Princess Sumaya جامعـــة
University الأميـرة سميّـة
for Technology للتكنولوجيا

Design and Implementation of an Autonomous Parking System using Reinforcement Learning Techniques

By

Ahmad Arrabi & Hamza Daoud

Supervised by

Dr. Amjed Al-Mousa

Submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE
in

COMPUTER ENGINEERING

at

PRINCESS SUMAYA UNIVERSITY FOR TECHNOLOGY

Amman, Jordan

Spring 2021/2022

This is to certify that I have examined

this copy of an engineering documentation by

Ahmad Arrabi & Hamza Daoud

And have found that it is complete and satisfactory in all respect,
And that any and all revisions required by the final Examining Committee have been made

---

Dr. Amjed Al-Mousa

Acknowledgments

The team would like to show their thanks and gratitude to Dr. Amjed Al-Mousa who supervised this work and always provided his mentorship and guidance. Special thanks to Eng. Mohammad Shannak for providing the JetRacer AI kit for testing, without it we could not have completed our tests adequately. We would like to also thank our families and friends who supported us throughout this time, and always kept us motivated to keep working. Note that both team members were hesitant about this idea at first, and neither expected it to work. But after some sleepless nights, and a lot of debugging, we found a way to make it done. So, kudos to us!

Abstract

Recently, the definite results of machine learning methods in autonomous driving applications have become more conspicuous than ever. Specifically, Reinforcement learning (RL) was one of the most prominent adopted approaches to these kinds of problems. A major obstacle in RL is the deployment of the trained models (transfer learning) to reality, where noise and uncertainty are inevitable.

This work provides the design and implementation of an autonomous parking system utilizing RL methods. The system is designed to manage and control a vehicle to park in a specified parking position. It can be considered an application of a sim-to-real problem, as a trained RL-based model should ultimately be transferred to control the vehicle. The simulation environment was developed using OpenAi's 'gym' toolkit in conjunction with the 'PyGame' library for a graphical interface. The system was tested on a prototype using NVIDIA's 'JetRacer' robot. The training was conducted using the Proximal Policy Optimization (PPO) algorithm.

The system utilizes an input coming solely from a top-view camera planted at the top of the parking space (no dependencies would be required from the controlled vehicle). A costume object-detection algorithm was developed to detect the vehicle and extract features from it. The developed algorithm was implemented by subtracting a reference image from the input frame to isolate the vehicle and other objects in the parking. The extracted features represented the state space of the RL model, which are: four relative distances between the vehicle and the parking, rectangular coordinates, and the heading angle of the vehicle. The action space was a discrete one, in which each action represented a combination of speed and steering angle of the vehicle.

The results of the system were successful as the successful parking rate reached 96% with no collisions with any objects. While the line crash percentage did not exceed 9%. The testing was conducted on multiple samples and scenarios of the parking setup.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

With the rise of populations and congestion in urban environments, the need for intelligent solutions became progressively evident. The research focused on developing these intelligent solutions vary from Advanced Driving Assistance Systems (ADAS) [1] to Intelligent Transportation Systems (ITS) [2]. The fundamental purpose of such systems is to provide a safe, convenient, and smooth driving experience for drivers and passengers.

In ADAS, different modules and features are added to vehicles to achieve a safe and uncomplicated driving and parking process. These modules can be applied for different purposes, e.g., sending warning signals to the driver, providing a more efficient braking mechanism, or even in some cases, taking partial control of the vehicle. On the other hand, smart parking systems can be considered a part of ITS, which is a general term for any transportation system that achieves intelligent behavior. Intelligent parking systems usually incorporate different features, e.g., an efficient searching algorithm to search for the best parking location, i.e., a location that minimizes the time and congestion of the parking procedure. Also, navigation systems and path planning algorithms could be adopted.

Artificial Intelligence (AI) and Machine learning (ML) are viable approaches that could be adopted to solve different problems of intelligent parking systems [3]. The developed AI-based designs could generalize solutions and be applied in different situations where rule-based systems and traditional programming methods are inconsistent. The main advantage of these AI-based systems lies in their ability to handle scenarios where uncertainty is pervasive, e.g., autonomous vehicles (both self-driving and self-parking).

Reinforcement Learning (RL) has proven to be a suitable approach for tackling autonomous driving problems [4]. A lot of research is being done in this field and advancements are consistently emerging. A topic that is still under development is transfer learning in RL, which deals with transferring policies learned in simulations to the real world, sometimes referred to as sim-to-real transfer. Most of the research in this field is either concerned purely with simulations or hardware, while a few combined both.

This work provides the design and implementation of a fully automated parking system in which a vehicle can successfully park solely depending on commands (which actions to take) from the parking system.

1.2 Objectives

The objective of this work is to design, implement, and test a prototype of a parking system that can control a robot car and navigate it to park in a predefined parking slot. The navigation process should be as fast as possible and free from any form of collision.

The control scheme of the robot is captured by a pre-trained RL-based model. The training should be performed in a simulated environment that depicts the parking space.

1.3 Design Overview

This section gives a high-level overview of the design to be achieved. The design should be able to achieve all objectives mentioned within the design requirements and constraints that will be specified in sections 1.4 and 1.5.

The parking process begins with the robot being placed in a fixed initial location. The objective, as stated, would be to successfully park the robot in a predefined parking slot. To achieve it, a trained RL-based model would be deployed into the system to take full control of the robot and park it autonomously, without any intervention from the driver.

The proposed autonomous RL-based parking system can be divided into three main sub-systems:

- Hardware
- Simulation environment
- RL model design

Figure 1 illustrates an overall diagram of the parking system, its inputs, outputs, and workflow. Initially, a camera, planted on top of the parking space, captures a top view of the parking, and sends it to the processing unit. From the captured view, the system should send commands and control the car to park successfully.

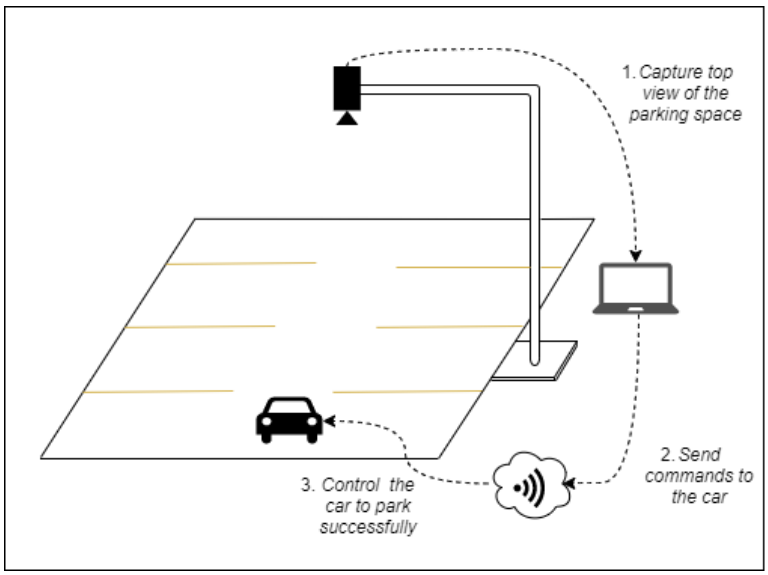

Figure 1: System Overview

1.3.1 Hardware

The first part of the design, hardware, would be composed of a camera, robot car, CPU, and the physical parking setup. This sub-system is responsible for testing the trained RL model on a prototype of a real car. All components shall depict real-life scenarios as much as possible, e.g., the width of the parking slot relative to the car, maximum steering angle, the responsivity of breaks, and speed of the car.

1.3.2 Simulation Environment

Designing the simulated parking environment and building it to be a replica of the parking space are the first steps of the simulation design process. Additionally, choosing the appropriate simulation tool and software is crucial, as the gap between the simulation and hardware should be as little as possible. Any chance of error between them must be mitigated. Thus, the simulated environment must allow us to acquire the same inputs as the hardware setup, the top view of the parking space.

1.3.3 RL Model

The RL model is responsible for taking the best sequence of actions to park the car successfully. The design of such a sub-system would be the process of choosing the state space (inputs), action

space (outputs), algorithms used in training, reward function, and policy function, in addition to all details of the architecture of the agent.

1.4 Design Requirements

There are multiple requirements in the design that shall be considered for a successful parking system. These requirements range from hardware structure and characteristics to simulation efficiency and RL model design. The developed design should obey all requirements mentioned in this section.

Note that this is a design of a prototype of a parking system. Thus, requirements would change and scale up if a commercial system was considered, in addition to added requirements for a more realistic model, e.g., interaction with pedestrians and security-oriented problems.

1.4.1 Hardware Requirements

The hardware requirements are those concerned with all hardware components and their arrangements., e.g., robot car dimensions, parking slot width, and types of sensors used.

The following are the hardware requirements:

1. The system should be able to deploy the trained RL model and control the robot
2. The parking setup should contain multiple spots
3. The width of the parking spots should be at most 150% of the robot's width
4. The robot should be a prototype of a 4-wheeled car
5. The robot's length should be between 15 and 50 cm, with aspect ratios mimicking a real car
6. The system should use a camera, sensors, or a combination of both

1.4.2 Simulation and Model Requirements

All training of the RL model should be done through a simulated environment before hardware implementation. The requirements of these elements (simulation and model) are as follows:

1. The simulation environment should be a replica of the built parking setup
2. The model should be RL-based

3. The physics of the agent should represent the robot, it should move forwards and backward with the ability to turn in both directions
4. The agent should be trained for perpendicular parking only
5. Computer vision algorithms and image processing techniques should be used to process the input image stream

Note that an optional feature that could be added is the ability for the agent to search for an empty parking spot on its own.

1.4.3 Performance Requirements

The expected performance of the designed system is as follows:

1. Collision rate should not exceed 10%
2. The parking time should not exceed 90 seconds

1.5 Design Constraints

Considering this work is only a prototype, its constraints may not impact the design greatly. There are two main constraints in this design, economic and environmental constraints.

1. **Economic:** The cost of the parking setup and robot should not exceed 200 JDs
2. **Environmental:** The robot should be battery operated

1.6 Load Distribution

Table 1: Task distribution of the project

Task	Task Distribution (%)		Level of Completion (%)	Time
	Hamza Daoud	Ahmad Arrabi		
Literature Review	25	75	100	2 Weeks
Overall System Design	50	50	100	2 Weeks
Investigated Hardware Design Options	80	20	100	2 Weeks
Investigated Simulation Design Options	20	80	100	2 Weeks
Hardware Setup Design & Implementation	50	50	100	4 Weeks
Simulation Environment Design & Implementation	20	80	100	4 Weeks
Image Processing	80	20	100	3 Weeks
Object Detection Algorithm	80	20	100	3 Weeks
RL Model Design	40	60	100	5 Weeks
RL Model Training	20	80	100	3 Weeks
Testing & Evaluation	50	50	100	4 Weeks
Hardware Deployment	50	50	100	4 Weeks
Documentation	25	75	100	Throughout the project

1.7 Organization of the Document

The document is organized as follows: Chapter 2 presents the needed theoretical background to fully comprehend and develop the designed system. Additionally, a literature review of different approaches and methods was conducted in the same chapter. Chapter 3 describes the design process, from the analysis of all requirements and constraints to the comparison of different design approaches that could be implemented. Then, ultimately developing a complete design to solve the proposed problem. Moving on to chapter 4, which presents the findings of the work and all quantitative analyses of the results. Also, examine whether the requirements were fully fulfilled or not. Chapter 5 is the final chapter and it concludes all that was discussed and tested, in addition to any recommendations or propositions for future works.

2 Background and Literature Review

2.1 Background

2.1.1 Reinforcement Learning

Reinforcement learning is a branch of machine learning that is concerned with sequential decision-making problems. Its main objective is to find the best sequence of actions an agent should take in order to reach a goal state (objective). The main difference between RL and other machine learning techniques; supervised and unsupervised learning, is the way the training data is handled. In conventional settings, a dataset is developed independently from the algorithms. Whereas in RL, data is generated and collected from a repetitive process in which an agent interacts with its environment to produce different features, e.g., reward, penalty, and state action pairs. These generated features are then given as inputs for the RL model to train on.

In an abstract view, as illustrated in Figure 2, any RL training process can be described as follows:

1. An agent takes actions (based on a given policy), thus, interacting with the environment
2. According to the action taken, the environment generates a new state and a reward as feedback for the agent. The reward could positive or negative (penalty)
3. Based on the feedback, the agent adjusts its parameters (policy) to maximize its rewards and minimize its penalties
4. Repeat from step 1 to reach a given reward value or to iterate through a defined number of episodes

The preceding sequence would ultimately enable the agent to take the correct actions, actions that maximize rewards, given its state, hence, learning.

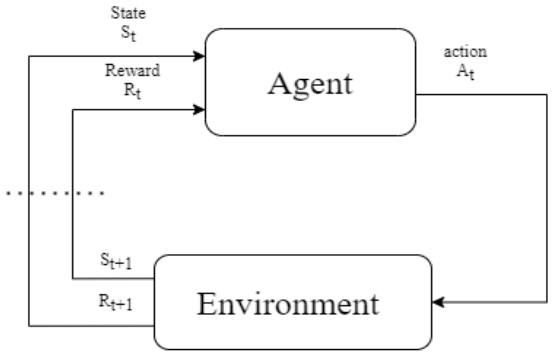

Figure 2: Abstract RL model

The jargon of RL can be opaque at first. Thus, an overview of different terms of RL that will be used throughout this work is presented.

1. Agent

The agent is the object that interacts with the environment, it generates actions, and acquires states. An agent is composed of both a policy and a RL algorithm. Examples of agents are robots, cars, players in a game, or a trading system in a financial trading environment.

2. Environment

The environment is all of what is outside the agent. It is responsible for generating rewards and states based on the actions taken by the agent.

3. State space

A state space is all possible states an environment can generate. Meaning that any possible state is a subset of the state space. E.g., the state space of a parking environment could be the cartesian coordinates of the vehicle (agent).

4. Action space

An action space is all possible actions an agent could take. Meaning that any possible action is a subset of the action space. E.g., the action space of a parking environment could be the acceleration and steering angle of the vehicle (agent).

5. Policy

A policy is a mapping between the state space and the action space of the agent. Given a set of observations (state) the policy determines what actions to take. The policy can be described by a

mathematical function, whether it was a probability distribution or a nonlinear function approximation method like a neural network (deep-RL).

At the initial stages of training, the policy would be inaccurate leading to little reward, this is where RL algorithms come in. The algorithm is responsible for updating the policy to maximize the cumulative rewards. In the setting of policies as Artificial Neural Networks (ANN), the RL algorithm would be responsible for training the ANN.

This way the overall goal of the agent is to use RL algorithms to update its policy as it interacts with the environment. So that ultimately, given any state, the agent would always take the best action. This setting can be illustrated in Figure 3.

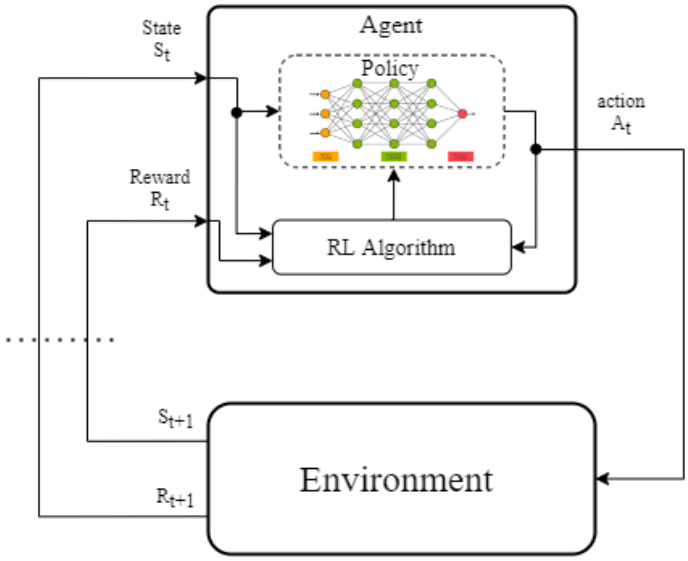

Figure 3: Detailed RL model

6. Reward

A reward is a value, positive or negative, that depicts how well the agent performed given its state. It is a fundamental metric in RL as the model would be reinforcing good actions and preventing bad ones. I.e., updating the policy to achieve the largest accumulated reward over its sequence of actions.

7. Episode

An episode is the combination of all states that come in between the initial and terminal state. It is where the training would be accomplished.

2.2 Literature Review

2.2.1 Autonomous Parking Systems

- Path Following Approaches

The work in [5] was a comprehensive thesis that studied different RL algorithms to park a car in different difficult situations. A key remark that we must take into account is that this work focused on developing a methodology of learning how to guide a vehicle towards a parking target following a predetermined path. Thus, it was a path following problem using RL.

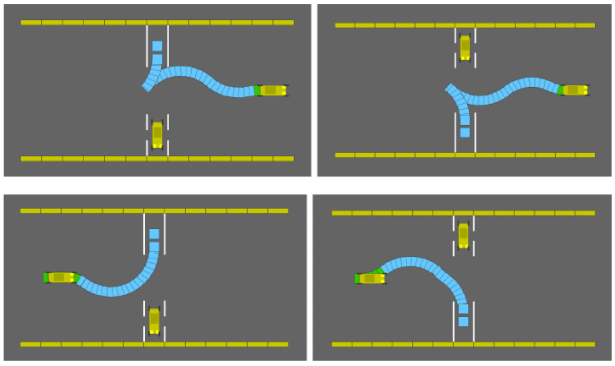

Figure 4: Path following method for vehicle parking. Reprinted from [5].

The chosen simulation environment was the autonomous driving simulation tool Highway-env, which is based on OpenAI's gym environment toolkit. Figure 4 shows a sample of the environment.

The developed state space was a combination of the agent's position (coordinates) along with its direction and how far it is from the planned path. The action space was a continuous one represented by the speed and steering angle of the vehicle.

The methods of path planning and following were vulnerable and showed inaccuracies in certain scenarios, e.g., when a path overlaps with itself so the agent would not know what line to follow.

- Reinforcement Learning-Based Approaches

- Continuous Action Space

The literature in [6] proposed an RL-based approach for automated parking. A simulation environment using 'preScan' was developed first, then, deployed on a real vehicle (a MATLAB-based kinematic model was constructed). Their method included two parts, parking slot tracking,

using computer vision, and RL-based vehicle control. Parking slot tracking was used to provide a continuous position of the parking slot for the RL model, and the model controlled the steering wheel angle of the vehicle. An overview of the system is shown in Figure 5.

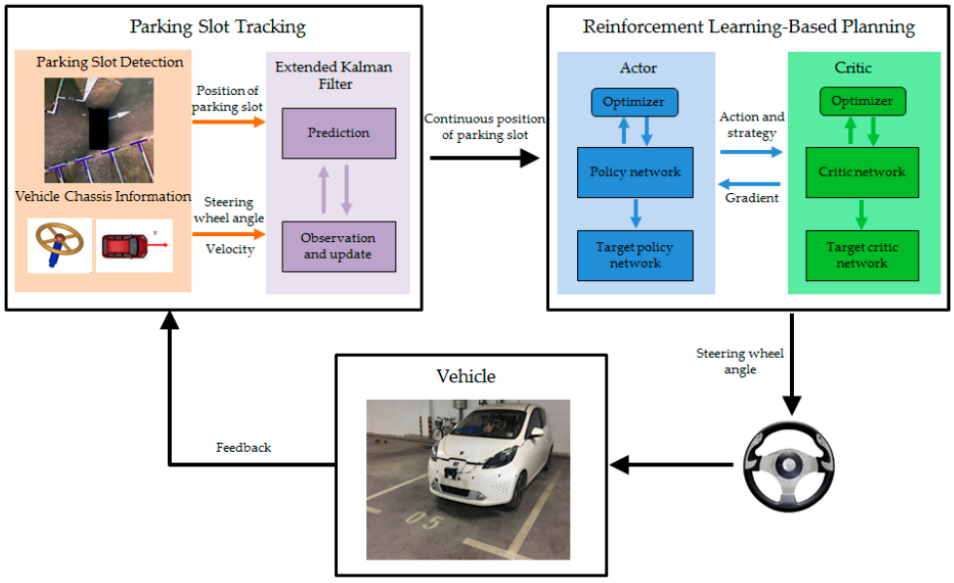

Figure 5: Overview of the RL-based end-to-end parking met. Reprinted from [6].

The parking slot tracking system included four fisheye cameras planted in the front, rear, left, and right sides of the vehicle. The four angles were used to generate a surround (top) view of the parking, as illustrated in Figure 6. From the generated surround view, the positions (coordinates) of the edges of the parking slot were extracted. These four coordinates were then used as an input to the actor neural network. The network is illustrated in Figure 7.

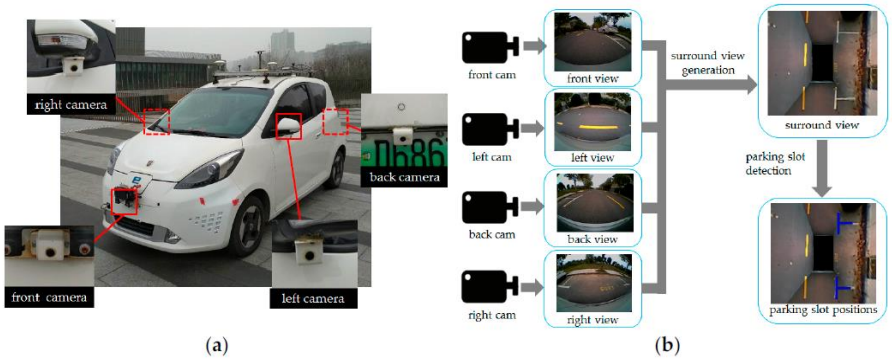

Figure 6: Surround view parking slot detection system: (a) Test vehicle and camera installation location. (b) Surround-view generation and parking detection. Reprinted from [6].

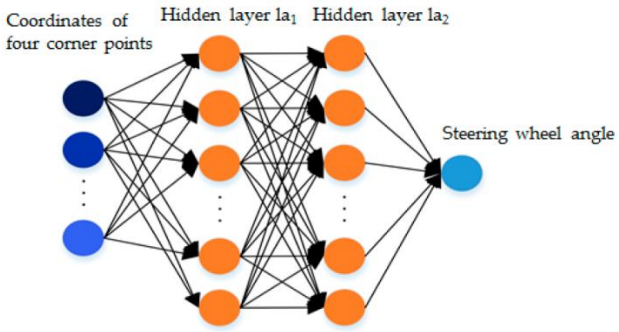

Figure 7: Actor network. Reprinted from [6].

- Continuous Action Space

Another work on RL-based methods is presented in [7], where a framework for self-parking simulation was proposed. The framework employs a Proximal Policy Optimization (PPO) algorithm for generating a model to park a vehicle in moderately complex parking scenarios. The simulated environment utilized unity ML agents to represent the vehicles (Figure 8 and Figure 9) in addition to the 'gym' library to train the RL model.

The agent was a car with 8 sensors attached to its sides. The state of the agent was a total of 25 values captured from the parking lot divided as follows:

- Sixteen values from the 8 sensors, each sending 2 values. The first one was a Boolean value indicating the detection of the object and the second one was the distance of a detected object to the sensor.
- Six values represent the agent's position, the target parking slot position, and the displacement between agent and target.
- Three values were the car orientations in a degree angle, the linear velocity, and the angular velocity of the agent.

The action space was 2 real numbers, the torque applied to the car, and the steering angle. Thus, the action space was continuous.

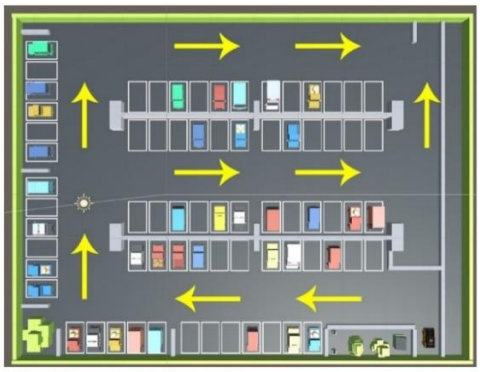

Figure 8: The parking lot environment in unity engine. Reprinted from [7].

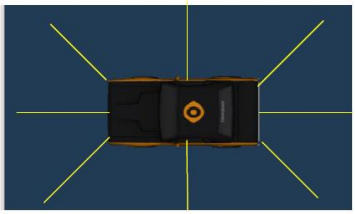

Figure 9: Agent sensor outline. Reprinted from [7].

Table 2 shows the rewards scheme for the RL training.

Table 2: Reward and penalty scheme. Reprinted from [7].

Reward function	**Value**
Timeout (40 seconds)	-1000
Parking Successfully	+5000
Parking Successfully (Orientation config)	+950 +(45 per degree angle)
Getting closer to the target parking slot compare to previous step	+50 per unit distance
Collision with Obstacles	-300 per hit
Every Step	-1
Stay in Collision	-1 per step

- Discrete Action Space (Q-learning)

Furthermore, the author in [8] implemented the Q-learning algorithm in a simulated 2D game-like environment, the environment was designed as a top view of the parking space, as depicted in Figure 10. The environment consisted of three main elements: vehicle, parking slot (target), and obstacles (walls). The agent (vehicle) and the obstacles were represented as polygons, and

collisions were modeled geometrically by partial overlapping of the car polygon and any obstacle. The work only considered fixed obstacles, no dynamic obstacles were implemented (ones that represent humans and a more realistic environment).

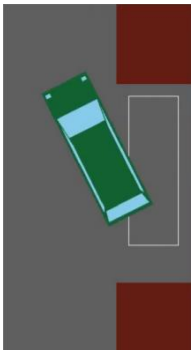

Figure 10: Simulated environment. Reprinted from [8].

The state space was finite and discrete, represented by the position and orientation of the car, reaching a total of 54000 states. The action space was the finite set of maneuvers to move the car (forwards/backward left/right) reaching 30 actions, as in Figure 11.

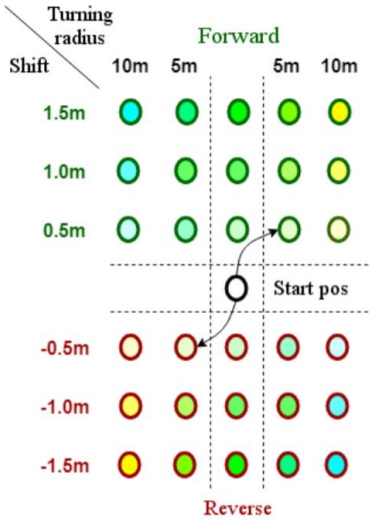

Figure 11: Action space. Reprinted from [8].

The rewarding policy scheme was as follows:

- Action leads to the final target (+1000)
- Collision (-200)
- After each step (-5)

The success rate of the parking reached 81% after 1,134,000 episodes and converged to reach a perfect score of 100% after 16,200,000 episodes.

This work can be considered similar to our proposed approach due to the analogy of the simulated environments; both are top views of a parking space. The differences would be evident in the choice of state and action spaces.

2.2.2 Automated Control using Reinforcement Learning

One of the first approaches to utilize raw visual input data to control a vehicle was the paper represented in [9]. Even though the problem was not necessarily concerned with vehicle parking, the control method of the vehicle could be adopted.

This work proposed an architecture to use raw visual data as input to a RL model. The system autonomously learns how to extract relevant information out of the high-dimensional stream of images, which allows it to learn a control policy for a racing slot car that achieves the learning goal of driving the car as fast as possible without crashing. The significance of this method relies on the fact that no prior information about the dynamics of the vehicle was provided, and no computer vision methods were applied to extract any information from the input images. An overview of the system is shown in Figure 12 and Figure 13.

Figure 12: Overview of the system. Reprinted from [9].

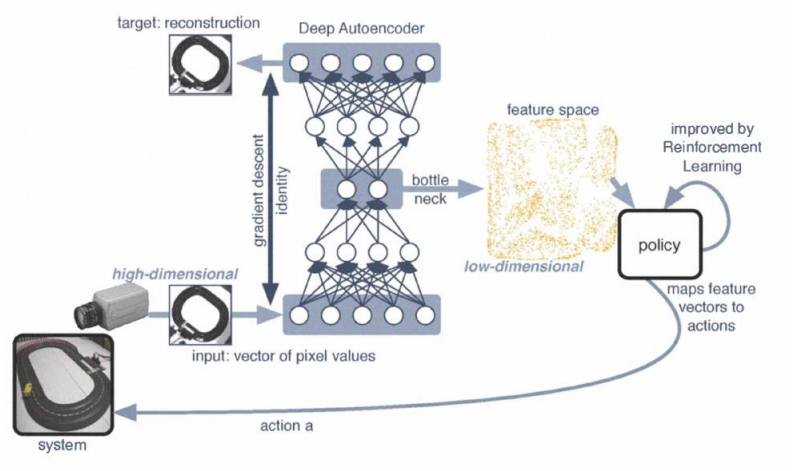

Figure 13: System architecture. Reprinted from [9].

The deep encoder neural network architecture was adopted to extract the feature space from the high-dimensional input images. The state space of the RL model is the low-dimensional feature space, and the action space is a voltage value that controls the slot car. The input image size was 52x80 pixels with a total of 7000 training images (for the encoder network) and the control decisions of the system were four decisions per second (image frequency of the camera).

2.2.3 Simulation to Reality Transfer in Reinforcement Learning

Transferring a learned policy from simulation to reality is not a simple process as it may lead to inaccuracies and degradation of the performance of the agent. The source of this degradation comes from the gap and differences between simulations and reality, i.e., it is hard to exactly model the randomness and uncertainty of reality. A systematic survey of sim-to-real transfer learning in RL was undertaken in [10], where multiple robotics projects and papers were compared and analyzed.

A variety of simulations were used, the most prominent being ROS, PyBullet, and MuJoCo. These simulations are 3D libraries and tools that provide accurate models that represent both the environment and the robot.

The main conclusions of this work are the methods of sim-to-real transfer, which can be described in Table 3.

Table 3: Methods of sim-to-real transfer learning

Method	Description
Zero-shot (direct) Transfer	To build a realistic simulator so that the model can be directly applied in real-world settings
System Identification	To build a precise mathematical model for a physical system, can be a meticulously exhausting process
Domain Randomization Methods	Instead of building a detailed model of the robot, a highly randomized simulation could be built to obtain a distribution close to real-world data
Domain Adaptation Methods	This method attempts to extract features from both simulation and reality to obtain a unified feature space to train the RL agent
Learning with Disturbances	Adding noise in the training process of the model, e.g., noisy rewards
Simulation Environments	To carefully choose the simulation environment and to assure that the agent's specifications match the real robot

3 Design

3.1 Design Requirements

3.1.1 Hardware Requirements

1. The system should be able to deploy the trained RL model and control the robot
2. The parking setup should contain multiple spots
3. The width of the parking spots should be at most 150% of the robot's width
4. The robot should be a prototype of 4-wheeled car
5. The robot's length should be between 15 and 50 cm, with aspect ratios mimicking a real car
6. The system should use a camera, sensors, or a combination of both

3.1.2 Simulation and Model Requirements

1. The simulation environment should be a replica of the built parking setup
2. The model should be RL-based
3. The physics of the agent should represent the robot, it should move forwards and backward with the ability to turn in both directions
4. The agent should be trained for perpendicular parking only
5. Computer vision algorithms and image processing techniques should be used to process the input image stream

3.1.3 Performance Requirements

3. Collision rate should not exceed 10%
4. The parking time should not exceed 90 seconds

3.2 Analysis of Design Requirements

The design requirements were divided into 3 main parts: hardware, software, and performance requirements. The hardware requirements mainly tackle the practicality of the system, i.e., the efficiency of the designed prototype. While the software requirements are the ones related to the RL model and simulation part. These focus on how well the simulation represents the parking setup and reality. The performance requirements are related to the output of the system, i.e., how well it acts in different test cases. Also, how stable and reliable the system is.

3.2.1 Hardware Requirements

The hardware requirements are the ones considering the physical prototype implementation of the system, where the testing stage would be applied. Given that the system is RL-based, the hardware must support such models. I.e., The system should allow the transfer from simulation to reality. Meaning that any observations taken in the simulation should be recreated from the hardware components used, e.g., cameras or sensors. Also, the control scheme of the agent should be mapped accurately with the robot car.

The parking setup would depict a real parking scenery, having multiple spots, each with a width not exceeding 150% of the car's width. Regarding the robot's characteristics, a 4-wheeled car with dimensions similar to a real vehicle should be used. The steering mechanism of the robot should be given careful consideration as it is needed to turn in a sensible way. E.g., no rotation around the axis is allowed, nor an unrealistic steering angle above 45°.

The components used to feed inputs to the system could be a combination of cameras and sensors. The main inputs to the RL model would be extracted features from a top view image of the parking space, so there is no need for sensors at this moment. Although the addition of sensors could assist in improving the results of the system, their modeling on the simulation would be prone to errors and could lead to inconsistencies between the simulation and hardware. Thus, the use of sensors could be adopted to handle some features in the parking system rather than feeding inputs to the model that controls the vehicle. E.g., a sensor to assert the starting position of the car.

3.2.2 Simulation and Model Requirements

A fundamental part of the system is to build an accurate, reliable, and realistic parking environment for the development of the RL model. The environment should allow the training and development

of RL algorithms with customizable features. The design of problem-specific observation and action spaces is key, with the ability to change and build a parking setup as desired.

The environment should also allow for the design of different reward systems, which is a fundamental step in the learning process. Different reward systems should be proposed and tested as they are responsible for developing the behavior of the agent enabling it to reach its objective.

The physics of the agent should replicate the real vehicle as much as possible, this similarity would minimize the error between the real vehicle and the agent.

The objective of the RL model is to park the agent in a perpendicular way only. All training will be done for this objective. Future work or extra features could be added to the system to allow different kinds of parking.

3.2.3 Performance Requirements

The performance requirement of the parking system is to park successfully and fast. A successful parking would be considered a case where the vehicle parks without any overlap with the lines. The ideal performance would be to have a 0% collision rate. The minimum parking time is 90 seconds, which means that the responsiveness of the system is essential. The vehicle should respond to the actions taken from the model with minimum delay, in addition to extracting features from the input image rapidly. Multiple schemes of adjusting the system's overall speed could be considered, e.g., the system's clock would be as the slowest component's clock.

3.3 Analysis of Design Constraints

The cost constraint of the system would impact the type of components used in the prototype setup. Different alternatives could be considered but they all need to adhere to the maximum cost and produce appropriate results.

Regarding the environmental cost, the robot needs to be battery operated. Meaning that the power consumption of the parking process should not be high. Considering the parking time is at most 90 seconds, this point may not be applicable.

3.4 Different Designs Approaches/Choices

The process of selecting a suitable and efficient design is a meticulous one, from choosing hardware components and structure to simulation software and tools. There were multiple options

to build the parking system, each having its benefits and shortcomings. This section provides an analysis of different options faced in the design phase.

3.4.1 Simulation Environments

An essential feature in any simulation environment is its ability to be modified and built into a replica of the physical parking model. The comparison of different software and tools will be based on multiple criteria. E.g., the cost of the software or if the tool allows building environments from scratch, in addition to other points that will be discussed at the end of the section.

1. PyGame with gym

Gym is a python library developed by OpenAi to implement and develop RL algorithms [11]. This toolkit makes no assumptions about the structure of an agent, making it compatible with various applications and domains. I.e., you can design your own agent alongside its environment purely based on the problem at hand, and gym will allow the implementation of RL algorithms to provide solutions. OpenAi's motivation for this work was to provide a suitable benchmark for RL algorithms and applications.

Gym can be integrated with many third-party environments to produce RL-based solutions in different areas of expertise. A library that would be suitable for this work is PyGame, a python-based library that allows the design and simulation of 2D games. The flexibility of modifying the environment gives this toolbox a great advantage, as it would allow us to mimic the hardware setup as much as possible. The correspondence between the physical parking and the simulation (game) would be a valuable step toward diminishing the error when transferring the trained model into hardware. I.e., readings in simulation and hardware will be close.

Furthermore, not everything could be built from scratch, as a lot of open-source environments and car models exist already. All that needs to be done is to modify these models to represent the designed system, e.g., number of parking locations, physics of the robot, or dimensions of the parking slots.

An example of a prebuilt environment is 'Highway-env' [12]. There are many inherited environments from it, like the parking environment represented in Figure 14. It provides a simple simulation of a parking scenario where a car's objective is to park in the green box (arbitrary parking slot). Both discrete and continuous action spaces exist for the car agent.

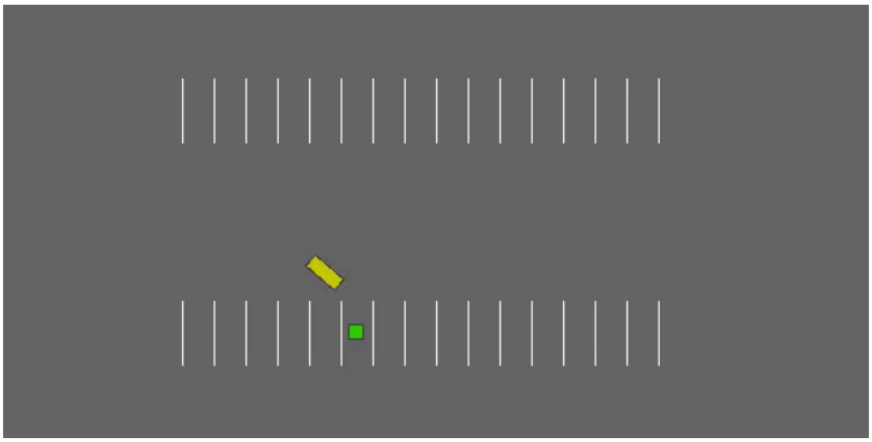

Figure 14: Parking environment of 'Highway-env'. Reprinted from [12].

With some modification to this parking environment, another could be developed to match our parking system. Thus, allowing a successful training and deployment.

A disadvantage of this parking environment is that it may be too simplified, i.e., the physics of the agent is not realistic. However, this shortcoming can be fixed by a testing process on the physical robot where its physics is modeled and built into the simulation.

2. MATLAB

MATLAB is a powerful matrix-based numeric computing environment developed by MathWorks. MATLAB provides a variety of libraries that include ML applications and simulations, in addition to handy toolboxes in various domains. Two possible tools that can be applied in this work are the reinforcement learning and the automated driving toolboxes.

The main advantage of using MATLAB would be the reinforcement learning toolbox, which provides many prebuild ready-to-use RL algorithms, e.g., DQN, PPO, SAC, and DDPG [13]. It also offers the capability of creating agents and environments either from scratch or using built-in ones from the toolbox.

Another toolbox that must be taken into consideration is the automated driving toolbox [14]. It provides numerous algorithms and sensors for testing advanced driver-assistance systems (ADAS). Different parking scenarios can be created using multiple possible visualization simulations (Unreal Engine and Cuboid Driving Simulations).

The main challenge with this approach is to be able to connect these two toolboxes to build the parking system with the RL model. Also, the method of controlling vehicles in the automated

driving toolbox is by collecting data through checkpoints, or by planning a path with checkpoints embedded within it [15], as can be seen in Figure 15. This method may be not applicable when deploying the trained model on hardware.

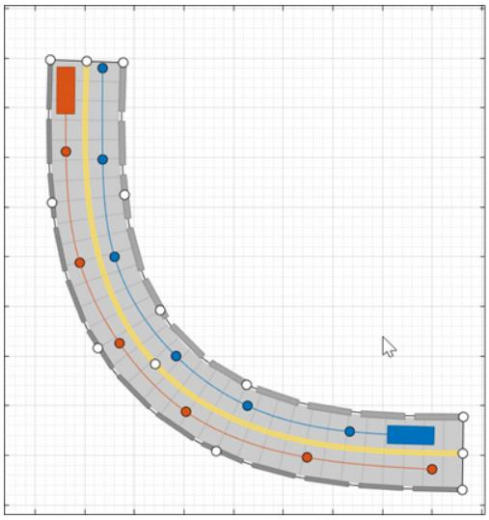

Figure 15: MATLAB Autonomous driving simulation. Reprinted from [14].

Another challenge would be the financial cost of using such tools, even though students have a discount on MATLAB, the cost would be a big portion of the stated cost of the project. Thus, using different software is preferred.

3. Unity

Unity is a game design engine that provides both 2D and 3D simulations. Recently, it is being applied in different domains, e.g., industrial robotics, intelligent agents, and RL. The intersection between game design tools and RL lies in the ability of simulating custom-built environments. Unity supports full flexibility in the design of both the environment and the agent. Just like PyGame, the flexibility in the environment design would allow us to develop a close representation of the hardware setup.

Unity provides a toolkit specialized in RL agents design, the 'Unity Machine Learning Agents' toolkit [16], shown in Figure 16. This toolkit includes many pre-built agents in addition to multiple helpful features that aid the training process, e.g., the variety of ready-to-use training algorithms and the diverse neural network tuning features.

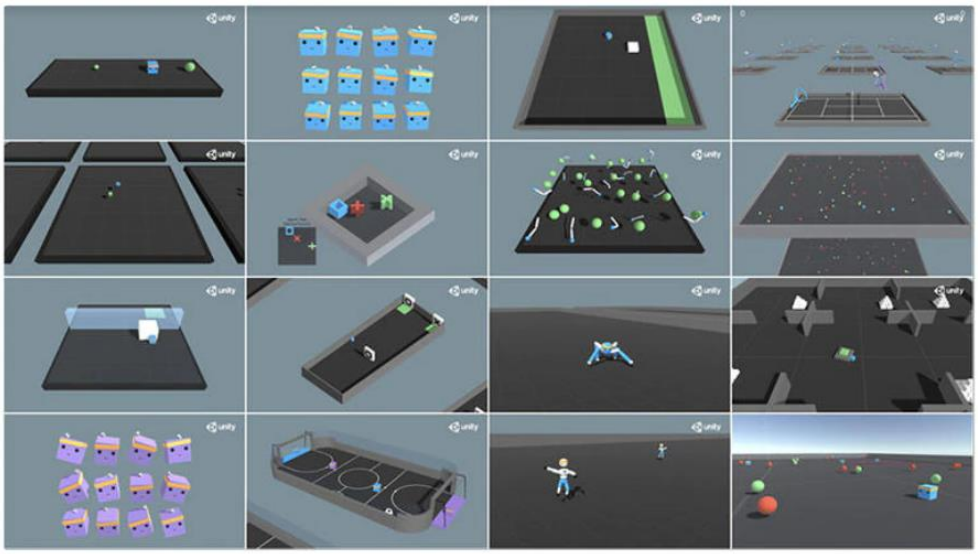

Figure 16: Unity ML agents. Reprinted from [16].

Unity ML agents can be thought of as the gym library mentioned earlier, a tool that enables the implementation of RL. First, a scene is built (2D or 3D) with all entities needed, e.g., characters, vehicles, parking spots, and lights. Then, using the 'ML agents' toolbox, all characteristics of the RL model's environment are specified. E.g., state space, action space, agent, reward function, policy, training algorithms, and step.

4. Carla Driving Simulator

Carla is an open-source urban driving simulator targeted for researchers. Multiple autonomous driving algorithms were developed and tested on it [17]. Carla is most suitable for simulating real-life scenarios, e.g., interacting with pedestrians, traffic lights, intersections, or anything concerning controlling a vehicle in a city.

Carla's environment modification is concerned with the design of the scenery in the simulation, e.g., weather conditions, city design, and the number of pedestrians on the street, as Figure 17 depicts. Also, all input to any model is solely dependent on sensor or camera readings from the vehicle. The control scheme of the vehicle is composed of adjusting the throttle, steering angle, brake, handbrake, reverse option, and gear of the car. This fixed model of the car generates a constraint of developing a hardware model that obeys these control rules, which may lead to a more complex design.

Figure 17: Carla simulator environment alteration. Reprinted from [17].

Regarding the cost of Carla, as mentioned, it is a free open-source tool. However, it is only compatible with Linux-based systems, which may lead to unnecessary complications in preparing the appropriate devices for training.

5. ROS

Robot Operating System (ROS) is a 3D simulator focused on robotics applications and research [18]. ROS allows the creation of any design of a robot taking all details into account, e.g., accuracy of sensor readings and motor operations. This flexibility in design makes ROS very compatible and more convenient to be applied on hardware. Different parking environment designs can also be created from scratch using ROS, as shown in Figure 18.

Figure 18: ROS simulator parking environment example. Reprinted from [19].

ROS can be integrated with the gym library to develop and train RL algorithms. This integration comes with a heavy computational cost, as both ROS and gym need a dedicated GPU to work properly. Having them both operating at the same time on the same GPU may be computationally expensive, which in turn leads to slow training time and rendering problems in the simulation.

Akin to Carla, ROS only operates on Linux-based operating systems. Thus, the same problems stated above would be faced if ROS was used.

An important note that must be considered is the fact that ROS is a robotics design tool. So, intuitively, it would be more fit for a robotics-centered problem. Our work is concerned with the design of a complete parking system, not just the robot car. Different parking elements may be needed for analysis in the future. Ultimately, the key part of choosing the simulation tool is in the process of building the environment, not the robot.

Table 4: Simulation environments comparison

Feature	PyGame	MATLAB	Unity	Carla	ROS
Create custom environments	Yes	Yes	Yes	No	Yes
Predefined RL algorithms	Can be linked with gym	Yes	Yes	Can be linked with gym	Can be linked with gym
Prebuilt Car agent	Yes	No	Yes	Yes	No
Observation space possible	2D top-view image or any defined element in simulation	2D top-view image or sensors	Sensors or any defined element in simulation	Sensors and cameras	Sensors and cameras
Car control	Depends on agent design	Through checkpoints	Depends on agent design	Built-in mechanism	Depends on the robot design
Compatibility with hardware	Mapping between agent and hardware must be considered	A physical model of the robot can be created	Mapping between agent and hardware must be considered	Hardware must be built based on Carla vehicle	compatible with hardware, accurate robot design can be built
Cost	None	29.0$ Per student	None	Only operates on Linux operating system	Only operates on Linux operating system

A comparison between the considered simulation options is presented in Table 4. Overall, all environments have their pros and cons. Some have financial costs, while others only work on specific devices with certain requirements, and a few are computationally expensive.

The most important feature in this work is the ability to design and to have full control over the environment, as it would lead to the closest representation of the hardware model. Both PyGame and Unity provide these components, but the first would be scripted in Python while the latter in C#. The group's expertise is Python-oriented and using C# would need a learning curve that could take time. Thus, concluded from all advantages of PyGame mentioned in this section, it will be the environment chosen to hold the training of the RL model.

3.4.2 Hardware Setup

The essential requirements for the hardware part are the following two components, a camera and the robot that represents the vehicle.

1. Camera

The main 3 features that should be in the camera are its wide-angle, good quality, and direct interface. Cameras usually vary in their angle of view, to detect more parking spots the used camera should be wide not a telephoto. Figure 19 shows the different angles of view in cameras.

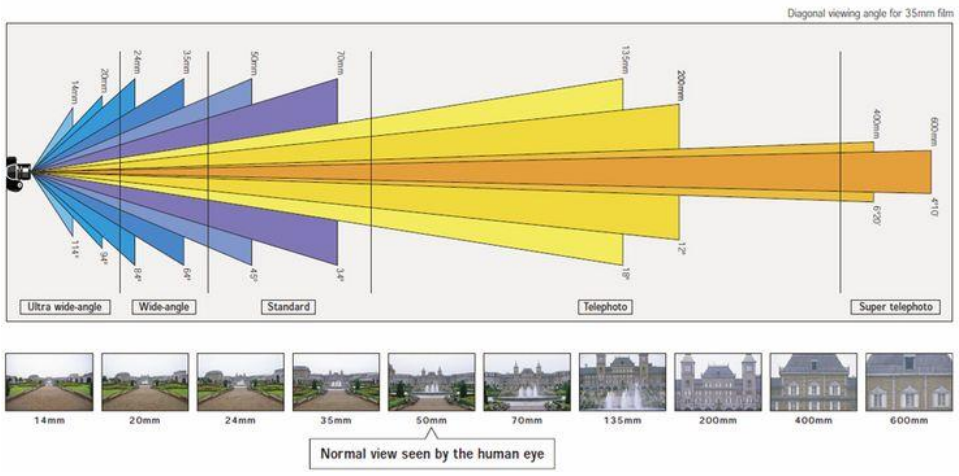

Figure 19: Different angles of view

Regarding the image quality, the used camera should be able to provide the system with a good quality image as a bad one would corrupt the processing part. This noise results in a system not being able to provide reliable data. Also, the camera interface with the server is a very crucial part

of the decision. As there are cameras hard to interface like CCTV cameras, while others are easier to interface like USB webcams.

The camera options were a webcam, Depth camera, or a smartphone. Table 5 compares the 4 options analyzed.

Table 5: Camera options comparison

Feature	Jelly Comp Webcam	Logitech C525 Webcam	Smartphone	Depth Camera
Angle of view	Wide	Flat	Wide	Wide
Quality	Poor quality	High quality	High quality	High quality
Interfacing	Directly connected with a USB cable	Directly connected with a USB cable	Directly connected with a USB cable but needs an external software	Directly connected with a USB cable
Cost	Low	Low	Low	High

Ultimately, the smartphone was the option chosen based on the previous table, as it was the best option that guarantees optimal results.

2. Robot

The main features that influence the decision of choosing a robot are WIFI connectivity and possessing an accurate steering mechanism. The robot should have WIFI connectivity so it can receive the commands from the server (will be discussed in the coming sections). Also, it should have a steering mechanism as seen in Figure 20, as this mechanism represents a real vehicle. If any other mechanism of turning was used it would be dependent on the rotation direction of the motors, which is unrealistic and would be conflicted with the design requirements.

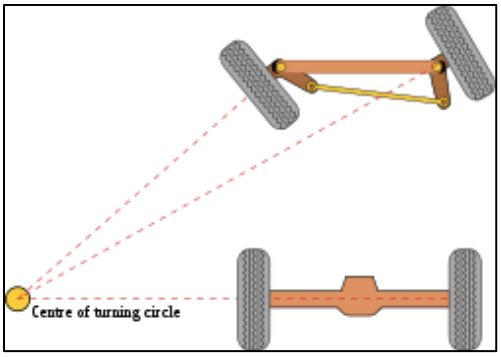

Figure 20: Steering mechanism

The robot options were JetRacer AI Kit or building a robot from scratch using Arduino Microcontroller.

Table 6: Robot design comparison

Feature	Jetracer AI Kit	Arduino Microcontroller
WIFI Connectivity	Yes	Needs external module
Steering Mechanism	Yes	Yes
Room For Error	Low	High
Cost	High	Mid

Ultimately, the Jetracer AI kit was the option chosen based on the previous comparison, as it is more reliable and has less room for error in steering. This option also follows all design requirements and constraints, it is battery operated and depicts a real vehicle.

Figure 21: JetRacer AI Kit

3.5 Developed Design

The final design of the system was not finalized and concluded from the first try, but through the analysis and testing of different proposed designs. In this section, the discussion of all designs proposed will be presented, from the RL modeling to the hardware configurations.

3.5.1 Overall Design

For the system to achieve a successful parking, it will need to go through different states/phases. Initially, a system calibration should be carried out by the system administrator. This calibration is done only at the setup of the system when it is first deployed. The objective of the calibration is to adjust and tune the top view camera to get a full view of the parking environment. Also, to manually assign parking positions to the correct parking spots.

The second step would be to be able to connect to the vehicle and send the appropriate commands to control it. The connection is done through a local network that connects the client, vehicle, with the server, CPU. This architecture can be viewed in Figure 1. This connection is established as soon as the vehicle parks at the entrance of the parking. Once the connection is established, the system takes control of the vehicle and starts the parking procedure.

Regarding how the parking procedure is completed, the vehicle should first be parked in a constant initial position. This initial position is the entrance of the parking, once the vehicle enters, complete control of it will be handed to the system. The RL model would not be directly called to start the parking, but first, a check on the parking locations is conducted. Based on this check, each parking spot would be classified as occupied or free. Then, the vehicle navigates to a position near the goal parking and the RL model is then deployed. Thus, the parking procedure is divided into 2 main stages, a deterministic one, and a RL-based one.

3.5.2 RL Model & Simulation

As concluded from 3.4.1, the simulation environment that will be used is 'Parking-env' [12], which is a subset of 'Highway-env', a simple gym-based 2D environment for autonomous driving scenarios, see

Appendices. The setup of the simulation parking would be modified to have six parking spaces as in Figure 22. Notice that the dimensions were also changed for both the vehicle and the parking spaces, the original setup could be seen in Figure 14.

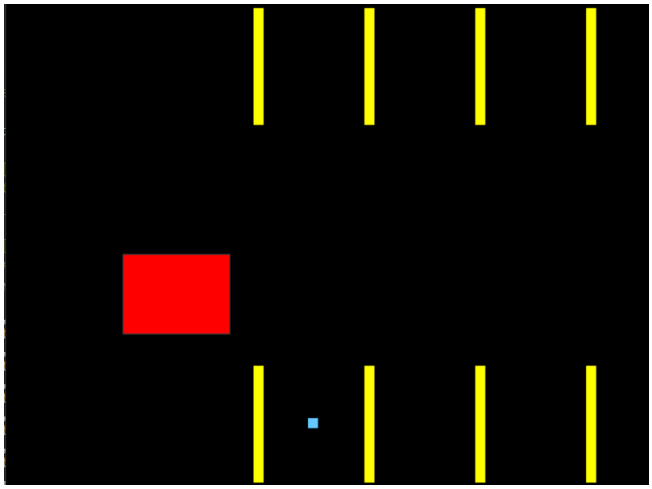

Figure 22: Simulation environment setup

The use of a helpful library as gym would allow the possibility of implementing different RL algorithms in the training stage. Also, it would allow flexibility in the design of both the action and state spaces. Note that all code regarding the simulation can be found in the Github repository of the project [20].

The design of the RL model would only consider a single parking spot, i.e., the agent will be trained to park in a single parking space. This simplification of the model goal would allow more room for testing and trying different state and action spaces, in addition to the analysis of multiple reward systems and faster training.

Throughout the work, and with testing, the architecture design of the RL model had gone through multiple changes, see 3.5.2.3. But mainly, an initially proposed architecture was analyzed where a single model would be trained to park the agent perpendicularly in reverse position. This design had multiple shortcomings, from quickly converging to a false objective, to achieving poor results. Thus, another design was developed and implemented.

The architecture of the ultimate RL model consisted of 3 different models that communicate with each other. The communication process could be considered as a handover in the control of the agent at certain stages of the parking process. Each model was trained separately to achieve a certain objective through the parking procedure. The improvement of this architecture could be

seen in dividing the problem into multiple simple tasks, then, tackling each at a time. An illustration of the proposed architectures can be seen in Figure 23.

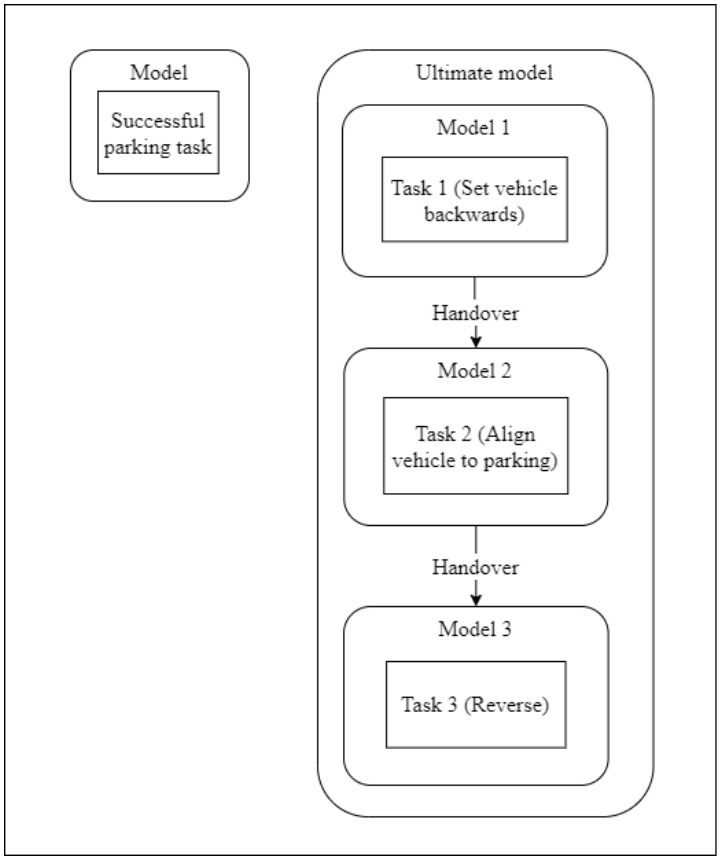

Figure 23: Initial and enhanced RL model architectures; left is the initial and right is the enhanced

Figure 24 shows the tasks of each RL model. The first task would be to get the agent to the right of the goal with a heading angle close to 35°. Then, the next task is to get the agent aligned with the parking, and finally, to reverse into the goal and park with no collisions with the lines.

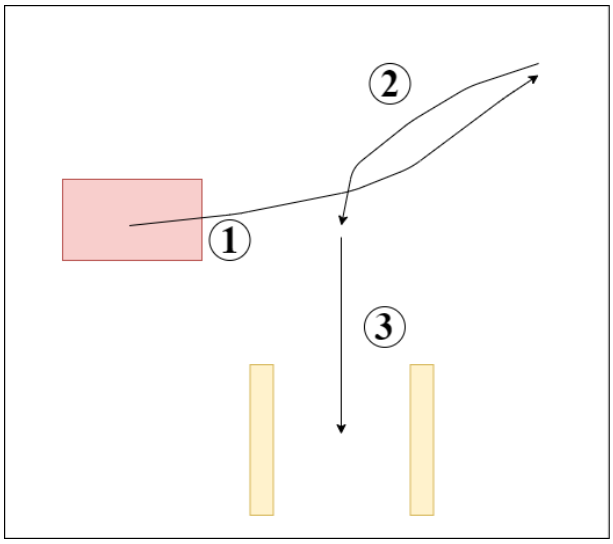

Figure 24: Models tasks

3.5.2.1 State Space

An optimal state space captures meaningful information about the agent, which assists it to achieve its goal, i.e., maximizing cumulative reward. The goal of the agent was to park perpendicularly in a single parking spot, starting from a fixed initial position till reaching the goal state.

The initial state space design consisted of 4 distances that represented how far the vehicle is from the goal parking spot. The distance metric used for calculation was the Euclidean distance, and the order of the points was preserved. To further illustrate the observations, Figure 25 shows a sample of them. As can be seen, the observations are the distances between each edge of the vehicle rectangle with its corresponding corner of the parking. The reason behind choosing this kind of observation is that it indicates how far/close the agent is from the goal.

This design was tested and trained on, but it was concluded that the results could be further enhanced. The enhancement of the observations was in the addition of the location of the agent in the parking space (rectangular coordinates) and its heading angle. Figure 25 shows the final observations developed. The rectangular coordinates gave the agent an indication of where it is in the parking space, while the tilting angle introduced how aligned the agent is with the parking goal.

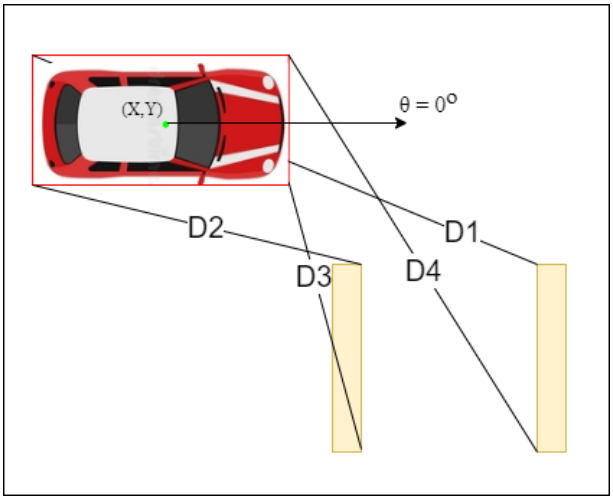

Figure 25: Final developed observations

To represent the designed state space in the simulation environment, a dictionary was used. From Figure 26, the state space was a dictionary of 3 ‘box’ objects. A box is a data structure developed from the gym environment, it is a space inherited from the base space() class in gym. It represents a tensor that takes the following parameters: minimum value, maximum value, data type, and shape. A tensor is a vector-like data structure that is heavily used in machine learning due to its optimization features in training [21]. The minimum and maximum values are usually used for normalization proposes, while the shape could be customized to match the designed observations, e.g., the position was of shape (2,), a 1x2 tensor. A sample of the observation could be seen in Figure 26.

Observations = {

Distances :	D1	D2	D3	D4
Heading :	θ			
Position :	X	Y		

}

Figure 26: Observations data structure in the simulation

3.5.2.2 Action Space

The action space would be responsible for the movement of the agent. As the agent is a vehicle, the actions taken should be related to a combination of its speed, steering angle, brakes, or acceleration. There are two main types of action spaces, continuous and discrete. A continuous

action is represented by a real value, $x \in \mathrm{R}$, while in a discrete action space, each action is an integer $a \in 0,1, \ldots n : n \in N$ that represents a certain behavior of the agent. Both spaces were applicable to the problem at hand, but a discrete one would be much simpler in terms of computational power and training time [22].

The designed action space was a set of combinations of the agent's speed and steering angle. For its speed, three possible values were chosen, forwards, backward, and break. And for the steering angle, the set of angles between -30° and 30° with a step size of 10 was chosen. The maximum steering angle of 30° was not chosen arbitrarily, but it is like most realistic vehicles, as the actual maximum angle lies within a close range of 30°. With this action space, the requirement of the agent obeying realistic physics is achieved.

Table 7 describes the actions of the agent, each combination of speed and angle is considered an action, note that the negative and positive values of speed indicate backward and forward movements respectively. E.g., action 11 moves the agent forwards with an angle of 0°, and action 0 moves it backward with an angle of -30°.

Table 7: Action space

		Steering Angle						
		-30°	-20°	-10°	0°	10°	20°	30°
Speed	-1	0	3	6	9	12	15	18
	0	1	4	7	10	13	16	19
	1	2	5	8	11	14	17	20

3.5.2.3 Reward System

One of the most fundamental stages in the design of the RL model was the reward shaping. The rewards that an agent receives by doing actions, are the reason behind the 'learning' of the agent. Depending on how the rewards are shaped, the objective and behavior of the agent change. There were a lot of trials of different reward systems throughout the training phase where most of them were failed attempts. Some systems converged to a different goal state than desired, while others did not converge at all but acquired a random-like behavior. In this section, the discussion of all reward systems applied and tested will be held.

- Sparse Rewards:

Sparse rewards are a type of rewards in RL models where feedback is given in discrete form. The feedback depends on the outcome of the action taken by the agent. E.g., if an action's result was the goal state, then the reward would be at its highest, while if an action led to a crash, then it would receive a high penalty. This means that the agent needs to figure out the way to achieve its objective, with no indication or feedback on whether it is going on the right path or not. The sparse rewards system implemented was the following:

- Reach goal state: +2000
- Line crash: -2000
- At each step: -1

The negative penalty on each step was to prevent time-wasting, as if it was not there, the agent would just not move. Thus, maximizing reward.

The result of the above reward scheme led to the agent crashing as quickly as possible. This outcome was due to it not being able to achieve the goal state. Thus, concluding that all possible outcomes are a crash, so crashing quickly would lead to the least cumulative reward.

- Distance Rewards:

After the failed attempt of the sparse rewards, schemes where distance feedback was sent to the agent were applied. Note that the agent's goal was to park, which minimizes the 4 distance observations. These distances could be minimized by returning the negative mean value of them as feedback in each step. So, when the agent gets closer to the goal, the penalty would be lower and vice versa. As goes for when the negative distance between the rectangular coordinates (x,y) of the agent and the goal was returned. These two reward systems were applied, and all led to the same result. The common result was that the model did not converge but started to drift away from the parking. This behavior was due to the high penalty of the line crash, making the agent not able to try to get close to the parking.

The applied reward systems were as follows:

- Reach goal state: +2000
- Line crash: -2000
- At each step:

$$1) \ -\frac{(Obs_1+Obs_2+Obs_3+Obs_4)}{4} \tag{1}$$

$$2) \ -\sqrt{\left(X_{agent}-X_{goal}\right)^2+\left(Y_{agent}-Y_{goal}\right)^2} \tag{2}$$

As a result of the poor outcome of the previous rewards, a minor change was applied. The episode did not end at the line crash but only a penalty was assigned to the crash. This proposed solution would allow the agent to explore the environment more, allowing the chance to park successfully to grow. Note that this system may seem counterintuitive as the goal is to park with no collision with any lines. But there is a chance that even if it crashes at first, it would eventually converge to stop crashing and park successfully.

The result of this enhanced version of the reward system was that the agent had found the goal state, but its behavior was not the desired one. The movement of the agent was in a straight line getting closer to the parking as fast as possible, while crashing and not giving much attention to the lines. This behavior was thought to be fixed with more training iterations, but that was not the case. After more training iterations, the agent converged to stop moving as it would lead to minimizing the cumulative reward taken.

- Divide and Conquer Rewards

The next reward system developed was implemented by dividing the parking objective into multiple tasks. Each task would have its contribution to the total reward received. The idea behind this method was referenced from, where their reward consisted of 2 parts, one to get aligned with the parking and the other to get closer to it.

The initial division of the reward function was applied through the equation

(3).

$$R = R_{parallel} + R_{distance} + R_{line\ crash} \tag{3}$$

The first parameter $R_{parallel}$ is responsible for the agent to be aligned with the parking goal. The alignment would be calculated as the error rate between the current heading angle of the agent and the ideal -90º, as in equation *(4)*. Note that this case is concerned with reverse parking only, this is the reason behind choosing -90º as the ideal case. If the parking was in a perpendicular forward way, then the ideal heading would be 90º. The distance reward was

calculated through equation (5), which is the Euclidean distance (l2 norm) between the center point vector of the agent and the goal position. While the final penalty was given for any line crash with the parking space. Note that in this system the episode did not end with the line crash but only a penalty was given. Also, notice that all rewards were assigned a negative value as most of them were error values, deviations from an ideal case. This reward system maximizes equation (3), which is a sum of negative error values, meaning that the best outcome of the system would be to optimize the equation to be equal to zero (maximum value of error).

$$R_{parallel} = -\left(Heading_{\,agent} - (-90°)\right) \quad (4)$$

$$R_{distance} = -||\boldsymbol{V}_{agent} - \boldsymbol{V}_{goal}||_2 \quad (5)$$

$$R_{line\ crash} = -10 \quad (6)$$

When the training was conducted, multiple additions and adjustments to the above reward system were introduced. These additions were an attempt to fix problems and unwanted behavior from the agent that occurred throughout training. E.g., a common problem was that the agent was biased toward a shifted location to the goal. An attempt to mitigate this bias was to add a parameter in equation (3) that stressed on the horizontal alignment of the agent with the goal.

The final result, after all adjustments were considered, was that the agent successfully got aligned with the parking but did not manage to reverse and park without crashing. It would get stuck at the entrance of the parking going forwards and backward but not entering. This state was reached due to the penalty on the line crash, the agent became aware of the lines and concluded that any state near the line is considered a poor one. A proposed solution was to lower the line crash penalty, but it did not converge, and the same results occurred with more training iterations.

- Multiple Models Rewards:

The concluded design of the reward system, after all the mentioned attempts, was to divide the model into multiple models, each with its own goal and a reward function, as in Figure 23. This way, each model would establish its own optimal policy to solve its task. Having all rewards ultimately accumulate into a single equation may not be beneficial in complex tasks, as each parameter in equation (3) can update the common policy and alter any update

from other parameters in the equation. Splitting the final model into a model per task would mitigate this problem. Table 8 describes the goal of each task. Note that the episode ending at a line crash was enabled in this method.

Table 8: Models goal and initial states

Model	Goal	Initial state
Task_1	Get the agent to a position right of the parking goal with a heading angle near 30º	Left of the parking goal with a heading angle of 0º
Task_2	Get the agent to be aligned with the parking goal horizontally with a -90º heading angle	Goal of task_1
Task_3	Reverse the agent to enter the parking goal with no collisions	Goal of task_2

Two methods were adopted to implement the aforementioned reward system, one involving optimizing a multiplication inverse function of a value that reflects the heading and the location of the agent. And the other was to calculate the Root Mean Squared Error (RMSE) between the current state and the goal state. Each observation would be multiplied by a weight to stress on the relative importance of the observation in achieving its objective.

Equation *(7)* shows the multiplication inverse function used in the first approach. Ra was referenced from equation *(4)* with the change of the goal angle, which was dependent on the task at hand. Rx was the horizontal distance between the agent and the relative goal position of the specified task. E.g., in task 1 the goal position would be located to the right side of the parking goal, while in task 2 the goal was to be aligned exactly above the parking. The division by 100 could be considered as a normalization technique to make the reward values in a range close to the crash and goal state penalty and reward. The agent received a penalty of -2000 on any line crash and a +5000 reward if it reached the goal state.

$$\frac{1}{(R_a+R_x)/100} \tag{7}$$

This reward system was further tested and deployed on hardware due to its successful results, as will be further shown in chapter 4. The movement of the agent was not the most optimal one, even

though the time of parking was way lower than the minimum requirements, thus, the next reward function was proposed and further tested.

The second reward function relied on the RMSE between the current state and the goal state. The observation and goal vector in equation *(8)* are the observation dictionaries transformed into vector forms, refer to Figure 26. After the distance measurement with the goal state, the dot product with the weight vector would be computed. The weight vector gives certain observations higher importance based on the task being solved.

$$\sqrt{\left|\boldsymbol{V}_{obs} - \boldsymbol{V}_{goal}\right|_2 \cdot \boldsymbol{w}} \tag{8}$$

$$\boldsymbol{w} = [0.05, 0.05, 0.05, 0.05, 1, 1, 1] \tag{9}$$

This reward scheme led to the most optimal behavior of the agent; parking as fast as possible. Further results and how the models performed on hardware will be analyzed in chapter 4.

3.5.3 Prototype Design

As mentioned in 3.5.1, an initial system calibration needs to be done when the system is deployed. In this section, a comprehensive analysis of how the prototype was handled will be presented. The analysis will focus mainly on how the observations (state space) got extracted out of the top-view image of the parking. Also, for the model to be deployed properly, a protocol needs to be developed to communicate with the vehicle, allowing it to understand the output of the RL model (Action Space).

3.5.3.1 Image Processing and Object Detection

The input image will go through a processing pipeline that extracts the observations from it. The pipeline stages are as follows:

- Four-Point Transformation and Resizing:

Four-point transformation is the process of transforming the image to be flatter and to obtain a bird's-eye view (top-down view). This view would enable the image to be closer to the 2D simulation environment. Figure 27 shows the result of the transformation.

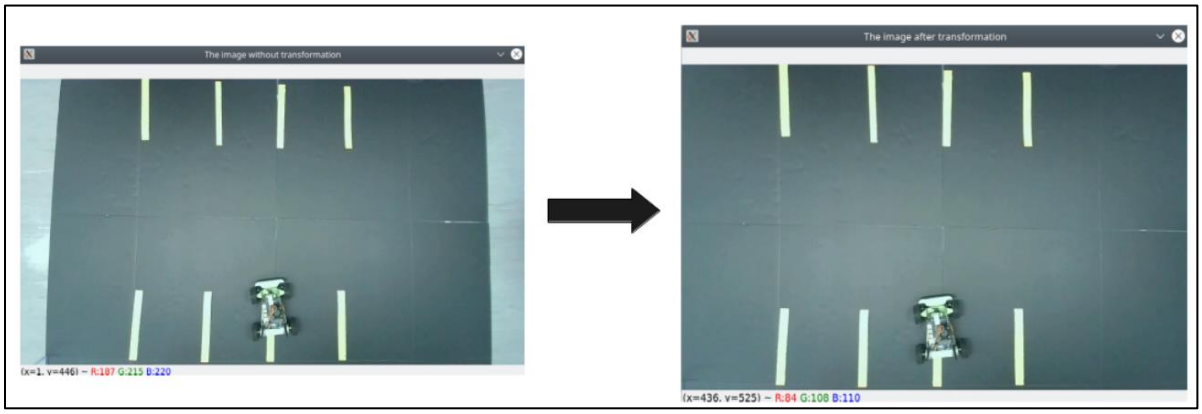

Figure 27: 4-point transformation

The transformation function uses the following two functions from the 'OpenCV' package [23]:

- *cv2.getPerspectiveTransform()*
- *cv2.warpPerspective()*

Where the output of the first function is the input to the second. The first function, *getPerspectiveTransform*, was used to change the point of view of the image. For it to work properly; the 4 points along with the image dimensions needed to be passed to it. Note that the four points will be provided by the user, and the new dimensions will be calculated based on them.

The first step in finding the new width and height of the image is to order the 4 points. The order of the points will be fixed, and the output of this step should be an array of the points ordered as follows: top-left, top-right, bottom-right, and bottom-left.

For the width, two distances shall be calculated. The distance between the bottom-right and the bottom-left points, and the distance between the top-right and the top-left points. The maximum of those 2 distances (widths) would be assigned as the width of the image. Similarly, for the height, the two distances were between the top-right and the bottom-right points, and the top-left and bottom-left points.

The next function, *warpPrespective*, generates the final transformed image. It takes the following parameters as inputs:

- The original image
- The transformation matrix (output of *getPerspectiveTransform*)
- The dimensions of the new image

Note that after the transformation, the image will have new dimensions, as calculated previously. But to make the image as close as possible to the simulation environment, it will be resized to match the simulation. The frame size in the simulation was 640 x 480 pixels. Figure 28 shows the image after all transformations mentioned above.

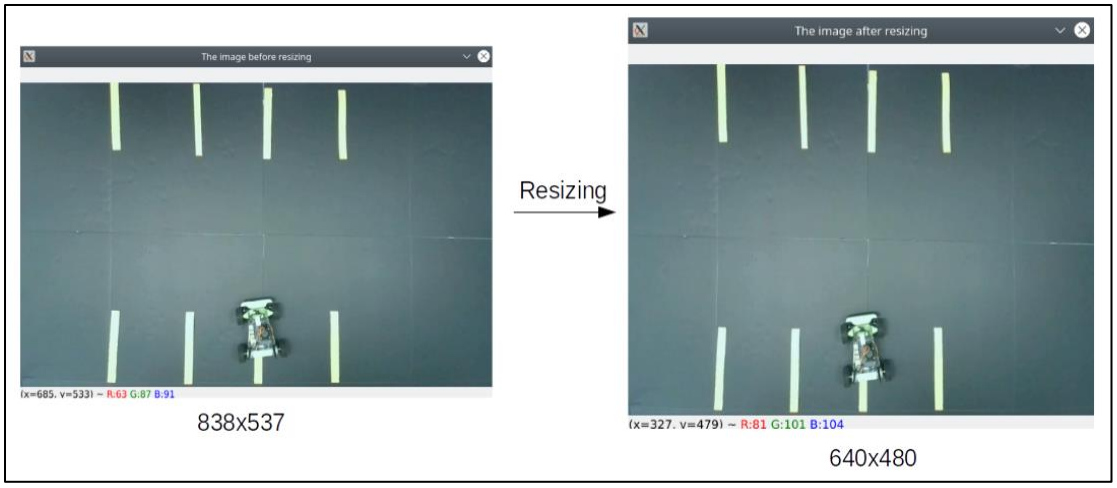

Figure 28: Resized image

- Object Detection

As part of the calibration process, the system will ask the user to take an image of the parking completely empty, this image will be called the "reference image". When the calibrated system runs, all vehicles in the parking should be detected. To achieve this detection, the system will first transform (discussed in the previous subsection) both the reference image and any input frame. Then a Subtraction of the 2 images would be generated, Figure 29. The generated difference represents any object in the frame that does not exist in the reference image. That way, all vehicles/objects will be detected. The subtraction was done using the 'subtract' function from OpenCV [23].

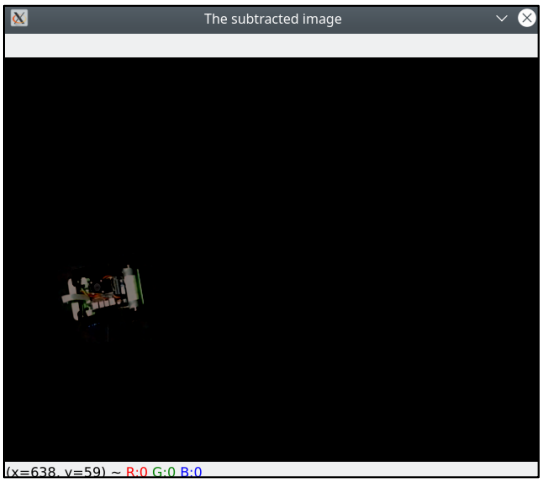

Figure 29: Reference subtracted from frame

After generating an image with all objects in the parking, a distinction scheme between the vehicles already parked and the target vehicle that needs control should be done. This process will be further discussed in the next sections.

- Frame Processing

After the subtraction process, the resulted image needs to be processed for a clear and accurate object detection. The processing on the frame was divided into two main parts. The first was to find the contours of the detected object (vehicle), while the second was to process these contours to get the best representation of the object.

A. Process the Frame and Find Object Contours

The initial processing technique adopted was to apply a greyscale transformation to the image. The implementation was done using *cv2.cvtColor()* function from OpenCV [23]. The greyscale transformation would make the image 2 dimensional instead of 3, which simplifies the next processing parts. This transformation is shown in Figure 30.

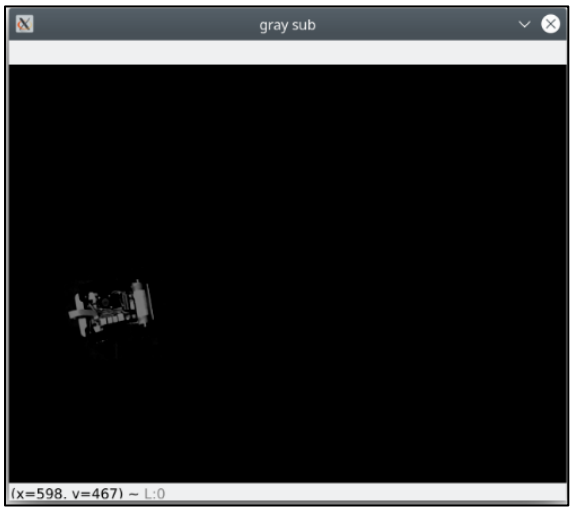

Figure 30: Greyscale image

The second transformation applied was dilation. The main purpose of it was to make the object represented as one entity as much as possible. The dilation sets any pixel to be the maximum value in the neighborhood of that pixel. In a binary image, the pixel will be 1 if any of the neighbors of that pixel is 1, in that way all the holes and the gaps in the object will be filled. Thus, the object will be one entity. Note that the sharpness of the dilation depends on the number of iterations done on the image, as seen in Figure 31.

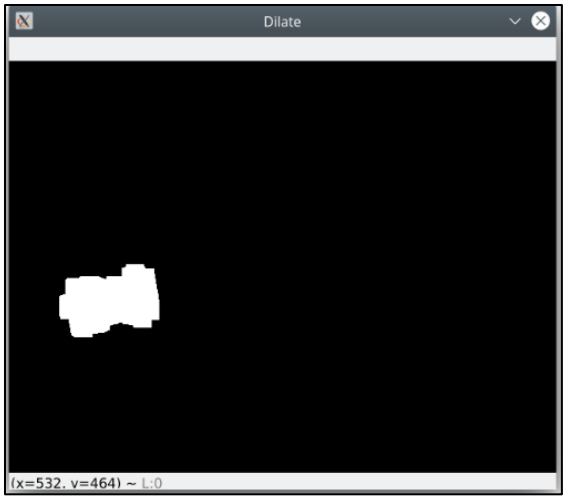

Figure 31: Greyscale image after dilation (10 iterations)

Next, erosion was applied to the image. The purpose of erosion can be thought of as the opposite of the dilation, as it uses the minimum value instead of the maximum. Its output was a more compact representation of the object, see Figure 32. It sets any pixel to the minimum value in the neighborhood of that pixel. In binary images, the pixel will be 0 if any of the neighbors of that pixel is 0. That way all, the objects would be smaller and more compact than the dilated image. Analogous to dilation, the sharpness of the erosion depends on the number of iterations done on the image, as seen in Figure 32.

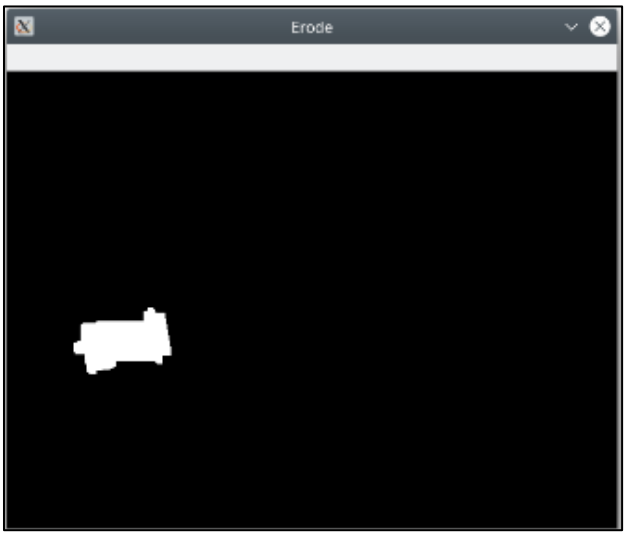

Figure 32: Dilated image after erosion (7 iterations)

Note that the dilation and erosion were implemented through *cv2.dilate()* and *cv2.erode()* functions. The number of iterations, in both transformations, depends on a lot of factors, e.g., the

quality of the camera and lighting conditions. The chosen approach to finding the best combination of iterations was trial and error. It was concluded that the dilation iterations should be set to 20 and erosion to 7, as seen in the above images.

After all transformations were done, the function *cv2.findContours()* was implemented to the processed image. This function returns the contours of all objects in the image. The next step would to be process these contours to get an accurate representation of the object.

B. Processing the Contours

After getting all contours of the objects, the goal would be to classify these objects as a parked vehicle or the vehicle that needs to be controlled (parked at the entrance). The inputs used for this classification were the contours and the parking locations' coordinates.

First, the function *cv2.minAreaRect()* was applied. It takes one of the contours as input and finds the minimum area rectangle that bounds the object. Note that this process is different from finding a normal box that bounds around the object. The difference lies in the consideration of the rotation of the rectangle that surrounds the object, in the minimum area rectangle, rotation (angle of the car) is considered and calculated. But in the other method, inserting a box on top of the detected object, when the object was rotated the box would just expand and shrink.

Before using the minimum area rectangle method, the problem of not being able to extract the angle of rotation was faced. To further illustrate the rotation problem, Figure 33, compares the two methods. Where the green rectangle represents the minimum area rectangle and the red one represents the normal bounding box.

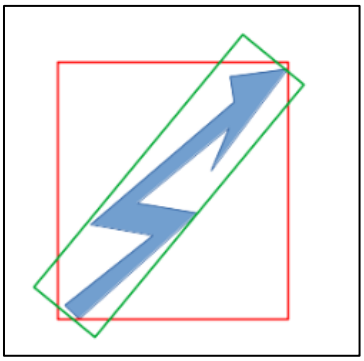

Figure 33: Normal bounding box and minimum rectangle area

The output of the *cv2.minAreaRect()* was the following:

- Center point of the box
- (Width, Height)
- Angle of the rotation

To get the coordinates of the corners of the box, *cv2.boxPoints()* was used. It takes the above 3 parameters, in a data structure of Box2D, and returns the corner points as an array.

To find if the parking location is occupied or free, the system compares between the center point of the rectangle (detected object) and the center of one of the parking locations, iterating through all objects and all parking spots.

After all mentioned transformations and processing, the system would ideally detect one contour that represents the car that needs to be parked. If the result of detection was more than one object, meaning that problems occurred in the subtraction process, most likely noise, then the area of the detected objects should be considered. Only the object with the closest area to the vehicle would be considered, any object besides it would be dropped as noise.

Ultimately, the system should have one object to control, the object will be represented as an array of coordinates that represents the corners of the detected rectangle, in addition to the Box2D tuple that was generated previously. The detected rectangle represents the vehicle as in Figure 34; any movement of the car will be immediately depicted by the rectangle. By using this representation of the vehicle, the state space extraction became possible and more accurate, which is discussed next.

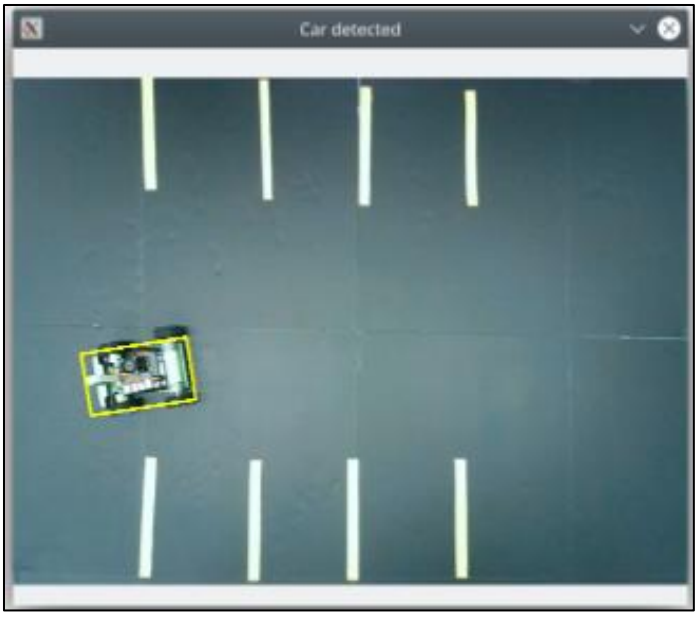

Figure 34: Vehicle detected

- Vehicle Corners Determination

To extract the observations from the image properly, the distances between the 4-corners of the vehicle and the 4-corners of the parking must be calculated in the right order, as in Figure 25. Thus, the system should be capable of determining the correct order of the 4-corners of the car (front-right, front-left, back-right, back-left). The procedure is divided into two connected parts, the first was to determine the order of the points while the car is in the initial position, while the second was when the car starts moving. The discussion of these two parts is presented next.

A. Determining the Initial Order of the Points

The goal of this part is to determine the order of the point while the car is located in the initial position. In the previous step, the system should have acquired 4 points that represent the car. Since the initial position is fixed, the heading of the car will also be fixed (to the right). Thus, initially, the system would determine the order of the 4 points based on their coordinates as follows:

- front-right: maximum x and maximum y
- front-left: maximum x and minimum y
- back-right: minimum x and maximum y
- back-left: minimum x and minimum y

Figure 35 illustrates the idea of ordering the 4 points, note that the blue, green, red, and purple points are the front-right, front-left, back-right, and back-left respectively.

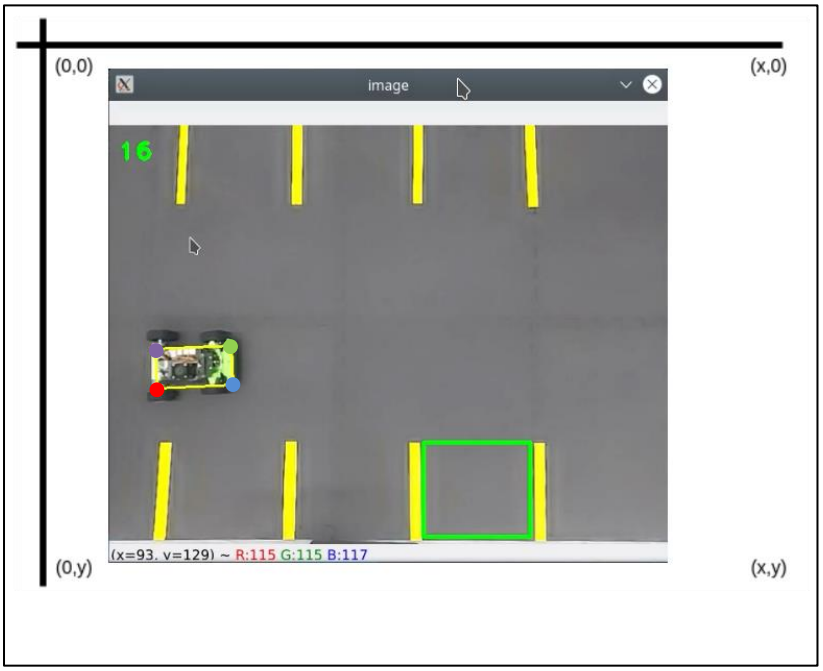

Figure 35: The vehicle in the initial position and the 4-corners ordered

B. Keeping Track of the Points

When the car starts moving, the points will change, and the system will have 4 different points. To keep track of the order of the points, the system will assign each new value (from the new input frame) the closest old value based on the Euclidean distance between them. The distance will be calculated using equation (10).

$$d = \sqrt{(x_2 - x_1)^2 - (y_2 - y_1)^2} \tag{10}$$

To clarify, assume that the old points were [(2, 55) (6, 263) (89, 270) (89, 40)] and the new points from the new frame are [(90, 41) (9, 54) (95, 265) (9, 260)]. For the system to figure out the new value of (2,55); The distance between that point and all other points should be calculated. Then, the closest point to it should be the new coordinates of (2,55). This process is illustrated in Figure 36. In that way, the order of the points will be conserved even while the vehicle is moving.

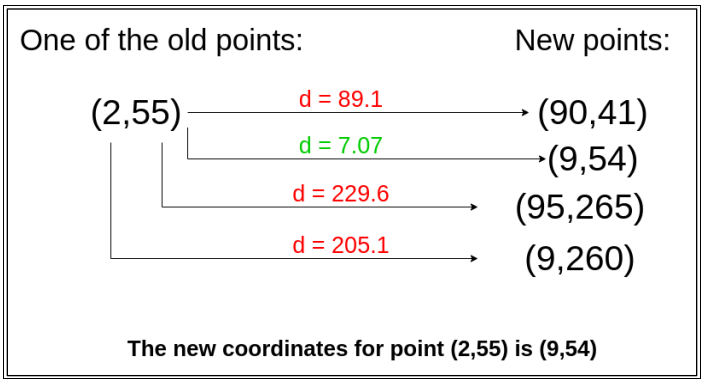

Figure 36: Updating the vehicle points order

- Getting the Observations (State Space) Ready for the Model

After all steps discussed previously, the system should be ready to construct all that data needed for the model to operate and make predictions.

A. Distance Observations

As discussed above, the system should calculate the distances between the car and the parking. At that point, the system should be able to determine the order of the points. So, by applying the Euclidean distance equation in (10), the distances between all points of the vehicle and the parking are calculated in the right order as in Figure 37.

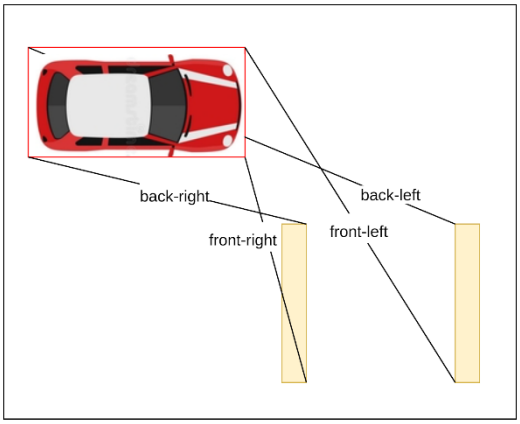

Figure 37: State space distances with the right order of the points

B. Heading of the Vehicle

The system will calculate the heading of the vehicle by using the 4 points and their order obtained previously. First, a reference vector pointed to the right was created. Then, the heading of the

vehicle would be the angle between the reference vector and the vehicle vector, as shown in Figure 38.

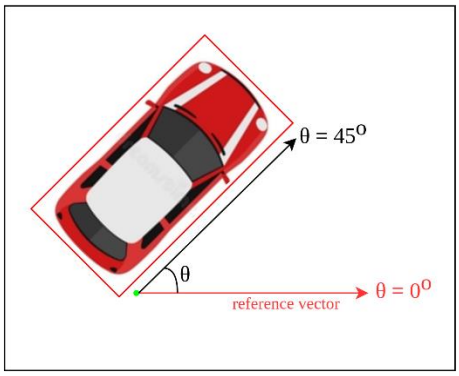

Figure 38: The angle between the reference vector and the vehicle vector

To calculate the magnitude of the heading, a vector generated from two aligned points of the vehicle should be first generated. The new vector is acquired by subtracting the 2 points as explained in Figure 39. Ultimately, the value of the angle between the reference vector and the vehicle vector would be calculated by applying the cosine inverse to Equation.

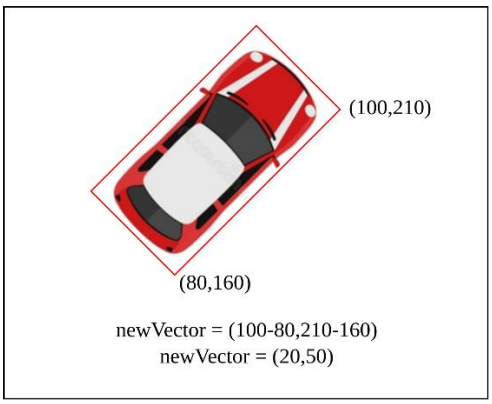

Figure 39: Calculating the new vector

$$cos\ \theta = \frac{\boldsymbol{V}_{ref} \cdot \boldsymbol{V}_{vehicle}}{|\boldsymbol{V}_{ref}|\ |\boldsymbol{V}_{vehicle}|} \tag{11}$$

By applying the previous equation, the sign of the heading will not be calculated properly. E.g., if the vehicle is in a 90° position or in a -90° position; the equation will produce 90° in both cases. To solve this problem, the system will keep checking the position of the points, and based on them

the right sign will be assigned to the angle. I.e., when the front points are higher than the back ones, then the heading is -90°, and when the front points are lower, then it would be 90°.

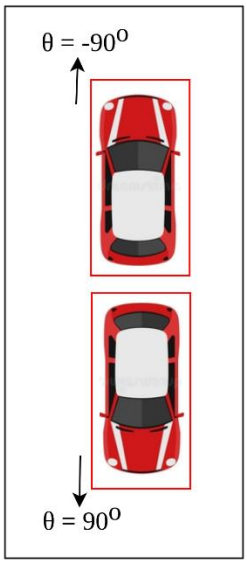

Figure 40: The difference between the -90° position and the 90° position

C. Vehicle Position

To get the position of the vehicle, the system uses the output of the function discussed previously, *cv2.minAreaRect()*. One of the outputs of that function is the center of mass of the detected rectangle of the vehicle.

4 Results

4.1 Prototype Setup

To test the developed RL model on hardware, a prototype that mimics the one in the simulation was needed to be built, see Figure 41. The prototype consisted of three main parts, a server, a client, and a local network to connect them. The client was the Jetson representing the controlled vehicle, the server was a local laptop as the CPU of the system, and the network was established through a router. The input stream of images was taken from the top-view camera, this was the only input taken by the system, excluding any manual commands, e.g., park in a specific position.

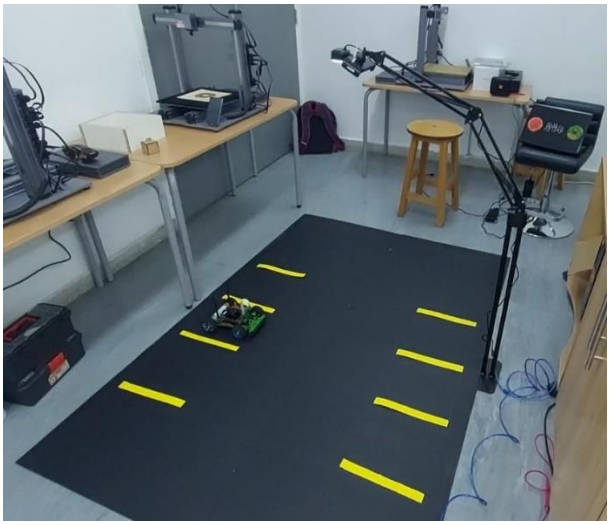

Figure 41: Built prototype

Note that no training was done nor continued on hardware, all models used were trained previously and transferred. The models were all saved on the server side and called when needed.

4.2 Hardware Deployment

Before applying the trained model on the hardware system, some parameters were needed to be tuned for the system to work properly. Even though the simulation was designed to exactly mimic the hardware setup, there still existed points of deviation between the two. To fix this problem, a mapping scheme in the dimensions/coordinates system of the two was developed. Another point that needs to be mentioned is that the developed model was trained to park in a single position. This position has its unique rectangular coordinates, thus, when parked in a different location, the model would observe states never seen nor trained on before. These new states would lead to taking

inconsistent and error-prone actions. Consequently, pre and post processing layers were added at the input and output of the RL model to make it dynamic and able to park in all positions. Also, one of the design requirements was that the system should work on both free and occupied parking locations. Due to the object detection scheme discussed before, the existence of other objects in the frame would provoke problems in the detection process of the vehicle. Finally, an extra safety precaution was implemented to make the system more realistic and valid for production.

4.2.1 Mapping Between Simulation and Hardware

When the RL model was first deployed on hardware, it was noticed that the observations did not match with the ones in simulation. This remark was concluded when both the vehicle and the agent were in the exact same position and heading but produced different observations. This inconsistency of the observations was mainly in the position argument, as the heading and distance observations are relative and common between nearly all parking spots.

To map between the observation of simulation and hardware, the architecture illustrated in Figure 42 was developed. Two layers were added to the RL model architecture, a preprocessing and a postprocessing one. The first would be reflected on the observations, while the latter on the actions taken.

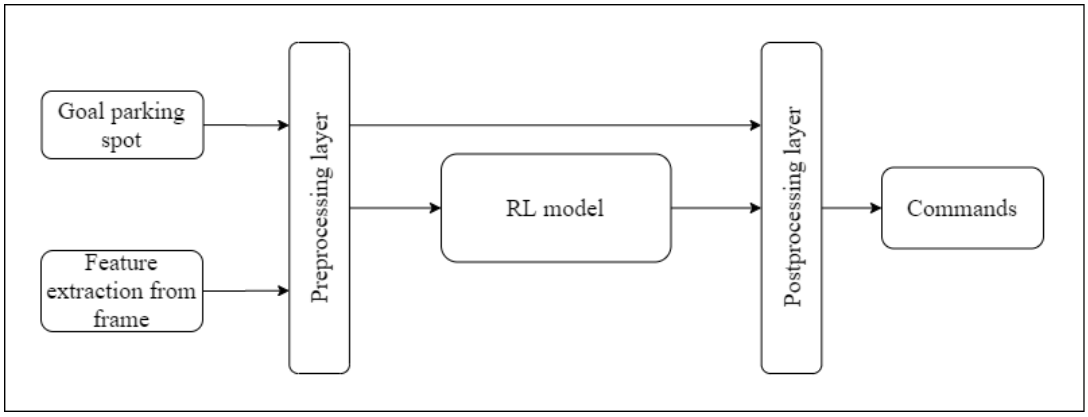

Figure 42: RL model architecture on hardware

The mapping scheme was to find a reference point between the simulation and the input frame. A point that is common between the two so that an offset would be added to the frame to transform its coordinates to the ones in simulation. Keep in mind that the goal of this task was to 'trick' the model into thinking it is in the same position as in simulation. This will also be the case in the next section when the vehicle is needed to park in multiple spots.

The reference point that was used was the center of the first parking position (bottom left parking). The goal was to find the offset that matches this point in both simulation and hardware. This offset would then be added to all observations coming into the model. This offset was named the 'mapping offset'.

4.2.2 Dynamic Parking in Multiple Locations

As discussed above, another offset was needed to be added to make the model dynamic and adapt to multiple parking locations. The offset considered only the horizontal position of the vehicle, as the distance between each parking location on the same side is constant. Thus, if this distance was subtracted from the vehicle position, then it would be as if it is parking on the first spot.

Another preprocessing step that needed to be added was to reverse the vertical position and the heading of the vehicle only if it was parking in a spot at the top. The negation of the vertical position would act as a reflection of the centerline of the parking. Keep in mind that the whole coordinate system of the simulation was different from the input frame. Both systems are illustrated in Figure 43.

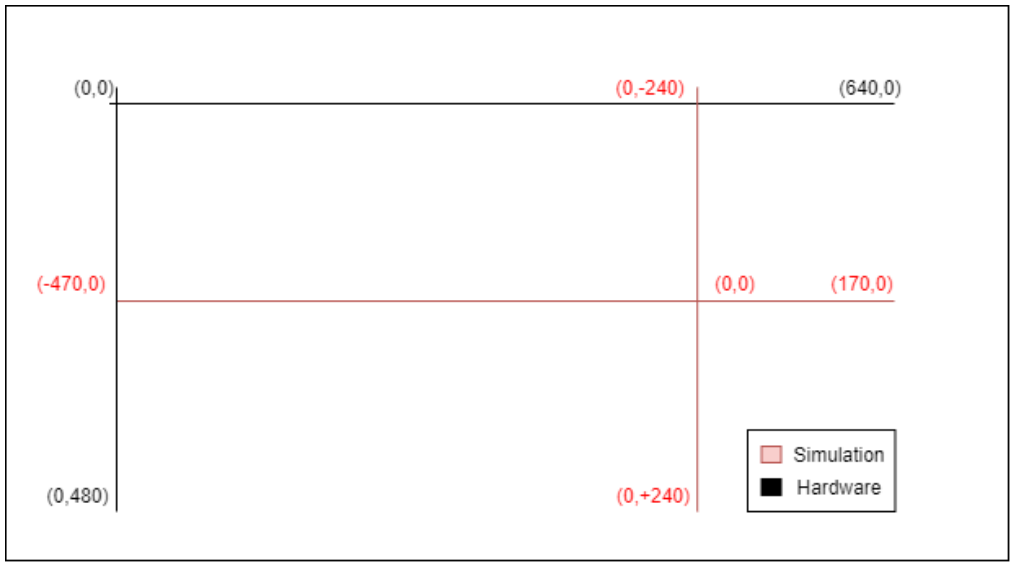

Figure 43: Coordinate system of simulation and hardware

It can be concluded from these coordinates systems that another mapping (offsets) needs to be added to make them match. The transformation was done by shifting the origin of the old system (hardware) to the new one (simulation). Thus, the offset would be -240 on the horizontal axis and -470 on the vertical axis.

It should be stressed that the preprocessing layer of the model heavily depends on the target parking location. So, it consists of a total of 3 offsets to the positional observations and a possible negation applied to both the heading and vertical position of the vehicle.

The total offset applied in the preprocessing layer can be calculated through the following equations.

$$(X_{offset}, Y_{offset}) = (X_{map\,1}, Y_{map\,1}) + (X_{map\,2}, Y_{map\,2}) + (X_{goal\ Offset}, Y_{goal\ Offset}) \quad (12)$$

$$(X_{map\,1}, Y_{map\,1}) = (-240, -470) \quad (13)$$

$$(X_{map\,2}, Y_{map\,2}) = (X_{first\ goal\ sim} - X_{first\ goal}, Y_{first\ goal\ sim} - Y_{first\ goal}) \quad (14)$$

$$(X_{Offset\ goal}, Y_{Offset\ goal}) = (|X_{vehicle} - X_{goal}|, 0) \quad (15)$$

Moving on to the postprocessing layer, which controls the actions sent to the vehicle, the goal parking location is also sent as an input to it. Based on this input, the actions are mapped. If the goal spot was a parking at the bottom, nothing would be applied to the actions, and they will remain as is. But if the goal spot was at the top, then actions must be switched so that each angle is inverted. E.g., if the action commanded the vehicle to move forwards with an angle of 30°, then it should be changed to -30°. This inversion scheme is due to the movement of the top and bottom parking spots being exactly a mirror of each other.

4.2.3 Free and Occupied Parking Detection

The next requirement of the system was to operate in a state of both free and occupied parking locations. The scheme of detecting an occupied parking was discussed in 3.5.3.1, this section will focus on how the existence of other objects could affect the behavior of the model. Based on the object detection scheme used, the system does not differentiate between any objects in the frame. So, by default, it would assign the element of 'controlled vehicle' to any random object detected in the frame. To solve this problem, two filters were added to only select objects that could possibly

be a vehicle. This means that any noise in the image would not be taken into consideration after applying the filters. Which implies that all vehicle objects would be detected, parked, or controlled.

The first filter would be to bound the area of the detected objects into a range between a minimum and a maximum threshold. The threshold values were selected by adding/subtracting a margin to the ideal area of the controlled vehicle. The ideal area was an approximate value, developed heuristically based on the used dimensions, that would work on multiple vehicle sizes. This filtration will exclude any small objects in the frame, e.g., noise. The second filter was to exclude all objects that are parked. This exclusion was done by comparing the position of the detected object with the parking position. Finally, the controlled vehicle would be the one with the area closest to the ideal case. The pipeline of identifying the controlled vehicle from all other objects is illustrated in Figure 44.

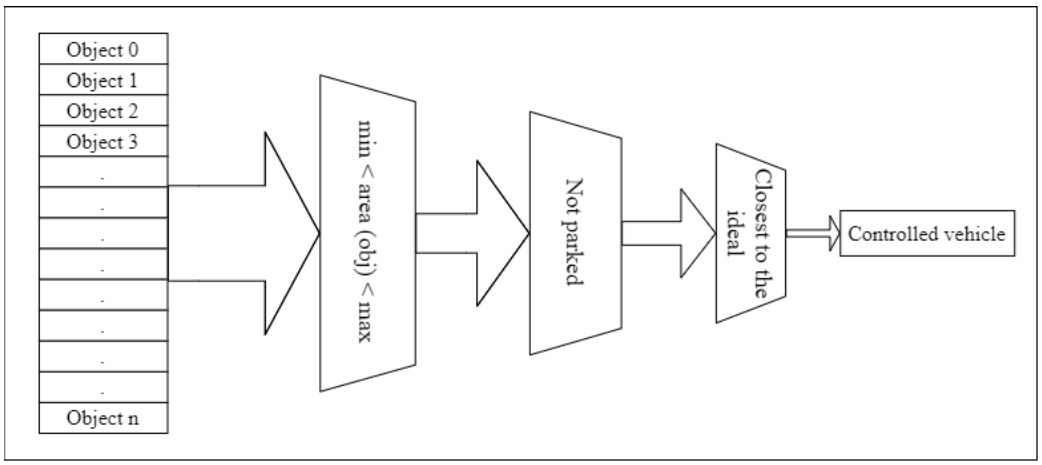

Figure 44: Controlled vehicle detection pipeline

4.2.4 Safety Precautions

The final part of deploying the model to the system would be to make sure that it is safe and reliable. The main safety constraint that needs implementation is the handling of crashes or any possibility of a crash; the context of a crash here is with other objects/vehicles, not lines.

- Initial Safety Check

As an initial step, the system does not start operating until a safety check is conducted. This check consists of making sure that the street is empty. If objects existed on the street, then there is a chance for the controlled vehicle to crash them. Thus, the system would send the user a prompt informing him that the environment is not safe to park, and the vehicle will not move.

- Object Handling

The next safety precaution would be to stop the system if an object suddenly appeared while the vehicle was parking. E.g., if a human interfered with the parking process and walked into the vehicle. The system would stop executing and close the connection with the client (vehicle). Then, human interference would be needed to handle such an event, moving the vehicle out of the parking space or back into the entrance.

- Connection Error

The final system configuration was to instantly stop the vehicle when the connection is lost with the server. This case is critical, as if it was not handled, the vehicle would conserve its last state and keep moving with the most recent command received. Meaning that if any error occurred in the communication process between the server and the client, the system would detect it and stop operating.

4.3 Simulation Results Discussion

This section focuses on the training results of different model designs. The algorithm used in training was 'Proximal Policy Optimization' (PPO), which is an enhanced version of the 'Actor to Critic' (A2C) algorithm. Regarding the actor and critic networks, the default architectures provided by the stable-baselines-3 library were used. These architectures consisted of fully connected layers where the input layer represented the observations, and the output layer represented the actions performed. Note that this algorithm was considered a black box in this work, as no modifications or adjustments were applied to it.

The evaluation metrics of the models were the mean cumulative episode reward and length. The mean cumulative reward should be representative of the behavior of the agent, the higher the better. Thus, the graph of this metric ideally should always be in an increasing trend. The mean length of all episodes would indicate the speed of the agent; if it reaches the goal quickly or takes all steps allowed, or if it does not even reach it but crashes. Note that all graphs presented next will have different relative cumulative reward values, as each would be calculated using a specific design function. This difference could be seen in the vertical axis, but regarding the horizontal axis, in all plots, it represents the total training iterations performed.

Note that all training in this work was performed on local devices with specs described in Table 9.

Table 9: Device trained on

	Device 1	Device 2
Device name	HP Pavilion Gaming 15 [24]	Asus TUF Gaming F15 [25]
Processor	Intel Core i7-8750H 2.20GHz	11th Gen Intel core i7-11500H 2.3GHz
RAM	16 GB	16 GB
GPU	NVIDIA GeForce GTX 1050Ti	NVIDIA GeForce RTX 3050

- Distance Rewards

Per the discussion in 3.5.2.3, the ultimate reward system was not developed until different reward schemes were attempted. One of the first reward system implementations was when the negative distance between the agent and the center point of the parking was returned at each step. Thus, the goal would be to minimize this value, which minimizes the distance and gets the agent closer to the goal. The results of this system are shown in Figure 45 and Figure 46. It can be noticed that the trend of the plot is decreasing, which means that this reward system would eventually collapse and drift from reaching the goal state. At first, early 400k iterations, the results were getting better, but as training continued, the agent would find the fastest way to crash. This could be due to the high penalty of the crash, yet this state being an inevitable one. So, the agent would find the fastest way to reach it.

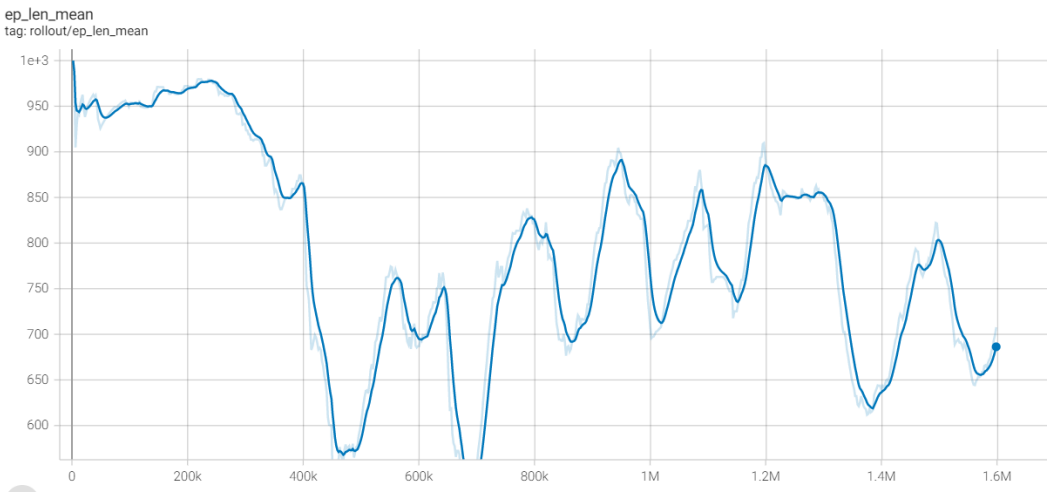

Figure 45: Mean episode reward of distance from goal reward function

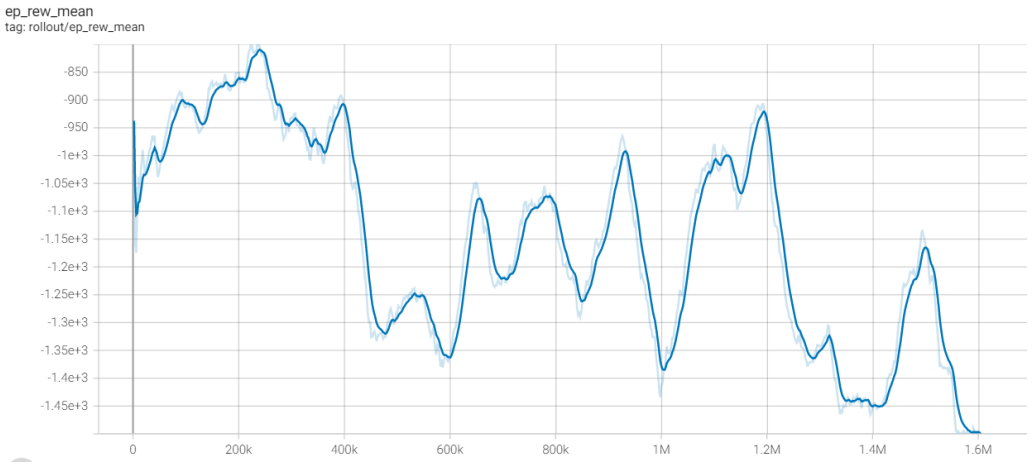

Figure 46: Mean episode length of distance from goal reward function

- Divide and Conquer Rewards

The first reward system that resulted in an agent that takes close to desired actions was when the reward function was divided into 3 parts, as in Equation (3). This function led to the results seen in Figure 47 and Figure 48. It achieved a mediocre behavior where it goes forwards to the right and reverses trying to enter the parking but failing. As training continued, it did not optimize its performance but converged to always crash.

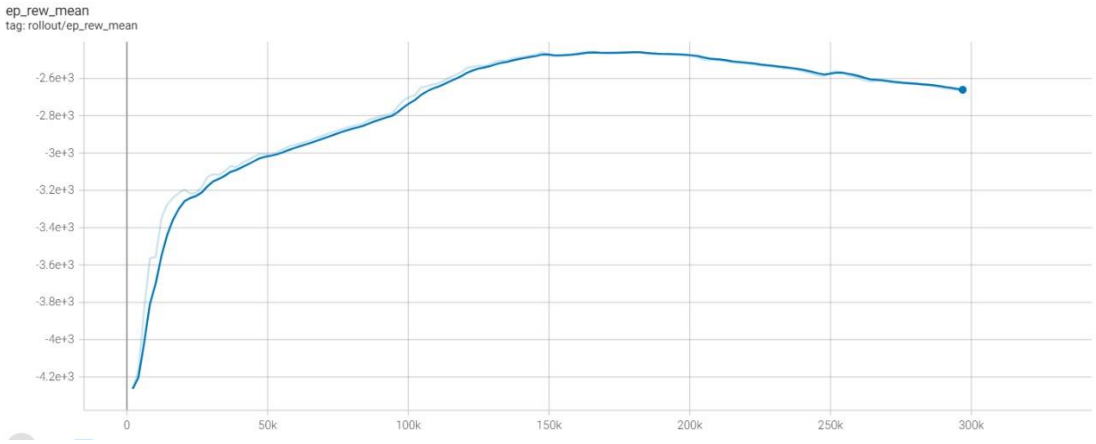

Figure 47: Mean episode reward of divided reward function

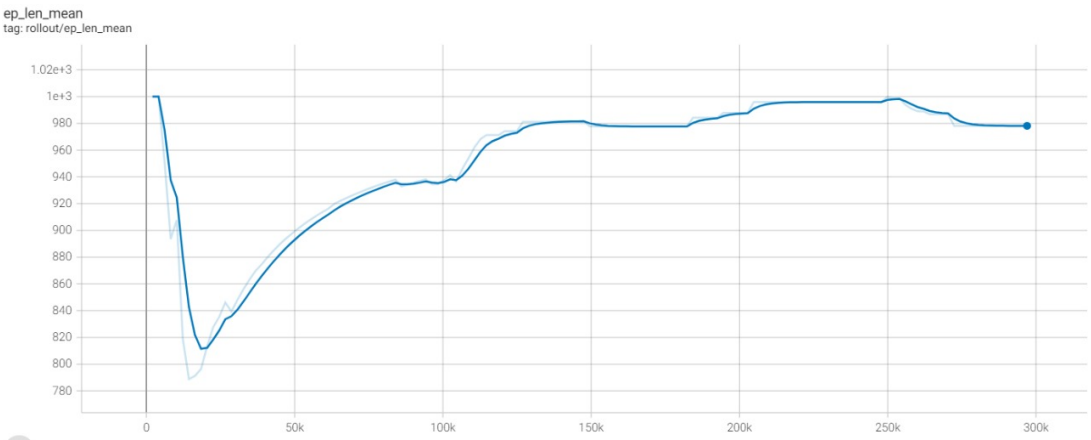

Figure 48: Mean episode length of divided reward function

A method of transforming the reward system to a piecewise function, equation (16), was applied to help solve the problem above. When the reward function is represented as piecewise, the step rewards would be given to the agent in hierarchical order. Meaning that, initially, the rewards would be shaped to achieve a sub-task, and later, depending on some conditions, the objective would change. This scheme would help pave the path for the agent to achieve the goal state.

$$R = \begin{cases} R_{parallel} : initially \\ R_{parallel} + R_{X-allignement} : Heading = -90° \\ R_{parallel} + R_{X-allignement} + R_{distance} : Heading = -90° and\ X_{agent} = X_{parking} \end{cases} \tag{16}$$

The first sub-task was to get the agent perpendicular to the parking. Then, making sure that the horizontal alignment was also correct, directly above the goal. Finally, the distance is added to the calculations, going backward to park.

The results of such system can be shown in Figure 49 and Figure 50. The model achieved the initial sub-task partially, meaning that it did not get aligned exactly to 90° but close. As training continued it started to drift away from the parking. This behavior may be the result of combing all sub-tasks into a single equation. I.e., whatever changes were applied to the policy to achieve a sub-task, were affecting the older ones made by the previous task.

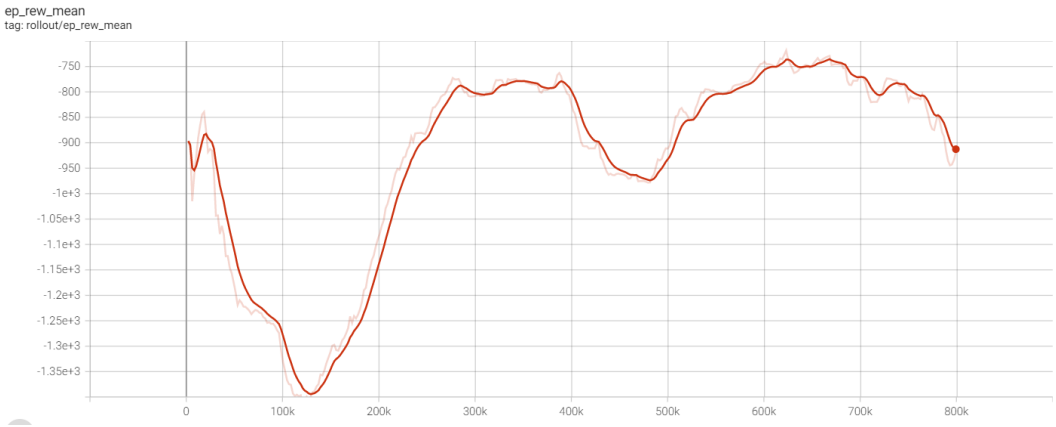

Figure 49: Mean episode reward of hierarchal reward system

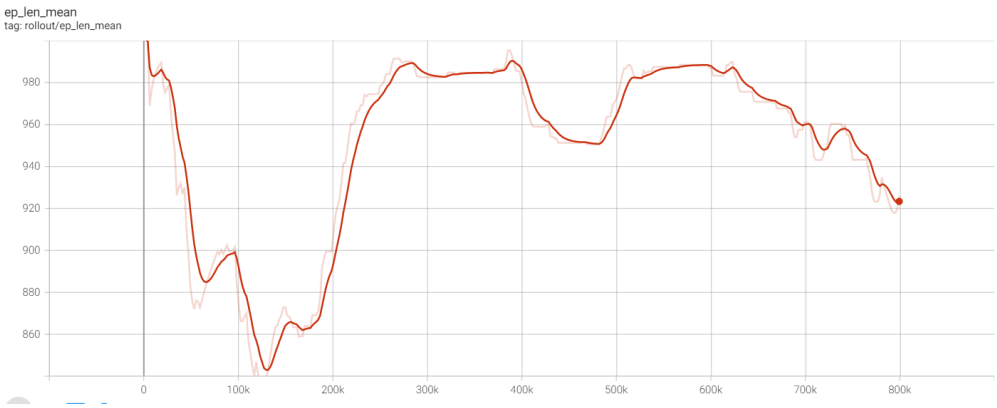

Figure 50: Mean episode length of hierarchal reward system

- Multiple Models Rewards

The idea behind the design of 3 separate models, as in Figure 23, was a solution to the problem stated in the above design. If 3 models were trained, one for each task, then none would be affected by any previous models. The tasks were divided as discussed in 3.5.2, and the results of each model are presented below.

Due to the simplification of the objective of the RL models, the training iterations needed to observe good results would shrink. This can be clearly seen in all of the plots of mean rewards and lengths of each model.

As mentioned before, the goal of task 1 was to reach a position right of the parking goal, with an angle close to -30°. As seen in Figure 51, the jump in rewards at around iteration 120k was due to the agent reaching the goal state. Any training after this point was to optimize its performance and

reach the goal with minimal penalty. The training was stopped at iteration 240k which took 6 hours to train using device 1 in Table 9.

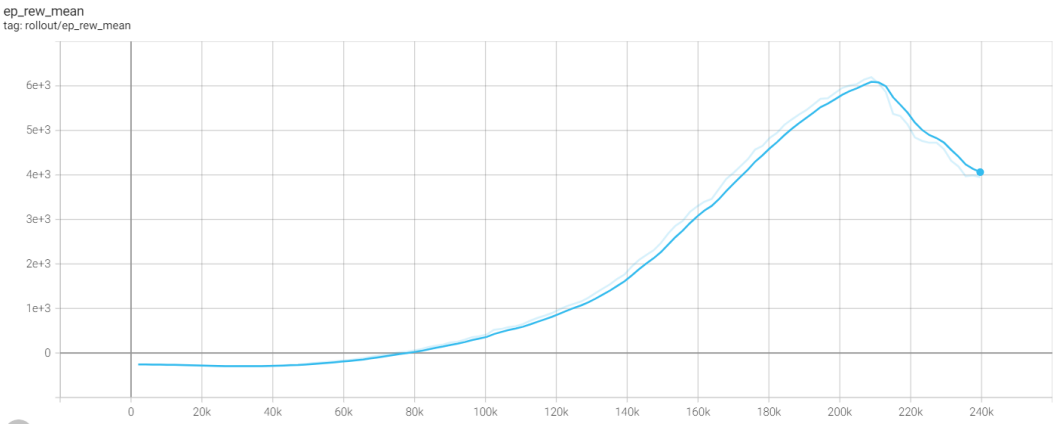

Figure 51: Mean episode reward of task 1

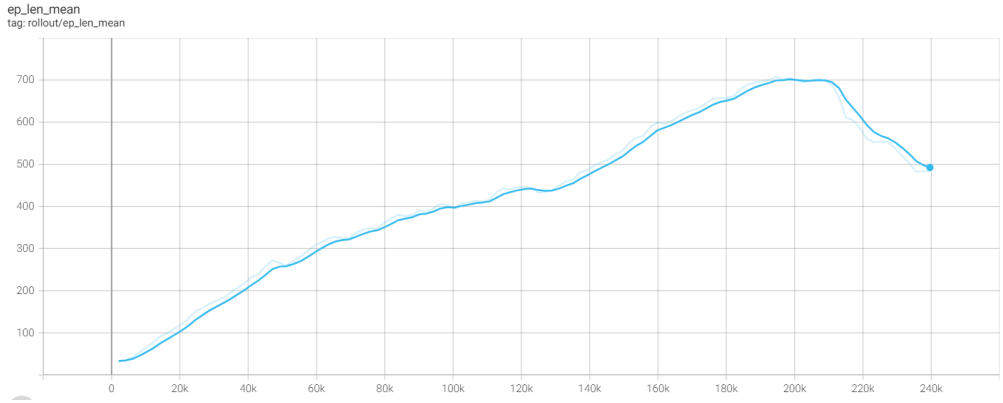

Figure 52: Mean episode length of task 1

Task 2 would be considered the most complex one in all 3 models. Regardless, it reached the goal state at iteration 400k, which took 14 hours to train. The model continued training as an effort to optimize the actions taken to reach this goal. The training was stopped at iteration 1 million, which took around 17 hours to train.

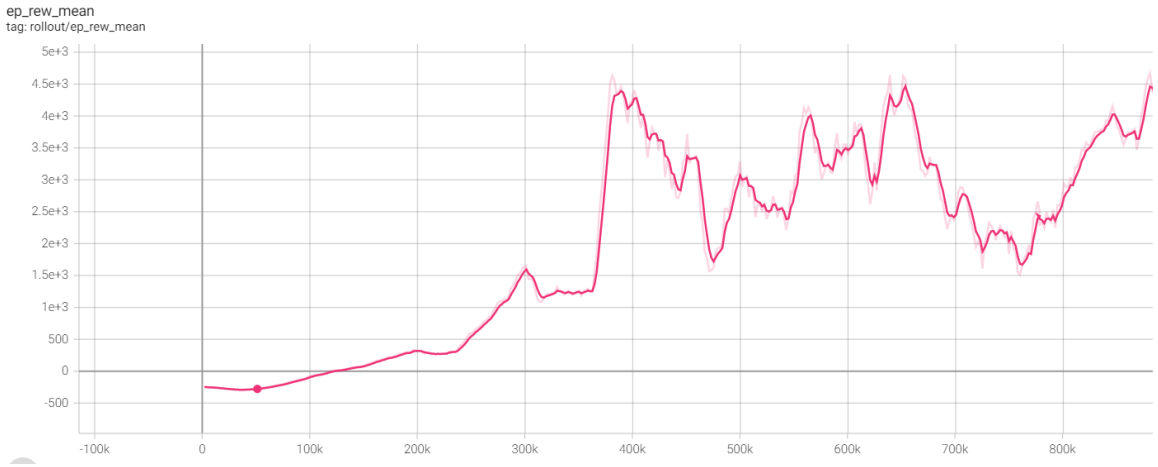

Figure 53: Mean episode reward of task 2

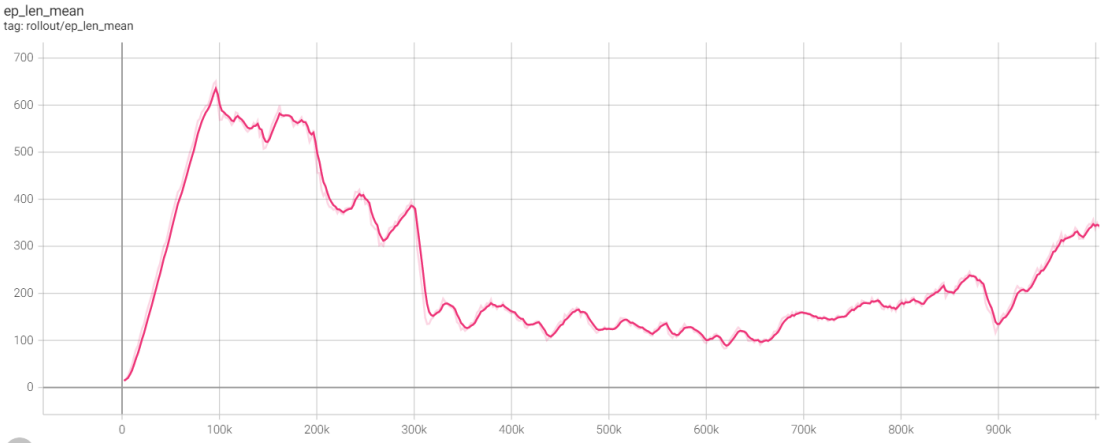

Figure 54: Mean episode length of task 2

The final model, which is the simplest, was designed to make the agent reverse and park with no collisions. It took the least time between the 3 models to stabilize on good results, i.e., reaching the goal state. The total training time for this model was 4 hours, relatively low compared to the others.

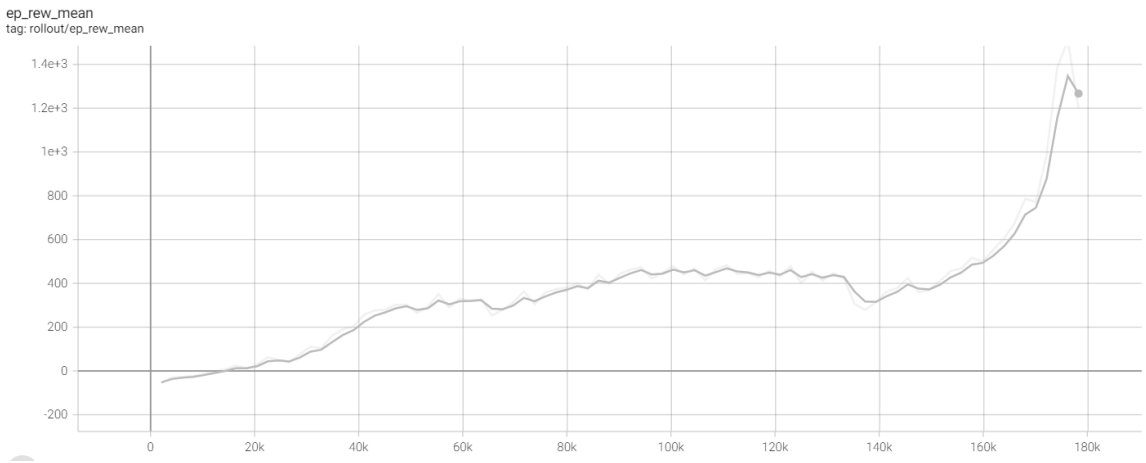

Figure 55: Mean episode reward of task 3

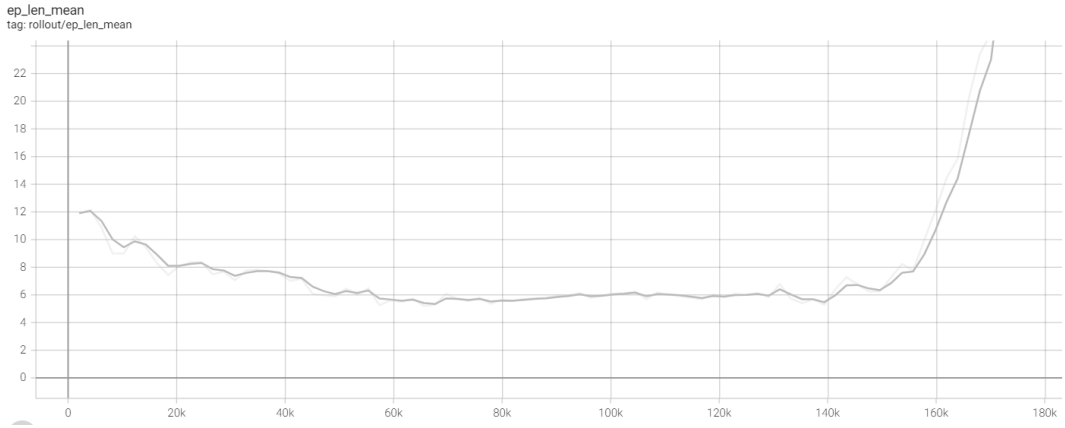

Figure 56: Mean episode length of task 3

4.3.1 Improvements and Optimization

After developing a model that achieves the objective of the system, the time for improvements and optimization commenced. Improvements range from optimizing the models, i.e., minimizing the total out new reward systems to achieve different results, e.g., actions taken to reach the goal state, to trying forward parking.

- Only Receive Rewards with Correct Movement

An experiment where the agent only received a reward when it gets closer to the goal was implemented. This means that when the agent deviates from the parking, it would get zero rewards and not a negative value as before. This way, it would not get stuck in a position that generated 'false positive' rewards, i.e., rewards that falsely reinforce unwanted actions. The results of such system are represented in Figure 57 and Figure 58.

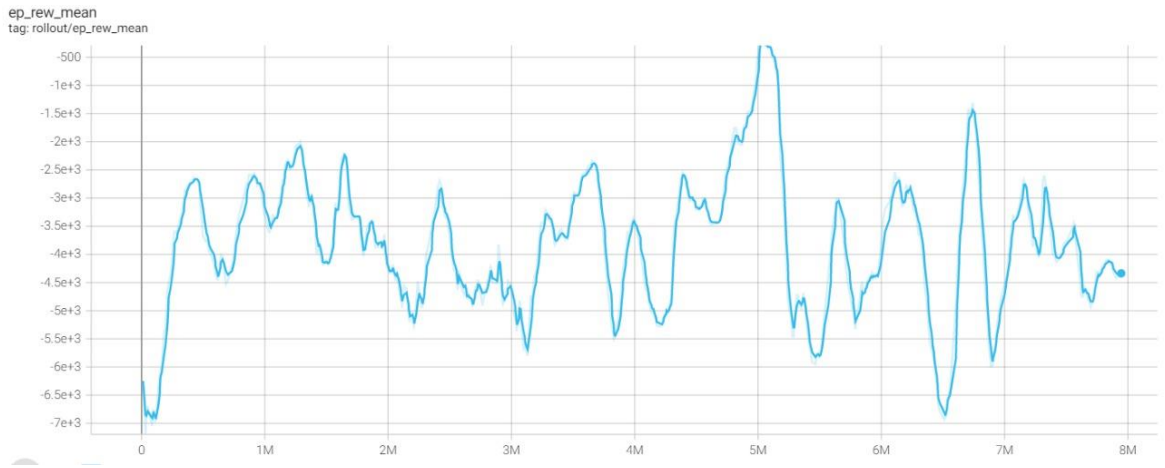

Figure 57: Mean episode reward of a reward system experiment; only receiving rewards when moving towards the goal

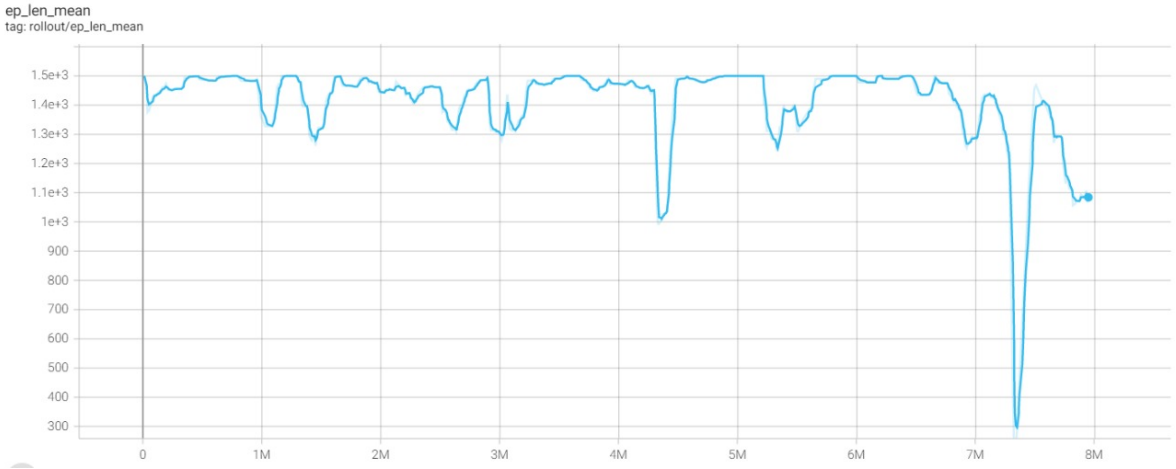

Figure 58: Mean episode length of a reward system experiment; only receiving rewards when moving towards the goal

It can be concluded from the graph above that the learning curve was unstable, oscillations and sharp edges occurred frequently throughout training. This system resulted in a policy that navigates the agent completely above the parking goal, but not allowing it to enter the boundaries of it. This was due to the high penalty assigned to the line crash. Also, this could be the reason for the oscillation in the rewards, i.e., crashes when it gets close resulting in a high penalty, then not crashing but trying to enter the parking, and so on. A combination of this model and the task 3 model could be developed and attain better results.

- Optimal Task 2

As mentioned in 3.5.2.3, a more optimal model was developed for task 2, to try to minimize the action taken to get aligned with the parking. That model was trained on a different reward function

that depends on the RMSE between the current and goal states. As seen in Figure 59 and Figure 60, the policy converged to an optimal solution where the agent takes fewer actions to get aligned with the parking. The results of this model were the best yet, but when applied to hardware, it failed to achieve these results. This could be due to the inconsistencies and errors between simulation and hardware. Meaning that even though a model is not optimal, but is responsive and adapts to change, then it would produce better results than a perfect one in the simulation. A simulation could be thought of as an ideal scenario where some external parameters are not taken into consideration, e.g., tire friction, motor speed, or tilting angle of a vehicle.

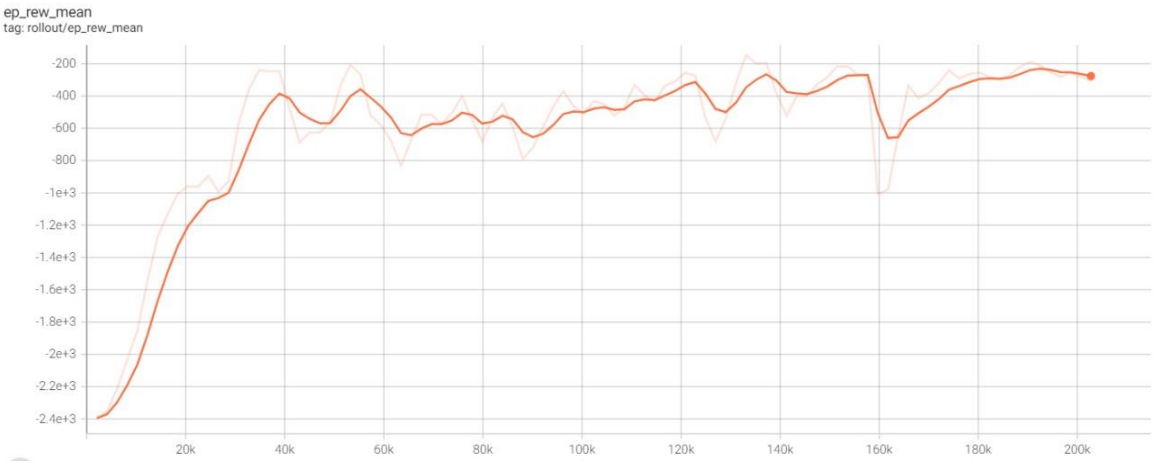

Figure 59: Mean episode reward of an optimal task 2 model

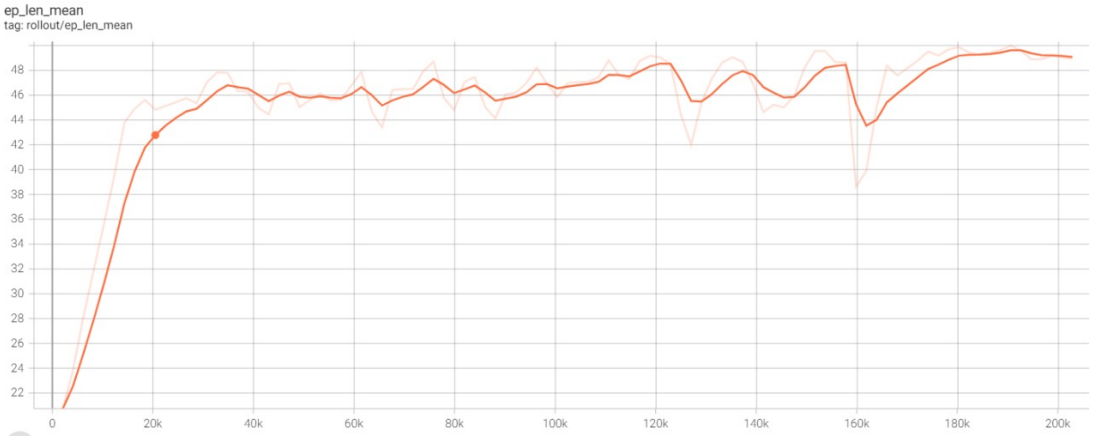

Figure 60: Mean episode length of an optimal task 2 model

- Forward Parking

The final enhancement of the RL model was to try to make the agent park forwards instead of reverse. This model would only modify task 2 where the objective heading would be 90° instead

of -90°. As with the previous enhancement, this model did not produce any notable results when applied to hardware. The results of this model are presented in Figure 61 and Figure 62.

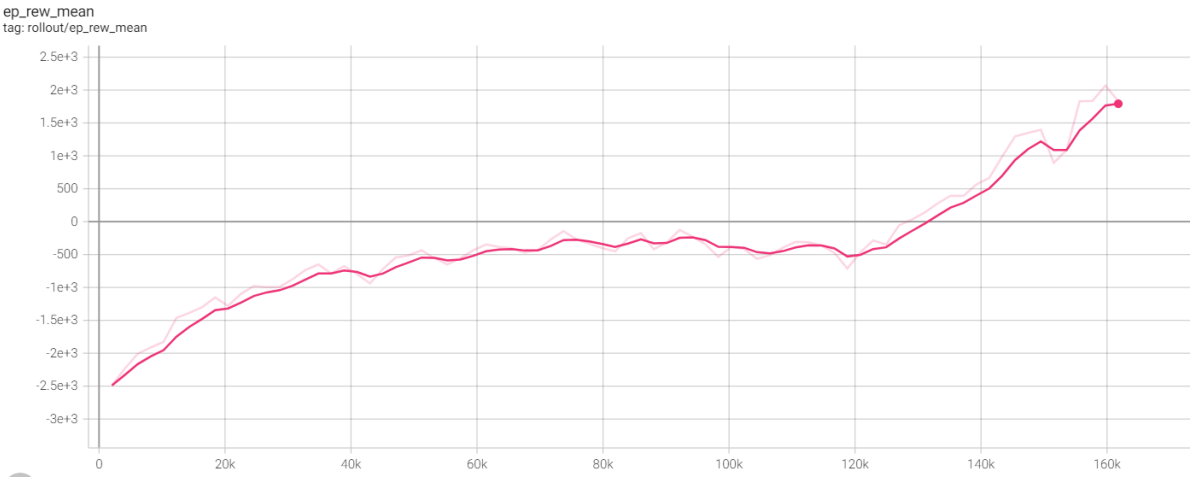

Figure 61: Mean episode reward of forward parking model

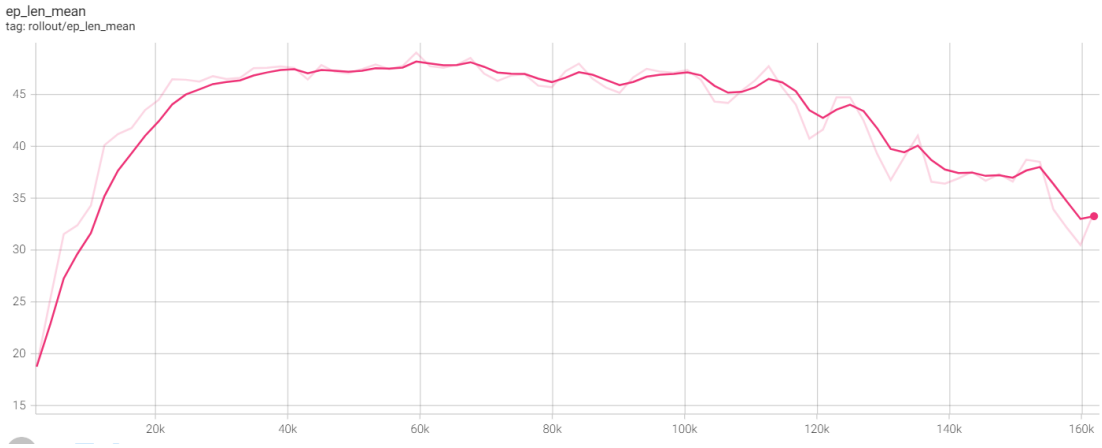

Figure 62: Mean episode length of forward parking model

4.4 System Testing

This section discusses the results of different test samples collected from the system. The samples depicted different scenarios and corner cases with which the system would deal with. This process could be considered a quality control stage in the system design pipeline. The analysis of system performance and how well it accomplished the requirements would be held. The importance of the testing process lies in the following outcomes:

- Proposing any improvements to the system design

- Analyzing the extent of fulfillment of the design requirements
- Debugging and finding any corner cases not handled
- Performance tracking

The criteria of evaluation were the number of successful parking, number of line and object collisions, total time taken to park, and number of actions taken. A successful parking would be a parking that results in a vehicle that is parked inside the boundaries of the parking with no collision with the line. The line collision flag would represent a line crash that happened while the vehicle was parking. The time and total actions taken would represent the efficiency of the model.

4.4.1 Park in a Single Parking Spot

The first test case was performed on a single parking spot, as shown in Figure 63. The objective of this test was to check if the model was handled accurately and transferred smoothly to hardware. The test was initiated from the same starting position which was at the entrance of the parking, and the results are shown in Table 10.

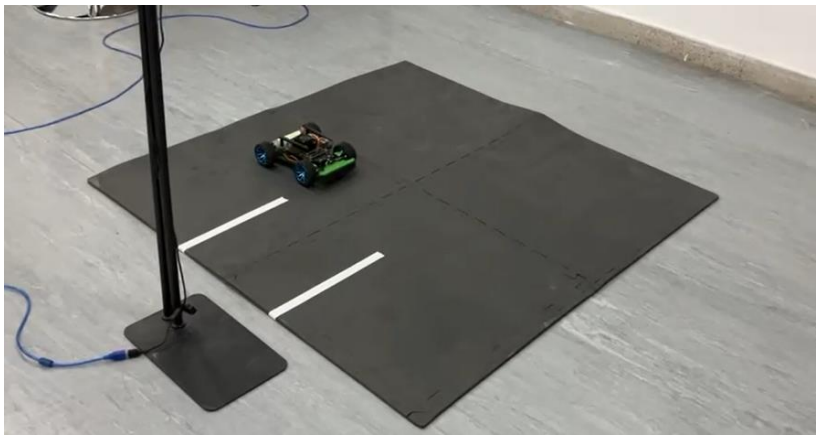

Figure 63: First test case

Table 10: Test case 1 results

Sample	Successful parking	Line collision	Time (s)
1	**1**	**0**	**15**
2	**1**	**0**	**15**
3	**1**	**0**	**14**
4	**1**	**0**	**18**
5	**1**	**0**	**23**
6	**1**	**0**	**15**
7	**1**	**0**	**16**
8	**1**	**0**	**17**
9	**1**	**0**	**14**
10	**1**	**0**	**20**
11	**1**	**0**	**13**
12	**1**	**0**	**14**
13	**1**	**0**	**15**
14	**1**	**0**	**15**
15	**1**	**0**	**15**
16	**1**	**0**	**22**
17	**1**	**0**	**11**
18	**1**	**0**	**15**
19	**1**	**0**	**15**
20	**1**	**0**	**16**
Sum	**20**	**0**	**Avg 15.9**

As seen from the results, the successful parking accuracy reached 100%, and line crashes never occurred. The most notable result of this test case was the time to park. The average time for the vehicle to park was 15.9 seconds, which is way below the required time of 90 seconds. This optimized performance demonstrates the success of the RL model, which opens the room for further enhancement in other areas, e.g., security, user experience, different methods of parking, or adding more parking spots.

4.4.2 Park in all Parking Spots (Empty)

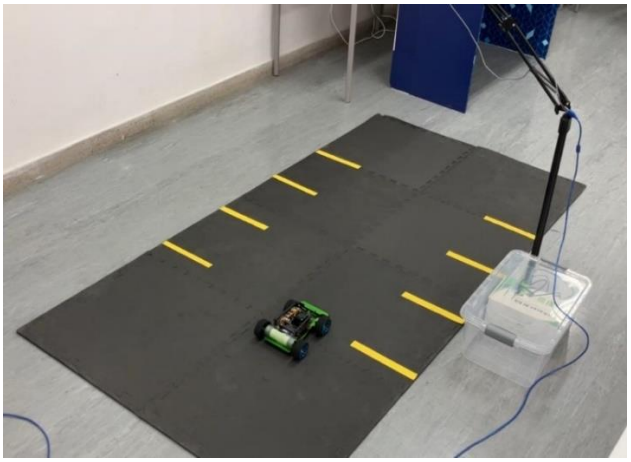

Figure 64: Second test case

After confirming the success of the RL model in the most basic case, the parking setup was constructed and ready to be tested. Each parking was numbered as in Figure 65, the numbering was not arbitrary; even numbers were at the bottom, and odd at the top. The test was conducted by making the vehicle park in each spot 10 times. Note that here, the preprocessing and postprocessing layers are needed to handle the inputs and outputs of the model. The logic behind the handling of different parking positions is illustrated in Figure 66. Note that the actual navigation of the vehicle consists of two parts, a deterministic one, and a model-based one. The first was responsible for moving the vehicle to the entrance of the parking, and the latter was to initiate the parking procedure.

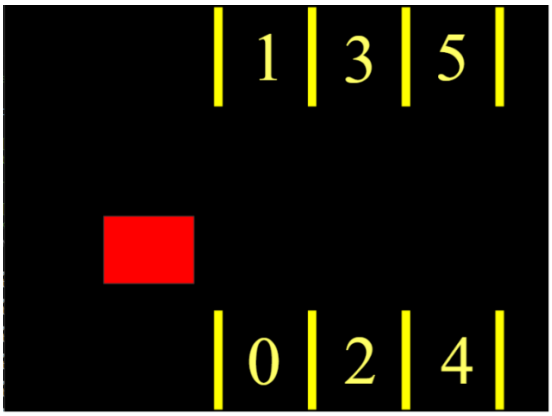

Figure 65: Parking numbering

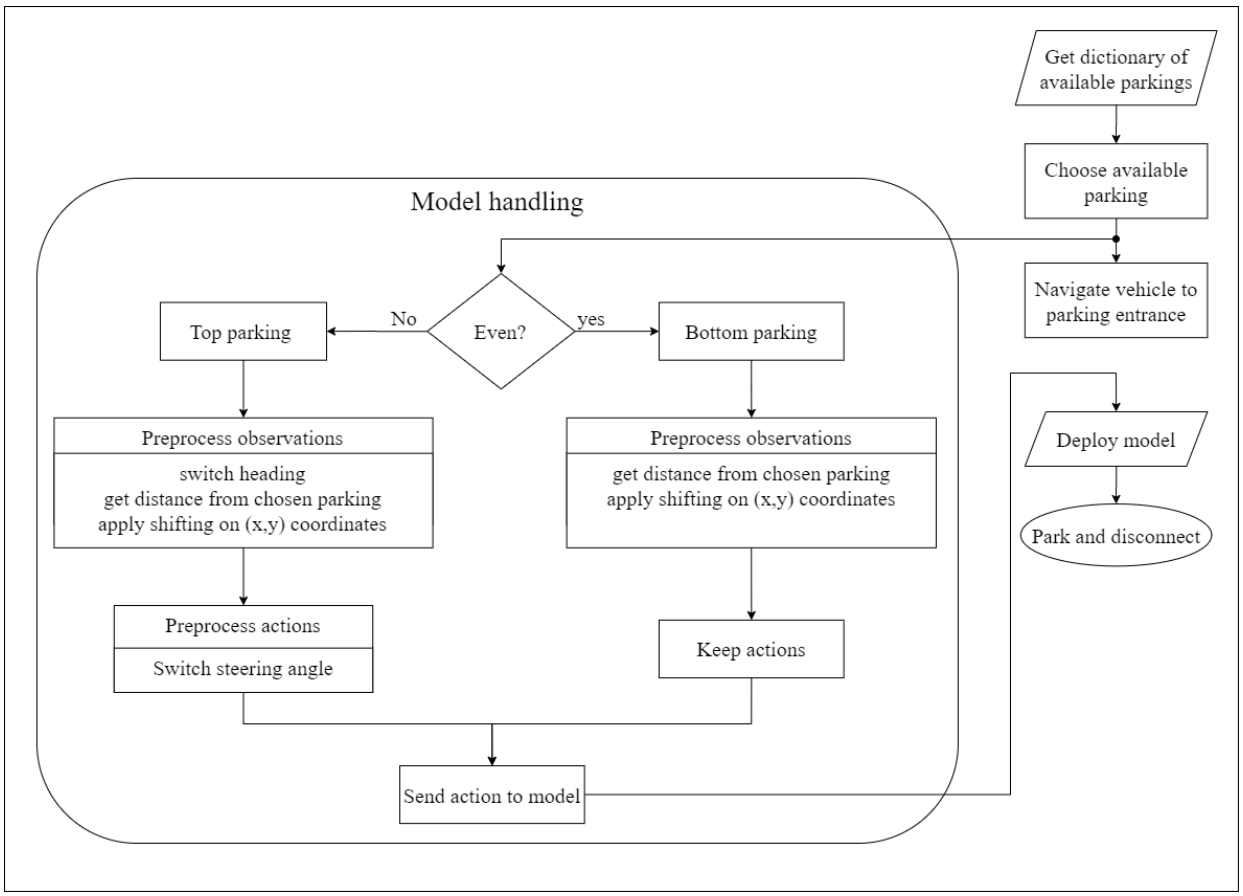

Figure 66: System flow of multiple parking spots

The test results of this case are shown in Table 11. It can be noticed that the rightmost parking locations were responsible for the errors in the model. The more the parking setup is expanded the more room for error occurs. This could be due to the error in the input to the model, i.e., the error in the dimensions of the image on its edges. Thus, if an exact top-view image was generated, then the model would perform better. But still, the successful parking rate reached 95%. The average time it takes to park was still relatively low, 22.1 seconds with 60 actions.

Table 11: Test case 2 results

Sample	Parking spot	Successful parking	Line collision	Time (s)	Number of actions
1	0	1	0	16.4	42
2	0	1	0	16.6	40
3	0	1	0	14.8	43
4	0	1	0	18.1	51

Sample	Parking spot	Successful parking	Line collision	Time (s)	Number of actions
5	0	1	0	15.3	44
6	0	1	0	14.6	39
7	0	1	0	15.1	43
8	0	1	0	12.5	36
9	0	1	0	15.2	44
10	0	1	0	14.4	40
11	2	1	0	18.3	46
12	2	1	0	23.3	65
13	2	1	0	16.7	44
14	2	1	0	20.9	59
15	2	1	1	20.9	61
16	2	1	0	23.9	69
17	2	1	0	20.5	58
18	2	1	0	24.1	68
19	2	1	0	25.5	74
20	2	1	0	18.5	53
21	4	1	0	25.0	57
22	4	1	1	17.0	59
23	4	1	0	17.9	52
24	4	1	0	24.4	68
25	4	1	1	21.6	61
26	4	1	0	20.5	51
27	4	1	0	43.6	121
28	4	1	0	29.0	79
29	4	1	0	22.1	61
30	4	1	0	23.3	65
31	1	1	0	18.8	54
32	1	1	0	17.0	49
33	1	1	0	16.5	47

Sample	Parking spot	Successful parking	Line collision	Time (s)	Number of actions
34	1	1	1	16.4	44
35	1	1	0	19.6	57
36	1	1	0	27.8	64
37	1	1	0	24.4	63
38	1	1	0	30.7	79
39	1	1	0	19.4	47
40	1	1	0	26.9	74
41	3	1	0	43.5	92
42	3	1	0	43.1	121
43	3	1	0	20.3	54
44	3	1	0	16.9	47
45	3	1	0	23.1	68
46	3	1	0	29.1	84
47	3	0	1	25.9	71
48	3	1	0	17.5	47
49	3	1	0	18.1	52
50	3	1	0	21.4	62
51	5	1	0	20.6	58
52	5	1	1	19.2	56
53	5	1	0	17.5	50
54	5	0	1	24.9	69
55	5	1	0	27.0	61
56	5	1	0	31.9	72
57	5	1	0	27.0	77
58	5	1	0	19.4	55
59	5	0	1	22.4	55
60	5	1	0	28.0	59
Sum		57	8	**Avg** 22.1	60

4.4.3 Park in all Parking Spots (Occupied)

The next test case had the same logic and control as the previous one, but the difference lay in the handling of occupied parking spots. The detection and handling of this case was thoroughly discussed in 4.2.3. This case included a fully occupied parking except for the goal location, Figure 67 shows a sample of this case when the goal was location 2.

Figure 67: Sample of test case 3

The results of this test case are represented in Table 12. No line nor object collision occurred, and the average time to park was 21.6 seconds.

Table 12: Test case 3 results

Sample	spot	Successful parking	Object Collision	Line Collision	Time (s)	Actions
1	0	1	0	0	13.4	38
2	0	1	0	0	10.7	37
3	2	1	0	0	21.4	62
4	2	1	0	0	22.1	63
5	4	1	0	0	28.4	79
6	4	1	0	0	22.3	63
7	1	1	0	0	24.9	65

Sample	spot	Successful parking	Object Collision	Line Collision	Time (s)	Actions
8	1	1	0	0	20.6	59
9	3	1	0	0	24.8	60
10	3	1	0	0	31.2	84
11	5	1	0	0	17.9	41
12	5	1	0	0	22.0	49
Sum		12	0	0	**Avg** 21.6	59

4.4.4 Safety Tests

4.4.4.1 Park in Fully Occupied Parking Setup

After testing the RL model performance in different scenarios, the following test is specified for the system stability and efficiency. The first test was to command the vehicle to park in a fully occupied parking setup, as shown in Figure 68. This could be considered a safety check on the system, to observe its output in such cases.

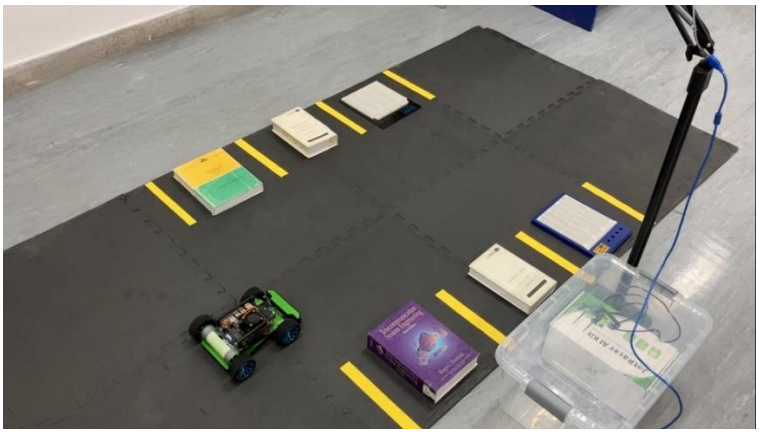

Figure 68: Fully occupied test case

The vehicle in this case did not move, and the server disconnected its connection with it. The prompt sent to the admin is shown in Figure 69.

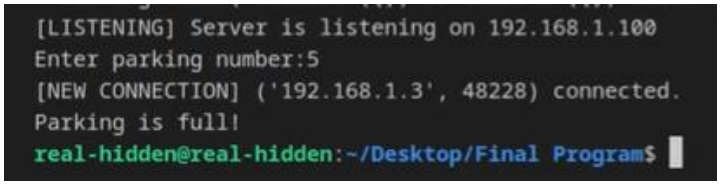

Figure 69: Fully occupied parking results

4.4.4.2 Dealing with Objects in the Parking Environment

To make the system more realistic, the feature of handling objects (vehicle or not) was added. An object that needs handling was considered any object that could possibly alter the parking process, i.e., any object that could possibly collide with the vehicle. As in that case, the parking environment would be considered not safe. A not-safe environment is an environment that includes any object other than the controlled vehicle on the street, considering that the range of movement of the controlled vehicle is the whole street. Note that the system would be able to handle both static and moving objects. This implies that even if the environment was safe at the start, the system would adapt if an object appeared. The results of interrupting the parking with a static or a moving object are discussed next.

- Static Objects

This case was to test the system whether it would start parking the vehicle or not if the environment was not safe, Figure 70. The test was to plant an object on the street and command the system to initiate the parking process. The results were that the system detected the object, did not move the vehicle, and disconnected.

Figure 70: Not safe parking environment

- Dynamic Objects

This test was conducted by starting the parking procedure normally, but then throwing an object on the street while the vehicle was moving, notice the thrown object in Figure 71. The results are shown in Figure 72, where the system detected the thrown object, stopped the vehicle, and disconnected.

Figure 71: Moving object test

Figure 72: Result of a moving object appearing

4.5 Validation of design requirements within the realistic constrains

Requirement	Validation
The system should be able to deploy the trained RL model and control the robot	The hardware and software parts were mapped and tested successfully
The parking setup should contain multiple spots	Six parking spots were designed, 3 top and 3 bottom
The width of the parking spots should be at most 150% of the robot's width	The system was tested on 150% parking widths
The robot should be a prototype of 4-wheeled car	JetRacer Ai kit was used for testing the system, which depicts a real car
The robot's length should be between 15 and 50 cm, with aspect ratios mimicking a real car	The JetRacer's dimensions were 19x25 cm
The system should use a camera, sensors, or a combination of both	The only input to the system was the top view camera, no sensors were needed
The simulation environment should be a replica of the built parking setup	The simulation was built using the 'gym' environment to match the parking setup exactly
The model should be RL-based	The PPO algorithm was used to train the RL model
The physics of the agent should represent the robot, it should move forwards and backward with the ability to turn in both directions	The JetRacer uses a steering mechanism, which is the most similar to a real vehicle
The agent should be trained for perpendicular parking only	A RL model was developed to park the agent reverse perpendicularly
Computer vision algorithms and image processing techniques should be used to process the input image stream	Multiple image processing techniques, e.g., dilation, erosion, and four-point transformation, were used to extract features from the input stream. Additionally, a problem specific object detection algorithm was developed
The collision rate should not exceed 10%	The successful parking rate reached 96%, and the collision rate with other objects was 0%, while the line collision rate did not exceed 9%
The parking time should not exceed 90 seconds	On average the parking time was 19 seconds. The fastest parking took 11 seconds, while the slowest took 44 seconds

4.6 Cost Analysis

The cost requirement of the systems was that a maximum of 200 JDs would be allocated to the design of the prototype. Table 13 summarizes the total cost of the prototype, the actual cost is the amount paid to build the prototype. While the market cost is the cost of a component if it was to be purchased. Notice that the actual cost did not exceed the maximum threshold, this was due to the fact that most components were already provided beforehand and not purchased.

Table 13: Prototype cost

Component		Actual Cost (JD)	Market Cost (JD)
Parking setup	Matt	50.0	50.0
	Camera stand		
	Cables		
	Router		
JetRacer Ai kit		0.0	100.0
Camera		0.0	40.0
Total		50.0	190.0

5 Conclusion and Future Work

It was established that new technologies and ML can drastically improve the driving experience for both drivers and passengers. Also, the possible research and applications of RL in driving assistance systems and autonomous driving are growing immensely.

During the development of this work, it was concluded that RL would make a great solution for autonomous parking tasks. This work provided the design and implementation of an RL-based autonomous parking system, where an RL model would be developed on a simulation, then, transferred to hardware. The sim-to-real transfer was the most intimidating part of the work, as not much previous research was done on the problem. Even when done, not much tackled the parking task.

The system relied on a single input taken from a top-view camera planted at the top of the parking. It is worth mentioning that most work done in autonomous driving would rely on sensors or a combination of sensors and cameras. Depending on a single input may be thought of as inefficient or too simple, but one of the objectives of this work was to make the design as simple as possible. Every feature that was needed for the model was extracted from the image and no sensors were needed, thus, achieving the goal. Additionally, this type of input would make the system applicable to any type of vehicle. This generalization could make the system considered an application of smart buildings design.

The testing results proved that you do not need the most complex simulation environment to transfer the learned policy to hardware. As long as the policy is optimal, the features (inputs) were extracted accurately, and the model was handled correctly (mapping between simulation and hardware), then the results should not fail (or be that different from simulation). Keep in mind that the design of simulation is crucial as the more it represents the real environment the higher the accuracy of transfer.

The system was tested on a variety of cases, from parking in a single position, to parking in a fully occupied parking. The successful parking rate reached 96%, while the collision rate with other objects/vehicles was 0%. The line crash collision reached 9%. All these results adhere to the system design requirements. Of course, further improvements could be made with more training time.

Keep in mind that this work was a prototype of the system, which means that if it was moved to production, then it would grow in complexity. The first thing that would need to be implemented is to have a controller that could translate the model's output to commands that control the vehicle. Additionally, more cameras would be planted at the top of every 4 parking spots, not 6 as the decline in accuracy was noticed in the test results of the last two locations. This means that the more spots detected in the image the lower the accuracy gets, as the dimensions on the edges of the frame would fail to represent real ones. An implementation scheme for this case would be to have multiple cameras connected (serial camera). Which ultimately produced an image with frames from all cameras combined.

Another enhancement could be to implement more parking methods, e.g., perpendicular forwards, or parallel. These would need their own reward system and models. Also, note that all testing was done on the JetRacer Ai kit, but if more tests were done on different robots, then they would become more concrete.

An optimization idea that could be implemented in the future is to update the policy when testing the model on hardware. This way, the model would adapt to any inconsistencies between simulation and hardware, and the policy would become optimal. This was not done due to the design of the gym environment, it would not allow the policy to be updated from external sources, only if it was commanded from the environment directly.

6 References

[1] A. Ziebinski, R. Cupek, H. Erdogan and S. Waechter, "A Survey of ADAS Technologies for the Future Perspective of Sensor Fusion," *Computational Collective Intelligence,* pp. 135-146, 2016.

[2] F. Zafari, S. A. Mahmud, G. M. Khan, M. Rehman and M. Zafar, "A Survey of Intelligent Car Parking Systems," *Journal of Applied Research and Technology,* vol. 11, pp. 714-726, 2013.

[3] M. Veres and M. Moussa, "Deep Learning for Intelligent Transportation Systems: A Survey of Emerging Trends," *IEEE Transactions on Intelligent Transportation Systems,* vol. 21, pp. 3152-3168, 2020.

[4] R. B. Kiran, I. Sobh, V. Talpaert, P. Mannion , A. Sallab, S. Yogamani and P. Perez, "Deep Reinforcement Learning for Autonomous Driving: A Survey," *IEEE Transactions on Intelligent Transportation Systems,* pp. 1-18, 2021.

[5] D. Moreira, "Deep Reinforcement Learning for Automated Parking," 2021.

[6] P. Zhang, L. Xiong, Z. Yu, P. Fang , S. Yan, J. Yao and Y. Zhou, "Reinforcement Learning-Based End-to-End Parking for Automatic Parking System," *Sensors,* vol. 19, 2019.

[7] B. Thunyapoo, C. Ratchadakorntham, P. Siricharoen and W. Susutti, "Self-Parking Car Simulation using Reinforcement Learning Approach for Moderate Complexity Parking Scenario," in *17th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2020.

[8] L. Lai, "Automatic parking with Q-Learning," 2018. [Online]. Available: https://github.com/leoll2/Autoparking.

[9] S. Lange, M. Riedmiller and A. Voigtländer, "Autonomous reinforcement learning on raw visual input data in a real world application," in *The 2012 International Joint Conference on Neural Networks (IJCNN)*, 2012.

[10] W. Zhao, J. P. Queralta and T. Westerlund, "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey," *2020 IEEE Symposium Series on Computational Intelligence (SSCI),* pp. 737-744, 2020.

[11] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540,* 2016.

[12] E. Leurent, "An Environment for Autonomous Driving Decision-Making," GitHub repository, 2018. [Online]. Available: https://github.com/eleurent/highway-env.

[13] Mathworks, "Reinforcement Learning Toolbox," 2019. [Online]. Available: https://www.mathworks.com/products/reinforcement-learning.html. [Accessed 26 Nov. 2021].

[14] Mathworks, "Automated Driving Toolbox," 2018. [Online]. Available: https://www.mathworks.com/products/automated-driving.html. [Accessed 18 Nov. 2021].

[15] M. Holen, K. Knausgard and M. Goodwin, "An Evaluation of Autonomous Car Simulators and their applicability for Supervised and Reinforcement Learning," *IEEE Transactions on Intelligent Transportation Systems,* vol. 99, pp. 1-18, 2021.

[16] A. Juliani , V. Berges, E. Teng , A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar and D. Lange, "Unity: A General Platform for Intelligent Agents," arXiv preprint arXiv:1809.02627, 2020. [Online]. Available: https://github.com/Unity-Technologies/ml-agents. [Accessed 4 Dec. 2021].

[17] A. Dosovitskiy, F. Codevilla, A. Lopez and V. Koltun, "CARLA: An Open Urban Driving Simulator," *arXiv preprint arXiv:1711.03938,* 2017.

[18] Stanford Artificial Intelligence Laboratory et al., "Robotic Operating System," 2018. [Online]. Available: https://www.ros.org.

[19] M. Al-Nuaimi, S. Wibowo, H. Qu, J. Aitken and S. Veres, "Hybrid Verification Technique for Decision-Making of Self-Driving Vehicles," *Journal of Sensor and Actuator Networks,* vol. 10, p. 42, 2021.

[20] A. Arrabi and H. Daoud, "Graduation project code," [Online]. Available: https://github.com/AhmadArrabi/highway-env.

[21] P. Goldsborough, "A Tour of TensorFlow," *Computing Research Repository (CoRR),* 2016.

[22] A. Kanervisto, C. Scheller and V. Hautamaki, "Action Space Shaping in Deep Reinforcement," *Computing Research Repository (CoRR) - arXiv,* 2020.

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

[24] HP, "HP Pavilion Gaming Laptops," [Online]. Available: https://www.hp.com/us-en/shop/mdp/laptops/pavilion-gaming. [Accessed 23 5 2022].

[25] Asus, "Asus TUF Gaming Laptops," [Online]. Available: https://www.asus.com/Laptops/For-Gaming/TUF-Gaming/ASUS-TUF-Gaming-A15/. [Accessed 23 5 2022].

Appendices

6.1 Environment Characteristics

Any developed gym environment is inherited from the default 'Env' class, which has the following mandatory members:

- action_space: Inherited from the 'Space' class and provides a specification for allowed actions in the environment
- observation_space: Inherited from the 'Space' class and specifies the observations (states) of the environment
- reset (): This method resets the environment to its initial state
- step (): This method allows the agent to take an action and returns information about the outcome of the action, e.g., step reward and the end-of-episode flag

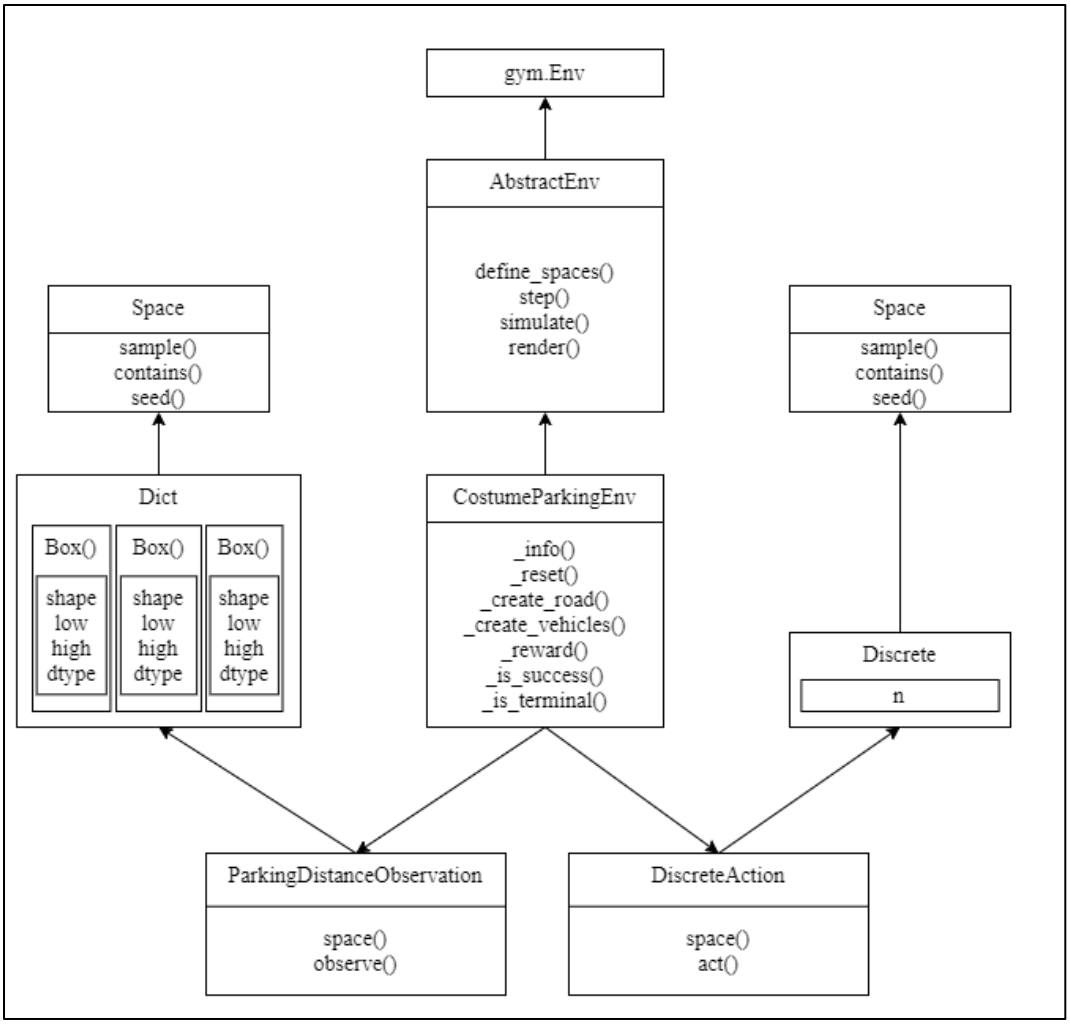

Figure 73: Class hierarchy in the simulation

The built environment was class-based, i.e., object-oriented programming was heavily used. All code provided for the simulation can be found in the appendices. This structure was preferred as the gym environment in its nature is also class-based. Figure 73 illustrates the structure of the parking environment built.

The base class of the developed parking environment was AbstractEnv, which inherits the default gym environment. The abstract environment is a generic environment for various tasks involving a vehicle driving on a road. Modifications on this base environment would be in the form of changing the size of the vehicle and roads, with adding parking-related features and methods, e.g., updating the goal state. Table 14 gives an overview of the methods used and how they are related to the parent and child environments.

Table 14: Methods used in all environments

Method	AbstractEnv	ParkingEnv	Parameters
Define_spaces()	Set the types of spaces of observation and action	Inherited from AbstractEnv	None
Step()	Perform an action and step the environment dynamics (takes an action as a parameter)	Inherited from AbstractEnv	Action
Simulate()	Perform several steps of simulation with constant action	Inherited from AbstractEnv	Action
Render()	Create a PyGame based window viewer to render simulation images	Inherited from AbstractEnv	None
_info()	Return a dictionary of additional information about the step	Overwrites AbstarctEnv to return more data about the step	Observation Action
_reset()	Reset the environment to its initial configurations Call define_spaces() Reset flags and time (step counter)	Overwrites AbstarctEnv to initialize parking-related variables and flags Create roads and vehicle	None
_create_road()	Not implemented	Create a road composed of straight adjacent lines	Parking spots
_create_vehicle()	Not implemented	Create vehicles and add them to the road	None
_reward()	Not implemented	Return the reward associated with performing a given action and ending up in the new state	Action

_is_success()	Not implemented	Check whether the current state is a goal state	None
_is_terminal()	Not implemented	Check whether the current state is a terminal state	None

The customization of the graphical interface of the roads is done through the _create_road() and _create_vehicle() methods. There, it was possible to change the parking setup dimensions, e.g., parking width, spacing of the street, and vehicle size.

Later, as will be further discussed in 3.5.2.3, multiple reward systems were analyzed and tested. All implementation of these reward systems was done through the compute_reward() method, which is called inside the _reward() method.

A keynote in the design of the simulation was to carefully consider its speed (frequency). The speed would be represented as the number of frames taken per action. E.g., if 20 frames were taken per action then the speed would be relatively low compared to when 1 frame per action was taken. In the first, the action taken to move the vehicle would stay constant for 20 frames which makes it more noticeable, while in the latter, the movement would be robust, and actions would change rapidly. The major obstacle in the proposed design was the deployment of hardware, meaning that everything considered in the simulation should adhere to the vehicle's physics and be as realistic as possible. The robust movement of the agent, when the frames taken per action were low, did not achieve the aforementioned requirement. Thus, the frequency was modified to be a fixed value that leads to the most realistic movement of the agent, more on this topic will be presented later in the chapter.