Drift Racing with Machine Learning

Submitted by

Tan Wei Jie

In partial fulfilment of the

requirements for the Degree of

Bachelor of Engineering (Computer Engineering)

National University of Singapore

B.Eng. Dissertation

Drift Racing with Machine Learning

By

Tan Wei Jie

National University of Singapore

2019/2020

Project ID: H041930

Project Supervisor: Assoc Prof Martin Henz

Deliverables:

Report: 1 Volume

Abstract

Drifting is a driving technique where drivers intentionally oversteer while navigating a bend. It also occurs under extreme conditions such as icy roads and after a collision, and therefore, the ability of autonomous vehicles to execute controlled manoeuvres while drifting can be critical to its safety. This report presents a novel solution for autonomous drifting of a $1/10^{th}$ sized remote-controlled car, using a single camera, a convolutional neural network and hardware popular among Maker culture such as the Raspberry Pi. This is done as part of a larger project for the design and implementation of an autonomous robot race-car competition for students of NUS School of Computing and Faculty of Engineering, to tap onto the current trend of Autonomous Robots, Self-driving Cars and Deep Learning.

Keywords:

Machine Learning, Deep Learning, Convolutional Neural Network, Computer Vision, Robotics, Autonomous Robots, Robot Operating System, Drifting, Experiential Learning

Implementation Hardware and Software:

Raspberry Pi v4 4GB, Raspbian Buster, Robot Operating System Melodic Morenia, Google Colaboratory, TensorFlow v1.15, TensorFlow Lite

Acknowledgement

The team wishes to thank Associate Professor Hugh Anderson and Lecturer Boyd Anderson for their advice over the course of the project, as well as Assistant Professor Brian Low for allowing us to run this competition as part of the CS3244 Machine Learning module project. The team would also like to thank members of team TESLAH, team Déjà vu, and team Learning Machine for participating in the competition as part of their coursework for the CS3244 Machine Learning module. Last but not least, a big thanks to the staff at SOC Makers Lab, in particular Kenny, for their hardware-related assistance.

List of Figures

List of Tables

Abbreviations

Abbreviation	Meaning
DQN	Deep Q Network
GPU	Graphics Processing Unit
IMU	Inertial Measurement Unit
PWM	Pulse Width Modulation
RC (car)	Remote-controlled (car)
RL	Reinforcement Learning
ROS	Robot Operating System
TCP	Transmission Control Protocol

Table of Contents

1 Introduction

1.1 Background and Motivation

With the current trend and interest in Autonomous Robots, Self-driving Cars, and Deep Learning, an autonomous race-car competition was proposed for students of NUS School of Computing and Faculty of Engineering, in which students compete by programming their cars to be the fastest in an autonomous race. The aim of the competition is to provide an experiential learning platform for students to try their hands at robotics and Deep Learning, while providing a gateway for them to gain further interest and experience in these subjects. To make this competition more exciting and distinct from similar autonomous race-car competitions, a "drifting" aspect is to be incorporated to the competition.

Drifting is the result of the loss of traction between the wheels of a car and the ground. Sometimes, it is used as a driving technique, where drivers intentionally oversteer the car while manoeuvring a turn or a bend. Figure 1 below illustrates of the difference between normal driving and drifting while navigating a bend.

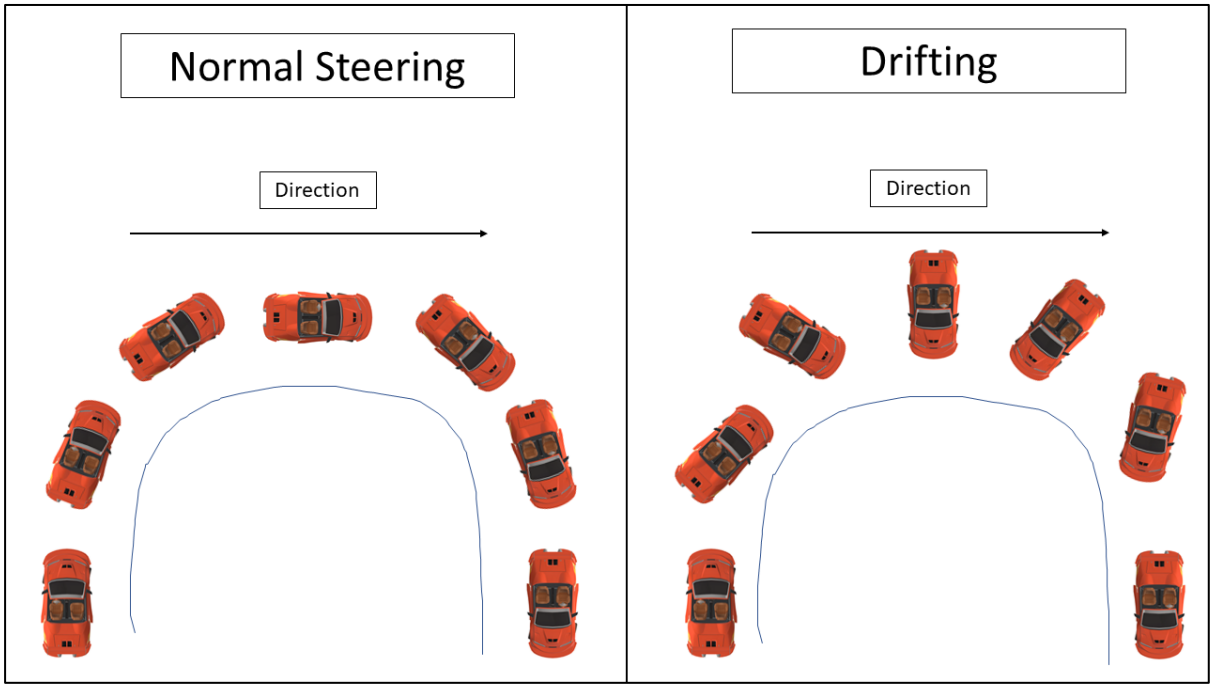

Figure 1: Illustration of Difference Between Normal Steering and Drifting About a Bend

Other times, it may not be intentional, such as when the floor is icy, or if the car suffers a collision. Studying drifting can thus provide some insight into improving the safety of autonomous vehicles.

1.2 Scope of Report

This project is carried out in a team of 3 students, including myself and 2 others: Henry Ang and Melvin Leo. As a result, the entire report on the project is split up into 3 volumes, each covering the scope of our own individual contributions to the project.

While each report can be read as a standalone, it is recommended to read all 3 reports for a better overall picture of the project. The focus of this report is on the software implementation of a novel solution to programming a drifting controller, as well as the experiments done to demonstrate feasibility.

1.3 Deliverables

The key deliverables for this project are (1) the design and implementation of an autonomous robotics racing car competition, and (2) an implementation of a "contestant" to demonstrate the feasibility of our proposed competition.

For (1), the sub-deliverables include: design and fabrication of a racetrack for the competition, a list of stock hardware to be distributed to contestants, race format and competition rules. For (2), a "contestant" has to be implemented and demonstrate drifting around our proposed track, with video evidence for proof-of-concept and publicity purposes.

Mainly, this report covers the software implementation of deliverable (2). In addition, the idea behind the design of the racetrack can also be seen from the experiments conducted. Finally, the competition rules are also briefly covered in this report. The other deliverables are covered in the scope of the reports by Melvin and Henry.

1.4 Outline of Report

This report can be split into 3 parts. The first part is on the literature review and proposed solutions, as covered in Chapters 2, 3, and 4. By the end of Chapter 4, the final proposed solution is chosen, with justifications, for testing and implementation in our experiments to demonstrate feasibility.

The second part details the implementation of the robot car as seen in Chapter 5, as well as the experiments conducted to demonstrate proof-of-concept and feasibility, seen in Chapter 6.

The third part of this report discusses the implementation of the competition, detailed in Chapter 7. Finally, the report is summarized in Chapter 8, along with a discussion on the limitations of the work done and possible further work.

This report also contains a number of QR codes that point towards resources, such as videos, that are either pertinent to the discussion, or supplementary material. Readers are encouraged to visit these links by scanning the QR code using their smartphone cameras, especially material that is pertinent to the discussion.

2 Proposed Solution I: Sustained Drift based on Reinforcement Learning

2.1 Literature Review 1: Autonomous Drifting using Simulation Aided Reinforcement Learning

The original idea that inspired this project came from a particular YouTube video, where Mark Cutler and Jonathan P. How (2016) demonstrate the results of their reinforcement learning framework by teaching a remote-controlled car how to maintain a sustained drift. To achieve this, they employed Probabilistic Inference for Learning Control (PILCO), which is a reinforcement learning algorithm to learn an optimal drift controller to control the RC car.

$$S_t = \left[V_x, V_y, \dot{\theta}, \omega\right] \quad (1)$$

To solve a problem with reinforcement learning, typically, the programmer has to define certain aspects of the problem setup, such as the agent, the environment, the reward function, and importantly in our case, what constitutes the Markovian state information. Cutler and How (2016) describe that the Markovian state information in their experiment consists of 4 main components as shown in Equation (1) above: the velocities Vx and Vy in the car frame, the turn rate, and the current wheel speed of the car respectively. They measure the velocities in the car frame with an external motion capture system, and the turn rate and wheel speed with onboard sensors (likely to be wheel encoders and/or an Inertial Measurement Unit (IMU) that is typical for wheeled robot odometry measurements).

Supplementary Material 1:

Original Inspiration for this Project.

Link: https://www.youtube.com/watch?v=opsmd5yuBF0

2.2 Literature Review 2: Autonomous Drifting RC Car with Reinforcement Learning

An experiment similar to the that in Cutler and How (2016) was later implemented by Bhattacharjee, Kabara, and Jain (2018), as detailed in their report(s), which was split into 3 volumes: the Reinforcement Learning (Algorithm) report, the Hardware report, and the Software report. In their experiments, they attempted to implement the drift controller with a more common reinforcement learning algorithm, the Deep Q-Network (DQN), which is a deep learning variation of the Q-Learning algorithm. In addition, they also employed some of the improvements to the vanilla DQN, such as the Double Q-Learning and Dueling Network Architecture. The DQN algorithm, when implemented with these 2 improvements, is often referred to as the Double Dueling DQN. In addition to the DQN algorithm, they also experimented with the PILCO algorithm, and made comparisons between the performance of these 2 algorithms.

$$S_t = \left[x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}\right] \tag{2}$$

Importantly, Bhattacharjee et al (2018) initially experimented with formalizing the reinforcement learning Markovian state information as shown in Equation 2 above: “the x-coordinate, y-coordinate, angular orientation, x-velocity, y-velocity and angular velocity with respect to the world reference frame". Additionally, their reward function emphasizes on maintaining a constant radius with respect to an arbitrary point, ideally resulting in a circular drift around this point. However, they were not successful in this endeavour in obtaining an optimal drift controller, and instead, found more success with a Markovian state encoding similar to that in Cutler and How’s (2016) experiment, without the current wheel speed of the car**.** This is shown in Equation (3) below.

$$S_t = \left[V_x, V_y, \dot{\theta}\right] \tag{3}$$

From these 2 reports in Chapters 2.1 and 2.2, an initial solution with the use of Reinforcement Learning was proposed, as detailed in the next chapter.

2.3 Proposed Reinforcement Learning Approach

For the reinforcement learning approach, the initial plan was to build on the experiments by Bhattacharjee et al (2018), through deploying further improvements to the DQN architecture and using a similar Markovian state encoding as described in Equation (3). Since we did not have the resources to a motion capture system used in Cutler and How (2016), we hoped to achieve similar performance in data collection with manipulation of an Inertial Measurement Unit (IMU). The initial plan was thus as follows:

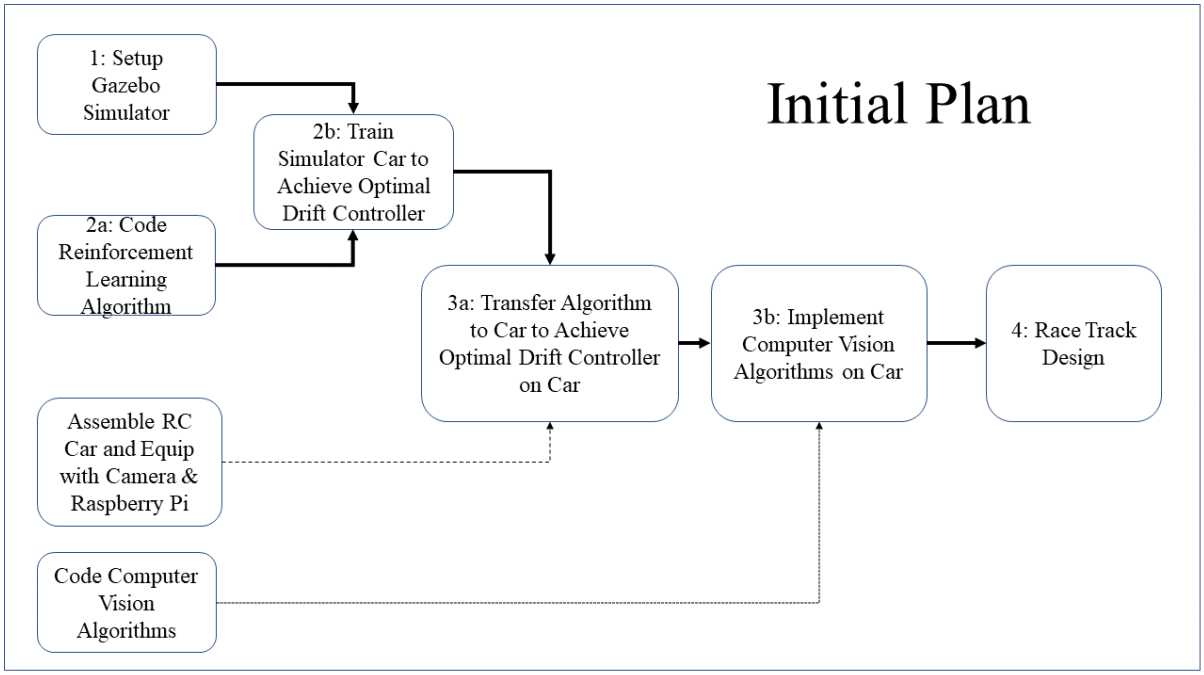

Figure 2: Outline of Initial Plan based on Proposed Reinforcement Learning Solution

To compensate for the foreseen reduced accuracy of the setup, more improvements would be employed to the DQN model that Bhattacharjee et al (2016) used. It seemed that many of the improvements to the original DQN model are independent of one another, and therefore could be implemented together. Indeed, there exists an algorithm known as RAINBOW, as reported by Hessel et al (2018), which combines six of these independent improvements. Hessel et al (2018) reports that the RAINBOW algorithm has demonstrated state-of-the-art performance on the Atari 2600 benchmark.

2.4 Feasibility and Limitations

Unfortunately, even with the idea to improve on the deep learning model, the original assumption of using an IMU to estimate the information necessary for the algorithm was a naïve assumption, and after a consultation with Professor Hugh Anderson, it was concluded that more sophisticated sensors and algorithms were needed to obtain the state information necessary for the algorithm to work well, as the IMU readings would be far too noisy, from his experience.

It was then further concluded[1] that acquiring a setup that can produce state information that was adequately clean for our use was a non-trivial task and would probably **not be feasible** given our budget and resources. Therefore, the focus shifted towards searching for another method of programming a drift controller.

[1] Appendix A – Comparison of Common Autonomous Robot Sensors for Obtaining Odometry

3 Proposed Solution II: Naïve Drifting with Convolutional Neural Network

As elaborated on in Chapter 2, the initial proposed method of using Reinforcement Learning to do drifting was not feasible given the hardware and resource limitations. Further literature review was done in the hopes of coming up with an alternate approach to achieve a drift controller, that satisfies the hardware and resource limitations.

3.1 Literature Review 3: Stanley: The robot that won the DARPA Grand Challenge

Thrun el al (2006) describes the implementation of an autonomous car, Stanley, used for the 2005 DARPA Grand Challenge. The content most pertinent for this discussion is Section 9 of Thrun el al (2006), on Real-Time Control of the car. Thrun et al (2006) describes that the controller for Stanley is split into a velocity controller and a steering controller. The steering controller predicts steering angle based on Euclidean distance to the nearest path segment, trajectory error and current speed of the body (note this is not the desired speed), while the velocity controller computes speed as a minimum of 3 recommendations not related to the desired steering angle. From this, it can be seen that the controllers are able to derive their respective values for velocity and steering **independently**.

This leads to an insight that for controlling a car, the control command can be decomposed into independently controlling the steering of the car and the throttle of the car. This idea is considered for coming up with a way to program a drift controller for the car.

3.2 Literature Review 4: End to End Learning for Self-Driving Cars

Nvidia's report "End to End Learning for Self-Driving Cars" by Bojarski et al (2016) describes the use of a Convolutional Neural Network (CNN) to map the steering angle of a life-sized car to achieve autonomous self-driving. To do this, they took 72 hours of image data from a camera attached to a car during normal driving and recorded the steering angle. This set of data (of images and matching steering angle) was then used to train the neural network. During test time, real-time image data from the camera of the car is fed to the CNN, which outputs a desired steering angle for the car to achieve. This algorithm is sometimes

also known as “Behavioural Cloning” and is a form of supervised learning rather than reinforcement learning.

Although this algorithm was **not** intended for drifting, we hoped that it can generalize to drifting controls. In fact, an advantage that this has over the reinforcement learning approach is that this algorithm outputs a continuous value steering angle, while the RL algorithms such as DQN typically output a discrete action from a predefined set of possible actions in the action space.

3.3 Proposed Deep Learning & Computer Vision Approach

By combining the insight from Thrun et al (2006) and the idea from Bojarski et al (2016), the following idea for drifting was proposed:

We would send control commands to the car that consists of:

1. Desired steering angle as predicted by a trained convolutional neural network, that we hope can generalize to drifting controls
2. Set a constant throttle, exact value to be determined over the course of the experiments

The convolutional neural network will be trained using input images captured by an onboard camera on the race-car, and the labels for these input images will be the corresponding steering controls.

For this proposed solution, note that we are computing values for desired steering angle and throttle independently of each other, which is an important point of evaluation in later chapters.

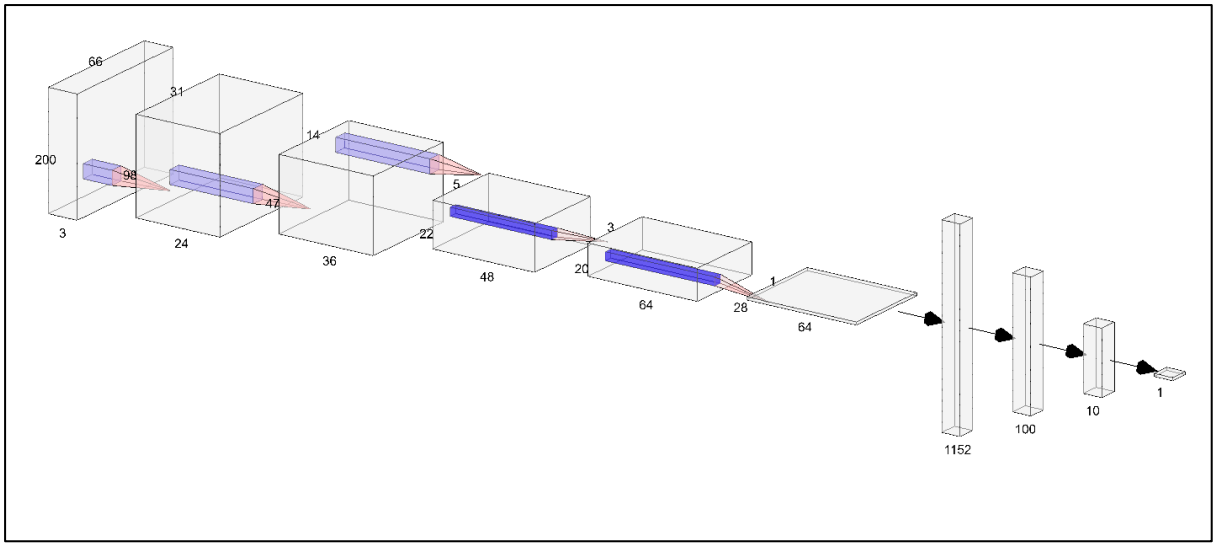

Figure 3: Convolutional Neural Network Architecture for Proposed Solution 2

(Drawn using http://alexlenail.me/NN-SVG/)

The neural network architecture used in this approach will the same as that described in Bojarski et al (2016). An illustration of this is shown in Figure 3 above. Note that the normalization layer, for mapping pixels values of the input image from [0, 255] to [-1, 1], is omitted in this illustration. The single output at the end of the network corresponds to desired steering angle.

The key benefit of this approach is that the only sensor needed is a camera, which is both cheap and easy to mount onto a remote-controlled car, thus mitigating the problems faced for the proposed reinforcement learning solution.

3.4 Implementation on Speed Dreams Simulator

When this solution was proposed, some time had to be allocated for procuring and preparing the necessary hardware. A decision was made to try this solution in a virtual simulator, in an attempt to draw some insights that may be of use for future experiments. After researching a number of simulators, we attempted this in the open source racing simulator, Speed Dreams 2. A brief overview of the, unfortunately, short-lived experiments on the simulator is detailed in this and the following chapter.

Using Speed Dreams, we attempt to train a neural network such that it is able to control the car in the game to perform drifting. To do this, we first have to obtain training data. This is

done by playing the game in first-person perspective and manually drifting the car within the game ourselves, while capturing both our controls and the first-person perspective image displayed by the game. This data is then used to train a neural network. The resulting neural network model is then used to play the game.

Figure 4: Sample from Speed Dreams Experiments

3.5 Results and Limitations

Unfortunately, the experiments did not demonstrate any reliable drifting that can be considered adequate to demonstrate feasibility of the solution. Intuitively, it felt like the speed had to be controlled in order to achieve drifting around the bends in the game, instead of attempting to set a constant speed. This, however, would contradict the insight we obtained from Thrun et al (2006).

Nevertheless, this exercise seemed to demonstrate that the convolutional neural network can recognize, to some extent, when, where, and how to turn. As a result, we foresee potential in using coloured tape to demarcate the track margins for the track design of our future experiments and the competition.

4 Proposed Solution III: Improved Drifting with Convolutional Neural Network

From the experiments in the simulator done in the previous chapter, it was concluded that the proposed approach based on Thrun et al (2006) and Bojarski et al (2016) was inadequate for the problem of drifting. Although it demonstrated promise in ability to recognize when to turn and in which direction to turn to, it intuitively felt like the speed had to be controlled to achieve drifting around the bends. Since this would contradict the insight drawn from Thrun et al (2006), more research had to be done to evaluate this insight.

4.1 Literature Review 5: Toward Automated Vehicle Control Beyond the Stability Limits: Drifting Along a General Path

In late 2019, a team at Stanford's Dynamic Design Lab succeeded in programming a life-sized DeLorean car to drift autonomously. In the discussion of the implementation of the drift controller to achieve such a feat, Goh, Goel and Gerdes (2020) discuss an insight about the relationship between the actuators of a car and the dynamics of the problem. They assert, multiple times throughout the report, that (a) during normal driving, the steering angle of the car is used for path tracking and the throttle is used to control the velocity of the car, but (b), when the car is drifting, the throttle and steering work in coordination to achieve the drifting effect.

(a) is similar to the insight from Thrun et al (2006), but Goh et al (2020) qualifies it for normal driving instead of controlling a car in general. However, (b) clearly implies that decomposing the control command for drifting was naïve. In order to drift, both throttle and steering values need to be computed together.

Using this newfound insight by Goh et al (2020), an improved solution was proposed.

4.2 Improved Deep Learning & Computer Vision Approach

The previous solution was to map image data to steering angle, and using a convolutional neural network, estimate a function to predict the desired steering angle based on the real time image feed.

With the insight from the literature review of Goh et al (2020), a solution was proposed to not only map just the steering angle, but map both the steering angle and the throttle of the car.

The new solution is described as follows.

We would send control commands to the car that consists of: 1. Desired steering angle, and 2. Throttle controls Both of which are predicted together, as a single tuple, by a convolutional neural network using the same instance of image data. The convolutional neural network is trained in a similar way as before, with image data as input, but labelled with the corresponding steering and throttle controls.

Figure 5 below shows the new neural network for Proposed Solution 3, which is largely identical to that in Figure 3, except that the final output has 2 values instead of 1, this being the desired steering angle and the throttle values.

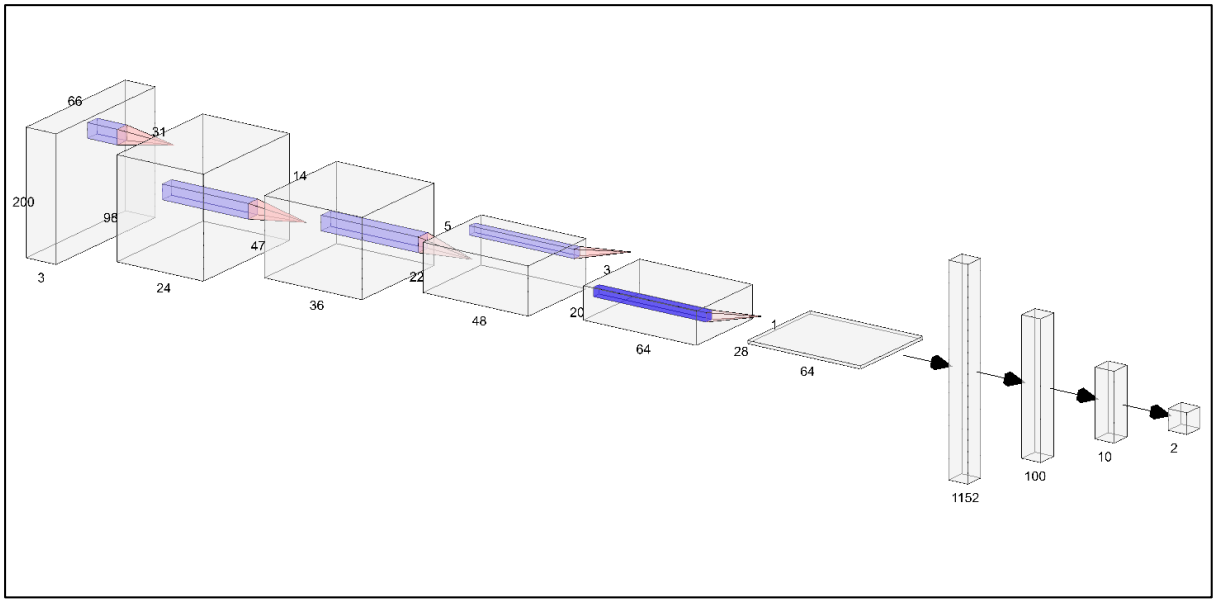

Figure 5: Convolutional Neural Network Architecture for Proposed Solution 3.

(Drawn using http://alexlenail.me/NN-SVG/)

This was an approach that had not been taken before for drifting and was deemed feasible and worth trying out on the physical RC car, which by the time this new approach was proposed, was fully procured and assembled.

5 Software Implementation for Car

5.1 Hardware Overview

The following shows an overview of the hardware used in the experiments. Only the hardware that is pertinent to the discussion and understanding of the software architecture are discussed here, using Figure 6 and the accompanying Table 1. Further details such as the full list of hardware, as well as the justification for certain choices of hardware, are not in the scope of this report, but can be found in Henry Ang's report on the hardware.

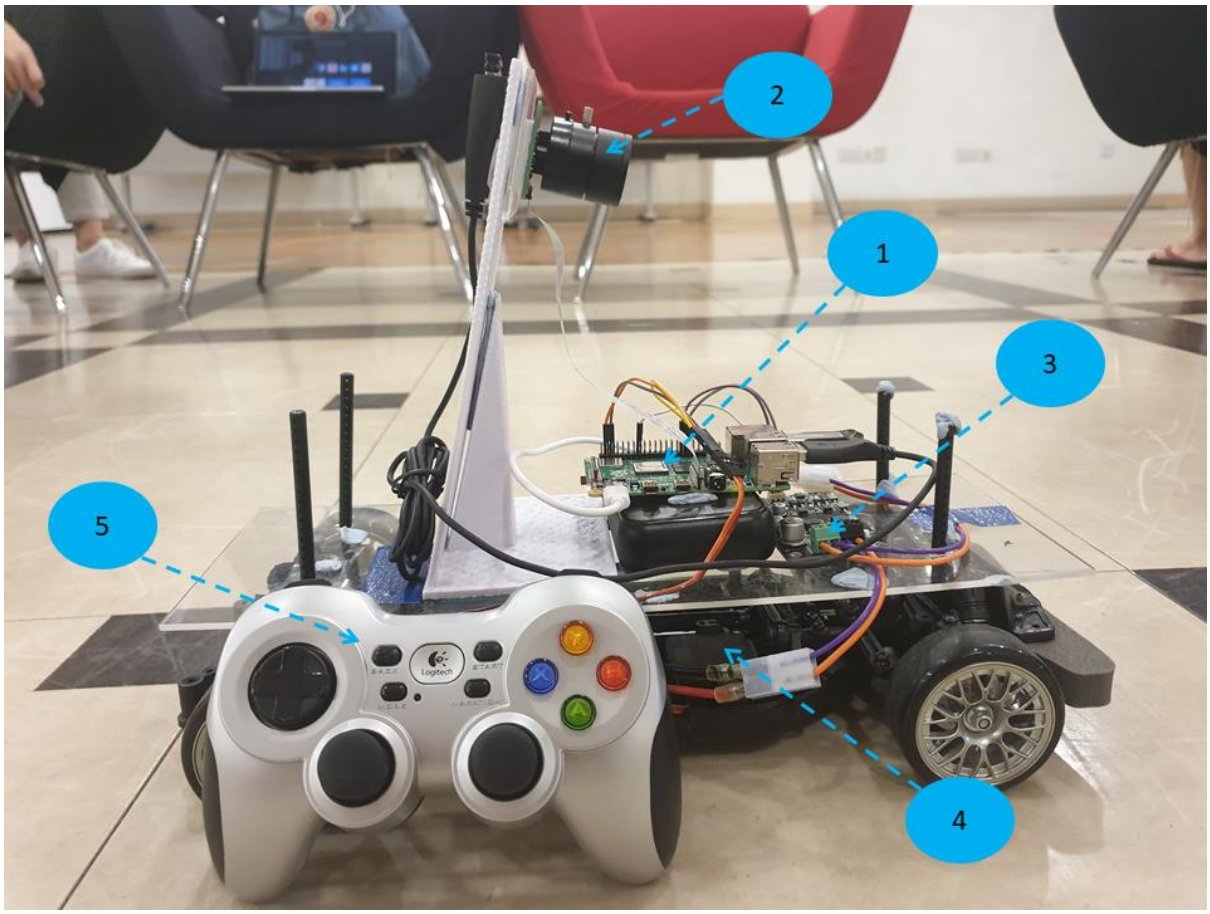

Figure 6: Overview of Programmable Hardware

Label	Name	Description
1	Raspberry Pi v4 4GB	The main brain of the system, where all the onboard computation is done.
2	Raspberry Pi Camera	The main sensor of the system. Used to capture images for the proposed solution described prior.
3	H-bridge	First of the two **actuators** of the system, that is used to control the motor of the car. Programmed using a PWM signal, from a range of 0 to 100. A higher PWM results in higher speed.
4	Servo	The second **actuator** of the system, that is used to control the steering of the car. Programmed with a "desired steering angle", from a continuous range of -90 degrees to 90 degrees.
5	Logitech F710 Wireless Joystick Controller	Used to control the car manually.

Table 1: Description of Programmable Hardware

5.2 The Robot Operating System

For this software implementation, the Robot Operating System (ROS) is used as middleware. The rest of this chapter assumes that the reader has at least a beginner level knowledge of how ROS works, as defined by the official ROS tutorials.

However, to summarize, the key characteristics of ROS are:

- **Nodes**: pieces of code programmed to perform certain tasks, each node can contain any number of *publishers* and *subscribers*.
- **Topics**: "pipes" that are used to send *messages* between nodes, each *topic* can only transfer *messages* of the same *message type*. Topic names are usually denoted with a forward slash, for example "/topic_name", "/joy".
- **Messages**: information transferred between nodes on *topics*.
- **Publishers**: used by a *node* to publish *messages* onto a *topic*.
- **Subscribers**: used by a *node* to receive *messages* from a *topic*.

The entirety of the onboard software is implemented on the Raspberry Pi v4. To aid in explaining the design of the system, 3 iterations of the ROS architecture are shown below, each iteration serving a different purpose.

5.3 Drive-by-Wire

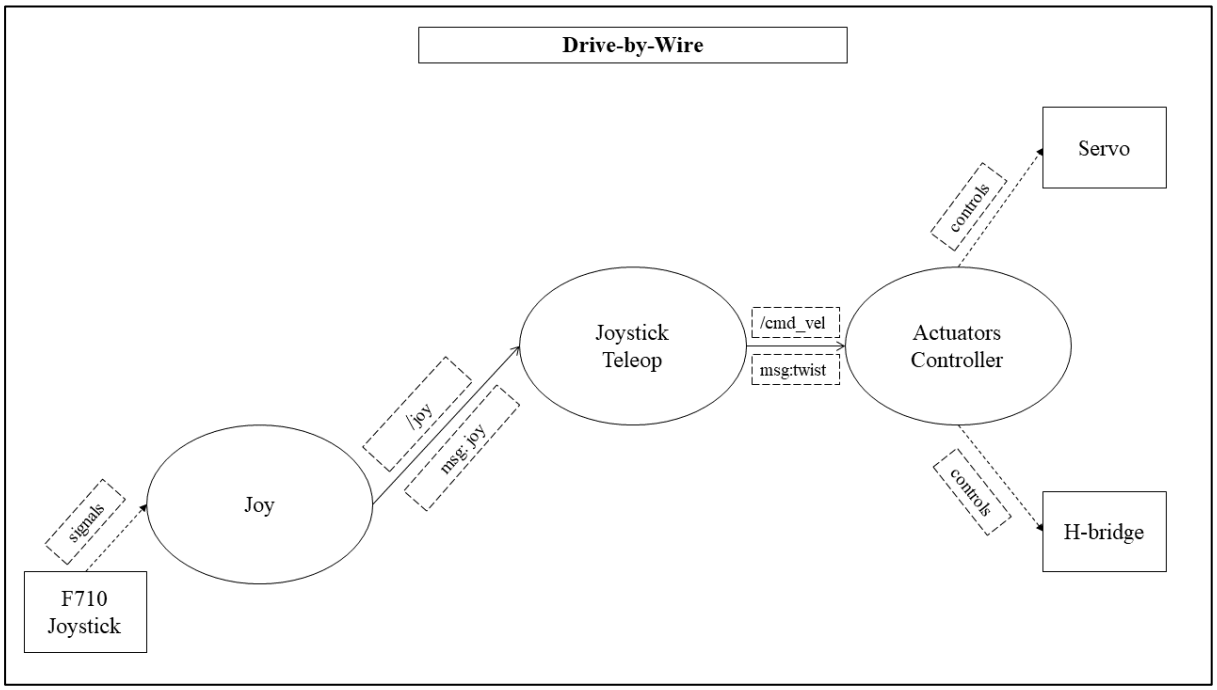

Figure 7: Drive-by-Wire Software Architecture

Figure 7 above shows an abstract view of the ROS architecture necessary for the RC car to be in drive-by-wire condition. In the case of our robot, drive-by-wire refers to the state when the robot is able to be fully controlled using our wireless Logitech F710 joystick controller. This architecture consists of 3 nodes: *joy*, *joystick_teleop*, and *actuators_controller*.

The *joy* node is cloned from the official ROS page. It is used to interface the Logitech F710 Joystick Controller with our system. It receives the signals from the controller as input, and outputs a message of type *sensor_msgs/Joy*[2] onto the topic */joy*[3]. This message contains information about which button is pressed and how much each axis on the joystick is pushed.

The *joystick_teleop* node has 2 main responsibilities. Its primary responsibility is to receive the output of the *joy* node through subscribing to the */joy* topic and interpret these messages. The buttons on the joystick correspond to a change in state of the car, while the joystick axes correspond to manual driving commands. The mapping of the controls are described in the Figure 8 and the accompanying Table 2 below.

[2] http://docs.ros.org/melodic/api/sensor_msgs/html/msg/Joy.html
[3] It is both unfortunate yet understandable that the official joy package creates a node called “joy”, and outputs a message of type “Joy” onto a topic named “/joy”.

Figure 8: Logitech F710 Wireless Gamepad with Labels

Label	Name	Meaning
1	A button	Switch to **Autonomous** Mode
2	B button	Switch to **Brake** mode
3	X button	Switch to **Manual** mode
4	Left Joystick Axis	Move horizontally to control the steering
5	Right Joystick Axis	Move vertically to control the throttle

Table 2: Control Mapping for Wireless Controller

From the table above, it can be seen that the robot has 3 different states / modes: **Autonomous, Manual, and Brake**. This state is saved as a variable within the *joystick_teleop* node, and changed whenever one of the buttons A, B, or X are pressed. This relates to the secondary responsibility of joystick_teleop, which is to act as the arbitrator between the output of the autonomous navigation node (details later) and the manual controls from the joystick.

Finally, the *joystick_teleop* node outputs the necessary command velocity information on the topic */cmd_vel* as messages of type *nav_msgs/Twist*. This message contains information on both how much to steer and how hard to use the throttle of the car.

Last but not least, the *actuators_controller* node will interpret the output of *joystick_teleop*, and using the command velocity, control the actuators of the car as necessary. Both actuators

are controlled with the help of external libraries. The servo library accepts a floating-point angle between -90 and 90 degrees, while the library to control the H-bridge accepts a value between 0 and 100 corresponding to the desired PWM signal to be sent to the H-bridge.

Additionally, it can also be noted that the joystick controller acts also as our emergency stop switch. By pressing the B button on the joystick, a signal is sent to the system to stop the car. From our experiments, it is seen that the car can stop, from top speed, in less than a second.

5.4 Training Data Collector

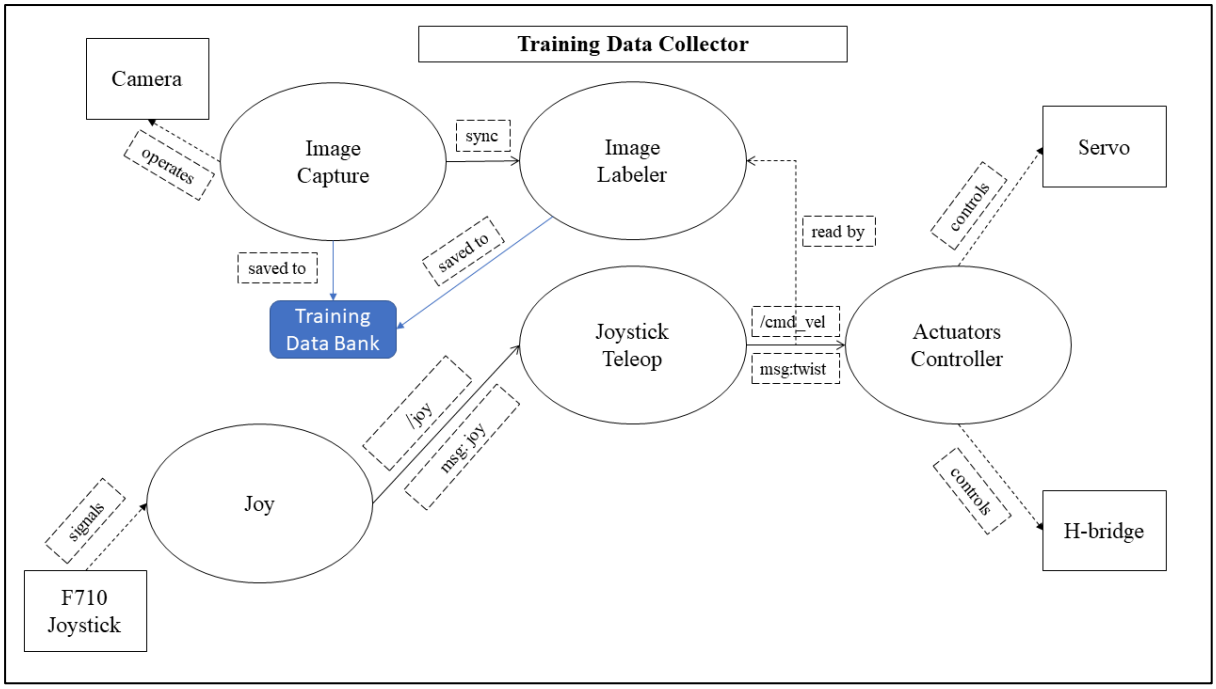

Figure 9: Training Data Collector Software Architecture

Figure 9 above shows an abstract view of the nodes needed to collect training data from the RC car, which is necessary to train our neural network. It builds on the Drive-by-Wire version of the system, as seen in Figure 7, by adding 2 more nodes: *image_capture* and *image_labeler*. This iteration of the system is used to collect training data for training a neural network.

The *image_capture* node simply operates the Raspberry Pi Camera to take pictures as quickly as possible from an image stream. Each image captured is saved to a data bank on an external memory drive, to be used as a dataset for machine learning training later. With every image taken, the node sends an arbitrary signal to sync with the *image_labeler* node.

The *image_labeler* node subscribes to the control command topic, */cmd_vel*, to record the latest command. It also subscribes to the signal sent out from the *image_capture* node. Whenever it receives such a signal, it will save the corresponding command velocity and message into a CSV file, which is also stored in the data bank on the external memory drive. An example of the final CSV file is shown below in Figure 10.

	A	B	C
1	image_path	steering_angle	speed
2	0.jpeg	0	0.18040134
3	1.jpeg	0	0.18040134
4	2.jpeg	0	0.255284131
5	3.jpeg	0	0.255284131
6	4.jpeg	0	0.271924734
7	5.jpeg	0	0.288565367
8	6.jpeg	0	0.288565367
9	7.jpeg	0	0.288565367
10	8.jpeg	0.546527088	0.246963814
11	9.jpeg	0.920973122	0.188721642
12	10.jpeg	0.41340211	0.188721642
13	11.jpeg	0	0.113838859
14	12.jpeg	0	0.113838859
15	13.jpeg	0	0.113838859
16	14.jpeg	-1	0
17	15.jpeg	-1	0
18	16.jpeg	-1	0
19	17.jpeg	-1	0.022315456
20	18.jpeg	-1	0.18040134
21	19.jpeg	-1	0.18040134
22	20.jpeg	-1	0.18040134
23	21.jpeg	-1	0.18040134
24	22.jpeg	-1	0.36348027
25	23.jpeg	-1	0.837770045
26	24.jpeg	-1	1
27	25.jpeg	-1	1
28	26.jpeg	-1	1
29	27.jpeg	-1	1

Figure 10: Sample Training Data Labels

With the data obtained from this framework, we can train a neural network, which will be discussed in Chapter 5.6.

5.5 Autonomous Navigation

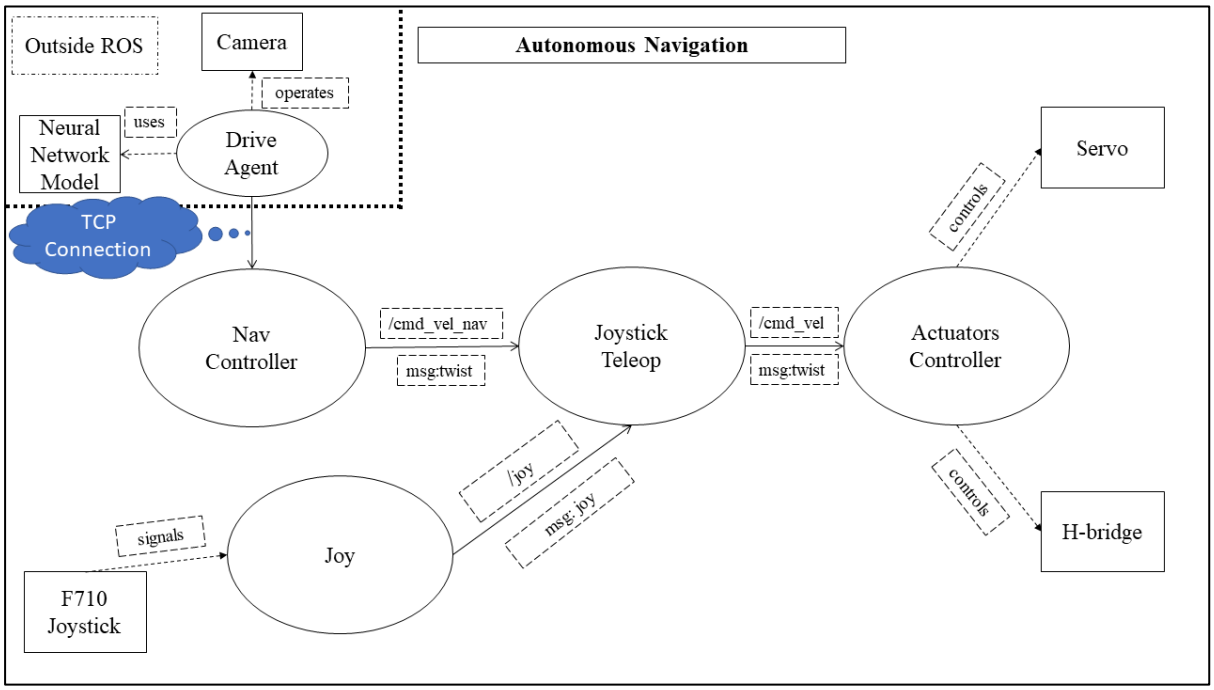

Figure 11: Autonomous Navigation Software Architecture

Last but not least, Figure 11 above shows the architecture of the car for autonomous navigation. It also builds on the Drive-by-Wire version of the system as seen in Figure 7, by adding the *nav_controller* node, and another piece of code that is outside of ROS space, *drive_agent*. This iteration of the system is used to test the vehicle **after** training a neural network.

The *drive_agent* code is used to operate the camera during autonomous navigation and runs the trained neural network model. Each image captured by the camera is used for prediction by the neural network. The output of the neural network forms the command to be sent for autonomous navigation. Unfortunately, the machine learning library used, TensorFlow Lite, runs on Python 3, while the current version of ROS still runs on Python 2, therefore, *drive_agent* cannot be run as a ROS node. To overcome this limitation, drive_agent is run as a python code outside of the ROS space and communicates with ROS using a TCP connection. For this TCP connection, *drive_agent* acts as the TCP client.

In ROS space, *nav_controller* is set up as a TCP server to connect to drive_agent. The output of *drive_agent* is then sent over the TCP communication and received by nav_controller, and

nav_controller then repackages this message into a ROS message of type *geometry_msgs/Twist*[4] and sends it to joystick_teleop node using the */cmd_vel_nav* topic.

The *joystick_teleop* node, upon receiving the autonomous control message from the */cmd_vel_nav* topic, checks the current state of the vehicle: if the current state is not "autonomous", this message will be dropped. Otherwise, the message will be forwarded.

5.6 Training the Neural Network

To train a neural network, we opt to use the TensorFlow library, as its mobile equivalent, TensorFlow Lite, is known to perform well on devices such as the Raspberry Pi v4. The entirety of the training process is done on Google Colaboratory, because we want to ensure that all future competitors will at least be able to make use of cloud GPUs and are not too disadvantaged if they do not own devices with GPUs for training themselves.

The dataset obtained from the Training Data Collector architecture in Chapter 5.4 is used to train models that are in turn used for the Autonomous Navigation architecture in Chapter 5.5. An abstract view of the entire training process on Google Colaboratory is shown in the self-explanatory diagram below.

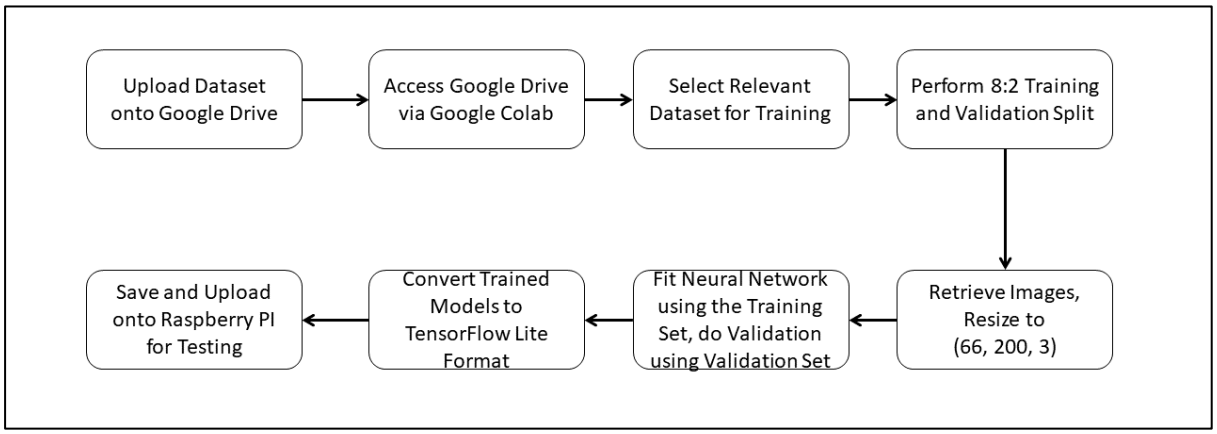

Figure 12: Process of Training a Neural Network on Google Colaboratory

[4] http://docs.ros.org/melodic/api/geometry_msgs/html/msg/Twist.html

6 Experiments and Results

6.1 Initial Track Design and Methodology

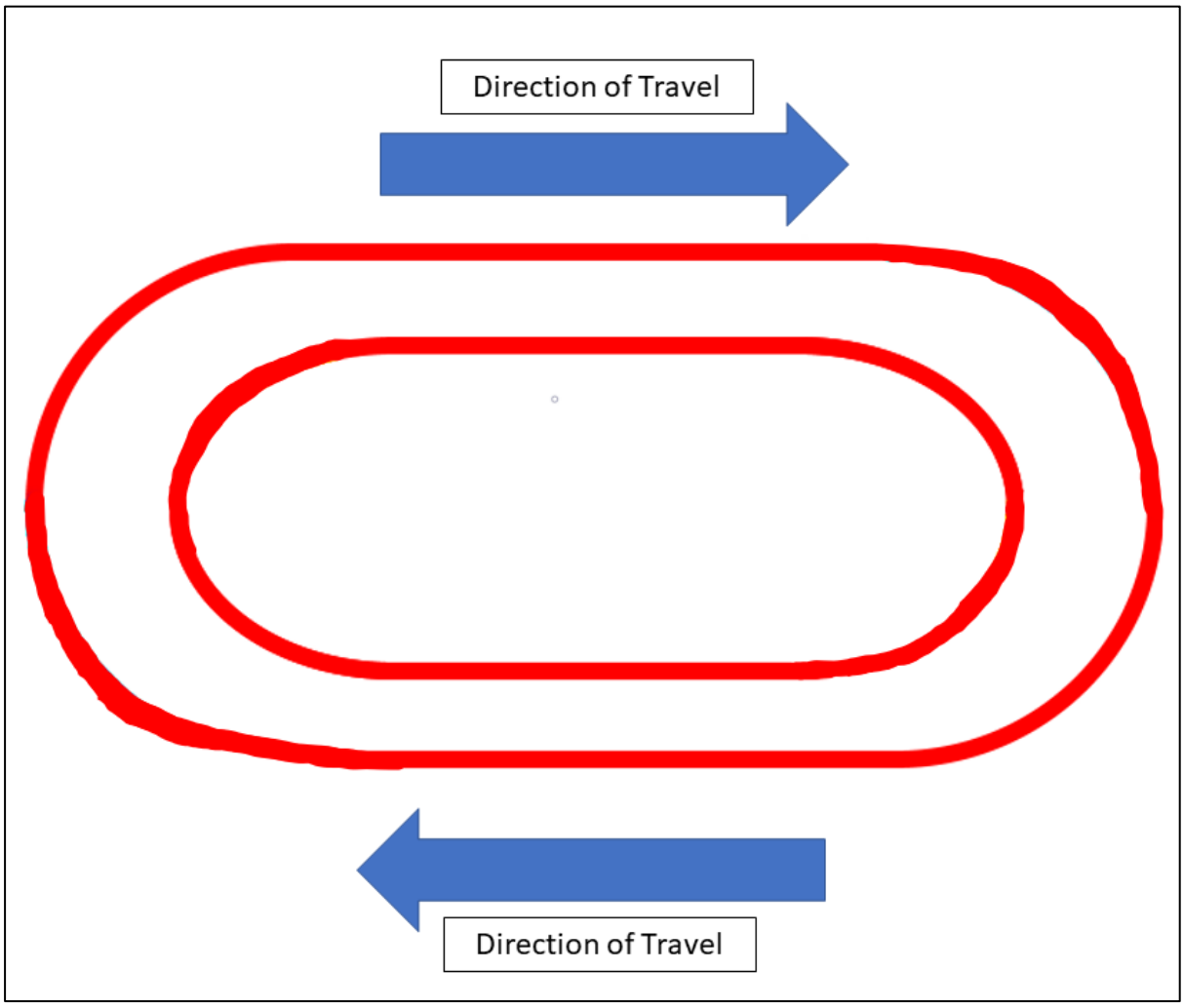

Figure 13: Illustration of Initial Track Iteration

Figure 13 above shows an illustration of the initial track design. For the experiments, we will use a simple oval-shaped track to determine the feasibility of the proposed solution described in Chapter 4. The track is demarcated using coloured cloth tape, on the plastic floor of the Multipurpose space in NUS COM2 building.

The experiment procedure is as follows:

1) Use Training Data Collector and drive the car around the track to collect training data
2) Train the neural network using the training data
3) Evaluate the trained model using Autonomous Navigation
4) Repeat

For each run, we want to qualitatively evaluate the performance of the car. This evaluation can be divided into 3 areas, as illustrated in Figure 14 below.

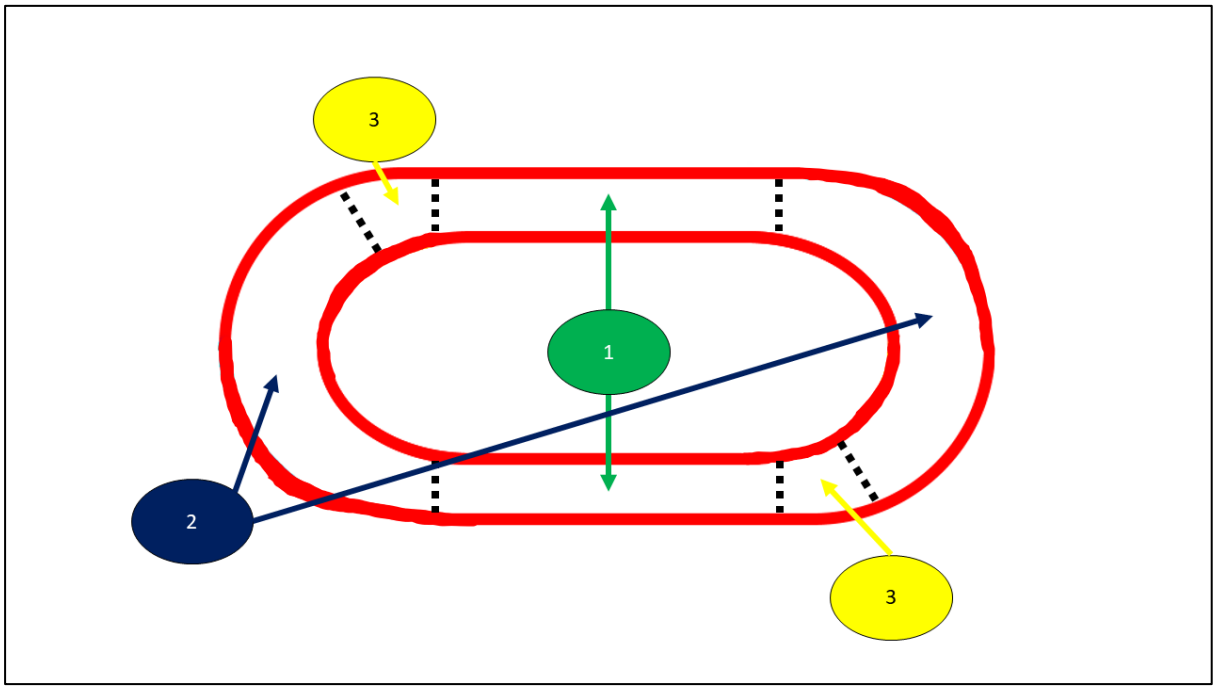

Figure 14: Illustration of Areas on Track for Qualitative Evaluation

At the areas marked (1), we want to evaluate the quality of the normal driving. This should be the simplest part for the neural network. If the car is within the area demarcated by the tapes, then it need only maintain a steering close to 0, while the speed prediction is not crucial to successfully complete this stretch.

At the areas marked (2), we want to evaluate the quality of drifting. We want to see if the car is able to autonomously perform drifting similar to the illustration shown in Figure 1.

At the areas marked (3), we want to evaluate how the car is able to recover after a drift into the normal orientation for straight driving.

By observing and evaluating the performance of the car using these 3 areas, we can determine the overall quality of the algorithm and controller.

6.2 Hypothesis and Problem Statement

To reiterate, for the experiments, we are attempting to prove feasibility of the solution proposed in Chapter 4, which is to use a Convolutional Neural Network and input images to

predict steering and throttle commands such as to produce both driving and drifting of the car about the track.

It is first assumed that the neural network can learn the concept of driving and drifting, and this is backed up by some intuition and the Universal Approximation Theorem.

In the experiments, we aim to design a track, with our original track as a basis, such that we can give enough visual information to train the neural network properly. We want minimal changes to the track, so as to not give too many hints such that the competition would become too easy. By doing incremental changes to the track until we see adequate drifting and driving, we will ideally arrive at (A) a working demonstration of our proposed solution, and (B) a track design for the competition.

6.3 Results on Initial Iteration of Track

On the initial track that is made from a single colour of tape, the results left much to be desired. From the 3 criteria we look at, we see that the car is simply unable to start a drift or perform a drift motion that is anywhere close to that of the training data. Since it cannot perform a drift, the third criteria fails by default as well. However, we observe that even for the first criteria, the car is also unable to drive straight, which should be the simplest task. Indeed, when we observe the values produced by the models, there seems to be a constant bias of steering to either the left or right, depending on the particular model. Therefore, it is deduced that there is a lack of sufficient visual cue to provide enough hints to the neural network about the desired motion. The results are summarized in the table below:

Area	**Observation**	**Pass / Fail**
1: Straight driving	Always a bias to one side instead of driving straight	Fail
2: Drifting	Does not follow the track, arbitrary drifting	Fail
3: Recovery	Fails by default since (2) is failed	Fail

Table 3: Qualitative Evaluation of Results on First Track Iteration

Evaluation

From these results, the key takeaway is that the car fails even the simplest task of normal driving in Area 1. It can be surmised that the algorithm fails to learn the relevant features

from the track when only a single colour of tape is used. Therefore, to alleviate this problem, more visual cues need to be added to our track to improve the training of the neural network model.

6.4 Results on Second Iteration of Track

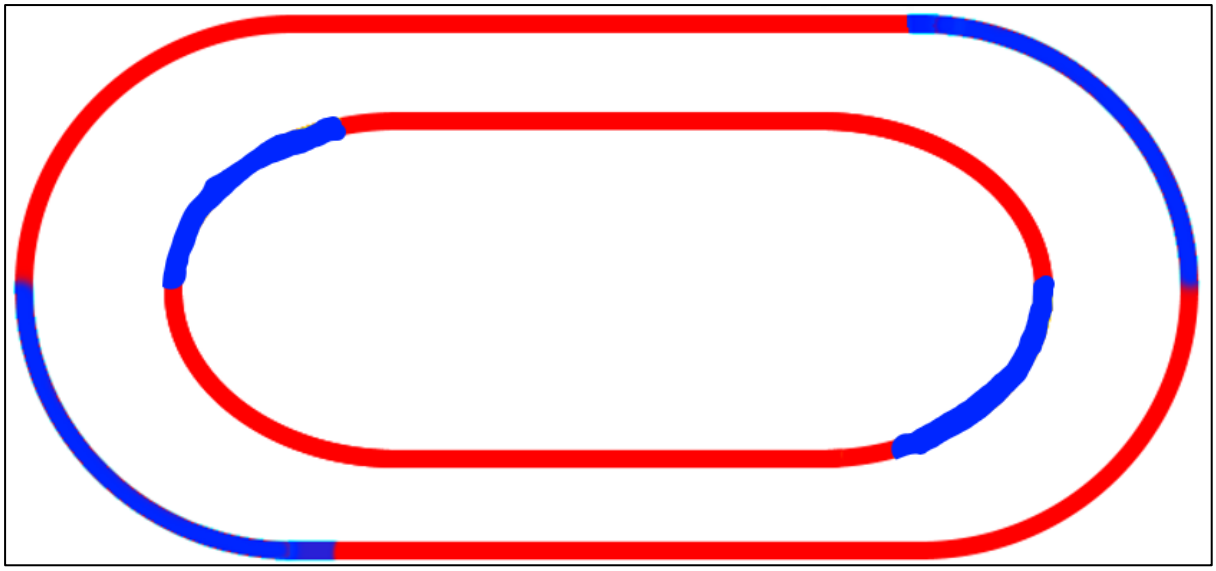

Figure 15: Illustration of Second Track Iteration

Based on the performance of the robot on the first track iteration, it is surmised the track that consists of only a single colour tape provides insufficient information for the machine learning algorithm to adequately estimate the target function. Therefore, parts of the track are replaced with a second colour, blue. Blue tape is added to the outer ring to denote the start of a bend, and also to the inner ring to denote the end of the bend. The idea here is summarized below:

If the car sees	**Action**
Red tape on left and red tape on right	Drive normally
Blue tape on left and red tape on right	Start a drift and perform drifting
Red tape on left and blue tape on right	Recover from drift and prepare to drive normally

Table 4: What the Car Should See and What Action It Should Take for Second Track Iteration

An example of the performance of the car here is captured on video and can be seen via the QR code provided on this page.

Pertinent Material 1:

Footage of typical experiment on second track iteration.

Link: https://www.youtube.com/watch?v=oBOaVnyU4mY

The results on the second track iteration are summarized in the table below:

Area	Observation	Pass / Fail
1: Straight driving	Drives straight and within the tracks	Pass
2: Drifting	Performs drifting	Pass
3: Recovery	Works only occasionally	Fail

Table 5: Qualitative Evaluation of Results on Second Track Iteration

In general, the car is barely able to complete 1 round most of the time, with only 1 "lucky" attempt where it was able to complete 3 full rounds.

Evaluation

The results show that our assumption from the first track iteration, that there needs to be more visual cues, turns out to be largely correct, as simply by adding in 1 more colour to differentiate Areas 1, 2, and 3 results in a model that can perform straight driving and drifting, but often fails to recover after the drift.

From observation, the reason for failure appears to be largely due to the orientation of the car during drifting, as seen in the following images, all of which are snapshots of the video in the QR code above.

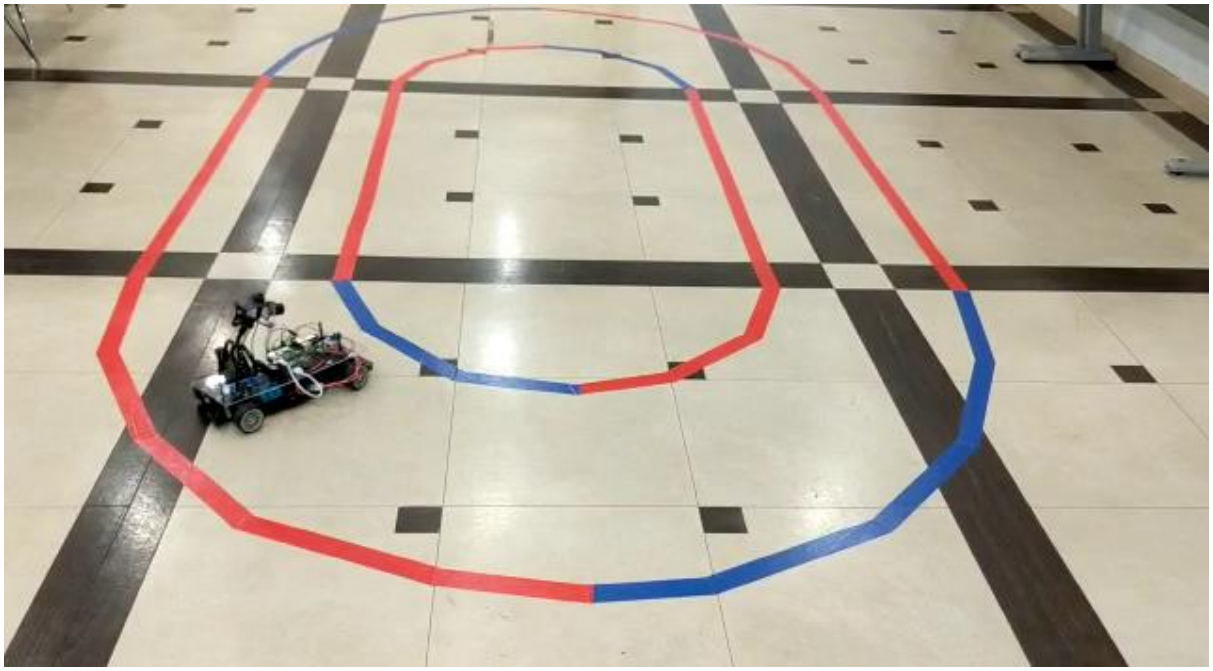

Figure 16: Orientation of Car at End of a Drift[5]

From Figure 16 above, we see that at the end of a drift, the car's orientation is usually such that it is facing "inwards" at the end of a drift. This obviously shows that our assumption with regards to the final row in Table 4 on what the car should see during "Recovery" was incorrect. Instead of seeing blue on the right and red on the left as we assumed, the input image is likely seeing mostly blue instead, which causes confusion and, in the video, causes the car to turn to the right as if it is at the start of the drift area.

[5] Image captured at (0:09 / 0:15) from video of experiment https://www.youtube.com/watch?v=oBOaVnyU4mY

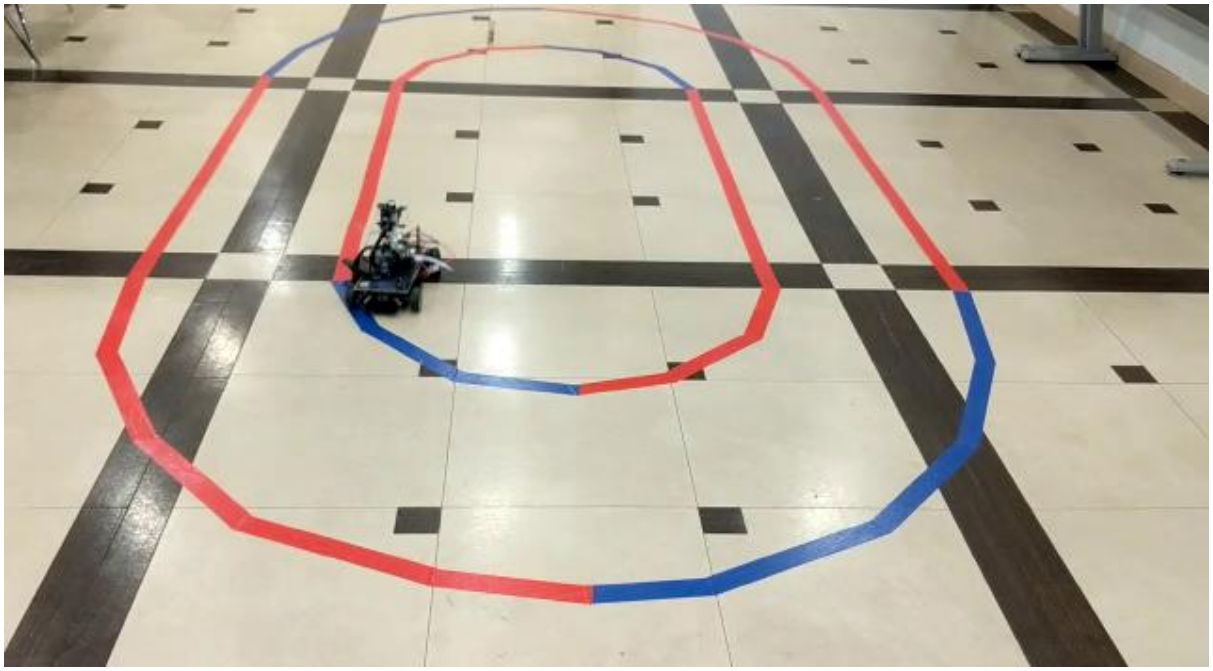

Figure 17: Car Steers Right Due to Confusion, Part 1 of 2

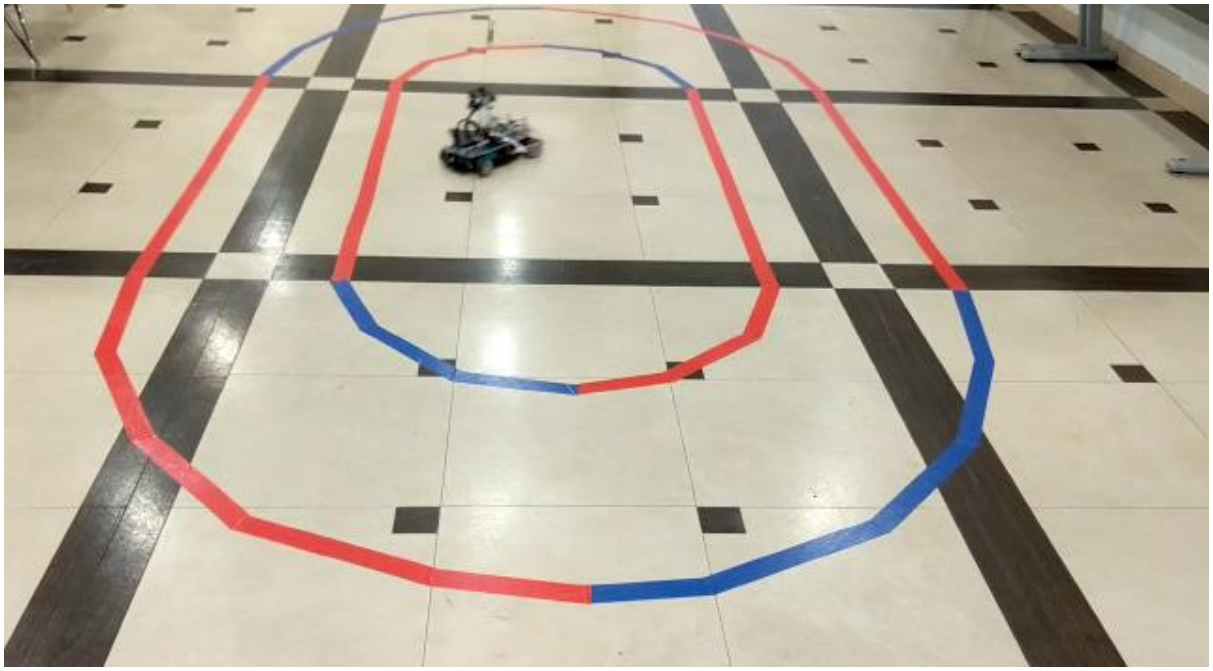

Figure 18: Car Steers Right Due to Confusion, Part 2 of 2

Therefore, our improvement for the following track iteration is to make this area for recovery more distinct from Area 2, in such a way that even in the orientation seen in Figure 16, the chances of confusion are minimal.

6.5 Results on Third Iteration of Track

To reduce the confusion between the start of a drift and the end of a drift and taking into account the orientation of the car when it performs drifting, the inner lane at the end of the drift is replaced with yellow tape, as seen in the figure below.

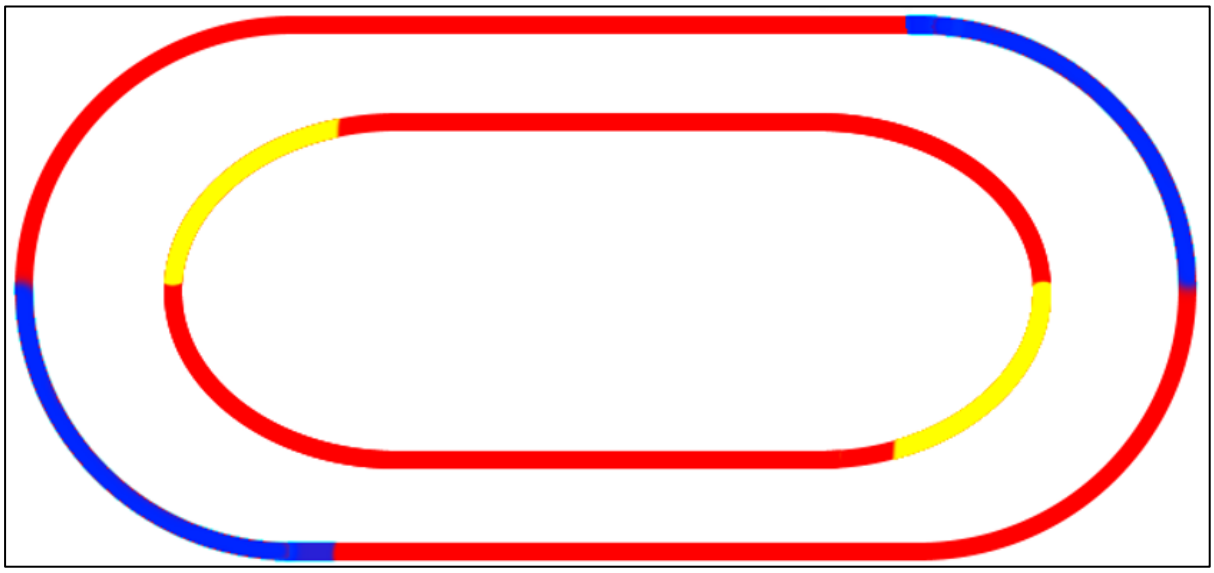

Figure 19: Illustration of Third Track Iteration

The results of the experiments on this third track iteration is also captured on video and can be seen via the QR code on this page.

Pertinent Material 2:

Footage of experiment on third track iteration.

Link: https://www.youtube.com/watch?v=JbdmZDMvEMo

Again, the results are summarized qualitatively in the table below:

Area	Observation	Pass / Fail
1: Straight driving	Drives straight and within the tracks	Pass
2: Drifting	Performs drifting	Pass
3: Recovery	Able to recover well and as needed	Pass

Table 6: Qualitative Evaluation of Results on Third Track Iteration

In general, the car is now easily able to complete at least 10 full rounds continuously, with even a particular run being able to complete more than 20 rounds, fully demonstrating the success of the experiments.

Evaluation

From these results, it can be seen that:

1) The proposed solution from Chapter 4 is feasible and adequate for drifting
2) Earlier assumptions that there was a lack of visual cues proven to a large extent

From these results, we can also surmise that for the creation of any general track for the competition, we need to demarcate the areas that correspond to drifting and recovery to achieve this result.

With this, we have successfully demonstrated a robust controller for autonomous drifting using a single camera, and also arrived at a possible track design for the competition.

The results of these experiments were used to promote and secure funding for the competition, as well as convince students from the CS3244 Machine Learning module to join our competition as part of their module project.

6.6 Noteworthy Observations

Over the course of the experiments, some important observations were noted during training of the neural network models.

Observation #1: Robustness of Controller

It is observed, from experiments conducted on the final track iteration, that although sometimes the car ends up outside the track as part of its drifting, the controller is robust enough to realise that it needs to drive the car back into the track[6]. It is surmised that this "feature" is because during the collection of training data, the driver does sometimes miss the correct controls and ends up driving outside of the track for a short moment, although he does drive the car back in as quickly as possible. This results in the neural network "realising" that

[6] See (0:28 to 0:30) of video from Pertinent Material 2 in Chapter 6.5

in general, the car should be between the tapes, otherwise, it has to correct itself appropriately.

This leads to us believing that having slightly noisy data, where the driver does not always drive perfectly, may actually be beneficial to the learning algorithm in creating a controller that knows how to stay within the track markings.

Observation #2: Minimum Validation Loss

From our experiments, it is noted that for any model to successfully perform well on the track, the validation loss, based on mean square error, is at most 0.08. Of course, this does not mean any model with less than 0.08 loss will work, as the model might have overfitted to the data instead. Therefore, models with good validation loss (less than 0.08) must still be tested onboard the car to verify and evaluate its quality.

Still, this observation helps, as a rule of thumb, in narrowing down the number of models that need to be tested, since models with a validation loss higher than 0.08 are less likely to succeed anyway, so there is little value in testing them.

Observation #3: Training Epochs

On the topic of validation loss, it is also noted that in every instance of training the neural network, epochs greater than 10 converge to a validation loss of between 0.25 to 0.278. This implies that, for this experiment and based on our typical datasets, there is no point in setting training epochs greater than 10 for training the neural network.

7 Implementation of Competition

After demonstrating feasibility of the approach, the team organized the inaugural drift racing competition, which we named the **AutoDrift Grand Prix NUS 2020**. We approached students of the CS3244 Machine Learning module for volunteers to compete in the competition as part of their module project deliverable.

7.1 Software for Competitors

For contestants, a GitHub repository was set up with some of the code that can serve as a starting point for contestants to begin writing their code. The started code distributed to competitors is the entirety of the Autonomous Navigation architecture, as described in Chapter 5.5, with the omission of some code that relates to the output layer of the neural network.

The idea here is to give to competitors all the code that is required to "Drive-by-Wire" and provide them with some help in interfacing the Machine Learning library with ROS (in particular, the TCP connection).

The code is provided via a repository on GitHub, which can be found on the QR code below. In addition, a website was set up using GitHub pages, to serve as an introduction to the competition, which can also be found in the GitHub repository below.

Supplementary Material 2:

GitHub repository for contestants

Link: https://github.com/PokkaKiyo/Autodrift-Grand-Prix-NUS

7.2 Racetrack on Vinyl Surface & Promotional Video

Work was put into transferring the track design from the plastic floor of the Multipurpose space onto a piece of vinyl, which the team procured and fabricated. The idea was to have a track that can be rolled up and transported. This new track fabricated on vinyl sheet is shown in the following figure. Some differences in performance of the car on vinyl were observed and are noted in Melvin's report.

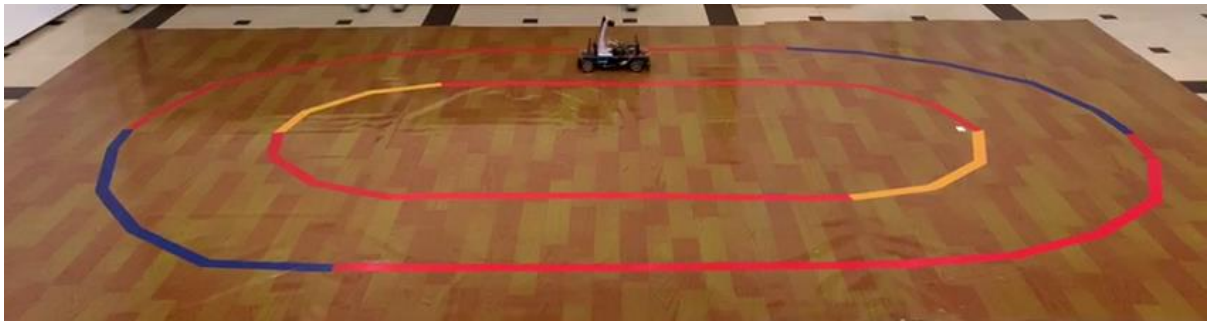

Figure 20: Racetrack on Vinyl Sheet

Additionally, a promotional video for the competition was made. The footage was taken by mounting a smartphone onto the front of the car and taking a video from there. This promotional video can be found on the official GitHub pages website, or by scanning the QR code below.

Supplementary Material 3:

Promotional footage for competition

Link: https://www.youtube.com/watch?v=Pj0B8jCV61k

7.3 Competition Rules

Furthermore, a set of rules for the competition were set up for contestants, with advice from our supervisor.

AutoDrift Grand Prix NUS 2020

Version 1.1, 5 March 2020

Hardware

During the actual time trial, only the original car with no modifications is allowed.

During the Competition

1. Each team will be allowed up to 3 attempts, and a total of 15 minutes allocated for all the attempts.
    a. Before an attempt, the teams can place their car wherever they like, and the timing starts when they press the start button on the joystick.
    b. In between the attempts, teams can make any modifications such as changing the code, cleaning the wheels etc.
    c. The best attempt will be used for determining the winner.

2. Contestants will be allowed to connect 1 external computer to the Raspberry Pi wirelessly (e.g. SSH / VNC).
    a. This is for starting and stopping your script(s). When the car is autonomous, contestants are not allowed to do anything from their remote computer which will affect the car's performance.

3. Contestants to connect 1 wireless joystick, to start and stop the car.
    a. When the car is autonomous, one person from the team to hold onto the controller and be prepared to stop the car. If the car goes out of control and has to be stopped, that attempt will be void.
    b. Teams are not allowed to manually control the car once it has gone fully autonomous. Otherwise, that attempt will be void.

Winning Criteria

1. To **pass**: contestants need to complete the track 5 full rounds autonomously, clockwise direction.

2. For contestants that succeed in completing 5 full rounds, the total time taken from the start of the attempt to completing the 5^{th} round will be used to compare with other contestants who pass. The team with the shortest amount of time taken will be the winner.

3. During an attempt, if the car leaves the track, we will freeze the number of rounds completed as an integer, and the counting will resume as soon as the car enters the track again.
    a. Leaving the track is defined as when none of the wheels are within the track or touches the lanes of the track.

8 Conclusion

8.1 Summary

In summary, this report mainly discusses the software implementation of a drift controller for a $1/10^{th}$ sized RC car. The solution proposed is a novel approach that combines a convolutional neural network with the insights provided by Goh et al (2020) in their report on drifting a life-sized DeLorean. Experiments with this solution demonstrated adequate and robust drifts on our track, on 2 different surfaces, plastic floor and vinyl sheets. This finding demonstrates feasibility of the proposed drift racing competition.

8.2 Limitations

Firstly, the solution proposed is not robust to physical changes to the robot body, such as significantly modifying the weight of the car, or rearranging the weight of the car such that the balance of the car changes. This is because the only input to the neural network are images captured by the front facing camera attached, which does not contain information related to any physical changes on the actual car itself. In such instances, the neural network model typically has to be retrained using new training data to perform adequately well.

Secondly, the current iteration of the competition only has a single racetrack design. While this is enough to show feasibility of the algorithm, it does not yet show the ability of the algorithm to generalize across different racetracks.

8.3 Recommendations for Further Work

In this section, 3 recommendations for further work are proposed.

The first recommendation, as inspired from the second limitation described in the previous chapter, is to have multiple tracks, including an unseen track, for the competition. Currently, the competitors train and test their algorithm on the same track. To test the robustness of their algorithms and design, multiple tracks could be used such that we can evaluate the general solution, rather than a solution specific to a certain track.

The second recommendation is to investigate having multiple cars in the track racing at the same time. This is clearly a much more difficult task but would greatly increase the quality and appeal of the competition.

Last but not least, the third recommendation is to incorporate other types of sensors. In this report, a successful implementation of a drift controller using a single camera has been demonstrated. However, by incorporating more sensors, it is likely that a better solution can be derived. It also allows competitors to have greater freedom to design their systems, and not be limited by the sensors provided. Some sensors recommended are detailed in the table below, in increasing order of perceived difficulty:

Sensor	**Purpose**
Inertial Measurement Unit (IMU)	For gathering odometry data, such as acceleration.
(Optical) Wheel Encoders	For measuring wheel rotations, wheel speed, odometry data.
Lidar	For Simultaneous Localization and Mapping (SLAM) algorithms.

Table 7: Sensor Recommendations for Further Work

References

Bhattacharjee S. (2018) Autonomous Drifting RC Car with Reinforcement Learning (Reinforcement Learning Report). Final Year Project, University of Hong Kong.; 2018.

Bojarski, Mariusz & Testa, Davide & Dworakowski, Daniel & Firner, Bernhard & Flepp, Beat & Goyal, Prasoon & Jackel, Larry & Monfort, Mathew & Muller, Urs & Zhang, Jiakai & Zhang, Xin & Zhao, Jake & Zieba, Karol. (2016). End to End Learning for Self-Driving Cars.

Bojarski, M., Yeres, P., Choromanska, A., Choromanski, K., Firner, B., Jackel, L., & Muller, U. (2017). Explaining how a deep neural network trained with end-to-end learning steers a car. arXiv preprint arXiv:1704.07911.

Cutler, M. and How J. P. (2016). Autonomous drifting using simulation-aided reinforcement learning. 2016 IEEE International Conference on Robotics and Automation (ICRA), Stockholm, 2016, pp. 5442-5448.

Goh, J. Y., Goel, T., & Christian Gerdes, J. (2020). Toward automated vehicle control beyond the stability limits: drifting along a general path. Journal of Dynamic Systems, Measurement, and Control, 142(2).

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... & Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In Thirty-Second AAAI Conference on Artificial Intelligence.

Hindiyeh, R. Y., & Christian Gerdes, J. (2014). A controller framework for autonomous drifting: Design, stability, and experimental validation. Journal of Dynamic Systems, Measurement, and Control, 136(5).

Jain, R. (2018) Autonomous Drifting RC Car using Reinforcement Learning (Hardware Report). Final Year Project, University of Hong Kong.; 2018.

Kabara, K. (2018) Autonomous Drifting RC Car with Reinforcement Learning (Software Report). Final Year Project, University of Hong Kong.; 2018.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.

Mur-Artal, R., Montiel, J. M. M., & Tardos, J. D. (2015). ORB-SLAM: a versatile and accurate monocular SLAM system. IEEE transactions on robotics, 31(5), 1147-1163.

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohband, S., Dupont, C., Jendrossek, L.-E., Koelen, C., Markey, C., Rummel, C., van Niekerk, J., Jensen, E., Alessandrini, P., Bradski, G., Davies, B., Ettinger, S., Kaehler, A., Nefian, A. and Mahoney, P. (2006), Stanley: The robot that won the DARPA Grand Challenge. J. Field Robotics, 23: 661-692. doi:10.1002/rob.20147

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. In Thirtieth AAAI conference on artificial intelligence.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., & De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. arXiv preprint arXiv:1511.06581.

Zubov I., Afanasyev I., Gabdullin A., Mustafin R. and Shimchik I. (2018) Autonomous Drifting Control in 3D Car Racing Simulator. 2018 International Conference on Intelligent Systems (IS), Funchal - Madeira, Portugal, 2018, pp. 235-241

Appendix A

Comparison of Common Autonomous Robot Sensors for Obtaining Odometry

The table below summarizes a comparison between common sensors used for obtaining the state information necessary for the use of the reinforcement learning solution as proposed in Chapter 2 of this report. For evaluation, 3 criterions are considered: cost, mechanical difficulty of implementation, and other qualitative remarks.

Sensor (and Algorithm)	**Evaluation**
IMU	Mechanical Difficulty: Low
	Cost: Can range from low to very high
	Data collected is too noisy for IMU to be the main sensor.
Optical Encoders mounted on the wheels of the RC Car for counting wheel rotations	Mechanical Difficulty: High
	Cost: Moderately High
	Does not give Velocity along the Y-axis
Lidar (for SLAM)	Mechanical Difficulty: High
	Cost: High
	Generally effective but not in our particular setup.
Camera (for SLAM)	Mechanical Difficulty: Low
	Cost: Low
	From experience, not particularly reliable, especially on fast moving objects.

Table A: Comparison of Common Autonomous Robot Sensors for Obtaining Odometry