Interim report (COMP4801)

# AUTONOMOUS DRIFTING RC CAR WITH REINFORCEMENT LEARNING

January 29, 2018

**Supervisor: Dr. D. Schnieders**

Sourav Bhattacharjee (3035123796)

Kanak Dipak Kabara (3035164221)

Rachit Jain (3035134721)

# Abstract

The advent of self-driving cars has pushed the boundaries on how safe passenger automobiles can be, but most modern self-driving car systems ignore the possibility of a car slipping resulting from inclement weather or driver error. Passengers and bystanders would benefit heavily if self-driving cars could handle slipping by learning to drift with the turn rather than against it (by applying the brakes, or turning away which is the instinctive action), preventing many fatalities [1].

Our project is aimed at studying the drifting (over steering of a car that results in the loss of traction of the rear wheels) of an autonomous remote controlled (RC) car. We use reinforcement learning techniques and algorithms to design a controller for an RC car that learns to drift without human intervention. Reinforcement learning is a branch of machine learning that primarily deals with learning a control agent from trial-and-error, much like how humans learn by interacting with the environment. Reinforcement learning has in recent years been used to learn all sorts of robotic controllers and even defeat the best human player at Go. It is an exciting realm of machine learning, and we decided on using it to teach an RC car to maintain a steady state circular drift. As for the technique employed, we use double dueling deep Q-networks and Q-learning as our primary algorithm. However, using reinforcement learning (RL) typically requires many interactions with the environment before learning anything useful. Since robotic systems are prone to wear with use, we implemented a simulator by modeling the car dynamics, where we run most iterations of the learning algorithm. In addition, since it is imperative to define the reward function appropriately to make sure that our agent learns the right behaviour in the shortest time possible, we also use potential based reward shaping to shape the rewards the agent receives.

The design philosophy behind the entire system we implemented to learn the best way to drift revolves around modularity, allowing each component in our implementation to be developed in isolation and easily replaceable with a different implementation. Currently, we are halfway through our project with the double dueling DQN algorithm and simulator completed and the hardware assembly of the RC car is nearly complete as well.

# Acknowledgement

We would like to thank our supervisor, Dr. Dirk Schnieders, for guiding us throughout in the project. We are also grateful for the help received from a friend, Mr. David Ip, who helped us acquire the hardware needed for this project and would also like to express our sincere gratitude to Mr. Keith Chau for guiding us through the writing of this report.

# Contents

# List of Tables



# List of Figures

# 1 Introduction

Before discussing the implementation details of the project, it is crucial to understand the background of the problem we are trying to solve, and the actual scope. This section addresses exactly that and highlights the need to study and solve the problem of drifting, and outlines how we plan to do so with an approach based on simulation aided reinforcement learning. For the purposes of this report, drifting is defined as the over steering of a car which results in the loss of traction of the rear wheels. This results in the front wheels pointing in the opposite direction to the turn and the car appears to be moving sideways.

## 1.1 Background and motivation

Passenger vehicles usually implement stability control in a number of ways like differential braking [2], active steering [3] [4] or integrated chassis control [5] [6] [7]. Other methods, based on independent wheel torque, have also been developed to make passenger vehicles more stable. However, these methods function by making sure that the tires avoid slipping. In doing so, these methods essentially restrict the operation of the vehicle. Similarly, control algorithms in current self-driving car systems (Anti-lock brake systems, Electronic stability control etc.) try and mitigate the chances of slipping due to its unpredictable nature [8]. Sufficiently lowering the speed of the car and making turns that are not too tight will mostly prevent slipping, but this does not consider cases where the system must make evasive moves to avoid crashes (in this case the speed and turn angle will most likely be sharp) or when a car is already in a slipping state due to the driver's fault. For example, hydroplaning, which refers to a situation where a layer of water builds up between the car tires and the road, is a major reason for vehicle accidents. According to the United States' Department of Transportation, 15.97% of all vehicle crash fatalities in the United States [1] are attributed to wet and icy roads. An autonomous car system must be prepared for the scenarios outlined above to ensure the safety of the passenger and bystanders, regardless of the weather conditions or the state of the car. To reduce fatalities and ensure that these car systems are as robust and safe as possible, it is absolutely essential to

study drifting, and eventually deduce how they can respond quickly to unintentional slipping states as those encountered due to hydroplaning. Not only can drifting be useful to steer out of these unintentional slipping states, but can also be useful in taking full advantage of the capabilities of a vehicle to avoid accidents in emergencies.

Many of the systems discussed above try to tackle the issue of stability control and slipping by approaching it as an optimal control and open looped problem with explicit dynamics model. Approaches using optimal control are often deterministic and use closed-form expressible equations of motions. The resulting policies depends entirely on the model used to compute them. Sometimes, these restrictions on the model neglect parts of the true system either because they are non-linear or they are just not well-enough understood to be expressed in equations. We thus propose a method that is developed using a technique that does not rely on explicit equations of motion, but rather on an implicit understanding of the world obtained by trial and error.

One of our other motivations to study drifting is that Paul Frère [9] points out the usefulness of drifting to turn fast around sharp bends. Since high-speed stability is of greater importance to ordinary touring vehicles and competition cars, they tend to understeer, and for a competition car average circuit time is improved by going fast through fast bends while slowing down through sharp ones [9]. However, a car is able to undoubtedly turn sharp bends faster by drifting because the yaw angle formed by the drift brings the vehicle in line with the straight path following the bend even before the vehicle completes the turn [9].

## 1.2 Objective

The objective of this project is to get a remote controlled car to learn to drift autonomously. This paper proposes a framework for learning the best way to drift using simulation aided reinforcement learning. We believe reinforcement learning is a desired approach to solving the problem since we would like our algorithm to learn the best way to drift without having to input the dynamics of the system

explicitly. Then the learned optimal drift policy or strategy in simulation will be transferred for further learning on the physical RC car.

## 1.3 Scope

The area of drifting falls into two categories – sustained drift and transient drift. Due to the wide breadth of the two categories and the time constraints, our project will mainly focus on sustained drift, and more specifically steady state circular drift. In particular, our first objective is to get the RC car to maintain a steady state circular drift because it is the easiest to accomplish. Additionally, if time allows, we may also try to get the car to learn how to escape a drift state and return to a safe stop or continue in a straight-line motion. Furthermore, we may try to experiment with getting the car to drift in a pattern like the pattern "8", for instance, to see how much more difficult it is compared to having the car in a state of steady circular drift.

## 1.4 Deliverables

There are three major deliverables in this project:

1. Reinforcement Learning (RL) algorithms – We will implement and test various RL algorithms like Double Dueling Deep Q-networks with Q-learning, Iterative Linear Quadratic Regulator and Actor-Critic to perform steady state circular drift and compare their performance. These algorithms will run on the simulator (described below) to give an optimal policy for drifting, which will then be transferred to the RC car to validate the algorithm.

2. Simulator – We will run the RL algorithms on a simulated environment that approximates the RC car in a world with physics that mimics the real world. The environment can be used to test and improve different RL algorithms efficiently and quickly, without causing any wear and tear or damage to the RC car.

3. A remote controlled (RC) Car – This car will be a 1/10th scale model of an actual car, integrated with sensors (Inertial measurement unit combined

with magnetometer and optical sensors) for measuring data like the position, speed of the car and polar angles, to perform steady state circular drifting. The optimal policies created on the simulator will be transferred onto the RC car to validate our results.

## 1.5 Outline of Report

The remainder of this report proceeds as follows. First, we will provide a literature review on the various methods that have been used to implement steady-state drifting, the possible RL algorithms that we may use for development and the motivation behind using neural nets to approximate optimal drift strategy in the project. Next, we will give a detailed description of all the different components involved in making the system, and the architecture design that combines them. Then, we will talk about the status of the project's development, results obtained, and the various difficulties faced so far. Finally, we will conclude by discussing the future steps for the months remaining, and a few possible extensions to the project we may decide to tackle.

# 2 Literature review

## 2.1 Optimal control approach

Sustained drift with various optimal control techniques has been explored through multiple prior research. For instance, Velenis et al. [10] described a simple single-track vehicle model using equations of motion to design a 'sliding' control policy to stabilize steady state conditions using basic acceleration/braking applied to the wheels. Similarly, Hindiyeh and Gerdes [11] developed an open-loop control policy using nested feedback loops to attempt stable drift equilibrium. They too developed a complex model of the vehicle, chassis and wheels to form the basis of their control policy. On the other hand, Wu and Yao [12] created a control algorithm to stabilize a RC drifting car by balancing the tail sliding with counter-steering measures to prevent slipping during circular motion. Their system is based on understanding the dynamics of the car, including the planar force and moment generated by the car's wheels during drifting. These modeled approaches work well in scenarios where the model encapsulates the various dynamics of the real-world, but do not work well when the dynamics of the world are not understood completely to be modeled by equations of motion. The open-loop approach of the optimization cannot be implemented in the presence of uncertainties [13]. Thus, a better approach, which is independent of the underlying models, is needed.

This is the perfect use case for learning-based methods, specifically Reinforcement Learning (RL), in particular model-free approaches. Since model free RL algorithms learn policies by directly interacting with the environment, the policies are dependent on the real-world instead being reliant having a known model. The subsequent sections discuss exact model-free approach taken to solve our problem.

## 2.2 Reinforcement learning and potential reward shaping

Reinforcement learning techniques are employed in this project to learn an agent that maximizes the sum of expected future rewards [14], which is much like how humans learn by interacting with their environment. As illustrated in Figure 1, the agent interacts with the environment according to a policy by taking actions

and evaluates how good or bad taking a particular action in a particular state ($A_t$) is by observing the next state it transitions to ($S_{t+1}$) and the reward it receives along the way in the next time step ($R_{t+1}$). A state space is all the possible states that an agent can experience in the environment at any particular time while an action space is the set of all possible actions an agent can take [14]. A policy is a function that maps from the state space to an action [14]. More concretely, a policy is a function $\pi : S \rightarrow a$, where $S$ and $a$ are the state space and an action in the action space respectively. If an action $\pi(s)$ is taken by an agent in state $s$, it is said that the agent is acting according to policy $\pi$. The goal of any reinforcement learning problem is to find a policy $\pi$ that maximizes the expected sum of discounted future rewards (reward at state $s$ is given by $R(s)$),

$$E_{s_0,s_1,\ldots} = \left[\Sigma_{t=0}\gamma^t R(s_t)|\pi\right] \tag{1}$$

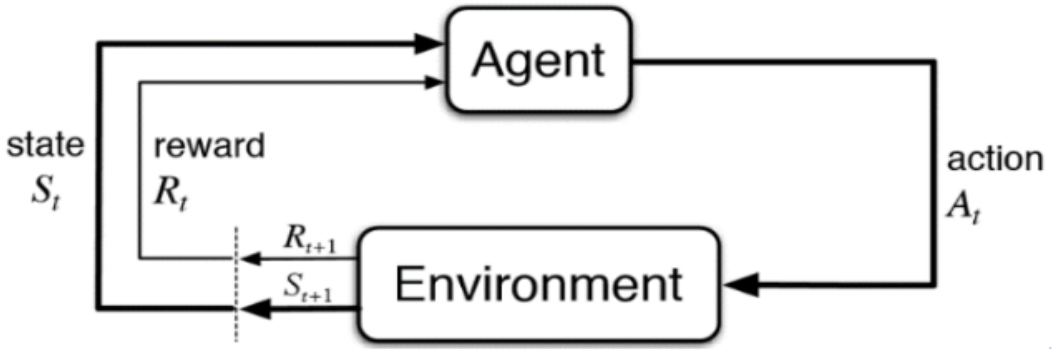


Figure 1: The reinforcement learning architecture.

The most imperative component of the reinforcement learning framework is defining the rewards. The reward function is like a semantic that controls the policy learned by an RL agent. So, it is absolutely essential to come up with a proper reward function that encourages our learning agent to behave in a way we

want it to [15]. Andrew describes one of these approaches to defining this reward through potential based shaping in [15]. The fundamental idea behind potential based reward shaping is that the learning agent is rewarded along the path we want it to follow and not just a huge reward at the end of achieving a goal. Another property of potential-based rewards is that it avoids the agent from being stuck in a sub-optimal positive reward loop and does not alter the optimal policy [15]

## 2.3 Dueling double deep Q-network

The final step in our methodology is coming up with the proper reinforcement learning algorithm to use. For our project, $Q$-learning [14] is used to find an optimal policy for drifting. The reason $Q$-learning was chosen as our tool to tackle the problem of finding an optimal drift policy is that it is an off-policy learning algorithm and is model-free. That means the algorithm learns by "looking over someone else's shoulder" without being explicitly told the dynamics of the system. This allows it to explore the state space by using a stochastic behaviour policy, $\beta$, while converging to a deterministic optimal policy, $\pi$. The algorithm is represented with a $Q$-learning neural network. Let's say at time step $t$ the state of the car is $s$, the action chosen according to the current policy is $a$ and the next state the car ends up in after taking the action $a$ is $s'$. According to the online $Q$-learning algorithm, the target for the $(s_t, a_t)$ pair is given by

$$y_t = R(s_t, a_t) + \gamma max_{a' \in A} Q_\phi(s', a'), 0 \leq \gamma < 1 \tag{2}$$

where $0 \leq \gamma < 1$ is the discount factor [14] and $\phi$ is the parameters of the neural net. Thus, the weights, $\phi$, of the neural network are adjusted to account for the error in the target and current value via optimization methods like gradient descent. Concretely,

$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(s_t, a_t)(Q_\phi(s_t, a_t) - y_t) \tag{3}$$

The basic online Q-learning algorithm has no convergence guarantees. (3) is not strictly a proper gradient descent since the target itself is ever changing and dependent on the parameters of the network. However, many research attempts

have been made to increase the chance of convergence, the ideas of which have been incorporated into our implementation. Firstly, in the online Q-learning algorithm, the state-action pairs are correlated (the next state is highly dependent on the current state and action taken). To overcome this problem, we use an approach similar to [16] and draw random batches of experience from an experience buffer, which holds the agent's past experiences. Secondly, we use two networks instead of one - a target network and a primary network, which overcomes the problem of overestimation of action values as described in [17]. The target network is used to get an estimate of the target in (2) while the parameters of the primary network are updated using optimization methods. The parameters of the target network are updated towards that of the primary network at a rate $\tau$ [17].

Finally, inspired by [18], we use a dueling Q network architecture with separate value streams and advantage streams. The reason behind doing so is to allow the network to learn the state value functions independently from the advantage of taking an action and remove the coupling to any specific action. An analogy to this is imagining seating at the beach and enjoying the sunset. The reward of the that moment is tied to only the state that an individual exists in and does not depend on any action taken.

We explored a number of other reinforcement learning algorithms and settled on $Q$-learning for a number of reasons. Firstly, we encountered an algorithm called $PEGASUS$ [19], which works with both continuous state and action spaces. The $PEGASUS$ algorithm works by making the simulated environment more deterministic by removing the randomness. This is done by choosing a string of random numbers that is used throughout the training as an input into the simulator such that for a particular state-action pair it outputs the same next state (no longer stochastic). It then builds a regression model on the state-action space [19]. In our project, since we have discretized the action space, we can afford to use just regular $Q$-learning with value function approximation to generalize to the continuous state space as discussed in a previous subsection.

Another algorithm that we came across is $PILCO$ [20]. In [21], Cutler and How describe a framework where we can initialize the policy using an optimal control software and combine that with PILCO run both in simulation and data gathered from the actual RC car to learn a policy in a data efficient manner. The idea behind PILCO is that during the learning phase, the algorithm also models the uncertainty of the environment using Gaussian processes [22]. The added advantage of doing this over regular model-based learning is that the agent is more cautious to assigning higher priority to actions in policies that are more uncertain [20]. The learning is simulation-aided, which means most of the learning takes place in a simulator. The algorithm also converges quickly to an optimal policy. Although, the idea of learning an optimal policy really quickly using $PILCO$ is appealing, we decided to use $Q$-learning instead because it is much simpler to implement than $PILCO$. Given that none of us had prior experience with reinforcement learning, we decided to use an algorithm that is much simpler conceptually and quicker to implement within the time constraint set for the project. We do, however, intend to explore using $PILCO$ in our project eventually if we get get time towards the end.

# 3 Methodology

With the relevant decisions made after a comprehensive literature review, we now talk about the design of this project. Since this project involves many different components, it is important to understand each one of them in isolation before the actual implementation can be discussed. This section describes the four major components that form the system, and the justification behind them.

## 3.1 Reinforcement learning and Autonomous Drifting

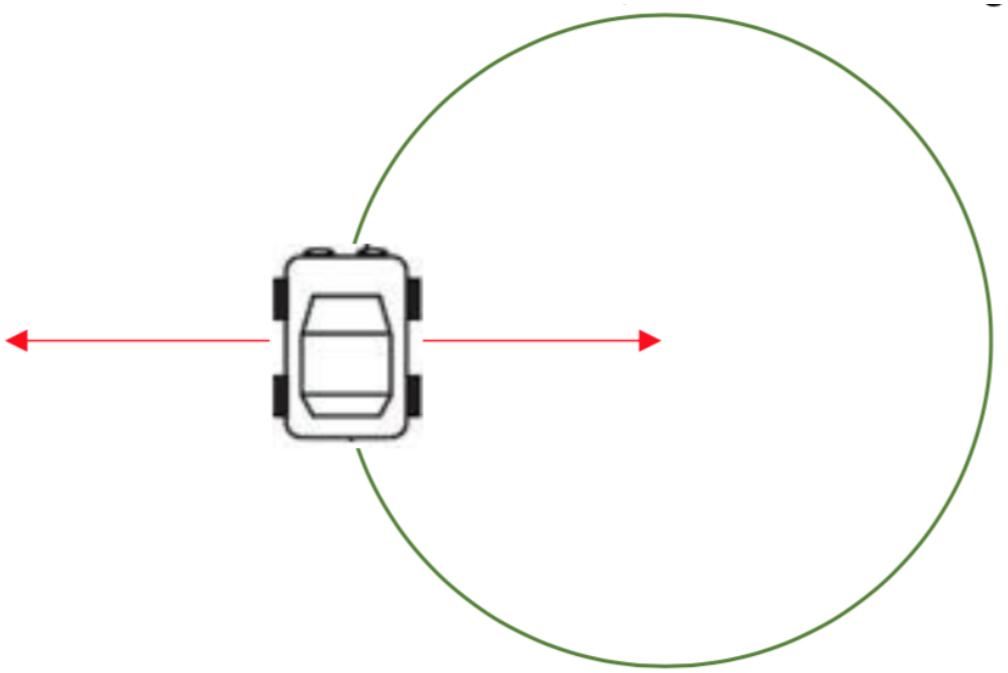

Figure 2: The reward function for autonomous drifting.

After discussing the reinforcement learning framework previously in the literature review, Figure 2 helps to relate it to our project. More concretely, for our project the reward for the agent is computed based on the position of the car in the 2 dimensional x-y space. Position space is restricted to the 2 dimensions because it will result in less computation done by the agent to converge to an optimal drift policy. The reward is maximized if the car manages to maintain a fixed radius,

$r$, from the centre of the circular drift trajectory, and the further it deviates from this circular track the larger the penalty it receives (negative reward).

To ensure effective exploration of policies, the agent needs to have a stochastic policy initially which eventually converges to a deterministic optimal policy [14]. This is a great approach for many RL problems, but cannot be afforded since exploring policies with an actual physical robot will lead to wear of the robot itself. Therefore, these iterations are run in a simulator. However, the dynamics of the drifting RC car cannot be easily expressed in equations and a method is devised to circumvent that entirely, as described in the subsequent sections.

### 3.1.1 Model identification of car dynamics

As previously discussed, our learning algorithm works with a simulator. However, expressing dynamical equations to represent the physical system is difficult. Thus, to fit a model of the dynamics of our drifting RC car, a different approach is taken. The car is first driven with a remote control for several minutes and record the states that it visits and the actions that are taken at each state. The state of the car is $s = [x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}]$, where $x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}$ are the x-coordinate, y-coordinate, polar angle, x velocity, y velocity and angular velocity respectively. The action, $a$, is an integer where $a \in A$ and $A = [65, 75, 85, 90, 95, 105, 115]$ which determines the steering angle. $A$ is the entire discrete action space.

In [15], Andrew describes the symmetries present in trying to control a helicopter autonomously using reinforcement learning. There are certain symmetries in the motion the of the helicopter that can be modelled into the learning to make sure the agent converges to an optimal policy quickly. Similarly, there are many symmetries in our system of the RC car. For example, it does not really matter whether the car is at the origin or at a point $(20, 25)$ when it needs to pick an action so as to rotate clockwise from that state. These symmetries will be encoded into our model by fitting the model not to the states in spatial coordinates but in the car body coordinates $s_{rc} = [\theta, \dot{x_{rc}}, \dot{y_{rc}}, \dot{\theta_{rc}}]$. Locally weighted linear regression [23] is used to model the dynamics of the car with $s_t$ (state at time $t$) and $a_t$ (action taken at time $t$) as the inputs and the state differences as the output (eg,

$\delta\dot{x}_t = \dot{x}_{t+1} - \dot{x}_t$). This model is fitted for each of the elements of the states. The reason behind choosing locally weighted linear regression as our tool for model identification is that given a test state-action pair, already seen states and actions that are similar will have a greater effect on predicting the output for the test pair than states and actions that are less similar. That is, similar state-action pairs are given a higher priority weight.

### 3.1.2 Value function approximation

After modeling the physical system, the next hurdle that needs to be overcome is that of the continuous state space. RL algorithms for small number of discrete states are relatively simple because the action values (the total expected discounted reward when an agent is at state $s$ and takes action $a$ [14]) for the individual states can be stored in a simple lookup table. What makes this approach difficult in our project is the fact that our RC car has a continuous state space and infinitely many states. For example, the $x$ and $y$ coordinates for the car's position and the linear velocities $\dot{x}$ and $\dot{y}$ are all real and continuous, contributing to the infinite state space. Thus, a different approach is needed to generalize to the state space for our RC car. That is why a function approximator will be used, like a neural net, to approximate the $Q$-values and generalize over the entire state space for the RC car. More concretely, given a state the car is in, $s$ as the input, our function approximator will output $\hat{q}(s, a, \mathbf{w})$, where $\mathbf{w}$ are the parameters of the function approximator (weights in the neural net) for all $a \in A$. $\hat{q}(s, a, \mathbf{w})$ will give us the approximate $Q$-value for the state-action pair and we do not need to store the action values in a table for each pair.

### 3.1.3 Reward definition

For our project, we use potential based reward shaping to make sure the car follows a circular pattern [15]. In addition, a negative reward is added that penalizes for deviation of the car's path from the target trajectory. Since the target trajectory is circle, an equation for the circle can be obtained and the reward for the deviation is the negative of the squared error between the car's actual position and the target trajectory. Moreover, to discourage large actions (drastic changes

in steering angle), the reward will also contain the negative of the squared value of the action. Our final reward is a summation of all these individual rewards. More concretely, the final reward, $R$ is related to the potential reward, $R_p$, the deviation reward, $R_d$, and the reward arising from action magnitude, $R_a$ as

$$R = R_p(\Delta\theta) - R_d(d, r) - R_a(a), \text{where} \tag{4}$$

$$R_p(\Delta\theta) = \begin{cases} 1, \text{if } \Delta\theta > 0 \\ -1, \text{otherwise} \end{cases} \tag{5}$$

$$R_d(d, r) = (d - r)^2 \tag{6}$$

$$R_a(a) = a^2 \tag{7}$$

### 3.1.4 Double Dueling DQN

As previously discussed in the literature review section of this report, we settled with using double dueling deep Q-networks for Q-learning of the action values due to the fact that it is an off policy and model-free algorithm. This also means that we do not explicitly need to know the dynamics of the system. The dueling architecture of our network in the implementation is illustrated in Figure 3. Our network comprises of a few fully connected layers and dropout layers initially, which then branches into two separate advantage and state value streams to combine again into a final fully connected layer to output the final action values. Although we plan to experiment with the number of the hidden layers their size to get an optimal network, our current implementation comprises of 3 fully connected layers initially of hidden size 500 and dropout 0.5. The 3 fully connected layers in the separate streams have a size of 300 each.

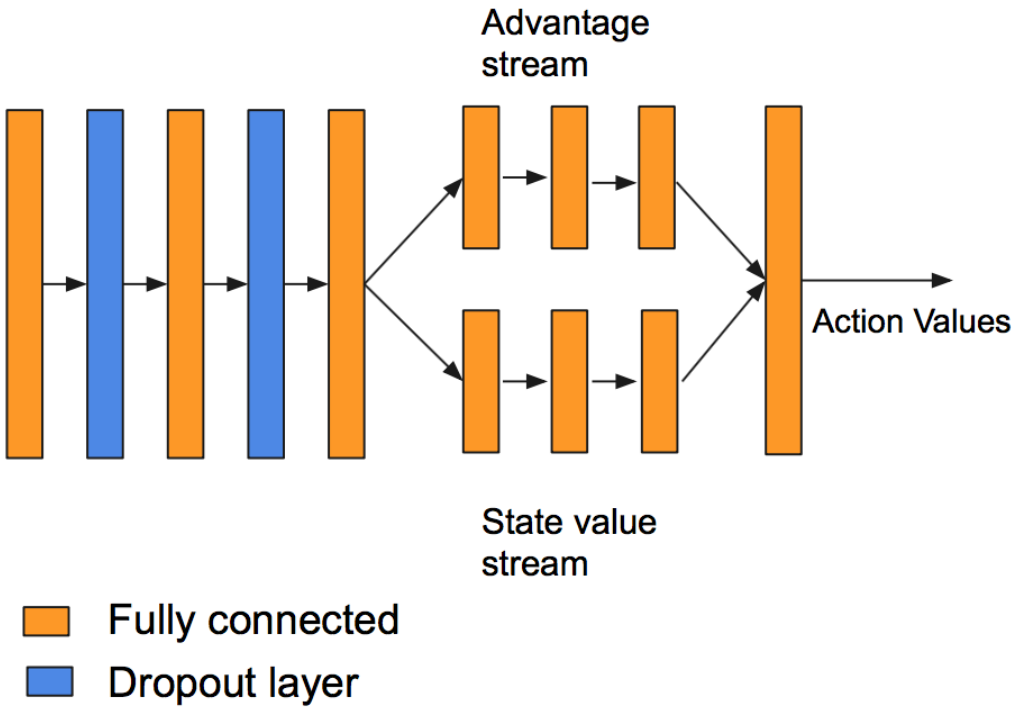


Figure 3: Dueling architecture.

### 3.1.5 Reinforcement Learning Framework

The various components of the RL algorithm described above are simple to implement without the use of any libraries, but having the ability to test various computational algorithms for the same [agent, action, environment] space without making changes to the rest of the system was important in order to have a robust and structured system where we could easily test various RL algorithms. The ideal solution would be to have a plug-and-play system where we could switch out different RL algorithms without having to modify other aspects of the system.

OpenAI Gym is an open-source library that is built on the tenet of providing all the tools necessary to develop and compare Reinforcement Learning algorithms. OpenAI Gym allows us to define an environment, which the agent can act upon using actions. The environment then returns the observation and reward. Essentially, this allows us to use the principle of Separation of Concerns, where the environment is just a 'black box' to the RL agent, and hence we can switch out agents easily without having to change out the environment. It is the OpenAI Gym environment's responsibility to query the position of the car and give an appropriate reward. It is also the OpenAI Gym environment's responsibility to accept any action that is supplied to it by the RL agent, and perform it on the car (real or simulated).

This separation of concern will allow us to test different RL agents that use different learning styles to figure out the most efficient strategy for learning.

## 3.2 Remote Controlled Car

The actual RC car will be connected with an Arduino microcontroller that will act as the hub of all computations that need to be performed to take actions on the car. The Arduino will have the following components:

- Inertial Measurement Unit (IMU) sensor – Inertial Measure Unit is an electronic device which is used to get motion data of a body such as acceleration, orientation, angular velocity etc. It uses a combination of gyroscope (angular velocity and orientation), accelerometer (acceleration) and magnetometer and provides reliable position for stabilization. We use MPU9250 which is a 9-axis Motion Processing Unit – 3-axis for gyroscope, 3-axis for accelerometer and 3-axis for magnetometer. MPU9250 uses I2C protocol for the communication. The other IMU that could have been used was MPU6050 but since MPU9250 is the world's smallest 9-axis motion tracking device with smaller chip size, better performance and low power consumption, we choose MPU9250 (SFUptownMaker). Also, MPU6050 is a 6-axis motion tracking device (3-axis for gyroscope, 3-axis for accelerometer) and lacks magnetometer.

- XBee module – XBee is a wireless module which provides end to end communication service. It creates its own network and can communicate with other XBees in its network. It needs atleast one more XBee for communication. Figure 2 shows the network made by two XBees. One XBee is connected to the Arduino and the other XBee is connected to the laptop. They create their own network and communicate with each other wirelessly. Similarly, in the project, one XBee is connected to the Arduino in the car and the other XBee is connected to the system. The XBee connected to Arduino sends the motion data received from IMU to the XBee connected to system. Similarly, the XBee connected to the system sends commands to XBee connected to the car.

Figure 4 describes the detailed architecture of the components on the RC car (blue boxes). There are two actuators on the RC car, a servo motor to control

the steering direction and a brushless electric motor to control the throttle. The motor will be connected to an Electronic Speed Control (ESC) module to provide power and regulate the voltage consumption by the motor.

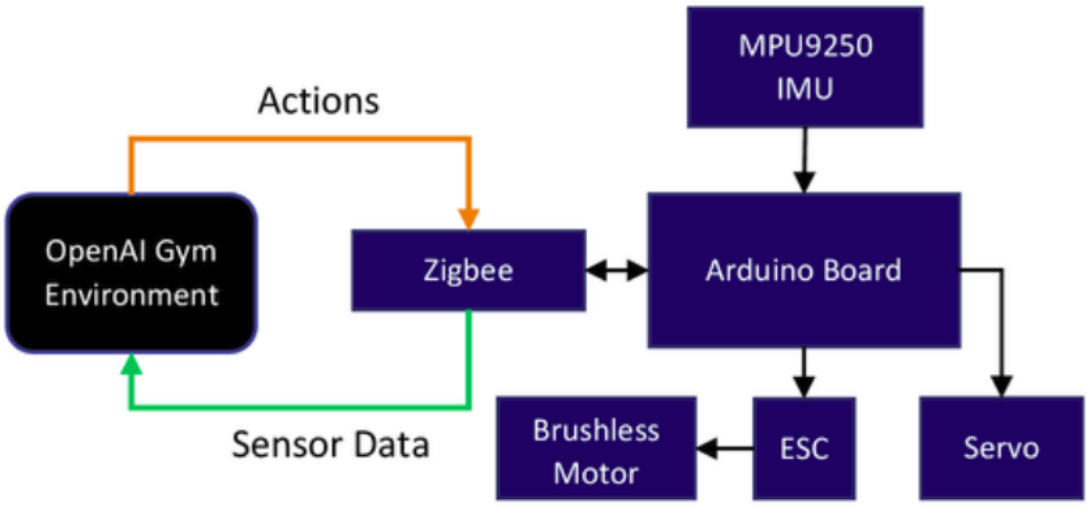


Figure 4: Architecture of RC Car.

The ESC module is connected to the Arduino board, and the throttle actions received by the Arduino board are passed to the ESC to be executed on the motor. The throttle for the motor will be kept constant to simplify the learning problem.

The servo is connected to the Arduino directly, and the steering commands are passed directly to the servo. The servo we use takes a discrete set of integers that determine the angle. The values we will use for the our steering angle is the set [65, 75, 85, 90, 95, 105, 115]. The value 90 means the servo is in the center. Values less than 90 steers the car to the left while values greater than 90 steer it to the right. After experimentation, we found that the minimum value that the servo takes is 65 and the maximum is 115. Any value below or above the limits will damage the structure of the car. Moreover, a decision was reached on using steps of 10 for the values because smaller steps do not cause discernible change in steering angle.

Actions commands from the OpenAI environment are passed to the XBee module, which forwards them to the Arduino board. The board then make the decision to forward it to the appropriate actuator. Similarly, the sensor data is collected from the IMU by the Arduino microcontroller and forwarded to Zigbee module, which forwards it further to the OpenAI environment at 50Hz, a frequency chosen to balance the computation load and human latency perception.

## 3.3 Simulation of environment

Even with the Open AI Gym environment optimally set up, it will take agents thousands of iterations before they are able to figure out the optimal strategy for reward maximization. Running many iterations on the RC car is not feasible for a few obvious reasons:

- The hardware will get worn out quickly
- The battery cannot last for thousands of iterations
- The car may get damaged, since the agent will experiment with random moves in early stages of the algorithm

Hence, to circumvent these problems, we need to test our algorithm in a simulated environment before we can move to the actual environment.

Gazebo is an open-source simulation library which allows us to "rapidly test algorithms, design robots, and train AI system using realistic scenarios". It uses the Open Dynamics Engine (ODE) to provide a robust physics engine, with properties like friction, gravity etc., in a simulated world. It also allows us to easily create a robot that will try to match the behavior of the real RC car. This is crucial, since we want our algorithm (that is trained on the simulation) to work in the real world, not just on the simulation. Along with the robot, we also get some pre-defined sensors that we can add to our simulated robot to easily test out the complete system.

## 3.4 Communication

Open AI Gym and Gazebo need some channel of communication between them so that Gazebo can provide the sensor data needed ((x,y) position, throttle level etc.) to the Gym environment which is used to estimate a reward. The Gym environment also needs to send action commands to the Gazebo simulator so that it can move the car around. We could have used simple forms of communication like sockets, HTTP requests/responses, but this would couple the two components too tightly, making it difficult to switch out Gazebo for the RC car later. Having tightly coupled components also makes it difficult for different team mates to work on different aspects of the project. Hence, it was crucial to have a communication protocol that was as decoupled and asynchronous as possible.

Robot Operating System, or ROS, is one such framework that allows us to set up a decoupled, asynchronous communication system. At its core, ROS provides a messaging interface for different processes (referred to as ROS nodes in ROS nomenclature) to communicate with each other, among other features. ROS provides a queuing mechanism for a ROS node to send a message to another ROS node by posting a message in a ROS queue that can be read by another ROS node, completely asynchronously. ROS nodes do not need to worry about other nodes and how they are implemented, and rather only need to know the address of the ROS queue. By doing this, we can create a completely decoupled communication network, allowing us to work on modules independently, and interchangeably.

In context to our project, the ROS network that will be used to connect the various components is denoted in Figure 5. Black rectangles denote processes running as ROS nodes, while the arrows represent the ROS queues, and the text beside each arrow denotes the message type that will be passed in each queue. This figure represents the entire system architecture. Note that both the RC car and the Gazebo simulator accept the same data, and send the same data. This is what will allow us to use the two interchangeably without affecting the RL agent. From the agent's perspective, it is simply performing actions on a car, and does not care about whether it is a real or simulated one.

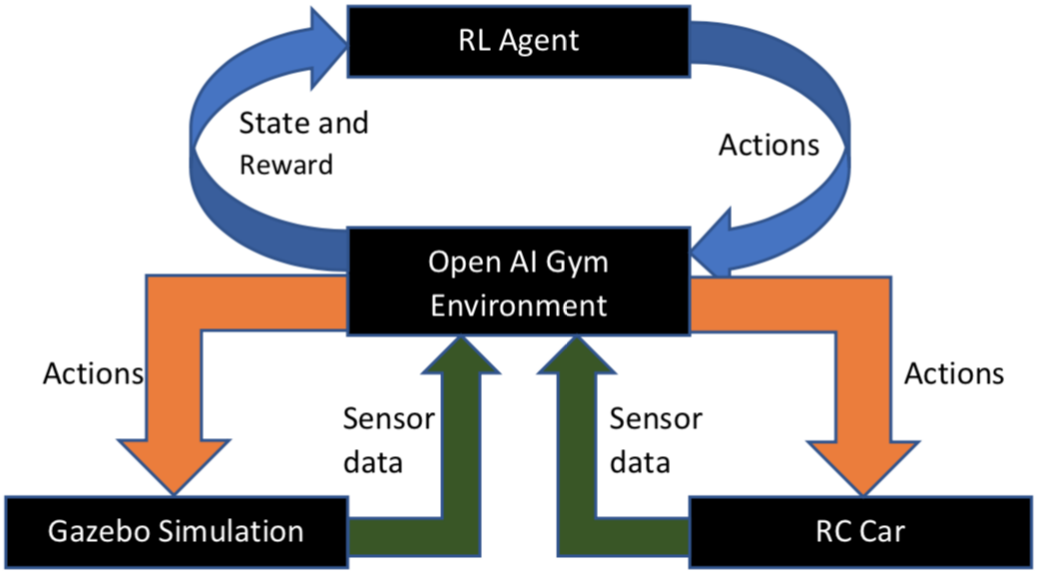


Figure 5: Architecture of ROS Network.

The general flow of information would be:

1. RL Agent sends action to Gym Environment via the Actions ROS queue.
2. Gym Environment forwards action to either the simulator or the car (depending on the settings) using the second Actions ROS queue.
3. The RC car/Gazebo would execute the action, and send its new sensor data on the Sensor data ROS queue.
4. The Gym Environment calculates the reward based on the new state, and sends it to the RL agent using the State and Reward ROS queue.

Now that we have a complete understanding of the various components and how they interact with each other to implement RL, we will now look at the current status of these components.

# 4 Current status and results

We have made considerable progress with our project last semester. We have already assembled our remote controlled car as shown on Figure 6. Before we purchased any of the equipment, the different electronics had to be selected appropriately to find the right balance between size (since we had a limited amount of space on the chassis of the car) and performance. Other than these two factors, we also had to account for the total power consumption of each component and find the right battery that could provide enough voltage to each of the components without damaging the equipment. Electronic components used include the electronic speed controller (ESC), which helps to regulate the amount of current delivered to the motor in the car. This in turn helps to maintain and control the speed of the car. Since the motor has a voltage requirement of 8.4V, the power to it is delivered from a 2S Lithium Polymer (LiPo) battery pack, which is at the center of chassis in Figure 6. Furthermore, a battery elimination circuit (BEC) along with a voltage regulator is used to obtain a 5V supply to power the other electronic components on the car, such as the servo that controls the steering angle and the microcontroller that sends the throttle and steering commands. It is important to note that we have fitted the car with drifting wheels (as shown in the Figure 6) instead of regular rubber RC tires to make it easier for the car to slide and drift.

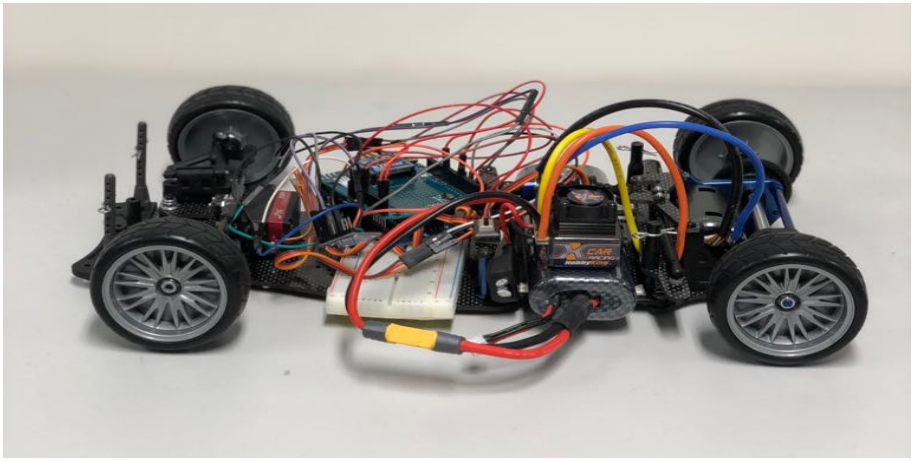

Figure 6: The assembled car.

To ensure that the whole system runs wirelessly, we use two XBee modules which work using the I2C Protocol. One XBee id connected to the microcontroller on the car, while the other XBee is connected to a computer running the main script on Arduino Board using the Servo Library. We mapped the standard WASD keys to be able to control the car using a keyboard, and get the IMU data back from the car every time a new throttle/steering command is sent.

In addition, we have also implemented the simulator in Gazebo to help us model the environment. The first task was to set up the two interfaces needed in order to communicate with the OpenAI Gym environment – the first interface was needed to accept action commands (i.e. steering and throttle commands), while the second was to send sensor data from the RC car to the OpenAI environment. We set up the ROS queues for both these data types, and also created different scripts needed to initialize each queue, the Gazebo ROS node and launch the simulated environment.

Once we had the interfaces set up, the most important task was to model the actual RC car in the environment. Our first version of the simulated car was extremely unstable, as the weight distribution, among other factors, was unequal. To tackle this issue, we measured various physical properties of the car, and input these as variables in the simulator. This stabilized the car considerably. Figure 7 shows the car in the simulator.

Finally, we have completed the implementation of the Double Dueling DQN algorithm described previously. The DQN was implemented in Tensorflow, which is a popular framework for creating deep neural networks.

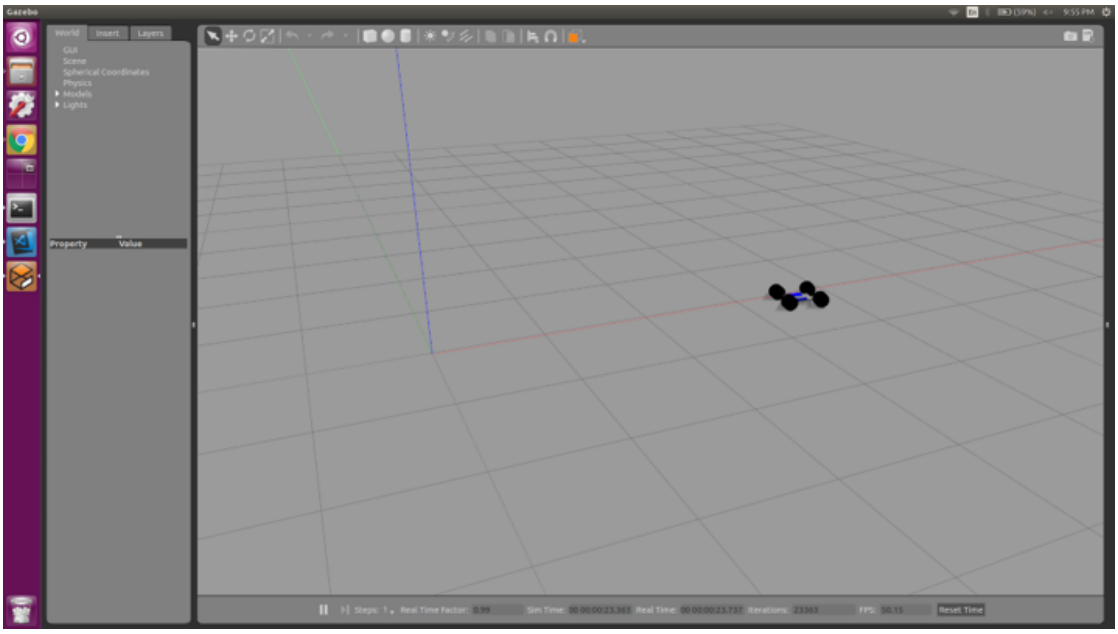

Figure 7: Our simulator with the car in the modeled environment.

To ensure that the implementation of the algorithm was correct, it was used to solve two baseline problems in the RL realm - Mountain Car and Cart Pole. To quantify the performance of the algorithm, we use two metrics - the mean loss and mean reward of the agent over the number of steps of the algorithm. The loss is formulated as the squared difference between the expected action value and the target action value. As the agent learns the expected action values over number of steps, the value converges with the target action value, and hence the loss should decrease over number of steps. The reward is simply the total reward the agent earns over an episode of the algorithm, which should be increasing over the number of steps as the agent learns the appropriate behavior.

Both these trends can be observed for the Mountain Car problem in Figures 8 and 9 below.

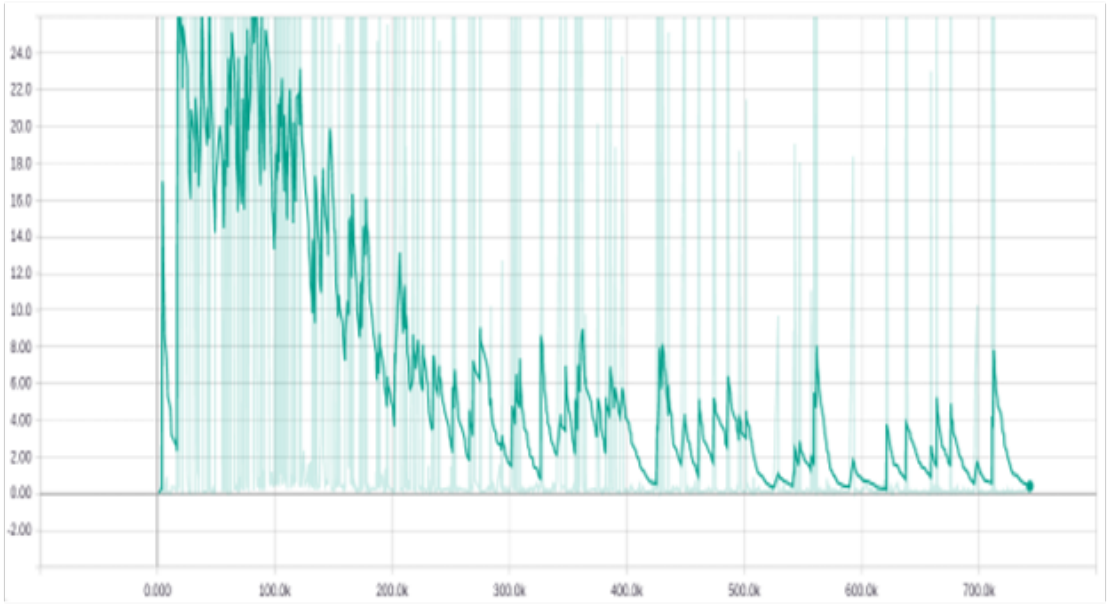


Figure 8: Mean Loss for Mountain Car over number of steps.

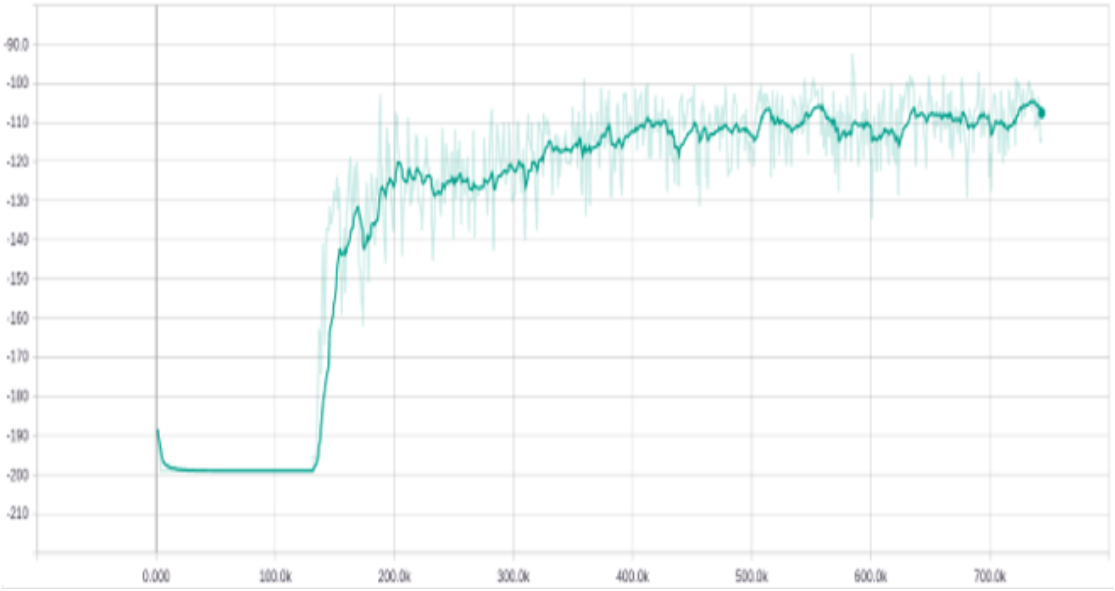


Figure 9: Mean Reward for Mountain Car over number of steps.

Similarly, we can observe these trends for the Cart Pole problem in Figures 10 and 11 below.

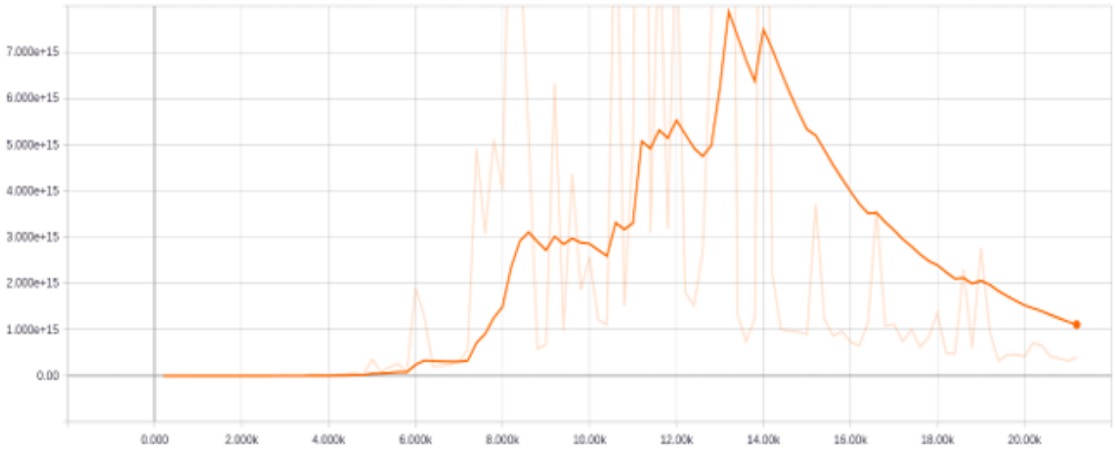


Figure 10: Mean Loss for Cart Pole over number of steps.


Figure 11: Mean Reward for Cart Pole over number of steps.

Finally, once the required behavior was observed in the baseline problems and we were confident our implementation is correct, the first iteration of the double dueling DQN algorithm was executed on the simulated car. Figures 12 and 13 illustrates the trend mean loss and mean reward of the agent after 17 hours of training respectively. Once again, the results are congruent with what we had expected.

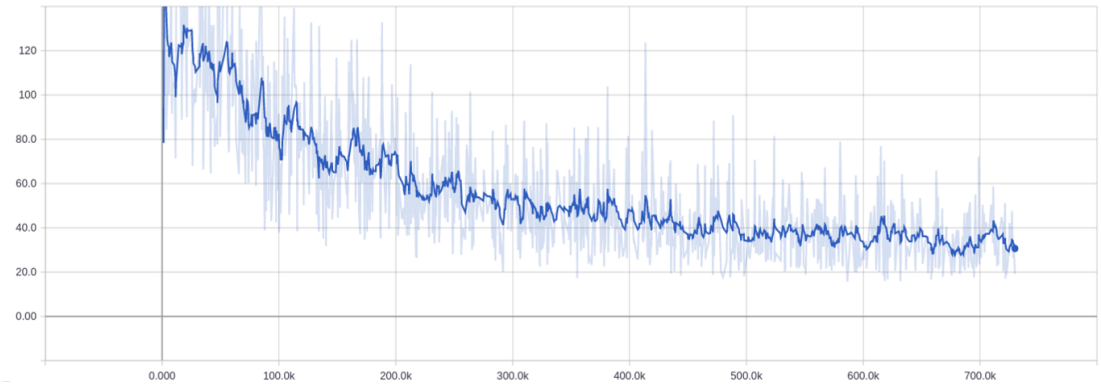


Figure 12: Mean Loss for Simulated Car

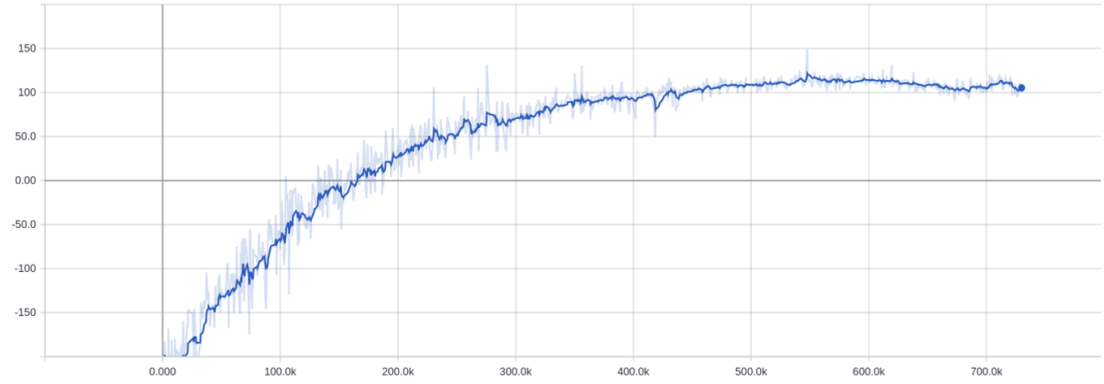


Figure 13: Mean Reward for Simulated Car

# 5 Difficulties encountered

There are a few challenges that we have faced already. Firstly, the simulator that we implemented initially was imperfect and the car in it was behaving strangely, particularly with respect to the amount of effort needed to get it to move in a line. We soon realized the cause for this was that physical quantities, such as the mass of the car, were not modeled perfectly in our simulator, which was causing it to behave differently to what we had expected. Consequently, we introduced such physical parameters into our modeling to resolve the issue.

Secondly, the IMU sensor used was incredibly noisy and the measurement readings we were initially getting were inaccurate. To work around that issue, we passed our data through a low-pass filter to filter out as much of the noise as possible.

# 6 Future plan and contingency plan

Some crucial components of our project needs to be completed, and they will be the focus for the next few months. Firstly, we plan to install an optical sensor on the chassis to get the position of the car, similar to how a mouse gets its position on the computer screen, and we plan to finish this by the end of January. In addition, so far, we have used a simulator for the learning which gave us a model of the optimal drift policy to bootstrap off the learning in the actual car. To validate the model, we will transfer it to the physical RC Car. By transferring the policy onto the physical car, there can be some unforeseen issues and thus, we plan to finish t by the end of February. Algorithms like the double dueling DQN we have already implemented are model free, make no assumption of the dynamics of the system and learn by virtue of trial and error. As a result, they typically require more iterations to converge to an optimal policy. To make our policy model converge in fewer number of iterations, we plan to explore model based approaches as well, like Iterative Linear Quadratic Regulator and Dynamic Differential Programming. Although the dynamics of the system are not explicitly formulated, such algorithms first try to build a model of the dynamics based on interactions of the agent with the environment, which is then used to plan an optimal trajectory. Since such methods first try to model the dynamics instead, the overall policy will converge to an optimal one in much fewer iterations. We plan to finish this by mid-March, and compare the performance of both model-free and model-based approaches in learning an optimal drift policy. Finally, we will continue iterating on the algorithms to get the car to drift in patterns leading up to the final presentation. Table 1 on the next page outlines the future plan.

Since we are dealing with the hardware implementation in our project as well, it is always probable to run into unforeseen issues when transferring the optimal policy to the physical RC car. Thus, it is important to to prepare for them, and we have a contingency plan to tackle two potential problems which we might face. Firstly, it is probable that the optical sensor data will not represent the accurate position of the car. To tackle this issue, we plan to use computer vision and camera triangulation to estimate the position of the car. A tag can be mounted on the car

and a camera can be suspended on the ceiling which can help us with estimating the position of the car. Secondly, the explicit reward function we have used so far might be too simple to encourage sustained circular drift. In such a case, we plan to use inverse reinforcement learning to learn an implicit representation of the reward function instead.

| **Milestone** | **Description** | **Due data** |
|---|---|---|
| M1 | Install an optical sensor onto the chassis of the car to get accurate position data. | 31-01-18 |
| M2 | Transfer the learned optimal policy from simulation to the physical RC car and validate results. | 28-02-18 |
| M3 | Explore model-based approaches to finding the optimal drift policy, like iterative LQR and DDP. | 15-03-18 |
| M4 | Iterate on algorithms and run tests. | 14-04-18 |
| M5 | Final report and Phase 3 deliverable on **April 15, 2018** | 15-04-18 |
| M6 | Final presentation on **April 20, 2018** | 20-04-18 |

Table 1: Detailed future project plan and milestones.

# 7 Conclusion

To summarize, we justified why autonomous drifting cars are important and how drifting can be useful in emergencies to avoid accidents. As we have already discussed, current self driving cars and stability control techniques try to avoid slipping tires and in doing so, restrict the capability of the car. However, we need to exploit the full capability of a car during emergencies. So clearly, having an autonomous drifting car that learns an optimal drift control policy using our methods can help reduce the number of accidents caused by hydroplaning and make roads safer.

Motivated with this intention, we proposed a framework that uses state of the art model-free reinforcement learning algorithms like double dueling deep Q-networks to learn an optimal controller for drifting an RC car to maintain a state of steady circular drift. As discussed earlier, using a model-free approach makes no assumption of the dynamics of the system, which is unknown. Furthermore, as already mentioned, exploring policies by trial and error to find an optimal one in a physical robot like our car is not feasible because it leads to wear of the robot. So, we proposed designing a simulator where our agent explores and learns this optimal controller. Furthermore, we briefly discussed the advantages of Double dueling DQN and Q-learning over other algorithms like PILCO. We also outlined how we used potential based rewards to reward our agent in small quantities along the path of the trajectory we want it to follow instead of a huge reward at the terminal point. This helps to make sure our learning agent learns the behaviour we want.

Next, we explained what we have accomplished so far in terms of assembling the car, implementing the simulator and running training iterations of the algorithm on the simulator. Results from training the double dueling DQN were also presented. As previously discussed, the results look promising and appear to be what we had expected. As for the future plan, we hope refine our implementation of the double dueling DQN and transfer the policy to the RC car by middle of February. In the meantime, we will also try to obtain position data using optical sensors as discussed

previously and incorporate Kalman filters to fuse it with the sensor information from the inertial measurement unit to obtain a better estimate of the car's position and reduce noise. In the months following that, we will explore model-based learning algorithms like iterative LQR and DDP. Although the dynamics of the system are not known, we will build a model to estimate the dynamics instead using function approximators like Gaussian processes and neural nets. We expect model-based algorithms to converge faster because we try to learn the dynamics first. We hope to then compare the two approaches in learning an optimal drift policy before our final submission in April.

# References

[1] S. Saha, P. Schramm, A. Nolan, and J. Hess, "Adverse weather conditions and fatal motor vehicle crashes in the united states, 1994-2012," *Environmental Health*, vol. 15, 2016.

[2] A. T. van Zanten, R. Erhardt, G. Landesfeind, and K. Pfaff, "Vehicle stabilization by the vehicle dynamics control system esp," *IFAC Mechatronic Systems, Darmstadt, Germany*, pp. 95–102, 2000.

[3] J. Ackermann, "Robust control prevents car skidding," *IEEE Control Systems Magazine*, vol. 17, pp. 23–31, 1997.

[4] K. Yoshimoto, H. Tanaka, and S. Kawakami, "Proposal of driver assistance system for recovering vehicle stability from unstable statesby automatic steering," in *Proceedings of the IEEE InternationalVehicle Electronics Conference*, 1999.

[5] A. Hac and M. Bodie, "Improvements in vehicle handling through integrated control of chassis systems," *International Journal of Vehicle Design*, vol. 29, no. 1, 2002.

[6] J. Wei, Y. Zhuoping, and Z. Lijun, "Integrated chassis control system for improving vehicle stability," in *Proceedings of the IEEE Inter- national Conference on Vehicular Electronics and Safety,*, 2006.

[7] A. Trachtler, "Integrated vehicle dynamics control using active brake steering and suspension systems," *International Journal of Vehicle Design*, vol. 36, no. 1, pp. 1–12, 2004.

[8] F. Zhang, J. Gonzales, K. Li, and F. Borrelli, "Autonomous drift cornering with mixed open-loop and closed-loop control," in *Proceedings IFAC World Congress*, 2017.

[9] P. Frère, *Sports Car and Competition Driving*. Bentley, 1969.

[10] E. Velenis, D. Katzourakis, E.Frazzoli, P.Tsiotras, and R.Happee, "Steady-state drifting stabilization of rwd vehicles," *Control Engineering Practice*, vol. 19, 2011.

[11] R. Hindiyeh and J. Gerdes, "A controller framework for autonomous drifting: Design, stability, and experimental validation," *Journal of Dynamic Systems, Measurement, and Control*, vol. 136, 2014.

[12] S.-T. Wu and W.-S. Yao, "Design of a drift assist control system applied to remote control car," *International Journal of Mechanical, Aerospace, Industrial, Mechatronic and Manufacturing Engineering*, vol. 10(8), 2016.

[13] E. Velenis, E. Frazzoli, and P. Tsiotras, "On steady-state cornering equilibria for wheeled vehicles with drift," *Institute of Electrical and Electronics Engineers*, 2009.

[14] R. S. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[15] A. Y. Ng, "Shaping and policy search in reinforcement learning.," *PhD thesis, EECS, University of California, Berkeley*, 2003.

[16] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. arXiv: `1312.5602`. [Online]. Available: `http://arxiv.org/abs/1312.5602`.

[17] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. arXiv: `1509.06461`. [Online]. Available: `http://arxiv.org/abs/1509.06461`.

[18] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015. arXiv: `1511.06581`. [Online]. Available: `http://arxiv.org/abs/1511.06581`.

[19] A. Y. Ng and M. I. Jordan, "Pegasus: A policy search method for large mdps and pomdps.," *In Uncertainty in Artificial Intellicence, Proceedings of Sixteenth Conference*, pp. 406–415, 2000.

[20] M.Deisenroth, D.Fox, and C. Rasmussen, "Gaussian processes for data-efficient learning in robotics and control," *Pattern Analysis and Machine Intelligence, IEEE Transactions*, vol. 99, 2014.

[21] M. Cutler and J. P.How, "Autonomous drifting using simulation-aided reinforcement learning," *2016 IEEE International Conference on Robotics and Automation(ICRA)*, 2016.

[22] C. Rasmussen and C. Williams, *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA, 2006.

[23] C. Atkeson, S. Schaal, and A. Moore, "Locally weighted learning.," *AI Review*, 1997.