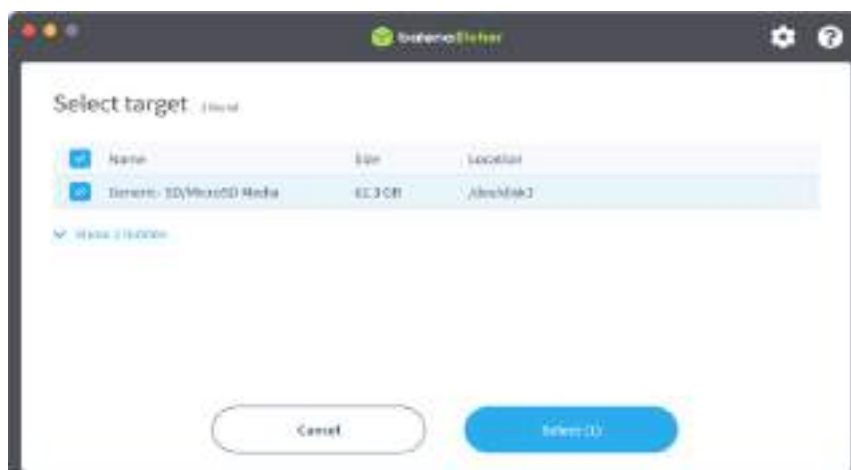


Figure 41: SD card selection

1. After installing the system, insert the SD card to Jetson Nano and switching the power on.

6.5.4 Problems on Jetson Nano and their solutions

Running the camera on Jetson Nano requires a more extensive programming setup, while on Raspberry Pi it only requires `cap=cv2.VideoCapture(0)`. When running the camera on Jetson Nano, more parameters are needed, as demonstrated in Figures 42 and 43.

```
1  def gstreamer_pipeline(  
2      capture_width=480,  
3      capture_height=240,  
4      display_width=480,  
5      display_height=240,  
6      framerate=30,  
7      flip_method=0,  
8  ):  
9      return (  
10         "nvargb2rgb! " "  
11         "video/x-raw(memory=MM), " "  
12         "width=(int)%d, height=(int)%d, " "  
13         "format=(string)NV12, framerate=(fraction)%d/1 ! " "  
14         "nvvidconv flip-method=%d ! " "  
15         "video/x-raw, width=(int)%d, height=(int)%d, format=(string)BGRx ! " "  
16         "videocolor ! " "  
17         "video/x-raw, format=(string)BGR ! appsink" "  
18     ) % (  
19         capture_width,  
20         capture_height,  
21         framerate,  
22         flip_method,  
23         display_width,  
24         display_height,  
25     )  
26 )  
27  
28 cap = cv2.VideoCapture(gstreamer_pipeline(flip_method=2), cv2.CAP_GSTREAMER)
```

Figure 42: The camera code from Jetson Nano

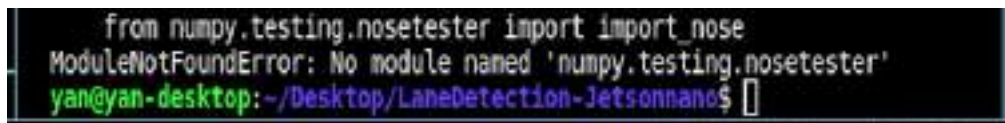
```
67 cap = cv2.VideoCapture(0)
```

Figure 43: The camera code from Raspberry Pi 4

In addition, when running a compiled program on Jetson Nano, certain problematic prompts appear. Before they can be fixed, it is necessary to install some additional software support, whereas in Raspberry Pi 4 and NCS2, such problems do not exist. Some examples of the problems occurring with Jetson Nano are presented as follows.

- The following error message appears: `ModuleNotFoundError: No module named 'numpy.testing.nose'`

Solution: This problem can be fixed by running `scipy => 1.1.0`.

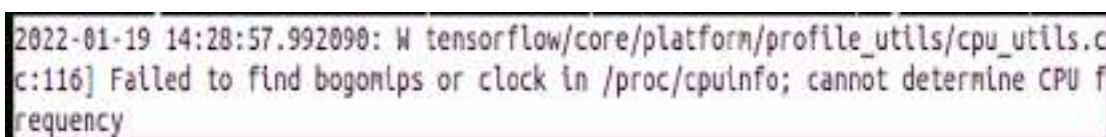


```
from numpy.testing.nose import import_nose
ModuleNotFoundError: No module named 'numpy.testing.nose'
yan@yan-desktop:~/Desktop/LaneDetection-Jetsonnano$
```

Figure 44: No module named 'numpy.testing.nose'

- The following error message appears: *Failed to find bogomips or clock in /proc/cpuinfo; cannot determine CPU frequency*

Solution: This problem can be fixed by running `python3 heelo.py`.

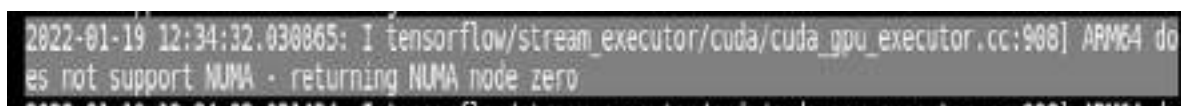


```
2022-01-19 14:28:57.992090: W tensorflow/core/platform/profile_utils/cpu_utils.c
c:116] Failed to find bogomips or clock in /proc/cpuinfo; cannot determine CPU f
requency
```

Figure 45: Cannot determine CPU frequency

- The following error message appears: *ARM64 does not support NUMA – returning NUMA node zero*

Solution: This problem can be fixed by running `apt-get install -y libcudnn8 nvidia-cudnn8`.



```
2022-01-19 12:34:32.030865: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:908] ARM64 do
es not support NUMA - returning NUMA node zero
```

Figure 46: ARM64 does not support NUMA

6.5.5 Installation of Neural Computer Stick 2

Because NCS2 depends on other devices with a USB interface to run, users need to install the driver in another device, for instance Windows or Raspbian. The installation method described on the official Intel website [31] can be used as a reference.

Although the website indicates that the device can run on the Linux and Mac operating systems, the process is not particularly straightforward. Firstly, while the drive could previously be installed on the Linux system, this system is no longer supported. For MacOS, an older version is required, as it only can run on versions 10.13, 10.14 or 10.15. In the end, the final composition appears as demonstrated in Figures 47-49.



Figure 47: The macOS version used

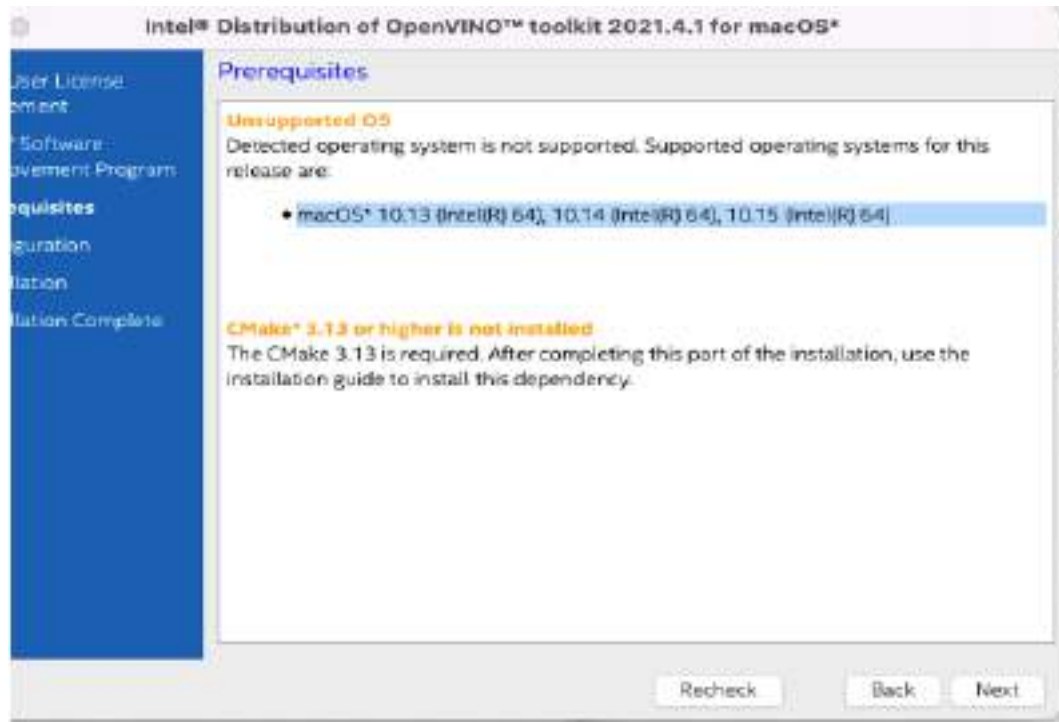


Figure 48: Notification message when installing the driver on macOS



Figure 49: Notification message when installing the driver on Jetson Nano

7 Results and evaluation

In this section, the results for the comparison of running Python on Jetson Nano, Raspberry Pi and Neural Compute Stick 2 are presented. They will then be evaluated to determine which one is the most suitable for traffic sign training and lane detection when using Python.

7.1 Comparison on running the training programming

In order to conduct a relative comparison on different edge devices, the trafficSign_training file was used to run it on the Raspberry Pi 4, Jetson Nano and NCS2. To complete the calculation of the time the program runs on each of the devices, the following code was used:

```
18 import time
19 start_time = time.time()
20 print("--- %s seconds ---" % (time.time() - start_time))
```

Figure 50: Programming for time calculation

This code calculates the running time of the entire program by subtracting the start time from the completion time.

7.2 Observations on Raspberry Pi 4

In order to give Raspberry Pi 4 its compact and thin design, it has no cooling device installed on it. However, the temperature still has a certain impact on the running time of the program.

During the tests, when turning the Raspberry Pi on, and while nothing is running on the device, the CPU temperature rises to around 50 to 60 celsius degrees. When installing libraries, such as OpenCV, Pandas and NumPy, the temperature climbs up to around 70 celsius degrees. Then when Raspberry Pi is running trafficSign_training file in room temperature of approximately 20 celsius degrees, the temperature of the CPU climbs up to 78 celsius degrees.


```

Test Score: 0.0650186300877182
Test Accuracy: 0.98275864
-----7437.359803199768 seconds-----
pi@raspberrypi:~/Desktop/LaneDetection-raspberrypi-final $

```

Figure 51: Total processing time of Raspberry Pi 4

While running the code, GL-Z software can be used to monitor the GPU working status as seen in Figure 52. However, it is not possible to clearly determine whether the GPU contributes to the calculation process, because the GPU remains unchanged even when other programs are running.

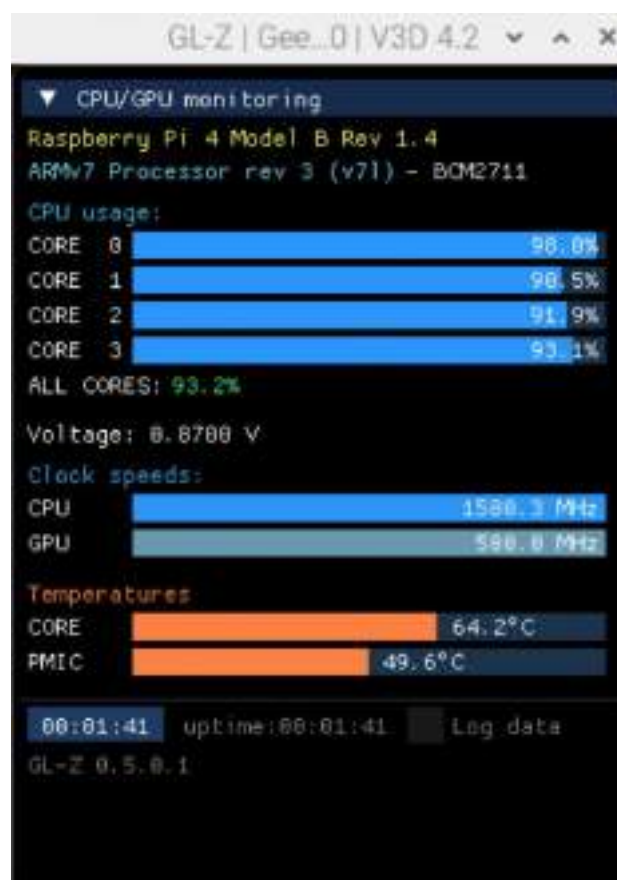


Figure 52: GL-Z software monitoring the GPU

7.3 Observations on Jetson Nano

The results seen on Jetson Nano are quite the opposite. Since the device includes a heat sink, the CPU temperature can be maintained between 40 and 50 celsius degrees at all times, in addition to which the Jetson-Stat software shows that the GPU is working during training. The training time is also shorter compared to Raspberry Pi 4.

```

y: 0.9037 - val_loss: 0.0468 - val_accuracy: 0.9886
Test Score: 0.04423504539350129
Test Accuracy: 0.9886743426322937
---4882 seconds---
van@yan-desktop:~/Desktop/LaneDetection-Jetsonnano1$

```

Figure 53: Total processing time on Jetson Nano

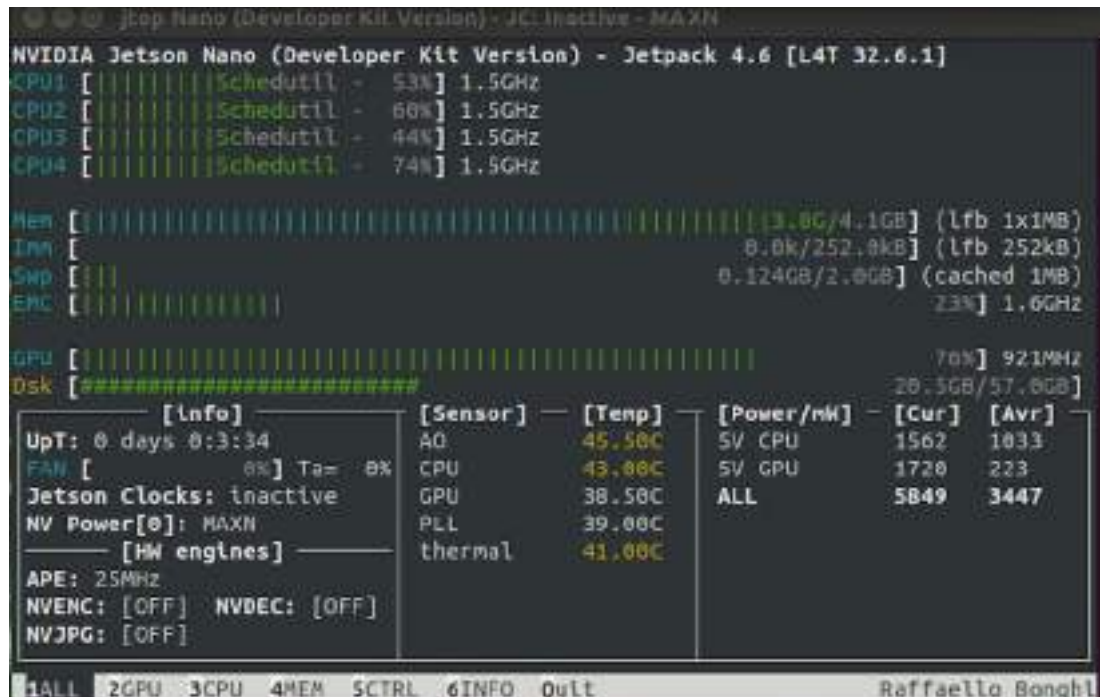


Figure 54: Jtop software monitoring the GPU

7.4 Observations on Neural Compute Stick 2

The NCS2 device was run on the USB3 interface of the Raspberry Pi 4. During the testing phase, the device did not seem to significantly improve the running time on Raspberry Pi. The obtained result is presented in Figure 55.

```

Test Score: 0.000311287059459524
Test Accuracy: 0.9823270
-----7130.1416001910045 seconds-----
(openvino) pi@raspberrypi:~/Desktop/LaneDetection-raspberrypi-Final $

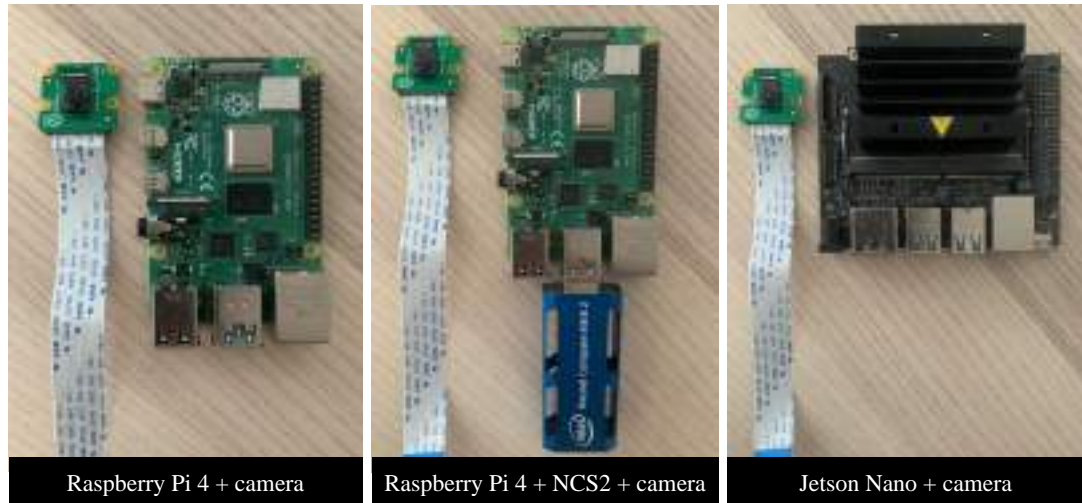
```

Figure 55: Total processing time on Raspberry Pi 4 + NCS2

7.5 Edge device comparison for image processing speeds

Speed is an important indicator in image processing, particularly in the context of autonomous vehicle technology where adequate reaction speed in rapidly changing situations is critical. The objective of this section is to detect differences

between the image processing speeds on the three edge devices depicted in Figure 56, and as a result, determine which of the devices reaches the best results for traffic sign recognition and lane detection in terms of speed.



*Figure 56: Edge devices + camera
Own elaboration*

In order to compare the performance of these three devices, the concept of Frames Per Second (FPS) is introduced here to evaluate the image processing speeds of the devices. FPS indicates the number of frames per second of an image that is displayed, and is a measure of graphics rendering performance and smoothness. A high FPS indicates that the image is updated faster and will therefore appear smoother, whereas a low FPS can cause image lag. However, while computing speed may affect the performance of graphics rendering, it should not be regarded as the only indicator. The FPS value also affected by other factors, such as graphics card performance and memory speed.

In this project, the devices were complemented with small camera which was used to read varying traffic situations. To facilitate comparison during the reading process, two screen captures for each of the traffic signs and lanes were taken at random moments to obtain two randomized speed results for image recognition. The FPS measure is used here as a general indicator.

The comparison was conducted through the following steps:

1. Prior to introducing any driving paths or traffic signs to the camera, blank images without such elements were used to measure the standard speeds on the edge devices.

2. In the next step, thirteen distinct and commonly seen traffic signs were introduced to the camera to see how quickly they were identified by the different edge devices. Here, both miniature plastic traffic signs as well as images displayed on a screen were used.
3. Finally, self-constructed lanes were laid out firstly to verify the code's ability to detect lanes, and secondly to measure the image processing speeds on the different edge devices. It should be noted that as the test road was built by hand, the side borders were not always completely parallel. As a result of this misalignment, the road measurements appear somewhat distorted in the image.

Next, the code used is introduced in Figure 57. This is followed by Table 4 presenting the randomly captured images alongside with the results measured in FPS which were obtained in the moment of capturing the images.

```
start_time = time.time()
display_time = 0.1
fc = 0
FPS = 0
while True:
    success, imgOriginal = cap.read()
    imgOriginal = cv2.resize(imgOriginal, (480, 740))
    Result = getTrafficSign(imgOriginal)
    fc += 1
    TIME = time.time() - start_time




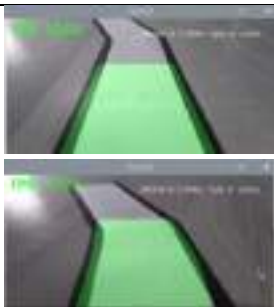





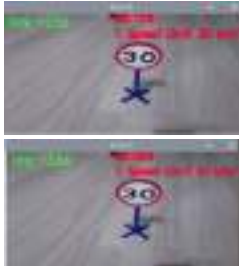


    if (TIME) >= display_time:
        FPS = fc / (TIME)
        fc = 0
        start_time = time.time()









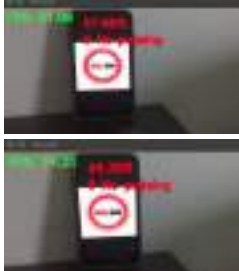




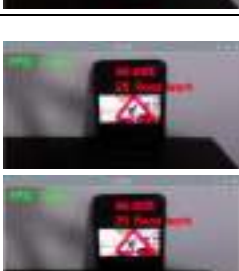

    fps_disp = "FPS: " + str(FPS)[1:]

    # Add FPS count on frame
    Result = cv2.putText(Result, fps_disp, (10, 25),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
    #cv2.imshow('Video Stream w/ FPS', image)

    cv2.imshow("Result", Result)
    cv2.waitKey(1)
```

Figure 57: Programming for time calculation

	Raspberry Pi 4 (With camera)	Neural Compute Stick 2 on Raspberry Pi 4 (With camera)	Jetson Nano (With camera)
Pixels	480*240	480*240	480*240
Without traffic sign or lane			
FPS	28~31	28~30	50~60
			
Lane detection			
FPS	8~10	7~8	6~8
			
Traffic sign identification			
FPS	10~15	11~17	20~24
Stop			
30km/h limit			

Traffic sign identification				
FPS	10~15	11~17	20~24	
50km/h limit				
60km/h limit				
No passing				
No entry				
Road work				



















Traffic sign identification			
FPS	10~15	11~17	20~24
Turn right ahead			
Turn left ahead			
Bumpy road			
Go straight or right			
Ahead only			
Beware of ice/snow			

Table 4: Lane detection comparison (Own elaboration)

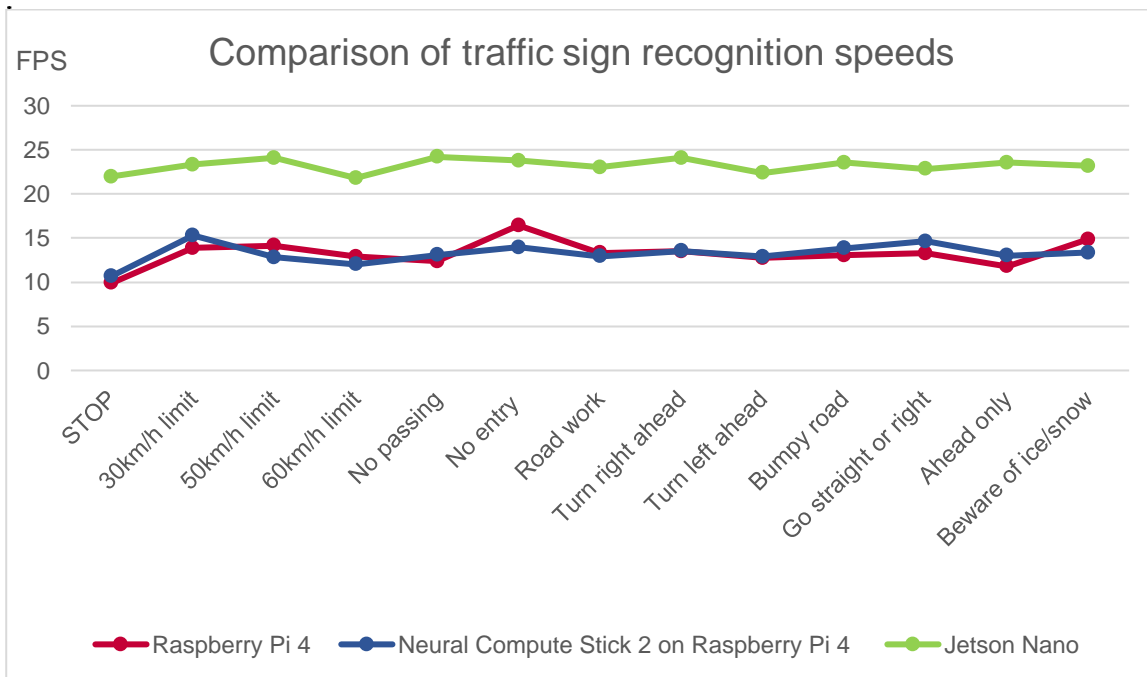


Figure 58: Comparison graph of traffic sign recognition speeds

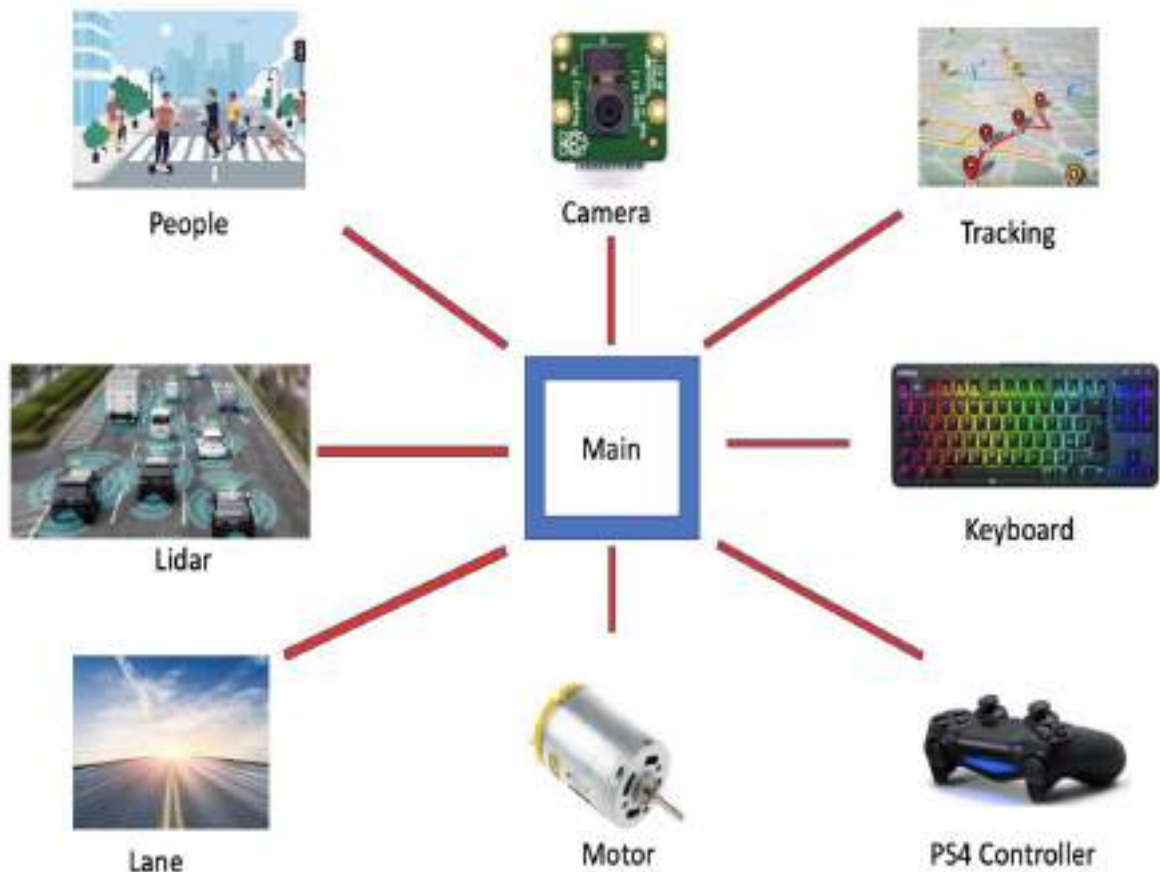
It can be concluded that while the FPS of the three devices do not differ much when running the `lane_detection_main` code, the differences are more noticeable when running the `get_traffic_sign`. These differences are visualized in Figure 58, which marks the higher FPS measurement for each of the randomly captured image pairs during image recognition tests.

As illustrated by the comparison graph, Jetson Nano is significantly faster regardless of the traffic sign in question. Raspberry Pi 4 and NCS2 on Raspberry Pi 4 showed similar results, with a very slight improvement in speed in eight of the thirteen traffic sign recognition runs. However, based on these tests, the differences between Raspberry Pi 4 with NCS2 and without are not clear enough to draw definite conclusions on the processing speed.

7.6 Donkey car

As an extension to this project, a donkey car was constructed to serve as an intelligent small-scale car, equipped with a road and traffic sign recognition system. The algorithm of this system is based on deep learning and end-to-end control, which gives it the ability to recognize different roads and traffic signs and thus, enabling automatic driving. In this project, a camera was installed on the donkey car in order to capture data of the environment.

This part explains how an edge device, in this case Raspberry Pi, was used to run the donkey car and how the communication between software and hardware was realized. Figure 59 demonstrates the variety of information that can be utilized by the donkey car in autonomous driving.



*Figure 59: The donkey car control schematic
Own elaboration*

To serve the purpose of this project, the lane detection for the donkey car was enabled through three stages as follows:

1. Firstly, a series of images was collected through the use of code containing information collected in the camera file, the joystick file, the data collection file and the motor control file.
2. In the next stage, the PC was used to process all the data, including the analysis of lane images and traffic sign training, in order to enable the donkey car to detect its course and take responsive action.
3. The last step was to use all the files to control the donkey car running on the lane and to observe its performance. The schematic diagram is presented in Figure 60.

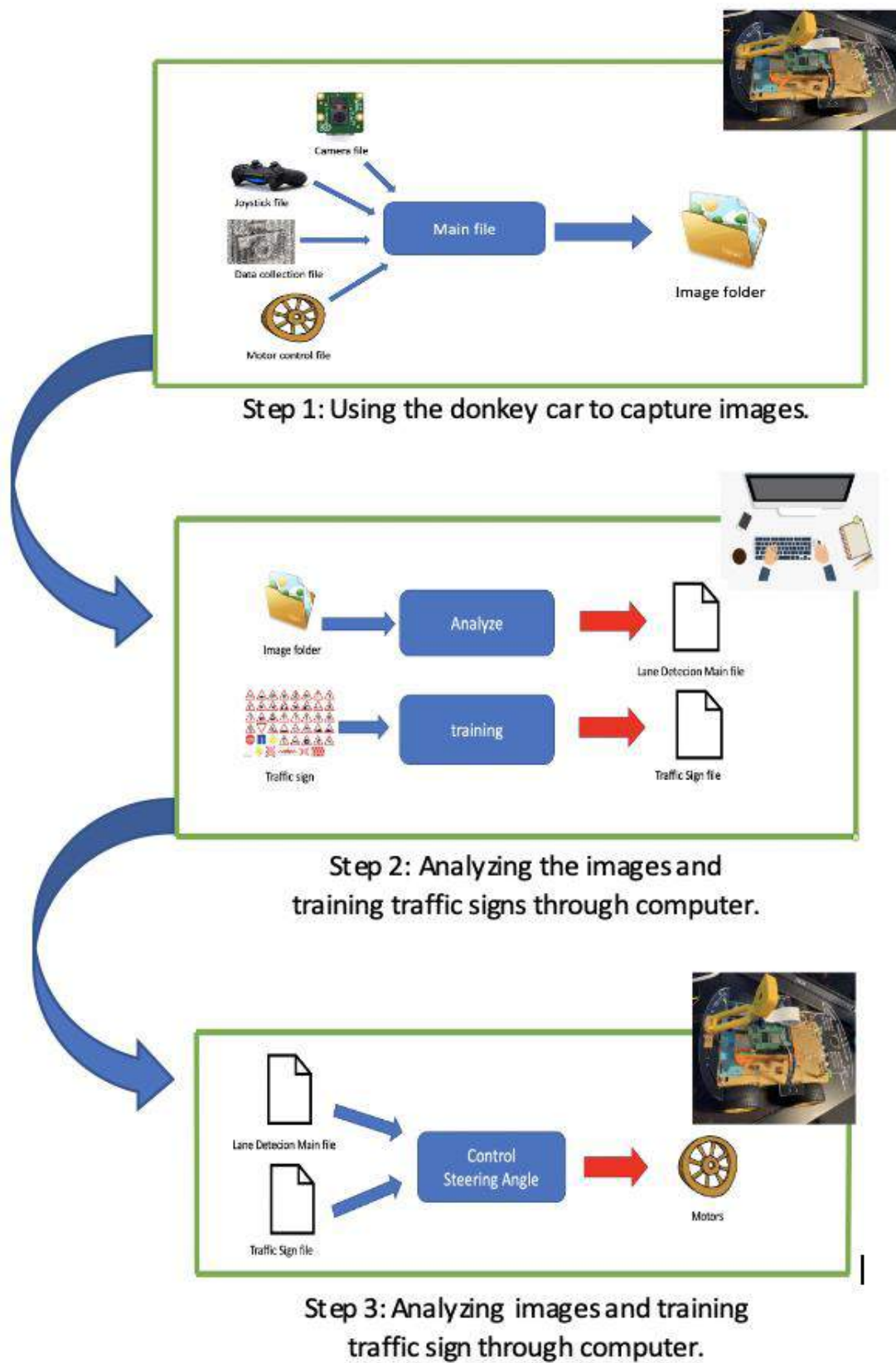


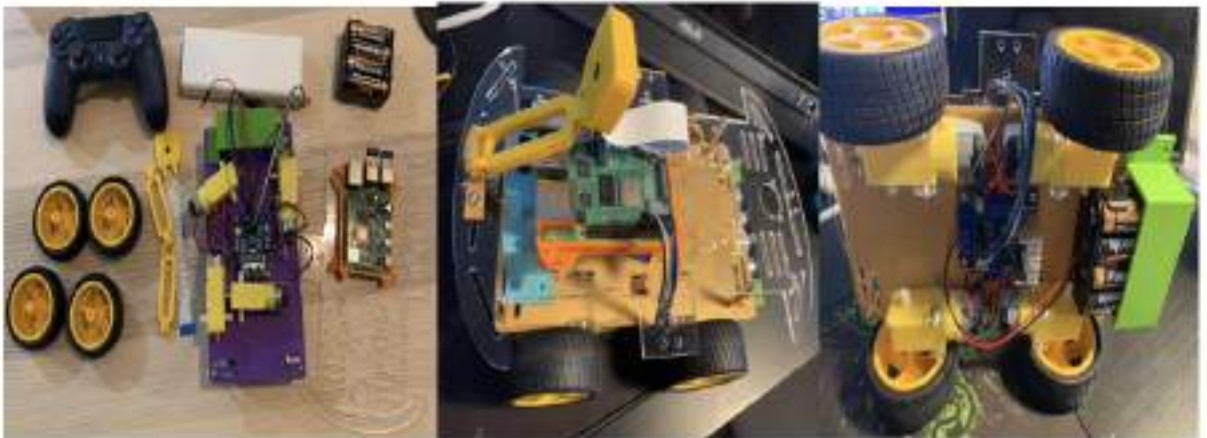
Figure 60: The schematic diagram
Own elaboration

7.6.1 The donkey car setup

In order to capture images for analysis and to put the training into practice to evaluate the performance on different edge devices, a donkey car was constructed with the following composition:

- 1 x Raspberry Pi 4
- 1 x L298 Dual Full Bridge Driver
- 1 x Raspberry camera
- 8 x AA size batteries
- 1 x Powerbank (the output must be 5V/3A to drive the Raspberry Pi 4)
- Various 3D-printed parts
- 1 x PS4 controller

Due to various size-related issues, some 3D-printed parts had to be replaced. In the end, the final donkey car setup is as demonstrated in Figure 61 below.



*Figure 61: The donkey car setup
Own elaboration*

The L298N motor driver permits high voltage and can be driven by both DC motors and stepper motors. One driver chip can control two DC geared motors to perform different actions simultaneously in the voltage range of 6V to 46V. The module is presented in Figure 62.

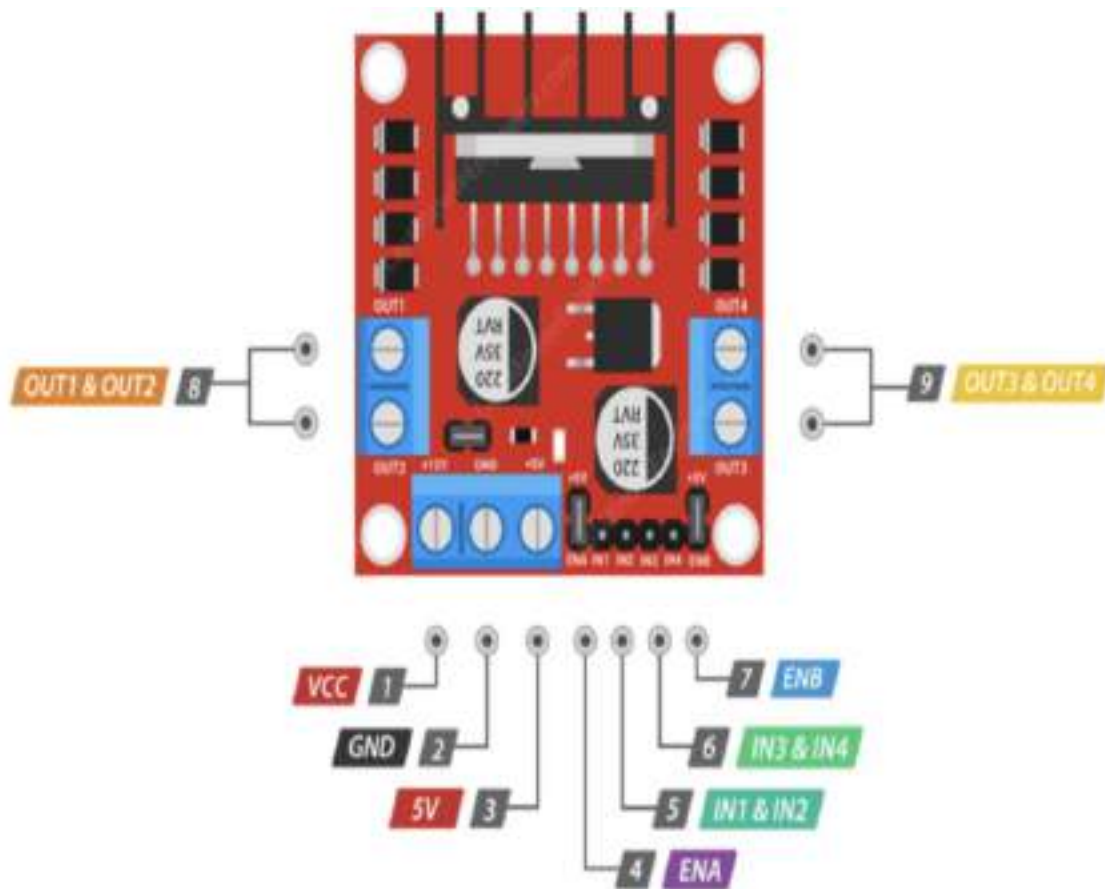


Figure 62: L298N Motor Driver Module Pinout [32]

7.6.2 Results on the donkey car

As the final step, the previous programming is applied to a simulated traffic scenario. An edge device, in this case Raspberry Pi 4, is used to control the donkey car to determine how accurately it is able to react to an example traffic scenario. The camera installed on the donkey car captures images and transmits them to the edge device, where they are processed and used to control the speed of the motor.

To test the programming, a simple traffic scenario including a curving lane and two traffic sign types is created. As shown in Figure 63, the donkey car collects two images in real time; one to determine the center of the lane to ensure the car follows the road direction, and the other one to identify the traffic sign. In addition, when encountering the traffic sign recognized by the program, the car responds with an appropriate decision.

During the test, the motor power of the donkey car is set at 20% of the maximum motor power. When testing the speed limit function, upon encountering the 60 km/h sign, the car adjusts its speed accordingly by increasing it to 30%. When testing the

stop sign function, upon encountering the stop sign, the car stops for 3 seconds.



Figure 63: Testing a traffic scenario

8 Conclusion

While it can be concluded that each of these edge devices provides a solid platform for AI interference at the edge, conducting a fair comparison between them is not a straightforward task. In terms of hardware performance in the context of this thesis project, some minor differences could be detected during the tests. Speed in terms of FPS was selected as an indicator to determine differences between the edge devices.

When testing image recognition speed, Jetson Nano achieved higher FPS rates compared to Raspberry Pi and NCS2 when there were no traffic signs or lanes in the camera view. Upon testing traffic sign recognition, Raspberry Pi and NCS2 showed similar results in terms of FPS, whereas Jetson Nano performed the traffic sign recognition at a rate approximately 50-60% faster compared to the other two. Although all three devices demonstrated similar FPS rates during the tests for lane detection, Raspberry Pi performed slightly faster than the other two.

On a more general level, some key advantages and disadvantages were identified for each device during the research, as well as the implementation stage. These are summarized in Table 5 below.

Raspberry Pi 4	Jetson Nano	NCS2
Pros		
User-friendly interface. Suitable for beginners who want to get started in the field of AI or learn more about Python programming. Plenty of resources available online. Programs from other computer devices can run directly in the Raspberry Pi 4 with essentially no subsequent changes.	Running the training code was significantly faster than with the other two devices. Better temperature control due to heat sink.	Reliability (decentralized, network connections not required).

Raspberry Pi 4	Jetson Nano	NCS2
Cons		
<p>Overheating in the CPU.</p> <p>The program crashes easily.</p> <p>No restart button, the only option is unplugging the power.</p>	<p>Not recommendable for beginners.</p> <p>Compatibility issues; Every API must work well together before the programming can be run properly; running certain programs requires changes to the dictionary and program details to meet the specific needs of Jetson Nano.</p> <p>Crashes easily (e.g. during the installation of OpenCV).</p> <p>Lacking a restart button, leaving unplugging the power as the only option.</p>	<p>Cannot be used independently; requires installation to another device with a USB port in order to be used.</p> <p>Tedious installation process.</p> <p>Did not work with Jetson Nano.</p> <p>The latest macOS is not supported.</p>

*Table 5: Summary of differences between edge devices
Own elaboration*

9 References

- [1] A. Gupta, A. Anpalagan, L. Guan and K. Shaharyar, "Deep learning for object detection and scene perception in self-driving cars: Survey, challenges, and open issues.," *Array*, vol. 10, 2021.
- [2] V. & M. O. Santos, "Special issue on autonomous driving and driver assistance systems.," in *Robotics and Autonomous Systems 91*, Elsevier, 2017, pp. 208-209.
- [3] R. Liao, "Huawei says it's not a carmaker — it wants to be the Bosch of China," 2021a. [Online]. Available: <https://techcrunch.com/2019/07/26/ethics-in-the-age-of-autonomous-vehicles/>. [Accessed 20 12 2021].
- [4] R. Gordon, "Explained: Levels of autonomy in self-driving cars," CSAIL, 10 December 2020. [Online]. Available: <https://www.csail.mit.edu/news/explained-levels-autonomy-self-driving-cars>. [Accessed 19 12 2021].
- [5] European Commission, "On the road to automated mobility: An EU strategy for mobility of the future. Eur-lex.europa.eu. <https://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2018:0283:FIN:EN:PDF>," 17 May 2018. [Online]. Available: <https://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=COM:2018:0283:FIN:EN:PDF>. [Accessed 31 12 2021].
- [6] D. Yifan, "Autonomous software-defined control," 22 July 2021. [Online]. Available: <http://finance.people.com.cn/BIG5/n1/2021/0722/c1004-32165709.html>. [Accessed 29 November 2021].
- [7] T. Brell, R. Philipsen and M. Ziefle, "Suspicious minds?—users' perceptions of autonomous and connected driving," *Theoretical issues in ergonomics science*, vol. 20, no. 3, pp. 301-331, 2019.
- [8] K. Chen, "The World's #1 Autonomous Driving Competition Begins," 21 April 2021. [Online]. Available: <https://technews.tw/2021/04/21/who-is-the-best-autonomous-vehicle/>. [Accessed 21 December 2021].
- [9] Y. Huang, "Autonomous driving technology is coming in force, five years later, half of China's new cars will be autonomous," TechNews, 2020. [Online]. Available: <https://technews.tw/2020/11/12/self-driving-cars-in-china-will-boom/>.
- [10] J. Wang, "Overcome the challenges of autonomous driving technology development and accelerate the launch of a new era of intelligent life," 22 December 2020. [Online]. Available: <https://view.ctee.com.tw/technology/25655.html>. [Accessed 2 January 2021].

- [11] E. Burns, "Aprendizaje profundo (Deep learning)," ComputerWeekly, September 2021. [Online]. Available: <https://www.computerweekly.com/es/definicion/Aprendizaje-profundo-deep-learning>. [Accessed 31 December 2021].
- [12] M. Dhande, "What is the difference between AI, machine learning and deep learning?," Geospatial World, 7 March 2020. [Online]. Available: <https://www.geospatialworld.net/blogs/difference-between-ai%EF%BB%BF-machine-learning-and-deep-learning/>. [Accessed 15 01 2022].
- [13] Y. LeCun, Y. Bengio and G. Hinton, "Deep Learning," Nature, 7 May 2015. [Online]. Available: <https://doi.org/10.1038/nature14539>. [Accessed 5 January 2022].
- [14] I. E. Sarker, "Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions," SN Computer Science, 18 August 2021. [Online]. Available: <https://doi.org/10.1007/s42979-021-00815-1>. [Accessed 09 January 2022].
- [15] J. Chen and X. Ran, "Deep Learning With Edge Computing: A Review," *Proceedings of the IEEE*, vol. PP, pp. 1-20, 2019.
- [16] K. Zhang, X. Gui, D. Ren, J. Li, J. Wu and D. Ren, "Zhang K., Gui X., Ren D., Li J., Wu J. & Ren D. (2019). Mobile Edge Network Computing Download and Content Caching Survey," *Journal of Software*, vol. 30, no. 8, pp. 2491-2516, 2019.
- [17] A. Ferdowsi, U. Challita and W. Saad, "Deep Learning for Reliable Mobile Edge Analytics in Intelligent Transportation Systems: An Overview," *IEEE Vehicular Technology Magazine*, vol. 14, no. 1, pp. 62-70, 2019.
- [18] D. J. Yeong, G. Velasco-Hernandez and J. Barry, "Sensor and Sensor Fusion Technology in Autonomous Vehicles: A Review. Sensors 2021.," *Sensors*, vol. 6, 2021.
- [19] M. Véstias, R. Duarte, J. Sousa and H. Neto, "Moving Deep Learning to the Edge," *Algorithms*, vol. 13, no. 5, 2020.
- [20] W. Shi, X. Zhang and Q. Zhang, "Edge Computing: State-of-the-Art and Future Directions," *Journal of Computer Research and Development*, vol. 56, no. 1, pp. 69-89, 2019.
- [21] J. Gonzales, "Machine learning edge devices: benchmark report," Tryo Labs, 9 October 2019. [Online]. Available: <https://tryolabs.com/blog/machine-learning-on-edge-devices-benchmark-report>. [Accessed 03 February 2022].
- [22] Python, "General Python FAQ," [Online]. Available: <https://docs.python.org/3/faq/general.html#general-python-faq>. [Accessed 03 December 2022].
- [23] OpenCV, "About," [Online]. Available: <https://opencv.org/about/>. [Accessed 08 January 2022].

- [24] TensorFlow, "Overview," [Online]. Available: <https://www.tensorflow.org/overview>. [Accessed 15 January 2022].
- [25] DeepAI, "Convolutional Neural Network," [Online]. Available: <https://deepai.org/machine-learning-glossary-and-terms/convolutional-neural-network>. [Accessed 29 February 2022].
- [26] CSDN, "In-depth look at autonomous driving - Image recognition," 15 December 2016. [Online]. Available: https://blog.csdn.net/weixin_33829657/article/details/89128806. [Accessed 09 01 2021].
- [27] Seb, "Foundations of Deep Learning for Object Detection: From Sliding Windows to Anchor Boxes", May 2022. [Online] Available: <https://programmatically.com/foundations-of-deep-learning-for-object-detection-from-sliding-windows-to-anchor-boxes/>. [Accessed 22 Oct, 2023].
- [28] OpenSource, "What is a Raspberry Pi?," [Online]. Available: <https://opensource.com/resources/raspberry-pi>. [Accessed 7 January 2022].
- [29] NVIDIA, "Jetson Nano Developer Kit," [Online]. Available: <https://developer.nvidia.com/embedded/jetson-nano-developer-kit>. [Accessed 1 December 2022].
- [30] VISO.AI, "Intel Neural Compute Stick 2 – AI Vision Accelerator Review 2021," [Online]. Available: <https://viso.ai/edge-ai/intel-neural-compute-stick-2/>. [Accessed 28 December 2021].
- [31] Intel, "Get Started with Intel® Neural Compute Stick 2," [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/guide/get-started-with-neural-compute-stick.html>. [Accessed 20 November 2021].
- [32] Last Minute Engineers, "Interface L298N DC Motor Driver Module with Arduino," [Online]. Available: <https://lastminuteengineers.com/l298n-dc-stepper-driver-arduino-tutorial/>. [Accessed 10 January 2022].
- [33] K. Korosec, "Waymo's driverless taxi service can now be accessed on Google Maps," TechCrunch, 3 June 2021. [Online]. Available: <https://tcrn.ch/3ighhEM>. [Accessed 29 December 2021].
- [34] V. Ayumi, L. M. Rere, M. I. Fanamy and A. M. Aryurthy, "Optimization of Convolutional Neural Network using Microcanonical Annealing Algorithm," in *ICACSIS*, 2016.
- [35] P. Virtanen et al., "SciPy 1.0: fundamental algorithms for scientific computing in Python," *Nat Methods*, vol. 17, pp. 261-272, 2020.