Izgradnja modela autonomnog vozila

Oršolić, Ivan

UNIVERSITY OF ZAGREB
FACULTY OF ORGANIZATION AND INFORMATICS
VARAŽDIN

Ivan Oršolić

BUILDING A SELF-DRIVING RC CAR

MASTER'S THESIS

Varaždin, 2020.

UNIVERSITY OF ZAGREB

FACULTY OF ORGANIZATION AND INFORMATICS

V A R A Ž D I N

Ivan Oršolić

Student ID number: 0016114170

Study programme: Information and Software Engineering

BUILDING A SELF-DRIVING RC CAR

MASTER'S THESIS

Mentor :

Assoc. Prof. Dr. Sc. Markus Schatten

Varaždin, February 2020.

Ivan Oršolić

Statement of originality

I hereby state that my Master's thesis is an original result of my work and that I did not use any sources other than those stated in it through the process of writing it. Through the creation of this work ethically appropriate and acceptable methods and techniques of work have been used.

The author has confirmed this by accepting the regulation in the FOI radovi system

---

Summary

The goal of this thesis is to provide a blueprint for building an autonomous RC scale car model. The thesis covers the important aspects of such model vehicles, the building and integration of the hardware platform necessary to control the vehicle via software and the end-to-end machine learning pipeline which allows a deep learning model to take the input from the hardware platform and control the movement and behavior of the vehicle.

Keywords: artificial intelligence; deep learning; end-to-end machine learning; behavioural cloning; autonomous vehicles; computer vision; embedded hardware;

Table of contents

1. Introduction 1

2. Methods and techniques of work 3

3. Building the hardware platform 4

3.1. RC Car models 4

3.1.1. Scale 4

3.1.2. Body type 5

3.1.3. Electric motors 6

3.1.4. Servo motors 7

3.1.5. Electronic Speed Controller (ESC) 8

3.1.6. Receivers 9

3.1.7. Batteries 9

3.2. Electronics 10

3.2.1. Embedded system 10

3.2.2. Batteries 11

3.2.3. Servo and ESC Driver 12

3.2.4. Storage 12

3.2.5. Cameras 12

3.2.6. Other sensors 13

3.2.7. Wireless 13

3.2.8. Controller 13

3.3. Assembly 14

3.3.1. RC Car assembly 14

3.3.2. Mounting plate assembly 16

3.3.3. Assembling the Jetson Nano 19

3.3.4. Connecting the PCA9685 to the Jetson Nano 22

3.3.5. Connecting the ESC and servo to the PCA9685 24

4. Software 26

4.1. Base operating system installation 26

4.2. Kernel hacking 26

4.2.1. Obtaining the kernel source 27

4.2.2. Adding the external USB firmware blob to the kernel 27

4.2.3. Adding zswap support to the kernel 28

4.2.4. Building the kernel 29

4.2.5. Booting from the custom kernel 30

4.3. DonkeyCar installation 31

4.3.1. Installing and configuring Donkey . . . 32
4.4. Car calibration . . . 32
4.5. Connecting a controller . . . 33
4.6. Testing the Donkey pipeline . . . 34

5. Modding the simulator . . . 36
5.1. Modifying the camera resolution . . . 36
5.2. Updating the default donkey-gym port . . . 38
5.3. Creating 3D version of the RC car model . . . 39
5.3.1. Editing the camera . . . 39
5.3.2. Testing the modified model . . . 39
5.3.3. Adding additional parts and changing the center of mass . . . 39
5.4. Simulator camera calibration . . . 41

6. Camera calibration . . . 42
6.1. Camera distortions . . . 42
6.1.1. The pinhole camera model . . . 42
6.1.2. Types of distortions . . . 44
6.1.3. Removing distortions using OpenCV . . . 44
6.1.4. Finding the camera's intrinsic and extrinsic parameters . . . 47
6.2. Camera calibration implementation . . . 48

7. Lane line extraction . . . 52
7.1. Mapping the 3D world on a 2D sensor plane . . . 52
7.1.1. Implementing a perspective transform . . . 55
7.2. Extracting lane lines from the warped images . . . 58

8. Artificial intelligence . . . 62
8.1. Convolutions . . . 62
8.2. Dense layers . . . 62
8.3. Activation functions . . . 63
8.3.1. Rectified Linear Unit . . . 63
8.3.2. Leaky ReLU . . . 64
8.4. Regularization layers . . . 65
8.4.1. Dropout regularization . . . 65
8.4.2. Batch normalization . . . 65
8.5. Using a custom model with Donkey . . . 65
8.5.1. Creating a first custom model . . . 66
8.6. Optimizing and training a model . . . 69
8.6.1. Data preparation . . . 69
8.6.1.1. Training set, cross validation/dev set and test set . . . 69
8.6.1.2. Splitting the data (size-wise) . . . 69
8.6.1.3. Data distribution . . . 70
8.6.1.4. Determining if a model has high variance or bias (or both) . . . 70

8.6.2. Basic training recipe . . . 71
8.6.2.1. Solving high bias . . . 71
8.6.2.2. Solving high variance . . . 71
8.6.3. Regularization . . . 71
8.6.3.1. Why regularization helps . . . 72
8.6.3.2. Applying regularization layers . . . 73
8.6.4. Defining custom metrics for model performance . . . 73
8.6.5. Minimum requirements of the model . . . 75
8.6.6. Data augmentation . . . 76
8.6.7. Early stopping . . . 77
8.6.8. Hyperparameter optimisation . . . 78

9. Modeling and training the RC car to autonomously drive and perform behaviours 79
9.1. First iteration of the new proposed network architecture . . . 80
9.2. The idea of teaching a vehicle to drive autonomously . . . 81
9.2.1. Implementing the proposed architecture . . . 82
9.3. Second iteration of the new proposed network architecture . . . 87
9.3.1. Recording behavioural data . . . 88
9.3.2. Incrementally adding a behavioural subnetwork to the previously implemented model . . . 89

10. Testing the implemented proposed architecture . . . 90

11. Conclusion . . . 92
11.1. RC car limitations . . . 92
11.2. Power limitations . . . 92
11.3. The simulator . . . 93
11.4. Moving beyond the Donkey platform . . . 93
11.5. Final thoughts . . . 94

Bibliography . . . 99

Table of figures . . . 104

Table of tables . . . 105

1. Appendix: Camera calibration code . . . 106
1.1. getObjectAndImagePoints . . . 106
1.2. calibrateCamera . . . 107
1.3. undistortImage . . . 107

2. Appendix: Custom simulator code . . . 108
2.1. ResolutionSetter class . . . 108

3. Appendix: First custom architecture code . . . 109

3.1. Model class 109
3.2. Model implementation 109

4. Appendix: Advanced model code 111
4.1. Model class 111
4.2. Model implementation 113

5. Appendix: Advanced behavioural model code 115
5.1. Model class 115
5.2. Model implementation 117

1. Introduction

The topic of this Master's thesis is building an autonomous vehicle from scratch, and more specifically, a self-driving RC car. The goal of the thesis is to build a model capable of autonomous driving on the track, while demonstrating the capability to perform behaviors such as lane changing. The thesis will go through the entire process of building such a vehicle, starting from the very RC car model and the embedded hardware platform, to the end-to-end machine learning pipeline necessary for automated data acquisition, labelling and model training.

The main motivation behind the selected topic is the fast-moving progress of applied artificial intelligence (AI) and the predicted importance of autonomous vehicles on the future of humanity, from independent mobility for non-drivers and low-income individuals, reduced pollution, traffic and parking congestion to increased safety on the roads. [1] Autonomous vehicles are also predicted to be relied on in some of the most complex human planned endeavours, such as space exploration [2]. The meteoric rise of AI along with deep learning (DL) methods and frameworks, have made possible the creation of such an autonomous vehicle without expensive laboratories and years of research.

Currently, there are a number of private companies as well as academic groups working on autonomous vehicles and their integration into existing regulations, laws and society itself. Some of the leading companies are Google Waymo, which already has vehicles without a safety driver autonomously driving around Phoenix, Arizona, USA [3], Uber which is also testing driverless autonomous vehicles as of 2020 after some delay due to the autonomous vehicle fatally injuring a pedestrian in 2018 [4] and Tesla which has the most widely used autonomous vehicle system in the world [5], albeit on a less autonomous level than Waymo.

Aside from the technological obstacles that are yet to be surmounted, one big barrier precluding autonomous vehicles on roads are of an ethical and legislative nature. As mentioned earlier, Uber, Tesla and others have already had fatal accidents involving autonomous vehicles, which prompted investigations [6] and a long overdue discussion concerning the liabilities that come with allowing vehicles to autonomously navigate real streets with real humans on them, focusing specifically on impossible situations in which a fatal collision may be unavoidable in which the vehicle will have to decide what course of action to take, from self-sacrifice which would mean sacrificing the passengers the car is transporting to intentionally fatally injuring other traffic participants [7]. Other issues include the cost and scalability of self-driving technology and the standardisation of safety assurance in such vehicles, for which models and frameworks have been proposed just in the last year or two [8].

The advantages and quality of life improvements autonomous vehicles offer range from safer and less congested roads, reduced parking and fewer vehicles per capita to up to several thousands of dollars saved per year in travel time reduction, fuel efficiency, parking benefits and crash costs. [9] Through a public research done by Howard and Dai on 107 likely autonomous technology adopters in Berkeley, California, the adopters are attracted to the potential safety benefits and other amenities, while mostly being concerned about the liabilities explained above, the costs of such technology and losing control over the vehicle and letting it drive itself.

It's obvious, with everything stated above, as well as with the well known rise of artificial intelligence, that the field of autonomous vehicles is at its very beginnings and that it will have an important long term impact on society, through both financial and ethical aspects. The accessibility of such technology should be made more broadly available to researches and students if the field is to continue progressing through increased discussions on important topics and not stagnate in a *winter of autonomous vehicles*, which is one of the reasons behind the topic of this thesis.

2. Methods and techniques of work

Throughout the thesis, many free and open source (FOSS) tools and frameworks will be used, without whom such work could not be feasible in such a short period of time, that writing a Master's thesis allows to have.

The entirety of the implementation of the project is written using the Python scripting language and a number of its libraries and application programming interfaces (APIs). For the mathematical parts programmed in Python, the thesis uses the NumPy library [11], a mathematical library for scientific computing in Python.

For higher-level neural network (NN) modeling and training, the thesis uses Keras [12], a high-level API for rapid NN prototyping, which runs on top of TensorFlow [13], an end-to-end open source ML platform. For various computer vision and image processing tasks, the thesis uses OpenCV [14], an open source computer vision and machine learning library.

The thesis heavily uses, builds on and modifies the DonkeyCar [15] platform, an open source do-it-yourself (DIY) self driving platform for small scale cars, which provides an interface between the higher-level models created with Keras and the low-level RC car electronics and controls. Along with the DonkeyCar platform the Unity game engine is used to modify the Donkey simulator in order to expand its functionalities as needed for the thesis.

The main approach to the neural network architecture modeling is rapid prototyping by collecting smaller data-sets using the simulator, training the model and testing it in the simulator. After verifying the validity of the model and making sure it performs well on smaller data-sets, the model is trained for longer periods of time on bigger data-sets obtained from multiple sources.

3. Building the hardware platform

This chapter describe the components necessary to build the hardware platform which will running all of the software on the RC car, from low-level car controls such as steering and throttle to high-level NN software and models. Multiple alternatives will be considered and explanations will be provided for each selected piece of hardware.

3.1. RC Car models

The process of constructing a scale RC car model could be, and is, a topic on its own. But for the purposes of this thesis, the following sub chapters will focus only on the aspects most important in the context of building an self-driving autonomous RC car.

3.1.1. Scale

Most RC car models are scaled down versions of their real-life equivalent, with the scale expressed as a ratio of the size of the car the model is representing and the size of the model itself (real-life size : RC model size). Some of the most common scales, ordered by size, are: $(1:n), n \in \{8, 10, 16, 18\}$. The trade off between smaller and bigger scales is:

- a larger scale model can carry more electronics without damaging the electronic motors with the added weight
- a smaller scale model does not need as large tracks as the larger models

Larger scale models can also easily reach speeds up to 100 kilometres per hour, which may also be a consideration if high speed autonomous driving is preferred.

For the purposes of this thesis, a scale size of $1:16$ would have been enough to carry the hardware platform while maintaining safe electronic motor operation, but a $1:10$ scale was chosen for future upgradeability and higher maximum speed the model can achieve.

Figure 1: RC scale comparison
(Source: ModelToyCars)

3.1.2. Body type

RC car body types can roughly be grouped into four distinct categories:

- Race/Street body types are the fastest type, made for paved, flat surfaces and meant for on road use only. They are also the most stable type for on-road uses, due to their low center of gravity.
- Buggy body types are both an off-road and on road RC car. They can go off-road, unlike the race/street type, but at the price of being slower than them on road.
- Truggy body types are also an in-between body type, but they lean more to the off-road role, as opposed to the buggies. They are faster off-road than buggies, but slower than them on-road.
- Truck body types are monster truck models. They are bad for on-road self-driving applications, due to their high center of gravity, because of which they easily flip on high-speed turning on roads. Off-road, they are the fastest RC type.

Figure 2: Race/street body type
(Source: Tamiya)

Figure 3: Buggy body type
(Source: RampageShop)

Figure 4: Truggy body type
(Source: Goolsky)

Figure 5: Truck body type
(Source: Traxxas)

For the purposes of this thesis, the race/street body type was chosen for its low center of gravity which provides the greatest possible stability for on-road purposes, as well as its maximum speed of 100 kilometres per hour (for the particular model chosen for the thesis). The goal of the thesis is to emulate a real-life scenario of autonomous driving on roads, which is why the off-road capabilities were deemed unnecessary and the other body types were not considered.

3.1.3. Electric motors

The electric motor is one of the most important parts of the RC car, but in the context of this thesis, the only thing of interest is whether the motor is brushed or brushless. A brushed motor uses a set of wound coils as its rotor, which act as an (reversible) electromagnet with two poles, and a brushless motor uses a permanent magnet as its rotor. [21]

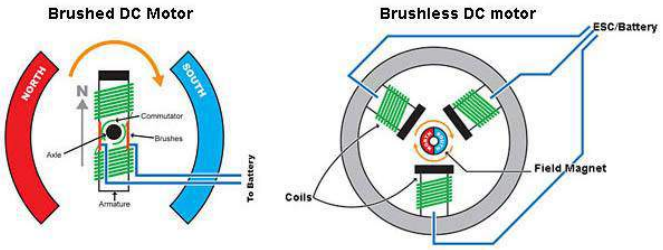

Figure 6: Brushed vs. brushless comparison
(Source: ThinkRC)

Much of the technical complexities of both types can be disregarded while focusing on only the following trade-offs between the two:

- Brushed motors are cheaper, simpler and better for off-road, but they are also heavier, bigger, have worse power efficiency and wear out faster. [21]
- Brushless motors have long lifespan, much better speed and handling and better power efficiency but they are much more expensive and worse for off-road uses. [21]

It is important to note that motors can easily be swapped, which makes deciding between the two much easier. For this thesis, a brushed motor variant was chosen mainly due to the much lower price. Most RC cars come with an electric motor, and if they do not, any motor that the chassis manufacturer has listed as compatible can be used, be it a brushless or a brushed one.

3.1.4. Servo motors

RC cars use a servo motor to control steering wheels of the car. The official DonkeyCar documentation [22] states that the servo typically expects around 4.8V to 6V input on the power wire (varies by car) and a PWM control signal on the signal wire. Typically, the three wires are colored black-red-white, or brown-red-yellow, where:

- the dark wire (black/brown) is ground
- the center wire (red) is power,
- the light wire (white/yellow) is control.

The official DonkeyCar documentation [22] also states that the control signal is RC-style PWM, where one pulse is sent 60 times a second, and the width of this pulse controls how left/right the servo turns. When this pulse is:

- 1500 microseconds, the servo is centered;
- 1000 microseconds, the servo is turned all the way left (or right)
- 2000 microseconds, the servo is turned all the way in the other direction.

Figure 7: Servo motor
(Source: Solarbotics)

Other that it is necessary for a functional RC car model, there are no specific characteristics one should look for when obtaining a servo motor for that use. Most RC car models come with one, and if they don't, any servo that fits in the car chassis can be used.

3.1.5. Electronic Speed Controller (ESC)

The official DonkeyCar documentation [22] states that: "*The role of the ESC is to take a RC PWM control signal (pulse between 1000 and 2000 microseconds) in, and use that to control the power to the motor so the motor spins with different amounts of power in forward or reverse.*"

It also states that the ESC PWM signals are the same as for the servo motor, where 1500 for an ESC would mean that the electric motor will be stopped.

The ESC connects to the RC car battery and provides power to the servo motor.

Figure 8: Electronic speed controller
(Source: Tamiya)

For the purposes of this thesis, any ESC that comes together with an RC car can be used. If it does not come provided by the manufacturer of the RC car and if a brushless motor is used, it is important to pay attention that the separately obtained ESC can be used with brushless motors. Otherwise, any ESC that the RC car manufacturer has listed as compatible can be used.

3.1.6. Receivers

A wireless receiver allows an RC car to be controlled via a wireless controller. Many RC cars come with an receiver, which can operate on any number of frequencies, the most common one being 2.4GHz.

Figure 9: Wireless receiver
(Source: RC-Hobbies)

For the purposes of this thesis, a receiver is not necessary, and in fact, it is better if an RC car without a receiver is used since it minimises the risk of the RC car electronics being tailor made to that specific model of receiver. If an RC car that comes with a receiver is used, it can be taken out.

3.1.7. Batteries

Detailed explanations of different battery technologies are out of the thesis scope and will be disregarded, while focusing only on the things of importance in the context of the thesis, which would be deciding between Nickel Metal Hydride batteries (NiMH) and Lithium Polymer batteries (LiPo) and the amount of energy stored in them:

- LiPo batteries provide much longer driving times, but are also more expensive.
- NiMh batteries provide much shorter driving times, but are more cheaper.

The best option are LiPo batteries, regardless of their increased price, since driving time is one of the most important factors of an RC car. While choosing a battery it is important to make sure it matches the voltage of the ESC the RC car is using, and to also obtain an appropriate charger for the batteries.

Since batteries can be damaged by excessive discharge and carry the risk of exploding during charging, it is highly recommended to buy additional safety equipment, most notably a LiPo charging bag and a LiPo alarm.

Figure 10: LiPo batteries and charger (Source: FlashRC)

Figure 11: LiPo discharge alarm (Source: GreensModels)

Figure 12: LiPo charging bag (Source: FuseModel)

A LiPo alarm is connected to the LiPo battery via the provided wiring ports and can be set to sound a loud alarm when the battery gets discharged between a certain threshold, in order to stop the battery from being damaged by excessive discharge.

A LiPo charging bag is a fire and explosion resistant bag inside which a LiPo battery is placed while charging, to minimise the spread of fire in an event of an explosion.

The RC car described built for this thesis uses two LiPo batteries, a 3000mAh and a 6000mAh 7.4V batteries.

3.2. Electronics

Aside from an RC car model, other electronics as well as an embedded system is necessary to enable controlling the car via software and to deploy and run ML models and control the car with them. The following sub chapters explain what such components were used through the process of building an RC car for this thesis, along with possible alternatives.

3.2.1. Embedded system

The most important part of the hardware platform is the embedded system which runs the operating system which will in turn be running all the software necessary for the end-to-end machine learning pipeline and the lower level software which allows RC car control.

There are multiple possible alternatives to choose from, with varying graphics processing units (GPU), random access memory (RAM) sizes and central processing units (CPU):

- Nvidia Jetson Nano with a 128-core dedicated GPU (768 GFLOPS), 4GB of shared LPDDR4 RAM and a Quad-core ARM CPU (1.43 GHz)
- Raspberry Pi 4B without a dedicated GPU, 4GB of LPDDR4 RAM and a Quad core ARM CPU (1.5GHz)
- Google Coral TPU Dev Board with an dedicated TPU co-CPU(4 TOPS), 4GB of LPDDR4 RAM and a Quad core ARM CPU (1.5GHz)

Figure 13: Nvidia Jetson Nano
(Source: Franklin)

Figure 14: Google Coral Board
(Source: Coral)

Figure 15: Raspberry Pi 4 Model B
(Source: RaspberryPi)

At first look, the Google Coral Board seems like the most powerful choice, but according to benchmarks made by Nvidia [29] the board cannot run a lot of complex models and full TensorFlow models, and for the ones that it can, it doesn't have a great advantage. There are some instances where using a Google Coral board is much better than using a Jetson Nano, but they do not fit into the purposes of this thesis.

The Raspberry Pi is the weakest of the three alternatives, but it is also the cheapest, costing half the price[31] of the Jetson Nano[29] and three times less than the Coral[30], but it simply lacks the processing power for the complex models this thesis implements later on.

The main difference between the Nano and the Coral is that the Nano is more similar to a full computer with a dedicated GPU that can do generalised tasks, and the Coral is a specialised board that performs much better on a very small subset of tasks.

The best choice for the purposes of this thesis is the Jetson Nano.

3.2.2. Batteries

The Jetson Nano uses 5 watts (W) of power through its microUSB port, but can also use up to 10W of power through the barrel jack adapter, which effectively doubles the processing power of the dedicated GPU.[29]

It can be powered from the RC car battery, but the necessary power converting components would be complicated to build and in case the RC loses power, the entire embedded system would lose power as well.

For those reasons, the hardware platform described in this thesis uses a 20 000 milliampere hour (mAh) battery which can provide up to 2 amperes (A) of power through a dedicated USB-A port, or up to 65W of power using a special power-delivery USB-C port.

3.2.3. Servo and ESC Driver

In order to control the RC car servo motor and ESC with the Nano, a specialised piece of equipment that will serve as an interface between the twoo needs to be used. The platform described in this thesis will use a PCA9685 16 Channel 12-bit pulse width modulating (PWM) servo driver board. [32] This board allows the Jetson Nano to send PWM signals to the RC Car by sending Inter-Integrated Circuit (I2C) signals through its general-purpose input/output (GPIO) header. This will be explained in more detail in the following chapter.

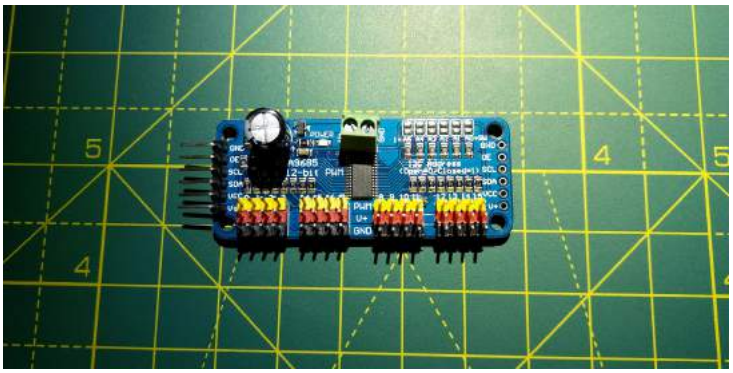

Figure 16: PCA9685

3.2.4. Storage

The Jetson Nano requires the use of a MicroSD card as its primary storage component from which it runs its operating system (OS) and all other software. [29]

For the purposes of this thesis a 64GB MicroSD card will be used, but an external solid-state drive (SSD) enclosed in an USB 3.1 can, and in this thesis will, be used to achieve better storage performance, which will require creating a custom Linux kernel, which will be explained in the following chapter.

3.2.5. Cameras

The hardware build also requires a camera that will be used to gather images of the track the RC car will be driving on, which will be passed as input to an end-to-end machine learning pipeline which will then infer appropriate car control values.

The Jetson Nano supports the use of both USB and MIPI CSI-2 cameras. [29] The build in this thesis will be using an USB action camera with a 170-degree field of view (FOV). Any USB camera and MIPI CSI-2 cameras that the Jetson Nano manufacturer has stated as compatible can be used, while keeping in mind that bigger FOV is desirable in the context of this thesis.

3.2.6. Other sensors

Aside from cameras, other sensors can be used to enable the creation of more complex and accurate models. Although no additional sensors are used in the build described by this thesis, the following sensors are relatively inexpensive and could provide much better results in the context of autonomous driving:

- An inertial measurement unit that uses a combination of accelerometers, gyroscopes, and sometimes magnetometers to measure the speed and direction the car is travelling in, as well as other forces acting on the car.
- An ultrasonic proximity sensor that measures the distance between an obstacle and the car by measuring the time it takes an ultrasonic signal to reach and bounce back from the obstacle to the car.

3.2.7. Wireless

To allow wireless communication with the Jetson Nano, and in turn, the RC car, an M.2 key E wireless card can be built into the Jetson Nano, which would allow Bluetooth and 2.4 GHz WiFi communication between the Nano and a dedicated host personal computer (PC).

The build described in this thesis uses an Intel 8265 M.2 card to allow wireless communication. Other cards that the Nano manufacturer has listed as compatible can be used.

3.2.8. Controller

In order to obtain the most accurate possible driving data, use of a gamepad controller is recommended, due to its high resolution analog inputs. The build in this thesis uses an Xbox One controller, but other controllers listed in the following chapter can be used.

3.3. Assembly

The last step in building the hardware platform is assembling all of the components described previously. The following sub chapters will describe the assembly process.

3.3.1. RC Car assembly

The RC car used for this thesis is a Tamiya TT-02 1:10 scale model car that comes as an ready-to-race (RTR) kit. Any other car that has an ESC and a servo driver can also be used.

Figure 17: PCA9685
(Source: Lindinger)

Most RC cars are sold completely unassembled, with the manufacturer providing detailed instructions on assembly. The assembly process can vary by complexity and the time necessary to complete it.

By contrast, RTR kits come either fully assembled and ready-to-race out of the box, which means they require very little assembly. The above mentioned TT-02 kit required only the assembly of minor chassis elements and wheels.

The figures below shows parts of the assembly process for the TT-02 RTR kit:

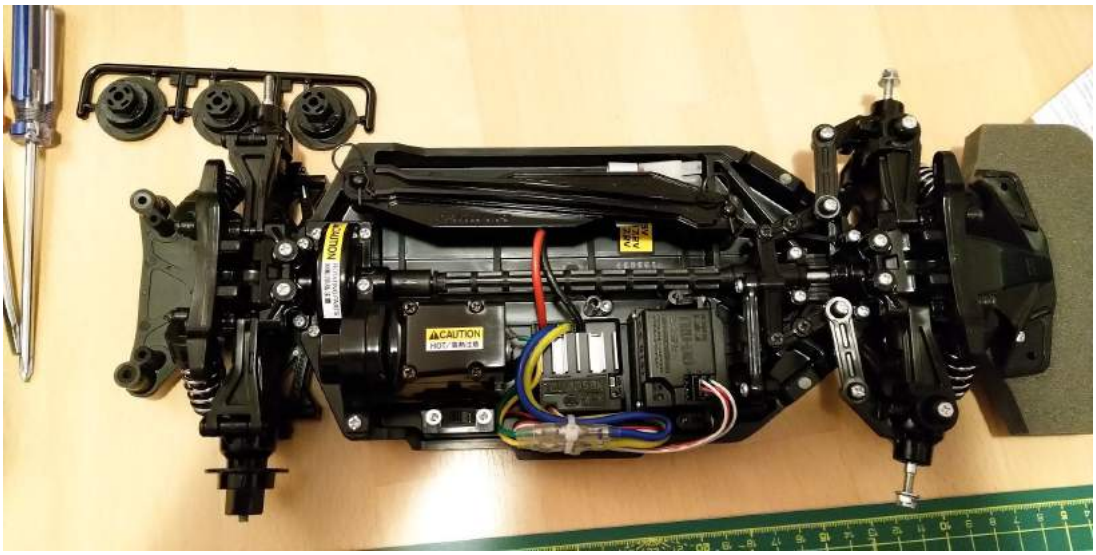

Figure 18: TT-02 RTR Assembly

Figure 19: TT-02 RTR Assembly

3.3.2. Mounting plate assembly

In order to mount the electronics assembly on the RC car, creating a mounting plate as a platform for the electronics is necessary. Two easy ways to do so is to either 3D print custom designed plates or to just create a mounting plate using a piece of aluminium, plexiglass or similar material.

During the process of building the mounting assembly for this thesis, two approaches were used: 3D printing custom plates and creating multi-levelled mounting plates using a special type of aluminium.

The following 3D model was created and used as a guideline for both approaches:

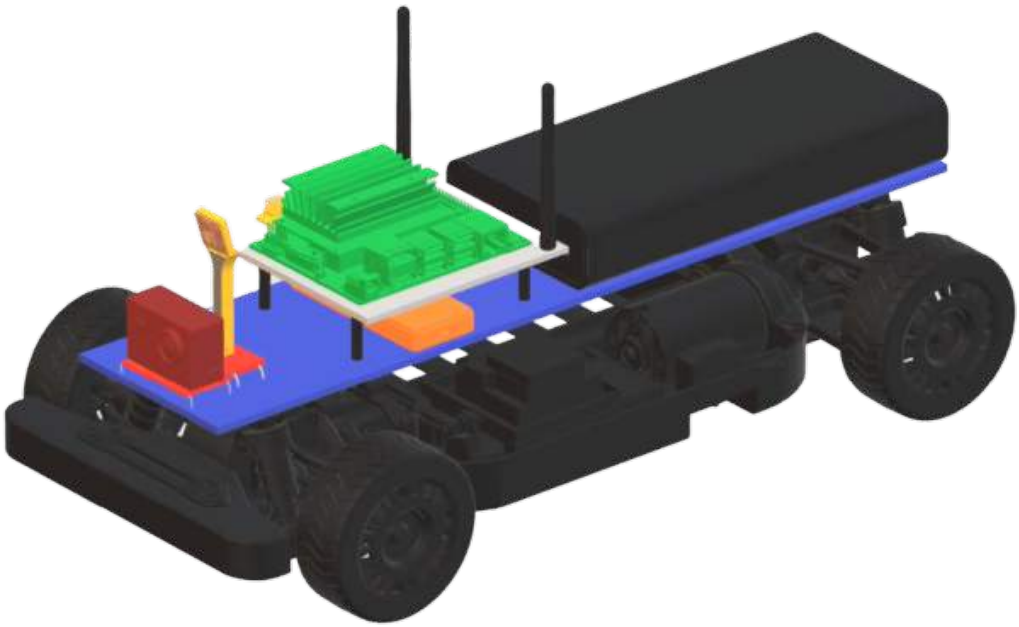

Figure 20: 3D model of the hardware assembly

The 3D printed mounting plates were created using polylactic acid (PLA) filament. After some time of usage, the 3D printed parts proved to be fragile upon RC car crashes and were replaced by a hand-made improved version using a special type of aluminium that proved to be much more stable and resistant to crashes.

The figure below shows the 3D printed mounting plate parts and the final result after mounting all of the electronics on the plates:

Figure 21: 3D printed mounting plates

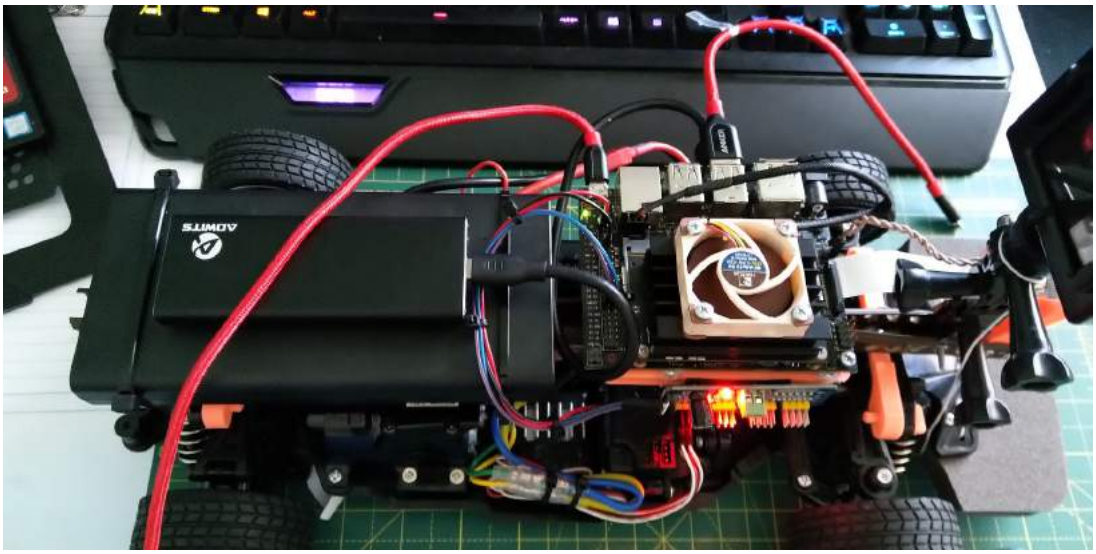

Figure 22: The first version of the hardware assembly

It is visible from the figure above that the electronics were relatively unsafely mounted on the car, with the center of mass being unevenly distributed due to a lack of a larger mounting plate on which the battery could be better positioned on. Also, the approach shown above did not allow for antennae usage, which would need to be mounted on a very secure surface since their connectors are relatively sensitive to impacts.

The next and final iteration of the mounting plate design on which the hardware assembly is secured on used very light but very strong aluminium, which was cut to the proper size and drilled to allow a second level to be mounted which used a softer, but rigid polyurethane sheet on which the embedded system was mounted on. The figure below shows the aluminium sheet used:

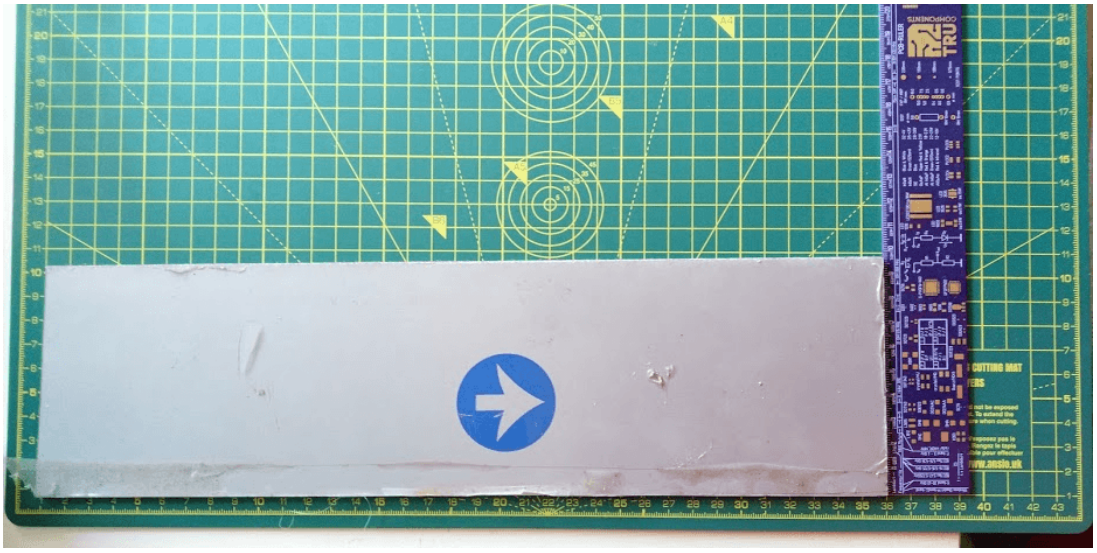

Figure 23: Aluminium mounting plate

And the figure below shows the final iteration of the assembly, which also incorporated wireless antennas.

Figure 24: The final assembled version of the RC car

3.3.3. Assembling the Jetson Nano

After assembling the RC car and the mounting plates, the rest of the electronics can be assembled together and mounted on the RC car. The first step is the Jetson Nano, to which a MicroSD, wireless card and antennae need to be plugged in.

To get to the MicroSD and M.2 wireless card slot on the Nano board, first it's necessary to remove the two screws shown in the figures below, pop the module board up and pull it out.

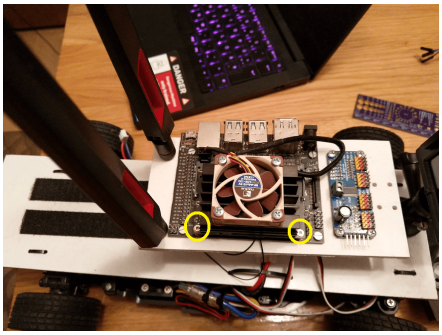

Figure 25: The two screws holding the module in place

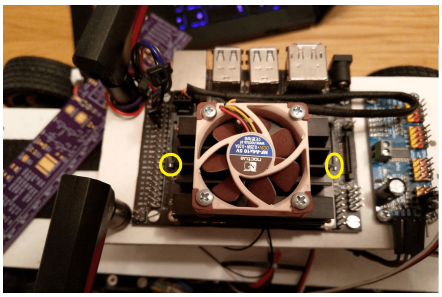

Figure 26: Latches that hold the module in place

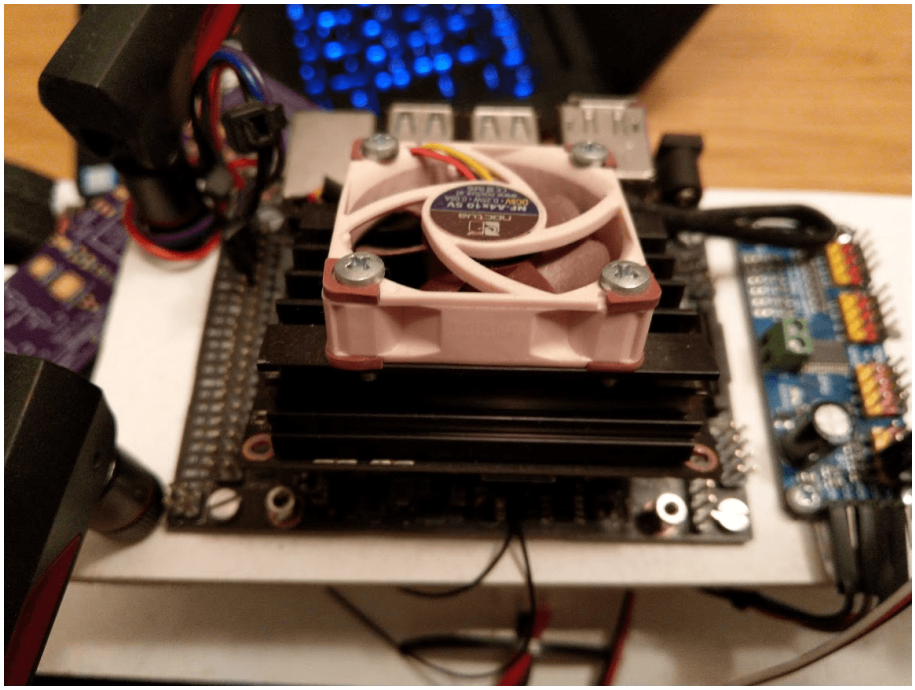

Figure 27: The module pops out

The MicroSD card slot can be seen after pulling out the module, as well as the M.2 slot where the wireless card goes:

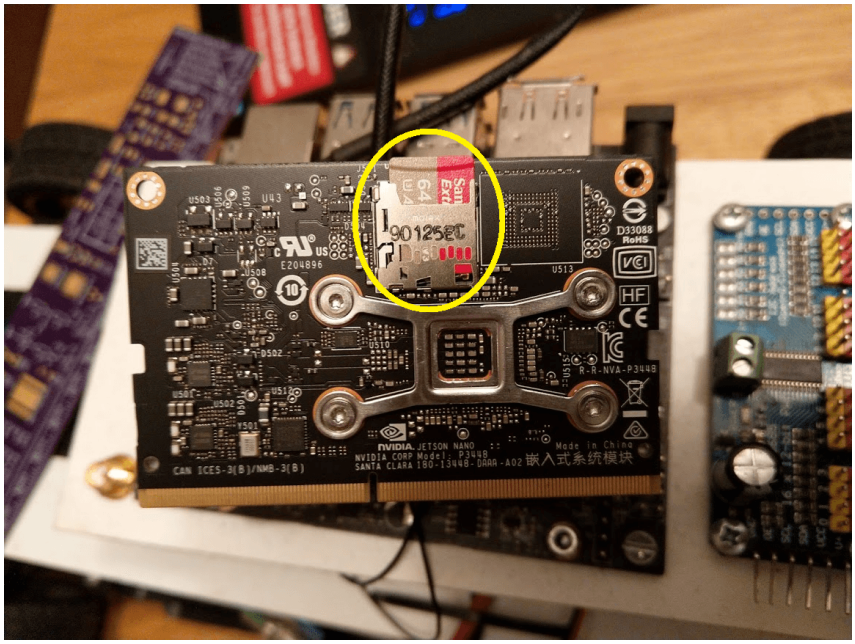

Figure 28: MicroSD Slot

This is also a good time to install a fan on the Jetson Nano to allow for later overclocking with the previously mentioned barrel jack adapter that can take up to 10W of power, effectively doubling the dGPU processing power:

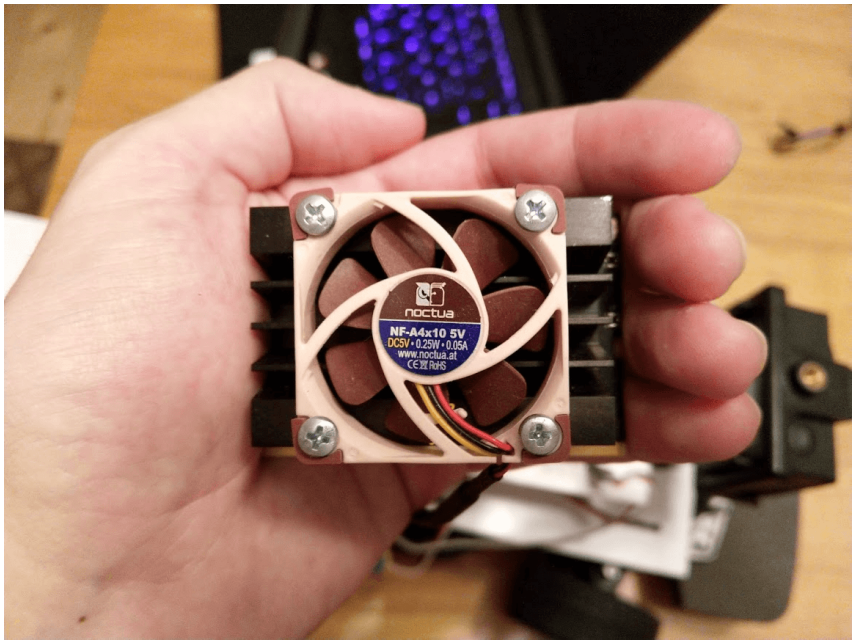

Figure 29: Installing a fan on the Nano module

The installed fan also needs to be connected via the 3-pin (or 4-pin) connector to the Jetson Dev board just below the Ethernet port:

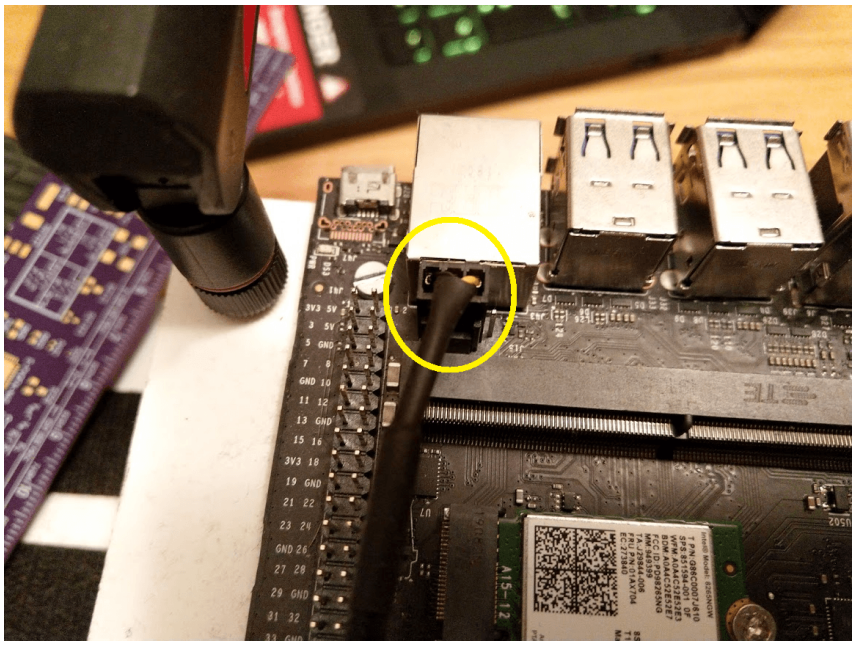

Figure 30: Fan PWM connector

The M.2 wireless card (with IPEX4 antennae connectors plugged in) should be installed on the Jetson Dev board just above the slot into which the Nano module gets plugged into:

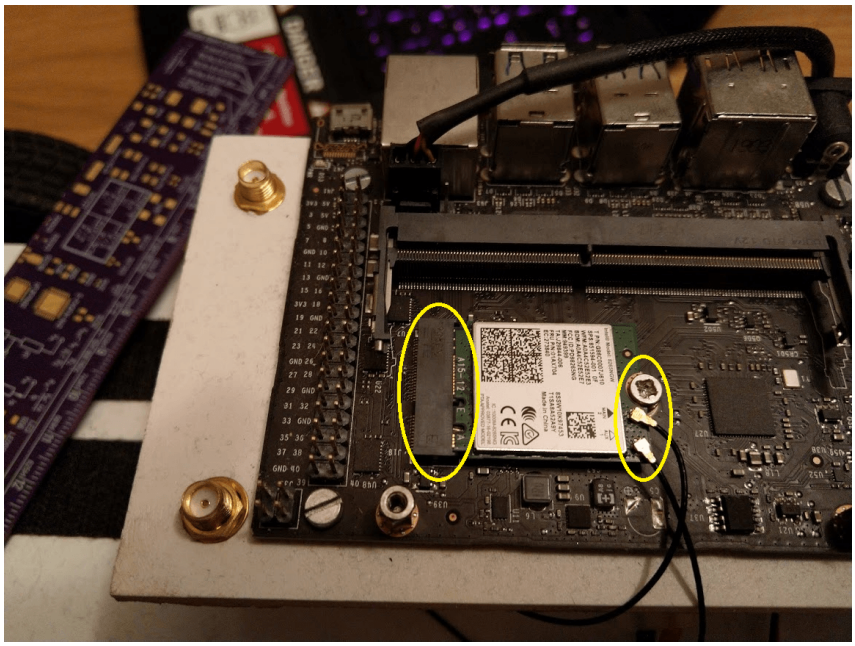

Figure 31: Installing the M.2 card

Finally, this the Nano module can be plugged back in and the antennae connectors and screws secured to the mounting plates, with the hardware assembly looking something similar to the following figure:

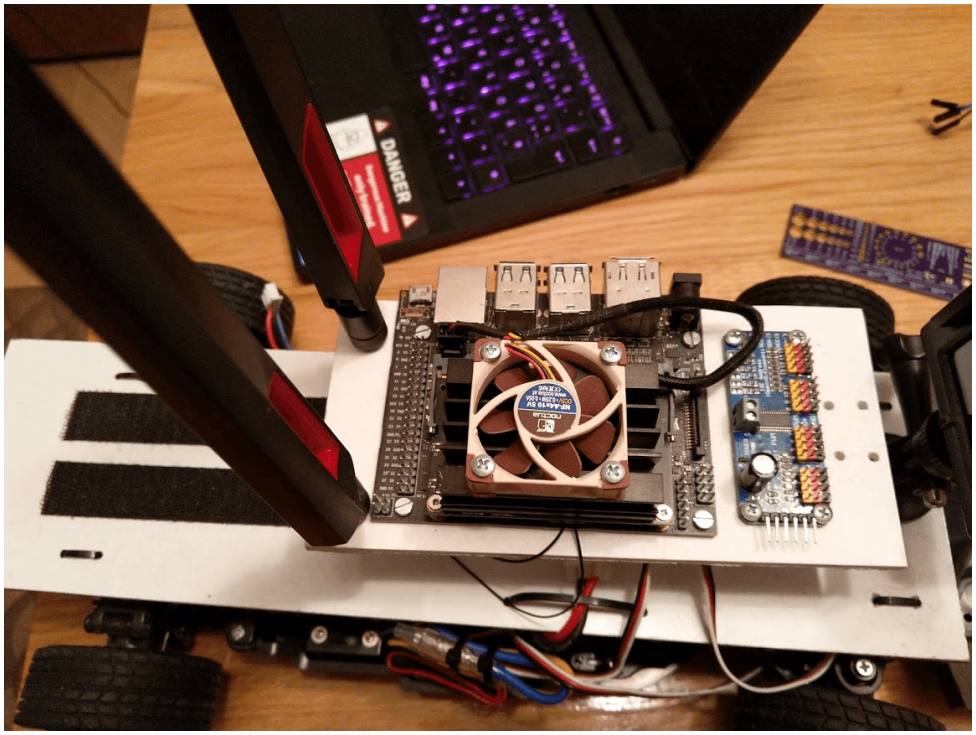

Figure 32: Jetson Nano assembled

3.3.4. Connecting the PCA9685 to the Jetson Nano

The next step is connecting the PCA9685 board to the Jetson Nano using its I2C header and the PCA9685 board to the RC car ESC and servo motor, as shown in the diagram below:

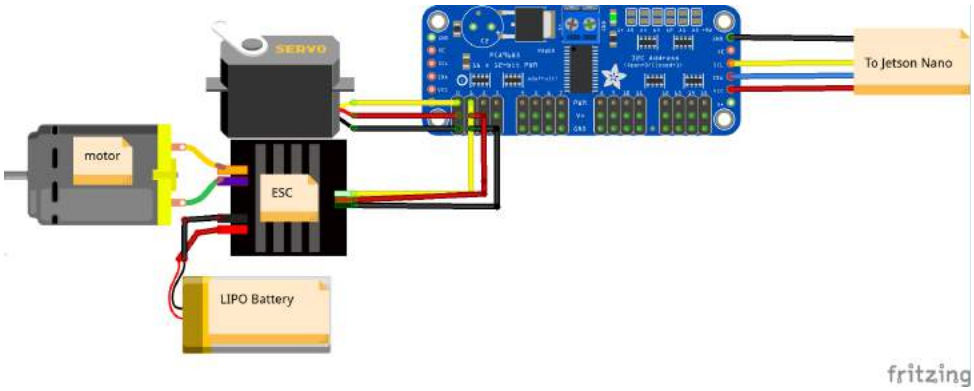

Figure 33: Jetson Nano assembled

As shown on the diagram above, the I2C header highlighted below needs to be connected to the Jetson Nano GPIO header:

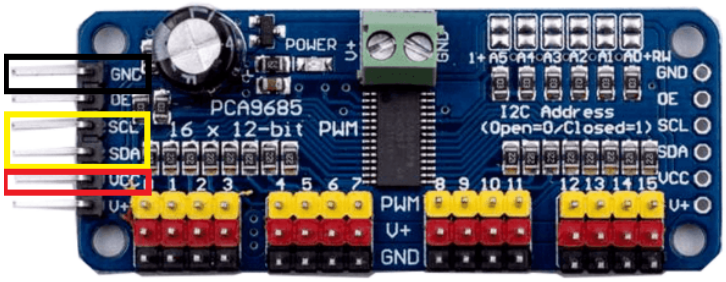

Figure 34: Jetson Nano assembled

By consulting the Jetson Nano pinout [34] the two I2C buses that the Jetson Dev Board has can be identified, highlighted in yellow below, the ground pins are highlighted in black and the 3.3V pins are highlighted in red:

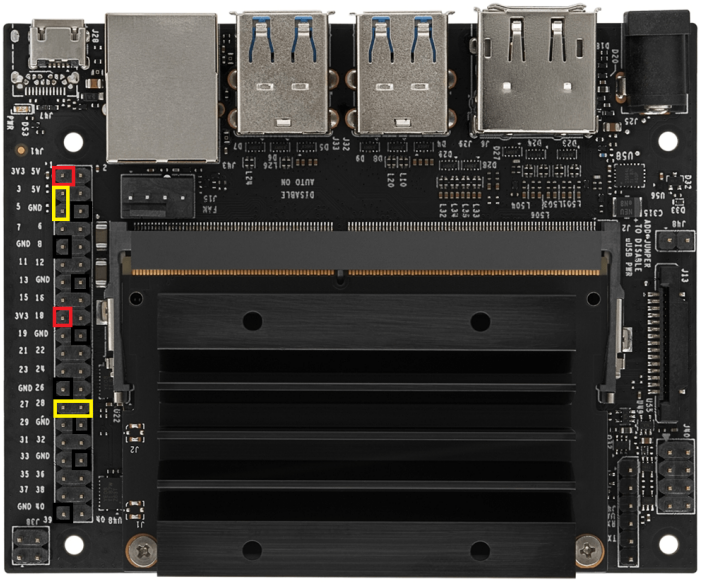

Figure 35: Jetson Nano pinout

- 3V3: labeled red, position 1 and 17, connect PCA VCC pin to either of them
- GND: labeled black, all over the place, connect PCA GND pin to any of them
- I2C busses: labeled yellow, there are two of them:
  - Bus 0: position 27 (SDA) and position 28 (SCL)
  - Bus 1: position 3 (SDA) and position 5 (SCL)
- Connect the PCA SDA and SCL pins to either bus
- Note: the SDA/SCL pins come in pairs, a pair/bus has to be chosen and used together

Finally, the wires can be connected to the Nano header:

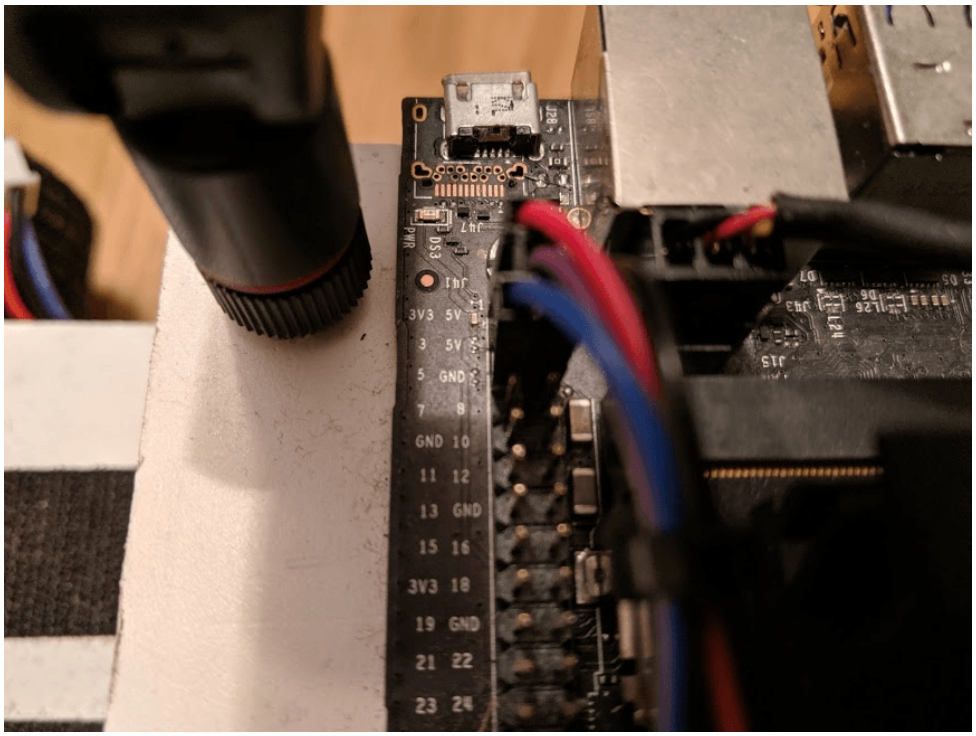

Figure 36: Jetson Nano I2C bus connected

3.3.5. Connecting the ESC and servo to the PCA9685

The last step in the hardware chapter is connecting the ESC and servo motor from the RC car to the PCA9685. First it's necessary to find the ESC and servo connectors on the RC car. This can be either done by tracing the connector wires from the components:

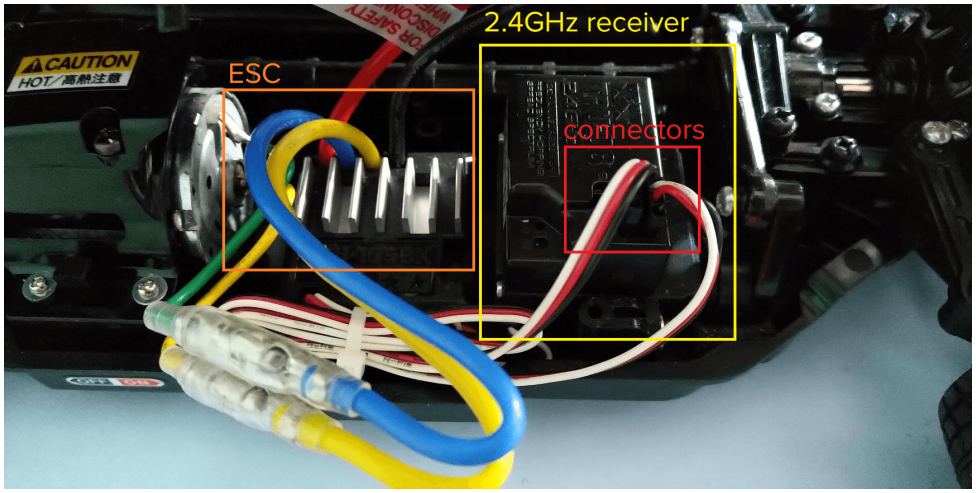

Figure 37: The ESC and servo motor connectors in an RC car

If the RC car comes with a receiver out of the box, the connectors can easily be found by looking into the wireless receiver: The connectors need to be plugged out from the receiver

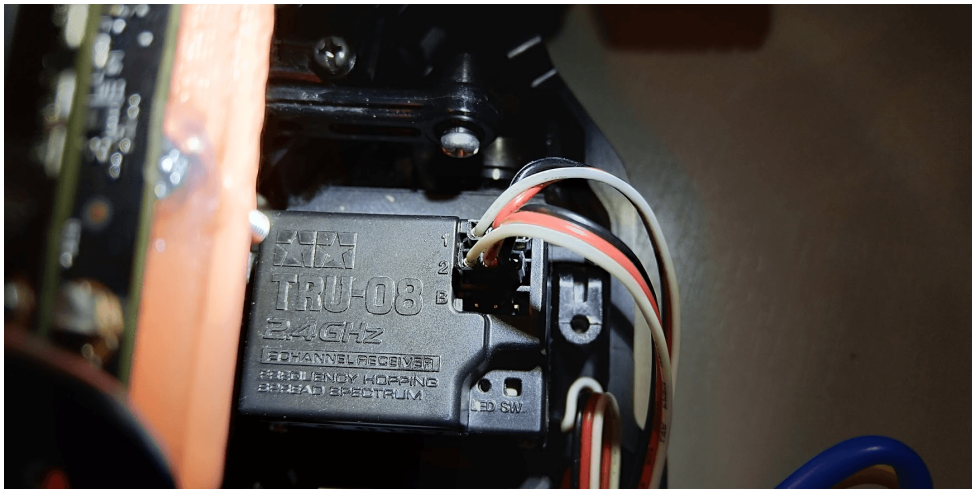

Figure 38: The RC car receiver with ESC and servo connectors

and connected to the PCA9685 board into any two channels, preferably the zeroth and first, for easier reference and with care taken that the color coding of the connectors is aligned with the PCA9685 board:

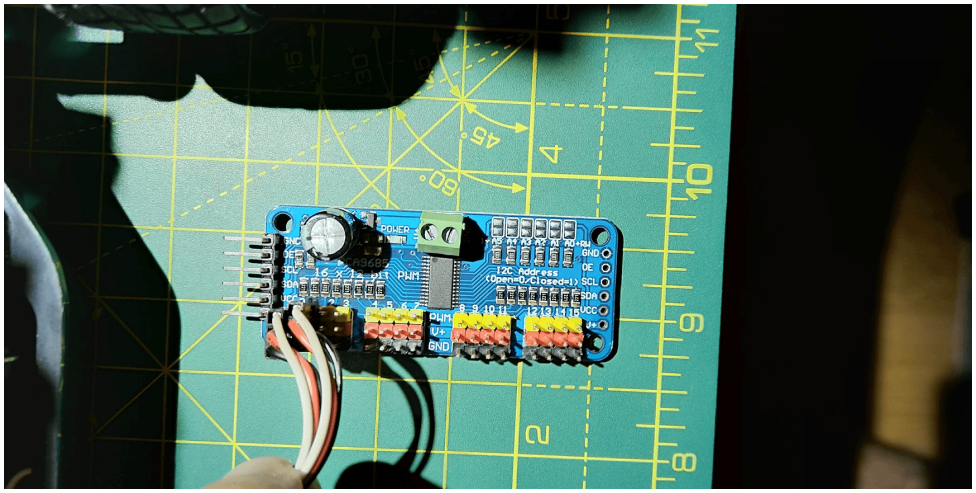

Figure 39: The PCA9685 connected to the ESC and servo

After connecting the PCA9685 to the ESC and servo motor of the RC, the hardware assembly part of the build is complete.

4. Software

This chapter describes the process of installing all of the software necessary in order to create and train models using an end-to-end ML pipeline which takes inputs from the camera and outputs the steering and throttling controls directly to the RC car.

4.1. Base operating system installation

The first step is installing the base operating system on the Jetson Nano and can be performed very easily by downloading the Nvidia SD card image from the official Jetson Nano documentation and flashing it to the MicroSD card. [35]

4.2. Kernel hacking

This sub chapter is optional, but improves the Jetson Nano system performance by allowing the OS to boot from an external SSD which is much faster than the MicroSD card in both latency and sequential reading speeds. This sub chapter follows Syonyk's guide to patching the Jetson Nano firmware.[36]

The idea is to use an external SSD drive as the root file system for the OS, but that usually is not possible due to:

- The USB 3 ports requiring the kernel to load USB firmware on boot to enable them to work, which means the USB ports won't work until the device boots up
- That same USB firmware is stored on the root file system, which should be on the external SSD if it's going to be the boot device for the OS
- Which would mean that the firmware needs to be loaded from the SSD in order to use the SSD

The solution to the above problem is to patch the Linux kernel and embed the firmware needed right into it, so it doesn't need to read it from the root file system, which can then be freely put on the external SSD.

While patching the kernel, there is another thing that could provide a noticeable performance boost, zswap, which also requires adding same-filled page merging support for to the kernel.

All of the below described procedures should be ran on the Jetson Nano, and they take a while to perform, since taking kernels apart, patching them and building them takes a while even on regular workstations or servers, and doing it on an embedded device makes it even more time consuming.

4.2.1. Obtaining the kernel source

The Jetson Nano does not run the stock Linux kernel, rather a custom Nvidia build of it, for which Nvidia provides the source code, which can be found under board support package (BSP) sources under Jetson Nano on the Nvidia Jetson Linux Driver Package (L4T) download page.[37]

After downloading the kernel source files, they should be unpacked and the directory should be changed to the current kernel version directory:

```
tar -xf public_sources.tbz2
cd ~
tar -xf ~/Downloads/public_sources/kernel_src.tbz2
cd ~/kernel/kernel-4.9
```

To enable the kernel that is being modified to work with the Nano board, the current kernel (that the Nano is running) needs to it:

```
zcat /proc/config.gz > .config
```

4.2.2. Adding the external USB firmware blob to the kernel

The external USB firmware blob that allows the OS to use USB devices needs to be copied to the new kernel:

```
cp /lib/firmware/tegra21x_xusb_firmware ./firmware/
```

A Linux kernel and its modules can be modified using the menuconfig tool, shown in the figure below, can be used:

```
sudo apt-get install libncurses5-dev
make menuconfig
```

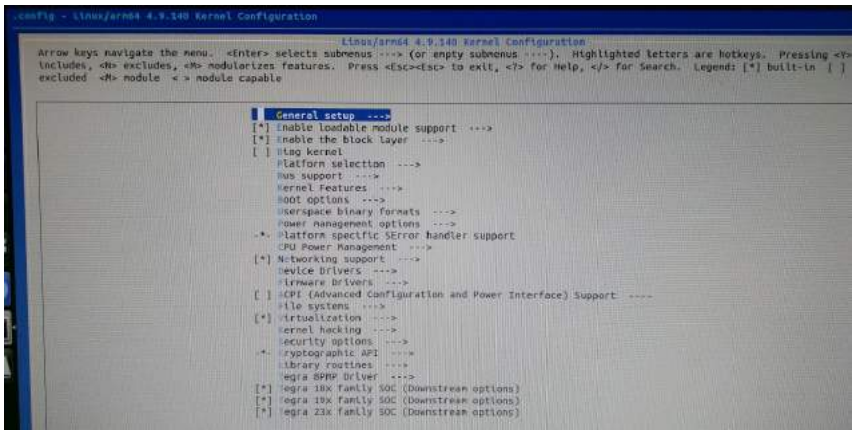

Figure 40: Linux kernel configuration using menuconfig

When selecting kernel features in menuconfig, a star (*) denotes a built-in feature and M denotes a module. Stars should be selected for features that need to be built in.

To build in the external USB firmware blob into the kernel binary:

- Select: ***Device Drivers***
- Select: ***Generic Driver Options***
- Select: ***External firmware blobs to build into the kernel binary***
- Select: ***Type: tegra21x_xusb_firmware***
- Exit and save the new configuration

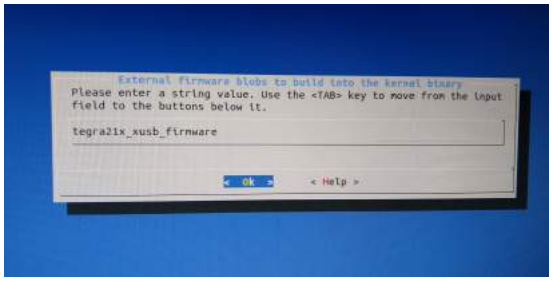

Figure 41: Adding the external firmware blob to the kernel binary

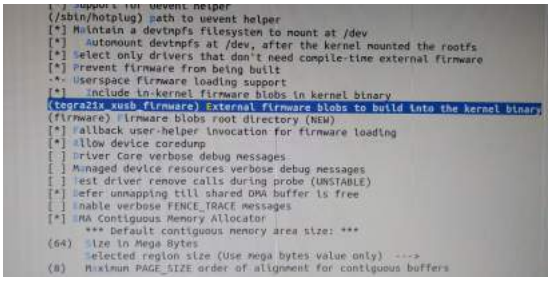

Figure 42: Saving the added external firmware blob

4.2.3. Adding zswap support to the kernel

First, some features need to be enabled using menuconfig:

- Select: ***Kernel features***
- Select: ***Enable frontswap to cache swap pages if tmem is present***
- Select: ***Compressed cache for swap pages (EXPERIMENTAL) (NEW)***
- Select: ***Low (Up to 2x) density storage for compressed pages***

The next step is back-porting same-filled page compression from a newer version of the kernel to the 4.9 that is being modified. The new feature can easily be patched into the older kernel by finding the zswap source code on the Linux GitHub repository, choosing the commit that implements same-filled page compression[38] and adding a .patch suffix at the end of the commit URL to get the patch file that adds the feature to a kernel:

```
# Downloading the patch file
wget https://github.com/torvalds/linux/commit/
a85f878b443f8d2b91ba76f09da21ac0af22e07f.patch

# Changing to the kernel directory
cd ~/kernel/kernel-4.9
patch -p1 < ~/Downloads/a85f878b443f8d2b91ba76f09da21ac0af22e07f.patch
```

```
# Replacing the not yet implemented memset_l call with a regular memset
sed -i 's/memset_l(page, value, PAGE_SIZE \/ sizeof(unsigned
↪  long));/memset(page, value, PAGE_SIZE);/g' mm/zswap.c
```

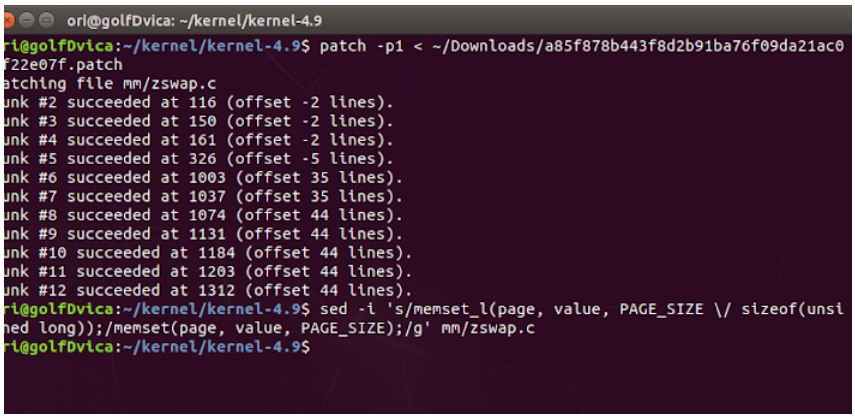

Figure 43: Patching the kernel

4.2.4. Building the kernel

After adding all necessary changes to the kernel, it can be built and placed on the boot partition by running:

```
make -j5 # -j denotes the number of threads
sudo make modules_install
sudo cp /boot/Image /boot/Image.dist
sudo cp arch/arm64/boot/Image /boot
```

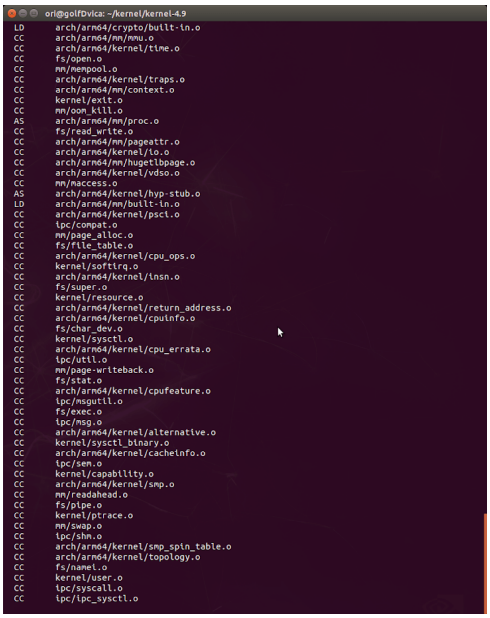

Figure 44: Building the kernel

Building the kernel takes a very long time, since it is running on the embedded platform that has relatively limited processing power. After the build and a reboot, running the following command verifies that the new kernel is being used:

```
# The stock kernel returns '4.9.140-tegra'
# The custom kernel should return only '4.9.140'
uname -r
```

4.2.5. Booting from the custom kernel

If the command above confirms that the custom kernel is working, the root partition can be transferred to the external SSD by:

- Plugging the SSD in
- Wiping the partition table
- Creating a GPT partition table
- Creating a new EXT4 volume 4 GB smaller than the SSD
- Creating a 4 GB swap partition

```
# Wiping the partition table
sudo dd if=/dev/zero of=/dev/sda bs=1M count=1

# Creating a GPT partition table, then creating a new EXT4 volume
# Creating a Linux swap partition (4GB) - arrow over to "Type" and select
↪  "Linux swap"
# Going over to "Write", typing "yes" and then quitting
sudo cfdisk /dev/sda

# Making an ext4 volume and a swap partition
sudo mkfs.ext4 /dev/sda1
sudo mkswap /dev/sda2

# Mounting the partition and copying the root filesystem to it
sudo mkdir /mnt/root
sudo mount /dev/sda1 /mnt/root
sudo mkdir /mnt/root/proc
sudo apt -y install rsync
sudo rsync -axHAWX --numeric-ids --info=progress2 --exclude=/proc /
↪  /mnt/root
```

The OS can be booted from the external SSD by:

- Editing /boot/extlinux/extlinux.conf so the kernel points at /dev/sda1 (the SSD) instead of /dev/mmcblk0p1 (the microSD)
- Enabling zswap in extlinux.conf

```
sudo sed -i 's/mmcblk0p1/sda1/' /boot/extlinux/extlinux.conf
sudo sed -i 's/rootwait/rootwait zswap.enabled=1/'
↪  /boot/extlinux/extlinux.conf
```

4.3. DonkeyCar installation

According to the official Donkey documentation: "*Donkey is a high level self driving library written in Python. It was developed with a focus on enabling fast experimentation and easy contribution.*" [22]

For this thesis, Donkey will be used as an interface between the RC car and the neural net that should drive it. As shown on the figure above,the camera should send data from the

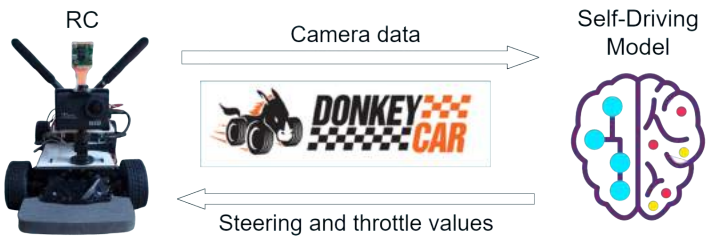

Figure 45: Donkey platform as an interface
(Source: DonkeyCar)

RC to a model which would analyse it and tell the RC where to steer and how fast to go, in order to stay on the road. Donkey provides an interface to do just so, without having to worry about the details of:

- (Pre)Processing the camera data
- Converting the steering/throttle values to actual signals for the RC to understand
- Actually steering the RC and increasing/decreasing throttle
- Collecting and labelling training data
- Defining and training custom models
- Controlling the car using a Web interface or a controller
- Using a simulator to rapidly test and train models

4.3.1. Installing and configuring Donkey

There are two ways Donkey can be used to create a self-driving RC car:

- Doing everything on the RC car/Jetson Nano, including model training
- Training and developing the model on a host PC, running and testing it on the RC

The second options is much more feasible, since a PC usually has much greater processing power to train and develop complex models, preferably by using a GPU. But it is also possible to exclusively use the Jetson Nano without a dedicated PC for model training and testing. Detailed instructions for installing the Donkey platform on both the RC car and the host PC, as well as the simulator, can be found on the official Donkey documentation website. [22] To create and use the Donkey platform, the official documentation states that a new Donkey application should be created using the following command:

```
donkey createcar --path <dir> [--overwrite] [--template <donkey2>]
```

After creating a new Donkey application, the most important file that should be edited is the ***myconfig.py*** file where the camera type, the I2C bus used on the Nano and the RC calibration values that will be explained in the following subchapter should be edited in.

4.4. Car calibration

To control the RC car using the Jetson Nano, the Linux user on the Nano should be added to the i2c group:

```
# Adding the user to the i2c group
sudo usermod -aG i2c %USERNAME%
# Rebooting so it takes effect
sudo reboot
```

To verify that the PCA9685 board is properly connected to the Nano, the following command should be run with the number of the I2C bus the PCA9685 board is connected to:

```
sudo i2cdetect -r -y %BUS_NUMBER%
```

If the PCA9685 is properly connected to the Nano, the command above should output:

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: 40 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: 70 -- -- -- -- -- -- --
```

The numbers 40 and 70 in the command output above denote the I2C address of the PCA9685 board and they should be written in the ***myconfig.py*** file under the following settings:

```
PCA9685_I2C_ADDR = 0x40
PCA9685_I2C_BUSNUM = 1
```

Now the Donkey application can be used to detect the PWM values for steering and throttle by using the following command along with the channel number used for throttle or steering on the PCA9685:

```
donkey calibrate --channel %THROTTLE_OR_STEERING_CHANNEL%
↪  --bus=%I2C_BUS_NUMBER%
```

To calibrate both steering and throttle, the number 370 should be used as a starting PWM setting and should be increased or decreased in increments of 10 until the maximum and minimum values are found for throttle and steering. After finding out the values, they should be written down into the ***myconfig.py*** script under the following settings:

```
STEERING_RIGHT_PWM = # The value which steers the car completely to the
↪  right.
STEERING_LEFT_PWM = # The value which steers the car completely to the
↪  right.
THROTTLE_STOPPED_PWM = # The neutral throttle value.
THROTTLE_FORWARD_PWM = # The maximum forward throttle value.
THROTTLE_REVERSE_PWM = # The maximum reverse throttle value.
```

4.5. Connecting a controller

To control the RC car, any controller can be used that can be mounted at the ***/dev/input/js0*** file on the Nano, either via Bluetooth or by using a cable.

After connecting the controller, the Donkey ***createjs*** command can be used to create a custom mapping that is saved into a Python class and can be imported and used by the main Donkey ***manage.py*** class.

The ***myconfig.py*** file already has a number of predefined controller mappings for the most common controllers, which can be used by uncommenting the ***CONTROLLER_TYPE*** setting and writing the appropriate mapping name.

4.6. Testing the Donkey pipeline

After installing the Donkey platform and calibrating the RC car and setting up a controller, the Donkey pipeline can be tested by building a test track, collecting a small amount of testing data, training a built-in Donkey model and testing the RC by letting it autonomously drive around the same test track.

The test track can be a simple circle with the lane lines made using some tape, such as:

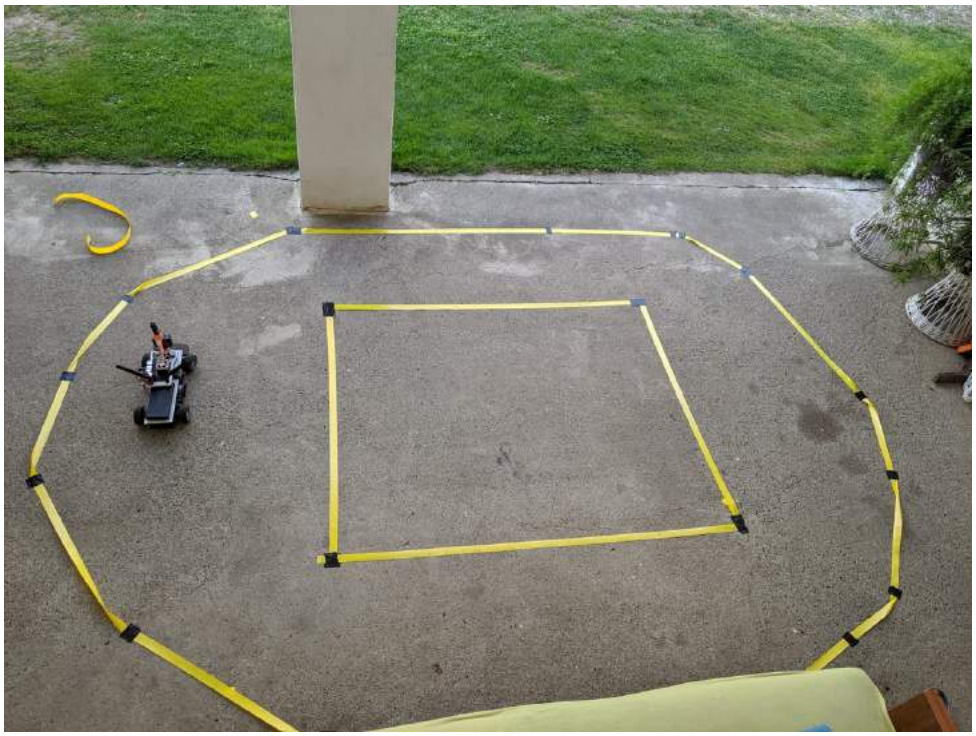

Figure 46: First test track

Or it can be a more complex one, with a very contrasted background as the asphalt and well defined and up to scale lanes and lane lines, such as:

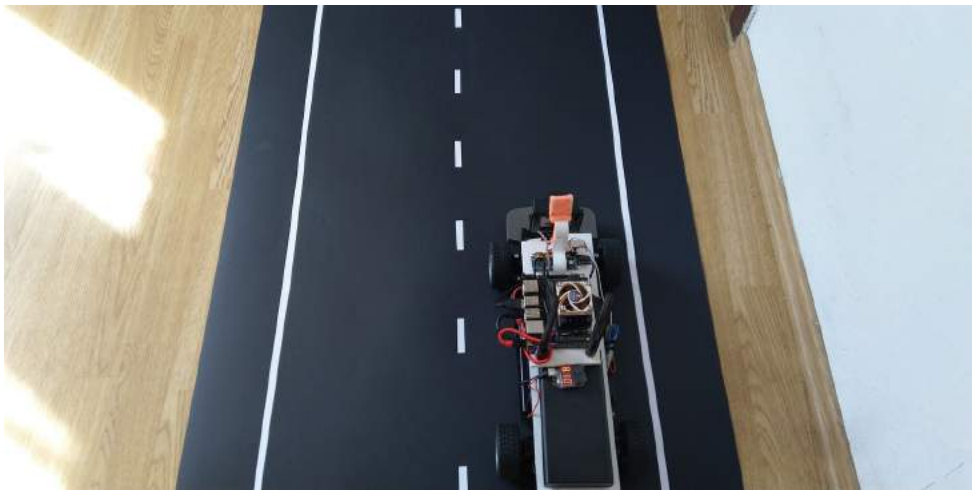

Figure 47: Fancier test track

After creating the test track and placing the RC car on it, training data can be collected by running the ***manage.py*** script with the ***drive*** flag, and the ***- -js*** flag if a controller is used:

```
python manage.py drive --js
```

After the above command executes, the RC car will automatically start recording labelled training data when it starts moving around the track. A good training dataset should have at least 10 recorded laps of the test track without any major errors being made. After recording enough data, the data should be copied from the ***data/tub_**_**-**-**** folder to the host PC, and trained using the ***manage.py*** script along with the ***train*** flag and a predefined Donkey model, a list of whom can be found inside the official Donkey documentation [22]:

```
python ~/mycar/manage.py train --tub /path/to/the/dataset/tub --model
↪  ~/mycar/models/theNameOfTheModel.h5 --type=architectureName
```

When the training completes, the trained model file should be copied back to the RC car and can be used to autonomously drive the car using the following command:

```
python manage.py drive --model ~/mycar/models/yourModel.h5
```

The RC car should be able to successfully autonomously drive at least one lap around the track, which confirms that the Donkey pipeline is correctly set up.

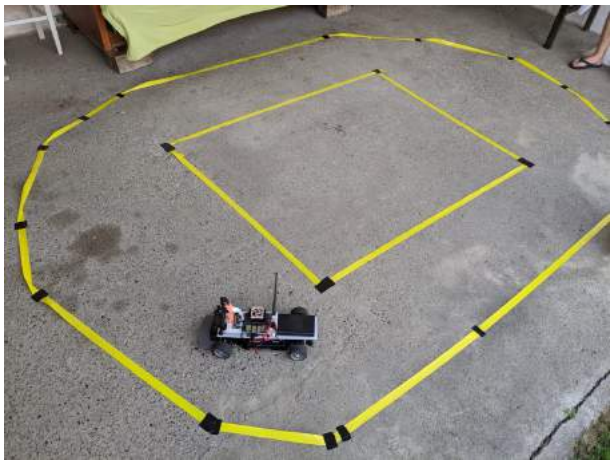

Figure 48: RC driving around the first test track

If the Donkey pipeline is correctly set up and the RC car can autonomously drive around the test track, everything needed to build and train custom ML models for the RC is finished.

5. Modding the simulator

In order to use the simulator to gather training data that's larger than the default 160x120 image size, a modified version of it has to be created. First off, the original simulator made by Tawn Kramer made for Donkey [39] should be cloned from GitHub.

```
git clone --single-branch --branch donkey https://github.com/tawnkramer/sdsandbox
```

Unity also needs to be installed, after which the project can be opened in it.

5.1. Modifying the camera resolution

After opening the project in Unity, the default RC prefab (donkey), can be found in the ***Prefabs*** folder.

Inside the prefab hierarchy the ***cameraSensorBase*** which contains the ***CameraSensor***, which is the RC camera sensor that is used to generate the training data (take pictures from inside the simulator).

After selecting the ***CameraSensor***, its visible that there is a ***CameraSensor*** script connected to it. It can be opened up by double clicking on it or finding it in the lower left project viewer inside ***AssetsScriptsCameraSensor.cs***.

The width and height fields of the class should be made static, so they can be edited them from other scripts:

```
// Default resolution
public static int width = 640;
public static int height = 480;
```

The parameters of the ***ReadPixels*** function should be edited to use the defined width and height:

```
tex.ReadPixels(new Rect(0, 0, width, height), 0, 0);
```

The ***CameraHelper*** script also needs to be edited to use the above defined static fields for the width and height:

```
Texture2D texture2D = new Texture2D(CameraSensor.width,
↪  CameraSensor.height, TextureFormat.RGB24, false);
texture2D.ReadPixels(new Rect(0, 0, CameraSensor.width,
↪  CameraSensor.height), 0, 0);
```

Now the menu scene can be opened and a Dropdown element added to it menu by right clicking on the Canvas and selecting UI > Dropdown. This dropdown will serve as a resolution picker and inside the inspector panel all of the resolutions that want to be used can be added as options:

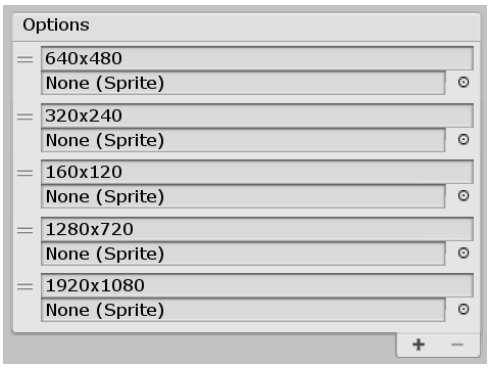

Figure 49: Resolution dropdown

The dropdown should be resized and placed on an appropriate position on the menu:

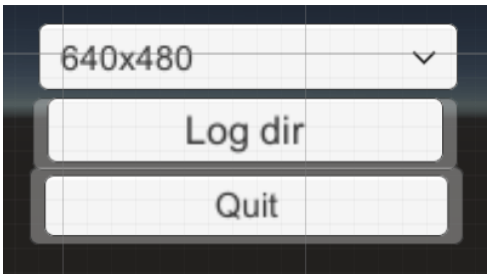

Figure 50: Resized resolution dropdown

The final step is to go back to the Scripts folder and add a new class called ***Resolution-Setter***. This class will set the resolution inside the CameraSensor class based on the selected option in the dropdown. The source code for this class can be seen in the Appendix (2.1).

After the ***ResolutionSetter*** class is created, it the dropdown element created earlier can be set to use it by selecting the dropdown element and dragging the newly created class on it:

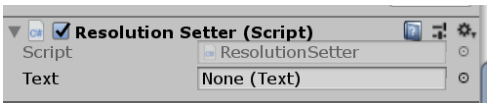

Figure 51: Resolution setting script

And that's it. Now the output images will have the resolution that is set in the main menu using the dropdown.

5.2. Updating the default donkey-gym port

One thing that has to be edited in ***donkey*** itself before the modded simulator can be used:

- The ***dgym.py*** file found inside ***donkeycar/parts/*** should be open
- The port number should be changed from 9090 to 9091:

```
def __init__(self, sim_path, port=9091, headless=0,
↪  env_name="whatever-env-name-here", sync="asynchronous"):
```

Now the simulator can be used with the ***manage.py*** script.

If the simulator is compiled to a different directory than the default one, or the projectexecutable name was changed, be sure to update the ***myconfig.py*** file, specifically the ***DONKEY_SIM_PATH*** variable. Here's a sample output image when the dropdown is set to 640x480:

Figure 52: 640x480 image

5.3. Creating 3D version of the RC car model

To get better data for a specific build of an RC model, the default model that comes with the simulator should also be modified so it accurately represents the RC model being trained. Most notably, the camera position, resolution and field of view should be edited. Multiple cameras can also be added and the center of mass of the vehicle can be edited, as well as any other properties that would make a difference to more realistically represent the RC model.

5.3.1. Editing the camera

To edit the camera of the default model, in the ***donkey*** prefab, the ***cameraSensorBase*** and the ***CameraSensor*** can be modified.

On the right, in the ***Inspector*** panel, under the ***Camera*** settings, a bunch of settings related to them can be edited. The FOV value should match the FOV the real RC camera has, and the ***Fisheye*** and ***Post Processing*** scripts can be disabled.

The camera can also be moved to different positions using the move and rotate tools and the editor on the center of the screen, and pivot/angle can be used to match it to the real RC camera position and orientation.

5.3.2. Testing the modified model

Any scene can be started, such as the warehouse scene, zoomed up to the track and the prefab can be dragged and dropped onto it. After dropping it on the track, the camera sensor can be selected to see what its output now looks like:

Figure 53: Camera preview

5.3.3. Adding additional parts and changing the center of mass

A custom 3D model of the electronics that are on the RC can also be created and inserted into the default Donkey model, or an entire new model can be created.

After creating a 3D model, it can be dragged and dropped into the Models folder. The added elements can then be added to the default prefab and positioned and resized to fit them on the RC.

If any elements need reordering or individual editing, but the element looks blue, like the ***power bank v1*** element shown in the figure below, they can be right clicked unpack prefab can be selected:

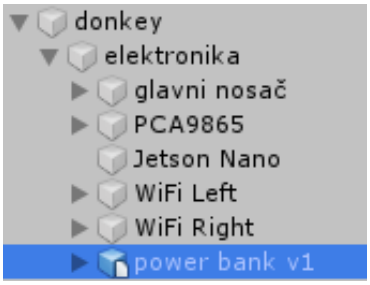

Figure 54: Packed prefab

It should then be possible to be individually move parts around.

If significant changes to the model were made, a lot of parts were added, the RC's center of mass has probably changed, and that should be reflected in the simulator by changing the defined center of mass, since it's desirable to as realistically handle corners just as it would in real life, and if the center of mass is higher up because a bunch of stuff has been added, it could perform great in the simulator but tip over in real life.

The ***centerOfMass*** element can be found inside the donkey prefab, moved and pivoted around just like any other element:

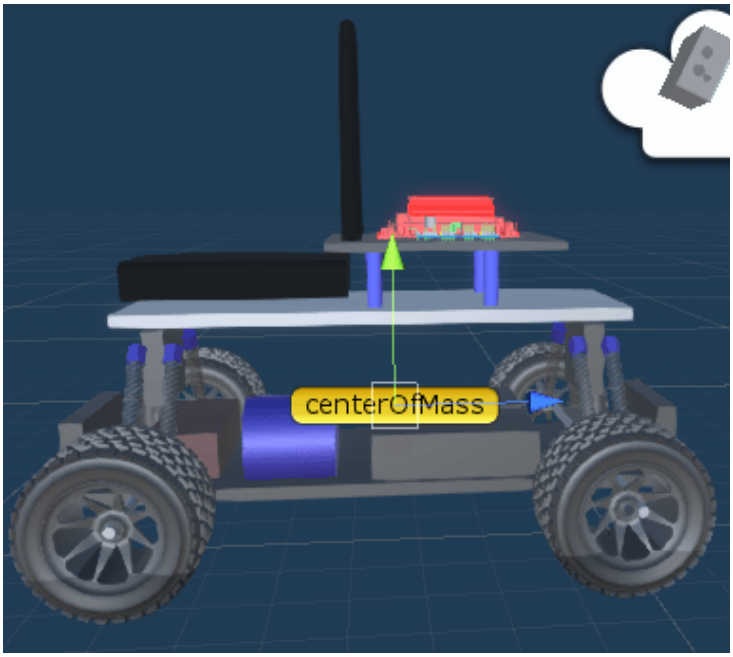

Figure 55: Center of mass

The simulator can now be built and the custom 3D RC car model can be tested.

5.4. Simulator camera calibration

The simulator camera should be calibrated just like the real RC car camera, and to do so, a checker-board object needs to be added into the simulator so photos of it can be taken with the simulator RC camera. This can be achieved by adding a ***GameObject*** object to any scene in the simulator and applying the checker-board pattern that was printed earlier as its texture.

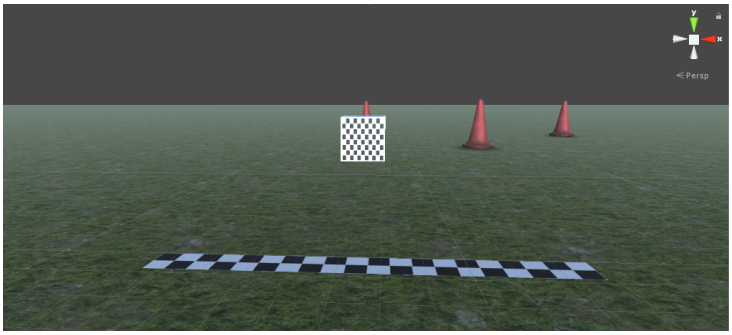

Figure 56: Simulator calibration

The cube proportions can be made to fit the original checker-board size, e.g. an A4 paper, by clicking on the cube and editing the scale values in the ***Inspector*** panel and setting them to, e.g. 0.297 for X and 0.21 for Y, since an A4 is 29.7 cm x 21.0 cm.

After that, the simulator can be run and screenshots can be taken from at least ten different angles, just like when calibrating a real camera. Here's what it could look like:

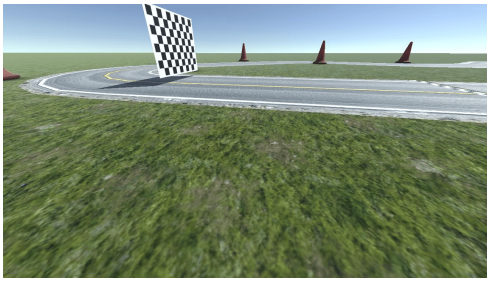

Figure 57: Simulator calibration image 1

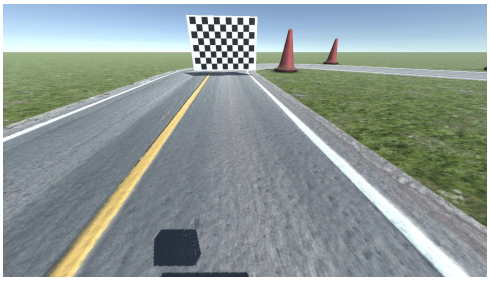

Figure 58: Simulator calibration image 2

6. Camera calibration

This chapter will explain some common problems that occur when cameras try to map the 3D world onto a 2D sensor plane. It will also describe methods that can be used to make lane line detection from input images easier.

6.1. Camera distortions

Since the RC car uses only cameras to obtain the entirety of its data, which will be used to drive the car around the real world, a lot of trust is placed in them. Which refers to trusting that they'll provide the car with accurate representations of real world 3D objects as 2D images that will get fed into its neural network.

Cameras shouldn't be inherently trusted to accurately represent the world they see, since they, albeit cheap and easy to use, come with all sorts of issues when it comes to mapping the 3D world onto a 2D sensor/image correctly. But luckily, there are ways of solving those issues, which will be described in the following sub chapters

6.1.1. The pinhole camera model

The pinhole camera model is a model of an ideal camera, that describes the mathematical relationship between the real world 3D object's coordinates and its 2D projection on the image plane. [40]

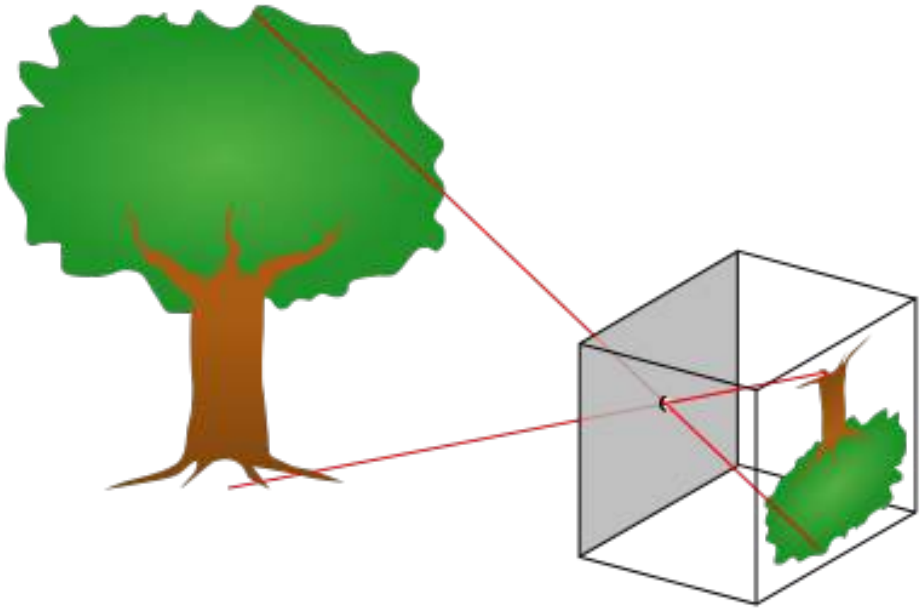

Figure 59: Pinhole camera model
(Source: Wikiwand)

Pinhole cameras were the very beginning of photography, and are used even today to explain basic photography to students. They posses some advantages over regular lens cameras [40]:

- They have near infinite depth of field; everything appears in focus.
- No lenses are used to focus light, so they have no lens distortion and wide-images remain absolutely rectilinear.

Basically, the smaller the pinhole gets, the more the resolution increases, until it reaches the diffraction limit, at which point the image just gets darker and blurrier. Also, the smaller the pinhole, less light comes in, so the exposure time needs to be increased. [40] Which brings up the big issue with them; Their exposure times are really long, which causes significant motion blur around any moving objects or it causes their complete absence if they've moved too fast. [40] Is it possible to get a small pinhole that gets a lot of light? A convex lens can be used, for starters.

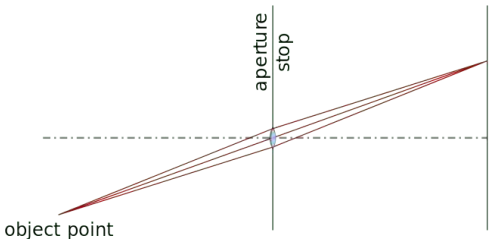

Figure 60: Convex lens
(Source: HowThingsWork)

Why does this help: instead of a single ray of light illuminating a single image point, now pencils of light illuminate each image point. [41]

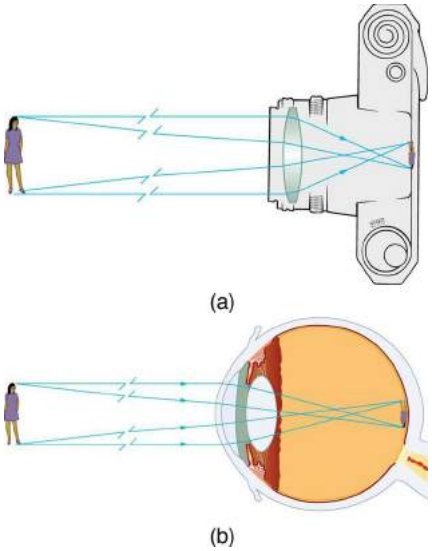

Figure 61: Camera lens
(Source: OpenStax)

But of course, lenses bring the issues mentioned earlier [40]:

- They have finite aperture so blurring of unfocused objects appears.
- They contain geometric distortions due to lenses, which increase closer to the edges of the lenses.

6.1.2. Types of distortions

The first, and most common type of camera lens distortion is called radial distortion.[43]

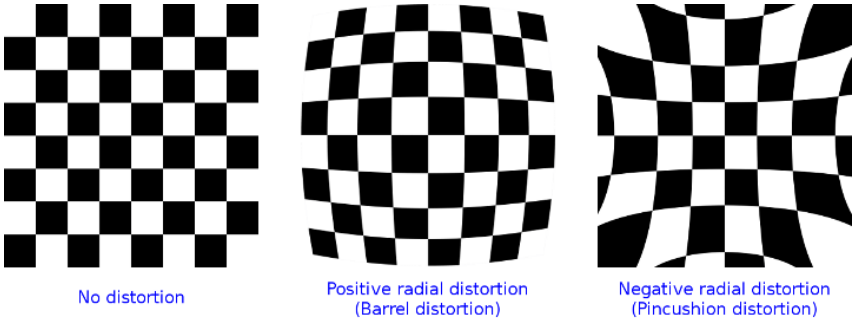

Figure 62: Types of distortions
(Source: Bradski)

There are two types of this distortion, the positive or barrel radial distortion, and the negative or pincushion radial distortion. [43]

In the context of self driving RCs, barrel distortion is the most often distortion, that will most probably be caused by fish eye lenses, since it's desirable to get as big a field of view as possible. Some action cams even have a FOV of 170 to 180 degrees, which causes a lot of positive radial distortion.

The other type of distortion possible is called tangential distortion, which occurs when the lens is not aligned perfectly parallel to the imaging plane (the sensor).[14] It causes the image to look tilted, which is obviously bad since some objects look further away than they really are.

6.1.3. Removing distortions using OpenCV

The radial and tangential distortions can be described using a couple of coefficients [14]:

- k_n coefficients will describe radial distortion
- p_n coefficients will describe tangential distortion

The worse the distortion, the more coefficients are needed to accurately describe it. OpenCV works with up to six (k_1, k_2, k_3, k_4, k_5 and k_6) radial distortion coefficients, which

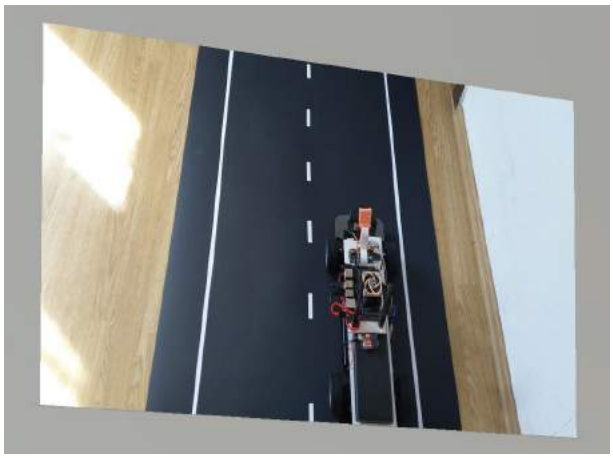

Figure 63: Tangential distortion

should be more than enough for the purposes of this thesis, and with two (p_1, p_2) tangential distortion coefficients.

If the barrel radial distortion type is present, k_1 will typically be larger than zero. If the pincushion distortion is present, k_1 will typically be smaller than zero. [14]

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{6.1}$$

- (X, Y, Z) are the coordinates of a 3D point that's being imaged (u, v) are the 2D coordinates of the projection point in pixels
- The first matrix after the equation is the camera matrix, containing intrinsic camera parameters
  - (c_x, c_y) defines the principle point which is usually the center of the image
  - f_x and f_y are the focal lengths expressed in pixel units
- The matrix containing the r_{mn} parameters is the joint rotation-translation matrix, a matrix of extrinsic parameters which describes camera motion around a static scene. It's used to translate the 3D coordinates to a 2D coordinate system, fixed with respect to the camera.

Since the RC car is imaging 2D images, to undistort them it's necessary to map the 3D coordinates to a coordinate system:

$$
\begin{aligned}
\begin{bmatrix} x \\ y \\ z \end{bmatrix} &= R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \\
x' &= x/z \\
y' &= y/z \\
u &= f_x * x' + c_x \\
v &= f_y * y' + c_y
\end{aligned}
\tag{6.2}
$$

Also, since the RC car is not using a pinhole camera, the distortion coefficients need to be added to the model:

$$
\begin{aligned}
x'' &= x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2 p_1 x' y' + p_2 \\
y'' &= y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1 (r^2 + 2 y'^2) + 2 p_2 x' y' \\
&\text{where} \quad r^2 = x'^2 + y'^2 \\
u &= f_x * x'' + c_x \\
v &= f_y * y'' + c_y
\end{aligned}
\tag{6.3}
$$

Since the primary interest is in efficiently removing the radial distortion, the formulae above are using Fitzgibbon's division model [44] as opposed to Brown-Conrady's even-order polynomial model [45], since it requires fewer terms in cases of severe distortion. It is also a bit easier to work with, since inverting the single parameter division model requires solving a one degree less polynomial than inverting the single-parameter polynomial model. [46]

6.1.4. Finding the camera's intrinsic and extrinsic parameters

After laying out all of the formulas used to correct radial and tangential distortion, the only question that remains is how to get the intrinsic and extrinsic camera parameters. For those purposes, the OpenCV ***calibrateCamera*** function will be used, along with its ***findChessboardCorners*** function. [14]

The calibrateCamera function is based on Zhang's ***A Flexible New Technique for Camera Calibration*** [47] and Caltech's Camera Calibration. [48] It needs the coordinates of a 3D object that's being imaged and its corresponding 2D projected coordinates in order to detect the intrinsic and extrinsic parameters of the camera used to image the object. To easily get those coordinates, using a chessboard is recommended. A chessboard is an object with a known geometry and it has easily detectable feature points. Such objects are called calibration rigs or patterns, and OpenCV has a built-in function that uses a chessboard as a calibration rig, the ***findChessboardCorners*** function.

The ***findChessboardCorners*** function attempts to determine whether the input image is a view of the chessboard pattern and automatically locate the internal chessboard corners. The great thing with this is that a 3D object (a chessboard) can be printed out, whose geometry is well known, and its 3D coordinates can be mapped to the 2D image projection. The 3D points of the chessboard from the real world are called object points and their 2D mappings on the image are called image points. So, after printing out a chessboard and taking multiple pictures of it from different angles to better capture the camera distortions, they can be fed to the ***findChessboardCorners*** function, which returns the detected object points and corresponding image points, which can be used to calibrate the camera.

The ***calibrateCamera*** function, given the object points and image points by the ***findChessboardCorners*** function, performs the following [14]:

- Computes the initial intrinsic parameters. The distortion coefficients are all set to zeros initially.
- Estimates the initial camera pose as if the intrinsic parameters have been already known.
- Runs the global Levenberg-Marquardt optimisation algorithm [49] to minimise the reprojection error, that is, the total sum of squared distances between the observed feature points ***imagePoints*** and the projected (using the current estimates for camera parameters and the poses) object points ***objectPoints***.
- The function returns a matrix with the intrinsic camera parameters and a matrix with the distortion coefficients, which can be used to undistort input images.

6.2. Camera calibration implementation

This sub chapter will use the implementation of the above mentioned camera calibration techniques (source code: Appendix 1.1, 1.2, 1.3), and demonstrate their results when applied to the action camera used in the RC car hardware platform, which has an advertised 170 degree FOV:

Figure 64: EleCam Explorer action camera

First, a checker board pattern needs to be created and printed to be used as a calibration rig. The following example will be used for the undistortion demonstration for the above mentioned camera:

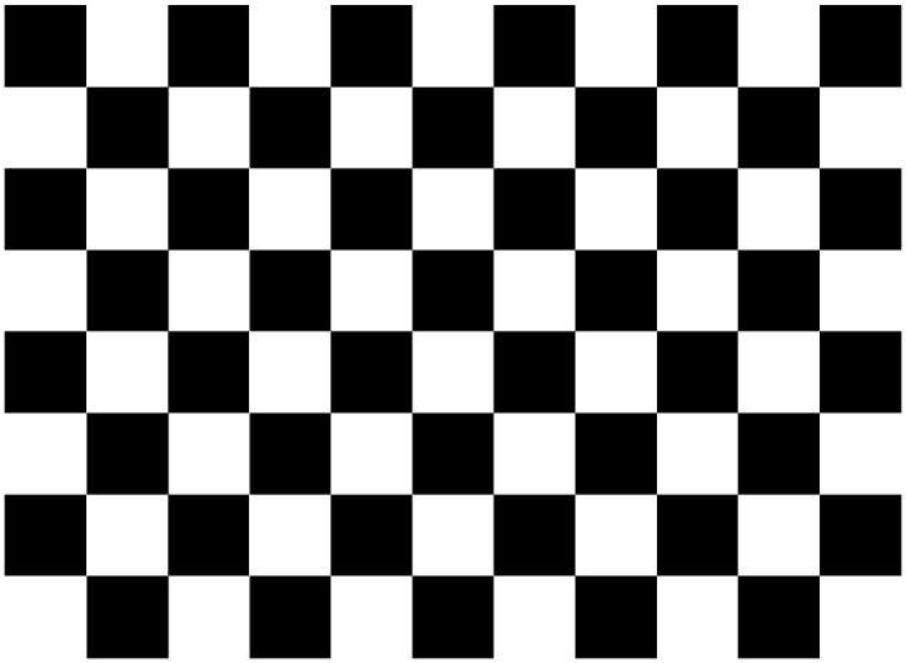

Figure 65: Checkerboard calibration rig

The calibration rig should be printed and photographed from at least ten different angles, to allow for precise calibration:

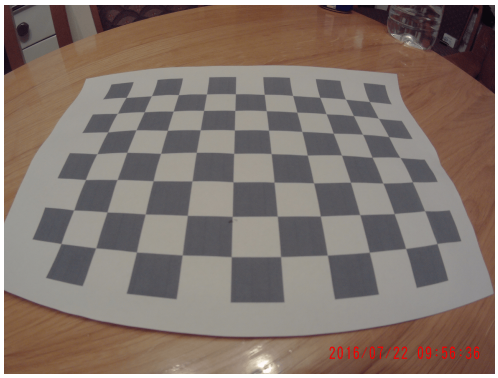

Figure 66: Calibration photo 1

Figure 67: Calibration photo 2

After processing the calibration images using the implemented calibration function (source code: Appendix 1.1), the function finds the following image points and maps them to the predefined object points:

Figure 68: Found imagepoints 1

Figure 69: Found imagepoints 2

The camera calibration functions can be used right before feeding an input image to the neural network using the following code:

```
# At the beginning of run
getObjectAndImagePoints()
calibrateCamera(inputImage.shape[1::-1])
# For every input image to the NN
undistortedImage = undistortImage(inputImage)
# Pass it along to the NN
```

Using the image points found using the calibration images the calibration function (source code: Appendix 1.2) can be called to obtain the undistortion matrix, which can then be used to call the undistort function (source code: Appendix 1.3) and get the following results:

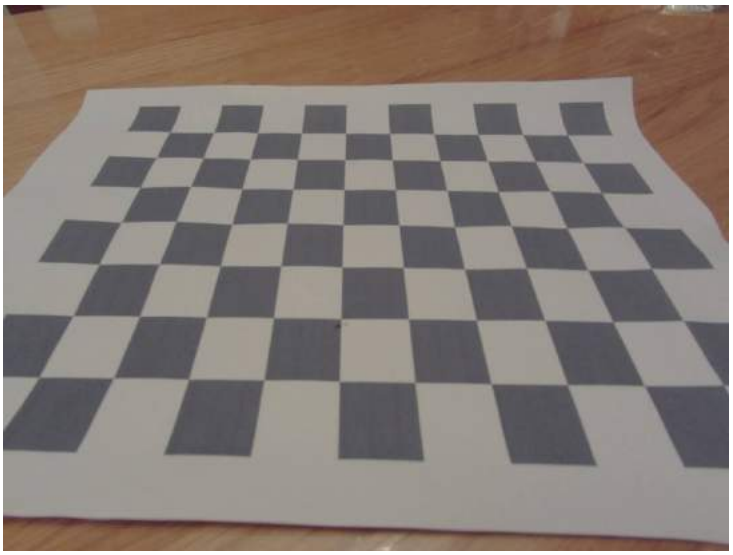

Figure 70: Undistorted image 1

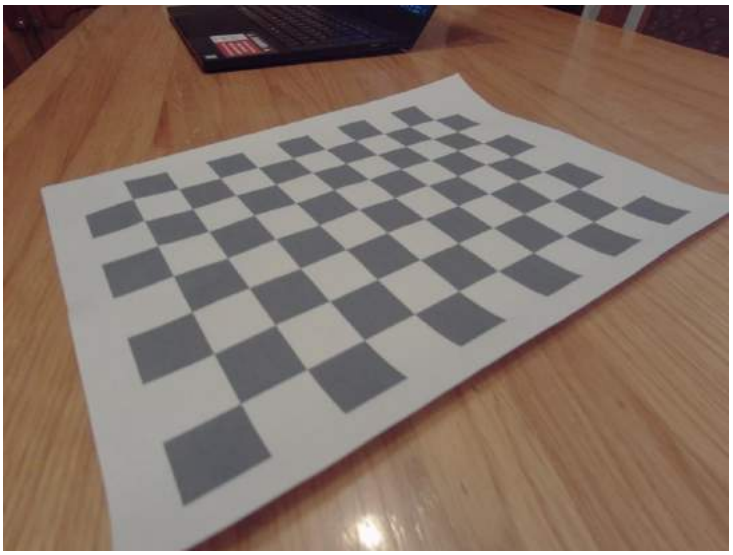

Figure 71: Undistorted image 2

7. Lane line extraction

This chapter will explain how to extract lane lines from raw images by performing a perspective transform on every input image to get a birds-eye view of the road and converting the image to HLS color space, extracting only the S channel and performing some thresholding on it to extract only the lanes.

7.1. Mapping the 3D world on a 2D sensor plane

An inherent problem that comes with mapping a 3D world onto a 2D sensor plane, which was also explained in the previous chapter, is that the 2D representation can't perfectly represent the real world, and shouldn't be taken for granted. For humans, this is easy to understand, but for a neural network that has only ever seen 2D images as inputs, it couldn't be faulted for believing in those incorrect representations as true. This is very visible when taking an image of a road for an example:

Figure 72: An image of a highway

To the human reader, the image is a pretty normal image of a highway. No errors can be obviously seen. But stepping into the shoes of a neural network that looks at it without any previous knowledge of what the world, nontheless a road is, it could be fairly assumed that the lane lines intersect somewhere in the distance. That assumption is argued by the illustration on the following page:

Figure 73: An image of a highway with visualized intersection of lane lines

Based on this image alone, it could be fairly hypothesised that the lane lines are not in fact parallel, and that they get closer and closer as shown above, until they intersect. This is what the RC car would see when looking at a road:

Figure 74: An image of simulated road

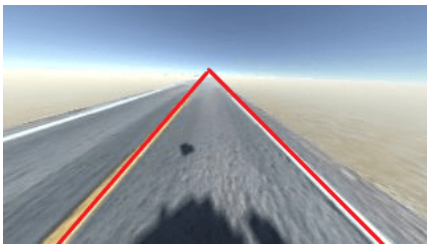

Figure 75: An image of simulated road with visualized intersection of lane lines

The only way the RC car, the neural network, could possibly know that the lane lines are parallel would be from real life experience and general knowledge about the world, or by having a bird's eye perspective in which the lane lines would obviously be parallel, as shown below:

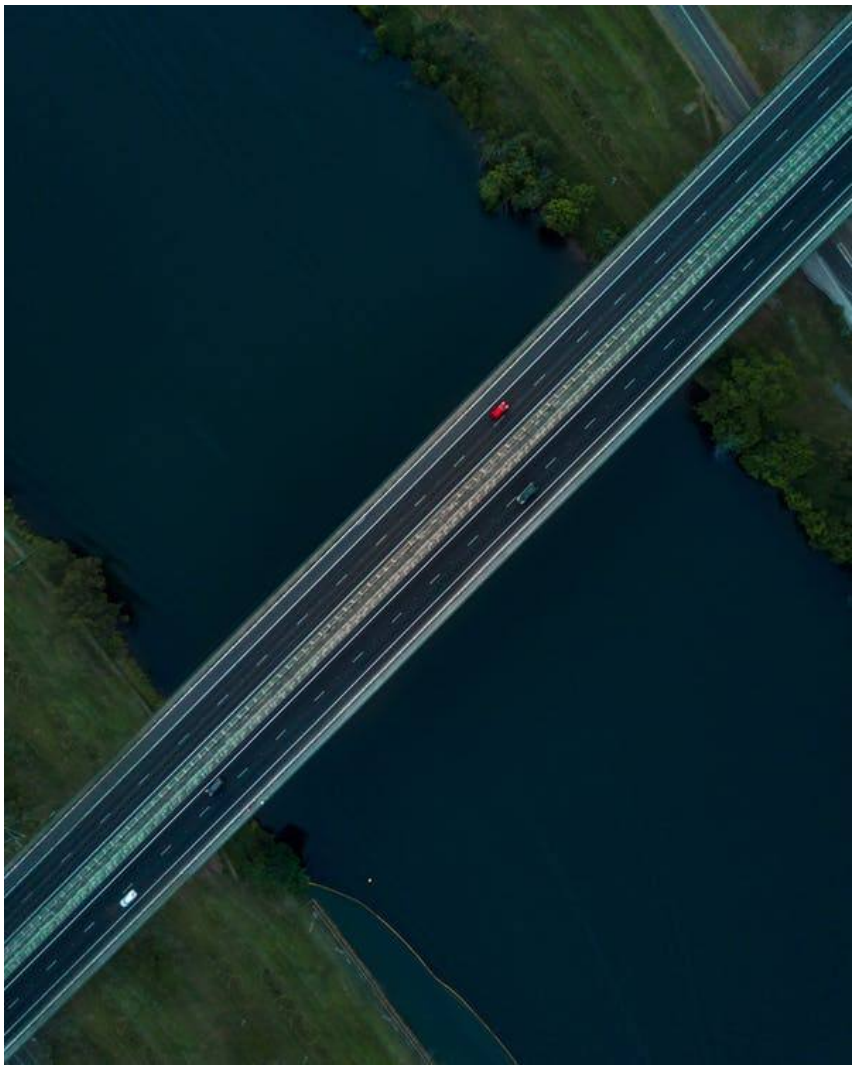

Figure 76: A bird's-eye view of a highway

Luckily, one of the options is much easier than the other, so a method for transforming the frog's perspective of the car into a bird's-eye view will be explained and implemented.

The first step in creating a perspective transform from a frog's perspective to the bird's-eye view is to define a region of interest (ROI) that will take into account only the part of the image that is of any interest to the RC car, and transform it into another perspective by mapping those ROI coordinates to another set of defined coordinates.

7.1.1. Implementing a perspective transform

The following simulated road image will be used as a starting point for the definition of a ROI that will be transformed to a bird's-eye perspective:

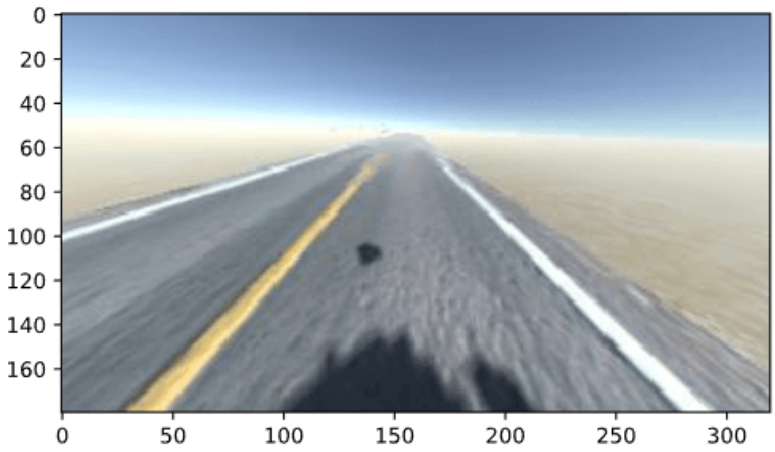

Figure 77: Starting perspective (320x180 pixels)

One region of interest could be defined using the following coordinates, illustrated on the figure below:

```
# Define the region of the image we're interested in transforming
regionOfInterest = np.float32(
                [[0,  180],  # Bottom left
                [112.5, 87.5], # Top left
                [200, 87.5], # Top right
                [307.5, 180]]) # Bottom right
```

Figure 78: Starting perspective

Using the above shown ROI as the starting point, a bird's eye perspective would mean that the green and blue lines are parallel, so the following destination coordinates can be defined:

```
# Define the destination coordinates for the perspective transform
newPerspective = np.float32(
            [[80,  180],  # Bottom left
            [80,    0.25],  # Top left
            [230,   0.25],  # Top right
            [230, 180]]) # Bottom right
```

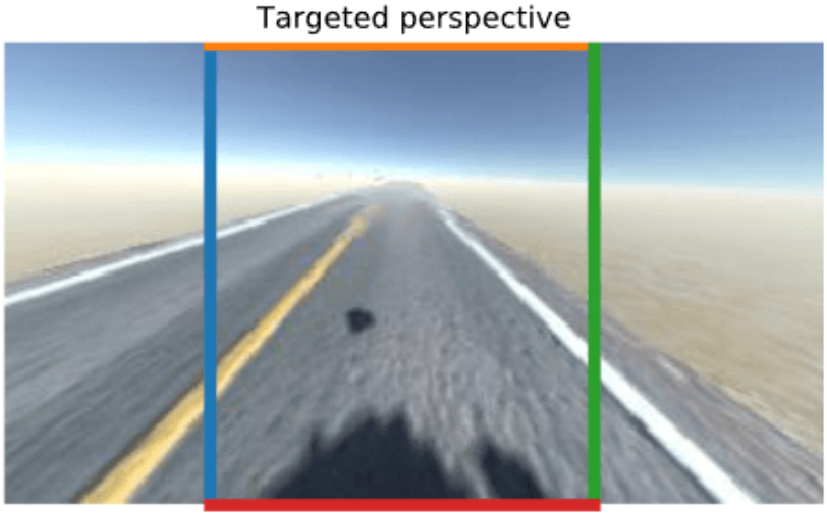

Figure 79: Starting perspective

Once the coordinates are, that's actually it. The rest of the implementation just calls a couple OpenCV functions, the first of which is ***cv2.getPerspectiveTransform***, which takes in the starting coordinates defined eariler (ROI) and the target coordinates and returns a matrix with which the original image can be warped to the target perspective:

```
# Compute the matrix that transforms the perspective
transformMatrix = cv2.getPerspectiveTransform(regionOfInterest, newPerspective)

# Compute the inverse matrix for reversing the perspective transform
inverseTransformMatrix = cv2.getPerspectiveTransform(newPerspective, regionOfInterest)
```

This transformation actually isn't that complicated, it's actually pretty simple math which generates a transformation matrix with which defines how to map some input to a defined output. Also, to unwarp the image back into its original perspective, an inverse transformation matrix can be obtained by calling the ***cv2.getPerspectiveTransform*** and simply switching the coordinate parameters (see code above).

Once a transformation matrix is calculated, input images can be warped to the destination perspective using ***cv2.warpPerspective***:

```
# Warp the perspective
# image.shape[:2] takes the height, width,
# [::-1] inverses it to width, height
warpedImage = cv2.warpPerspective(image, transformMatrix, image.shape[:2][::-1], flags=cv2.INTER_L
```

The end result is shown on the two figures below:

Figure 80: Starting perspective

Figure 81: Transformed perspective

7.2. Extracting lane lines from the warped images

One other thing that can ease the detection of lane lines from the above warped images is actually just converting the image into another colorspace and selecting just one color channel. OpenCV uses the blue, green, red (BGR) colorspace by default, which needs to be converted into the hue, saturation and lightness (HSL) colorspace [50] to easier detect lane lines:

```
# Convert the image to HLS colorspace
hlsImage = cv2.cvtColor(warpedImage, cv2.COLOR_BGR2HLS)
# Split the image into three variables by the channels
hChannel, lChannel, sChannel = cv2.split(hslImage)
```

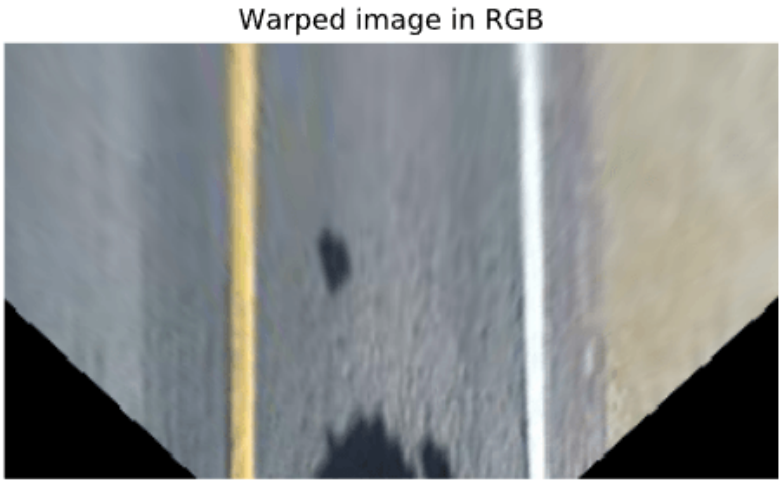

Figure 82: RGB warped image

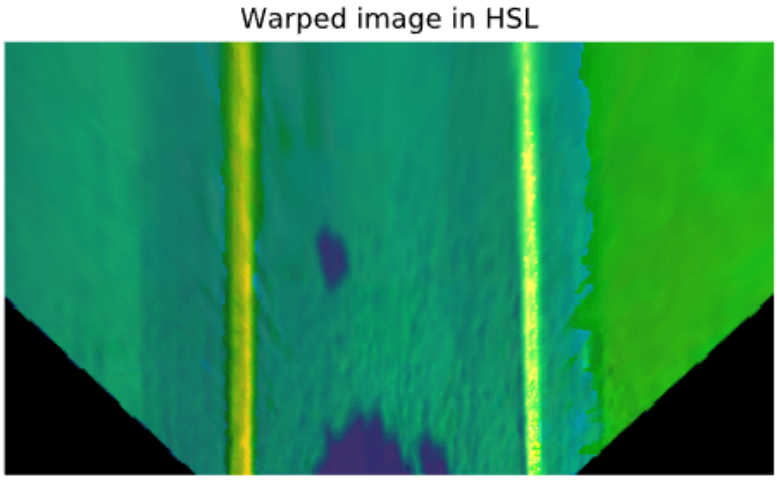

Figure 83: HSL warped image

After separating the image into three HSL channels, displaying each one individually, it's visible that the saturation channel is strongly correlated with the lane lines on the image, since the lane lines are always very saturated in color, be it regular white, orange or any hue inbetween, which makes this method robust to varying lane line colors in different lighting conditions or traffic situations:

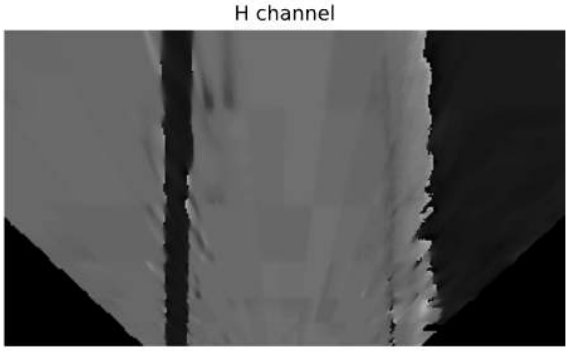

Figure 84: Hue channel

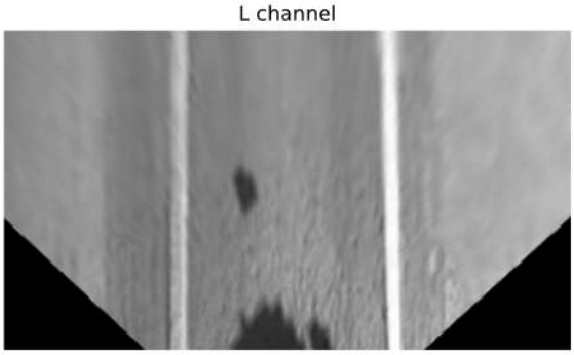

Figure 85: Lightness channel

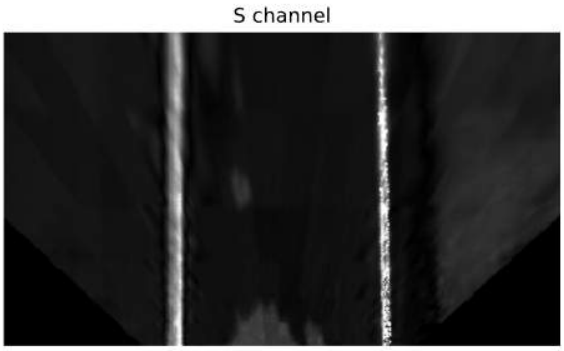

Figure 86: Saturation channel

There's one more thing that could make the lane lines pop out, as opposed to the lightly activated pixels on the asphalt, the sand on the right and the RC shadow in the middle of the lane: a simple thresholding of the values of the saturation channel image, which means keeping only the pixels that are above a certain level of saturation.

```
# Threshold the S channel image to select only the lines
lowerThreshold = 65
higherThreshold = 255

# Threshold the image, keeping only the pixels between the lower and higher
↪  threshold
returnValue, binaryThresholdedImage =
↪  cv2.threshold(sChannel,lowerThreshold,higherThreshold,cv2.THRESH_BINARY)

# Since this is a binary image, convert it to a 3-channel image so OpenCV can use it
# Doesn't really matter if RGB/BGR/anything else is used
thresholdedImage = cv2.cvtColor(binaryThresholdedImage, cv2.COLOR_GRAY2RGB)
```

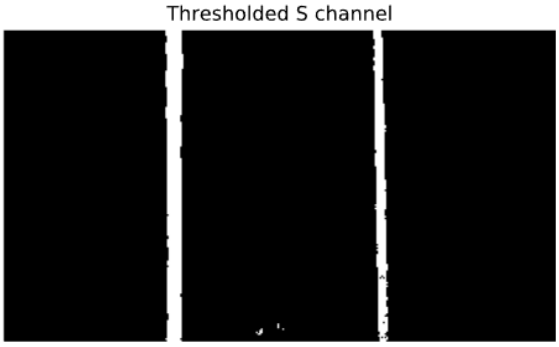

Figure 87: Thresholded S channel

Further on in the thesis, during the advanced custom model implementation, the thresholding of images as shown above will be used for image processing and input to the neural network. Another method that was tried during the writing of this thesis was taking a histogram of the above shown figure, which can easily be done by just summing up the values of the pixels by the X-axis (column), since it is a binary image whose pixels have values of either 0 and 1, and then using the histogram as a starting point for searching lane lines in the thresholded image and fitting a polynomial to the two separate lines, as shown below:

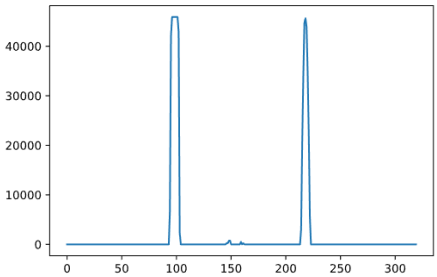

Figure 88: Histogram of the thresholded S channel

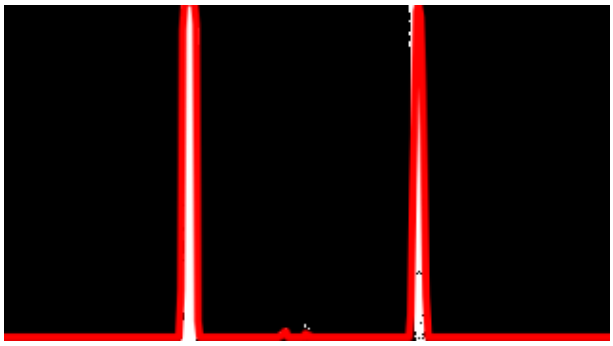

Figure 89: Overlaid histogram over the thresholded S channel

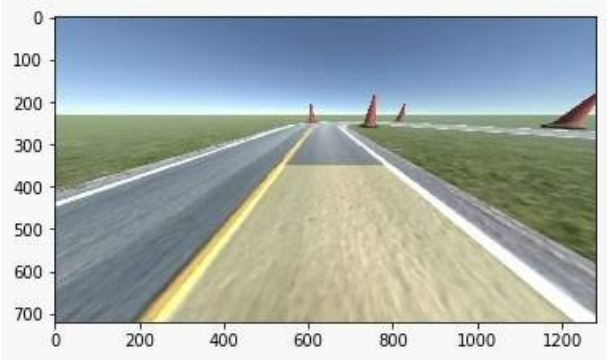

Figure 90: Visualized fitted polynomial on an unwarped image

The two main reasons why the method above wasn't used are:

- It's computationally expensive to do all of the processing and fitting for every single frame
- It can go haywire if the car moves the slightest out of the predefined ROI, and even if it's quickly corrected, since it uses the previous frames as a start for the next one, to be more computationally efficient, it would take some time for it to correct itself.
- Not using the previous frame and doing all of the work for every single frame was also tried, and it takes a lot of computational power to do so.
- Also, it's not ideal that the model relies so much on feature engineering and extraction, it's preferable for it to be more robust, even if it means it will be harder to train or take longer to train. By simply passing the thresholded image to a small convolutional network, it should be able to get all the information it needs from it, and it should be way less computationally expensive in the long run, even if a couple of additional convolutional layers were added to the model. Even if the fitted polynomials were passed to the network, it'd still have to be trained to interpret them, and taking into consideration the results of the short tests done, the conclusion was made that it would be less robust and even more computationally expensive to do so.

8. Artificial intelligence

This chapter describes the process of modelling, implementing and training custom neural networks that control the RC car through their outputs. The chapter will briefly introduce some of the basic building blocks of neural networks used, such as convolutions and dense layers, as well as their activation functions and ways of regularizing them.

The process of implementing a custom network architecture using Keras and interfacing it with the RC car using DonkeyCar will be described, after which a test model will be created using one of the first self-driving RC car models created. Then, new proposed architecture for modelling self-driving behaviour will then be proposed, implemented and trained to demonstrate its feasibility.

8.1. Convolutions

In the context of neural networks, Goodfellow et al. [51] define the convolutional operation as:

$$convolution(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{8.1}$$

Where the I denotes a two-dimensional image input, the K defines the kernel and the i, j arguments denoting the input indices. The output of the convolutional operation is often reffered to as the feature map. [51] They also define convolutional networks as: "... *simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers*". [51] The use of convolutions was first proposed by Fukushima in his **Neocognitron** paper, but the modern approach as they're used today was proposed by LeCun, Bengio, *et al.* in their **Convolutional networks for images, speech, and time series** paper.

Convolutional layers will be used throughout this thesis in order to map features from two-dimensional input images.

8.2. Dense layers

Dense, or fully connected layers are layers between which every unit (neuron) from one layer is connected with every other neuron from the other layer. Dense layers take the output from the previous layer as their input, which is then multiplied by the weights that connect the two layers and passed through an activation function to calculate the output for each unit in the layer.

These layers will be used throughout the thesis to let the network learn the mapping of the output feature maps from the convolutional networks to the output values (steering and throttle) of the network.

8.3. Activation functions

Activation functions are mathematical functions used in neural network units whose input is a weighted sum of the unit input an added bias value, which bind the input value to a range defined by the selected mathematical function that tells the neural network should the neuron be activated or not.

A wide range of mathematical functions can be chosen as the activation function for a given unit, but their purpose is often to introduce a non-linearity to the unit output, so various non-linear mathematical functions are chosen.

8.3.1. Rectified Linear Unit

The rectified linear unit (ReLU) is one of the most widely used activation functions today. It was first introduced by Hahnloser, Sarpeshkar, Mahowald, *et al.* in 2000., later used in the context of object recognition by Jarrett, Kavukcuoglu, Ranzato, *et al.* in 2009. and finally was popularized by Nair and Hinton in the context of restricted boltzmann machines in 2010.

ReLU improved on the sigmoid activation function, that was widely used until the paper on Restricted Boltzmann Machines in 2010, by simplifying it while still keeping its non-linear form.

The ReLU function is defined as:

$$f(x) \begin{cases} x & \text{when x=>0} \\ 0 & \text{when x<0} \end{cases} \tag{8.2}$$

The figure below shows the plot of the ReLU activation function:

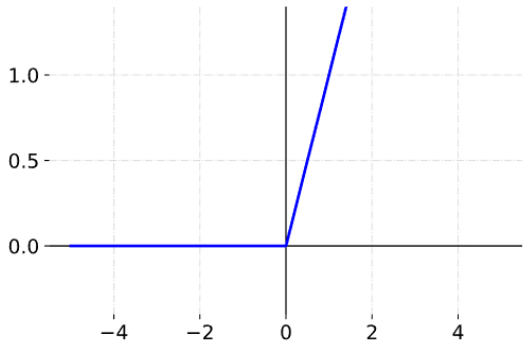

Figure 91: ReLU activation function

One significant downside when using the ReLU activation function is the possibility of **dying ReLU**, which occurs when a large negative value gets input to the activation function, which causes the unit in question to basically never activate again. To solve this problem, another activation function that builds on ReLU can be used.

8.3.2. Leaky ReLU

Leaky ReLU was introduced by Maas, Hannun, and Ng in 2013. as a method of solving the dying ReLU problem by introducing a parameter that defines a percentage of the value to keep if the activation input is negative.

The Leaky ReLU function is defined as:

$$f(x)\begin{cases} x & \text{when x=>0} \\ \\ \alpha \cdot x & \text{when x<0} \end{cases} \tag{8.3}$$

The α parameter can be set to any value, with the $\alpha = 0.3$ being the default setting in the Keras framework. The figure below shows the Leaky ReLU function plot with two possible α settings:

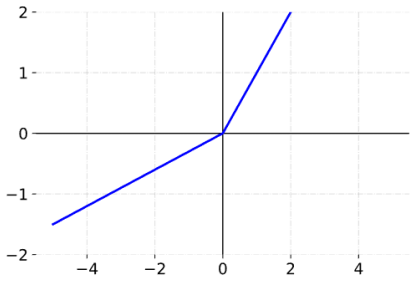

Figure 92: Leaky ReLU with $\alpha = 0.3$

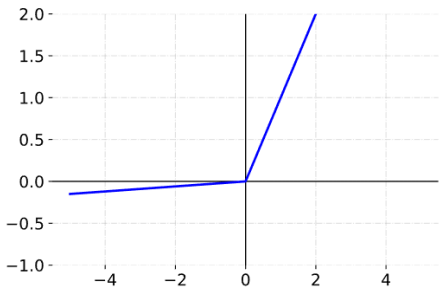

Figure 93: Leaky ReLU with $\alpha = 0.03$

The α parameter can also be treated as a parameter that the network should learn, which was proposed by He, Zhang, Ren, *et al.* in 2015 and called **Parametric ReLU** (PReLU). For the purposes of this thesis, through experimentation, it was determined that the Leaky ReLU activation function solved the dying ReLU problem appropriately, so PReLU will not be used as not to increase the number of parameters the network should learn, at the possible expense of improved performance.

8.4. Regularization layers

Goodfellow et al. state that: "A central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error." [51]

Two regularization layers that will be used throughout this thesis are dropout regularization and batch normalization.

8.4.1. Dropout regularization

To stop the model from overfitting the data, some neurons can be deliberately disabled while training the model. This in turn causes the model to be unable to rely on just a particular set of neurons, or a pathway of neurons, so it has to train multiple subsets of neural nets, which makes it much more difficult to overfit the data, since it never knows when some of the neurons will be knocked out. This method is called dropout regularization, which was introduced by Srivastava, Hinton, Krizhevsky, *et al.* in 2014.

8.4.2. Batch normalization

Another method of regularizing is batch normalization, which reduces internal covariate shift that its creators, Ioffe and Szegedy (2015.), define as: "*... the change in the distribution of network activations due to the change in network parameters during training.*".

Simplified, the output of the first layers feed into the subsequent layers, and so on which means that changing the parameters of a layer causes the distribution of inputs to subsequent layers to change, which batch normalization prevents.

8.5. Using a custom model with Donkey

All of the predefined Donkey models can be found in the ***keras.py*** script in the parts directory of the Donkey project. Donkey provides users with a base class called ***KerasPilot*** which provides an interface between Donkey and Keras by implementing several helper functions.

This class can be inherited by creating a new subclass, and a custom model can be implemented by specializing the model compilation and run functions. This subchapter will demonstrate an implementation, training and testing of a custom neural network architecture using Keras and DonkeyCar.

8.5.1. Creating a first custom model

Since the goal of this section is to only demonstrate how to create and use a custom model with Donkey, a model designed by Bojarski, Del Testa, Dworakowski, *et al.* from Nvidia will be used. The Nvidia team based their project on previous work made for the Defense Advanced Research Projects Agency (DARPA) seedling project known as DARPA Autonomous Vehicle (DAVE) [62], in which a sub-scale radio control (RC) car drove through a junk-filled alley way, which seems like a fitting way to start the custom architecture modeling for this thesis, by honouring the very beginning of autonomous RC cars. The architecture that will be implemented is shown below:

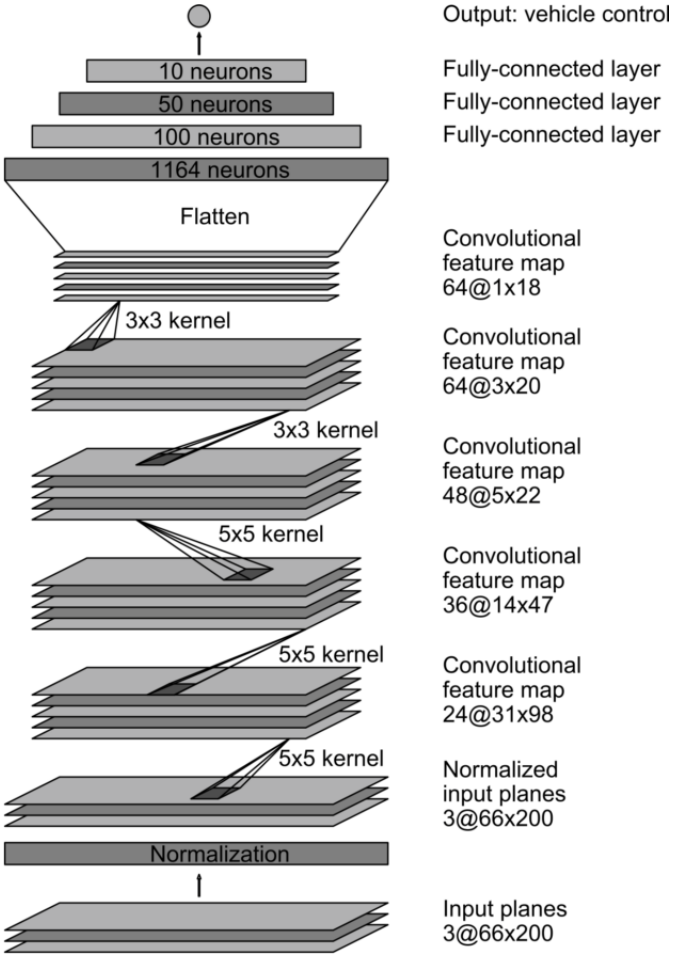

Figure 94: Nvidia model architecture
(Source: Bojarski, Del Testa, Dworakowski, *et al.*)

The architecture consists of 9 layers, including a normalization layer, 5 convolutional layers, and 3 fully connected layers. The authors explain that [61]:

- The first layer of the network performs image normalization. The normalizer is hard-coded and is not adjusted in the learning process. Performing normalization in the network allows the normalization scheme to be altered with the network architecture, and to be accelerated via GPU processing.

- The convolutional layers are designed to perform feature extraction, and are chosen empirically through a series of experiments that vary layer configurations. We then use strided convolutions in the first three convolutional layers with a 2×2 stride and a 5×5 kernel, and a non-strided convolution with a 3×3 kernel size in the final two convolutional layers.
- We follow the five convolutional layers with three fully connected layers, leading to a final output control value which is the inverse-turning-radius. The fully connected layers are designed to function as a controller for steering, but we noted that by training the system end-to-end, it is not possible to make a clean break between which parts of the network function primarily as feature extractor, and which serve as controller.

The first step in a custom model implementation compatible with DonkeyCar is to create a new class named ***nvidia.py*** which inherits the base ***KerasPilot*** class and implements all necessary methods:

```
from donkeycar.parts.keras import KerasPilot

class NvidiaModel(KerasPilot):
    def __init__(self, num_outputs=2, input_shape=(120, 160, 3), *args, **kwargs):
        super(NvidiaModel, self).__init__(*args, **kwargs)
        self.model = customArchitecture(num_outputs, input_shape)
        self.compile()

    def compile(self):
        self.model.compile(optimizer="adam",
                loss='mse')

    def run(self, img_arr):
        img_arr = img_arr.reshape((1,) + img_arr.shape)
        steering, throttle = self.model.predict(img_arr)
        return steering[0][0], throttle[0][0]
```

To make the new class usable with Donkey, a new type of model has to be defined in Donkey's ***utils.py*** script, inside the ***get_model_by_type*** function:

```
elif model_type == "nvidia":
            from donkeycar.parts.nvidia import NvidiaModel
        kl = NvidiaModel(input_shape=input_shape)
```

The next step is implementing the architecture shown above (Figure: 94) in Keras:

```
def customArchitecture(num_outputs, input_shape):

    img_in = Input(shape=input_shape, name='img_in')
    x = img_in
    # Dropout rate
    keep_prob = 0.9
    rate = 1 - keep_prob
    # Convolutional Layer 1
    x = Convolution2D(filters=24, kernel_size=5, strides=(2, 2), input_shape =
    ↪  input_shape)(x)
    x = Dropout(rate)(x)
    # Convolutional Layer 2
    x = Convolution2D(filters=36, kernel_size=5, strides=(2, 2),
    ↪  activation='relu')(x)
    x = Dropout(rate)(x)
    # Convolutional Layer 3
    x = Convolution2D(filters=48, kernel_size=5, strides=(2, 2),
    ↪  activation='relu')(x)
    x = Dropout(rate)(x)
    # Convolutional Layer 4
    x = Convolution2D(filters=64, kernel_size=3, strides=(1, 1),
    ↪  activation='relu')(x)
    x = Dropout(rate)(x)
    # Convolutional Layer 5
    x = Convolution2D(filters=64, kernel_size=3, strides=(1, 1),
    ↪  activation='relu')(x)
    x = Dropout(rate)(x)

    # Flatten Layers
    x = Flatten()(x)
    # Fully Connected Layer 1
    x = Dense(100, activation='relu')(x)
    # Fully Connected Layer 2
    x = Dense(50, activation='relu')(x)
    # Fully Connected Layer 3
    x = Dense(25, activation='relu')(x)
    # Fully Connected Layer 4
    x = Dense(10, activation='relu')(x)
    # Fully Connected Layer 5
    x = Dense(5, activation='relu')(x)
    outputs = []
    outputs.append(Dense(1, activation='linear', name='n_outputs1')(x))
    outputs.append(Dense(2, activation='linear', name='n_outputs1')(x))
    model = Model(inputs=[img_in], outputs=outputs
    return model
```

After training on a small dataset (<5000 records), the Nvidia architecture sucessfuly navigates the basic simulator track, albeit not perfectly, but successfully nonetheless. This means that the approach works and that it can be used for creating a more complex custom architecture.

8.6. Optimizing and training a model

The following subchapters were written through a longer period of time with the knowledge obtained through experimentation and model training, and through best practices shared in the literature and the machine learning community. Two most notable sources of insights and ideas behind the instructions written below are the ***Deep Learning*** book written by Goodfellow et al. [51] and the Deeplearning.ai specializations made by Andrew Ng [63].

8.6.1. Data preparation

The first step in preparing a dataset for training use is to split the dataset into multiple parts that serve a specific training, validation and testing purpose. One common practice is to split the data into three parts, explained in the following subchapter.

8.6.1.1. Training set, cross validation/dev set and test set

Figure 95: Training set, cross validation/dev set and test set

- A model can be trained the training set, and validated using the dev set.
- Since the dev set is used to tune the model hyperparameters, they're being fit to the data in the dev set. That's why after choosing the exact hyperparameters wanted and after completely training the model, it should be tested using the test set, which is data it has never ever seen before, and provides a "real-world" benchmark on how it performs.

8.6.1.2. Splitting the data (size-wise)

- If a small amount of data is used, in the tens of thousands the pre big-data way of splitting the data can be used:
  - 60% for the training set
  - 20% for the dev and 20% for the test set
- If the dataset has millions of records:
  - 1%, or about 10 000 examples can be used for the dev/test sets
  - 99% for the training set
- If the dataset has hundreds of millions of records or more:
  - 99.5% can be used for the training set
  - about 0.4% for the dev set
  - and about 0.1% for the test set

8.6.1.3. Data distribution

It is important to make sure that the training and dev/test sets come from the same distribution. An example would be:

- If the training data is really high in resolution, but the data used during inference is lower quality due to lower resolutions, shaky camera, dirty lenses, and so on, the model won't perform well.
- The training data should be as similar as possible to real world data used during inference, and as much possible situations should be covered, which includes different weather conditions, bumpy roads, and so on.

The main reason for this type of mismatch is that when there isn't enough training data, it is easily acquireable through other sources, for an example if a YouTube video of a car driving around town that has a super high res camera was taken and added to the training data, while the RC car has a low resolution camera used during inference.

8.6.1.4. Determining if a model has high variance or bias (or both)

- ***High variance:*** the model is overfitting the training data and performing poorly on new unseen data.
- ***High bias:*** the model isn't even performing well on the training data.
- ***High bias and high variance:*** the model underfits the training data but performs even worse on the dev data.
- By looking at how well the model performs on the training data, it can be determined how much bias it has; if it has a low error, it has a low bias, if it has a large error, it has large bias.
- By looking at how well the model performs on the dev set, it can be determined how much variance it has; if it performs much worse than on the training set, it has high variance, if it performs slightly worse or the same, it has low variance and generalizes well.

8.6.2. Basic training recipe

- The model should perform well on the training data (low bias)
- The model should perform well on the dev set data (low variance)

8.6.2.1. Solving high bias

- **Almost always works**: Try a bigger network: more hidden layers, more hidden units.
- **Sometimes works, but never hurts**: Try training it longer: give it more time or use more advanced optimization algorithms.
- **Maybe it'll work, maybe it won't**: Try finding a better neural net architecture: try finding an architecture that's proven to work for the specific problem the network should solve.
- **Consider asking**: is this even possible to do? If a classifier is being trained on very blurry low res images, is it even possible to do so? What is the base error for that problem?

8.6.2.2. Solving high variance

- **Best way to solve it**: Get more data, it can only help. But sometimes it's impossible to get more data.
- **Almost always helps**: Regularization
- **Maybe it'll work, maybe it won't**: Try finding a better neural net architecture. Same as for the large bias problem.

8.6.3. Regularization

To gain some intuition about regularization in neural nets, a walk through implementing L2 regularization in a neural net is described below. Since this thesis will be dealing mostly with computer vision with the self-driving RC car, batch normalization and dropout regularization will most likely be used rather than L2, which are used much more often in computer vision.

If a cost function is defined as:

$$\mathcal{J}(w^{[1]}, b^{[1]}, ..., w^{[l]}, b^{[l]}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \tag{8.4}$$

It can be regularized by adding a regularization term:

$$\mathcal{J}(w^{[1]}, b^{[1]}, ..., w^{[l]}, b^{[l]}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2n} \sum_{l=1}^{l} ||w^{[l]}||^2 \tag{8.5}$$

The first part of the equation uses the regularization parameter λ, which is a hyperparameter to be tuned using the dev set, and divides it by $2n$, which is just a scaling constant. The second part uses the Frobenius norm [64] (the L2 norm of a matrix), which sums up the squares of all the elements of all w matrices used:

$$||w^{[l]}||^2 = \sum_{i=1}^{n^l} \sum_{j=1}^{n^{[l-1]}} (w_{i,j}^{[l]})^2 \tag{8.6}$$

w is a $(n^{[l]}, n^{[l-1]})$ dimensional matrix, where the first dimension represents the number of units in the l-th layer, and the second represents the number of units in the previous layer.

When implemented, during backprop the $W^{[l]}$ matrix gets multiplied by $\frac{\lambda}{n}$, a number slightly smaller than 1, which is why it's also called weight decay [51], since the weight loses just a bit of it's value.

8.6.3.1. Why regularization helps

If the λ parameter in the regularization term below is set to a large value, it will set the values of $w^{[l]}$ very close to zero.

$$\mathcal{J}(w^{[1]}, b^{[1]}, ..., w^{[l]}, b^{[l]}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2n} \sum_{l=1}^{l} ||w^{[l]}||^2 \tag{8.7}$$

So what happens if $w^{[l]}$ is close to zero? It will cause $z^{[l]}$ to have a very small range of values, close to zero.

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]} \tag{8.8}$$

If $tanh(z)$ is used as the activation function:

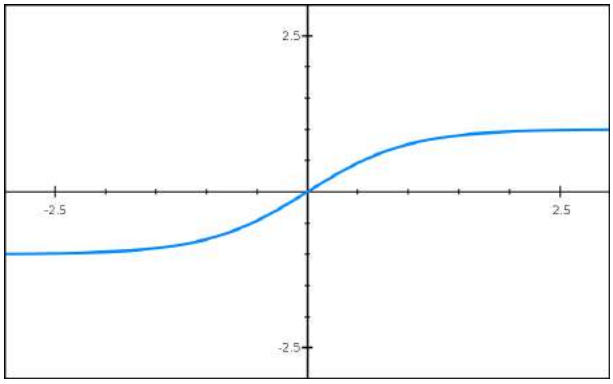

Figure 96: $tanh(z)$

If the value of $z^{[l]}$ can be only a small range of values close to zero, the activation function will start looking like a linear function, as shown below:

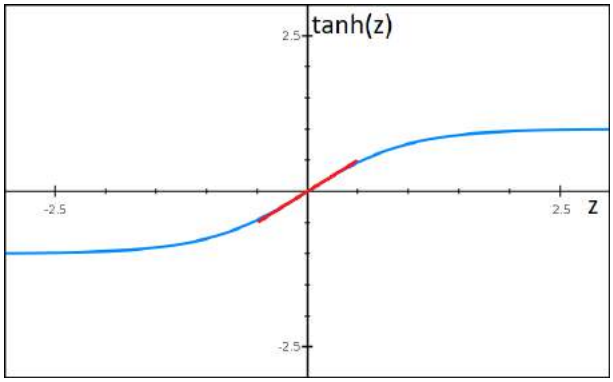

Figure 97: $tanh(z)$

Which in turn means every layer in the neural network will begin to look like a linear layer, which will cause the network to be able to approximate only a linear function to the dataset, which will cause the model to go from high variance to high bias.

Of course, λ parameter shouldn't ever be set to a very large value, and setting it to a reasonable value will get rid of a high variance problem and not cause high bias instead.

8.6.3.2. Applying regularization layers

To decrease variance, the dropout and batch normalization layers described in the earlier chapters will be used throughout the thesis.

8.6.4. Defining custom metrics for model performance

First, two terms will be defined using a classifier example; precision and recall.

- **Precision:** of the examples that the classifier recognizes as X, how many examples actually are X.
- **Recall:** of the examples that are actually X, how many of them does the classifier recognize as X.

These two metrics can be combined into a single number as a harmonic mean [65] of the two, which is often called the F1 score [66]:

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} \tag{8.9}$$

Now an example of trying to decide between three implementations of a classifier with 3 possible classes will be used:

Implementation/Classes	A (precision)	A (recall)	B (precision)	B (recall)	C (precision)	C (recall)
First implementation	95%	90%	89%	88%	92%	89%
Second implementation	93%	89%	92%	90%	92%	90%
Third implementation	89%	88%	95%	93%	90%	89%

Table 1: Deciding between three classifiers

From the table above, it is difficult to tell which classifier performs the best. This is what the table would look like if the F1 score was used for all of the possible classes:

Implementation/Classes	A (F_1 score)	B (F_1 score)	C (F_1 score)
First implementation	92.4%	88.5%	90.5%
Second implementation	91%	91%	91%
Third implementation	88.5%	94%	89.5%

Table 2: F1 score

Looks much simpler than the first table, but still could be simpler. Since the goal is most probably to make the classifier work as best as possible over all classes, take the average of all of the 3 F_1 scores can be calculated to get one simple number that measures how well the classifier works:

Implementation/Classes	Average F_1 score
First implementation	90.5%
Second implementation	91%
Third implementation	90.7%

Table 3: Averaged F1 score

The example above could've easily had a hundred different classes, with a dozen different implementations. That would've been impossible to look at even if F_1 scores were used for every class.

When trying out a bunch of different values of hyperparameters and implementation details, it's much faster to iterate and much easier to decide between all possible implementations if a single number is calculated that describes how well it performs. This is a pretty good way to get one, even considering the amount of some fine-grained details lost during the process. Those details can always be looked up after eliminating some implementations, while keeping only a few of the best ones to decide from.

8.6.5. Minimum requirements of the model

One important thing to notice is that there could be other important metrics to look at other than how accurate a classifier is.

A classifier that classifies obstacles the car will collide with if not avoided will be used. Surely, the goal is to correctly classify as much of those as possible, so a table is created using the previously mentioned metrics:

Classifier/Score	F_1 score
First classifier	92%
Second classifier	97%
Third classifier	99%

Table 4: Three classifiers compared

One could assume that the best classifier is the third one, which would be correct but only in terms of it correctly classifying as much of the objects in the car's path as possible, but there is one thing the table above doesn't take into account: the time it takes it to classify an object.

Classifier/Score	F_1 score	Average runtime
First classifier	92%	20 ms
Second classifier	97%	70 ms
Third classifier	99%	3000 ms

Table 5: Minimum requirements

So while the third classifier would recognize most of the objects the car was on a collision path with, by the time that it did, the car would have already crashed into them. Taking that in mind, the second classifier is obviously the better choice.

So it's important to keep in mind that while trying to optimize one of the metrics of the model, it is also important to pay attention to the other minimum requirements the model must have in order to actually be usable in the real world, as per the above real-time necessary example.

8.6.6. Data augmentation

The following track will be taken as an example of a track that the RC car would be trained on:

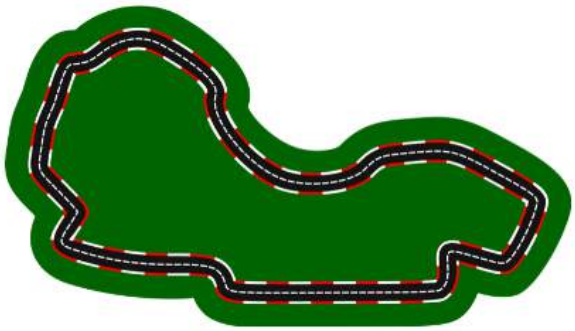

Figure 98: Melbourne Grand Prix Circuit
(Source: Reina)

After running a few laps on the track and saving the data, if the track was driven around in the standard counter-clockwise direction, the model will inadvertently be slightly biased to always turn a bit to the left. The reason for this is that globally, the car is always turning a bit to the left. So if only this data is included in the training set, the car will always be inclined to turn just a bit to the left.

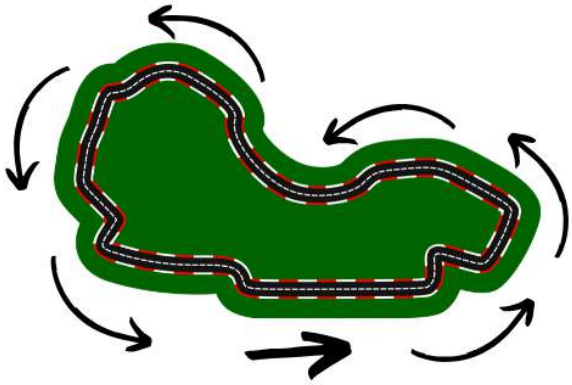

Figure 99: Melbourne Grand Prix Circuit
(Adapted from: Reina)

A way of solving this, other than driving around the track in both directions, is to create mirrored copies of all the images in the dataset:

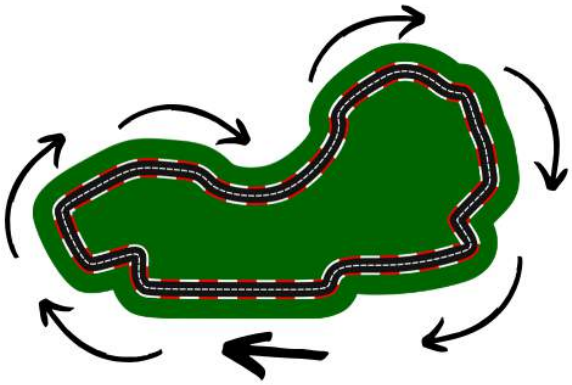

Figure 100: Melbourne Grand Prix Circuit
(Adapted from: Reina)

8.6.7. Early stopping

When training the model, the validation error should get lower over time, following the training error curve, but then at one moment it takes off and get much worse over time. One possible solution is stopping the training early, when the validation error starts to increase:

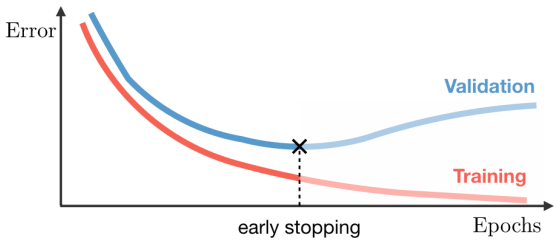

Figure 101: Early stopping
(Source: [68])

But by doing so, two optimization problems that were being solved as separate before are now coupled:

- The optimization of the cost function
- The prevention of overfitting to the dataset

By doing so, it will be impossible to maximally minimize the cost function, since it's obvious that the error could get much lower if the training wasn't stopped early, because it is simultaneously trying not to overfit the data.

8.6.8. Hyperparameter optimisation

One of the most important parts of training a model to perform well is to choose hyperparameters. They can be choosed by using the tools that come with machine learning frameworks, or they can be manually tuned by changing the values by a small amount and seeing the resulting effect on the model performance. Hyperparameters in order by importance are:

- learning rate (alpha)
- beta, number of hidden units, mini-batch size
- number of layers, learning rate decay

Hyperparameters should also be re-evaluated every once in a while, because as the amount of data increases, the ones chosen beforehand are perhaps no longer the best ones. This is often reffered to as hyperparameters *getting stale*. They can also get stale if a new GPU/CPU is used, the network is changed a bit or for any number of reasons.

9. Modeling and training the RC car to autonomously drive and perform behaviours

This chapter will introduce a new proposed type of network architecture which is based on training multiple specialized subnetworks in an architecture whose outputs would converge into a final decision-making layer. The chapter will explain the idea and reasoning behind the proposed architecture, explore similar ideas used in the industry and will demonstrate the plausibility of the proposed architecture by implementing two iterations of a model based on the architecture with iterative addition of specialized subnetworks, the first of which will extract lane lines from raw images using methods described in earlier chapters, and second of which will allow the car to perform behaviours, where lane changing behaviour will be used as a proof of concept.

9.1. First iteration of the new proposed network architecture

First, the image below will show the architecture diagram of the neural network architecture that will be proposed, for easier understanding and reference further on:

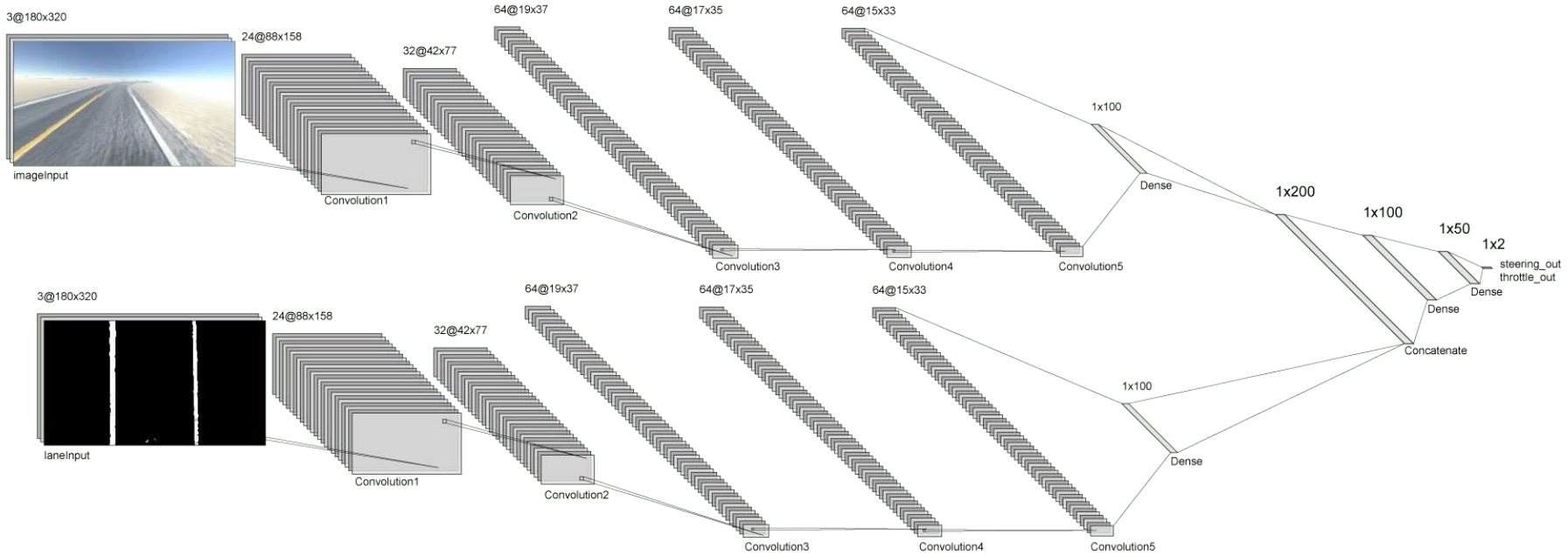

Figure 102: First iteration of the proposed architecture

The model will consist of two parallel CNNs, each of which end with a dense 100-unit layer, which are then concatenate and passed through three additional dense layers, and end with two linear activations. The model has about 6.5 million parameters, which take up about 2GB of VRAM, so it should be runnable on the Jetson Nano with half as much RAM to spare. One CNN takes the raw image as its input and the other takes a thresholded image using the extraction procedure explained in the previous chapter.

9.2. The idea of teaching a vehicle to drive autonomously

While true that a whole separate (and rather big) CNN is an overkill for the thresholded binary image, the main reason for doing it is testing if its possible to have multiple specialized parts of the architecture all siphon in to a smaller final part (the last n dense layers) in order for the car to make a decision where to steer and how much to throttle, which is more or less how drivers do everything while they're in a car, parking, driving, they just combine multiple inputs:

- Checking their mirrors and dead angle to make sure they're safe to do a maneuver
- Recognizing a sign that says what exit they should take in order to get to their destination
- Actually knowing where they are with respect to that exit and by what path can they get to it
- Actually give the appropriate steering and throttle to actually perform the maneuver

As seen above, no matter what the driver is doing in a car, be it parking, changing lanes or driving to a destination, all they can really do to control the car is turn the steering wheel and control its speed (assuming no gear shifting), they're just taking in the input from their surroundings, mostly using their eyes, which they analyze through a series of specialized procedures which ultimately lead them to control their car based on the decisions they've made, in order to perform a maneuver.

So what was intended as the final part of this thesis is to have a series of specialized parts in the net, which could even be call smaller subnets, which would take the input images and extract highly specific data from it, using (relatively) specialized procedures, which would then plug into the final layer, along with the first convolutional network that uses the raw input image, which should give the final part of the network enough context about the world and enough information in order to appropriately control the RC. The above idea, explained using a diagram, would look something like this:

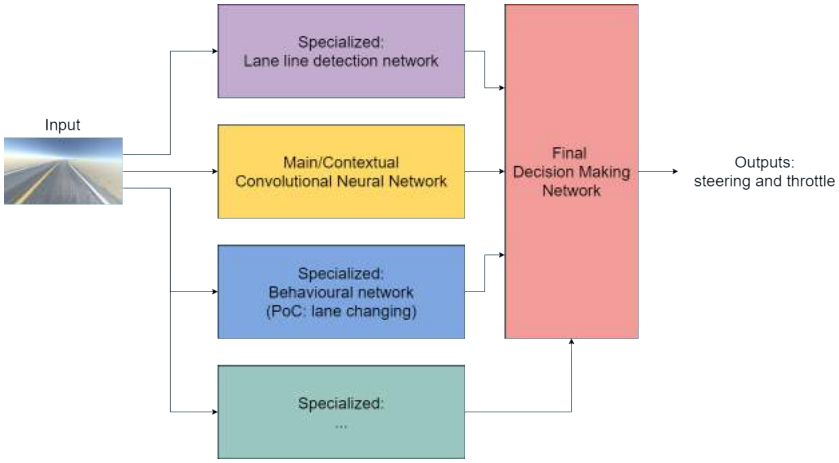

Figure 103: Multiple specialized subnets architecture diagram

While implementing the above described architecture, a similar idea was described by Karpathy in the ***Pytorch at Tesla*** presentation, where the author described the use of a single large shared backbone out of which 48 networks with a total of 1000 output predictions emerge, which are called ***HydraNets***. [69] The described approach used at Tesla makes more sense than the architecture shown in the diagram above, since it uses a single large shared backbone to enable the 48 separate networks to share the knowledge about the world that they would otherwise have to learn separately, but further on in the thesis, it will be shown that the architecture with multiple specialized subnets converging into one final dense layer also performs well.

9.2.1. Implementing the proposed architecture

The first step, as mentioned earlier, is to create a new class file, which will be named ***orimodel.py***:

```
class OriModel(KerasPilot):
    '''
    Custom model that takes an input image and feeds it and a preprocessed version
↪   of it to the model.
    The preprocessing converts the image to HSL color space, extracts the S channel
↪   and thresholds it.
    The thresholded S channel is passed to the model to help find lane lines easier.
    '''
    def __init__(self, model=None, input_shape=(180, 320, 3), *args, **kwargs):
        super(OriModel, self).__init__(*args, **kwargs)
        self.model = oriModel(inputShape=input_shape)
        self.compile()

    def compile(self):
        self.model.compile(optimizer=self.optimizer,
                loss='mse')
```

The model performs the lane extraction procedure at runtime, which was benchmarked and made sure that it doesn't in any way bottleneck or impact the inference performance:

```
def run(self, inputImage):
        # Preprocesses the input image for easier lane detection
        extractedLaneInput = self.processImage(inputImage)
        # Reshapes to (1, height, width, channels)
        extractedLaneInput = extractedLaneInput.reshape((1,) +
        ↪  extractedLaneInput.shape)
        inputImage = inputImage.reshape((1,) + inputImage.shape)
        # Predicts the output steering and throttle
        steering, throttle = self.model.predict([inputImage, extractedLaneInput])
        print("Throttle: %f, Steering: %f" % (throttle[0][0], steering[0][0]))
        return steering[0][0], throttle[0][0]
```

The lane extraction methods are also included as class methods:

```
def warpImage(self, image):
    # Define the region of the image we're interested in transforming
    regionOfInterest = np.float32(
    [[0,  180],  # Bottom left
    [112.5, 87.5], # Top left
    [200, 87.5], # Top right
    [307.5, 180]]) # Bottom right

    # Define the destination coordinates for the perspective transform
    newPerspective = np.float32(
    [[80,  180],  # Bottom left
    [80,    0.25],  # Top left
    [230,   0.25],  # Top right
    [230, 180]]) # Bottom right
    # Compute the matrix that transforms the perspective
    transformMatrix = cv2.getPerspectiveTransform(regionOfInterest, newPerspective)
    # Warp the perspective - image.shape[:2] takes the height, width, [::-1]
    ↪  inverses it to width, height
    warpedImage = cv2.warpPerspective(image, transformMatrix, image.shape[:2][::-1],
    ↪  flags=cv2.INTER_LINEAR)
    return warpedImage

def extractLaneLinesFromSChannel(self, warpedImage):
    # Convert to HSL
    hslImage = cv2.cvtColor(warpedImage, cv2.COLOR_BGR2HLS)
    # Split the image into three variables by the channels
    hChannel, lChannel, sChannel = cv2.split(hslImage)
    # Threshold the S channel image to select only the lines
    lowerThreshold = 65
    higherThreshold = 255
    # Threshold the image, keeping only the pixels/values that are between lower and
    ↪  higher threshold
    returnValue, binaryThresholdedImage =
    ↪  cv2.threshold(sChannel,lowerThreshold,higherThreshold,cv2.THRESH_BINARY)
    # Since this is a binary image, we'll convert it to a 3-channel image so OpenCV
    ↪  can use it
    thresholdedImage = cv2.cvtColor(binaryThresholdedImage, cv2.COLOR_GRAY2RGB)
    return thresholdedImage

def processImage(self, image):
    warpedImage = self.warpImage(image)
    # We'll normalize it just to make sure it's between 0-255 before thresholding
    warpedImage = cv2.normalize(warpedImage,None,0,255,cv2.NORM_MINMAX,cv2.CV_8U)
    thresholdedImage = self.extractLaneLinesFromSChannel(warpedImage)
    one_byte_scale = 1.0 / 255.0
    # To make sure it's between 0-1 for the model
    return np.array(thresholdedImage).astype(np.float32) * one_byte_scale
```

Finally, the model implementation in Keras starts by defining the input layers:

```
def oriModel(inputShape):

    # Dropout rate
    keep_prob = 0.9
    rate = 1 - keep_prob

    # Input layers
    imageInput = Input(shape=inputShape, name='imageInput')
    laneInput = Input(shape=inputShape, name='laneInput')
```

The first convolutional neural network takes the extracted lane image as its input and performs a series of convolutions, using dropout regularisation and the Leaky ReLU activation function in order to mitigate the dying ReLU problem described earlier:

```
    # Input image convnet
    x = imageInput
    x = Conv2D(24, (5,5), strides=(2,2), name="Conv2D_imageInput_1")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(32, (5,5), strides=(2,2), name="Conv2D_imageInput_2")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(64, (5,5), strides=(2,2), name="Conv2D_imageInput_3")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_imageInput_4")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_imageInput_5")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Flatten(name="flattenedx")(x)
    x = Dense(100)(x)
    x = Dropout(rate)(x)
```

The second, generalized convolutional neural network takes in the raw image as its input in order to gain as much information and contextual knowledge about the world as possible, and also uses dropout regularisation and the Leaky ReLU activation function:

```
# Preprocessed lane image input convnet
y = laneInput
y = Conv2D(24, (5,5), strides=(2,2), name="Conv2D_laneInput_1")(y)
y = LeakyReLU()(y)
y = Dropout(rate)(y)
y = Conv2D(32, (5,5), strides=(2,2), name="Conv2D_laneInput_2")(y)
y = LeakyReLU()(y)
y = Dropout(rate)(y)
y = Conv2D(64, (5,5), strides=(2,2), name="Conv2D_laneInput_3")(y)
y = LeakyReLU()(y)
y = Dropout(rate)(y)
y = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_laneInput_4")(y)
y = LeakyReLU()(y)
y = Dropout(rate)(y)
y = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_laneInput_5")(y)
y = LeakyReLU()(y)
y = Flatten(name="flattenedy")(y)
y = Dense(100)(y)
y = Dropout(rate)(y)
```

Finally, both feature maps created by the two convolutional neural networks can be converged into the third and final decision-making neural subnetwork, comprised of three dense layers and two output neurons that produce steering and throttle values for the RC car:

```
# Concatenated final convnet
c = Concatenate(axis=1)([x, y])
c = Dense(100, activation='relu')(c)
c = Dense(50, activation='relu')(c)

# Output layers
steering_out = Dense(1, activation='linear', name='steering_out')(o)
throttle_out = Dense(1, activation='linear', name='throttle_out')(o)
model = Model(inputs=[imageInput, laneInput], outputs=[steering_out,
↪  throttle_out])

return model
```

The model was trained using about 10 000 records made using the simulator on a randomly generated track. The training lasted for 12 epochs which took 21m 45s on an RTX 2060. The final validation loss was 0.003665 and the model has been tested and can successfully autonomously navigate a completely different, randomly generated track with some minor error (2-3 lane crossings) while staying on the road throughout the entire track.

The graphs below show the progress of the model training, specifically the model training loss, the model validation loss and the training loss overlaid on the validation loss:

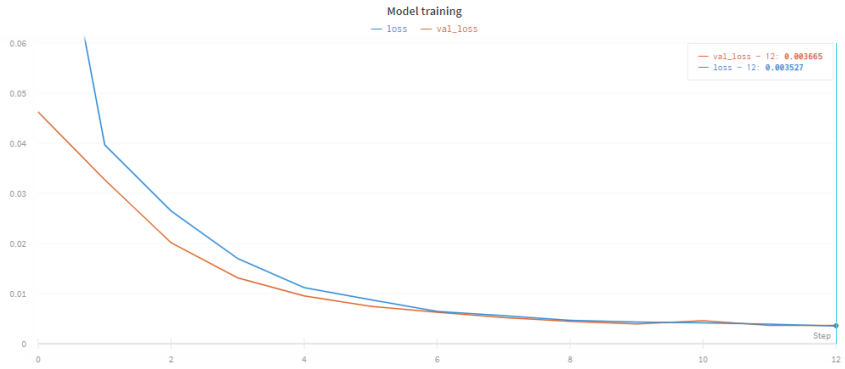

Figure 104: Training and validation loss graph

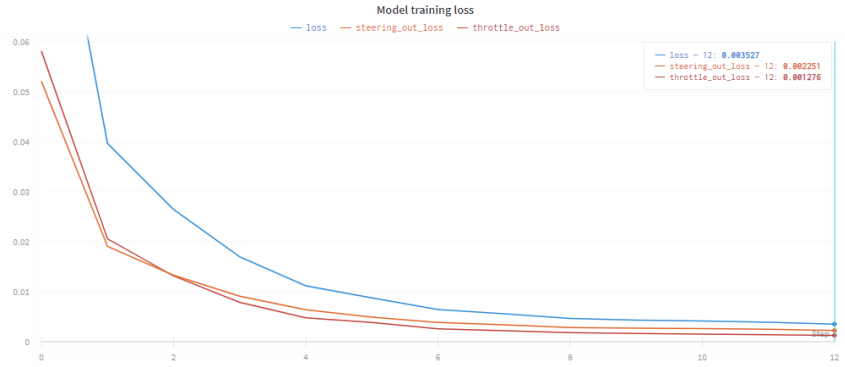

Figure 105: Training loss

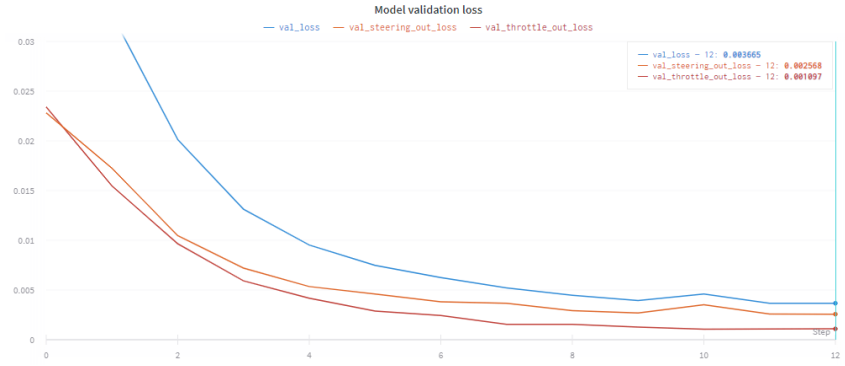

Figure 106: Validation loss

9.3. Second iteration of the new proposed network architecture

As before, first the architecture diagram will be shown below:

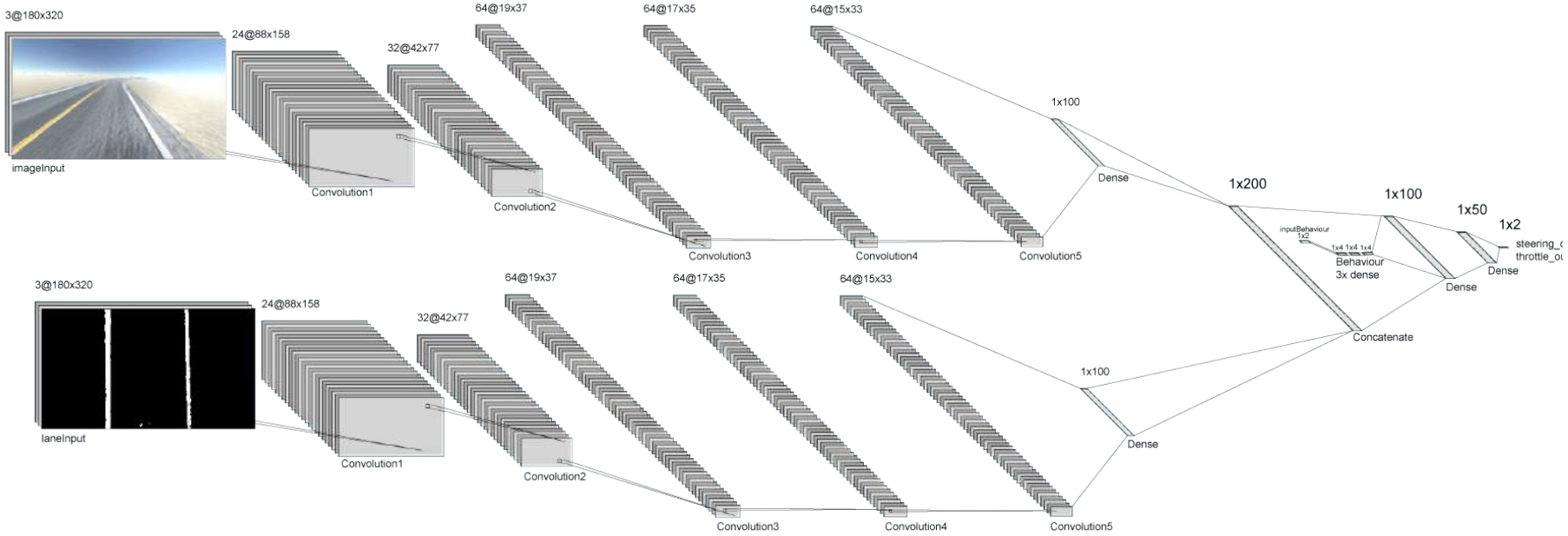

Figure 107: Second iteration of the proposed architecture

The new architecture just incrementally adds a new specialized subnetwork that is resposnible for learning behaviours. The behavioural subnetwork is consisted of three small dense layers which converge into the last decision-making subnetwork. It can be seen on the right side of the diagram, just before the first 100-unit dense layer of the last subnetwork.

9.3.1. Recording behavioural data

If a network is going to be trained for certain behaviours, it needs labeled training data with said behaviours. This can be done in many ways, even make other networks trigger certain behaviours, for an example if a classifier detects a stop sign, it could trigger a stopping behaviour. But for this proof of concept, a simple button click on a controller will be used.

DonkeyCar comes with some support for behavioural data recording and networks, which can be enabled by setting an appropriate variable in the configuration file. Then, a list of behaviours can be defined which are then mapped to an instance of the ***BehaviourPart*** class, which allows iterating through a given list of behaviours or explicit behaviour setting using the built-in class methods:

```
# List of behaviours
behaviours = ['Left_Lane', 'Right_Lane']

# Example behavioural data that the BehaviourPart class writes
# If a certain behaviours is triggered or iterated through
"behavior/state": 0,
"behavior/label": "Left_Lane",
"behavior/one_hot_state_array": [1.0, 0.0]
```

As explained earlier, a custom controller mapping can be used to map a button to the ***BehaviourPart*** method which iterates through a list of behaviours:

```
# The function which is used to set a button press to trigger a function
def set_button_down_trigger(self, button, func):
    # assign a string button descriptor to a given function call
    self.button_down_trigger_map[button] = func
```

9.3.2. Incrementally adding a behavioural subnetwork to the previously implemented model

The previous model implementation can just be upgraded by adding a behavioural subnetwork:

```
# New input layer
behaviourInput = Input(shape=(numberOfBehaviourInputs,), name="behaviourInput")

# ConvNet parts ...

# Behavioural net
z = behaviourInput
z = Dense(numberOfBehaviourInputs * 2, activation='relu')(z)
z = Dense(numberOfBehaviourInputs * 2, activation='relu')(z)
z = Dense(numberOfBehaviourInputs * 2, activation='relu')(z)

# Concatenating the convolutional networks with the behavioural network
o = Concatenate(axis=1)([z, c])
o = Dense(100, activation='relu')(o)
o = Dense(50, activation='relu')(o)

# Output layers ...

# Update the model inputs
model = Model(inputs=[imageInput, laneInput, behaviourInput], outputs=[steering_out,
↪  throttle_out])
```

The number of behaviour inputs should also be passed as an input to the ***OriModel*** class constructor, and the behavioural data as an input to the model. That way any number of behaviours can be easily trained.

After training the second iteration of the architecture on a even smaller dataset of around 10 000 records with about 20 lane changes for 13 epochs, the final validation loss was 0.003947. The model has been tested and the RC car can autonomously navigate a completely different, randomly generated track with behaviours performed without any issues while staying on the road throughout the entire track.

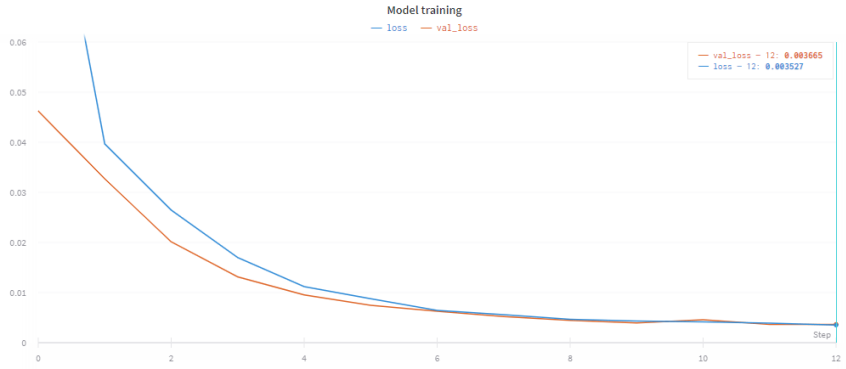

Figure 108: Training and validation loss graph

10. Testing the implemented proposed architecture

After performing a basic sanity check by training the implemented proposed architecture and verifying everything works as intended, the network was trained on a larger, but relatively a still very small data set which consisted of about 25 000 records with around 50 lane changes, generated using the simulator, due to constraints in available real world space which wouldn't allow such a complex track to be generated. The trained model was then tested in the simulator, which allows more difficult tests with a random complex generated track. The RC car successfully navigated the complete track, while performing lane changing behaviours on demand.

The RC car starts in the left lane behaviour and follows the track while staying in the left lane:

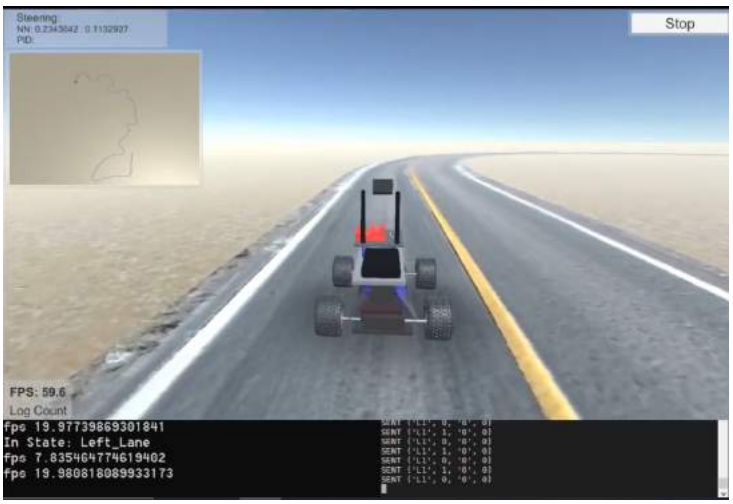

Figure 109: Simulator testing: Left lane behaviour

When a lane line changing behaviour is triggered, the RC car performs a switching manoeuvre, which the RC car has implicitly learned to delay if a tight corner is being made with higher speeds by following the data set examples:

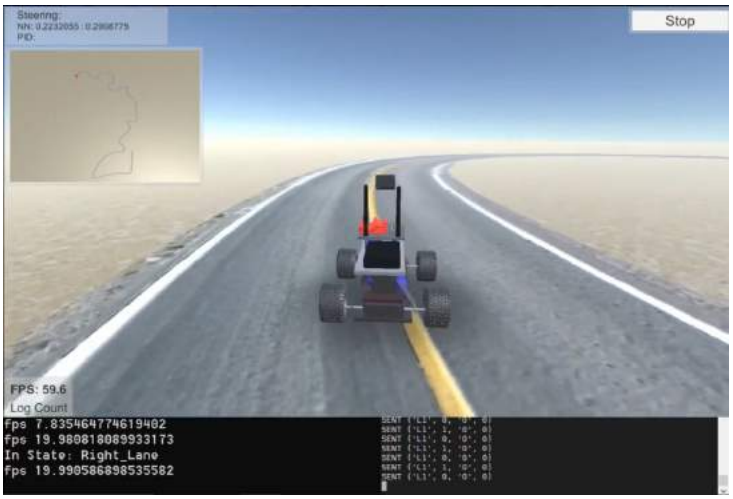

Figure 110: Simulator testing: Switching to right lane behaviour

The RC car then stays in the right lane while following the track, until a lane changing behaviour is triggered again:

Figure 111: Simulator testing: Returning to left lane behaviour

Due to the aforementioned real world constraints such a complex track could not be made since a very large surface area would be necessary. The reason being that the RC car model is a scale 1:10 model which means the turn radius of each wheel must be no larger than 15 degrees, and the turn radius of the car cannot be larger than 30 degrees, which in turn requires a larger surface area for corners and lanes. That being said, a smaller, very simple track was made to test just the lane keeping behaviour, which the RC car successfully navigated with minimal errors:

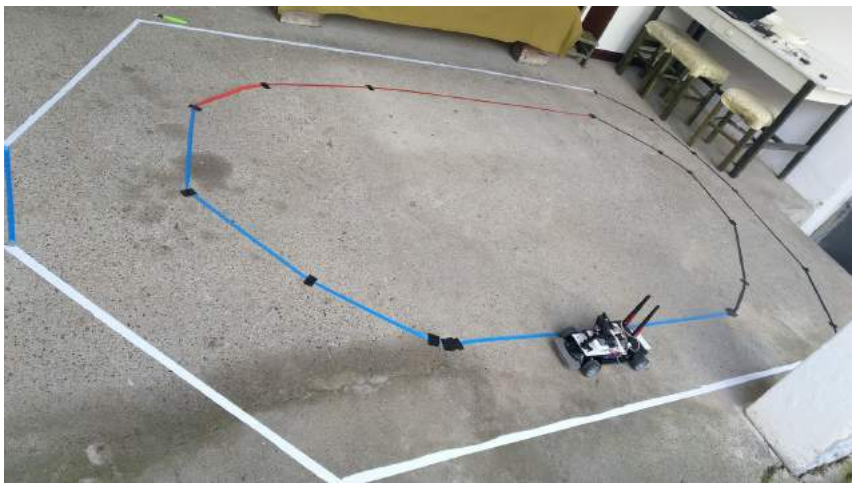

Figure 112: Real track testing: lane keeping

11. Conclusion

The thesis resulted in a fully functional custom hardware platform with an end-to-end machine learning pipeline that allows automated data collection and labelling, with complex models such as the second iteration of the proposed architecture implementation which allows the car to autonomously navigate randomly generated, never before seen tracks and even perform behaviours. The rapid prototyping pipeline has also proved functional, with the trained models even being interchangeable between the simulated vehicle and the real one, which could be used as a valuable tool for pre-training real vehicles to mitigate the otherwise unavoidable errors at the beginning stages of an autonomous vehicles training.

There are a number of avenues of further research, from applying reinforcement learning to the simulated environment to modelling even more complex behaviours and specialised sub-nets in the proposed, and experimentally proven architecture. The following sub-chapters will go through some of the hard-learned lessons, insights and suggestions for others that may try and reproduce this project.

11.1. RC car limitations

One of the biggest obstacles to real world testing is the sheer amount of surface area a track requires for the RC car to be able to navigate on it.

The easiest possible solution would be to get a smaller scale RC car, which should work just as good, but probably won't go as fast as a 1:10 would. If the 1:10 scale is non-negotiable due to speed and equipment requirements, one possible mitigation to this issue is to install longer turnbuckle shafts and 5mm hop ups to the RC car.

One other possible suggestion is to use a different body type, such as a truggy or a buggy, which by default have a slightly larger turn radius, but come with a number of stability and speed disadvantages for on-road use.

11.2. Power limitations

The Jetson Nano has proven unstable with a regular 5V@2A power supply with frequent shutdowns even when used without any peripherals (in a headless setup).

If a power bank that supports USB power delivery (PD) protocol is used, a direct current (DC) fast charge trigger polling detector board can be used, which takes in a PD input from one side and negotiates the necessary voltage for a regular DC USB output on the other side, which would easily provide the Jetson Nano with 5V@4A (10W) for the barrel jack power port. The Nano doesn't support the PD protocol by default, which is why such an intermediary board is necessary. The power bank used for the build in this thesis supports up to 65W of power delivery and supports a wide range of voltages and currents through its PD output port, so 10W would be no issue for it.

Another possible solution is to use the RC car LiPo battery to power the car and the Nano, which can be connected to the Nano through a DC-to-DC step-down or buck converter which would step down the 7.4 battery voltage to the 5V the Nano needs, while stepping up its current. This approach is not recommended since coupling the Nano to the RC car battery means that the Nano shuts down when the car battery is drained, which could mean data loss and all sorts of troubles. Also, this approach means that each time a car battery is switched, the Nano needs to be booted up again and eventual configuration and starting procedures have to be performed all over again.

11.3. The simulator

As noted earlier, there is no real substitute for real world data, and a simulator can simulate physics and other conditions with very limited fidelity. That being said, a well made and tuned simulator can be an incredible time saver and rapid prototyping tool, and should most definitely be used.

The simulator used throughout this thesis could be improved in many ways, which would allow for even more rigorous model testing and prototyping. For an example, different weather conditions could be added, such as snow or rain, higher resolution road and lane textures could be used, a procedural generated track could very easily be implemented which would allow for autonomous training if used in conjunction with the reinforcement learning approach.

Even without those improvements, the simulator has proven a very valuable tool for data collection, model testing and elimination. It saves a lot of time by rapidly providing feedback on model validity and performance before it gets tested in the real world through a lengthy procedure.

11.4. Moving beyond the Donkey platform

The Donkey platform proved to be an amazing tool in terms of time invested and results achieved. It implements a lot of features and absolves the user of a lot of domain knowledge and technical prerequisites. It is highly recommended to beginners without experience in robotics or embedded platforms to *get their feet wet* before moving on to more complex implementations.

That being said, Donkey has a lot of moving parts, unseemly code and some weird architectural choices. Also, it lacks support for more advanced sensors and platforms, but to be fair, that isn't its intended use. It's intended to be used by beginners interested in making a self-driving RC car with no or very little prior knowledge, and it does that splendidly with a great community behind it.

The logical next step for this project would be to create a more advanced and versatile platform using the Robot Operating System (ROS) instead of Donkey as the middle-ware, which would allow much more detailed low-level hardware controls, offer much better support for advanced sensors and hardware but also allow much better software and environment man-

agement and upgradeability. This would be an order of magnitude more difficult to implement than Donkey, but for more advanced users with experience in robotics it shouldn't present a great difficulty and would also be the much easier way to go in the long term.

11.5. Final thoughts

The project has proven to be a great way through which machine learning can be studied and applied through a hands on approach. It can seem a bit overwhelming for unassuming beginners, but when taken in bite sized steps, it is more than manageable while being a lot of fun. It also proves that working on autonomous vehicles, albeit smaller scale models, does not need to involve a huge financial investment or obligation, the necessary equipment can be obtained for a couple hundred euros or dollars and with a savvy building mindset can be made into a great platform which actually performs well in the real world. The fields of autonomous vehicles and AI in general is only beginning and this seems like a great and affordable way to introduce a practical aspect of them to many aspiring enthusiasts, some of which could end up working in and shaping the field.

Bibliography

[1] T. Litman, *Autonomous vehicle implementation predictions*. Victoria Transport Policy Institute Victoria, Canada, 2017.

[2] W. F. Truszkowski, M. G. Hinchey, J. L. Rash, and C. A. Rouff, "Autonomous and autonomic systems: A paradigm for future space exploration missions", *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 36, no. 3, pp. 279–291, 2006.

[3] A. J. Hawkins, *Waymo is first to put fully self-driving cars on us roads without a safety driver*, Nov. 2017. [Online]. Available: https://www.theverge.com/2017/11/7/16615290/waymo-self-driving-safety-driver-chandler-autonomous (visited on 02/18/2020).

[4] S. O'Kane, *Uber debuts a new self-driving car with more fail-safes*, Jun. 2019. [Online]. Available: https://www.theverge.com/2019/6/12/18662626/uber-volvo-self-driving-car-safety-autonomous-factory-level (visited on 02/18/2020).

[5] A. J. Hawkins, *Tesla's 'full self-driving' feature may get early-access release by the end of 2019*, Oct. 2019. [Online]. Available: https://www.theverge.com/2019/10/23/20929529/tesla-full-self-driving-release-2019-beta (visited on 02/18/2020).

[6] N. T. S. B. (NTSB), *Ntsb news release national transportation safety board office of public affairs*. [Online]. Available: https://www.ntsb.gov/news/press-releases/Pages/NR20200211.aspx (visited on 02/18/2020).

[7] J.-F. Bonnefon, A. Shariff, and I. Rahwan, "The social dilemma of autonomous vehicles", *Science*, vol. 352, no. 6293, pp. 1573–1576, 2016.

[8] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "On a formal model of safe and scalable self-driving cars", *CoRR*, vol. abs/1708.06374, 2017. arXiv: 1708.06374. [Online]. Available: http://arxiv.org/abs/1708.06374.

[9] D. J. Fagnant and K. Kockelman, "Preparing a nation for autonomous vehicles: Opportunities, barriers and policy recommendations", *Transportation Research Part A: Policy and Practice*, vol. 77, pp. 167–181, 2015.

[10] D. Howard and D. Dai, "Public perceptions of self-driving cars: The case of berkeley, california", in *Transportation research board 93rd annual meeting*, vol. 14, 2014, pp. 1–16.

[11] T. Oliphant, *NumPy: A guide to NumPy*, USA: Trelgol Publishing, 2006. [Online]. Available: http://www.numpy.org/ (visited on 02/01/2020).

[12] F. Chollet *et al.*, *Keras*, https://keras.io, 2015.

[13] Martı n Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: http://tensorflow.org/.

[14] G. Bradski, "The OpenCV Library", *Dr. Dobb's Journal of Software Tools*, 2000.

[15] A. C. Will Roscoe Tawn Kramer, *Donkeycar*, https://github.com/autorope/donkeycar, 2016.

[16] ModelToyCars, *Diecast scale - what does it mean? (with complete scale chart!)* [Online]. Available: https://www.modeltoycars.com/for_sale/pc/Guide-to-Diecast-Vehicle-Scale-d1.htm (visited on 02/01/2020).

[17] Tamiya, *Tamiya 58664 rc ford mustang gt4 30th anniversary tt-02 / tamiya usa*. [Online]. Available: https://www.tamiyausa.com/shop/110-4wd-shaft-drive-road-tt/rc-ford-mustang-gt4/tt-02/ (visited on 02/01/2020).

[18] RampageShop, *Redcat rampage xb-e 1/5 brushless 4x4 buggy*. [Online]. Available: https://www.rampageshop.com/rampage-xb-e.html (visited on 02/01/2020).

[19] Goolsky, *Goolsky jlb racing 21101 10.01 4wd elektro brushless 80 km / h high speed off-road truggy monster truck rtr rc auto*. [Online]. Available: https://www.amazon.de/Goolsky-Elektro-Brushless-Off-Road-Monster/dp/B01LYF8QBB (visited on 02/01/2020).

[20] Traxxas, *Stampede® 4x4: 1/10-scale 4wd monster truck*. [Online]. Available: https://traxxas.com/products/models/electric/67054-1stampede4x4 (visited on 02/01/2020).

[21] ThinkRC, *Brushed vs brushless motors*. [Online]. Available: https://www.thinkrc.com/faq/brushless-motors.php (visited on 02/01/2020).

[22] DonkeyCar, *Donkeycar official documentation*, DonkeyCar. [Online]. Available: http://docs.donkeycar.com (visited on 02/01/2020).

[23] Solarbotics, *Hxt900 9g / 1.6kg / .12sec micro servo*. [Online]. Available: https://solarbotics.com/product/25500/ (visited on 02/01/2020).

[24] Tamiya, *Rc tble-02s brushless esc / tamiya usa*. [Online]. Available: https://www.tamiyausa.com/shop/electronics/rc-esc-tble-02s-brushless/ (visited on 02/01/2020).

[25] RC-Hobbies, *Tamiya tru-08 2.4ghz receiver*. [Online]. Available: https://www.rchobbies.co.nz/tamiya-tru-08-2-4ghz-receiver/ (visited on 02/01/2020).

[26] FlashRC, *Traxxas chargeur ez-peak live 100w 2 x lipo 4s 6700mha id 2993g*. [Online]. Available: https://www.flashrc.com/traxxas/25401-traxxas_chargeur_ez_peak_live_100w_2_x_lipo_4s_6700mha_id_2993g.html (visited on 02/01/2020).

[27] GreensModels, *Lipo tester low voltage alarm buzzer*. [Online]. Available: https://greensmodels.co.uk/products/lipo-tester-low-voltage-alarm-buzzer (visited on 02/01/2020).

[28] FuseModel, *New plain large fire resistant lipo battery bag for safe charging storage*. [Online]. Available: http://www.fusemodel.com/product/New-Plain-Large-Fire-Resistant-Lipo-Battery-Bag-for-Safe-Charging-Storage.html (visited on 02/01/2020).

[29] D. Franklin, *Jetson nano brings ai computing to everyone*, 2019. [Online]. Available: https://devblogs.nvidia.com/jetson-nano-ai-computing/ (visited on 02/01/2020).

[30] Coral, *Coral dev board*. [Online]. Available: https://coral.ai/products/dev-board (visited on 02/01/2020).

[31] RaspberryPi, *Raspberry pi 4 model b specifications*. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/ (visited on 02/01/2020).

[32] SunFounder, *Pca9685 16 channel 12 bit pwm servo driver*. [Online]. Available: http://wiki.sunfounder.cc/index.php?title=PCA9685_16_Channel_12_Bit_PWM_Servo_Driver (visited on 02/01/2020).

[33] Lindinger, *Tamiya tt-02 chassis aufgebaut 1/10 ep 4wd mit 2,4ghz sender*. [Online]. Available: https://www.lindinger.at/at/fahrzeuge-und-boote/modelle/onroad-und-drift-autos/tamiya-tt-02-chassis-aufgebaut-1-10-ep-4wd-mit-2-4ghz-sender (visited on 02/01/2020).

[34] JetsonHacks, *Nvidia jetson nano j41 header pinout*. [Online]. Available: https://www.jetsonhacks.com/nvidia-jetson-nano-j41-header-pinout/ (visited on 02/01/2020).

[35] N. Developer, *Getting started with jetson nano developer kit*. [Online]. Available: https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit (visited on 02/01/2020).

[36] R. Graves, *Nvidia jetson nano: Desktop use, kernel builds, and deeper analysis*. [Online]. Available: https://syonyk.blogspot.com/2019/04/nvidia-jetson-nano-desktop-use-kernel-builds.html (visited on 02/01/2020).

[37] N. Developer, *L4t*. [Online]. Available: https://developer.nvidia.com/embedded/linux-tegra (visited on 02/01/2020).

[38] S. Desireddy and L. Torvalds, *Github linux repository: Zswap: Same-filled pages handling*. [Online]. Available: https://github.com/torvalds/linux/commit/a85f878b443f8d2b91ba76f09da21ac0af22e07f#diff-35c0f993808844bdbabad45bd7dea0 (visited on 02/01/2020).

[39] T. Kramer, *Github: Tawnkramer/sdsandbox*, 2020. [Online]. Available: https://github.com/tawnkramer/sdsandbox/tree/donkey/sdsim (visited on 02/01/2020).

[40] Wikiwand, *Pinhole camera*. [Online]. Available: https://www.wikiwand.com/en/Pinhole_camera (visited on 02/01/2020).

[41] HowThingsWork, *Howthingswork.org*, 2017. [Online]. Available: http://howthingswork.org/electronics-how-camera-lens-work/ (visited on 02/01/2020).

[42] OpenStax, *25.6 image formation by lenses*. [Online]. Available: https://pressbooks.bccampus.ca/collegephysics/chapter/image-formation-by-lenses/ (visited on 02/01/2020).

[43] Wikiwand, *Distortion (optics)*. [Online]. Available: https://www.wikiwand.com/en/Distortion_(optics) (visited on 02/01/2020).

[44] A. W. Fitzgibbon, "Simultaneous linear estimation of multiple view geometry and lens distortion", in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, IEEE, vol. 1, 2001, pp. I–I.

[45] D. C. Brown, "Decentering distortion of lenses", *Photogrammetric Engineering and Remote Sensing*, 1966.

[46] F. Bukhari and M. N. Dailey, "Automatic radial distortion estimation from a single image", *Journal of mathematical imaging and vision*, vol. 45, no. 1, pp. 31–45, 2013.

[47] Z. Zhang, "A flexible new technique for camera calibration", *IEEE Transactions on pattern analysis and machine intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000.

[48] J. Bouguet, "Matlab camera calibration toolbox", 2000.

[49] K. Levenberg, "A method for the solution of certain non-linear problems in least squares", *Quarterly of applied mathematics*, vol. 2, no. 2, pp. 164–168, 1944.

[50] Wikiwand, *Hsl and hsv*. [Online]. Available: https://www.wikiwand.com/en/HSL_and_HSV (visited on 02/01/2020).

[51] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[52] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position", *Biological cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[53] Y. LeCun, Y. Bengio, *et al.*, "Convolutional networks for images, speech, and time series", *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[54] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit", *Nature*, vol. 405, no. 6789, pp. 947–951, 2000.

[55] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, “What is the best multi-stage architecture for object recognition?”, in *2009 IEEE 12th international conference on computer vision*, IEEE, 2009, pp. 2146–2153.

[56] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines”, in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[57] A. L. Maas, A. Y. Hannun, and A. Y. Ng, “Rectifier nonlinearities improve neural network acoustic models”, in *Proc. icml*, vol. 30, 2013, p. 3.

[58] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”, in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[59] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting”, *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[60] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, *arXiv preprint arXiv:1502.03167*, 2015.

[61] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, “End to end learning for self-driving cars”, *arXiv preprint arXiv:1604.07316*, 2016.

[62] Y. Lecun, E. Cosatto, J. Ben, U. Muller, and B. Flepp, “Dave: Autonomous off-road vehicle control using end-to-end learning”, *Courant Institute/CBLL, http://www. cs. nyu. edu/yann/research/dave/index. html, Tech. Rep. DARPA-IPTO Final Report*, 2004.

[63] A. Ng, *What is machine learning? machine learning courses*. [Online]. Available: `https://www.deeplearning.ai/` (visited on 02/01/2020).

[64] W. MathWorld, *Frobenius norm*. [Online]. Available: `http://mathworld.wolfram.com/FrobeniusNorm.html` (visited on 02/01/2020).

[65] Wikiwand, *Harmonic mean*. [Online]. Available: `https://www.wikiwand.com/en/Harmonic_mean` (visited on 02/01/2020).

[66] ——, *F1 score*. [Online]. Available: `https://www.wikiwand.com/en/F1_score` (visited on 02/01/2020).

[67] Reina, *Melbourne grand prix circuit*. [Online]. Available: `https://www.kindpng.com/imgv/Tbbowi_reptile-grass-recreation-track-car-race-clipart-hd/` (visited on 02/01/2020).

[68] A. Amidi and S. Amidi, *Deep learning tips and tricks cheatsheet*. [Online]. Available: `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks` (visited on 02/01/2020).

[69] A. Karpathy, *Pytorch at tesla*. [Online]. Available: `https://www.youtube.com/watch?v=oBklltKXtDE` (visited on 02/01/2020).

List of Figures

1. RC scale comparison (Source: ModelToyCars) 4
2. Race/street body type (Source: Tamiya) 5
3. Buggy body type (Source: RampageShop) 5
4. Truggy body type (Source: Goolsky) 5
5. Truck body type (Source: Traxxas) 5
6. Brushed vs. brushless comparison (Source: ThinkRC) 6
7. Servo motor (Source: Solarbotics) 7
8. Electronic speed controller (Source: Tamiya) 8
9. Wireless receiver (Source: RC-Hobbies) 9
10. LiPo batteries and charger (Source: FlashRC) 10
11. LiPo discharge alarm(Source: GreensModels) 10
12. LiPo charging bag(Source: FuseModel) 10
13. Nvidia Jetson Nano (Source: Franklin) 11
14. Google Coral Board(Source: Coral) 11
15. Raspberry Pi 4 Model B(Source: RaspberryPi) 11
16. PCA9685 12
17. PCA9685 (Source: Lindinger) 14
18. TT-02 RTR Assembly 15
19. TT-02 RTR Assembly 15
20. 3D model of the hardware assembly 16
21. 3D printed mounting plates 17
22. The first version of the hardware assembly 17
23. Aluminium mounting plate 18
24. The final assembled version of the RC car 18

25. The two screws holding the module in place 19
26. Latches that hold the module in place 19
27. The module pops out 19
28. MicroSD Slot 20
29. Installing a fan on the Nano module 20
30. Fan PWM connector 21
31. Installing the M.2 card 21
32. Jetson Nano assembled 22
33. Jetson Nano assembled 22
34. Jetson Nano assembled 23
35. Jetson Nano pinout 23
36. Jetson Nano I2C bus connected 24
37. The ESC and servo motor connectors in an RC car 24
38. The RC car receiver with ESC and servo connectors 25
39. The PCA9685 connected to the ESC and servo 25

40. Linux kernel configuration using menuconfig 27
41. Adding the external firmware blob to the kernel binary 28
42. Saving the added external firmware blob 28
43. Patching the kernel 29
44. Building the kernel 29
45. Donkey platform as an interface (Source: DonkeyCar) 31
46. First test track 34
47. Fancier test track 34
48. RC driving around the first test track 35

49. Resolution dropdown 37
50. Resized resolution dropdown 37
51. Resolution setting script 37
52. 640x480 image 38
53. Camera preview 39
54. Packed prefab 40
55. Center of mass 40

56. Simulator calibration 41
57. Simulator calibration image 1 41
58. Simulator calibration image 2 41

59. Pinhole camera model (Source: Wikiwand) 42
60. Convex lens (Source: HowThingsWork) 43
61. Camera lens (Source: OpenStax) 43
62. Types of distortions (Source: Bradski) 44
63. Tangential distortion 45
64. EleCam Explorer action camera 48
65. Checkerboard calibration rig 49
66. Calibration photo 1 49
67. Calibration photo 2 49
68. Found imagepoints 1 50
69. Found imagepoints 2 50
70. Undistorted image 1 51
71. Undistorted image 2 51

72. An image of a highway 52
73. An image of a highway with visualized intersection of lane lines 53
74. An image of simulated road 53
75. An image of simulated road with visualized intersection of lane lines 53
76. A bird's-eye view of a highway 54
77. Starting perspective (320x180 pixels) 55
78. Starting perspective 55
79. Starting perspective 56
80. Starting perspective 57
81. Transformed perspective 57
82. RGB warped image 58
83. HSL warped image 58
84. Hue channel 59
85. Lightness channel 59
86. Saturation channel 59

87. Thresholded S channel 60
88. Histogram of the thresholded S channel 60
89. Overlaid histogram over the thresholded S channel 61
90. Visualized fitted polynomial on an unwarped image 61

91. ReLU activation function 63
92. Leaky ReLU with $\alpha = 0.3$ 64
93. Leaky ReLU with $\alpha = 0.03$ 64
94. Nvidia model architecture (Source: Bojarski, Del Testa, Dworakowski, *et al.*) . . . 66
95. Training set, cross validation/dev set and test set 69
96. $tanh(z)$ 72
97. $tanh(z)$ 73
98. Melbourne Grand Prix Circuit (Source: Reina) 76
99. Melbourne Grand Prix Circuit (Adapted from: Reina) 76
100. Melbourne Grand Prix Circuit (Adapted from: Reina) 77
101. Early stopping (Source: [68]) 77

102. First iteration of the proposed architecture 80
103. Multiple specialized subnets architecture diagram 81
104. Training and validation loss graph 86
105. Training loss 86
106. Validation loss 86
107. Second iteration of the proposed architecture 87
108. Training and validation loss graph 89

109. Simulator testing: Left lane behaviour 90
110. Simulator testing: Switching to right lane behaviour 90
111. Simulator testing: Returning to left lane behaviour 91
112. Real track testing: lane keeping 91

List of Tables

1. Deciding between three classifiers 74
2. F1 score 74
3. Averaged F1 score 74
4. Three classifiers compared 75
5. Minimum requirements 75

1. Appendix: Camera calibration code

The following functions are implemented to allow the automated calibration of a camera, given a number of calibration rig photographs:

1.1. getObjectAndImagePoints

```
import numpy as np
import cv2, os, glob


objectPoints = []
imagePoints = []
cameraIntrinsicValues = []
# Distortion coefficients
cameraExtrinsicValues = []
def getObjectAndImagePoints():
    global objectPoints, imagePoints

    # Number of inside corners per row and column
    cornersPerRow = 10
    cornersPerColumn = 7

    # Initializing the object points to zero
    chessboardObjectPoints = np.zeros((cornersPerColumn * cornersPerRow, 3),
    ↪  np.float32)

    # Prepare a meshgrid for object points
    # (0,0,0), (1,0,0), (2,0,0) ..., (cornersPerRow,cornersPerColumn,0)
    # This can be done since the number of corners there are on the printed
    ↪  chessboard is known in advance
    chessboardObjectPoints[:,:2] = np.mgrid[0:cornersPerRow,
    ↪  0:cornersPerColumn].T.reshape(-1, 2)

    # List of calibration images
    images = []

    # To make sure the script can be run on any image filetype
    extensions = ['*.gif', '*.png', '*.jpeg', '*.jpg', '*.tiff']
    for extension in extensions:
        images.extend(glob.glob('calibration_images/'+extension))

    # Step through the list and search for chessboard corners
    for calibrationImageFileName in images:
        calibrationImage = cv2.imread(calibrationImageFileName)

        # The detector doesn't work well with images larger than 1280x720
        # So images will be resized until they're 720p or smaller
        height, width = calibrationImage.shape[:2]
        while width > 1280:
```

```
            width //= 2
            height //= 2
            calibrationImage = cv2.resize(calibrationImage, (width, height))

        # Convert it to grayscale
        grayCalibrationImage = cv2.cvtColor(calibrationImage, cv2.COLOR_BGR2GRAY)

        # Find the image points
        cornersFound, foundCorners = cv2.findChessboardCorners(grayCalibrationImage,
        ↪  (cornersPerRow, cornersPerColumn),None)

        # If corners were found on the images
        # Append the found image points and defined object points to global
        ↪  variables
        if cornersFound:
            objectPoints.append(chessboardObjectPoints)
            imagePoints.append(foundCorners)

            # Visualize the found corners
            cv2.drawChessboardCorners(calibrationImage, (cornersPerRow,
            ↪  cornersPerColumn), foundCorners, cornersFound)
            cv2.imshow('Preview', calibrationImage)
            cv2.waitKey(500)
```

1.2. calibrateCamera

```
def calibrateCamera(imageSize):
    global cameraIntrinsicValues, cameraExtrinsicValues, objectPoints, imagePoints
    retVal, cameraIntrinsicValues, cameraExtrinsicValues, rotationVectors,
    ↪  translationVectors = cv2.calibrateCamera(objectPoints, imagePoints,
    ↪  imageSize, None, None)
```

1.3. undistortImage

```
def undistortImage(image):
    return cv2.undistort(image, cameraIntrinsicValues, cameraExtrinsicValues, None,
    ↪  cameraIntrinsicValues)
```

2. Appendix: Custom simulator code

2.1. ResolutionSetter class

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class ResolutionSetter : MonoBehaviour
{
    //Attach this script to a Dropdown GameObject
    Dropdown m_Dropdown;
    //This is the string that stores the current selection m_Text of the Dropdown
    string m_Message;
    //This Text outputs the current selection to the screen
    public Text m_Text;
    //This is the index value of the Dropdown
    int m_DropdownValue;
    void Start()
    {
        //Fetch the Dropdown GameObject
        m_Dropdown = GetComponent<Dropdown>();
        //Add listener for when the value of the Dropdown changes, to take action
        m_Dropdown.onValueChanged.AddListener(delegate {
            DropdownValueChanged(m_Dropdown);
        });
        setCameraSensorRes(m_Dropdown.options[m_DropdownValue].text);
    }

    void DropdownValueChanged(Dropdown change)
    {
       setCameraSensorRes(m_Dropdown.options[change.value].text);
    }

    void setCameraSensorRes(string resolution){
        CameraSensor.width = System.Convert.ToInt32(resolution.Split('x')[0]);
        CameraSensor.height = System.Convert.ToInt32(resolution.Split('x')[1]);
    }
}
```

3. Appendix: First custom architecture code

3.1. Model class

```
class NvidiaModel(KerasPilot):
    def __init__(self, num_outputs=2, input_shape=(120, 160, 3), roi_crop=(0, 0),
    ↪  *args, **kwargs):
        super(NvidiaModel, self).__init__(*args, **kwargs)
        self.model = customArchitecture(num_outputs, input_shape, roi_crop)
        self.compile()

    def compile(self):
        self.model.compile(optimizer="adam",
                loss='mse')

    def run(self, img_arr):
        img_arr = img_arr.reshape((1,) + img_arr.shape)
        outputs = self.model.predict(img_arr)
        steering = outputs[0]
        throttle = outputs[1]
        return steering[0][0], throttle[0][0]
```

3.2. Model implementation

```
def customArchitecture(num_outputs, input_shape, roi_crop):

    input_shape = adjust_input_shape(input_shape, roi_crop)
    img_in = Input(shape=input_shape, name='img_in')
    x = img_in

    # Dropout rate
    keep_prob = 0.9
    rate = 1 - keep_prob

    # Convolutional Layer 1
    x = Convolution2D(filters=24, kernel_size=5, strides=(2, 2), input_shape =
    ↪  input_shape)(x)
    x = Dropout(rate)(x)

    # Convolutional Layer 2
    x = Convolution2D(filters=36, kernel_size=5, strides=(2, 2),
    ↪  activation='relu')(x)
    x = Dropout(rate)(x)

    # Convolutional Layer 3
    x = Convolution2D(filters=48, kernel_size=5, strides=(2, 2),
    ↪  activation='relu')(x)
    x = Dropout(rate)(x)

    # Convolutional Layer 4
```

```
x = Convolution2D(filters=64, kernel_size=3, strides=(1, 1),
↪  activation='relu')(x)
x = Dropout(rate)(x)

# Convolutional Layer 5
x = Convolution2D(filters=64, kernel_size=3, strides=(1, 1),
↪  activation='relu')(x)
x = Dropout(rate)(x)

# Flatten Layers
x = Flatten()(x)

# Fully Connected Layer 1
x = Dense(100, activation='relu')(x)

# Fully Connected Layer 2
x = Dense(50, activation='relu')(x)

# Fully Connected Layer 3
x = Dense(25, activation='relu')(x)

# Fully Connected Layer 4
x = Dense(10, activation='relu')(x)

# Fully Connected Layer 5
x = Dense(5, activation='relu')(x)
outputs = []

for i in range(num_outputs):
    # Output layer
    outputs.append(Dense(1, activation='linear', name='n_outputs' + str(i))(x))

model = Model(inputs=[img_in], outputs=outputs)

return model
```

4. Appendix: Advanced model code

4.1. Model class

```
class OriModel(KerasPilot):
    '''
    Custom model that takes an input image and feeds it and a preprocessed version
↪   of it to the model.
    The preprocessing converts the image to HSL color space, extracts the S channel
↪   and thresholds it.
    The thresholded S channel is passed to the model to help find lane lines easier.
    '''
    def __init__(self, model=None, input_shape=(180, 320, 3), *args, **kwargs):
        super(OriModel, self).__init__(*args, **kwargs)
        self.model = oriModel(inputShape=input_shape)
        self.compile()

    def compile(self):
        self.model.compile(optimizer=self.optimizer,
                loss='mse')
    def run(self, inputImage):
            # Preprocesses the input image for easier lane detection
            extractedLaneInput = self.processImage(inputImage)
            # Reshapes to (1, height, width, channels)
            extractedLaneInput = extractedLaneInput.reshape((1,) +
            ↪  extractedLaneInput.shape)
            inputImage = inputImage.reshape((1,) + inputImage.shape)
            # Predicts the output steering and throttle
            steering, throttle = self.model.predict([inputImage,
            ↪  extractedLaneInput])
            print("Throttle: %f, Steering: %f" % (throttle[0][0], steering[0][0]))
            return steering[0][0], throttle[0][0]

    def warpImage(self, image):
        # Define the region of the image we're interested in transforming
        regionOfInterest = np.float32(
        [[0,  180],  # Bottom left
        [112.5, 87.5], # Top left
        [200, 87.5], # Top right
        [307.5, 180]]) # Bottom right

        # Define the destination coordinates for the perspective transform
        newPerspective = np.float32(
        [[80,  180],  # Bottom left
        [80,    0.25],  # Top left
        [230,   0.25],  # Top right
        [230, 180]]) # Bottom right
        # Compute the matrix that transforms the perspective
        transformMatrix = cv2.getPerspectiveTransform(regionOfInterest,
        ↪  newPerspective)
```

```
        # Warp the perspective - image.shape[:2] takes the height, width, [::-1]
        ↪  inverses it to width, height
        warpedImage = cv2.warpPerspective(image, transformMatrix,
        ↪  image.shape[:2][::-1], flags=cv2.INTER_LINEAR)
        return warpedImage

    def extractLaneLinesFromSChannel(self, warpedImage):
        # Convert to HSL
        hslImage = cv2.cvtColor(warpedImage, cv2.COLOR_BGR2HLS)
        # Split the image into three variables by the channels
        hChannel, lChannel, sChannel = cv2.split(hslImage)
        # Threshold the S channel image to select only the lines
        lowerThreshold = 65
        higherThreshold = 255
        # Threshold the image, keeping only the pixels/values that are between lower
        ↪  and higher threshold
        returnValue, binaryThresholdedImage =
        ↪  cv2.threshold(sChannel,lowerThreshold,higherThreshold,cv2.THRESH_BINARY)
        # Since this is a binary image, we'll convert it to a 3-channel image so
        ↪  OpenCV can use it
        thresholdedImage = cv2.cvtColor(binaryThresholdedImage, cv2.COLOR_GRAY2RGB)
        return thresholdedImage

    def processImage(self, image):
        warpedImage = self.warpImage(image)
        # We'll normalize it just to make sure it's between 0-255 before
        ↪  thresholding
        warpedImage =
        ↪  cv2.normalize(warpedImage,None,0,255,cv2.NORM_MINMAX,cv2.CV_8U)
        thresholdedImage = self.extractLaneLinesFromSChannel(warpedImage)
        one_byte_scale = 1.0 / 255.0
        # To make sure it's between 0-1 for the model
        return np.array(thresholdedImage).astype(np.float32) * one_byte_scale
```

4.2. Model implementation

```
def oriModel(inputShape):

    # Dropout rate
    keep_prob = 0.9
    rate = 1 - keep_prob

    # Input layers
    imageInput = Input(shape=inputShape, name='imageInput')
    laneInput = Input(shape=inputShape, name='laneInput')

    # Input image convnet
    x = imageInput
    x = Conv2D(24, (5,5), strides=(2,2), name="Conv2D_imageInput_1")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(32, (5,5), strides=(2,2), name="Conv2D_imageInput_2")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(64, (5,5), strides=(2,2), name="Conv2D_imageInput_3")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_imageInput_4")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_imageInput_5")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Flatten(name="flattenedx")(x)
    x = Dense(100)(x)
    x = Dropout(rate)(x)
    # Preprocessed lane image input convnet
    y = laneInput
    y = Conv2D(24, (5,5), strides=(2,2), name="Conv2D_laneInput_1")(y)
    y = LeakyReLU()(y)
    y = Dropout(rate)(y)
    y = Conv2D(32, (5,5), strides=(2,2), name="Conv2D_laneInput_2")(y)
    y = LeakyReLU()(y)
    y = Dropout(rate)(y)
    y = Conv2D(64, (5,5), strides=(2,2), name="Conv2D_laneInput_3")(y)
    y = LeakyReLU()(y)
    y = Dropout(rate)(y)
    y = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_laneInput_4")(y)
    y = LeakyReLU()(y)
    y = Dropout(rate)(y)
    y = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_laneInput_5")(y)
    y = LeakyReLU()(y)
    y = Flatten(name="flattenedy")(y)
    y = Dense(100)(y)
    y = Dropout(rate)(y)
    # Concatenated final convnet
```

```
c = Concatenate(axis=1)([x, y])
c = Dense(100, activation='relu')(c)
c = Dense(50, activation='relu')(c)
# Output layers
steering_out = Dense(1, activation='linear', name='steering_out')(o)
throttle_out = Dense(1, activation='linear', name='throttle_out')(o)
model = Model(inputs=[imageInput, laneInput], outputs=[steering_out,
↪  throttle_out])

return model
```

5. Appendix: Advanced behavioural model code

5.1. Model class

```
class OriModel(KerasPilot):
    '''
    Custom model that takes an input image and feeds it and a preprocessed version
↪   of it to the model.
    The preprocessing converts the image to HSL color space, extracts the S channel
↪   and thresholds it.
    The thresholded S channel is passed to the model to help find lane lines easier.
    '''
    def __init__(self, model=None, input_shape=(180, 320, 3), num_behavior_inputs=2,
    ↪   *args, **kwargs):
        super(OriModel, self).__init__(*args, **kwargs)
        self.model = oriModel(inputShape=input_shape, numberOfBehaviourInputs =
        ↪   num_behavior_inputs)
        self.compile()

    def compile(self):
        self.model.compile(optimizer=self.optimizer,
                loss='mse')

    def run(self, inputImage, behaviourArray):
        # Preprocesses the input image for easier lane detection
        extractedLaneInput = self.processImage(inputImage)
        # Reshapes to (1, height, width, channels)
        extractedLaneInput = extractedLaneInput.reshape((1,) +
        ↪   extractedLaneInput.shape)
        inputImage = inputImage.reshape((1,) + inputImage.shape)

        behaviourArray = np.array(behaviourArray).reshape(1,len(behaviourArray))
        # Predicts the output steering and throttle
        steering, throttle = self.model.predict([inputImage, extractedLaneInput,
        ↪   behaviourArray])
        #print("Throttle: %f, Steering: %f" % (throttle[0][0], steering[0][0]))
        return steering[0][0], throttle[0][0]

    def warpImage(self, image):
        # Define the region of the image we're interested in transforming
        regionOfInterest = np.float32(
                        [[0,   180],  # Bottom left
                        [112.5, 87.5], # Top left
                        [200, 87.5], # Top right
                        [307.5, 180]]) # Bottom right

        # Define the destination coordinates for the perspective transform
        newPerspective = np.float32(
                        [[80,  180],  # Bottom left
                        [80,    0.25],  # Top left
                        [230,   0.25],  # Top right
```

```
                    [230, 180]]) # Bottom right
        # Compute the matrix that transforms the perspective
        transformMatrix = cv2.getPerspectiveTransform(regionOfInterest,
        ↪  newPerspective)
        # Warp the perspective - image.shape[:2] takes the height, width, [::-1]
        ↪  inverses it to width, height
        warpedImage = cv2.warpPerspective(image, transformMatrix,
        ↪  image.shape[:2][::-1], flags=cv2.INTER_LINEAR)
        return warpedImage

    def extractLaneLinesFromSChannel(self, warpedImage):
        # Convert to HSL
        hslImage = cv2.cvtColor(warpedImage, cv2.COLOR_BGR2HLS)
        # Split the image into three variables by the channels
        hChannel, lChannel, sChannel = cv2.split(hslImage)
        # Threshold the S channel image to select only the lines
        lowerThreshold = 65
        higherThreshold = 255
        # Threshold the image, keeping only the pixels/values that are between lower
        ↪  and higher threshold
        returnValue, binaryThresholdedImage =
        ↪  cv2.threshold(sChannel,lowerThreshold,higherThreshold,cv2.THRESH_BINARY)
        # Since this is a binary image, we'll convert it to a 3-channel image so
        ↪  OpenCV can use it
        thresholdedImage = cv2.cvtColor(binaryThresholdedImage, cv2.COLOR_GRAY2RGB)
        return thresholdedImage

    def processImage(self, image):
        one_byte_scale = 1.0 / 255.0
        warpedImage = self.warpImage(image)
        warpedImage =
        ↪  cv2.normalize(warpedImage,None,0,255,cv2.NORM_MINMAX,cv2.CV_8U)
        thresholdedImage = self.extractLaneLinesFromSChannel(warpedImage)
        return np.array(thresholdedImage).astype(np.float32) * one_byte_scale
```

5.2. Model implementation

```
def oriModel(inputShape, numberOfBehaviourInputs):

    # Dropout rate
    keep_prob = 0.9
    rate = 1 - keep_prob

    # Input layers
    imageInput = Input(shape=inputShape, name='imageInput')
    laneInput = Input(shape=inputShape, name='laneInput')
    behaviourInput = Input(shape=(numberOfBehaviourInputs,), name="behaviourInput")

    # Input image convnet
    x = imageInput
    x = Conv2D(24, (5,5), strides=(2,2), name="Conv2D_imageInput_1")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(32, (5,5), strides=(2,2), name="Conv2D_imageInput_2")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(64, (5,5), strides=(2,2), name="Conv2D_imageInput_3")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_imageInput_4")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_imageInput_5")(x)
    x = LeakyReLU()(x)
    x = Dropout(rate)(x)
    x = Flatten(name="flattenedx")(x)
    x = Dense(100)(x)
    x = Dropout(rate)(x)

    # Preprocessed lane image input convnet
    y = laneInput
    y = Conv2D(24, (5,5), strides=(2,2), name="Conv2D_laneInput_1")(y)
    y = LeakyReLU()(y)
    y = Dropout(rate)(y)
    y = Conv2D(32, (5,5), strides=(2,2), name="Conv2D_laneInput_2")(y)
    y = LeakyReLU()(y)
    y = Dropout(rate)(y)
    y = Conv2D(64, (5,5), strides=(2,2), name="Conv2D_laneInput_3")(y)
    y = LeakyReLU()(y)
    y = Dropout(rate)(y)
    y = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_laneInput_4")(y)
    y = LeakyReLU()(y)
    y = Dropout(rate)(y)
    y = Conv2D(64, (3,3), strides=(1,1), name="Conv2D_laneInput_5")(y)
    y = LeakyReLU()(y)
    y = Flatten(name="flattenedy")(y)
    y = Dense(100)(y)
```

```
y = Dropout(rate)(y)

# Behavioural net
z = behaviourInput
z = Dense(numberOfBehaviourInputs * 2, activation='relu')(z)
z = Dense(numberOfBehaviourInputs * 2, activation='relu')(z)
z = Dense(numberOfBehaviourInputs * 2, activation='relu')(z)

# Concatenated final convnet
c = Concatenate(axis=1)([x, y])
c = Dense(100, activation='relu')(c)

o = Concatenate(axis=1)([z, c])
o = Dense(100, activation='relu')(o)
o = Dense(50, activation='relu')(o)

# Output layers
steering_out = Dense(1, activation='linear', name='steering_out')(o)
throttle_out = Dense(1, activation='linear', name='throttle_out')(o)
model = Model(inputs=[imageInput, laneInput, behaviourInput],
↪  outputs=[steering_out, throttle_out])

return model
```