UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

Prototype Self-Driving Car Powered by A Neural Network

FINAL YEAR PROJECT REPORT

Name: Lakshan Dadigamuwa

ID No.: 14148781

Supervisor's Name: Dr. Ciaran MacNamee

Course: LM118 – Electronics and Computer Engineering

Academic Year: 2017 - 2018

E&CE Department of Electronic & Computer Engineering

Prototype Self-Driving Car Powered by A Neural Network

Name: Lakshan Dadigamuwa

ID No.: 14148781

Supervisor's Name: Dr. Ciaran MacNamee

Course: LM118 – Electronics and Computer Engineering

Academic Year: 2017 - 2018

Abstract

The rapid advancements in technology, especially in the automotive industry, have begun to improve the transportation industry, providing a smarter and a safer driving experience. Companies such as Tesla, Google and Audi are trying to enhance this driving experience by introducing self-driving cars capable of autonomously navigating through busy towns and cities while maintaining a high safety standard for occupants and other road users.

This project aims to develop a similar concept through a low powered Raspberry Pi running an Artificial Neural Network (ANN). A Multilayer Perceptron (MLP) Neural Network was created to train the toy car in order to facilitate the recognition of lane marking and predict the steering outcome based on the visual information received. An onboard PiCamera was used to detect street signs and traffic lights. The 'Haar-like feature' based cascade classifier was chosen for object recognition task, which exploits the fact that most similar objects share common regularities such as shapes and sizes that can be classified into distinct categories. The ultrasonic sensor complements these machine vision systems to work collectively to perform the driving behaviours of a self-driving car. ANN based autonomous driving was successfully achieved, despite the limitations of the Raspberry Pi's computing power and the car's mechanical shortcomings.

Acknowledgment

I would like to express my deepest gratitude to my supervisor, Dr. Ciaran McNamee, for his continuous support, guidance and the enthusiastic encouragement throughout this project.

I would also like to thank the ECE department and the technical staff for the use of laboratory equipment and a special thanks to Dr. Colin Flanagan and Raj Shah for their help and advice to bring this project to fruition.

Table of Contents

List of Figures

List of Tables

1 Introduction

In the current world we live in, the significance of mobility is often taken for granted, with most of us neglecting to realise that the transportation of humans and goods forms the fundamental basis of our civilization and economy [1]. As cities and populations continue to grow, demand for safer and more efficient transportation systems is becoming somewhat of a pervasive issue for future generations. New methods of traffic calming, improvements in infrastructure, and stricter laws have shown a reduction in vehicle related incidents over the years. In 2017, the Road Safety Authority recorded an 18% decrease in collisions and 15% decrease in number of deaths (186 to 159) for motor related fatalities compared to provisional Garda data for the full year of 2016. There was also a small decrease in deaths of vulnerable road users such as pedestrians, motorcyclists and cyclists between 2016 and 2017 [2].

Although this data suggests that there is some decline in road related accidents, the current transportation system is still incapable of keeping passengers and the general public from harm caused by irresponsible drivers. The need for an upgrade to our current transportation system is clearly evident, with the dawn of autonomous vehicles posing as the proverbial "light at the end of the tunnel" in the quest to reduce the number of road traffic accidents.

The exponential evolution of technology, especially in the automotive industry, is already beginning to influence our everyday lives. Nowadays, most cars are equipped with some form of computational software and hardware array used to provide a smarter and a safer driving experience. These advancements in vehicle technology, ranging from basic Anti-Locking Breaking Systems to more sophisticated systems such as adaptive cruise control, automatic breaking, lane keeping assist, and traction control have already helped to save countless lives and in doing so, paved the way to a new era of Advanced Driver Assistance systems (ADAS)[3].

In the near future, automated systems will be more ubiquitous. In the new Glucksman Library, human librarians are being replaced by an Automated Storage and Retrieval System (ASRS) [4], while in the automotive sector, more sophisticated tasks such as allowing a car to drive by itself on public roads are performed by cars such as the Model S from Tesla [5]. These advances will inevitably help to avoid work and road related accidents and to effectively increase the efficiency of how current systems work. Many companies and institutions such as Google, Uber, and Tesla, along with many prominent start-ups are already developing the next-generation autonomous vehicles that will alter our roads and lay the groundwork for intelligent transportation networks of the future.

1.1 The Project Aims

The modern concept of an autonomous car was first introduced in the late 1980s but the idea of an autonomous car only became mainstream when Tesla introduced their 'Autopilot' self-driving system in 2014. Since the introduction of the 'Autopilot' system, many individuals and companies have tried to recreate this concept using similar technologies with great success. This project was undertaken to test out the theory that a self-driving car with an artificial neural network could be implemented on a simple Raspberry Pi, with a camera and an ultrasonic sensor.

The primary aim of this project is to convert a remote-controlled toy car into an autonomous self-driving vehicle capable of handling four main tasks:

- Self-driving on a track
- Front collision avoidance
- Stop sign and traffic light detection
- Changing speed in response to speed signs

What may seem like an easy task for human drivers, for a computer, it would take a lot of data, processing power and a highly trained neural network to execute such tasks.

Originally, when this project was first started in 2015, the aim was to navigate the car in any environment using just proximity sensors such as ultrasonic sensors and infrared sensors to detect and avoid obstacles on its path. To increase the scope of the project, implementing artificial intelligence and machine vision capabilities yields a greater challenge and a better navigation system than using just the proximity and infrared sensors alone.

On a personal level, I aim to develop the ability to apply myself in an analytical way, developing my creativity, knowledge and exposure to artificial intelligence in a timely and articulate fashion. I aim to expand and develop my current understanding of machine vision and machine learning technologies.

1.2 Objectives

- Investigate current technologies used in vehicle autonomy and how they can be implemented on a Raspberry Pi.
- Create three HAAR feature-based cascade classifiers for stop sign, traffic light and speed sign detection.
- Collect training data by manually driving the car around a track using an onboard PiCamera.
- Train the multilayer perceptron which is a class of feedforward artificial neural network to recognise the track layout.
- Implement the ultrasonic sensor to detect objects in front of the car.
- Develop a working prototype of the toy car with the trained neural network and test it around a track with road signs in place.

1.3 Project Structure

From the initial research it was clear that to simplify this project, each segment of the project had to be broken down into individual tasks. It was also clear that each of these tasks had to be completed and tested prior to moving on to the next stage. To facilitate continuous research and development of each module independently and to simplify the debugging process, a systematic approach was adopted. Some of these individual tasks are discussed in the 'Objectives' in section 1.2 and will be discussed in greater detail in the 'System Design' chapter along with few minor tasks that were essential for the success of this project.

The advantage of this approach was that from early stages of the module testing phase, most of the bugs and discrepancies could be mitigated before entering the final stages of implementation and testing. Although this methodical structure meant following a strict timeline was essential, there was still enough incorporated flexibility around each module to facilitate an evolving design. The main disadvantage of this system was that some major issues with development and testing caused delays and interference to other modules thus effecting the time frame of the project.

1.4 Project Planning

During the pre-assessment of the project, a Gantt chart was proposed to layout the project schedule, illustrating the various tasks, objectives and their allocated timelines throughout the entirety of the project. Prior research and contact with the project supervisor paved the way to developing the scope of the project early, which was crucial for understanding and setting target milestones throughout the project. This systematic method of project development is highly evident in the Gantt chart as most of the tasks start at the end of the previous task. Since this was a homebrew project, most of the main components were already set up and tested out prior to meeting with the supervisor, which helped to convey the ideas and objectives of this project in a clear manner. Overall, the schedule was in line with the plan, although there were some minor inconveniences such as being unable to install the machine vision software library OpenCV on the Raspberry Pi and damaging the PiCamera due to static electricity, resulted in slight alterations to the planned timeline.

Note: There was a month-long break for exams and Christmas holidays outlined in the Gantt chart denoted by the task name 'T20' in Appendix 8.1.

1.5 Assumptions

During the scope of the project, a number of assumptions were made mainly in relation to the hardware and the mechanical properties of the car itself.

- From the research conducted on portable single-board computers, it was presumed that the processing power of the Raspberry Pi 3 model B would be sufficient to run a neural network that processes the data from a video feed to predict the steering outputs.
- The readily available 5-megapixel PiCamera with an angle of view of 53° horizontally and 41° vertically would be able to instantly detect the road signs at a close range.
- It was also assumed that the short length of the track laid out within the available space would be sufficient to gather enough data to train the multilayer perceptron (MLP) to a high degree of accuracy.
- A calculated assumption was made to power both the Raspberry Pi and the Motor HAT using a single 5V power bank with dual 2.4-amp and 2.1-amp current outputs respectively.
- From the beginning of the project, it was assumed that the mechanical properties of the car such as the torque and speed of the two motors and the traction on the wheels would be enough to drive and steer the car.

1.6 Report Structure

This report consists of 5 main chapters: introduction, existing technologies, utilised hardware & software, system design and results. Each of these chapters contains sub-chapters, exploring the theories, discoveries, results and conclusions to convey the idea and the design behind this project.

- Introduction – This chapter introduced the readers to the underlying idea behind the concept of an autonomous self-driving car, the inspiration behind it and the objectives that are hoped to achieve at the end of the project.
- Existing technology – Initial research conducted on current technology available for self-driving cars and automotive industry in general will be discussed. This chapter will also cover a comparison between the two mainstream self-driving cars in the market.
- Utilised hardware & software – An in-depth analysis of the hardware components and the software dependencies used throughout the project will be explored in this chapter.
- System Design – A comprehensive breakdown of the hardware segment and the software design will be conducted in this chapter with some sub-topics covering a detailed explanation of the code for a better understanding of the overall system.
- Results – All the final results and resolutions gathered throughout the project will be analysed with a brief discussion on the challenges faced while obtaining these results.
- Conclusion – Once the data has been gathered and the final testings has been conducted, a conclusion will be drawn discussing whether it is possible to implement a self-driving car running on a neural network is capable of being deployed on a simple Raspberry Pi.

1.7 Conclusion

Although the concept of an autonomous car began in the late 1980s, the technology is still in its infancy and only came to the mainstream market in the past few years. Further research and development conducted by technology companies and amateur technology hobbyists provides a valuable insight into the potential of this evolutionary transportation system and a promising start to the revolution of self-driving cars. The success of this project is a result of an in-depth research and development process conducted and shared by an online community of enthusiasts alike. Although the online resources made the project rather straightforward, at various stages of the project many unexpected obstacles presented themselves. Many of these challenges were overcome by creative and unconventional solutions, but car's mechanical issues slightly altered the expected outcome of the project.

2 Existing Technology

2.1 Introduction

Since the launch of the 'Autopilot' system, many other automotive companies have adopted their own versions of the self-driving systems into their vehicles. For research purposes, this project will solely focus on the Tesla's Autopilot system and Google's self-driving car 'Waymo' [5]. Essentially, both these systems exploit the same principles of machine vision and machine learning through neural networks with the primary difference being the way in which they collect the sensory data and how it is processed.

This section will also examine the hardware and software technologies implemented in these commercial vehicles and compare them to the technology that was used in this project.

2.2 Initial Research

2.2.1 Levels of driving automation

Over the years vehicles have been getting smarter and more accessible with state of the art technologies utilising an array of sensors and constant software updates to improve the driving experience of the users. Due to the different types of driving assistances available in the market, the Society of Automotive Engineers (SAE) has offered a useful framework to understand the incremental automation levels. [7]

- **Level 0 – No Automation**:
  The driver performs all aspects of dynamic driving tasks even with some degree of warning assistance.
- **Level 1 – Driver Assistance**:
  Driver and the automated system shares control over the vehicle. The vehicle is able to assist with some functions such as lane keeping, emergency breaking and adaptive cruise control, but the driver still handles accelerating, braking and monitoring of the surroundings. This level is referred to as the "hands on" level.
- **Level 2 – Partial Automation**:
  The so called "hands off" mode is the current level of autonomy being developed by most automakers. The vehicle can assist with steering or acceleration functions and allow the driver to disengage from some of the driving tasks but the driver must always be ready to intervene at any time if the automated system fails to respond.

- **Level 3 – Conditional Automation**:

  In the recent years, vehicle autonomy has been gradually transitioning from level 2 to level 3 "eyes off" automation where the vehicle itself controls nearly all the driving tasks including monitoring of its surrounding environment. The vehicle will handle situations that call for an immediate response, like emergency braking. The driver must still be prepared to intervene within some limited time.

- **Level 4 – High Automation**:

  Incrementing from level 3, level 4 autonomy is capable of steering, braking, accelerating, monitoring the vehicle and roadway as well as responding to events, determining when to change lanes, turn, and use signals. The so called "mind off" self-driving perform all above mentioned driving tasks even if a human driver does not respond appropriately to a request to intervene.

- **Level 5 – Full Automation**:

  At this level of autonomy, a steering wheel is optional as no human intervention is required. An example of this type of self-driving would be a robotic taxi.

2.2.2 Tesla Autopilot

Tesla's Autopilot system is a highly advanced driver-assistance system featuring level 2 autonomy, gradually advancing towards level 3 with each software and hardware revision. This semi-autonomous system offers features such as 'lane centring, adaptive cruise control, self-parking, ability to automatically change lanes without requiring driver steering, and enables the car to be summoned to and from a garage or parking spot' [8]. This new updated system employs an array of sensors ranging from twelve ultrasonic sensors, eight cameras to a long-range front radar system. The cameras provide 360 degrees of visibility around the car up to 250 meters of range, while the ultrasonic sensors complement this vision, allowing for detection of both hard and soft objects at nearly twice the distance of the original system. The forward-facing radar with enhanced processing provides additional data about the environment on a redundant wavelength that can see through heavy rain, fog, dust and even through the cars in front [9]. The Autopilot's vision system is based on monocular forward-looking camera technology which means it's highly improbable that the car can localise itself on a map. This can cause complications for the lane keeping function as GPS data can be quite inaccurate closer to the ground. However, the monocular camera with the help of seven other surrounding cameras can

detect the location and the curvature of the road lane markers, which is more than enough to simply keep the car in its lane and accomplish basic lane change manoeuvres [10].

Figure 1: Tesla Autopilot vision system in action

The use of sensory data to represent the environment around the car is a cost effective and a viable method for achieving level 4 in the scale of driving autonomy. All the visual data collected from the fleet of Tesla cars will be fed into 'Tesla Vision', the automaker's end-to-end image processing software which will continuously train their deep learning neural network for better prediction accuracy [11]. Tesla built the car and gradually implemented features through model upgrades and software revisions to enable driving autonomy.

2.2.3 Google's Waymo

In comparison, Google used a different strategy and built a car around artificial intelligence [10]. The importance of this radical approach is that it allows more flexibility to build a car more suited for level 5 autonomy without a driver onboard. The technology behind the Waymo is more advanced and expensive compared to the Tesla's Autopilot to enable full self-driving autonomy required for robotic taxis of the future. The Waymo exploits the properties of light using a 64-beam Light Detection and Ranging (LiDAR) sensor mounted on top of the car to create a 360-degree dynamic 3D model of its surrounding. This highly accurate reconstruction of its environment allows it to precisely localise itself within the map and with the complement of an array of other sensors allows the neural network to "tracks and predicts movements for all nearby vehicles, pedestrians, and other obstacles, so it's able to plan intelligent paths through

complex highway or urban environments" [10]. The modular design of the system allows it to be implemented on any new car with only few modifications to the car itself.

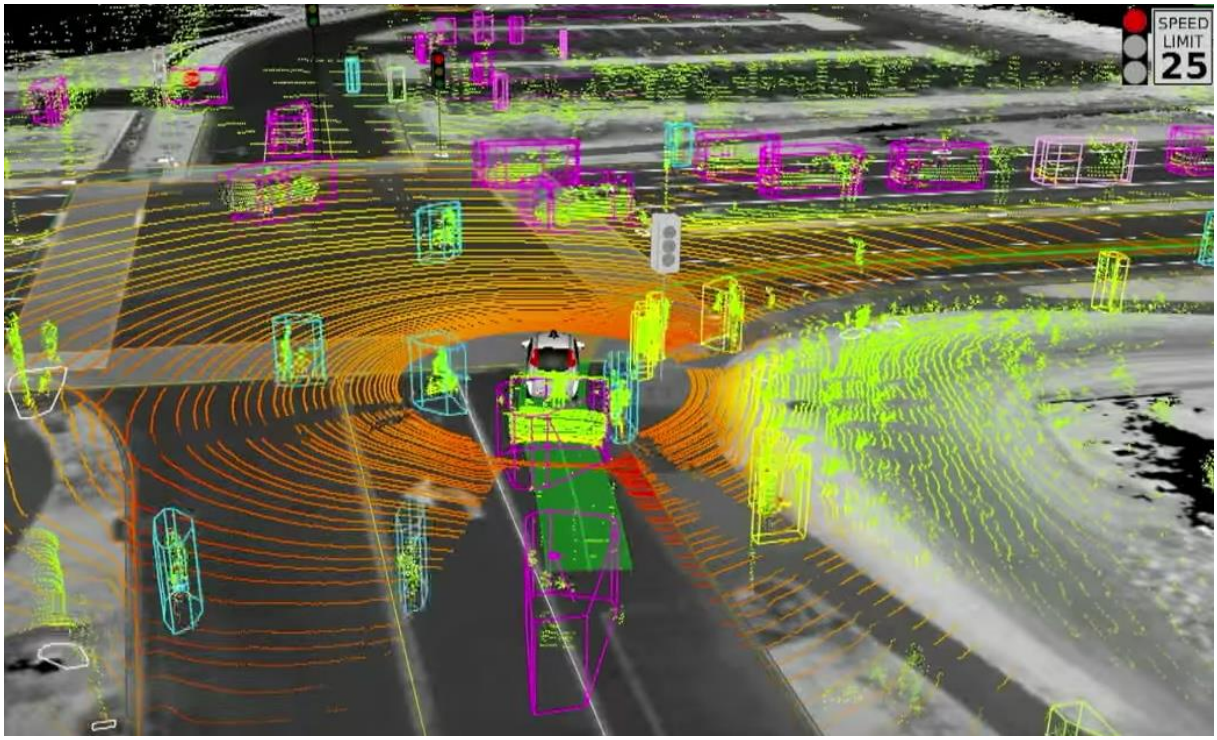


Figure 2: LiDAR representation of its surrounding.

Although the Google's autonomous driving system is more sophisticated and shows more promise towards a generation of true driver-less autonomy, this project chose to implement a version of Tesla's Autopilot system. This is due to the fact that LiDAR sensors of this scale are unavailable in the current market. The complexity of this system and the excessive price tag also makes it impractical for small prototype projects.

2.2.4 Choosing a classifier

In machine learning, there are many types of classification algorithms to choose from, each with their own advantages and disadvantages. The objectives and the needs of the project must be taken into consideration when deciding on a classifier to use. All the machine learning algorithms can be categorised into 3 different types:

- Supervise learning:

  "This algorithm consists of a target / outcome variable which is to be predicted from a given set of predictors. Using these set of variables, we generate a function that map inputs to desired outputs. The training process continues until the model achieves a desired level of accuracy on the training data" [12].

- Unsupervised learning:

  "In this algorithm, we do not have any target or outcome variable to predict / estimate. It is used for clustering population in different groups, which is widely used for segmenting customers in different groups for specific intervention" [12].

- Reinforcement learning:

  "Using this algorithm, the machine is trained to make specific decisions. It works this way: the machine is exposed to an environment where it trains itself continually using trial and error. This machine learns from past experience and tries to capture the best possible knowledge to make accurate business decisions" [12].

Most autonomous cars such as Tesla and Waymo adopted the reinforcement learning method where the cars are collectively being driven around for millions of miles to gather data from real world experiences to train their deep convolutional neural network. It was clear that neither unsupervised nor reinforcement learning methods were appropriate for this project as it is impractical to drive the prototype toy car around for hundreds of miles to gather data. Therefore, the supervised learning method was chosen as the basis for this project's machine learning algorithm.

After choosing the suitable category, the required machine learning algorithm must be chosen. There are hundreds of different types of algorithms available for object recognition based on all kinds of Mathematical Models [13]. Deciding on the type of learning category at the beginning of project reduces the number of algorithms available which are only applicable to your needs.

The most common types of classification algorithms in supervised learning are listed as follows:

1. Linear Classifiers: Logistic Regression, Naive Bayes Classifier
2. Support Vector Machines
3. Decision Trees
4. Random Forest
5. Neural Networks
6. Nearest Neighbour

From the initial research it was clear that most of these classification algorithms would not be suitable for this project. With respect to the table outlined below and the background knowledge from the Artificial Intelligence module, Neural Networks was preferred as the classifier of choice.

Table 1: Comparison of supervised learning methods [14]

Algorithm	Average predictive accuracy	Training speed	Prediction speed	Performs well with small number of observations?	Automatically learns feature interactions?
Linear regression	Lower	Fast	Fast	Yes	No
Logistic regression	Lower	Fast	Fast	Yes	No
Naive Bayes	Lower	Fast (excluding feature extraction)	Fast	Yes	No
Decision trees	Lower	Fast	Fast	No	Yes
Random Forests	Higher	Slow	Moderate	No	Yes
Neural networks	Higher	Slow	Fast	No	Yes

2.2.5 Neural Networks

"An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurones) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process." [15].

Neural networks are made up of few dozens to hundreds or even thousands of interconnected 'nodes' which are organised into series of layers containing an 'activation function'. Data is introduced to the neural network via the 'input layer', which is then processed through a network of hidden layers. "The connections between one unit and another are represented by a number called a 'weight', which can be either positive (if one unit excites another) or negative (if one unit suppresses or inhibits another). The higher the weight, the more influence one unit has on another." [16]. After the data is processed, the final result is delivered through the 'output layer' for predictions.

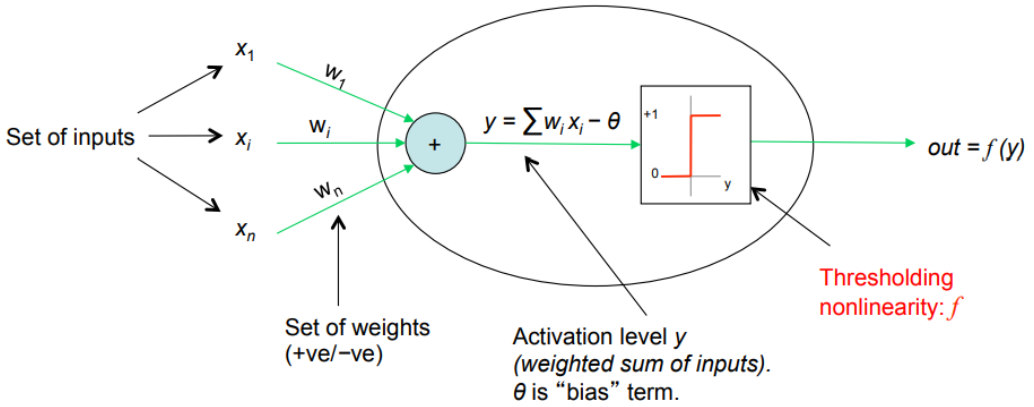


Figure 3: Input nodes processing through activation function to produce a single output

The main advantage of neural networks is that once its trained, the trained parametric file can be loaded on to any program for fast and accurate predictions. There are 38,400 (320×120) nodes in the input layer and 32 nodes in the hidden layer. The number of nodes in the hidden layer is chosen arbitrarily. There are four nodes in the output layer, where each node corresponds to the steering control instructions [17]. To find the minimum of a function, a first-order iterative optimisation algorithm known as the 'gradient descent' is used, which reduced the 38,400 input nodes into just 4 outputs.

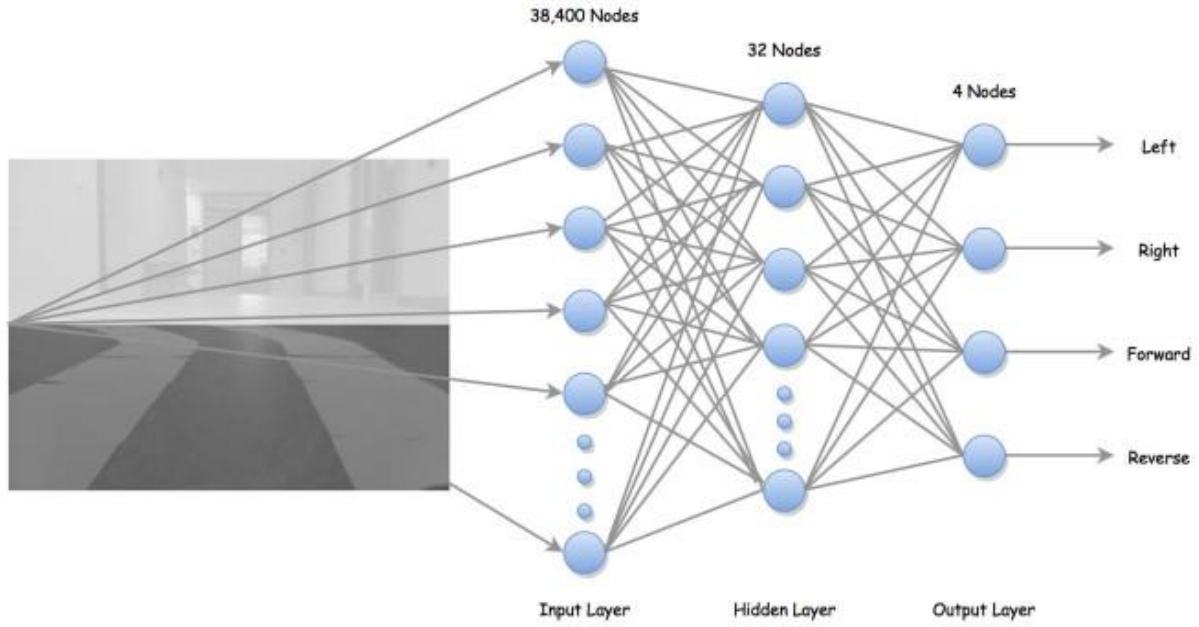


Figure 4: Neural network layout used in prototype self-driving car [17]

This neural network is trained using the OpenCV software library with the backpropagation method as a gradient computing technique, which will be discussed in the next sub-chapter.

2.2.5.1 Backpropagation

The most basic form of an activation function is a simple binary function that has only two possible results; 0 and 1. This is called a 'perceptron' where the function returns a 1 if the input is positive or 0 if the input is negative [18]. Since this project requires at least 4 different outputs: forward, reverse, left and right, a multi-layer perceptron must be implemented.

The 'cost' function in machine learning is an important concept as it measures how far away a particular solution is from an optimal solution to the problem to be solved. Learning algorithms search through the solution space to find a function that has the smallest possible cost. Minimizing this cost using gradient descent for the class of neural networks called multilayer perceptrons (MLP), produces the backpropagation algorithm for training neural networks. [19].

Backpropagation is an efficient method of computing gradients in neural networks. This is not a learning method, but rather a reliable computational trick which is often used in learning methods [20]. The backpropagation algorithm uses a computed output error to change the weight values in backward direction. To get this net error, a forward propagation phase must have been done before. This means feedforwarding the values, where the input at the input layer travels from the input to the hidden and from hidden to the output layer.

While propagating in forward direction, the neurons are being activated using the sigmoid activation function.

The formula of sigmoid activation is: $f(x) = \frac{1}{1+ e^{-input}}$ [21]

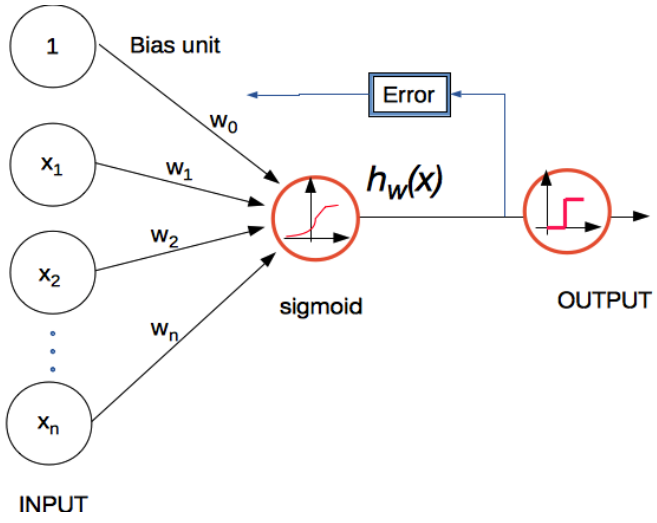


Figure 5: Errors being backpropagated after the sigmoid function. [22]

2.2.5.2 Object Detection Algorithm

"Object detection is commonly referred to as a method that is responsible for discovering and identifying the existence of objects of a certain class. An extension of this can be considered as a method of image processing to identify objects from digital images." [23]. The easiest and the least common way of detecting object is by simply classifying the object in the image according to its colour. However, from research and experiments it was found that this colour-coded approach was not ideal due to the changes in ambient lighting conditions would have a detrimental effect on the accuracy of the classifier.

Paul Viola and Michael Jones proposed the first real-time object detection framework in 2001 called the 'Viola-Jones object detection framework. Although it was possible to detect many object classes using this method, it was primarily aimed for face detection. The Viola-Jones algorithm gained a lot of attention for object detection in real-time applications due to its robustness, accuracy and speed [24].

Since this project requires detection of more specific targets, including non-human objects such as road signs and traffic lights, a more sophisticated method was therefore required. This problem was addressed by the so-called Haar-like features, developed by Viola and Jones on the basis of the 'General Framework for Object Detection' [25] proposed by Papageorgiou et. al in 1998 [23]. Haar-like features are digital image features used in object recognition. A Haar-like feature considers adjacent rectangular regions at a specific location in a detection window, sums up the pixel intensities in each region and calculates the difference between these sums. This difference is then used to categorise subsections of an image. The basic idea behind Haar basis function is that most objects have regularities that match Haar features. For example, in face detection, the Haar cascade classifier detects the eye regions being darker than the upper-cheeks, or the nose bridge region being brighter than the eyes.

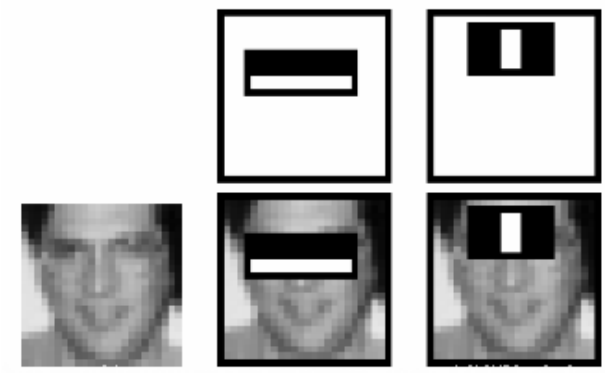


Figure 6: Example of edge and line features on a Haar-like feature facial recognition [26]

During the learning phase, a cascade of weak detectors is trained so as to gain the desired recall rate using AdaBoost [27]. It's called a weak classifier because it's unable to classify images on its own. The cascade classifier is simply a collection of multiple stages of filters whose objective is to detect Haar-like features. Cascading Classifiers are trained with several hundred "positive" sample views of a particular object and arbitrary "negative" images of the same size. After the classifier is trained it can be applied to a region of an image and detect the object in question [28].

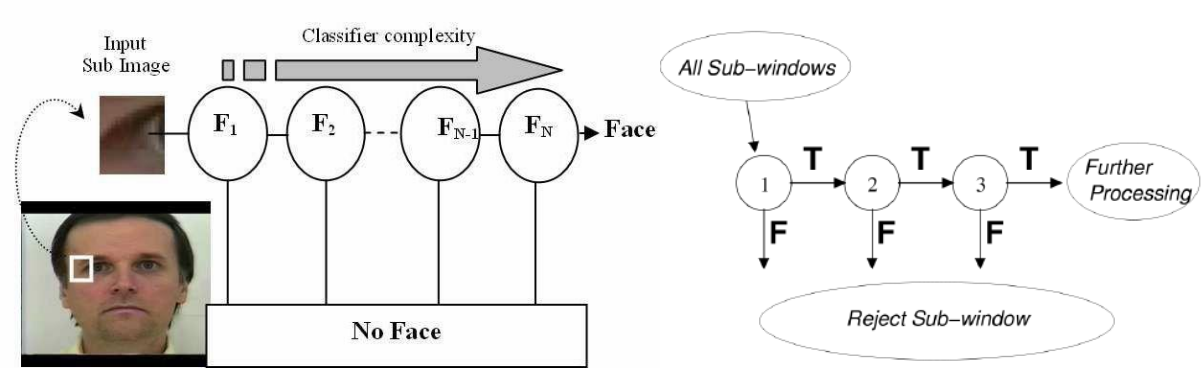


Figure 7: Cascade stages of architecture [29]

AdaBoost is a learning algorithm that helps to select the best features and to train classifiers that use them. The output of the other learning algorithms ('weak learners') are combined into a weighted sum that represents the final output of the boosted classifier [30].

So, when all of these things come together, we get a cascade classifier for detecting Haar-like features. The OpenCV software library provides trainers and classifiers for Haar-cascade detection. A breakdown of how the classifier was trained will be discussed in further detail in section 4.3.5.

2.2.6 Monocular vision

The Raspberry Pi computer module used in this project only supports one on board camera. Using two USB web cameras would be impractical as it would add extra weight and complexity to the design. This project adapted a geometry model of detecting distance to an object using monocular vision method proposed by Chu Ji, Guo Li and Wang (2004) [31].

"Monocular vision is vision in which both eyes are used separately. By using the eyes in this way, as opposed by binocular vision, the view is increased, while depth perception is limited" [32]. Depth perception is the visual ability to perceive the world in three dimensions and the distance of an object. Monocular cues such as motion parallax, depth from motion, perspective, etc. provides depth information when viewing a scene with one eye [33].

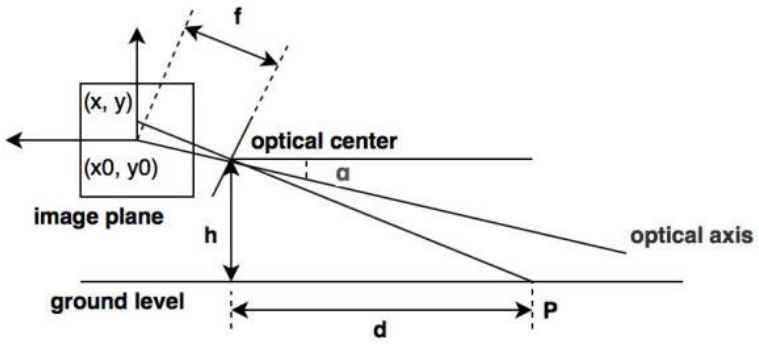


Figure 8: The geometry model of detecting distance in an image [17].

"P is a point on the target object; d is the distance from optical centre to the point P. Based on the geometry relationship above, formula (1) shows how to calculate the distance d. In the formula (1), f is the focal length of the camera; α is camera tilt angle; h is optical centre height; (x0, y0) refers to the intersection point of image plane and optical axis; (x, y) refers to projection of point P on the image plane. Suppose O1 (u0, v0) is the camera coordinate of intersection point of optical axis and image plane, also suppose the physical dimension of a pixel corresponding to x-axis and y-axis on the image plane are dx and dy. Then:

$$1)\ d = \frac{h}{\tan(\alpha + \tan^{-1}(\frac{y - y_0}{f})}$$

$$2)\ u = \frac{x}{dx} + u_0 \qquad\qquad v = \frac{y}{dy} + v_0$$

Let x0 = y0 = 0, from 1) and 2):

$$3)\ d = \frac{h}{\tan(\alpha + \tan^{-1}\frac{(v - v_0)}{a_y})} \qquad\qquad a_y = \frac{f}{dy}$$

v is the camera coordinates on y-axis and can be returned from the object detection process. All other parameters are camera's intrinsic parameters that can be retrieved from camera matrix. The camera matrix can be obtained by using the camera calibration function built into OpenCV. Ideally, a_x and a_y has the same value. Variance of these two values will result in non-square pixels in the image. The matrix below indicates that the fixed focal length lens on pi camera provides a reasonably good result in handling distortion aspect.

$$\begin{bmatrix} a_x = 331.7 & 0 & u_0 = 161.9 \\ 0 & a_y = 332.3 & v_0 = 119.8 \end{bmatrix}$$

Figure 9: Camera matrix returned after calibration

The matrix returns values in pixels and h is measured in centimetres. By applying formula (3), the physical distance d is calculated in centimetres." [17]

Order	1	2	3	4	5	6	7
Actual value (cm)	15	20	25	30	35	40	45
Measured value (cm)	15.5	19.7	24.1	27.5	32.0	35.2	39.0

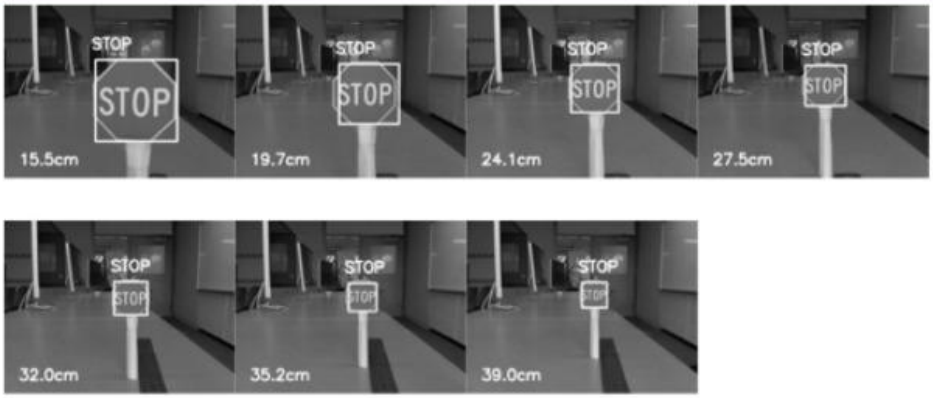


Figure 10: Experimental results of detecting distance using pi camera [17]

2.2.7 Conclusion

This section briefly discussed the main technologies, advanced machine learning algorithms and geometric maths behind the development of this prototype self-driving car. The two main competitors with self-driving technology are Tesla with their Model S and Google's Waymo, both of which uses entirely different systems of navigation. Tesla incorporates an array of sensors and cameras to sense the world around it whereas the Waymo utilises a light detection and ranging (LiDAR) sensor to localise itself and to visualise the environment around it. For this project, supervised learning method with the machine learning algorithm MLP neural networks was chosen for training the car to drive itself on the track. It was also decided that the backpropagation method would be used to efficiently compute the gradient in neural network to reduce the cost of finding an optimal solution. For object detection, Haar-like feature detection developed by Viola and Jones was used. The cascade classifier was trained using thousands of positive and negative images to increase its accuracy. The object detection was implemented in conjunction with monocular vision method to calculate the distance between the car and the road sign. From this research it was concluded that MLP artificial neural network is the most straightforward method of machine learning algorithm to implement on a small prototype projects. However, it was deduced that reinforcement learning with Convolution Neural Network algorithm would be the best choice for real world autonomous cars.

3 Utilised Hardware & Software

This project consisted of many hardware and software components working collectively together to successfully achieve the self-driving capabilities. This section will cover an analysis of each of these components specifications and functions in detail.

3.1 Hardware

3.1.1 Raspberry Pi 3 Model B

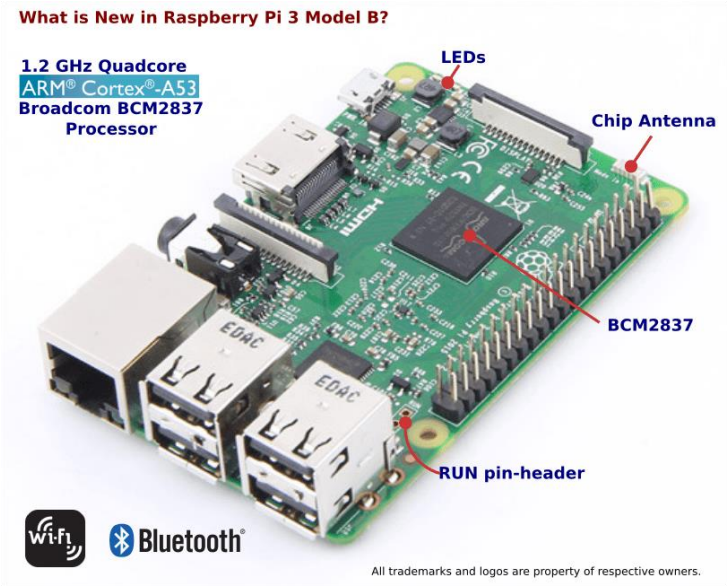


Figure 11: Raspberry Pi 3 Model B [34]

The Raspberry Pi is a credit card sized single board computer (SBC) developed by the Raspberry Pi foundation. Several generations of Raspberry Pis have been released. All models feature a Broadcom system on a chip (SoC) with an integrated ARM compatible central processing unit (CPU) and on-chip graphics processing unit(GPU) [35]. Raspberry Pi runs Debian based **GNU/Linux** operating system Raspbian. From the research it was found that the Broadcom BCM2837 SoC with a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor, and 1GB of RAM offered enough processing power to process the video feed through a neural network without the help of a full-size computer. The Raspberry Pi is rated for 5.1V and 2.5A power input, but from experiments it was found that a dual output 5V / 2.4A power bank could support both the Raspberry Pi and the Adafruit Motor HAT.

3.1.2 Pi Camera

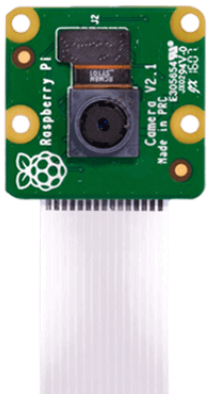


Figure 12: PiCamera

The Raspberry Pi camera module is perfect for applications where the size and weight are important such as drones or small mobile robots. The 5 megapixels sensor is capable of supporting 1080p30, 720p60 and 640x480p60/90 video and a maximum resolution of 2592 x 1944 pixels for static images [36]. Although this cheap generic camera offered great specifications for the task in hand, it was found that there are several technical shortcomings that would affect the overall performance of the object detection and tracking.

- "The camera data stream is processed by a Broadcom GPU running proprietary firmware made with proprietary tools referencing proprietary specifications. In short, there is no current possibility for user modification, enhancement, or emulation of the camera
- No manual exposure controls. Only various auto-exposure modes are available.
- Rolling shutter. While this limitation, versus a global shutter, is to be expected of an inexpensive camera, it distorts moving subjects in video." [37]

From the initial testing phase, it was found that this camera could perform object detection exceptionally as long as the object is steady and centre of the camera sensor with a close range. This performance dramatically deteriorated when the car started moving at a fast pace.

Further research showed that the 'Pixy (CMUcam5) Smart Vision Sensor' camera by Charmed Labs was designed specifically for object detection and tracking purposes. "This camera is fast enough to detect and track objects around it, but the object should be coloured in bright solid colour to make a good distinction from the surroundings." [36].

Figure 13: Pixy CMUcam5

The large field of view of 75 degrees horizontally and 47 degrees vertically with a higher resolution sensor would mean this camera would be an ideal candidate for object detection applications. The developer's website declares that the camera can learn to detect objects that you teach it and is able to output what it detects 50 times per second. [38]

3.1.3 Adafruit motor HAT

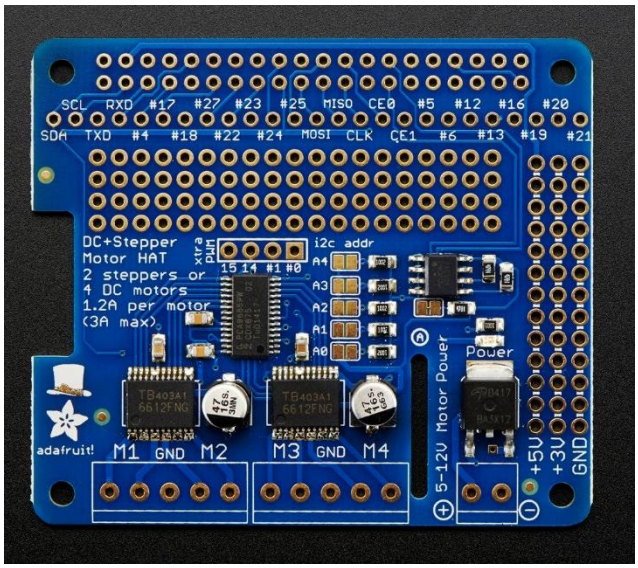


Figure 14: Adafruit Motor HAT

In the component testing phase, experiments were conducted on the car motors to find the power requirements of the motors. Using a multimeter, it was found that the rear motor requires at least 1 amp of current and 5Vs to run steadily.

Since the Raspberry Pi can only output roughly 300 mA of current from the 5V pin, it would be dangerous to connect the motor directly to the pulse width modulation (PWM) pin as the excessive current drawn could damage the Raspberry Pi. Therefore, it was recommended to use an external board for interfacing high powered motors to the Raspberry Pi.

The Adafruit motor HAT (Hardware Attached on Top) add-on was the ideal candidate for this project as it can drive up to 4 DC motors with full PWM speed control. It "use a fully-dedicated PWM driver chip onboard to control motor direction and speed. This chip handles all the motor and speed controls over I^2C. "The Inter-Integrated Circuit (I^2C) communication protocol is a protocol intended to allow multiple "slave" digital integrated circuits ("chips") to communicate with one or more "master" chips. Like the Serial Peripheral Interface (SPI), it is only intended for short distance communications within a single device." [39]. Motors are controlled by TB6612 MOSFET driver: with 1.2A per channel current capability" [40]. The easy setup and the straightforward python code made the implementation of this add on very simple and convenient.

3.1.4 Ultrasonic Sensor

Figure 15: HC-SR04 Ultrasonic Sensor

Ultrasonic sensors are devices that can detect the presence of an object by measuring the distance between the sensor and the object using sound waves. The transmitter indicated by 'T' sends out a sound wave at a specific frequency that reflects back from the object in front of it which is then picked up by the receiver indicated by the letter 'R'. The sensor is able to record the elapsed time between the sound wave being generated and the sound wave bouncing back to calculate the distance between the object using the following simple formula.

$$Distance = \frac{speed\ of\ sound\ \times\ time\ taken}{2}$$

This very simple, low powered device is ideal for variety of object detection applications. "It can provide between 2cm – 400cms of detection range with an accuracy up to ±3mms.

Basic operation principle:

1. I/O port TRIG trigger ranging to at least 10us high level signal;
2. The module automatically sends eight 40kHz square wave, automatically detects whether a signal returns;
3. A signal return to a high output through the I/O port ECHO high duration of ultrasound wave from the transmitter to the time of the return." [41]

This principle can be better observed in the timing diagram below

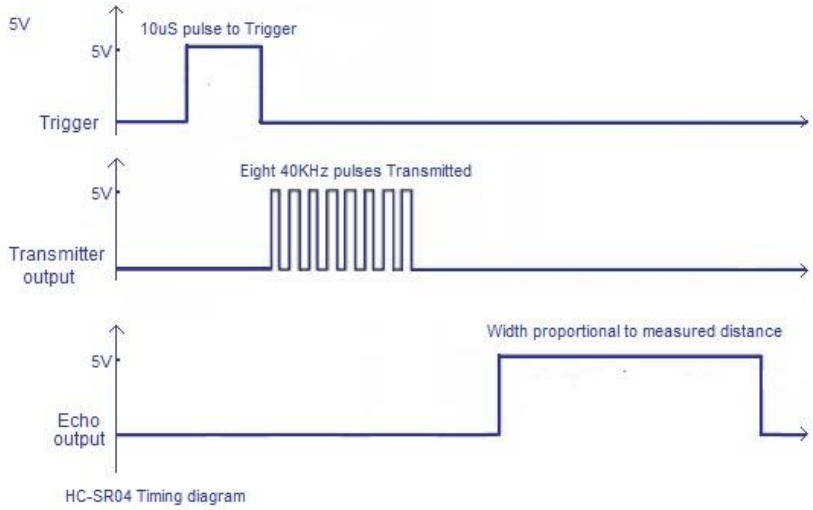


Figure 16: Ultrasonic Sensor timing diagram

3.1.5 Power bank

Finding a suitable power source with the required output voltage and current to facilitate both the Raspberry Pi and the motor HAT was particularly difficult. Using two separate battery packs would add too much weight to the car and using a wired power supply would be impractical for this purpose.

Figure 17: Ravpower RP-PB19 power bank

This 16,000 mAh power bank from Ravpower offered the best compromise for this problem as it is capable of simultaneously providing two separate power outputs with 2.4 amps and 2.1 amps with 5Vs each. The high capacity, light weight power pack is able to power both the Raspberry Pi and the motor HAT reliably for a long period of time while keeping the weight of the overall car to a minimum.

3.2 Software

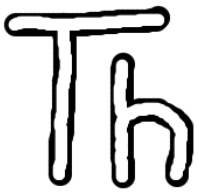
3.2.1 Raspbian Stretch

Raspbian is a Debian-based computer operating system for Raspberry Pi. Debian 9 (Stretch) is the latest version of Raspbian released on June 2017. The update was released with many bug fixes and optimisations over the predecessor [42]. The main reason for upgrading the operating system to Debian 9 was that the OpenCV software platform had problems installing and running on the version 8. This was due to the fact that the new version of OpenCV required libraries and packages that were only available in Debian 9 Stretch.

3.2.2 PyCharm IDE

PyCharm is an Integrated Development Environment (IDE) used in computer programming, designed specifically for the Python language. This IDE is favoured by many python developers for the great features it has to offer. PyCharm provides smart code completion, code inspections, on-the-fly error highlighting and quick-fixes, along with automated code refactoring and rich navigation capabilities. It also offers a collection of tools out of the box: an integrated debugger and test runner; Python profiler; a built-in terminal; and integration with major VCS and built-in Database Tools [43]. The free community edition of this IDE is preferred by many due to its versatility between cross platforms while providing the same functions and features as you would expect from a high-end professional IDE. For this project, PyCharm was only used on the computer running Windows operating system for writing and debugging the code and also for collecting and processing the training data through the neural network. Since PyCharm was not available for 'ARM' based computers, Thonny Python IDE was used for debugging and running the program inside the Raspberry Pi.

3.2.3 Thonny IDE



Thonny is a Python IDE for beginners that comes with the Raspberry Pi operating system. It is a very basic development environment with minimal tools and features. The reason for choosing this IDE was because it was easy to utilise and it supports different ways of stepping through the code, step-by-step expression evaluation for a faster debugging process. Since this software runs on the Raspberry Pi, any changes to the code can be instantly executed to check if the script performs as intended. One of the main drawbacks of using this IDE was that it only supported Python version 3.6 whereas majority of the project was developed in Python 2.7. Therefore, after each code modification, the script had to be executed through the terminal.

3.2.4 OpenCV

OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision applications. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. The library has more than 2500 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 and so on [44].

It supports a wide variety of programming languages such as C++, Python, Java, etc., and is available on different platforms including Windows, Linux, OS X, Android, and iOS. OpenCV-Python is a library of Python bindings designed to solve computer vision problems. OpenCV-Python makes use of Numpy, which is a highly optimized library for numerical operations with a MATLAB-style syntax [45].

For this project, OpenCV version 3.4 was chosen as it was the latest version of OpenCV available at the beginning of the project. Having failed to install the library using the normal 'pip install opencv-python' command through the terminal, the library had to be manually compiled with the help of an installation tutorial by Adrian Rosebrock [46].

Things to note when manually compiling OpenCV:

- Do not create a Python virtual environment if the sole purpose of your Raspberry Pi is to run this project. This will eliminate a lot of confusion and inconveniences throughout the project.
- The tutorial recommends using the command 'make -j4' which basically means utilise all 4 cores of the processor in the Raspberry Pi. From personal experience, it was found that this method fails the compilation after 90% of the process has been completed. Therefore, replace the 'make -j4' with 'make' command which would increase the compilation time from just 2 hours to over 5 hours but successfully completes the task.
- This time-consuming process drastically increases the temperature of the processor, so it is recommended to attach some heat sinks to the IC chips.
- It is also recommended that OpenCV 3+ version should be installed on Python 2+ as Python 3+ version doesn't fully support the OpenCV libraries at present.

4 System Design

4.1 Introduction

The inspiration for this project came from Zheng Wang's 'Self Driving RC Car' [17] project, in which a toy RC car was converted to a self-driving car with an artificial neural network. Although both of these projects are very similar, the core implementation of the two system designs were distinctly different. The main difference between the two systems being that the neural network on Wang's self-driving car was hosted on a computer for faster calculations and predictions, whereas this project tried to implement everything on the standalone Raspberry Pi itself.

Wang's Self-driving car design:

The Raspberry Pi along with the PiCamera collected the training data which was then processed through the neural network on a host computer. The client Raspberry Pi streamed the live video feed and the ultrasonic data to the computer via Wi-Fi at a rate roughly 10 frames per second. This meant that in actual driving situations, predictions were generated about 10 times a second, so a highly reliable, high speed Wi-Fi connection was needed to reduce the time lag of the frames. The car was driven by the original remote controller of the car which was wired on to an Arduino. The Arduino was connected to the computer and once a driving predication was made, the computer sends a command to the Arduino via the USB serial interface, which intern sends a signal to the remote control which is the transmitted to the car's onboard control circuit via radio waves to run the motors.

Since the toy car used in this project did not have an onboard motor control circuit and a remote control, and also to reduce the complexity of the design, a standalone method was chosen. This meant that the Raspberry Pi was solely responsible for processing and calculating predictions from the artificial neural network. This design greatly reduced the time lag of the video feed but the overall performance of the prediction rate was counteracted by weak computing power of the Raspberry Pi.

The changes made to the design and the code negatively affected some aspects of this project and caused issues with some of the driving functions which will be discussed in further detail in the following sub-chapters.

4.2 Hardware Design

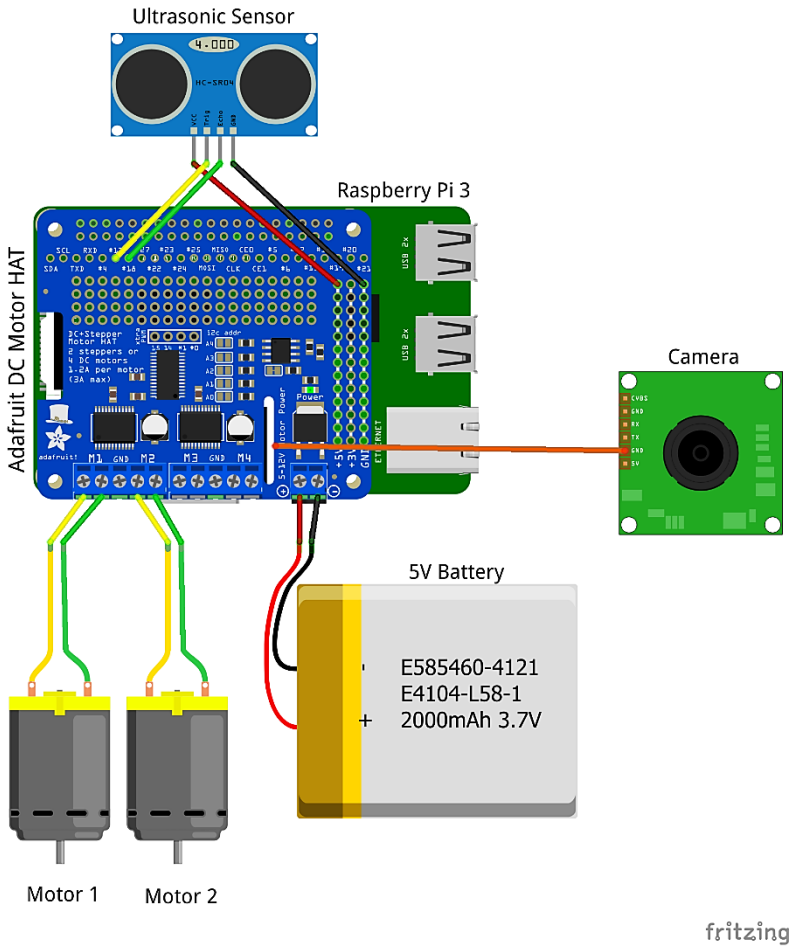


Figure 18: Hardware Layout of the system

The hardware layout of this system is very straightforward as there is only one front obstacle avoidance system and one camera for object detection in the design. Utilising the motor HAT from Adafruit eliminated the need for a custom-built motor driver circuit which would require extra space to fit in. This compact and light weight design was ideal for the small toy car that was being used for this project. Although the ultrasonic sensor is connected directly to the motor HAT prototyping area in the above schematic, it had to be built on a separate circuit which will be discussed in the next sub-chapter.

Motor 1 and motor 2 connected directly to the HAT represents the rear and front wheel motors respectively. The Adafruit motor HAT library provides a very simple way to program the motors independently of each other.

```
rearMotor = mh.getMotor(1)          # Defining rear motor is assigned to node M1
rearMotor.setSpeed(150)     # Sets motor speed between 0 (off) & 255 (max speed)
rearMotor.run(Adafruit_MotorHAT.FORWARD)    # Run rear motors in forward
```

The PiCamera module connects to the Raspberry Pi via a flat flex cable which runs through the motor HAT circuit. Great care must be taken when handling the camera module as most of the electronic components are susceptible to damage from static electricity. It should also be noted that when connecting and disconnecting the camera module, the Raspberry Pi should be turned off and disconnected from the power source to reduces the chances of damaging both the camera module and the RPi itself.

The above schematic shows a generic Li-Ion battery connected to the motor HAT, but in reality, both the Raspberry Pi and the motor HAT were powered by a single power bank with dual outputs mentioned in section 3.1.5.

4.2.1 Ultrasonic sensor schematic

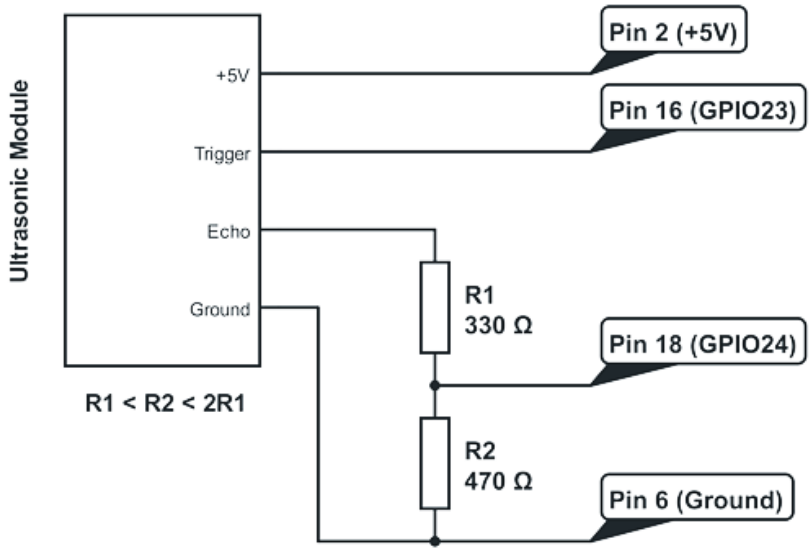


Figure 19: Ultrasonic sensor wiring diagram [47]

The input pin on the module is called the "trigger" as it is used to trigger the ultrasonic pulse. Ideally it wants a 5V signal but it works just fine with a 3.3V signal from the GPIO. The output "echo" on the other hand cannot be connected directly to the Raspberry Pi. The output pin is low (0V) until the module has taken its distance measurement. It then sets this pin high (+5V) for the same amount of time that it took the pulse to return. Since the GPIO voltage levels are 3.3 V and the lack of over-voltage protection on the Raspberry Pi, means that providing a 5V input could potentially damage the board internally. In order to ensure the RPi only gets an input of 3.3V, a basic voltage divider is implemented using two resistors.

$$Vout = Vin\ (\frac{R2}{R1+R2}) \qquad Vout = 5\left(\frac{470}{330+470}\right) = 3V$$

4.3 Software Design

4.3.1 Data Collection

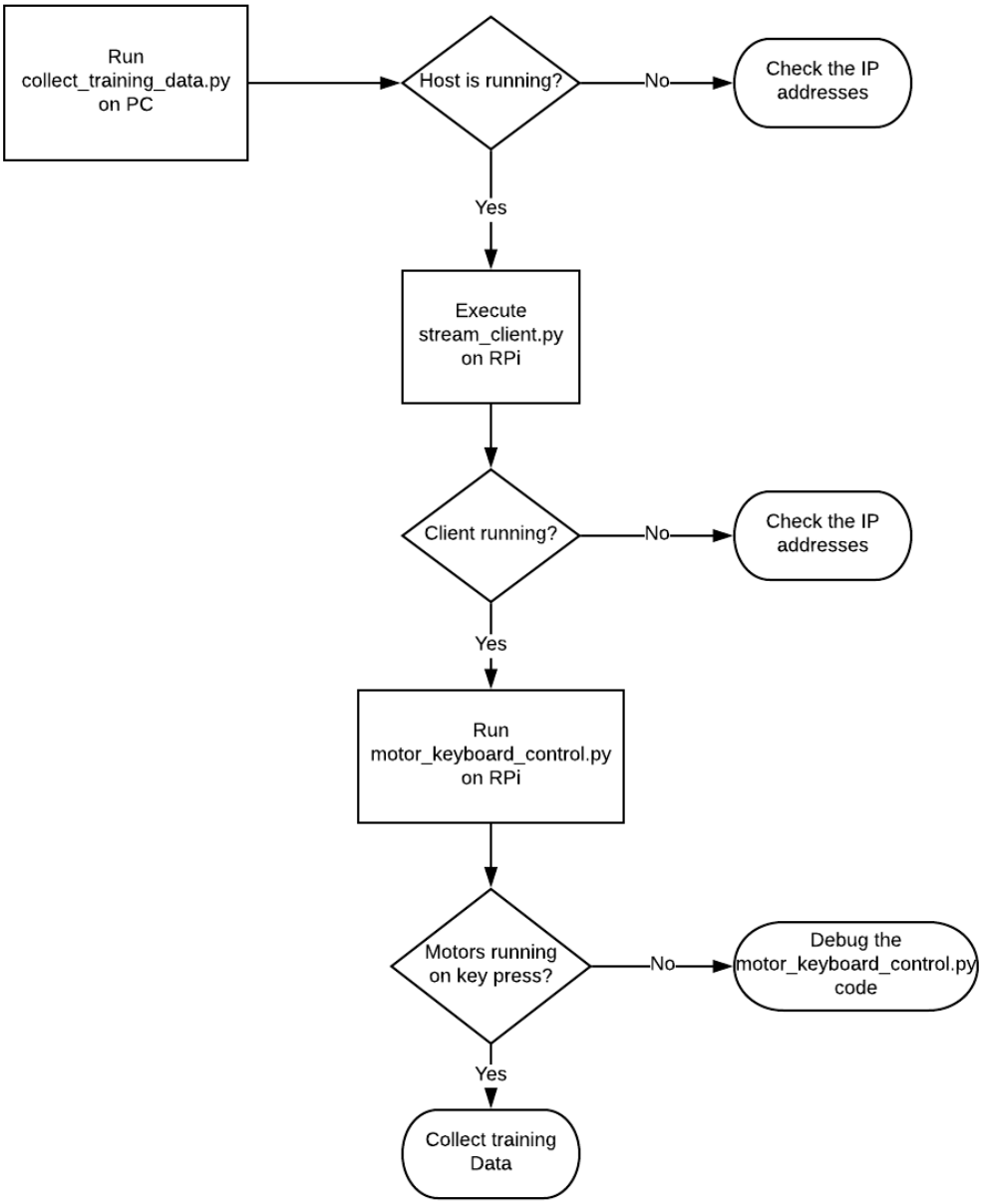


Figure 20: 'Collect_training_data.py' script flowchart

Since the computer has more processing power, it was decided to stream the video data of the track to the PC to be collected, processed and to train the neural network in a short amount of time. The Raspberry Pi streamed the video feed onto the PC via the local Wi-Fi network on the same port. The 'Collect_training_data.py' script received this data and started the frame capturing sequence after the car controlling keys were pressed. Some complications arose when trying to send keypress events from the PC to the RPi (Raspberry Pi) through the network and due to timing restrictions, a simple solution was used, where the car would be controlled using

the wireless keyboard connected to the Raspberry Pi, while simultaneously pressing the same keys on the PC keyboard to capture the image frames. This ensured that data was collected in synchronous with the movement of the car.

4.3.1.1 Brief Explanation of data collecting script

On execution of 'Collect_training_data.py' script, it tries to establish a connection with the Raspberry Pi on the same local network. At this point, 'stream_client.py' script should be run on the Raspberry Pi to complete the connection. Once the connection has been made, the script running on the PC decodes the information and converts the data into an 8-bit data type grey scale video feed. On image capture, the frame is cropped and converted to a 120 x 320 Numpy array. The lower half of the image to make it easier for the training script to only detect the track in the image. This helps to reduce the training time and to improve the accuracy of the neural network.

```
#Converting the image to an 8-bit grey scale image
image = cv2.imdecode(np.fromstring(jpg, dtype=np.uint8),
cv2.IMREAD_GRAYSCALE)

# select lower half of the image
roi = image[120:240, :]
```

The Numpy array is then reshaped into a one row array and the train image is paired with a train label.

```
# reshape the roi image into one row array
temp_array = roi.reshape(1, 38400).astype(np.float32)
```

PyGame library is used to create a window in which the keypress events can be detected as this cannot be done through the terminal.

```
# Creating a window of size 100 x 100
pygame.display.set_mode((100, 100), HWSURFACE | DOUBLEBUF |
RESIZABLE)
```

On a keypress event, the image capture sequence starts and temporarily saved in an Numpy.vstack array, which means taking a sequence of arrays and stacking them vertically to make a single array.

```
# Temporarily saving images and labels to an array of keypress
image_array = np.vstack((image_array, temp_array))
label_array = np.vstack((label_array, self.k[1]))
```

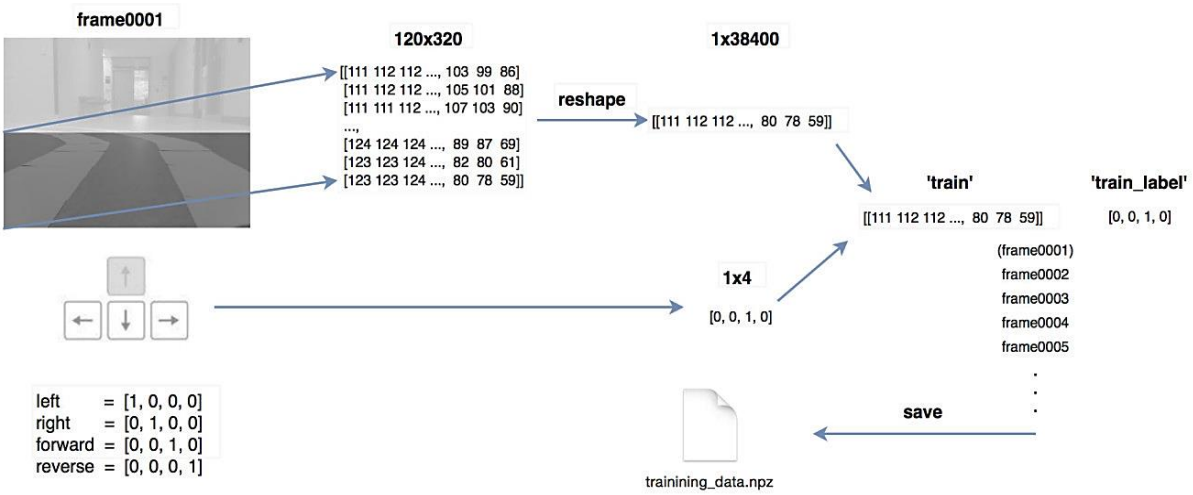


Figure 21: Illustration of data collection and saving process [17]

Finally, all paired image data and labels are saved to a folder called 'training_images' and the training data is saved to a folder called 'training_data' as .npz files. These Numpy.savez files are then used to train the neural network which will be examined in the next section.

4.3.2 Training the Multilayer Perceptron

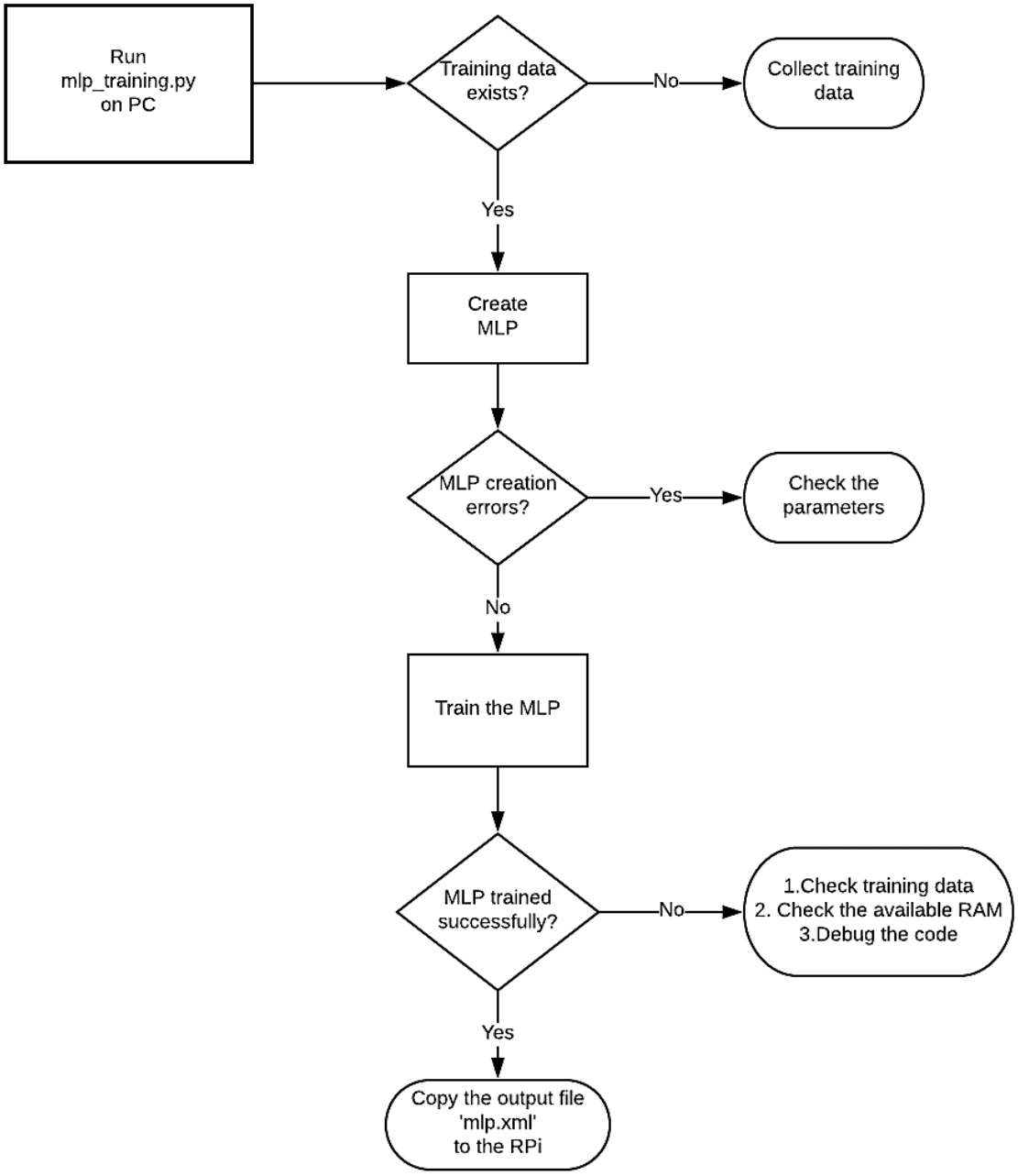


Figure 22: 'mlp_training.py' script flowchart

As mentioned previously, all the neural network training will be executed on the computer due to its high processing capacity. Once the training data has been collected from section 4.3.1, two folders are created; 'training_images' and 'training_data', which contains all the necessary information to train the multilayer perceptron we have created. Once the training has completed, the terminal will show the percentage of the accuracy of the neural network and it will also create a xml file containing training model data which will be used to drive the car in the final stage.

Things to note when running the 'mlp_training.py' script:

- The accuracy of the neural network depends on the amount of data you have gathered. To achieve a higher accuracy, create a longer track and collect data as much as possible.
- Processing a large amount of data requires an excessive amount of memory. Since the 32-bit version of Python 2.7 has an upper memory limit of 2GBs of RAM, the program will terminate after a couple of seconds with a 'MemoryError'. To overcome this issue, the script was modified to run on 64-bit version of Python 3.6 which eliminates the upper memory limit, but in doing so, most of the other libraries became incompatible with this version of Python. To resolve this issue, all the libraries had to be reinstalled from a repository of unofficial windows binaries for python extension packages [48].

4.3.2.1 Brief explanation of MLP training script

On execution, the script loads the training data from the Numpy.savez (.npz) files which we have collected in the previous section. Once the training data has been loaded, it then creates the multilayer perceptron using the parameters we have set.

```
# defines the layer size
layer_sizes = np.int64([38400, 32, 4])
# creates the artificial neural network multilayer
perceptron
model = cv2.ml.ANN_MLP_create()
# setting training method to Backpropagation
model.setTrainMethod(cv2.ml.ANN_MLP_BACKPROP)
# sets the symmetrical sigmoid function as the activation
function
model.setActivationFunction(cv2.ml.ANN_MLP_SIGMOID_SYM, 2,1)
```

As we have discussed earlier, the neural network is trained using the backpropagation method to reduce the cost by creating a gradient descent. The trained prediction is then tested against the true labels to calculate the accuracy of the MLP. With the amount of data gathered, the neural network was trained within couple of minutes with an accuracy of 64.77%. It was observed that every time the mlp_training script was run; the accuracy of the output model was drastically different from the previous one. The lack of training data makes the accuracy of this model highly unreliable which would ultimately have a significant effect on the final result of the project. The output file 'mlp.xml' is then used in the final autonomous driving script to predict the driving behaviour of the model.

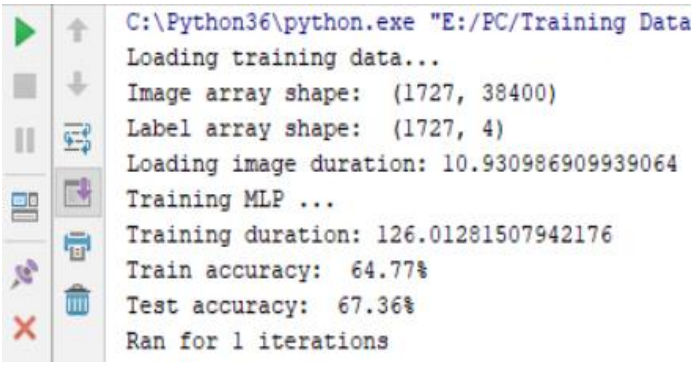


Figure 23: MLP Training 1st attempt

On the first attempt, the neural network generated a model with a reasonable 64.77% accuracy. This model was saved to the Raspberry Pi and the MLP training was executed again.

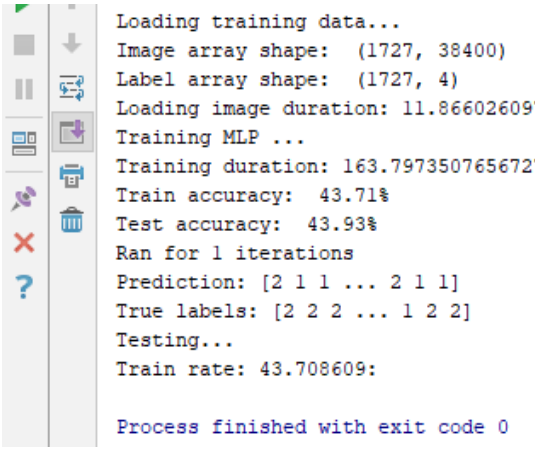


Figure 24: MLP training 2nd attempt

Keeping all the parameters unchanged, the script was executed again with the same training data as the first attempt. We can clearly observe that there has been a significant reduction in the accuracy of the trained model, almost 25% decrease.

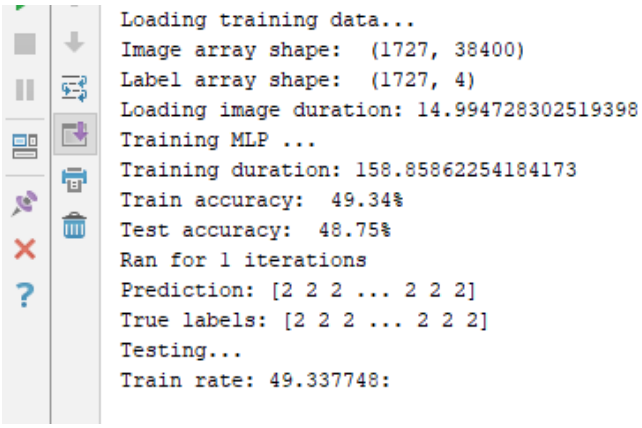


Figure 25: MLP training 3rd attempt

After the 3rd attempt it was clear that this multilayer perceptron was in fact highly unreliable and unsatisfactory for the purpose of this project. The lack of in-depth background knowledge on MLPs and backpropagation meant that the default parameters had to be used. The accuracy of the model was the ultimate deciding factor of the successfulness of this project. Due to this reason, the result from the first attempt was chosen as the predictive model for the project.

4.3.3 Autonomous Driving Script

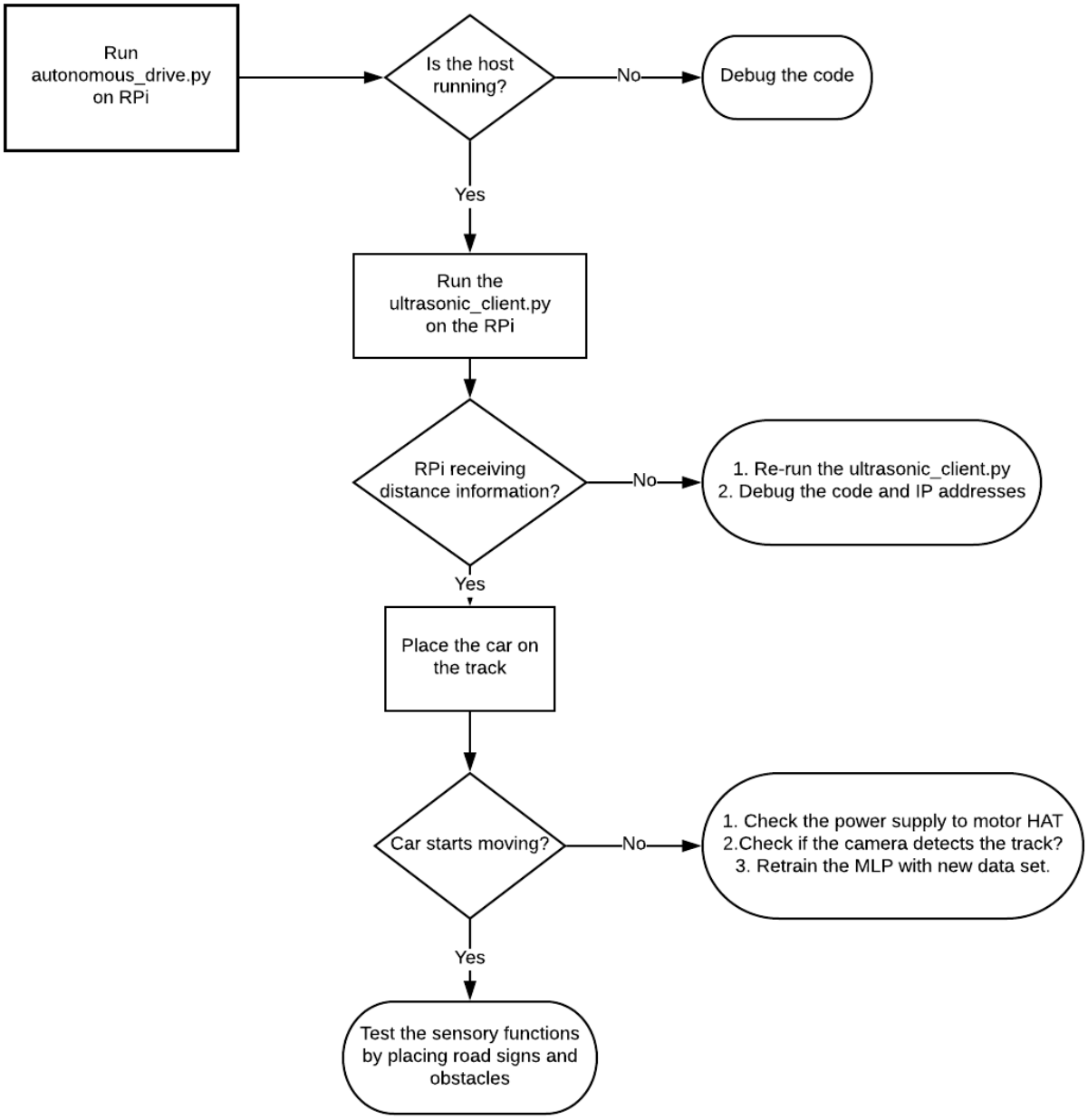


Figure 26: 'autonomous_driving.py' script flowchart

Since the objective of this project is to create a standalone autonomous car using a Raspberry Pi without the assistance from a computer, the 'autonomous_drive.py' script was run directly on the Raspberry Pi itself. From the flowchart it can be seen that the Raspberry Pi acts as both host and client to receive the ultrasonic data. This will be discussed in more detail later in this section. The trained 'mlp.xml' file obtained from the previous section will be used to calculate the driving and steering predictions of the car. If the sequence of operations outlined in the flowchart performs successfully, the car should start moving once it recognise the track layout. If the trained model was highly accurate, the car should be able to drive autonomously even if the layout of the track changes.

4.3.3.1 *Brief explanation of Autonomous driving script*

On execution, a multithread server program runs on the background which receives the sensory data from the ultrasonic sensor once the 'ultrasonic_client.py' script has been run. When trying to implement the ultrasonic sensor code into the 'autonomous_drive.py' script, a number of unavoidable errors emerged. After failing to find a suitable solution to the problems and due to time constraints, it was decided to stream the data within the Raspberry Pi from one script to another via the local network. Since the sensory data only take a minimum amount of bandwidth and the fact that it's been sent and received within the same device meant that there wasn't any significant time lag.

Once the 'autonomous_drive.py' script starts receiving data, a multilayer perceptron neural network is created, on to which the trained 'mlp.xml' file is loaded in the `class NeuralNetwork(object):`

```
self.model = cv2.ml.ANN_MLP_create()   #MLP created
self.model.load('mlp_xml/mlp.xml')     # trained model loaded
```

- The `RCControl(object)` class is created to define the prediction functions. Since we are only concerned with three directions of motions; left, right and forward for driving the car, three variables; 0, 1 and 2 are assigned respectively to each steering prediction. For example:

  ```
  def steer(self, prediction):
      if prediction == 2:
          rearMotor.run(Adafruit_MotorHAT.FORWARD)
          print("Forward")
  ```

  The last expression in this if-statement block stops the motors from running if no prediction can be made.

- The `DistanceToCamera(object)` class consists of the definitions for the camera parameters and the distance calculation using the monocular vision method.

  ```
  self.alpha = 8.0 * math.pi / 180
  self.v0 = 119.865631204
  self.ay = 332.262498472
  ```

  The above camera parameters are obtained from calibrating the PiCamera which will be discussed in further detail in section 4.3.4

```
def calculate(self, v, h, x_shift, image):
    # compute and return the distance from the target point
to the camera
    d = h / math.tan(self.alpha + math.atan((v - self.v0) /
self.ay))
```

The above defined function calculates the distance from the camera to the target object using the monocular vision method discussed in section 2.2.6.

- The `ObjectDetection(object)` class has three main functions defined in 'detect'.
    - **Function 1:** setting the parameters for object detection to be used in the next class.

```
cascade_obj = cascade_classifier.detectMultiScale(
    gray_image,
    scaleFactor=1.1,
    minNeighbors=5,
    minSize=(30, 30),
    flags=cv2.cv.CV_HAAR_SCALE_IMAGE )
```

The detectMultiScale object detects objects of different sizes in the input image. The image frame sizes are reduced by the amount the scaleFactor defined, while the minSize defines the size of the objects that can be ignored.

    - **Function 2:** Utilising the drawing function in OpenCV to draw rectangles around the regions of interests (ROI). It also writes a text above the ROI to represent the object it's detecting using the `'FONT_HERSHEY_SIMPLEX'` function.
    - **Function 3:** "To recognise the different states of the traffic light, some image processing is needed beyond detection. The flowchart below summarises the traffic light recognition process." [17]

Figure 27: Traffic light recognition process [17]

This process is broken down into steps below for a better understanding.

1. First, the image is converted to grey scale as it was pointed out in section 2.2.5.2 that detecting the RGB colour scheme of the image is impractical in real world scenarios. Therefore, the frames are converted to grey scale to detect the Haar-like features and also to reduce frame file size to make it easier for the Raspberry Pi to process the frames.
2. Then the frame is processed through the trained cascade classifier to detect the traffic light within the frame. At this point, the 'ObjectDetection' class applies a bounding box over the ROI.
3. A Gaussian blur is then applied inside the bounding box of the ROI to reduce the noise.

   ```
   mask = cv2.GaussianBlur(roi, (25, 25), 0)
   ```

4. Thirdly, to find the brightest spot in the ROI, the 'cv2.minMaxLoc' function is utilised to detect the brightest and the dimmest pixels in the region.

   ```
   (minVal, maxVal, minLoc, maxLoc) =
   cv2.minMaxLoc(mask)
   ```

5. To determine if the detected bright spot represents the red or the green light, the position of the brightest spot in the ROI calculated.

- Finally, the `VideoStreamHandler(object)` class handles all the object detection and prediction calculation operations. This class loads the trained cascade classifiers for both the stop signs and the traffic lights.

  This class activates the PiCamera with a resolution of 320 x 240 and a frame rate of 32 frames per second. This means that every second, 32 raw images are captured, converted to grey scale and applies a mask to the lower half of the image so the neural network doesn't have to process the unnecessary data.

```
for frame in camera.capture_continuous(rawCapture,
'bgr', use_video_port=True):
    image = frame.array
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # lower half of the image
    half_gray = gray[120:240, :]
```

The frames are then processed through the object detection phase and if an object is detected, calculate the distance to the object.

```
# object detection
v_param1 = self.obj_detection.detect(self.stop_cascade, gray,
image)
v_param2 = self.obj_detection.detect(self.light_cascade, gray,
image)

# distance measurement
if v_param1 > 0 or v_param2 > 0:
    d1 = self.d_to_camera.calculate(v_param1,self.h1,300, image)
    d2 = self.d_to_camera.calculate(v_param2,self.h2,100, image)
    self.d_stop_sign = d1
    self.d_light = d2
```

Each frame is then cleared in preparation for the next frame to be processed and the neural network starts making predications based on the previous frame.

```
# clear the frame in preparation for the next frame
rawCapture.truncate(0)

# neural network makes prediction
prediction = self.model.predict(image_array)
```

The ultrasonic sensor data is used to initiate the first stop condition, where if the distance from the sensor to the obstacle is less than 30cms, the motors are stopped.

```
if sensor_data is not None and sensor_data < 30:
    print("Stop, obstacle in front")
    self.rc_car.stop()
```

The second stop condition is initiated if the camera detects the stop sign 25cms ahead of it. This process sets a flag which activates a 5 second timer and set the flag to false once the timer has finished to drive on the car.

```
elif 0 < self.d_stop_sign < 25 and stop_sign_active:
    print("Stop sign ahead")
    self.rc_car.stop()
```

Finally, the third stop condition is associated with the traffic lights. If the camera detects the traffic lights at a distance of less than 30cms, the steps outlined above in the ObjectDetection class are engaged. If the car detects a red light, it will stay in position until the light is turned off. If it detects a green light, the car will proceed forward as normal.

```
elif 0 < self.d_light < 30:
    # print("Traffic light ahead")
    if self.obj_detection.red_light:
        print("Red light")
        self.rc_car.stop()
```

If none of the stop conditions are met, the neural network makes driving predictions according to the trained MLP model that was loaded at the beginning of the script.

```
self.rc_car.steer(prediction)
```

4.3.4 PiCamera Calibration

Camera calibration is an important aspect of this project as it is the deciding factor between successfulness and unsuccessfulness of object recognition. In order for the camera to detect the object and the monocular vision method to calculate the distance to that object, the camera must be calibrated to remove any distortion in the frame. A detailed guide on how to calibrate camera can be found on the OpenCv-Python Tutorials website [49]. Basically, the PiCamera is used to capture roughly 20 images of a chessboard layout at various angles. The grid layout of the chessboard image must match the layout specified in the code. i.e: 9x6

```
ret, corners = cv2.findChessboardCorners(gray, (9, 6), None)
```

These images are then processed through the camera calibrating script which will output the camera matrix, distortion coefficients, rotation and translation vector values to the terminal. These parameter values are then passed on to the 'autonomous_drive.py' script to calculate the distance from the camera to the detected object.

```
self.v0 = 119.865631204
self.ay = 332.262498472
```

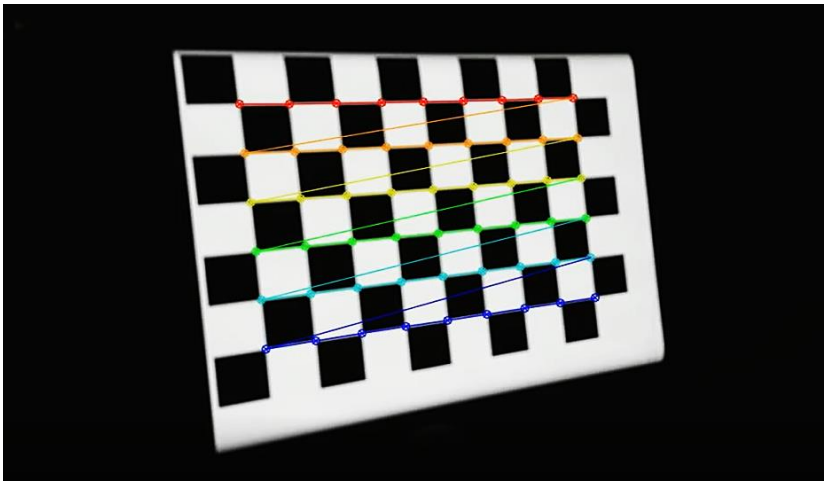


Figure 28: Camera calibration in process

4.3.5 Training the cascade classifier

One of the main hurdles of this project was training the cascade classifier to detect the stop sign, speed limit sign and the traffic lights. There are two applications in OpenCV to train cascade classifier: opencv_haartraining and opencv_traincascade. Although the newer opencv_traincascade written in C++ offers slightly faster performance with the same detection quality as the 'Haar' based one, it was decided that the opencv_haartraining method would be a better option for this project due to the fact that it is written in Python language and the abundance of tutorials on the internet on how to train the classifier using this method.

Two approaches were used to train the classifiers in order to increase the accuracy of the detection rate.

Approach 1: This approach to training the classifier was the most cumbersome method as it required manually gathering hundreds of positive and negative images to be trained on. The tutorial offered by Thorsten Ball on how to 'Train your own OpenCV Haar classifier' [50] was a very comprehensive guide on training the classifier but highly impractical due to the vast amount of data needed to be gathered manually and the lack of unique images of street signs on the internet. The website 'www.image-net.org' offers downloadable archives of thousands of images relating to any searched keyword. This approach would be much suitable for face recognition applications as you are able to download thousands of images of human faces from the above-mentioned website.

However, due to the lack of street sign images on 'image-net.org', 270 images of stop signs were manually downloaded from google images and captured by a phone's camera at different angles. These images were then trained against few hundred negative images (images that doesn't contain stop signs) to create the cascade classifier. This method proved to be unsuccessful due to the extremely low accuracy and the high false positive rate.

Figure 29: samples of positive images used in approach 1

Approach 2: This method was more convenient as it didn't require any positive images to be downloaded but instead manipulate the negative images by overlaying a single positive image over them at different angles and positions. Using the 'image-net.org' archive, 4300 negative random images were downloaded using a simple Python script. These images are then processed through a different Python script to add an overlay of a stop sign image on all of them. This method was somewhat successful in comparison to approach 1 due to the greater amount of training data, but the overall performance of the classifier was still poor as there was still a high false positive rate. This approach followed the tutorial offered by the YouTuber 'sentdex' on 'Creating your own Haar Cascade OpenCV Python Tutorial' [51].

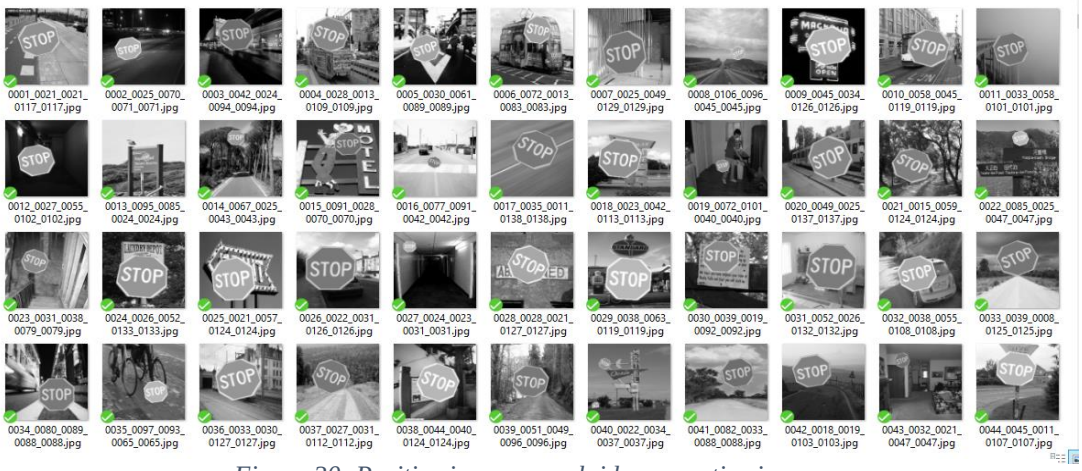


Figure 30: Positive images overlaid on negative images

However, during the testing phase it was found that both of these methods failed to deliver an acceptable false positive rate with a high degree of accuracy, and due to time constraints, it was decided to use pre-trained cascade classifiers found on the internet for this project. Sample images of this classifier in action can be seen in section 8.6, appendix 6. It can be seen that the ROI square constantly jumps around in the frame resulting a high false positive rate.

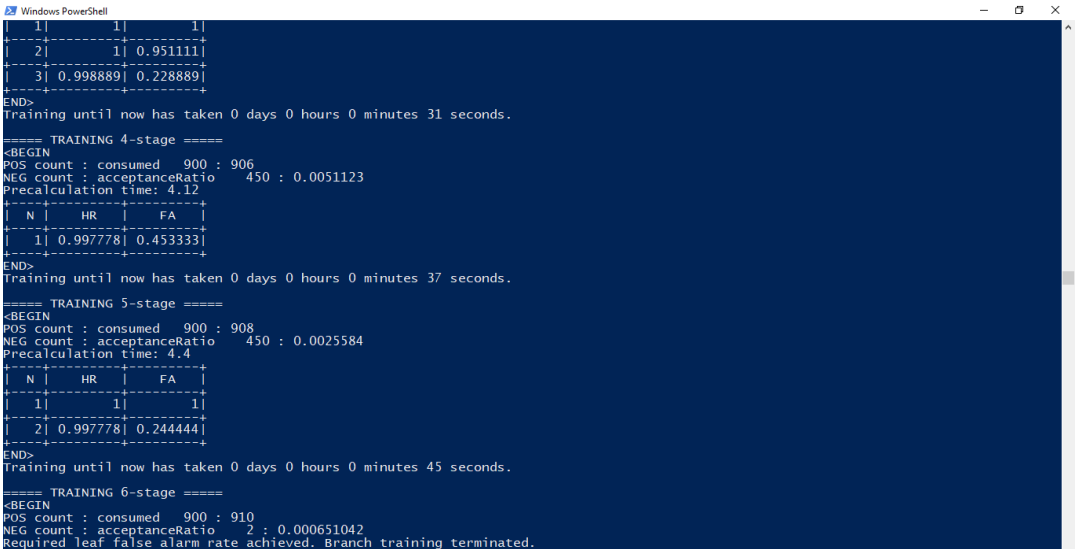


Figure 31: Process of training the cascade classifier

4.4 Alternate system design concept

As part of the continuous research conducted throughout the project execution, a variety of alternative methods and systems were identified which, had they been implemented, may have increased the project's potential for better success. However, due to time constraints and the limited background knowledge of the student in relation to neural networks, the prospect of altering the system design in response to more potentially efficient alternatives proved impractical and as such, were not adopted for use in this project.

As previously discussed in section 3.1.2, object recognition could have been improved by using the 'Pixy (CMUcam5) Smart Vision Sensor' camera as the main camera module for this project. One of the main drawbacks of the generic PiCamera was that the narrow field of view would obscure the road sign as the car gets closer to the object. Since the recognition works optimally at a distance of 25cms from the object, having a wider field of view offered by the CMUcam5 could enhance the recognition ability at a closer distance.

Another area that could be improved is the car's ability to recognise the lane markings of the track itself. This problem is unrelated to the physical constraints of the camera itself, but rather the way in which the neural network is trained. For the purpose of simplicity and practicality this project chose the multilayer perceptron as the class of neural network for training and predicting the driving behaviour of the car. Although it provides less accuracy and could be burdensome for the programmer as it relies heavily on how insightful and able the programmer is, it was the only practical machine learning algorithm that was capable of running on a Raspberry Pi with its mediocre computational power.

Convolutional neural networks (CNN) on the other hand offers much better image recognition performance compared to its counterpart multilayer perceptron. CNNs are basically a derivative of the standard MLP neural network optimised for two-dimensional patter recognition such as face detection or Optical Character Recognition (OCR). Instead of using fully connected hidden layers, the CNN introduces a special network structure, which consists of alternating named convolution and subsampling layers. Moreover, CNNs have fewer weights which are shared among all the repetitive blocks of neurons that are applied across space, increasing the learning efficiency by reducing the number of parameters being learnt. This allows the classifier to detect complex feature at a much faster rate at different positions. In order to have a data reduction, a sub-sampling operation called pooling is performed. This data reduction operation is applied to the predecessor convolution result by a local averaging over a predefined window [52]. This

basically means that deep learning models such as CNNs are capable of learning to focus on the right features themselves, requiring very little guidance from the programmer.

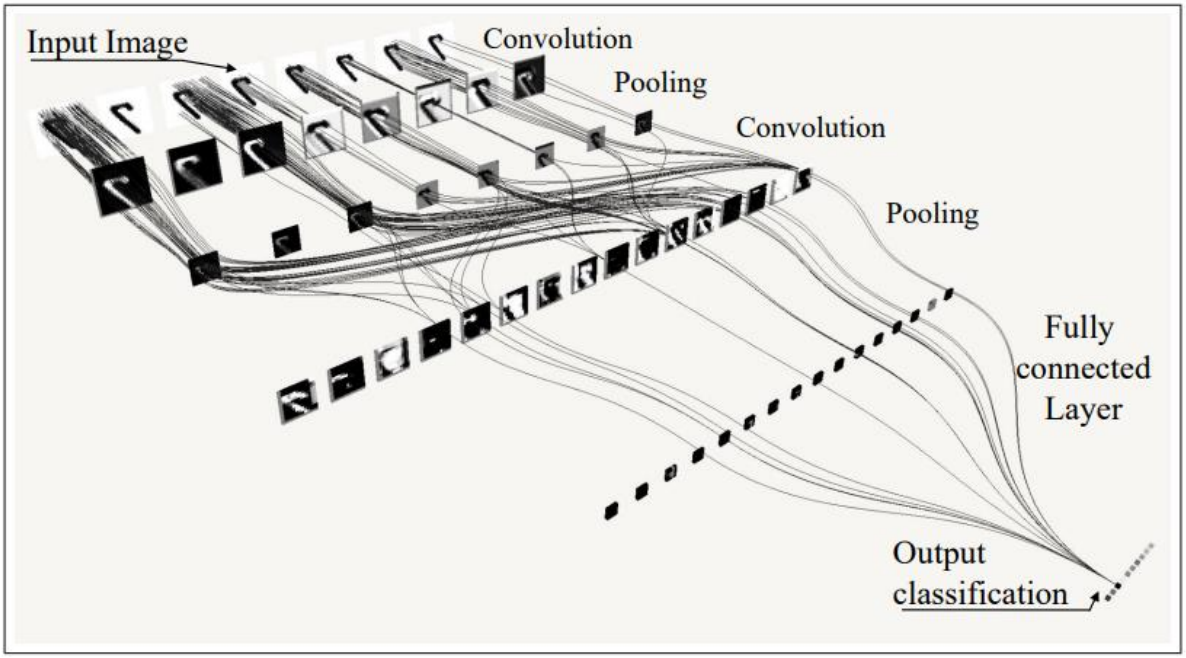


Figure 32: CNN architecture (classification stage) [52]

Although deep learning CNN is a much better candidate for self-driving cars than a standard MLP, it is not possible to implement this system on a small-scale project due to the following reasons:

1. **"Low speed—**cannot run real-time applications on embedded devices, for example, Raspberry PI and NVIDIA Jetson
2. **Very large model size**—models are too large to fit into small devices
3. **Large memory footprint**—requires too much RAM

The reason why CNN's are so slow, is that they spend most of their time on convolutional layers, and most of the storage on fully connected layers. Although some new models reduce or get rid of some fully connected layers, CNN's still need to run on a large GPU to accelerate the process of fast matrix multiplication. Therefore, small devices like the Raspberry PI (RAM of 2 GB) are unable to deploy very deep CNN models with high recognition rate." [53]. Moreover, CNNs requires a vast amount of training data which was unattainable given the circumstances of the project's environment. Therefore, due to the impracticality of this neural network, MLP was chosen as the class of neural network to be deploy on the Raspberry Pi.

4.4.1 CNN Self-driving car simulation

To prove the effectiveness of this concept, a simulation of this model was conducted using the Udacity's self-driving car simulator software [54]. The simulation model is developed using 'Keras' which is a powerful Python library for developing and evaluating deep learning models [55]. The model itself is being run on top of TensorFlow environment which is an open-source software library for dataflow programming allowing developers to design, build, and train deep learning models effortlessly.

How it works:

The simulator software offers two modes: training mode and autonomous mode and the option to select a track. Once the training mode and the track is selected, the training environment is generated with a 3D modelled car that has 3 virtual cameras on top of it pointing forward, left and right. Pressing the 'R' key on the keyboard starts the data collection process while being driven around the track. The 3 cameras simultaneously collects images at 3 different angles which are saved in a folder with a descriptive name which helps to identify the direction the camera is pointing at. During this training process, 19,548 images were captured at 320 x 160 resolution.

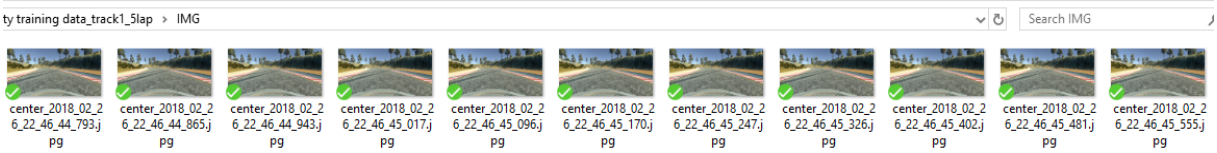


Figure 33: Saved images with a descriptive name

These image file locations are then automatically saved into a .CSV file with the control values that corresponds to the driving commands such as acceleration, break and steering angles at the specific time and location.

	A	B	C	D	E	F	G	H
1	\center_2018_02_26_22_46_44_793.jpg	\left_2018_02_26_22_46_44_793.jpg	\right_2018_02_26_22_46_44_793.jpg	0	0	0	8.58E-06	
2	\center_2018_02_26_22_46_44_865.jpg	\left_2018_02_26_22_46_44_865.jpg	\right_2018_02_26_22_46_44_865.jpg	0	0	0	4.28E-06	
3	\center_2018_02_26_22_46_44_943.jpg	\left_2018_02_26_22_46_44_943.jpg	\right_2018_02_26_22_46_44_943.jpg	0	0	0	3.77E-06	
4	\center_2018_02_26_22_46_45_017.jpg	\left_2018_02_26_22_46_45_017.jpg	\right_2018_02_26_22_46_45_017.jpg	0	0	0	8.37E-07	
5	\center_2018_02_26_22_46_45_096.jpg	\left_2018_02_26_22_46_45_096.jpg	\right_2018_02_26_22_46_45_096.jpg	0	0	0	2.57E-06	
6	\center_2018_02_26_22_46_45_170.jpg	\left_2018_02_26_22_46_45_170.jpg	\right_2018_02_26_22_46_45_170.jpg	0	0	0	1.34E-05	
7	\center_2018_02_26_22_46_45_247.jpg	\left_2018_02_26_22_46_45_247.jpg	\right_2018_02_26_22_46_45_247.jpg	0	0	0	2.48E-06	
8	\center_2018_02_26_22_46_45_326.jpg	\left_2018_02_26_22_46_45_326.jpg	\right_2018_02_26_22_46_45_326.jpg	0	0	0	2.03E-05	
9	\center_2018_02_26_22_46_45_402.jpg	\left_2018_02_26_22_46_45_402.jpg	\right_2018_02_26_22_46_45_402.jpg	0	0.11302	0	0.056624	
10	\center_2018_02_26_22_46_45_481.jpg	\left_2018_02_26_22_46_45_481.jpg	\right_2018_02_26_22_46_45_481.jpg	0	0.34178	0	0.286338	
11	\center_2018_02_26_22_46_45_555.jpg	\left_2018_02_26_22_46_45_555.jpg	\right_2018_02_26_22_46_45_555.jpg	0	0.562831	0	0.639088	
12	\center_2018_02_26_22_46_45_630.jpg	\left_2018_02_26_22_46_45_630.jpg	\right_2018_02_26_22_46_45_630.jpg	0	0.790211	0	1.3322	
13	\center_2018_02_26_22_46_45_706.jpg	\left_2018_02_26_22_46_45_706.jpg	\right_2018_02_26_22_46_45_706.jpg	0	1	0	2.290452	

Figure 34: .CSV file with control values corresponding to the specific frame of reference.

This data is then processed through TensorFlow with GPU support to accelerate the training process.

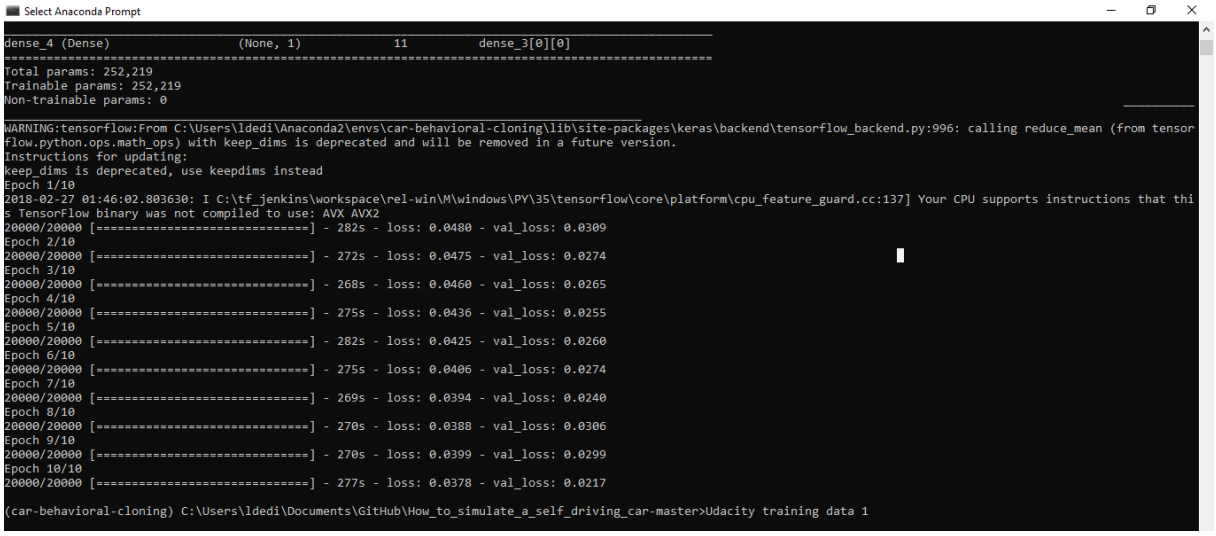


Figure 35: CNN training on TensorFlow environment

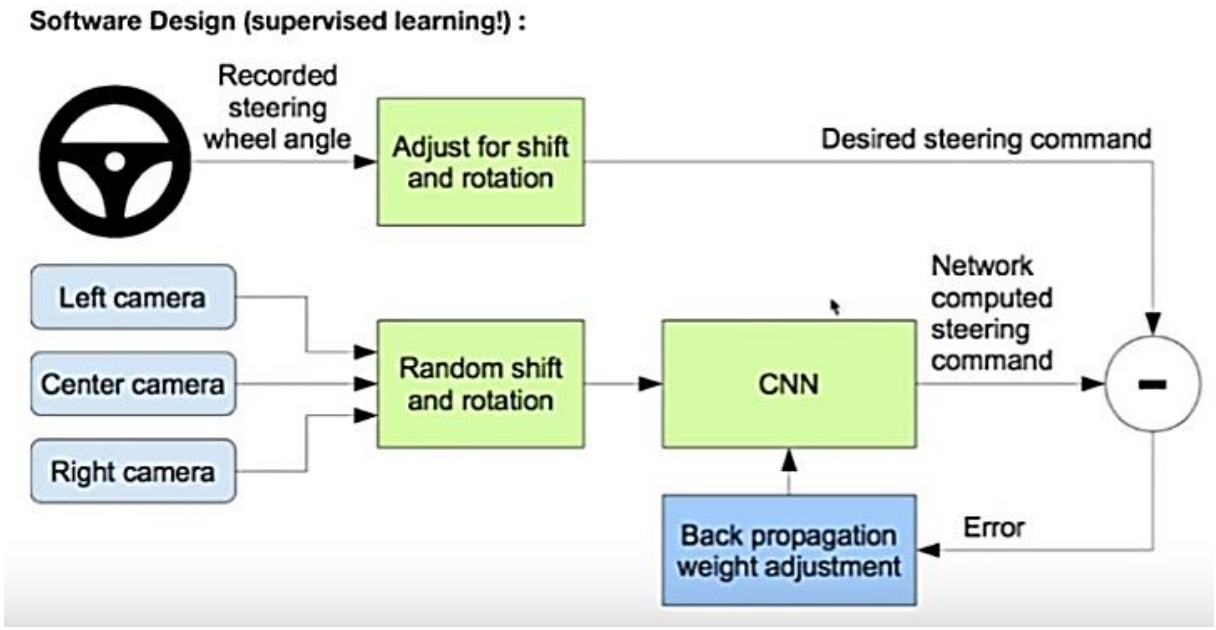


Figure 36: System design of the training process

Once the training was completed, it outputs a number of trained models in the form of a hierarchical data format 5 (.h5) files. One of these files can then be used to run the car in the autonomous mode offered by the software. Running the autonomous mode at each epoch, we can see the advancements and the improvements at each iteration of the convolution network. For example, on simulation it can be seen that the car drives more steadily at epoch 6 compared to epoch 0 which was the initially trained model.

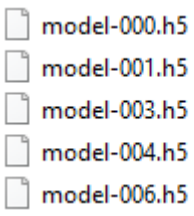


Figure 37: output models

Figure 38: Car driving autonomously using the trained CNN model

This simulation proved that a deep learning convolution neural network performs exceptionally well compared to a multilayer perceptron. Since this training method can handle a large amount of data yielding a highly accurate model, it is the algorithm of ANN preferred by almost every car manufacturer for training their self-driving vehicles.

4.5 Conclusion

During the system design phase many obstacles were encountered, ranging from software issues to hardware complications. Some of these problems were overcome using inconvenient yet effective solutions such as for example, using two keyboards to drive the car and to collect the training data. The hardware segment of this project was reasonably straightforward to implement with very little background knowledge on electronics required. Basic understandings of voltage divider laws and being able to measure current input and output values using a multimeter was adequate for setting up the Raspberry Pi and the ultrasonic sensor.

The software element of the project required thorough analysis of the sample Python code from Zheng Wang's Self Driving RC Car project [17] to understand the functions of each class and object. This was an essential step towards better understanding the functionalities and the capabilities of the program itself and how it can be adopted for the needs of this project. This chapter thoroughly explained the important fragments of the code and the process of collecting, training and deploying the data on to the Raspberry Pi.

Although the amount of detail covered in this topic can be overwhelming, the most important thing to remember when executing these programs is to remember that when moving between two operating systems and development environments, it is crucial to keep a track of all the data being used and the versions of the updated code. Throughout the entirety of the project, a vast amount of training data was collected and a number of trained models were generated. It is important to keep these training data and trained models organised in a well-structured folder directory in a removeable flash drive, both for safe keeping and to ease the process of transferring files from the PC to the Raspberry Pi.

To summaries the software process:

1. To collect the training data, run the script 'collect_training_data.py' on the PC and then 'stream_client.py' and 'motor_keyboard_control.py' on the Raspberry Pi.
   ➔ Simultaneously use the PC keyboard and the wireless keyboard connected to the RPi to drive the car while collecting the data.
2. To train the MLP with the collected data, run the 'mlp_training.py' script on the PC with Python 3.6 as the Python Interpreter. Then transfer the trained MLP model to the RPi.
3. Finally, to run the self-driving car, execute 'autonomous_drive.py' and the 'ultrasonic_client.py' on the RPi.

5 Results

5.1 Data collection

During the data collection process the car was manually driven around the track collecting images for training the neural network.

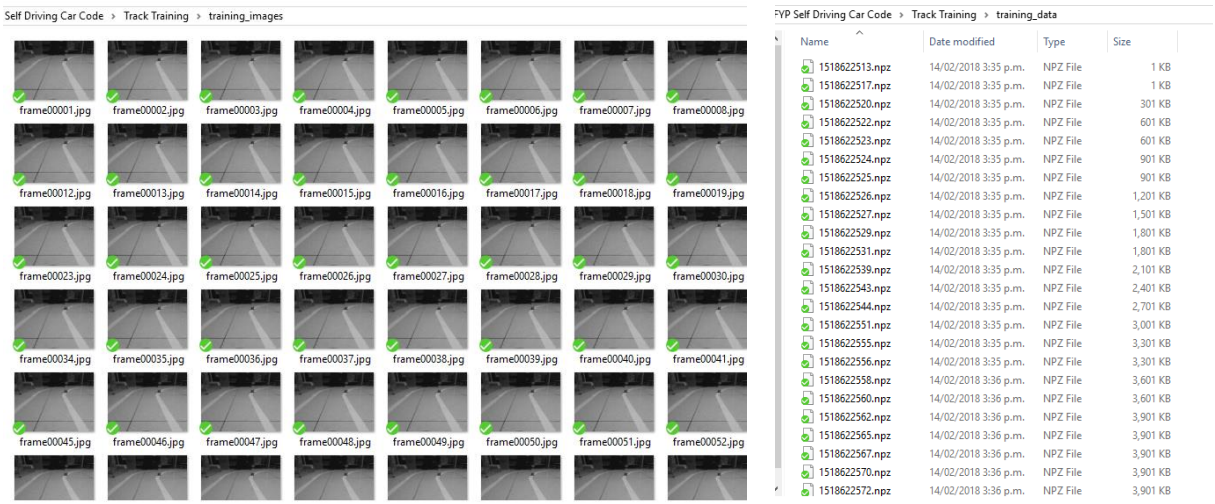


Figure 39: Collected image data and Numpy.savez files

Although there was a problem controlling the car via the PC as mentioned in section 4.3.1, the training data was successfully collected in synchronous with the position of the car. This ensured that the Numpy.savez files had all the information regarding the track layout in the current frame, the position of the car and the direction the car was moving.

5.2 Neural Network Training

During the neural network training stage many obstacles were encountered due to problems ranging from Python version incompatibilities to inaccuracy of the trained model. It was later identified that one of the main reasons for inaccuracy of the model was caused by the incompatibility of the Python version. As mentioned in section 4.3.2, the 32-bit version of Python 2.7 has a memory usage limit of 2 GBs which meant that to perform the training procedure, only a small amount of data could be used or otherwise the program terminates with an error. This led to a high degree of inaccuracy of the trained model, ultimately failing the car to navigate the around the track autonomously. This problem was fixed by installing the 64-bit Python 3.6 and modifying the 'mlp_training.py' script. This allowed for a greater amount of training data to be processed, thus increasing the accuracy of the MLP neural network.

Table 2: Accuracy of the trained model in relation to the amount of training data used

# Of Laps	Amount of Data	Accuracy
1	450 MB	45%
2	800 MB	68%
5	3.2 GB	80.20%

Once the underlying problem was fixed, further training tests were conducted, gradually increasing the amount of data and comparing the relationship between the input data and the resulting accuracy of the MLP model. During this process it was discovered that although the software limitation on memory usage was removed, the physically available RAM and the computational power was still constraining the training process. Some measures were taken to fix this problem such as reducing the resolution of the images and increasing the available virtual memory to 64 GBs but the process was still unsuccessful. Further testing showed that to achieve the highest rate of accuracy without system failure, the training data had to be restricted to roughly 3GBs in size. Presumably this accuracy could be improved by using a computer with higher specifications to handle the large amount of data needed for training.

Note: For demonstration purposes, the trained model with 64.77% accuracy was chosen as the 80% accurate model was trained outside of the University premises for testing purposes.

System Type	x64-based PC
System SKU	Aspire V3-572PG_0867_V1.04
Processor	Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz, 2401 Mhz, 2 Core(s), 4 Logical Processor(s)
Installed Physical Memory (RAM)	8.00 GB
Total Physical Memory	7.89 GB
Available Physical Memory	2.34 GB
Total Virtual Memory	71.9 GB
Available Virtual Memory	65.7 GB
Page File Space	64.0 GB

Figure 40: The PC system configuration used for this project

5.3 Autonomous Driving

The goal of autonomous driving was successfully achieved at the end of the project, despite the challenges encountered due to the limitations of the Raspberry Pi's computing power and the car's mechanical shortcomings. The road sign recognition aspect of the project performed exceptionally well with a high degree of accuracy and detecting the sign in under a second while being moved around. As previously mentioned, due to the high inaccuracy and unacceptable false positive rate, pre-trained cascade classifiers were used.

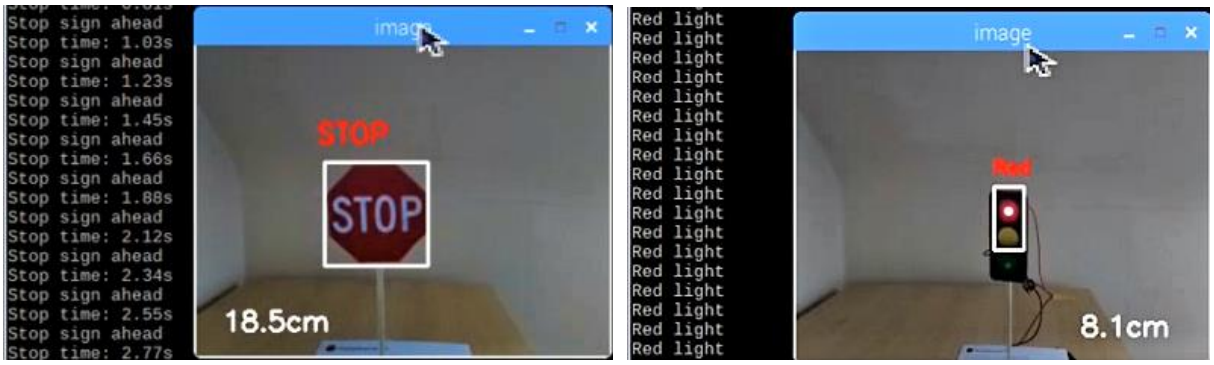


Figure 41: Haar cascade classifiers detecting stop sign and traffic light in real time

As you can see from the above figure, both classifiers can successfully detect the region of interest in the object but fails to accurately calculate the distance from the object to the camera. This is due to factors such as monocular vision is not as effective as binocular vision at depth perception so therefore it must rely solely on visual information received from just one camera to calculate the distance. This can be easily self-demonstrated by trying to visualise the distance to an object by covering one eye. Naturally, we all know it is hard to distinguish the depth or the distance to an object when viewed from just one eye. Same principle applies to monocular vision depth perception method we used in this project. The accuracy of the distance calculated can be improved by accurately calibrating the camera module and fine-tuning the camera parameters.

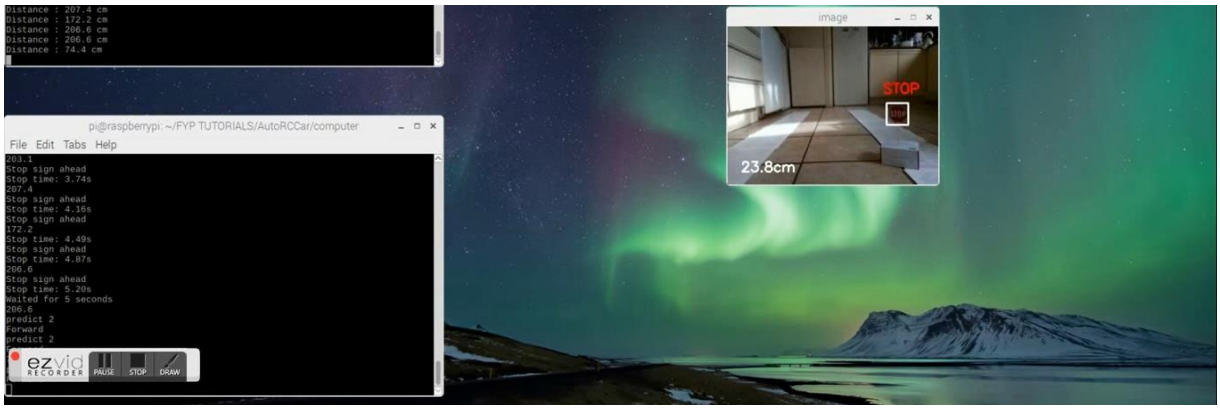


Figure 42: Commencing the 5 second count down after detecting a stop sign

Same cannot be said for ultrasonic sensor based obstacle avoidance system. With a range of 2cm – 400cms and an accuracy of ±3mms, ultrasonic sensors are ideal for any small scale low budget projects like this. Due to the high accuracy and the detection rate, the car was able to effortlessly come to a halt on detection of an obstacle at the specified distance. The only issue that influenced the effectiveness of this sensor was that fact that the data had to be streamed via the local network to the same device due to system design problems outlined in section 4.3.3.1. This introduced a small time-delay between the time it was detected to the time it was registered by the program. This delay only affected the driving results at faster speeds.

Figure 43: Obstacle avoidance system

Being unable to control the steering function due to mechanical issues, the car was only capable of navigating in a straight line. Although this maybe the case, the neural network still produced steering predictions which were observable in the terminal. As a result of the low accuracy of the MLP neural network, the car would sometimes generate false right or left steering predictions causing it to stop in the middle of the track.

5.4 Conclusion

Overall the project successfully achieved the goal of autonomous driving with road sign recognition. Although the car was able to navigate through the track and adhere to the road signs, some aspects of the final results were not as expected. Due to the challenges faced in the design and testing phase, some of the original objectives such as changing speed in response to speed signs and navigating around corners had to be neglected. This chapter examined the results obtained throughout the design and implementation phase and briefly discussed the challenges that altered the outcome of the final results.

6 Conclusion

This project addressed the feasibility of implementing a self-driving car on a Raspberry Pi running on an artificial neural network and presented an evaluation of machine learning systems that could improve the reliability and the accuracy by comparing two different neural network algorithms. During the initial research phase, it was realised that a different approach must be used to implement the machine vision capabilities in comparison to the technologies employed by most of the mainstream self-driving vehicles. Although it was found that the Convolutional Neural Network (CNN) algorithm used in majority of the autonomous cars offers a better performance with a high degree of accuracy, this project chose the low-level neural network algorithm; Multilayer Perceptron (MLP) for training and predicting the driving behaviour based on the visual information. This was due to the insufficient computing power on the Raspberry Pi being unable to process the heavy work load in real time. Despite the fact that the data was trained on a computer with more processing power than on a RPi, the highest accuracy achieved was only 80% due to memory constraints and processing power limitations on the computer. Therefore, to show the effectiveness of the CNN machine vision algorithm over the MLP method, the self-driving system was trained and processed in a computer simulation.

The student also encountered challenges when training the cascade classifier to identify the stop signs and the traffic lights. Although more than 4,000 positive and negative sample images were used to train the classifier, the false positive rate was still quite high, which was unacceptable for the purpose of this project. This issue was rectified by using pre-trained classifiers found on the internet due to time constraints and the lack of training data. The accuracy of image recognition was improved by better calibrating the PiCamera and tweaking the camera parameters in the code.

Although some features such as changing the speed of the car in response to speed signs and navigating through corners and turns on the track wasn't achieved, the final result of this project proved to be successful with all other aspects; self-driving on a track, road sign detection and front collision avoidance functions working. The outcome of this project could have been improved by using a different toy car with minimum physical constraints, utilising a better camera such as the Pixy CMUcam5, integrating the ultrasonic sensor distance measuring function directly into the main Python script and training the MLP ANN on a PC with greater computer power to increase the accuracy without running into system errors. Overall, this project proved that it is plausible to implement a self-driving car on a RPi running on an ANN.

7 References and sources of information.

[1]"Autonomous Cars and Society", *web.wpi.edu*, 2018. [Online]. Available: https://web.wpi.edu/Pubs/E-project/Available/E-project-043007-205701/unrestricted/IQPOVP06B1.pdf. [Accessed: 17- Feb- 2018].

[2]"Provisional Review of Fatal Collisions", *Rsa.ie*, 2018. [Online]. Available: http://www.rsa.ie/Documents/Fatal%20Collision%20Stats/Provisional_Reviews_of_Fatal_Collisions/RSA%20Provisional%20Review%20of%20Fatalities%2031%20December%202017.pdf. [Accessed: 17- Feb- 2018].

[3]"Advanced driver-assistance systems", *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Advanced_driver-assistance_systems. [Accessed: 17- Feb- 2018].

[4]"Phase 2 of the Glucksman Library at UL | Glucksman Library", *Ul.ie*, 2018. [Online]. Available: https://www.ul.ie/library/about/news-events/phase-2-glucksman-library-ul. [Accessed: 17- Feb- 2018].

[5]"Model S | Tesla", *Tesla.com*, 2018. [Online]. Available: https://www.tesla.com/en_IE/models. [Accessed: 17- Feb- 2018].

[6]"Autopilot", *Tesla.com*, 2018. [Online]. Available: https://www.tesla.com/en_IE/autopilot. [Accessed: 17- Feb- 2018].

[7]"Waymo", *Waymo*, 2018. [Online]. Available: https://waymo.com/. [Accessed: 19- Feb- 2018].

[8]"Autonomous car | Levels of driving automation", *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Autonomous_car#Levels_of_driving_automation. [Accessed: 20- Feb- 2018].

[9]"Tesla Autopilot", *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Tesla_Autopilot. [Accessed: 20- Feb- 2018].

[10]"difference between Tesla's autopilot system and Google's driver-less car", *www.quora.com*, 2018. [Online]. Available: https://www.quora.com/What-is-the-difference-between-Teslas-autopilot-system-and-Googles-driver-less-car-How-on-earth-did-Tesla-manage-to-pull-this-off-while-other-companies-including-Google-have-been-trying-to-do-it-for-years. [Accessed: 21- Feb- 2018].

[11] F. Lambert, "A look at Tesla's new Autopilot hardware suite: 8 cameras, 1 radar, ultrasonics & new supercomputer", *Electrek*, 2018. [Online]. Available: https://electrek.co/2016/10/20/tesla-new-autopilot-hardware-suite-camera-nvidia-tesla-vision/. [Accessed: 21- Feb- 2018].

[12] E. Codes and S. Ray, "Essentials of Machine Learning Algorithms (with Python and R Codes)", *Analytics Vidhya*, 2018. [Online]. Available: https://www.analyticsvidhya.com/blog/2017/09/common-machine-learning-algorithms/. [Accessed: 24- Feb- 2018].

[13] X. Li, "Improved AdaBoost Algorithm and Object Recognition Based On Haar-Like Training", *Theseus.fi*, 2018. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/113267/Li_Xiuyang.pdf?sequence=1. [Accessed: 26- Feb- 2018].

[14] K. Markham, "Comparing supervised learning algorithms", *Data School*, 2018. [Online]. Available: http://www.dataschool.io/comparing-supervised-learning-algorithms/. [Accessed: 27- Feb- 2018].

[15] C. Stergiou and D. Siganos, "Neural Networks", *Doc.ic.ac.uk*. [Online]. Available: https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#What is a Neural Network. [Accessed: 27- Feb- 2018].

[16] C. Woodford, "How neural networks work - A simple introduction", *Explain that Stuff*, 2017. [Online]. Available: http://www.explainthatstuff.com/introduction-to-neural-networks.html. [Accessed: 27- Feb- 2018].

[17] Z. Wang, "Self Driving RC Car", *wordpress.com*, 2017. [Online]. Available: https://zhengludwig.wordpress.com/projects/self-driving-rc-car/. [Accessed: 27- Feb- 2018].

[18]"Perceptrons - the most basic form of a neural network - Applied Go", *Applied Go*, 2016. [Online]. Available: https://appliedgo.net/perceptron/. [Accessed: 27- Feb- 2018].

[19]"Artificial neural network", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/Artificial_neural_network#Backpropagation. [Accessed: 27- Feb- 2018].

[20] W. propogation?, "What is the diffirence between SGD and back propogation?", *Stackoverflow.com*. [Online]. Available: https://stackoverflow.com/questions/37953585/what-is-the-diffirence-between-sgd-and-back-propogation. [Accessed: 28- Feb- 2018].

[21] J. Fröhlich, "Backpropagation - Neural Networks with Java", *Nnwj.de*, 2004. [Online]. Available: http://www.nnwj.de/backpropagation.html. [Accessed: 28- Feb- 2018].

[22] Mashimo, "Back-propagation for neural network", *Look back in respect*, 2015. [Online]. Available: https://mashimo.wordpress.com/2015/09/13/back-propagation-for-neural-network/. [Accessed: 28- Feb- 2018].

[23] S. Soo, "Object detection using Haar-cascade Classifier", *Pdfs.semanticscholar.org*. [Online].Available:https://pdfs.semanticscholar.org/0f1e/866c3acb8a10f96b432e86f8a61be5eb6799.pdf. [Accessed: 28- Feb- 2018].

[24]"Viola–Jones object detection framework", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/Viola%E2%80%93Jones_object_detection_framework#Feature_types_and_evaluation. [Accessed: 28- Feb- 2018].

[25] C. Papageorgiou, "A General Framework for Object Detection", *www.researchgate.net*, 1998.[Online].Available:https://www.researchgate.net/publication/3766402_General_framework_for_object_detection. [Accessed: 28- Feb- 2018].

[26] "OpenCV: Face Detection using Haar Cascades", *Docs.opencv.org*. [Online]. Available: https://docs.opencv.org/3.3.0/d7/d8b/tutorial_py_face_detection.html. [Accessed: 28- Feb- 2018].

[27] P. Viola and M. Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features", *Wearables.cc.gatech.edu*, 2001. [Online]. Available: http://wearables.cc.gatech.edu/paper_of_week/viola01rapid.pdf. [Accessed: 28- Feb- 2018].

[28]"Cascading classifiers", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/Cascading_classifiers. [Accessed: 28- Feb- 2018].

[29] V. Dedhia, "Computer Vision Talks", *Computervisionwithvaibhav.blogspot.ie*, 2015. [Online]. Available: http://computervisionwithvaibhav.blogspot.ie/2015/08/. [Accessed: 28- Feb- 2018].

[30]"AdaBoost", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/AdaBoost. [Accessed: 28- Feb- 2018].

[31] C. Jiangwei, J. Lisheng, G. Lie, Libibing and W. Rongben, "Study on method of detecting preceding vehicle based on monocular camera - IEEE Conference Publication", *Ieeexplore.ieee.org*, 2004. [Online]. Available: http://ieeexplore.ieee.org/document/1336478/?tp=&arnumber=1336478&url=http:%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D1336478. [Accessed: 02- Mar- 2018].

[32]"Monocular vision", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/Monocular_vision. [Accessed: 02- Mar- 2018].

[33]"Depth perception", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/Depth_perception#Monocular_cues. [Accessed: 02- Mar- 2018].

[34]"Raspberry Pi 3 Model B - Seeed Wiki", *Wiki.seeed.cc*. [Online]. Available: http://wiki.seeed.cc/Raspberry_Pi_3_Model_B/. [Accessed: 03- Mar- 2018].

[35]"Raspberry Pi", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/Rasberry_pi. [Accessed: 03- Mar- 2018].

[36]"The Raspberry Pi camera guide", *www.intorobotics.com*. [Online]. Available: https://www.intorobotics.com/raspberry-pi-camera-guide/. [Accessed: 03- Mar- 2018].

[37] R. Kinch, "Raspberry Pi", *Truetex.com*, 2016. [Online]. Available: http://www.truetex.com/raspberrypi#adapters. [Accessed: 03- Mar- 2018].

[38]"Overview - CMUcam5 Pixy - CMUcam: Open Source Programmable Embedded Color Vision Sensors", *Cmucam.org*. [Online]. Available: http://www.cmucam.org/projects/cmucam5. [Accessed: 03- Mar- 2018].

[39]"I2C - learn.sparkfun.com", *Learn.sparkfun.com*. [Online]. Available: https://learn.sparkfun.com/tutorials/i2c. [Accessed: 04- Mar- 2018].

[40]"Overview | Adafruit DC and Stepper Motor HAT for Raspberry Pi | Adafruit Learning System", *Learn.adafruit.com*. [Online]. Available: https://learn.adafruit.com/adafruit-dc-and-stepper-motor-hat-for-raspberry-pi/overview. [Accessed: 04- Mar- 2018].

[41]"HC-SR04 Ultrasonic sensor - ElectroDragon", *Electrodragon.com*. [Online]. Available: http://www.electrodragon.com/w/HC-SR04_Ultrasonic_sensor. [Accessed: 04- Mar- 2018].

[42]"Raspbian", *En.wikipedia.org*. [Online]. Available: https://en.wikipedia.org/wiki/Raspbian. [Accessed: 04- Mar- 2018].

[43]"PyCharm: Python IDE for Professional Developers by JetBrains", *JetBrains*. [Online]. Available: https://www.jetbrains.com/pycharm/. [Accessed: 04- Mar- 2018].

[44]"About - OpenCV library", *Opencv.org*. [Online]. Available: https://opencv.org/about.html. [Accessed: 04- Mar- 2018].

[45]"OpenCV: Introduction to OpenCV-Python Tutorials", *Docs.opencv.org*. [Online]. Available: https://docs.opencv.org/3.3.0/d0/de3/tutorial_py_intro.html. [Accessed: 04- Mar- 2018].

[46] A. Rosebrock, "Raspbian Stretch: Install OpenCV 3 + Python on your Raspberry Pi - PyImageSearch", *PyImageSearch*. [Online]. Available: https://www.pyimagesearch.com/2017/09/04/raspbian-stretch-install-opencv-3-python-on-your-raspberry-pi/. [Accessed: 04- Mar- 2018].

[47]M. Hawkins, "Ultrasonic Distance Measurement Using Python - Part 1 - Raspberry Pi Spy", *Raspberry Pi Spy*, 2012. [Online]. Available: https://www.raspberrypi-spy.co.uk/2012/12/ultrasonic-distance-measurement-using-python-part-1/. [Accessed: 05- Mar- 2018].

[48] C. Gohlke, "Unofficial Windows Binaries for Python Extension Packages", *www.lfd.uci.edu*. [Online]. Available: https://www.lfd.uci.edu/~gohlke/pythonlibs/. [Accessed: 06- Mar- 2018].

[49]"Camera Calibration — OpenCV-Python Tutorials 1 documentation", *Opencv-python-tutroals.readthedocs.io*. [Online]. Available: http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html. [Accessed: 12- Mar- 2018].

[50]"Coding Robin", *Coding-robin.de*. [Online]. Available: http://coding-robin.de/2013/07/22/train-your-own-opencv-haar-classifier.html#thorsten. [Accessed: 12- Mar- 2018].

[51]"Creating your own Haar Cascade OpenCV Python Tutorial", *Pythonprogramming.net*. [Online]. Available: https://pythonprogramming.net/haar-cascade-object-detection-python-opencv-tutorial/. [Accessed: 12- Mar- 2018].

[52]"A comparison study between MLP and Convolutional Neural Network models for character recognition", *Hal-upec-upem.archives-ouvertes.fr*, 2017. [Online]. Available: https://hal-upec-upem.archives-ouvertes.fr/hal-01525504/document. [Accessed: 13- Mar- 2018].

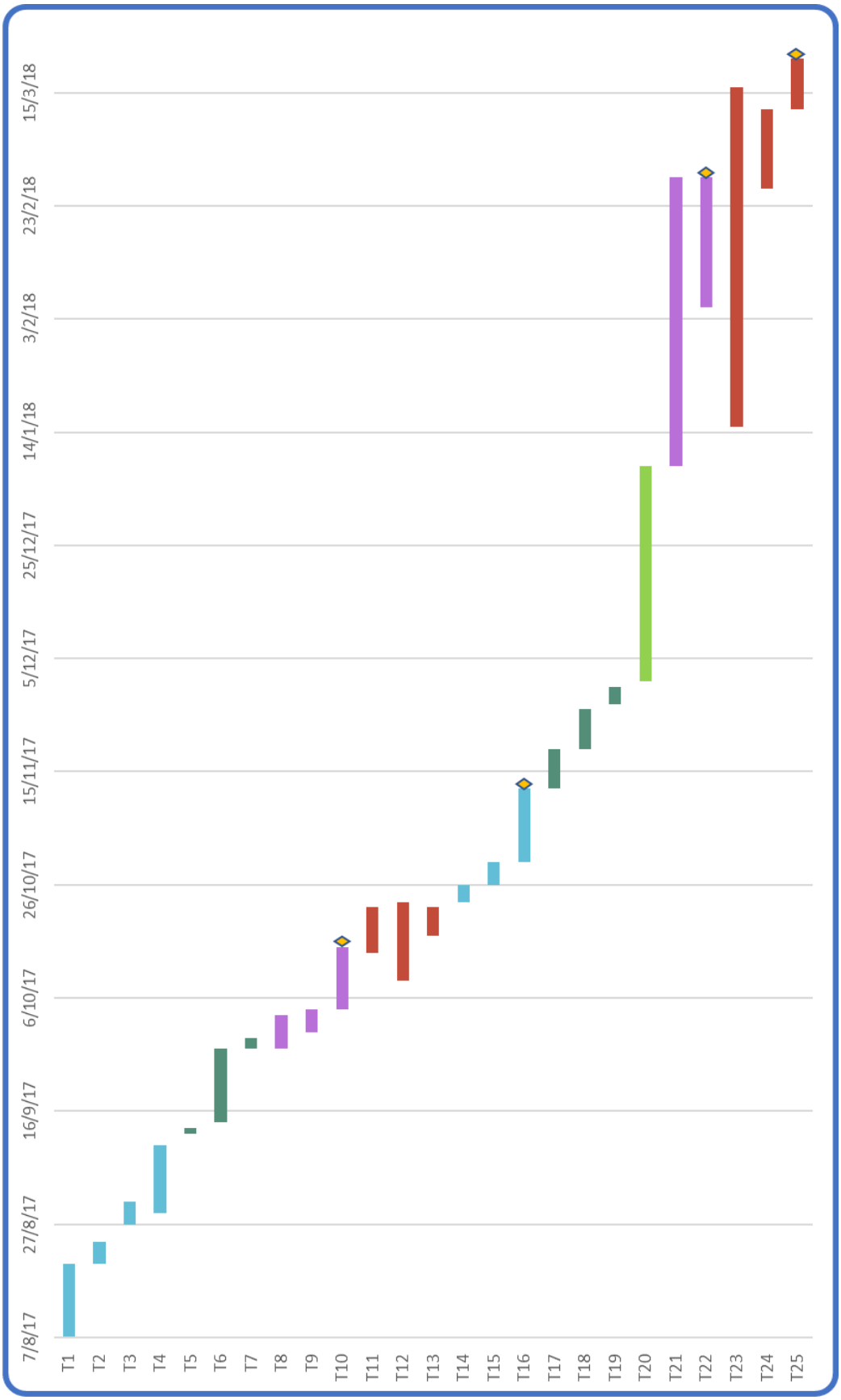
[53]"Accelerating Convolutional Neural Networks on Raspberry Pi", *Medium*. [Online]. Available: https://medium.com/@Synced/accelerating-convolutional-neural-networks-on-raspberry-pi-725600463fd0. [Accessed: 13- Mar- 2018].

[54]"How_to_simulate_a_self_driving_car", *GitHub*. [Online]. Available: https://github.com/llSourcell/How_to_simulate_a_self_driving_car. [Accessed: 14- Mar- 2018].

[55]J. Brownlee, "Develop Your First Neural Network in Python With Keras Step-By-Step - Machine Learning Mastery", *Machine Learning Mastery*. [Online]. Available: https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/. [Accessed: 14- Mar- 2018].

8 Appendices

8.1 Appendix 1: Gantt Chart



8.2 Appendix 2: Gantt Chart task descriptions

Task	Start Date	End Date	Duration (Days)	Description
T1	07/08/2017	20/08/2017	13	online research for similar projects
T2	20/08/2017	24/08/2017	4	purchasing required hardware
T3	27/08/2017	31/08/2017	4	setting up the raspberry pi & installing the required software
T4	29/08/2017	10/09/2017	12	Testing out the components using tutorial codes. i.e. Motors and Ultrasonic sensor
T5	12/09/2017	13/09/2017	1	Meet up with FYP supervisor Ciaran McNamee
T6	14/09/2017	27/09/2017	13	Start working on the Interim Report
T7	27/09/2017	29/09/2017	2	Layout the components in the car
T8	27/09/2017	03/10/2017	6	Program the Raspberry Pi HAT to test the car motors and calculate power requirements
T9	30/09/2017	04/10/2017	4	Purchase the required batteries or power pack for the system.
T10	04/10/2017	15/10/2017	11	Start designing the Track and traffic signs
T11	14/10/2017	22/10/2017	8	Test out the Raspberry Pi and the motor HAT with one power pack
T12	09/10/2017	23/10/2017	14	Work on the Presentation
T13	17/10/2017	22/10/2017	5	Test out the Raspberry Pi Camera
T14	23/10/2017	26/10/2017	3	Install the OpenCV software on the Raspberry Pi and required addons
T15	26/10/2017	30/10/2017	4	Set up and configure the OpenCV platform
T16	30/10/2017	12/11/2017	13	Train the OpenCV platform with negative and positive samples
T17	12/11/2017	19/11/2017	7	Test out the OpenCV algorithm with images of traffic signs
T18	19/11/2017	26/11/2017	7	Fine tune the algorithm with more test subjects at different distances
T19	27/11/2017	30/11/2017	3	Connect the Ultrasonic sensor to the Pi HAT and test it out using tutorial code
T20	01/12/2017	08/01/2018	38	Exams & Christmas Holidays
T21	08/01/2018	28/02/2018	51	Program to Pi HAT to control the Motors with the inputs from the Ultrasonic sensor and OpenCV
T22	05/02/2018	28/02/2018	23	Test the car on the track & fine tune the controls
T23	15/01/2018	16/03/2018	60	start working on the Final Report
T24	26/02/2018	12/03/2018	14	Work on the poster
T25	12/03/2018	21/03/2018	9	Prepare for final presentation

Legends:-

Duration: 

Milestones: 

8.3 Appendix 3: Poster

Prototype self-driving car powered by a Neural network

UNIVERSITY of LIMERICK
OLLSCOIL LUIMNIGH

Lakshan Dadigamuwa
BEng in Electronic & Computer Engineering

Introduction

This project will be exploring one of the main means of improving the safety and efficiency of transportation through the concept of autonomous self-driving cars using artificial neural networks.

Aim

The main aim of this project is to create a self-driving car capable of making system control and navigation related decisions based on information it receives from an on-board camera and an ultrasonic sensor.

The three main objectives this project hopes to achieve are:

- Self-driving on a given track.
- Front collision avoidance.
- Stop sign and traffic light detection.

Method

- This project consisted of three subsystems: input unit (camera, ultrasonic sensor), neural network training unit (computer) and a neural network processing unit (Raspberry Pi).
- Data collected from the camera was used to train a multilayer perceptron (MLP) neural network (NN).
- This trained MLP model was then deployed on the Raspberry Pi to predict the driving behaviours of the car.

Input unit:

- A PiCamera was used to collect training data from a track in order to facilitate the recognition of road signs and lane markings while driving.
- An ultrasonic sensor detects the presence of obstacles in front of the car.

Fig 1: Hardware layout of the toy car

Neural Network Training:

- Data is collected by manually driving the car around a track.
- On every keypress event, a number of training images are captured, converted to Numpy array and reshaped into a single row array.
- These captured frames are then combined with a training label and saved into a Numpy.savez (.npz) file.

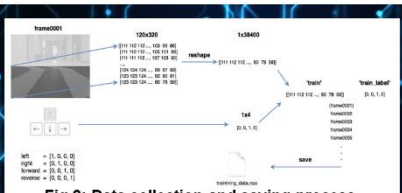


Fig 2: Data collection and saving process

- The collected data is then processed through a multilayer perceptron neural network to create a trained MLP model.
- The neural network is trained in OpenCV using the backpropagation method.
- This method allows for an efficient way of calculating the gradient in neural networks to find the minimum of a function.
- 38,400 nodes were chosen for the input layer and 32 nodes for the hidden layer. Input layers are reduced using gradient descent calculated by backpropagation, to 4 distinct outputs: left, right, forward and reverse.

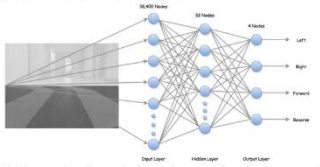


Fig 3: Neural network training by backpropagation

# Of Laps	Amount of Data	Accuracy
1	450 MB	45%
2	800 MB	68%
5	3.2 GB	80.20%

Table 1: Relationship between the data gathered and the accuracy of the MLP NN

The amount of training data had to be capped at around 3GBs due to the limited computational resources such as processing power and the amount of RAM available.

Object Detection:

- A Haar-like feature based cascade classifier was adopted for the object recognition part of the project.
- Hundreds of positive and negative images were processed to find regularities that match Haar features.
- The resulting cascade classifier is simply a collection of multiple stages of filters whose objective is to detect Haar-like features in the form of an xml file.

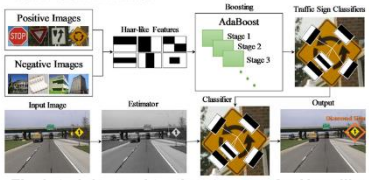


Fig 4: training and testing process for Haar-like feature

Hardware Design

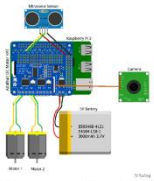


Fig 5: Hardware layout of the system

The trained MLP model was uploaded to the Raspberry Pi to be processed, thus producing a series of driving predictions based on the visual information received through the PiCamera. The predicted output is then transmitted to the motor HAT via the I²C (Inter-Integrated Circuit) communication protocol.

Results

The car is able to recognise traffic light signals and stop signs and sends a signal to the motors to act accordingly. This can only be achieved while the car is stationary. This is because the camera's field of view and the slow processing power of the RPi introduces a time lag which fails to recognise the road signs instantaneously.

Due to the low accuracy of the trained MLP model and the physical limitations of the car itself, the car fails to precisely navigate around a track.

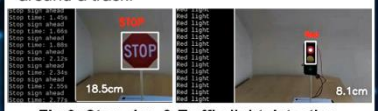


Fig 6: Stop sign & Traffic light detection

Conclusion and personal reflection

The results of this project demonstrated that it is indeed possible to drive a car autonomously with a neural network running on a low power Raspberry Pi.

Overall, the project was successful with minor set backs due to the accuracy of the neural network and the hardware limitations.

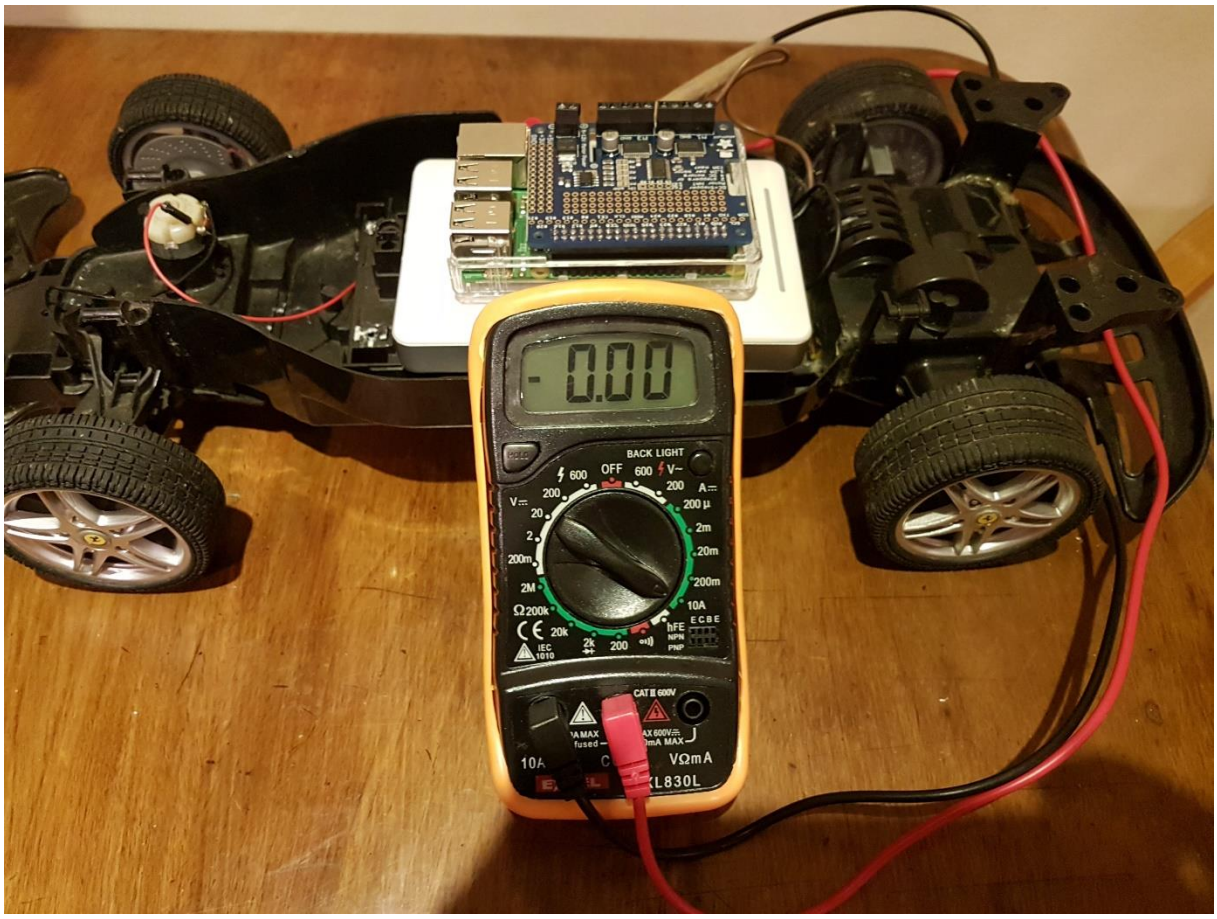
This project could be improved by using a convolution neural network with hardware designed specifically for that purpose.

Acknowledgements

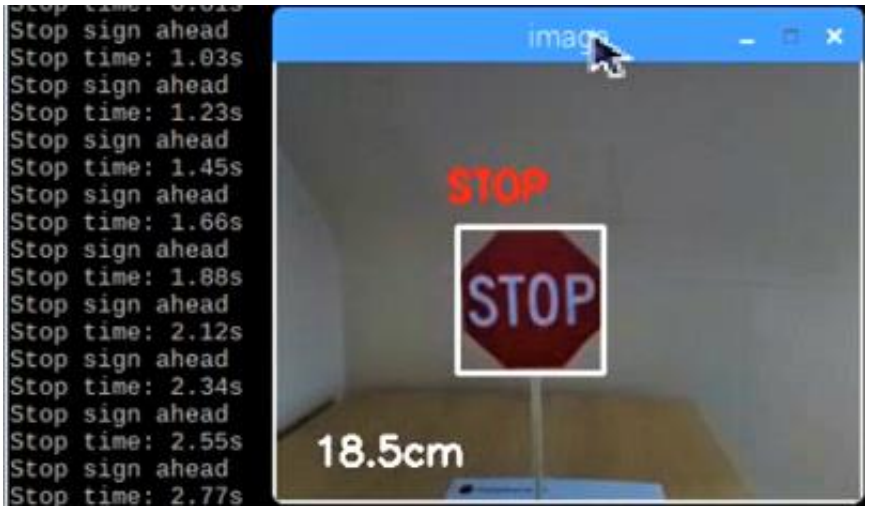
I would like to thank Dr Ciaran McNamee for all his help, guidance and supervision throughout the entirety of this project.

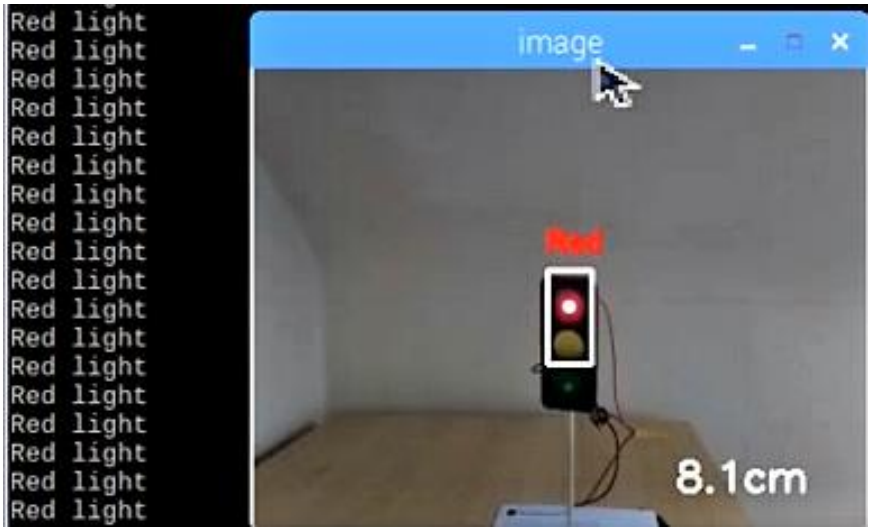
E&CE Department of Electronic & Computer Engineering

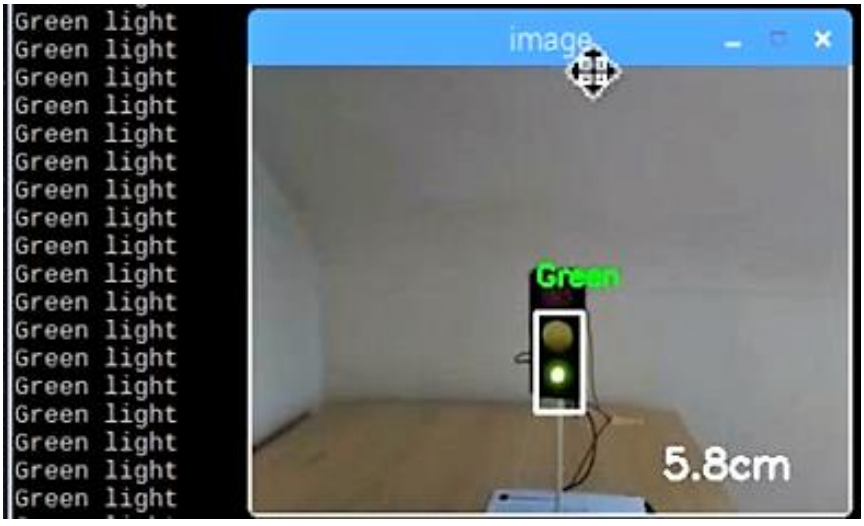
8.4 Appendix 4: Measuring output power from the motor HAT using a Multimeter



8.5 Appendix 5: Road sign detection results







8.6 Appendix 6: Low accuracy cascade classifier trained with 4000 samples