Universidade do Minho
Escola de Engenharia

Dinis Sanches Fernandes

A computer vision based navigation system for an autonomous vehicle model in a controlled environment using reinforcement learning

Dissertação de Mestrado
Mestrado Integrado em Engenharia Eletrónica Industrial e Computadores

Trabalho realizado sob a supervisão de:
Professor Doutor Fernando Ribeiro

janeiro 2023

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Índice

2 Acknowledgements

I want to take this moment to thank everybody that contributed directly or indirectly to the development of this dissertation. All of the people mentioned had a positive impact on my life by helping me grow, learn, or just by making my days better, this is what made me able to face all the challenges during these five years.

First I want to thank my parents Jorge and Manuela, my brothers Gil and Diogo, and my grandparents Natércia, Alzira, António, and Adolfo for making me the person I am today and for so many other things.

I want to thank my princes Madeleine, for supporting me when I needed the most and for making me the happiest man in the world.

I want to thank my supervisor Professor Fernando Ribeiro for helping whenever I asked for help and for introducing me to the laboratory of automation and robotics where I developed most of the dissertation and where I met many amazing people.

Finally I want to thank my close friends from Guimarães whom I shared many priceless stories: Tech-não, Spark, Alibabá, Fátima, Priest, Zombie-Pit, Amnésia, Viciadinha, Caixas, Bob, Galitos, Rammstein, Extrolha, Marroquina, Só-quer, Zé relojoeiro, Hulk, Graxas, MalhaBacas, and Central.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Guimarães, January 2023

Dinis Sanches Fernandes

4 Introduction

4.1 Motivation

Technology has been responsible for shaping lives in ways that could never be imagined before.

The development of the personal computer (PC) allowed people to have an accessible and intuitive interaction with the ever-evolving semiconductor technologies, presently it is an essential tool in people's daily lives. The development of the internet allowed people to access information in ways that were very difficult to imagine before. These devices are so essential to human lives that people don't appreciate them that much anymore, but there was a time when people had to live without them. For most, it is impossible to imagine a lifestyle without electricity but for a long time, humans relied only on fire as a source of energy. This is the power of Technology, something that today may seem impossible tomorrow may be just another basic part of our lives.

The book "AI - 2041" (Lee & Chen, 2021) describes 10 technologies that artificial intelligence (AI) will make available. One of the technologies described in the "The Holly Driver" chapter is autonomous driving. In the mentioned chapter the author depicts a world in which autonomous vehicles are a part of people's daily lives, in some parts of the world. In this world autonomous vehicles (AVs) allow people to live without the struggles of heavy traffic congestion which is a problem that has been around for many years and no solutions seem to have been found. This problem influences people's moods and productivity daily.

Lex Fridman, in an MIT lecture (*Self-Driving Cars: State of the Art (2019)*, 2019), also justified his motivation around AVs on the number of lives this technology will save, which is a legit concern taking into to account that "every year the lives of approximately 1.3 million people are cut short as a result of a road traffic crash" (WHO, 2022). Traffic can also become a problem for ambulances when carrying urgent patients. AVs will help mitigate this problem as priority vehicles will circulate much faster, therefore having a faster response to people in need.

Another important aspect of autonomous driving is the possibility of facilitating the access of people with less mobility anywhere. This lack of mobility may be due to many reasons such as age or handicapped and this technology will highly improve people's lifestyles by facilitating their mobility and improving their autonomy.

Today may seem a bit futuristic to imagine cities without massive traffic congestion, roads without

accidents, or cars without drivers but this will be one of the breakthroughs of our generation and it will become part of our lives just as today is the electricity, PCs or the internet. One day society will look back to road casualties and road traffic as a thing of the past and won't even understand how it was possible to live with it daily, just like today's people can't live without electricity.

4.2 Objective

The objective of this dissertation is to create a control system for the navigation of an autonomous vehicle model in a controlled environment. The control system has to be developed using reinforcement learning methods that input the frames of a camera (or cameras) and output the wheel's speed and angle. With a consistent model developed in simulation, the following objective is to use the trained model to control the vehicle in the real world. The vehicle and environment model inside the simulation should be as close as possible to the real world so that the model is easily adapted. Another objective is to study the possibility of optimizing the algorithm by training it in the real world.

The purpose of these objectives is to test the capability of reinforcement learning algorithms when applied to autonomous vehicles. Test reinforcement learning algorithm capabilities to interpret computer vision data and output suitable actions. The development of an environment, the autonomous vehicle model, and the respective actuators and sensors will allow others to test any control algorithm for autonomous driving.

Testing reinforcement learning (RL) algorithms' capabilities to achieve different goals are of interest to the scientific community because it allows developers to find out where the algorithms are having success and where they are failing. This diagnostic is what allows the constant improvement of algorithms. The same concept applies to autonomous driving, testing different approaches allows developers to identify what works best when developing the control system.

5 State of the art

5.1 Autonomous Driving

This section exposes the theoretical concepts of any autonomous driving system and analyses the autonomous driving industry, namely:

- Basic concepts of a self-driving vehicle
- Different levels of autonomous driving systems
- Different types of sensors used in the industry and respective strengths and weaknesses
- Main ways companies are using sensors to develop autonomous vehicles and what are their plans for the future

5.1.1 Perception, Localization, Planning and Control

"An autonomous vehicle is capable of using sensors to get a picture of its operating environment and make driving decisions based on that data without the intervention of a human operator" (Synopsys, 2022) That being said, the system of an autonomous vehicle gathers data from the sensors to obtain meaningful information about the environment which is then used to navigate through the environment. This ability to interpret the sensor's data in a meaningful way and the ability to control the vehicle's route is the basic competencies of an autonomous vehicle.

According to (Pendleton et al., 2017) "The core competencies of an autonomous vehicle software system can be broadly categorized into three categories, namely perception, planning, and control."

- Perception refers to the ability of an autonomous system to collect information and extract relevant knowledge from the environment
- Planning refers to the process of making purposeful decisions typically to bring the vehicle from a start location to a goal location while avoiding obstacles and optimizing over designed heuristics.
- Control competency refers to the vehicle's ability to execute the planned actions which have been generated by the higher level processes.

In a nutshell, perception is the ability to obtain data from the sensors and process it to take the important information from it and with this information, the system uses its planning ability to make choices such as: deciding what route to take from point A to point B and decide when to overtake another vehicle. The planning consists of higher-level processes that have to effectively be executed by the control ability. The control ability consists of lower-level processes that use the actuators to execute the planned actions.

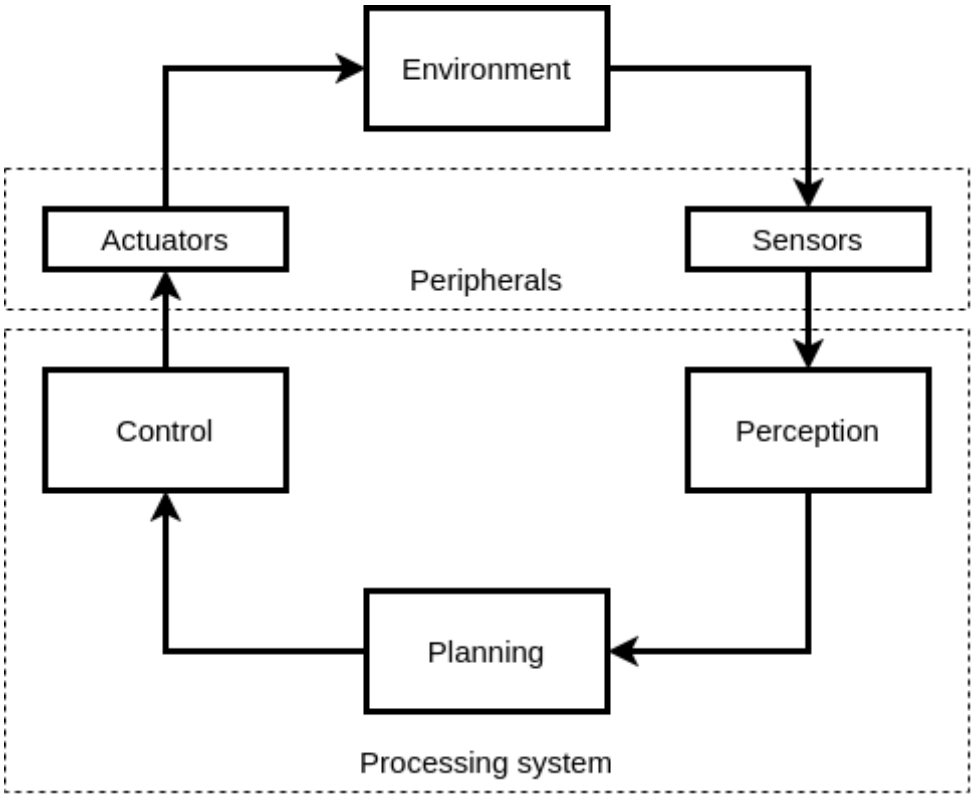


Figure 5.1: Perception Planning Control

Using sensors and feature extraction methods to process the received data the perception system acquires relevant data from the environment. The planning mechanism uses this data to ”plan” the best course of action. The control system receives ”orders” from the planning system and executes the planned actions directly on the actuators.

5.1.2 Levels of Driving Automation

A fully autonomous vehicle can drive anywhere without any human control and without previous knowledge about the location in which the vehicle is driving. Fully autonomous vehicles are not a reality yet, but companies have been developing systems with an increasingly higher level of automation. The effort of these companies resulted in a set of systems designed to help the driver - ADAS (Advanced Driver Assistance System). The most commonly used ADAS system is cruise control. This system allows the user to set the vehicle to a constant speed while having the foot off the throttle. This is especially useful when driving on the highway because trips are usually longer and the car speed is mostly constant. Cruise control allows the user to abstract himself from a component of driving by introducing automation in the system. These systems introduce some level of automation to the driving, by helping the driver with some tasks. The more driving systems are introduced to the vehicle the more autonomous it gets, while not being fully autonomous. To make the distinction between each level of automation a table was created (from SAE J3016™, 2021). This table consists of 6 levels, going from zero (No automation) to five (Fully autonomous car), each level describes the requirements for a system to be considered at that level. Figure 5.2 describes what each level of automation represents.

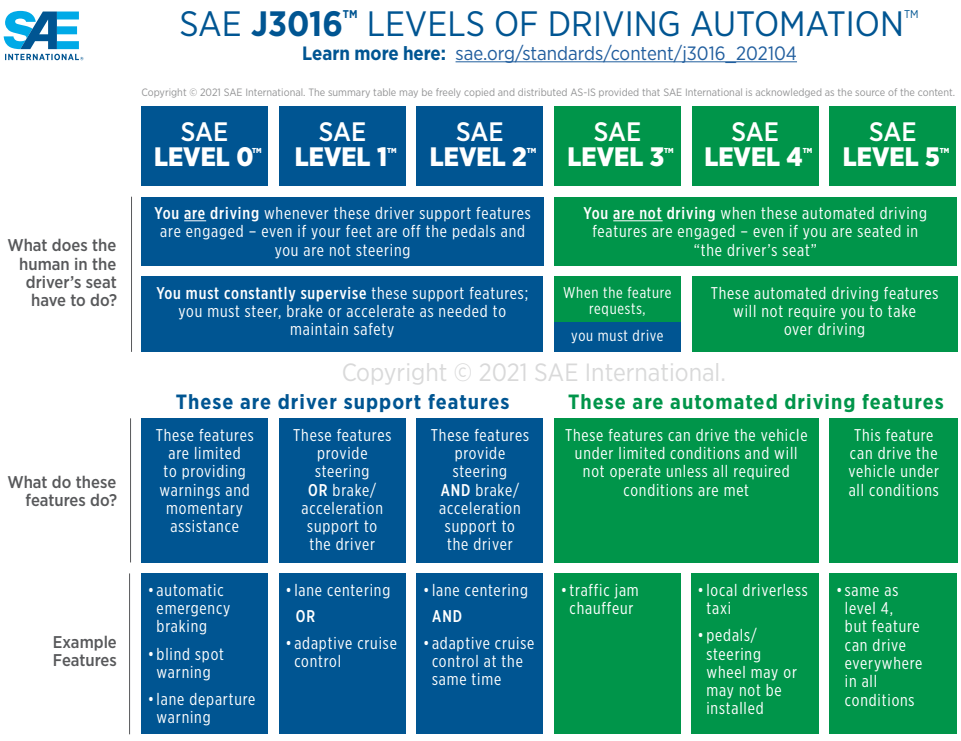


Figure 5.2: SAE Levels of Driving Automation (from SAE J3016™, 2021)

5.1.3 Sensorization of Vehicles

The perception system input of an autonomous vehicle is the data retrieved from the sensors. This subsection presents:

- The main sensors used in car industry
- How each sensor works
- The overall strengths and weaknesses of each sensor.

a) LIDAR

The LIDAR acronym means Light Detection and Ranging (Mazzari & Mazzari, 2021). The way this sensor works is very similar to a sonar, but instead of using sound, it uses light. It is a remote sensing technology that measures the distance between itself and a target (Mazzari & Mazzari, 2021).

The LIDAR has a transmitter and a receiver. The transmitter transmits laser beams in different directions. The laser beams are then reflected by the object hit. The laser beam is then received by the LIDAR receiver. The time it takes for the laser beam to return is called Time of flight (ToF). The ToF is used to calculate the distance between the LIDAR and the object. This process of sending a laser beam, receiving it, and calculating the distance is carried out in many different directions. With the data given by the sensor, it is possible to create a Point Cloud which consists of a map of the surroundings. The map may be 1D, 2D, or 3D but the one used in autonomous driving is the 3D Point-Cloud map.

The LIDAR is a really powerful sensor, as it outputs data that allows algorithms to accurately and efficiently build a 3D representation of the surroundings. Yet the LIDAR's capabilities are very limited when it comes to object recognition, and extreme weather (heavy rain due to the light's refraction) can affect its normal behavior. Another LIDAR disadvantage is that it is an expensive sensor.

b) Radar

Radars have been around for a long time and are used in many different fields, such as meteorology, road traffic control, etc. There are many different types of RADARs, one of the most popular and relevant to autonomous driving is called FMCW - Frequency-Modulated Continuous-Wave (Cohen, 2021). Radars measure the distance to an object and some radars are capable of measuring velocity simultaneously.

The radars have good range flexibility because they can measure objects that are less than a centimeter away but they can also measure objects hundreds of meters away.

It works by emitting electromagnetic (EM) waves that reflect when colliding with an obstacle (Cohen, 2021). The time it takes for the emitted frequencies to travel back is used to calculate the distance between the radar and the obstacle.

The obstacle's speed is calculated through the Doppler effect. The Doppler effect consists of a physical property in which the wavelength varies accordingly to the obstacle's speed.

Despite operating like the LIDAR, this sensor has a few differences, just like the LIDAR the Radar can not identify objects due to its low-definition modeling. To contrast, it is capable of operating under extreme weather.

c) Cameras & Computer Vision

Cameras are a very common gadget in people's daily lives. Nowadays in western countries, most people have at least one camera constantly in their pockets. It allows people to store photos or videos. A video is just a very fast succession of taking photos. The photos of a video are usually called frames. To understand how cameras can be used in machine learning algorithms it is important to first understand how this technology gathers data.

A camera consists mostly of a set of lenses and an electronic chip. The camera's lens focuses the light into the Chip. This Chip is composed of many photosensitive sensors. Each sensor is capable of measuring the amount of light. For each pixel, there are one or more sensors. Each sensor captures the amount of red, green, and blue for the respective pixel. The picture captured by the camera is a matrix of all the pixels.

Through the described method it is possible to store a matrix of pixels called frames. Each frame represents the light that crossed the camera's lens at a given time.

The use of cameras is great because, with this type of sensor, it is possible to translate what a human sees into digital values.

The increase in computer power and organized data available in recent years allowed the intensive development of a branch of AI called Machine Learning. Machine Learning associated with the data obtained through cameras has made huge breakthroughs in the field of computer vision.

The machine learning algorithms applied to computer vision use convolutional neural networks (CNN). CNN consists of mathematical models that require a process called training to accomplish the desired function. The training process requires a vast source of data and computer vision offers Machine Learning

exactly that.

CNN and computer vision allowed the creation of powerful algorithms capable of taking human-like decisions, such as identifying real-world objects with low error rates. Examples can be

- Image classification sees an image and can classify it (a dog, an apple, a person's face).
- Object detection can use image classification to identify a certain class of image and then detect and tabulate their appearance in an image or video.
- Object tracking follows or tracks an object once it is detected.
- Content-based image retrieval uses computer vision to browse, search and retrieve images from large data stores, based on the image's content rather than metadata tags associated with them.

Cameras operate very differently when compared to the previously referenced sensors. It is a High-resolution sensor that can identify objects due to its high resolution, it is cheap when compared with LIDAR and it outputs human-like data. Cameras also have their disadvantages like outputting 2D data and difficulties with low light environments. The 2D representation of cameras is a problem being solved by cameras such as the intel realsense. The intel realsense returns 2 matrices one with the color frame and the other with the object's distance in the frame to the camera.

5.1.4 Automobile Industry State of the Art

Presently many companies are developing autonomous vehicles, but there are mainly two approaches to this problem. On one hand, some companies rely on LIDAR sensors and sometimes even highly detailed maps. These companies accept the limitations of cameras and use other sources of data. On the other hand, some companies believe that the LIDAR is not needed because the information that it offers can be gathered with a camera that has a higher definition and is cheaper. These companies use cameras and robust computer vision algorithms to gather important environmental features.

This subsection presents:

- Two main companies researching and developing AVs
- An overview of the companies plans to commercialize its vehicles
- The sensors each vehicle uses
- The level of autonomy of each vehicle

a) Waymo

Waymo is a company that started with the google's self-driving car development in 2009 and since then they have made significant improvements in the Hardware of its vehicles and mainly in its embedded software, the Waymo Driver. This company offers two types of services, Waymo One and Waymo Via where the first consists of an autonomous taxi service in Metro Phoenix (located in Arizona, United States of America), and the second consists of an autonomous transportation service. This section describes the Waymo one service. Users of this service make use of an app to request a vehicle, the vehicle drives to point of pick-up set by the user and then drives the user to the set point of destination. The trip can be personalized to stop in different locations before the final destination.

Waymo vehicles uses the three previously described sensors, LIDAR, RADAR, and Cameras but it also uses a localization system. For a vehicle to be able to operate in a new area, Waymo has to first map the territory with a large detail, from lane markers to stop signs to curbs and crosswalks. With the data given by the localization systems, the vehicle can plan the route and know where the traffic signals and other important road points are. When the vehicle arrives at locations like a crossroad it uses its sensing to analyze the state of other vehicles and traffic signals to then act accordingly.

Waymo vehicles drive without any human intervention or supervision, the human does not at any point touch the steering wheel to control the car during a regular trip. These vehicles have some limitations, because of their dependency on a detailed localization system design by Waymo. This company designs vehicles that can drive autonomously yet are limited to a small area of operation therefore the level of autonomy is ranked Level 4 (Ackerman, 2021).

b) Tesla

Tesla is an American-based company "founded in 2003 by the engineers Martin Eberhard and Marc Tarpenning Eberhard served as its CEO and Tarpenning served as CFO" (Reed, 2020). The founders of this company aimed to build a fully electric vehicle that served "practical specifications that could arguably meet consumer needs". At the time no other company was able to develop a vehicle that could fit the consumer's needs, since all of them lacked batteries powerful enough to accelerate the car to highway speed, so the project of building an electric sports car was a very ambitious objective, but this is exactly what Tesla did. In 2006 the first Tesla Roadster prototype was announced, and two years later, in 2008, it entered production. This vehicle was powerful enough to accelerate to highway speed, with a top speed of 201 km/h (*Tesla Roadster (first generation)*, 2022) and had an appealing autonomy of 320Km

(*Tesla Roadster (first generation)*, 2022) The only problem with the vehicle was that it priced out most consumers, with a release cost over US$100,000. Because of these specifications, the first vehicle of Tesla was a great achievement since no company was able to create an electric car powerful enough to be highway legal. Despite this achievement by 2009, the company was facing some financial problems (Cao, 2021). In 2008 Elon Musk, who was the company's Board of Directors chairman since its investment of US$30.000.000, became the CEO of Tesla. In the same year (2008), the plans for the new vehicle, the Model S sedan, were announced. This was still a luxury car but with a much lower price of US$76,000. The Model S prototype was unveiled in 2011 and went into full production in 2012. By 2013 the company was able to present its first quarterly profit and since then they have been releasing not just vehicles but also Solar panels, Solar tiles to be installed on the roof of houses, started an energy distribution service, and most importantly power stations to recharge Tesla's vehicles. An important aspect is that the batteries used in Tesla's vehicles are produced by their own company. The production of batteries was an important investment for Tesla. To improve the batteries production Tesla announced the construction of multiple factories called Gigafactory. The Gigafactory is a facility designed for battery production. In 2016 the Model 3 sedan was announced retailing below US$70.000 an even lower price when compared with the previously released vehicle, the Model S sedan. Most recently Tesla has released the Model Y.

Cars produced by Tesla, besides having top of the line electric batteries also carry embedded software with constant improvements. This software includes the Tesla autopilot which offers active safety features and driving assistance features, this is carried out with the data acquired through the sensors. The sensorization of Tesla vehicles is a computer vision based, meaning that its perception uses mainly cameras. These vehicles also use ultrasonic sensors but the control system relies mostly on the data gathered by the cameras. The perception system is composed of three forward cameras, two forward-looking side cameras, one rearview camera, and twelve ultrasonic sensors. The three cameras (Wide, Main, and Narrow camera) mounted behind the windshield provide broad visibility in front of the car, and focused, long-range detection of distant objects. The sensors functions are described below:

- Wide Camera - 120 degree fish eye lens that captures traffic lights, obstacles cutting into the path of travel and objects at close range. Particularly useful in urban, low speed maneuvering
- Main camera - covers a broad spectrum of use cases
- Narrow Camera - provides a focused, long-range view of distant features. Useful in high-speed operation
- Forward Looking Side Cameras - 90 degree redundant forward looking side cameras that look for

cars unexpectedly entering the same lane as the vehicle on the highway and provide additional safety when entering intersections with limited visibility

- Rearward Looking Side Cameras - monitor rear blind spots on both sides of the car, important for safely changing lanes and merging into traffic
- Rear View Camera - The rear view camera is useful when performing complex parking maneuvers
- Ultrasonic Sensors - These sensors are useful for detecting nearby cars, especially when they encroach on your lane, and provide guidance when parking .

The Tesla autopilot includes three main functionalities, the TACC (Traffic-Aware Cruise Control), Autosteer, and Auto Lane Change. TACC is an advanced version of cruise control which when activated by the driver, sets the vehicle's speed to the same as the vehicle in front, this feature may also stop at stop signs and traffic lights if activated by the user. Autosteer adds steering assistance to TACC meaning that the vehicle automatically adjusts the car's steering to stay inside the lanes. The driver has to be aware at all times in case the vehicle's trajectory needs to be corrected, so to force the user to be aware, the vehicle has to detect the driver's hands on the steering wheel, or else it gives notifications to the user. In case the user does not respond to the notifications given by the vehicle the autosteer will be deactivated automatically and the driver will not be able to activate it for the remaining trip. When the autosteer is activated the user may use the auto lane change feature to make the vehicle change lanes automatically, yet as usual the driver has to be aware of other vehicles on the road before initiating the lane changing process. The described feature before being available to the user has to go through a process of hardware calibration, this is a one-time process that takes a couple of hours of driving on clearly marked roads. Tesla is constantly improving their embedded software and because of this, the vehicle includes the possibility of upgrading its embedded software anytime the company releases an update.

The autopilot developed by Tesla consists of driving assistance functionalities, which provide the driver with steering, braking, and acceleration support, this means the human is responsible for driving at all times, the software only gives the driver support features. The described functionalities make the Tesla autonomous driving system level 2 autonomy (Morris, 2022). Level 2 autonomy consists of a driving assistance system that helps the driver with lane centering and adaptive cruise control, being this achieved through the autosteer and TACC systems respectively.

5.1.5 Waymo Vs Tesla

At first sight, the level 4 of automation present in Waymo vehicles is much more impressive when compared to the level 2 of automation present in Tesla's autopilot driving assistant system, yet the Waymo features come at an expensive price. Waymo's driving system relies heavily on highly detailed maps. Maps may be simple to manage in a small and organized area, such as Metro Phoenix city, but when considering more extensive areas such as a whole country or even the whole world, these maps raise severe scalability problems since they rely on a constant update of the road signs, constructions, possible accidents that influence the traffic, etc. Besides the problem of scalability Level, 5 of autonomy is the future and for this level of autonomy to be achieved the vehicle must be able to drive in any location in any conditions, without the need for previous experience and for this the use of maps does not allow a vehicle to reach level 5 of autonomy. Another thing that might be a problem in the future for Waymo is the use of LIDAR. This highly complex sensor adds about US$7.500 (Moreno, 2021) to the vehicle's price. This cost is an improvement to US$75.000 (Moreno, 2021) the sensor used to cost. Cameras are a very cheap technology, and therefore it seems difficult for LIDARs to compete with cameras when it comes to price. Presently the LIDAR still has an advantage when compared to cameras, and that is the fact that cameras retrieve 2D data and the LIDAR retrieves data that can be used to build the car's surroundings 3D map. This is the main advantage the LIDAR has, yet this is something that neural networks can solve as described in the paper "Pseudo-LiDAR from Visual Depth Estimation: Bridging the Gap in 3D Object Detection for Autonomous Driving" (Wang et al., 2019). The referenced paper describes how, with neural networks and cameras, it was possible to have a depth perception with 74% accuracy in a 30 meters range. This type of signal processing makes LIDAR not very useful. Elon Musk described LIDAR as "a fool's errand" (Burns, 2019) and affirmed that "Anyone relying on LIDAR is doomed. Doomed!" (Burns, 2019) Because LIDARs are "expensive sensors that are unnecessary."(Burns, 2019) and because they are unnecessary they end up being "a whole bunch of expensive appendices." (Burns, 2019) Elon ended this controversial statement with the obvious "Like, one appendix is bad, well now you have a whole bunch of them, it's ridiculous"(Burns, 2019). Developments in computer vision algorithms and the investment of companies such as Tesla make a question of time before the scientific community finds a way of retrieving data through cameras in such a way that makes LIDARs useless, therefore the development efforts in autonomous driving should be put to camera-based systems. Tesla has not been able to present an autonomous vehicle with a higher level of autonomy because the company is putting all its efforts into creating technology that will be the future of autonomous driving. The Tesla approach is challenging but

much more rewarding in long term. On the other side of the spectrum there are companies like Waymo that are developing vehicles that today show much better results but in long term will be surpassed.

5.2 Reinforcement learning

Machine learning has three branches of algorithms: supervised, unsupervised, and reinforcement learning. Both supervised and unsupervised learning algorithms are mainly used to label input data. Meaning that these algorithms receive a dataset and output features about the input. The difference between supervised and unsupervised learning is that unsupervised learning gets non-labeled data, and supervised learning gets labeled data.

Reinforcement learning is very different from other machine learning algorithms. RL's approach is inspired by the training of animals. To teach a dog how to do something the trainer waits for it to do an action if the action is good it gets a treat if the action is bad it does not get a treat and sometimes even physical punishment. In RL there is an agent, the algorithm makes interpretation of the environment, the state, and the agent chooses an action accordingly to the present state and executes it in the environment. After executing the action in the environment the algorithm makes another interpretation, the next state, and assigns a reward. Both the reward and state are input to the agent and the process repeats itself across a number of iterations. The state works as a way of interpreting the environment so the agent can understand it and the reward is a way of knowing if the taken action is good or bad. For a state received the agent executes an action, and for this reason, it can be called state-action pair. Depending on the RL algorithm there is a step that gets the rewards acquired for the state-action pair and changes the agent to maximize the rewards obtained. After training the agent it should be able to execute the target task by just receiving the states. This high-level overview (5.7) explains any RL algorithm from the basics like Q learning to the most advanced like deep Q-learning (Mnih et al., 2013), deep deterministic policy gradient (Lillicrap et al., 2015), proximal policy optimization (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017), etc.

Machines do not have the perception that humans have of the real world so it is the developer's job to create mechanisms that allow machines to train agents to execute a target task. This subsection explains the main mechanisms that allow RL algorithms to learn.

a) Autonomous vehicle track

The environment in which the vehicle learns how to drive is the Portuguese national robotics champi-

onship autonomous driving track. The autonomous driving challenge has many different tasks to accomplish but the sole objective of this dissertation is to make a consistent control system capable of navigating around the track using as input a camera and as a control system an RL algorithm.

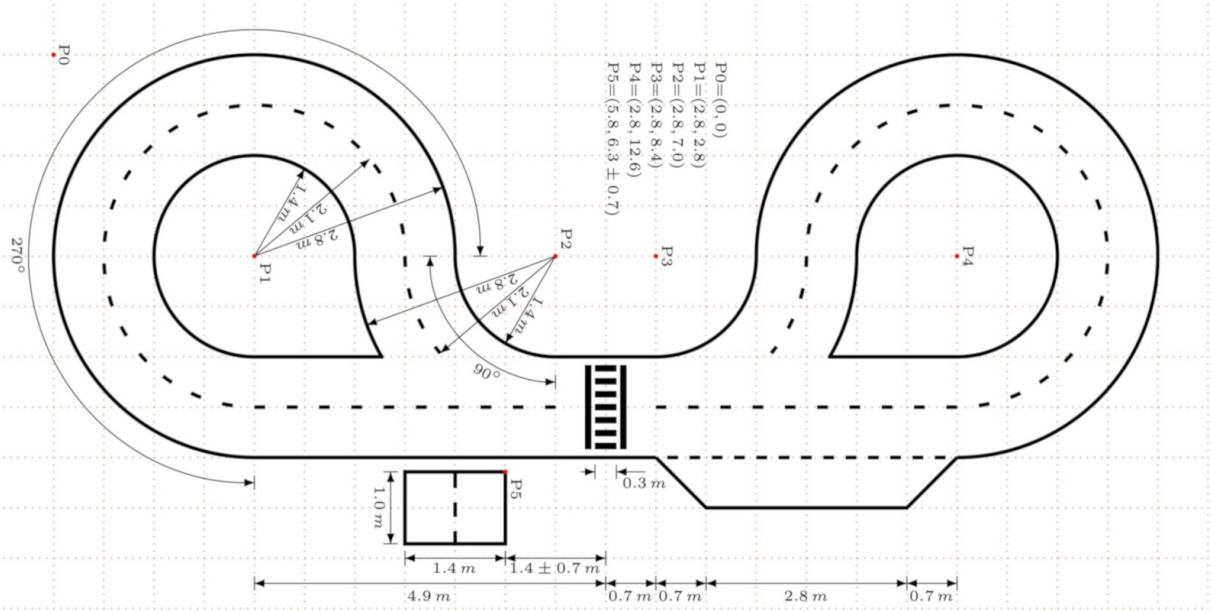


Figure 5.3: Portuguese national robotics championship autonomous driving track

5.2.1 Farama Foundation Gymnasium - Cart pole

OpenAI is a company that invests in the research and development of AI-based systems. Their goal is to develop AI systems that outperform humans in economically valuable tasks. OpenAI considers its mission fulfilled by helping other developers achieve the common goal. This intention is supported by the release of the gym library, which is an open-source library used to test and benchmark AI algorithms. The Gym library maintenance recently passed to a non-profitable organization named Farama Foundation which changed the library's name to gymnasium (Farama-Foundation, 2022).

This library offers a set of environments controlled by a set of functions. The cross-referenced python code 5.1 shows a basic example of the car pole environment execution. Info is a parameter that gives information about the simulation that is not supposed to be used to train algorithms.

- Line 1 - gymnasium library is imported
- Line 2 - Initiate *env* object, this object gives access to most library functionalities (Ex: *reset*, *step*, *action_space*), the parameter received is the target environment name

- line 4 - Resets the simulation and returns *observation* which is the state
- line 6 - Cycle that implements the maximum number of iterations
- line 8 - *Step* function receives as parameters an action to be executed, executes the action for a time step and returns some data. *Observation* is the state, *reward* is the reward value, and *terminated*, that returns 1 when the simulation reached a terminal state and 0 otherwise. The parameter set inside *step* function is *env.action.sample* which returns a random action from the environment's action space.
- line 10 - This condition resets the environment if the value returned by the *step* function is 1, else it continues running
- line 13 - Closes the environment

```
1  import gym
2  env = gym.make("CartPole-v1")
3
4  observation, info = env.reset()
5
6  for _ in range(1000):
7
8      observation, reward, terminated, info = env.step(env.action_space.
       sample())
9
10     if terminated:
11         observation, info = env.reset()
12
13 env.close()
```

Listing 5.1: Gym cart pole basic python example

The gymnasium environments are like video games but instead of being controlled by a human player, the objective is to be controlled by an algorithm (agent), cart pole is a gymnasium library environment. The agent's goal in this environment is to control a cart left or right to hold the pole on top for as long as possible, the cart's control is carried out by choosing action 1 or 0. Choosing action 0 makes the car go left and action 1 makes the car go right, the car's speed is not controllable, it either goes full speed left or full speed right.

The *observation* returns important information about the cart and pole model 5.4.

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

Figure 5.4: Cart pole gymnasium environment observation

It is important to notice that the code's flow 5.1 is similar to what is represented in 5.7. The difference is that in example 5.1 the agent takes random actions despite the reward and state returned. The objective of RL algorithms is to use states to gain experience and learn how to better execute the task to maximize the reward obtained.

All information about the environment can be found on the gymnasium documentation page (Farama-Foundation, 2022).

5.2.2 Grid World

Grid World is an environment in which the agent starts at a position, and the objective is to move up, down, left, or right to achieve the end state. There are two positions where the agent reaches the end. There is one position where it gets a positive reward of 1, and in the other, it gets a negative reward of -1. There is also a position with a wall meaning that the agent cannot move to that position. In this example the agent starts at position (2,0), the wall is in position (1,1), the positive reward is at (0,3) and the negative reward is at (1,3).

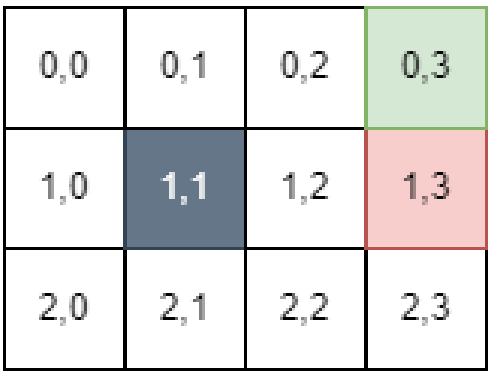


Figure 5.5: Grid World

For explaining purposes, two different variations of grid world will be used: Deterministic grid world and probabilistic grid world. Deterministic grid world does not have probabilities associated with the agent's state transitions. Considering that the agent is at state s and it executes action a the next state s' will always be the same. In probabilistic grid world, there are probabilities associated with state transitions. Considering that the agent is at state s and it executes action a there is a probability associated with what the next state s' is going to be. For the sake of simplicity, the only probabilistic state transition used in probabilistic grid world is the state transition shown in the cross-referenced table 5.1 and all the other state transitions are 100%.

State	Action	Next State	Probability
(1, 2)	UP	(0, 2)	50%
(1, 2)	UP	(1, 3)	50%

Table 5.1: Probabilistic Grid World State Transition

5.2.3 Basic Concepts

The basic concepts subsection explains how a basic RL algorithm is implemented. The objective is to expose how all the basic concepts of RL algorithms merge with the Bellman equation to solve the control and evaluation problem in grid world. The Bellman equation is not used in the algorithm implemented in this dissertation, and grid world is an environment used only for explaining purposes, but it is important to understand this approach to understand the more advanced algorithms and how they work in more complex environments.

a) Environment Agent Feature extraction

The agent is the algorithm's part that makes decisions, receives data from the environment, and learns. The environment is everything outside the agent, and it could be anything, from a video game, a simulation, the real world, etc. The environment interpretation is carried out by feature extraction methods which are designed by the developer and the objective is to translate the environment to features the agent can recognize.

b) States Rewards Actions

The state consists of feature(s) that describe the agent relative to the environment, and a well-designed state stores feature(s) that are useful for the agent to complete the desired task. The feature(s) can be any datatype, a single value, a tuple of values, an array of values, etc. The state is represented with s and the next state is represented with s'. In grid world the only important feature is the agent's position in the environment (grid) so the state will be state = (*agent_position*) Ex: state = (1,0). In cart pole there are more important features to be recognized: the cart position, cart velocity, the pole's angle, and the pole angular velocity. The agent state in the cart pole environment is a tuple of the mentioned features state = (*cart_position*, *cart_velocity*, *pole_angle*, *pole_angular_velocity*) state = (1.5, -2.3, 0.1, 2.1)

The reward is a value the agent receives when it reaches some state, and it should reward the agent for reaching the desired state and punish it for reaching an undesired state. Rewarding the agent means returning a higher reward value, and punishing the agent means returning a negative reward value. So the reward system should return a high reward for reaching the desired state and a lower reward for reaching an undesired state. A reward can be negative when reaching an undesired state. In this dissertation for a more intuitive reward description, a negative reward is referred to as a penalty. Designing a reward system is not straightforward, the RL agents do not perceive rewards like humans do. In the Alpha go documentary Frank Lentz said "AlphaGo says [...] it shouldn't matter how much you win by, you only need to win by a single point" (*AlphaGo*, 2017). The point of this quote is that an RL model does not value big rewards like humans. When it receives a big reward it does not even understand how the set of actions led it to that specific state, it understands better small rewards that evaluate its immediate actions. Grid world is a very simple environment, used for explaining purposes, which is why the positive reward is only given at the end, but in more complex environments, like the cart pole, the reward is given on every iteration. In cart pole a positive reward of one is given on every iteration, independently of how long the agent has managed to hold the pole, and the longer it holds the pole the more rewards it gets. If the reward system was based on checkpoints, it would not be as effective because the agent would have a greater challenge understanding how its action leads to that positive reward.

The agent interacts with the environment by choosing an action for each iteration. The action can be a value or a set of values (an action can set values for multiple actuators). For example, to control a car the driver has to control both the speed and the steering. To create an agent to drive a car it has to be able to control the steering and the speed at the same time. So for each iteration, the agent has to choose an action that consists of two values, the speed, and the steering angle. In the grid world and cart pole

environment, the agent only chooses one action. In the grid world the action can be Up, Down, Left, or Right in cart pole the action can be +1 (Right) or -1 (Left). In this example the agent can only choose an action from a limited action space meaning that the agent only has to choose the action with the largest chance of success. There are other environments where the agent has to choose the action's value, and this is the difference between continuous action space and discrete action space.

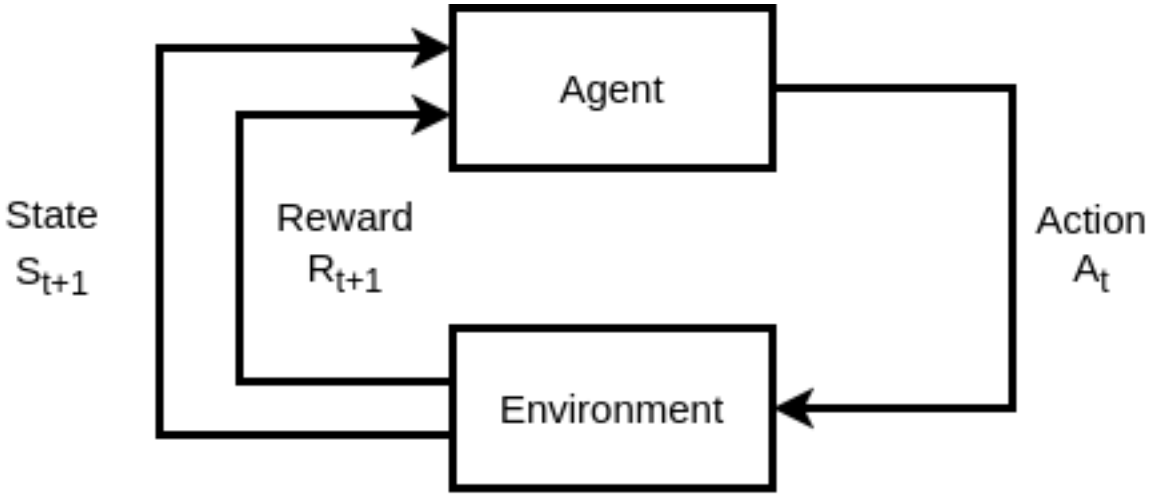


Figure 5.6: Reinforcement learning high level overview

c) Policy

There are multiple definitions for the word policy depending on the context but most of them agree that it consists of a set of ideas that define the course of actions an individual takes given the present state of things. A policy in RL is very similar, as it defines the action the agent takes given state s and is represented by $\pi(a|s)$. There are many ways of implementing a policy, the most basic is using a table that matches an action to a state, and this way the agent can output the best action for a given state. In Grid world the policy chooses the direction to go for a given state. An example of a policy for grid world is:

State	Action
(0,0)	RIGHT
(0,1)	RIGHT
(0,2)	RIGHT
(1,0)	UP
(1,2)	UP
(2,0)	UP
(2,1)	RIGHT
(2,2)	UP
(2,3)	LEFT

Table 5.2: Grid world simple policy example

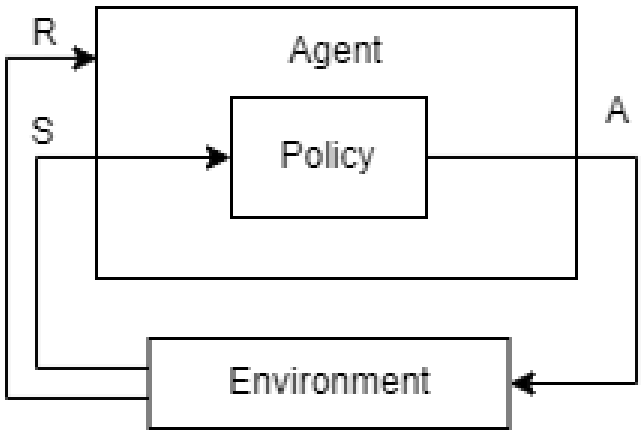


Figure 5.7: RL policy

d) Markov Decision Process

Markov decision process (MDP) (Bellman, 1957) is a concept that defines the basis of RL algorithms. MDP is dependent on other Markov concepts, like the Markov Property, Markov Process, and Markov Reward Process. The Markov property says that being at a state s_t the probability of transitioning to the state s_{t+1} is independent of the previous states $s_{t-1}, s_{t-2}, (...), s_{t-n}$. In grid world, the Markov property works like the following, if the agent follows the policy 5.2 when it reaches $s = (0, 0)$ the action to execute under the policy is RIGHT, so the probability of transitioning to state $s = (0, 1)$ is independent of the previous transitions: from $(1, 0)$ to $(0, 0)$, and from $(2, 0)$ to $(1, 0)$.

$$P[s_{t+1}|s_t] = P[s_{t+1}|s_1, (...), s_t]$$

A Markov process is defined by a state s and a probability P of transitioning to the next state s'.

$$P[s_{t+1} = s'|s_t = s]$$

A Markov reward process is defined by a state, state transition probability, a reward, and the discount factor. This is motivated by the principle that every goal can be described by a set of smaller rewards.

The Markov decision process is defined by a state, a state transition probability, a reward, the discount factor, and an action. The action allows the agent to control the states for which it wants to transition, to receive a reward.

e) Model based vs Model free

When executing an action there is always a probability associated with its outcome. When playing a game like Tic-tac-toe, chess, or go the player tries to predict the opponent's movements to plan the best way of attacking. The other player's move is not certain, but some moves are more likely to execute than others. Model-based algorithms calculate the next state probability according to its action. In the context of a tic-tac-toe game, the agent predicts the other player's reaction to its action, with this information it defines the policy.

A model-free algorithm (Swazinna, Udluft, Hein, & Runkler, 2022) calculates the return of each state through experience. The agent chooses an action a_1 for the present state s_t. The experience acquired by the agent allows it to make a the next state value approximation s_{t+1}. With this value, it can compare the state value reached when taking a different action a_2 for the state s_t. With this data, it improves its policy. It differs from the model-based algorithms (Swazinna et al., 2022) because it does not predict the probability of state transitions, as it chooses the action that leads to the state with the higher value.

f) Episodes Iterations

An iteration is the time step between the agent's actions. For each iteration, the agent takes an action, and it receives the next state and a reward. An episode includes all the iterations from the starting state until reaching a terminal state. An episode is a set of iterations.

This whole process represents an episode in Grid World:

Iteration	State	Action	Next State	Reward
1	(2,0)	Up	(1,0)	0
2	(1,0)	Up	(0,0)	0
3	(0,0)	Right	(0,1)	0
4	(0,1)	Right	(0,2)	0
5	(0,2)	Right	(0,3)	1

g) Control VS Prediction Problem

The objective of RL is to create a policy that is good at solving a task, this is done by running multiple episodes trying different policies evaluating each, and comparing to find the optimal policy. The objective is achieved by solving two problems:

- Control Problem - To find the best policy
- Prediction Problem - To finds the value of each state for the current policy (Modayil, White, & Sutton, 2011)

This is a cycle, where the prediction stage calculates the value of each state and then the control stage with the values calculated creates a policy and executes it.

h) Return

Return is the sum of all possible future rewards. The formula to calculate the return is:

$$G(t) = \sum_{\tau=0}^{\infty} \gamma^{\tau} R(t + \tau + 1)$$

In this formula for a given time step t, $G(t)$ is equal to the sum of Rewards $R(t + \tau + 1)$ multiplied by the discount factor γ raised to the value of τ. The variable τ represents all the following time steps. The discount factor, γ, represents the decaying importance of future rewards, i.e. the further away the reward is from the present time step the lower weight on the return value. To understand how this works mathematically it is easier to represent it as follows:

$$G(t) = R(t + 1) + \gamma R(t + 2) + \gamma^2 R(t + 3) + ...$$

If: $\gamma = 0.5$

$$G(t) = R(t+1) + 0.5R(t+2) + 0.25R(t+3) + ...$$

If: $\gamma = 0.9$

$$G(t) = R(t+1) + 0.9R(t+2) + 0.81R(t+3) + ...$$

The discount factor is a constant that should be regulated to control how far the future reward should be taken into account when calculating the return. This constant is a hyperparameter. Hyperparameters are values that should be regulated by the user to tune the training algorithm (Kiran & Ozyildirim, 2022).

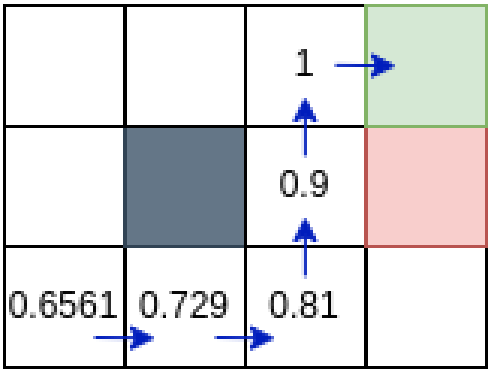


Figure 5.8: Return in Grid World

For example, in an episode the agent at a time step t is at state $s = (2, 0)$, it will follow the trajectory represented by the blue arrows in the cross-referenced figure 5.8 and the discount value is equal to 0.9 $\gamma = .9$ then the return at time step t is calculated as follows:

$$G(t) = R(t+1) + 0.9R(t+2) + 0.81R(t+3) + 0.729R(t+4) + 0.6561R(t+5)$$
$$R(t+1) = R(t+2) = R(t+3) = R(t+4) = 0$$
$$R(t+5) = 1$$
$$G(t) = 0.6561$$

i) Value Function and Bellman equation

The return usually is not directly applied in an RL algorithm and this is because the return associates a set of events (time steps) to be calculated. Instead of using the return to evaluate the potential of a given position, the algorithms use the value function. The value function associates a value given a state. One way to calculate the value of a state is by using the Bellman equation:

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma * V_k(s')]$$
$$\Delta = max_s|v_{k+1}(s) - v_k(s)|$$

Solving the prediction problem is finding a way of evaluating the potential of each environment state. The value function's objective is to solve the prediction problem, it evaluates the potential of each state so the control process can find a way of navigating to the states with higher potential. The Bellman equation is a way to calculate the value function. It is important to understand that the value function is a concept that can be approached using different techniques. The described example represents a way of estimating the value function but there are other methods.

The Bellman equation consists of an iterative process, which estimates the value of each state by calculating a new value at each iteration until the new value gets very close to the previous value. It is a process that repeats itself until it satisfies a condition that determines if the current estimator is close to the estimator calculated in the previous iteration. When the new value is very close to the previous value it means that the present estimation is good and does not need more improvement. The iteration is represented in the equation by k and $V_{k+1}(s)$ represents the new present state s value estimation. $V_k(s')$ represents the present next state value estimation. γ is the discount factor and r is the reward received in the present iteration. $p(s', r|s, a)$ is the probability of going to the next state s' and receive the reward r knowing that the present state is s and the action chosen by the agent is a. This component represents the environment's model. This means the algorithm creates a state transition estimator because when the agent is at a state s and executes the action a usually it is not 100% certain what the next state is going to be. For example, when a football player kicks the ball it is not totally certain that it is going to be a goal, but there is a probability to be a goal and a probability of not being a goal even when shooting from the same position. This concept works for RL algorithms too, when the agent is at a state s and executes the action a usually there is a probability associated with what is going to be the next state s' and the received reward r. Not all RL algorithms use this concept of creating an environment model and this distinguishes a model-free algorithm from a model-based algorithm. $\pi(a|s)$ represents the probability of the policy choosing action a given that the current state is s.

Trying to understand the Bellman equation with mathematics may be a bit abstract so the following pseudocode shows the Bellman equation implementation to evaluate a policy in grid world.

Algorithm 1 An algorithm with caption

while $True$ **do**
    $MaxVariation \leftarrow 0$
    for s in AllStates **do**
        if s not Terminal **then**
            $V_{old} = V[s]$
            $V_{new} = 0$
            for a in AllActions **do**
                for s' in AllStates **do**
                    if $a == \pi(a|s)$ **then**
                        $PolicieValue \leftarrow 1$
                    else
                        $PolicieValue \leftarrow 0$
                    end if
                    $r \leftarrow Rewards(s')$
                    $V_{new} \leftarrow V_{new} + PolicieValue * TransitionProb(s, a, s1) * (r + \gamma * V[s'])$
                end for
            end for
            $V[s] \leftarrow V_{new}$
            $\Delta = |V_{new} - V_{old}|$
            $MaxVariation \leftarrow \max(MaxVariation, \Delta)$
        end if
    end for
    if $threshold > MaxVariation$ **then**
        $Break$
    end if
end while

The objective of this function is to evaluate the environmental states. The value of each state in the Bellman equation is dependent on the policy, which in this example is the cross-referenced 5.2.

Transition Prob is the environment model, which in this example is defined by the user since this environment is made for learning purposes. The version used is Probabilistic grid world, so all state transitions are 100% certain except for the state transition referenced in the table 5.1. For this algorithm to work Transition Prob has to return zero in case the state transition is not referenced in the model.

The variable Max variation is used to register the biggest change in the state value for each evaluation iteration. When the MaxIteration variable value is greater than the threshold, the evaluation process finishes.

This algorithm loops through all the states to evaluate each (line 2), and verifies if the current state s is terminal in which case it will be skipped because terminal states do not get a value. If the state is not terminal the algorithm evaluates the current state s, starting by storing the previously stored current state value $V[s]$ and creating a variable V_{new} with value zero (lines 5 and 6). The variable V_{new} stores the new current state value s. For the current state, the algorithms loop through all the possible actions, and for each action, it loops through all possible states s' (lines 7 and 8). The agent can not just jump from start to finish in grid world so some state transitions are not possible, these state transitions have zero transition probabilities so these impossible state transitions will not have an impact on value calculus. Inside the cycles, the algorithm starts by defining the PolicyValue variable value which is equal to zero if the action a is equal to the action the policy chooses for the current state s if not the value is zero. This is because the policy is deterministic, not probabilistic. The algorithm gets the rewards for the next state s' (line 14) and calculates the current state s new value V_{new} (line 15). The value V_{new} sums the previous value of V_{new} because the state's value is the sum of all possible future rewards and this is the reason for having four cycles, one to repeat the process until MaxVariation is lower than the threshold, one to loop through all the state to calculate the respective value, one to iterate through all the possible action the agent takes at that state and one for all possible next states. The algorithm calculates the difference between the previously calculated state value and the newly calculated state value, the highest difference is stored (lines 18 to 20). The final step is to compare the highest difference when calculating the new value of each state and if it is under the threshold the policy evaluation process finishes.

j) Control Problem and Bellman Equation

The control system objective is to find the optimal policy. The control algorithm uses the value calculated for each state in the evaluation process and creates a policy that transitions to the states with higher

immediate and future value.

Algorithm 2 An algorithm with caption

1: $PolicyStable \leftarrow True$
2: **for** s in AllStates **do**
3: $BestAction \leftarrow None$
4: $BestValue \leftarrow -inf$
5: **if** s not Terminal **then**
6: $a_{old} \leftarrow \pi(a|s)$
7: **for** a in AllActions **do**
8: $Value \leftarrow 0$
9: **for** s' in AllStates **do**
10: $r \leftarrow Reward(s')$
11: $Value \leftarrow value + TransitionProb(s, a, s1) * (r + \gamma * V[s'])$
12: **end for**
13: **if** $Value$ > $BestValue$ **then**
14: $BestValue \leftarrow Value$
15: $BestAction = a$
16: **end if**
17: **end for**
18: $\pi(a|s) \leftarrow BestAction$
19: **if** $BestAction$!= a_{old} **then**
20: $PolicyStateble \leftarrow False$
21: **end if**
22: **end if**
23: **end for**

The variable $PolicyStable$ is initiated, and at the end of this process if this flag is still True it means that the policy remains unchanged and this means that the policy is already optimal. The algorithm loops through all the states in the environment (line 2). Inside the cycle initiates the variables $BestAction$ and $BestValue$, because inside this cycle the algorithm is going to find the best value action and is going to use these variables to register the greatest value and to which action this value is associated. It checks if the present state is not terminal (line 5) and then stores the action that previously was associated with the present state s in the variable a_{old} (line 6). The algorithm loops through all possible actions (line 7)

and initiates the variable value (line 8) that is going to store the total value of each transition done when the agent is at state s and executes one of the actions a. The algorithm loops through all the states to test all the possible next states s' when executing the action a in state s. Inside this cycle it calculates the value of each possible next state s' starting by retrieving the reward r for the next state s' (line 10). The algorithm then calculates the value for each next possible states s' and this uses the value calculated during the evaluation process $V[s']$. After evaluating all the possible states the algorithm compares the previous best value $BestValue$ with the value calculated now $Value$, if the value calculated now is greater it stores the action and the action value in the respective variables $BestAction$ and $BestValue$ (lines 13 to 15). The algorithm stores the $BestAction$ in the policy $\pi(a|s)$. If the present best action is different from the one stored previously it sets the policy as unstable (lines 19 and 20). The process of evaluation and control improvement should be repeated until the policy is stable.

5.2.4 Advanced concepts

The Bellman equation is the basis of RL algorithms but the most powerful algorithms do not use it. The more advanced algorithms, called Deep Learning, use neural networks to create a policy. Neural networks are basically "giant mathematical models" (Lee & Chen, 2021) that are inspired by the human brain.

Neural networks have achieved impressive results mainly in the field of computer vision. The most common example of neural networks is the algorithm that distinguishes cats and dogs. In some cases, this type of algorithm has been able to reach error rates much lower than solutions based on classic linear algorithms.

a) Neural Networks

The basis of neural networks is the neurons. A neuron has an input and an output. The input consists of a set of values (X1*W1, X2*W2, X3*W3). The set of values is then processed in two different operations (Sum, Activation). The first operation (Sum) is very simple, where all the values in the input plus the bias (B) are summed (X1*W1, X2*W2, X3*W3 + B). The first operation product is passed to the second operation (activation). This step consists of associating the previous operation product with the activation function. It is a simple mathematical operation that associates the input (Xsum) to an output (f(Xsum)), using a function (f). The function (f) can have many shapes, depending on the activation function chosen. The activation function chosen in this example is ReLU (Agarap, 2018).

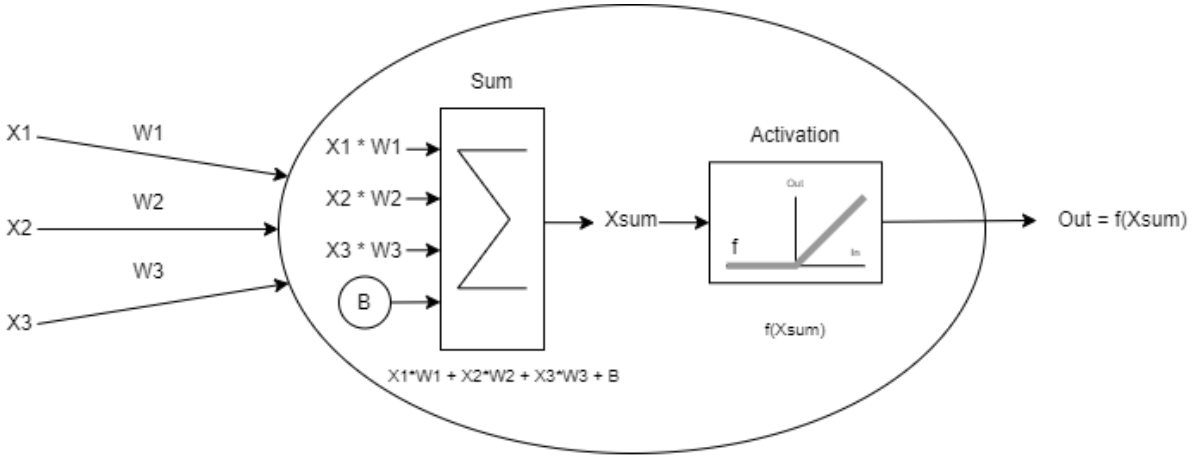


Figure 5.9: Neuron

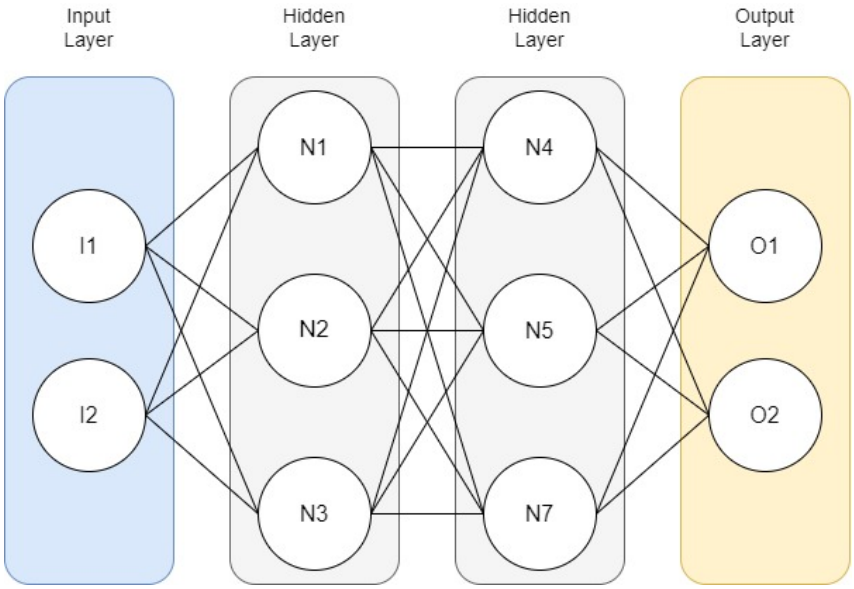


Figure 5.10: Neural Network

A neural network consists of a set of neurons connected in a structured way. In a neural network, there is always an input layer, an output layer, and a variable number of hidden layers. The number of neurons in each hidden layer and the number of hidden layers are defined by the user and should be tuned to obtain the best result possible. A layer consists of a set of neurons. Each neuron of a layer has an input for the output of each neuron (X1, X2, X3) of the previous layer and an output for each neuron of the next layer, this connection between any two neurons has a respective weight (W1, W2, W3). The neuron receives the values from the previous layers multiplied by the respective connection weight (X1*W1, X2*W2, X3*W3). The neuron represented in the 5.9 figure is a representation of any neuron in the neuronal network's second hidden layer 5.10. The neuron N5 receives the values from N1(X1), N2(X2) and N3(X3) neurons. The values received are multiplied by the weight connection and the resulting value

will be the input for the neuron's operations (Sum, Activation). The weights that connect the neurons and the biases of each neuron are tuned during the training process to create a neural network that works for the target task.

b) Weights and Biases

It is important to remember that the weights that connect all the neurons together and the bias of each neuron are the variables that are changed to tune the neural network.

$$W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ (...) \\ W_n \end{bmatrix} \qquad B = \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ (...) \\ B_n \end{bmatrix}$$

The weights and biases are usually referred to as the neural network's parameters and are represented through the θ symbol.

c) Loss Function

The loss function, also known as the cost function, is a way of calculating how badly the neural network performed for a given state. The greater the loss value is, the worse it performed at a given time step. It can be represented with J, L, or C. A simple and widely used loss function formula is the quadratic loss function:

$$L = (y - \hat{y})^2$$

In this formula, y represents the neural network's output, and $\hat{y}$ is the value the neural network should have outputted.

y	$\hat{y}$	L
1.1	1	0.01
2	1	1
0	1	1
3	1	4

d) Gradient

The policy gradient methods were proposed as an alternative to the algorithms that try to approximate the value function. These algorithms create a policy using the value function estimator created, but these algorithms have many limitations. This is why the paper "Policy Gradient Methods for Reinforcement Learning with Function Approximation" (Sutton, McAllester, Singh, & Mansour, 1999) proposed the policy gradient methods.

The gradient of a function f is ∇f and it calculates the function's f steepest ascent direction. Calculating the gradient of a function is just calculating the respective partial derivatives. The symbol θ represents a function variable/parameter.

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial \theta_0} \\ \frac{\partial f}{\partial \theta_1} \\ \frac{\partial f}{\partial \theta_2} \\ (\dots) \\ \frac{\partial f}{\partial \theta_N} \end{bmatrix}$$

In a two-dimensional space (x,y), the function gradient $f(x) = x^2$ is just a derivative $\nabla f = 2x$ because this function only has one partial derivative. If the gradient is measured at $x = 2$ the gradient is $\nabla f = 4$. The information that is taken from this operation is that for the function $f(x)$ at the point $x = 2$ the direction with the steepest increase is increment x with a slope of four. In reinforcement learning the gradient is usually used to minimize the loss function, so instead of maximizing the gradient, the objective is to minimize it, this is why RL uses a negative gradient ($-\nabla$). So for the function $f(x) = x^2$ the gradient in $x = 2$ is $-\nabla f = -4$ indicating the direction of descent.

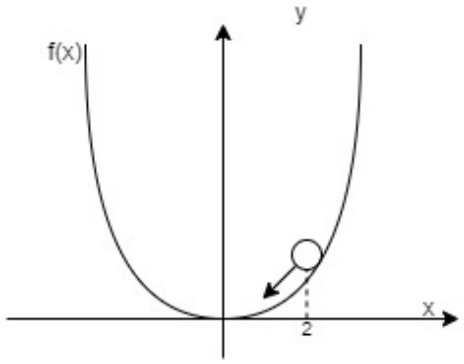


Figure 5.11: Quadratic Function Gradient

It is important to bear in mind that this is an overly simplified example of applying the gradient.

In a machine-learning context, the function to which the gradient is applied is more complex and has an unknown shape, therefore the direction of ascent is not as clear as a quadratic function in a two-dimensional space. What is done in machine learning is to take small steps in the gradient's direction until the local minimum is found, this is called gradient descent.

$$x' = x - \alpha \nabla f$$

In this formula, α represents the learning rate. The learning rate defines the step size in the gradient direction. Through the course of multiple iterations, this formula can find the local minimum of a function $f(x)$

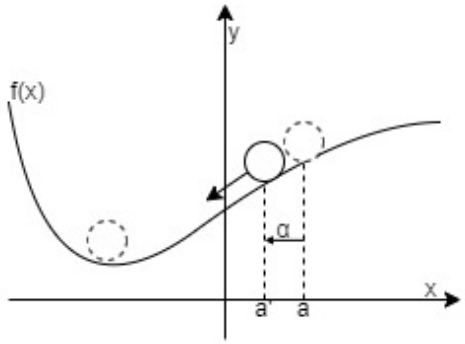


Figure 5.12: Gradient of f(x)

Defining the right learning rate is important because defining a learning rate that is too small leads to RL algorithms that take too long to converge to the local minimum and defining a learning rate too big leads to RL algorithms that can not converge to the local minimum.

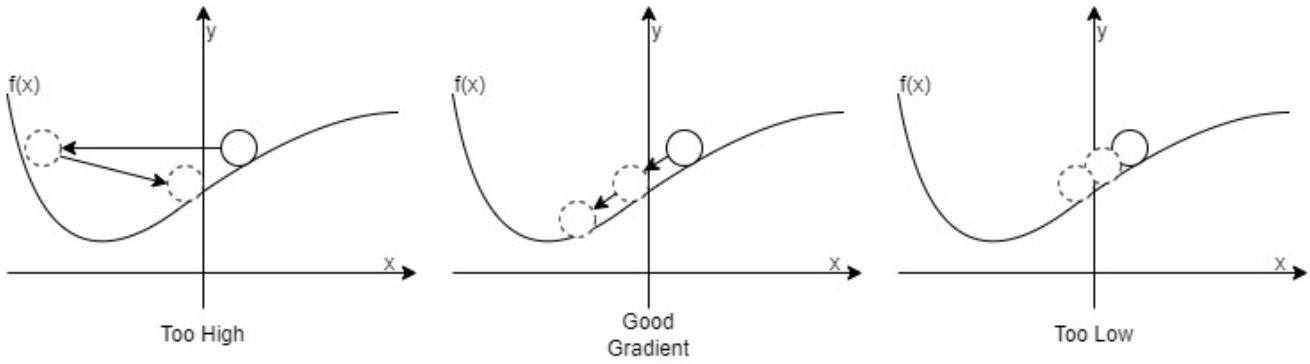


Figure 5.13: Learning rate tuning

The example shown works in a two-dimensional space but the gradient works with functions with N dimension, with N being any natural number. To better understand how this works in a space with more than two dimensions the following example shows how the gradient works in a three-dimensional space.

A quadratic function $f(x, y) = x^2 + y^2$ has the shape shown in the cross-referenced figure 5.14. If the variables $x = 1$, $y = 1$ the value of $f(1, 1) = 2$. The objective is to minimize the function f value so the first step is to calculate the gradient:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$$

The gradient descent algorithm is applied. In this example, the learning rate used is $\alpha = 0.1$

$$x' = \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 0.1 * \begin{bmatrix} 2 * 1 \\ 2 * 1 \end{bmatrix} = \begin{bmatrix} 0.8 \\ 0.8 \end{bmatrix}$$

So to minimize the function f value the variables should have the following values $x = 0.8$, $y = 0.8$ which is equal to the value $f(0.8, 0, 8) = 1.28$, this is a lower value than the one at the previous point $f(1, 1) = 2$. This operation can be visualized in the figure 5.14

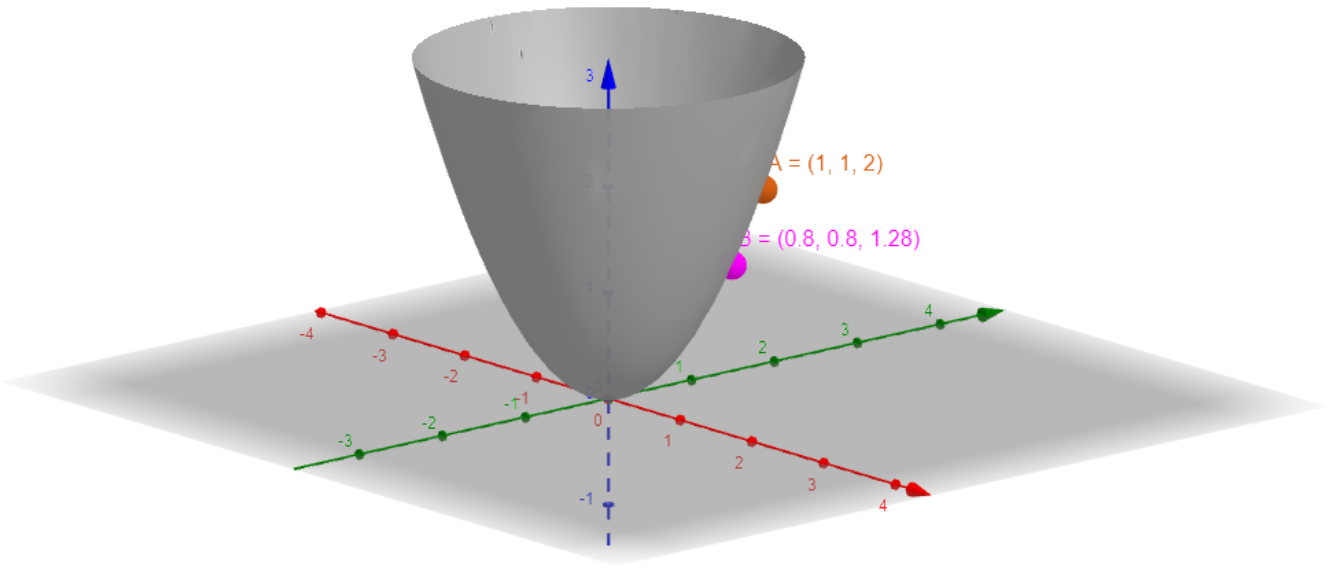


Figure 5.14: Quadratic 3D function gradient

So far this subsection has explained the mathematical process of applying gradient descent but its utility in RL may not be clear. An example follows to better understand how the gradient connects to the previously explained concepts.

If an agent implements it's policy using a neural network, tuning the policy consists of adjusting the neural network's parameters θ (weights, W, and biases B). Using the loss function the algorithm

evaluates how bad the policy is. So the objective is to tune the neural network's parameters θ to minimize the loss function L. What this subsection explains is that the gradient is capable of finding the minimum of a function by tuning some parameters. So the gradient is capable of adjusting the neural network's parameters θ to minimize the loss function.

$$\theta' = \theta - \alpha \nabla_\theta L$$

This formula represents the algorithm gradient descent applied to the loss function and the neural network's parameters. The gradient descent can be represented more clearly as follows:

$$\nabla_\theta L = \begin{bmatrix} \frac{\partial L}{\partial \theta_0} \\ \frac{\partial L}{\partial \theta_1} \\ \frac{\partial L}{\partial \theta_2} \\ (...) \\ \frac{\partial L}{\partial \theta_N} \end{bmatrix} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ (...) \\ \theta_N \end{bmatrix}$$

The symbol θ represents the weights (W) and biases (B) so this is the same as follows:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ (...) \\ b'_{N_b} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ (...) \\ b_{N_b} \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial L}{\partial b_0} \\ \frac{\partial L}{\partial b_1} \\ \frac{\partial L}{\partial b_2} \\ (...) \\ \frac{\partial L}{\partial b_{N_b}} \end{bmatrix}$$

$$\begin{bmatrix} w'_0 \\ w'_1 \\ w'_2 \\ (...) \\ w'_{N_w} \end{bmatrix} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ (...) \\ w_{N_w} \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \\ (...) \\ \frac{\partial L}{\partial w_{N_w}} \end{bmatrix}$$

e) Backpropagation

The gradient is given by a matrix of partial derivatives and a neural network consists of a set of neurons, it may seem strange how these two concepts connect. Applying a gradient to the neural network parameters is done through a process called backpropagation (Rumelhart, Hinton, & Williams, 1986). Calculating the gradient in a time-step is calculating all the partial derivatives in its matrix $\frac{\partial L}{\partial w}$.

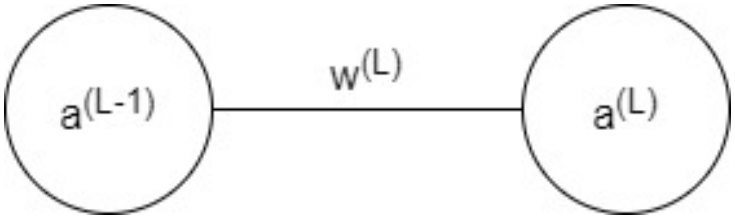


Figure 5.15: Two layers neural network

To understand how this derivative is applied to a neural network the cross-referenced figure 5.15 represents a neural network with only two neurons, the output $a^{(L)}$, and the input $a^{(L-1)}$, with a connection between them with a weight value of $w^{(L)}$. Supposing that there is a desired output of y now it is possible to create a loss function that compares the neural network's last layer with the desired output. In this example, the loss function will be a mean squared error $L = E[(a^{(L)} - y)^2]$.

The output neuron input is given by the previous layer neurons output, $a^{(L-1)}$, multiplied by the connection's weight, $w^{(L)}$, and summed the respective neuron's bias, $b^{(L)}$, so the neuron output, in this case, is given by: $z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$. The output layer value is given by: $a^{(L)} = \sigma(z^{(L)})$. Applying the chain rule it is possible to write the following:

$$\frac{\partial L_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial L_0}{\partial a^{(L)}} \tag{5.1}$$

Calculating each partial derivatives:

$$\frac{\partial L_0}{\partial a^{(L)}} = 2(a^{(L)} - y) \qquad \frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)}) \qquad \frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)} \tag{5.2}$$

With this data, it is now possible to calculate the $\frac{\partial L_0}{\partial a^{(L)}}$, which corresponds to the loss in iteration zero. The loss average is calculated across multiple iterations through the following formula:

$$\frac{\partial L}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial L_k}{\partial w^L}$$

This same process has to be executed for the biases of the neural network.

$$\frac{\partial L_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial L_0}{\partial a^{(L)}} \tag{5.3}$$

$$\frac{\partial z^{(L)}}{\partial b^{(L)}} = 1 \tag{5.4}$$

This process is easily escalated to any type of neural network. If the neural network has one more layer the chain rule is a little different but the logic is the same. The cross-referenced figure shows a neural network of three layers. Calculating the weight's gradient w^{L-1} at iteration zero is to calculate $\frac{\partial L_0}{\partial w^{L-1}}$

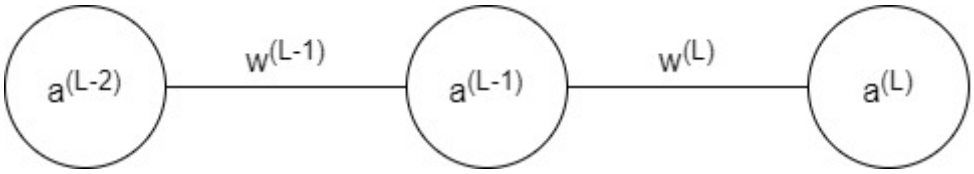


Figure 5.16: Three layers neural network

To easily find out the formula to update any neural network parameter using the chain rule it becomes more intuitive to use the scheme in the cross-referenced figure 5.17.

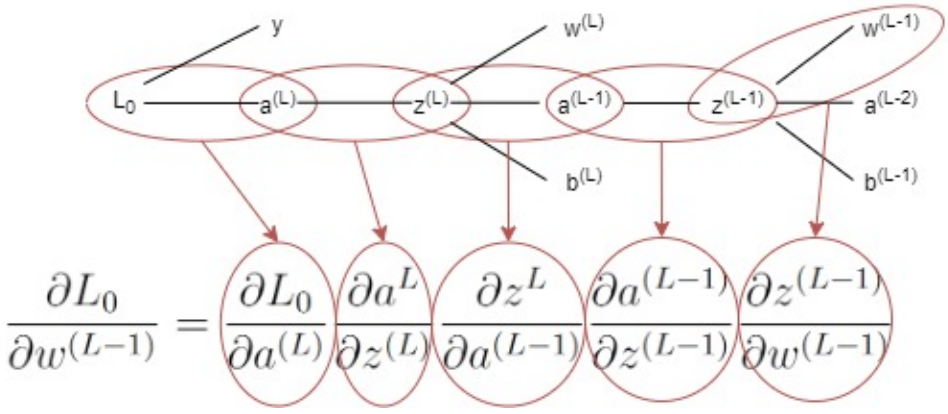


Figure 5.17: Chain rule from scheme to formula

After understanding how to calculate the gradient using backpropagation to update neural networks with N layers the next step is to understand how to update a neural network with N neurons in each layer. To illustrate the difference this example uses the neural network shown in cross-referenced figure 5.18. To identify the connection's weights, w, this example uses indexes that identify from which neuron to which neuron the connection corresponds.

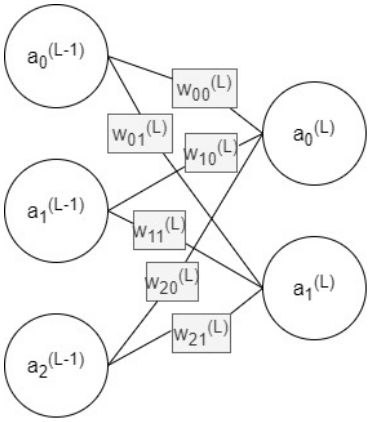


Figure 5.18: Neural network example

To calculate the gradient the only change to do is to z. The value of $z_0^{(L)}$, for example, is calculated using the following equation.

$$z_0^{(L)} = a_0^{(L-1)} w_{00}^L + a_1^{(L-1)} w_{10}^L + a_2^{(L-1)} w_{20}^L \tag{5.5}$$

f) Actor-Critic Algorithms

The limitation of some deep RL algorithms is that the actions have to be discreet, this means that there is a limited set of actions from which the agent can choose. The cart-pole environment has a discrete action space approach, the cart can only go right or left, and there is no variation in speed. A continuous action space approach would be choosing the cart's speed in either direction.

Actor-critic algorithms (Barto, Sutton, & Anderson, 1983) allow continuous action space using two neural networks, the critic and the actor. The actor is responsible for choosing an action. The critic is responsible for creating a model for the value function. The actor receives a state and outputs an action, and the critic receives a state and outputs the value for that state. The critic neural network is trained to get better value function predictions. The actor is trained to get a better policy.

g) Advantage function and GAE

The advantage function calculates the difference between the predicted return of a state and the actual return of a state when executing an action a at the state s.

$$A(s,a) = Q(s,a) - V(s) \tag{5.6}$$

Function $V(s)$ represents the expected state s return. Function $Q(s,a)$ represents the expected return of state s when executing action a. Essentially for a state s the advantage calculates the difference between the predicted expected return value ($V(s)$) and the value the expected return obtained by executing the action a ($Q(s,a)$). With this information the formula can be more clearly interpreted:

$$A(s,a) = E_\tau[G(\tau)|s_0 = s, a_0 = a] - \hat{V}(s) \tag{5.7}$$

The value represents a value function estimator which is why it gets a hat ($\hat{V}(s)$). In an actor-critic algorithm $\hat{V}(s)$ is obtained by inputting the state s to the critic neural network, and the output is the value of that state $\hat{V}(s)$.

The return of a state is the present reward summed with all the discounted future rewards. The discounted future consists of calculating the next state s_1 value so it can be calculated using the value estimator $\hat{V}(s_1)$. The expected return is calculated by summing the reward obtained in the state s to the predicted next state value $V(s_1)$.

$$E_\tau[G(\tau)|s_0 = s, a_0 = a] = r_0 + \gamma V(s_1) \tag{5.8}$$

$$\hat{A}(s,a) = r_0 + \gamma\hat{V}(s_1) - \hat{V}(s) \tag{5.9}$$

This formula induces bias to the advantage estimator because it is too dependent on the value estimator. A way of reducing this dependency on the estimator would be to extend the temporal difference by getting the following equation:

$$\hat{A}^{(n)}(s,a) = r_0 + \gamma r_1 + (...) + \gamma^{n-1} r_{n-1} + \gamma^n V(s_n) - V(s_0) \tag{5.10}$$

In this formula, if n is increased it gets less dependent on the estimator meaning less bias, but because this formula has more variables it has a higher variance if n is decreased it gets less variance but gets more dependent on the estimator meaning higher bias. So n is responsible for controlling the bias/variance trade-off.

The paper "High-Dimensional Continuous Control Using Generalized Advantage Estimation" in section 3 aims to produce "an accurate estimate of the discounted advantage estimation function" (Schulman, Moritz, Levine, Jordan, & Abbeel, 2015), to achieve this goal the paper proposes the general advantage

estimation (GAE). The GAE controls the bias/variance trade-off by calculating the exponential average for all n values. For a thorough comprehension of the GAE concepts, the mentioned paper should be analyzed. This dissertation will not develop the final formula deduction and why this formula controls the bias/variance trade-off. This dissertation understands that the following formulas, proposed in the mentioned paper, calculate a good advantage function estimation.

$$\hat{A}_t^{GAE(\gamma,\lambda)}(s,a) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{5.11}$$

$$\delta_t = r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t) \tag{5.12}$$

5.2.5 Proximal Policy Optimization

Proximal policy optimizations (PPO) is an actor-critic, model-free algorithm developed by Open AI. The algorithm PPO is proposed in the paper "Proximal Policy Optimization Algorithms" (Schulman et al., 2017). PPO improves some features proposed in the popular algorithms: policy gradient (PG) and trust region policy optimization (TRPO) (Schulman, Levine, Moritz, Jordan, & Abbeel, 2015). To understand PPO's innovation first it is essential to understand the other algorithms and what it's problems are.

Policy Gradient methods are based on the following loss function:

$$L^{PG}(\theta) = E_t[\log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{5.13}$$

The term $\pi_\theta(a_t|s_t)$ represents the probability of the agent taking each action and the term $\hat{A}_t$ represents the advantage estimation. The advantage is a higher positive number if the action taken is better than predicted and a higher negative number if the action taken is worse than predicted. More intuitively, this function rewards actions that end up being better than expected and punishes actions that end up being worse than expected, the higher the action probability the more it gets punished or rewarded. This concept is used in both TRPO and PPO.

The problem with the policy gradient approach is that the gradient updates are carried out without some kind of limit on the updates optimism. This is a problem because the advantage ($\hat{A}_t$) may have high values in situations not well justified and having high loss function values means making long jumps on the gradient updates, which associated with faulty data may lead to destructive policy updates. This optimism limit is what TRPO aims to improve.

$$\max_{\theta} E_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}\hat{A}_t - \beta KL[\pi_{\theta old}(.|s_t), \pi_\theta(.|s_t)]] \tag{5.14}$$

In this formula, the term $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)\hat{A}_t}$ is the probability of taking an action with the current policy parameters divided by the probability of taking that same action but with the old policy parameters, multiplied by the advantage. This term is similar to the policy gradient loss but instead of using log $\pi_\theta(a_t|s_t)$ it uses $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$. The second term implements the KL constraint which solves the destructive policy updates however the KL constraint doesn't work that well and therefore "additional modifications are required" (Schulman et al., 2017).

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \tag{5.15}$$

$$L^{CLIP}(\theta) = E_t[\text{min}(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{5.16}$$

$$L_t^{CLIP+S} = E_t[L_t^{CLIP}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{5.17}$$

The ratio $r_t(\theta)$ is used so the equation is easier to understand. L^{CLIP} chooses the lowest term, $r_t(\theta)\hat{A}_t$ or $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$. The second term clips the ratio, $r_t(\theta)$. Clipping means setting a minimum, $1-\epsilon$ and a maximum $1+\epsilon$. If $r_t(\theta)$ is smaller than $1-\epsilon$ the term clip returns $1-\epsilon$, if $r_t(\theta)$ is bigger than $1+\epsilon$ it returns $1+\epsilon$, for $r_t(\theta)$ between the maximum and the minimum it returns $r_t(\theta)$. ϵ is a constant with a value defined by the user and has a value between 0.1 and 0.3. The PPO paper also suggests using an entropy bonus "to ensure sufficient exploration". In the equation L_t^{CLIP+S}, $c2$ is a constant defined by the user that multiplies to the calculated entropy S.

The graphic 5.19 shows the clipping of L^{CLIP}. This shows the pessimist approach of PPO. It is pessimist because if the advantage is negative there is no limit. The advantage is negative when the action value ends up being worse than predicted by the value estimator, on the other hand, if the advantage is positive it means that the action value ended up being better than predicted by the value estimator. If the advantage is positive the step in the gradient is limited on the other hand if the advantage is negative the gradient update is not limited this is why PPO is considered to be a pessimist algorithm (Schulman et al., 2017).

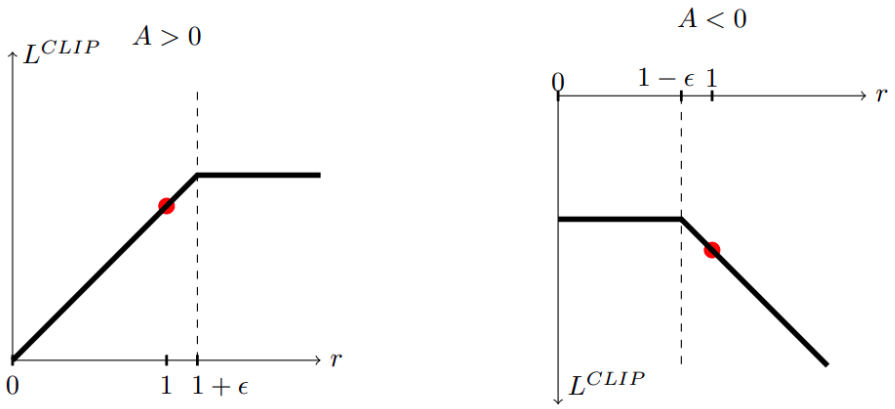


Figure 5.19: PPO loss function graphic

a) Critic loss function

In an actor-critic algorithm, there are two loss functions, one for the actor and one for the critic. The critic's goal is to estimate the state's value in the environment so the loss function has to indicate how good or bad the critic estimation is. A commonly used loss function is the mean squared error.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5.18}$$

The critic loss function measures the predicted value of a state vs the real state value, so the error is calculated using the following formula.

$$e = (V(s) - \hat{V}(s))^2 \tag{5.19}$$

The variable $\hat{V}(s)$ represents the value estimation done by the critic for the state s and $V(s)$ represents the real state s value. This formula calculates the critic's error when estimating the value of a state s. The real value $V(s)$ is given by summing the advantage A and the predicted value $\hat{V}(s)$.

$$V(s) = A(s,a) + \hat{V}(s) \tag{5.20}$$

5.2.6 Simulation Software

There are many robotics simulation softwares. In this dissertation three softwares were examined, Mujoco, CoppeliaSim, and Gazebo, all of them are open-source and each has its main advantages and

disadvantages. A more thorough analysis of each software's pros and cons can be done but in this dissertation, only the deciding factors that led to the software's choice will be described.

Mujoco is a software developed by DeepMind that out of the three options has the most accurate physics simulation engine. The downside is that it used to be proprietary software and just recently it became open source (Kocher, 2022), for this reason, there is not much online support and this makes the development of models and environments much harder. Mujoco was tested for two weeks and after some time, it was verified that developing a simulation environment would take too much time. Another deal breaker of Mujoco is its resource cost. Simulations with Mujoco are CPU heavy and AI algorithms already require a lot of computing power especially with computer vision.

CoppeliaSim is an open-source software created specifically to allow users "fast prototyping and verification" (CoppeliaSim, 2022). This software offers a GUI for intuitive and easy environment development. The user can develop an environment and a robot model using the GUI toolbar to drop the primitive shape into the environment and then create joints between the primitive shapes. The primitive shapes and joints can be configured to fit the user's needs. The joints can then be controlled through external scripts. The GUI interface allows development faster which is very appealing. Coppelia comes with a huge downside when compared with Gazebo which is the dependency on the GUI.

Gazebo is a popular open-source robotics simulator created by Open Robotics, this is positive because this means having useful resources available online such as tutorials that help understand how to use the software, having other users that have solved problems similar to the ones faced in this dissertation, and having modules developed by other users that can be integrated into the simulation environment. Another huge feature of Gazebo is the fact that a simulation can be executed with GUI off. This allows lighter execution which is very important because AI algorithms also require a lot of computer power and having tools that are too heavy will affect the algorithm performance and the simulation software. Gazebo has a great connection with robotic operating system (ROS) which allows users to develop ROS packages that describe the environment and then control the environment through ROS topics.

With all the data gathered from each simulation software, it was decided that Gazebo had the best feature trade-off in the context of this dissertation. The main reasons are that it allowed a lighter environment by turning off the GUI, its connection with ROS, and also the fact that both Gazebo and ROS are two popular softwares that have a lot of information online.

5.2.7 ROS

ROS stands for robotic operating system (Tellez, 2022), this open-source framework is popular and the

community has been active in the development of packages since 2010. The most basic ROS mechanisms are packages, nodes, topics and messages. The packages are executable programs that can create nodes, the nodes can be subscribed or publish to a topic. When a node is subscribed it means that it reads messages sent to the topic to which it is subscribed, if the node is a publisher it means that it sends messages to the topic to which it is a publisher. These mechanisms allow packages to share data, allowing modular development of a robotic system. The modular robotic system development is especially useful for the integration of packages developed by other users. For example if in a project a driver for a sensor is needed and this driver is already developed and shared the user can just download the package and use the generated nodes/services to retrieve data from the sensor.

In the context of ROS/Gazebo communication, when launching a gazebo package some topics are created to get messages from the simulation environment, these topics will send messages that store data such as the position of objects in the environment. To control joints in the simulation the package creates nodes that send messages to the target topic and this will make the joints behave according to the message.

6 Simulation Environment Development

To develop the simulation environment the tools selected were Gazebo as the simulation software and ROS as the middleware tool. The description of the autonomous vehicle, environment, control system and the intel real sense camera were carried out using ROS packages. Each ROS package has a launch file that describes what ROS does when launching the respective packages. The description is carried out using a set of ROS commands expressed in a markup language format. This chapter explains how each package was developed.

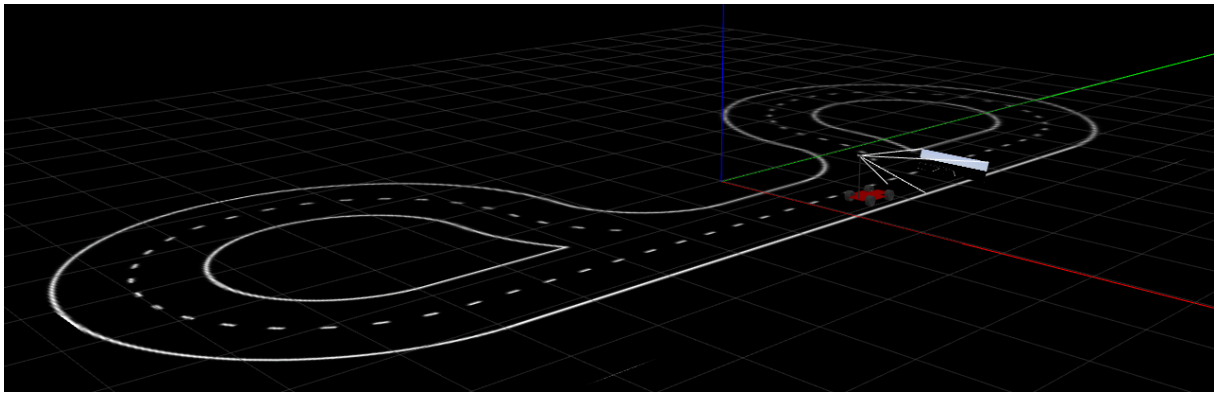


Figure 6.20: Simulation environment

6.1 Description of the vehicle

The package that describes the autonomous vehicle is named *autonomous_vehicle_description* and has four folders: launch, meshes, rviz and urdf.

- meshes - Stores all the STL files needed for the model's visual component
- launch - Stores the launch file
- urdf - Stores URDF (Unified Robotics Description Format) file with autonomous vehicle model description
- rviz - Stores automatically generated files, that convert the URDF model description into a RVIZ specific description

The autonomous vehicle description files use XML description. RVIZ (ROS visualization) is a software used to visualize the model that is described in the URDF file.

6.1.1 Launch File and Package Dependencies

The launch file function inside the *autonoumous_vehicle_description* package is to run the vehicle description in RVIZ. Launching the description file in RVIZ is useful because it allows the user to detect any syntax errors in the description, checks if the description is the intended, and finally tests any movable joints. To test the file it is more efficient to launch the model in RVIZ instead of Gazebo because it is a lighter software and is faster to launch.

This file starts by declaring a variable that stores the path to the model description. The variable is used to declare the path to the model description in the parameter server. Three nodes have to be initialized: *joint_state_publisher*, *robot_state_publisher*, and *rviz* node. The *rviz* node requires an argument that sets the path to which RVIZ stores the configurations. The launch file created in this project can be easily adapted to launch any URDF file in RVIZ by just changing the *robot_description* path.

```
<launch>

  <arg name="model" default="$(find autonomous_vehicle_description)/urdf/Ex1.xacro"/>
  <param name="robot_description" command="$(find xacro)/xacro --inorder $(arg model)" />
  <arg name="rvizconfig" default="$(find autonomous_vehicle_description)/rviz/urdf.rviz" />

  <node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" required="true" />

</launch>
```

Figure 6.21: Launch file description package

6.1.2 XML and URDF files

A Markup Language consists of a language that uses tags to describe elements in a document. Some Markup languages have a specific set of tags, for example, HTML offers a set of tags designed for a more efficient web design workflow. The model description in this dissertation uses URDF which is a specification of XML (Extensible Markup Language). XML is a markup language that unlike HTML does not have a predefined set of tags, the tags are specifically designed for a purpose. There is a specific set of XML tags designed to describe multibody systems which are URDF. URDF is an XML specification mainly used to describe a robotic model, normally to be used in a simulation.

6.1.3 URDF

XML language uses tags, and the XML specification used to describe the robot model is URDF. In

URDF the two main tags used to describe a robot are links and joints, where links describe objects and joints glue the objects together. When creating a joint between two links a father and child link have to be defined creating a hierarchy. The rest of this subsection explains thoroughly how the basic description and implementation of an URDF model works. One important thing to notice is that all the values that define sizes are expressed in meters and all the values that express angles are in radians.

A link tag describes a body, the body has three main attributes described through tags: visual, inertial, and collision. The link's visual and collision attributes shape are described using the geometry tag inside of which primitive shapes or meshes are used to specify the respective geometry. The tags used to define a geometry through primitive shapes are box, cylinder, or sphere. For each primitive shape, different parameters have to be defined. The box has a size parameter that consists of three values to define the three vertices size. The cylinder has a radius and length parameter in which the first refers to the radius and the second to the cylinder's length. The sphere has only one parameter which is the radius, this defines the sphere's radius. For a more detailed geometry description, there is a tag that allows the user to include mesh files. The mesh files are normally generated in 3D modeling software that allow the user to create complex shapes and then export them to a file that describes the shape created. There are multiple extensions used for mesh files but in this project, all the mesh files used are stored in STL files. The desired mesh file can be used to describe a geometry using the mesh tag. The most important parameter in this tag is the filename. The filename parameter is a string with the path to the desired mesh to be used on the respective link property geometry. Using meshes gives a lot more detail to the robotic model, but when applied to the link's collision properties it comes at a heavy computational cost. To avoid excessive CPU usage it is a great idea to simplify as much as possible the collision properties by using primitive shapes to describe the robot model instead of using meshes because meshes are much more detailed than primitive shapes and therefore require more CPU power than primitive shapes, slowing down the process.

To define the visual and collision attributes position the tag used is origin. Inside the origin tag, there can be set parameters such as XYZ which allows the user to set the link position in space according to the cartesian coordinates, and RPY which allows the user to set the pitch roll and yaw. Roll pitch and yaw is a way of setting the orientation in space, where roll sets the angle by rotating around X, pitch sets the angle by rotating around Y and yaw sets the angle by rotating around Z. It is important to notice that all the values set the respective link property origin relative to the center of the model which has coordinates x=0 y=0 z=0 and r=0 p=0 y=0.

The two most important tags to define in the link's inertial component are mass and inertia. The value

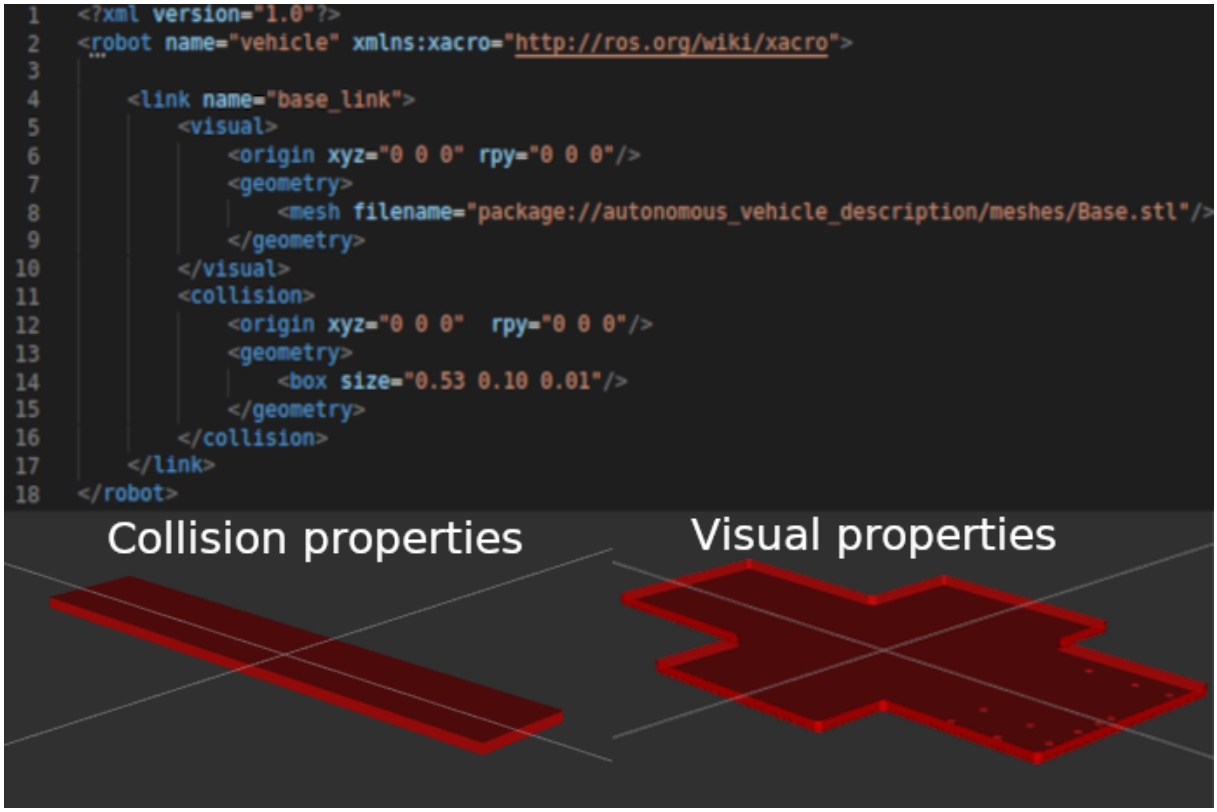


Figure 6.22: Link with visual and collision properties

parameter inside the mass tag sets the link's mass in kilograms. The inertial tag receives the value to set the respective link inertia matrix. The inertial matrix values are Ixx, Ixy, Ixz, Iyy, Iyz, and Izz. The values to be set are easily obtained by using a software such as Meshlab, where any mesh file can be added to then be processed to get the results expected by the user. By using "Compute Geometric Measures" functionality this software gives the user the mesh's inertial matrix values. If the values are too low, the simulation might crash because in an iteration if some values get to zero the math may not be physically possible. To avoid the type of problems described before a factor may be added to get higher values, "Transform: Scale" allows the user to choose a scaling factor and apply it.

Now that the link and its respective components can be defined using the tags described before, the next step is to define the model joints using the joint tag, which specifies how the links relate. To create a joint between two links the user has to set a parent link, a child link, and a type of joint. There are different types of joints of which revolute, continuous and fixed were the ones used to describe the autonomous vehicle model. A fixed joint fixes the two links. A continuous joint allows rotational movement between two links, and as the name suggests is continuous and along a specific axis without upper or lower limits of rotation. This type of joint is useful to describe movements such as the wheel of a car. A revolute joint allows rotational movements between two links, along a specific axis and unlike the continuous joint this

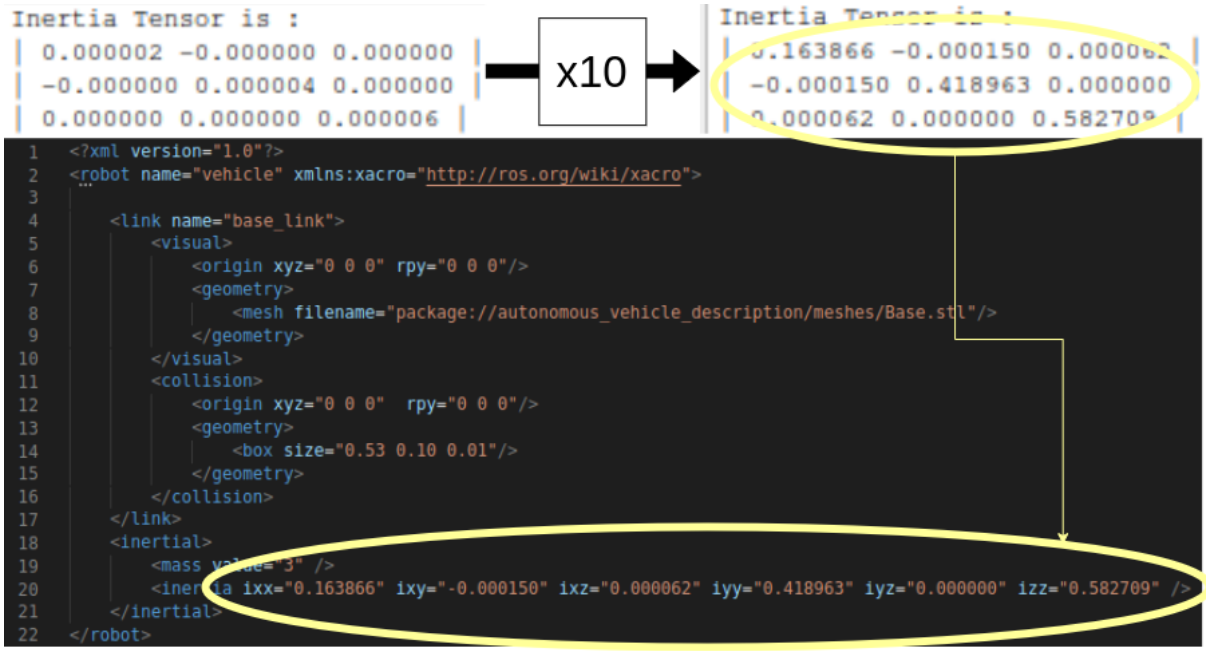


Figure 6.23: Setting the inertial properties with a 10x scale factor

has an upper and lower limit of rotation. This type of joint is useful to describe for example the steering wheel of a car because it rotates the wheel of a car according to a specific axis but with upper and lower limits of rotation. Origin property is used to set a child link position offset relative to the parent link origin. Stating the origin inside joint differs from stating the origin inside the link because it sets the whole link origin and not just a single property, and it also differs because it is relative to the parent link origin and not relative to the model origin. Inside origin there are two useful parameters that can be set which are XYZ which receives the cartesian coordinates to offset the child's position in space and RPY which offsets the child link angle. There is another useful property to be set which is limit. With this property, the user can set some limits to the joint, like effort lower limit, upper limit, and velocity depending on the type of joint used.

6.1.4 XACRO files

The model is described through URDF files. The URDF description is quite limited and for this reason, ROS offers the XACRO extension. The XACRO extension is a very powerful resource that allows users to do things such as link multiple files, create macros, set variables, and create mathematical equations.

The mentioned functionalities are very useful when developing the model description because it allows the user the possibility of creating macros to describe a shape that repeats itself in space and the macro can be used to describe the shape instead of having to rewrite the description again. The mathe-

```xml
<?xml version="1.0"?>
<robot name="vehicle" xmlns:xacro="http://ros.org/wiki/xacro">
  ""
    <link name="base_link">
        <visual>
            <origin xyz="0 0 0" rpy="0 0 0"/>
            <geometry>
                <mesh filename="package://autonomous_vehicle_description/meshes/Base.stl"/>
            </geometry>
        </visual>
    </link>

    <link name="Wheel">
        <visual>
            <!--origin xyz="0 0 ${VehicleHeight}"/-->
            <geometry>
                <mesh filename="package://autonomous_vehicle_description/meshes/LeftWheel.stl"/>
            </geometry>
        </visual>
    </link>

    <joint name="base_wheel_joint" type="continuous">
        <axis xyz="0 1 0"/>
        <parent link="base_link"/>
        <child link="Wheel"/>
        <origin xyz="0.185 0.15 0"/>
    </joint>

</robot>
```

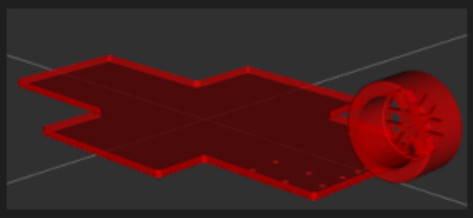


Figure 6.24: Joint between base and wheel

matical equation functionality and the variables allow the user to automate some processes in the model description which is very useful too, especially when used with the macro functionality.

The mentioned XACRO functionalities allow users to easily integrate other models already created by just including the file that describes it and declaring the macro in the XACRO file. This is carried out by just replacing the URDF file extension with XACRO. The user then can use the XACRO macro created to include the model.

6.1.5 Model development

When developing the model the first step consists of creating the root link. The root link cannot have any collision or inertial properties, as it can only have visual properties. In this description the root link does not have any properties, it is named *vehicle* and connects to the *base_link* through a joint. The *base_link* describes the autonomous vehicle base, the visual properties include the mesh file used to model the autonomous vehicle, meaning that it is a detailed representation of what the vehicle looks like in the real world. The collision properties geometry was described using a primitive shape in this case a box that had roughly the same size as the vehicle's base. The box that describes the vehicle's base collision properties is a bit smaller than the actual vehicle base, to avoid collision with wheels when turning. The *base_link*

visual and collision properties are described in the figure 6.22. To describe the inertia matrix inside the inertial properties an external software, Meshlab was used to calculate the values. The first values gathered were very close to zero and to avoid crashes in the simulation, as explained before, a factor of 10 was applied and new higher values were gathered and used in the model. Figure 6.23 shows the inertia matrix values gathered in Meshlab before and after applying the scaling factor. Four links were created to describe each vehicle wheel. To describe the wheels two XACRO macros were created to avoid repetition of the same tag structure, as the description of each wheel is equal. The visual properties are defined using a mesh file that is the real-world wheels representation. The collision properties are described using a cylinder primitive shape of the same size as the wheels. The cylinder rotates 90 degrees to have the same position as the wheels. The inertial matrix values inside the inertial properties were gathered using an external software, Meshlab, and once again a factor of 10 was applied to match the base factor and to get higher values. The back wheels are connected to the motors and the front wheels are connected to the axle. The motors representation connected to the wheels is carried out by creating a continuous type of joint that connects the base to each wheel. The front wheels do not have a motor but are connected to the axle and so the axle needs to be described. The vehicle axle turns in two different spots, one for each wheel, and for this reason, two links with box shape were created and attached to the *base_link* using fixed joints. The two mentioned links are meant to describe the axle turning points and for this reason, the links are distanced from the *base_link* accordingly. A joint was created between the axle points and wheels, this joint is of type revolute with a limit of 35 degrees for each side allowing the front wheels to turn like the axle does. Figure 6.25 shows the vehicle visual and collision properties and helps understand how the wheels turn around the axle.

6.2 Gazebo integration

The *autonomous_vehicle_description* package stores all the model description files but lacks the launch file that allows the user to launch the model inside Gazebo and a simulation environment description. The package described in this section is *autonomous_vehicle_gazebo* and stores the following folders:

- launch - stores the .launch file
- world - store .world file

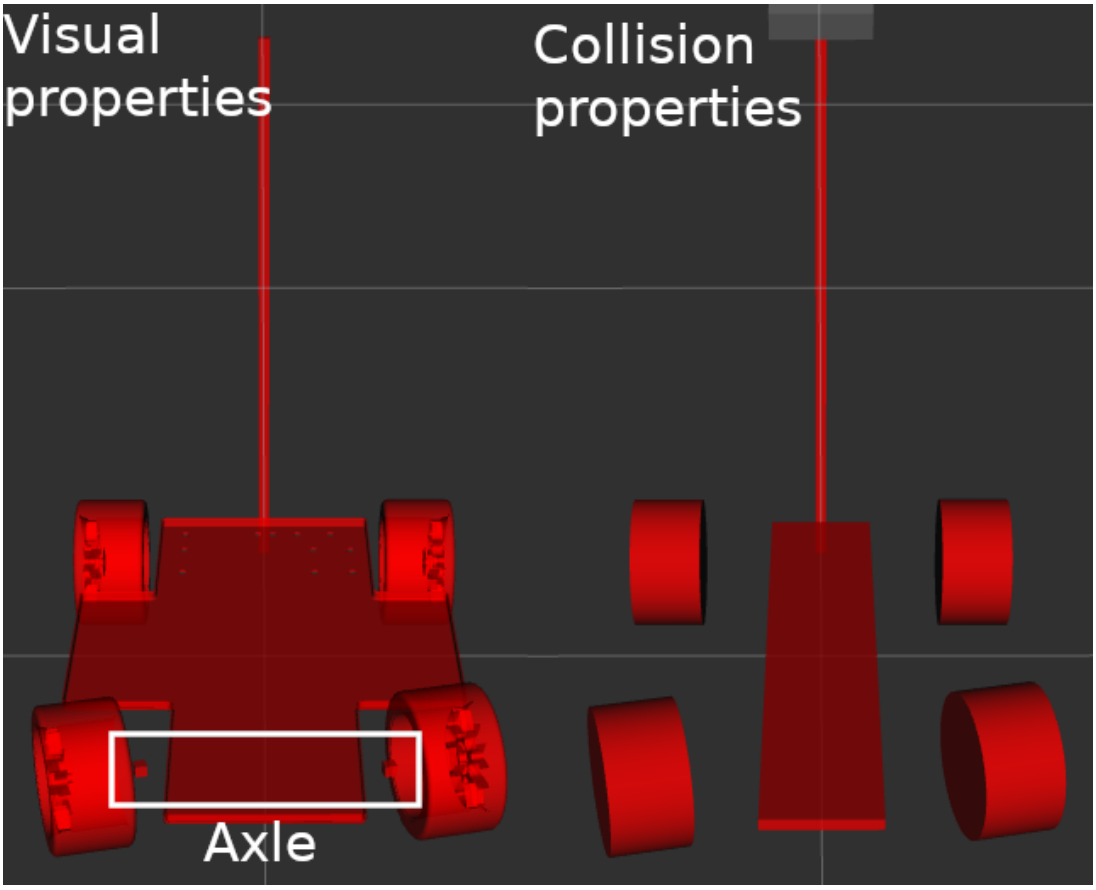


Figure 6.25: Vehicle collision and visual properties, with wheels turning left

6.2.1 Launch File and Package Dependencies

The launch file is the most important part of this package since its sole purpose is to include all the needed files to launch the model inside gazebo. The file starts by setting a few arguments that specify how the simulation should be launched. The model parameter stores the path to the URDF file with the model description and is used to open the model description in Gazebo using the URDF spawner which is a functionality from the package *gazebo_ros*, therefore this package is a dependency of *autonomous_vehicle_description*. With a standard set of tags, it is also possible to define many parameters about the simulation, defined in the arguments set at the beginning of the file, which can also be changed from the command line when launching the package. An important argument to set is the path to the .world file that describes the simulation environment. Another important set of arguments that *urdf_spawner* receives are the starting model coordinates, X, Y, Z. It is also possible to change other parameters using the arguments set at the beginning of the file like GUI which sets whether the simulation should launch the visual interface. Having the visual interface running can use a lot of computational resources, so this functionality is very useful to train a Machine Learning model because many iterations are needed for the

algorithm to learn, and the mathematical operations required to train the model are complex and require computational power. For this reason, it is of concern to developers to save as much CPU as possible on operations that are not important such as the GUI. The launch file ends by including the launch file from *autonomous_vehicle_control* package, which is explained in the next section.

```
<launch>
  "
    <!-- these are the arguments you can pass this launch file, for example paused:=true -->
    <arg name="paused" default="false"/>
    <arg name="use_sim_time" default="true"/>
    <arg name="gui" default="False"/>
    <arg name="headless" default="false"/>
    <arg name="debug" default="false"/>

    <arg name="model" default="$(find autonomous_vehicle_description)/urdf/vehicle_modelV3.xacro"/>

    <!-- resume the logic in empty_world.launch, changing only the name of the world to be launched -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
      <arg name="world_name" value="$(find autonomous_vehicle_gazebo)/worlds/autonomous_vehicle.world"/>
      <arg name="debug" value="$(arg debug)" />
      <arg name="gui" value="$(arg gui)" />
      <arg name="paused" value="$(arg paused)"/>
      <arg name="use_sim_time" value="$(arg use_sim_time)"/>
      <arg name="headless" value="$(arg headless)"/>
    </include>

    <param name="robot_description" command="$(find xacro)/xacro $(arg model)" />


    <arg name="x" default="1.8"/>
    <arg name="y" default="0.65"/>
    <arg name="z" default="0.1"/>
    <arg name="yaw" default="-1.57"/>


  <!-- push robot_description and spawn robot in gazebo -->
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
        args="-urdf -model autonomous_vehicle -param robot_description
        -x $(arg x) -y $(arg y) -z $(arg z) -Y $(arg yaw)"
        respawn="false" output="screen" />


  <include file="$(find autonomous_vehicle_control)/launch/autonomous_vehicle_control.launch" />


</launch>
```

Figure 6.26: Launch file gazebo package

6.2.2 Environment Description

To describe the simulation environment there is a .world file that describes it using XML tags. It is a very simple environment having only the sun and the track plane. The track plane is an object created and added by the user to the Gazebo models folder. The Gazebo models folder is located in .gazebo>models, the *.gazebo* folder is normally stored in the home directory. To create this object a folder was created with the name *track_plane* inside which there is a file named *model.sdf* to describe the track. The object description is similar to the predefined simulation plane, the difference is that the material is now a picture. The picture is the scaled track representation processed to show the road lines in white with black background. To set the material there is a folder with two other folders named, *scripts* and *textures*. The *textures* folder stores the picture that sets the plane material. Folder scripts store a file that describes the material.

6.3 Joint Control

This section explains how the "bridge" between the simulation and external scripts are related, and this is needed to allow the movable joints control from external scripts written in python. Most of the mechanisms implemented to control the joints are stored in *autonomous_vehicle_control*, but the packages referred to before (*autonomous_vehicle_description* and *autonomous_vehicle_gazebo*) lacked some information needed for the communication, so some changes were needed. The *autonomous_vehicle_control* includes the following folders:

- config - Stores YAML file that sets the joint control type
- launch - Stores the .launch file
- scripts - Stores the python scripts to control the simulation

6.3.1 Launch File and Package Dependencies

The launch file (6.27) first loads the YAML file by indicating it's path, this file describes the joints to control. The launch file than creates a node using *controller_manager*, the node created receives messages that control the simulation joints.

The first tag in the YAML file (6.30) should be the model namespace. For each joint to be controlled the user declares a tag with the controller name for each joint, which have to be declared inside the

namespace tag. For each controller at least two parameters have to be declared: the joint which declares the joint to be controlled, and the type that defines the hardware interface type. For each type of hardware controller, other parameters may be declared like the PID controller gains.

There is a tag that always needs to be declared which is the state controller in which the user defines the publish rate. The *ros_control* package defines the hardware interface. The controller is then loaded using *controller_spawner* which is a functionality from the *controller_manager* package, in this step it is important to change the arguments (*args*) to the same as the controllers names defined in the YAML file and *ns* to the same names as the namespace set when defining the plugin inside the URDF file (6.28) model description. The final line makes it possible for the messages to be visualized inside RVIZ, using the package *robot_state_publisher* making it a dependency of this package.

As mentioned before *controller_manager* was used in the launch file 6.27) making it a dependency for this package. The package *ros_control* is also a dependency because it defines the hardware interface for each controller.

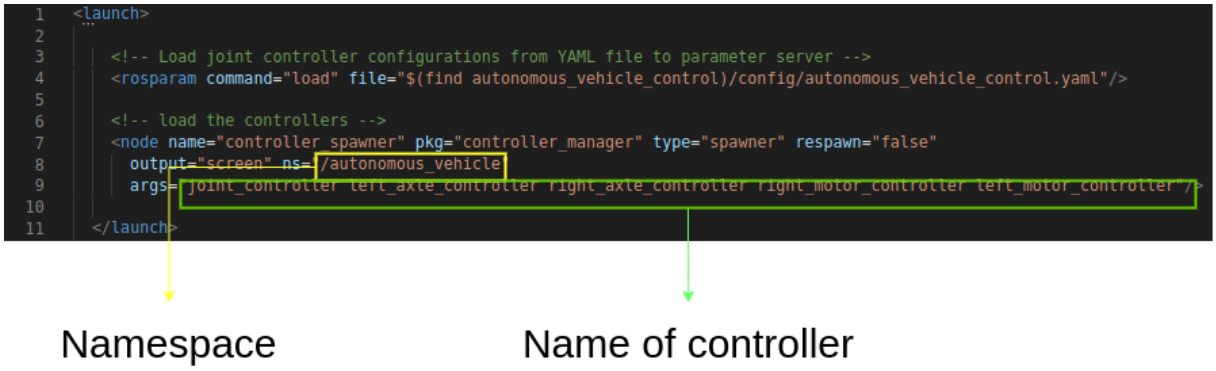


Figure 6.27: Launch file control package

6.3.2 Transmissions Plug-ins yaml

After creating a URDF file to describe the model with links and joints the user has to create a “bridge” between the model joints and ROS. To create this “bridge” firstly some changes have to be applied to the URDF model description. The first step is to add the *gazebo_ros_control* plugin that allows the implementation of custom interfaces between Gazebo and *ros_control*. Adding the mentioned plugin is easily carried out by pasting a standard set of tags to the end of the URDF model description file. The user has to change the robot namespace (*robotNamespace*, *autonomous_vehicle*) in the mentioned set of tags, and this should be the name to use in the YAML file and the launch file when using *controller_spawner*.

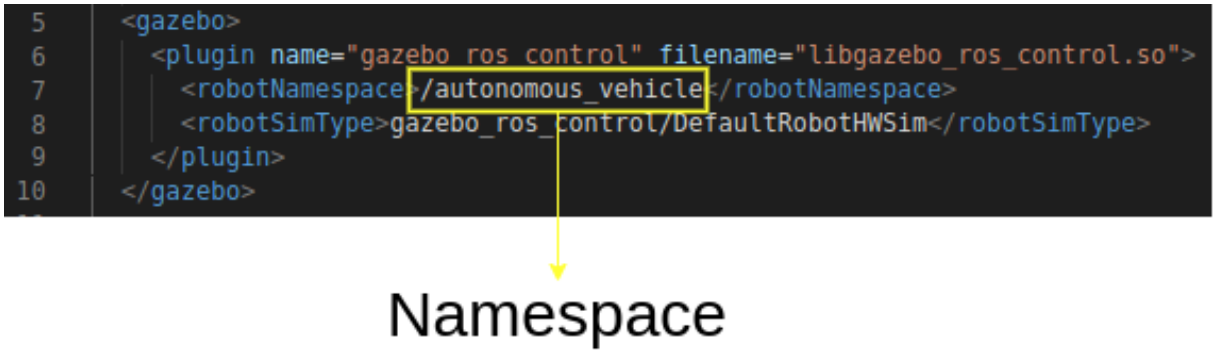


Figure 6.28: ROS control plugin in URDF description file

To describe the relationship between the actuator and the joint the user has to set transmissions, which are defined by inserting a set of tags with the link to be controlled and the hardware interface selected. The best way to set transmission is to get the standard set of tags and change some parameters. The set of tags defines parameters for both the actuator and the joint, where a name has to be defined for the transmission, the actuator, and the joint. The joint name has to correspond to the joint name to be controlled. For both the actuator and the joint a hardware interface has to be defined which has to be the same defined in the YAML file. Finally, the user can define a mechanical reduction inside the actuator tag. It is important to notice that the hardware interface is carried out using *ros_control* and offers a few different types of interfaces.

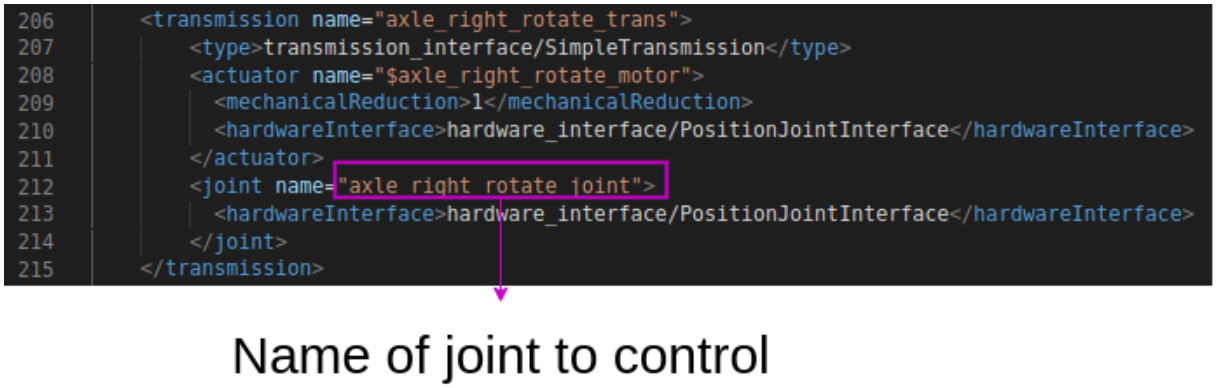


Figure 6.29: Define transmission in URDF description file

To finish the control "bridge" between simulation and control scripts the user has to create a YAML file in which all the joints to be controlled are referenced. In this file, the model name should be the same as the namespace (*robotNamespace*) mentioned in the URDF model and in the namespace (*ns*) mentioned in the launch file when loading the controller spawner. To define each joint control the user has to define a name, which has to be the same name used in the launch file when loading *controller_spawner* in args.

For each joint, a type of controller has to be specified and it has to match the hardware interface type defined in the transmission in the URDF description file. The type of controller used is *JointPositionController* and *JointVelocityController*, and the respective hardware interface types are *position_controllers* and *effort_controllers*.

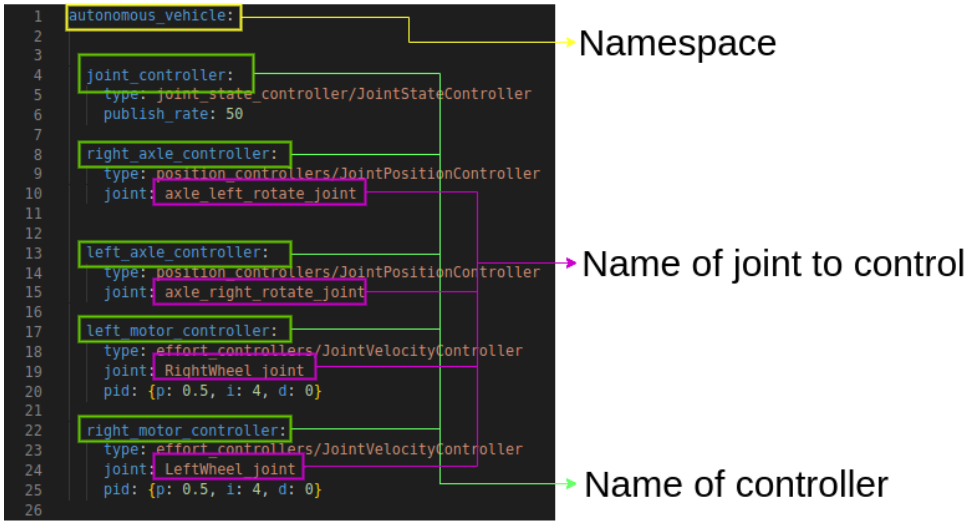


Figure 6.30: YAML file

6.3.3 ROS nodes and rospy

After setting all the referenced mechanisms the package is ready to be launched. After launching the control package a ROS topic is created for each joint actuator defined. When posting a message in a ROS topic the message is interpreted and the respective joint is updated according to the data given by the message. In this project, two types of hardware interfaces were used: *position_controllers* and *velocity_controllers*, and these work differently when posting a message to its topic. When posting a number to a *position_controller* joint it rotates until it gets an angle equal to the value posted in its topic. When posting a number to a *velocity_controller* joint it rotates with a speed equal to the value posted on its topic.

With the controller package running and it's topic working the user can use the rospy library to post any message to a ROS topic, allowing the user to control the simulation using python scripts.

6.4 Camera integration

This section explains how the intel realsense camera model was added to the autonomous vehicle

model and how plugins and ROS packages allowed retrieving data in simulation similar to the camera in the real world.

To achieve the mentioned goal some changes are required in the URDF model description. The two packages added that allow the intel realsense camera integration are *realsense2_description* which describes the camera model and *realsense_gazebo_plugin* which allows the user to get data from this sensor.

6.4.1 Intel realsense packages

Creating a intel realsense camera model and developing the respective plugin to make it work with Gazebo and ROS are two processes that are out of the scope of this dissertation and for this reason to achieve the goal of integrating the camera in the model, two packages were included. The process of adding the camera to the model shows the most powerful feature of ROS which is being able to easily add and connect different modules/packages to a project. With the mentioned functionality users can make use of modules already developed by other users allowing developers to contribute to each other's projects.

The camera model is stored in the *realsense2_description* package. This package stores various URDF descriptions of different camera models, launch files for the visualization of each camera model, and other essential files needed for the model description. The camera description uses meshes for the visual properties but with primitive shapes for the collision properties which is good to save computational resources. The Gazebo plugin files that allows retrieving data from the simulation like a real-world intel realsense camera are stored in the *realsense_gazebo_plugin* and for simplicity reasons, these files are not explained since its development is out of the scope of this dissertation.

6.4.2 Adding Camera Packages to model

To integrate the camera into the model the user must include the *realsense_gazebo_plugin* and *realsense2_description* packages in the catkin workspace. With the packages included the plugin and the camera model are ready to use. To use the sensor in a model description the user first has to use the XACRO extension and then include the file with its URDF/XACRO description using the XACRO tag include. The user sets the path to the URDF/XACRO sensor description and the XACRO macro that includes the sensor description is now ready to use in the file. To use the sensor macro the user uses the XACRO sensor tag and then changes the parameters accordingly to its needs. To better understand the tags to use it is a

good idea to analyze the example files *test_d[model]_camera.urdf.xacro* included by the developer. The *autonomous_vehicle_gazebo* package can now be launched and it generates topics in which the frames captured by the camera are published. The different topics post different types of data retrieved by the intel realsense camera, for example, the RGB camera and the depth camera are posted in different topics.

7 Control & PPO implementation

After creating the environment the following step is to create low-level code that retrieves data and controls the simulation environment. This code is then used to create more abstract code that implements the mechanisms of RL algorithms, in this case, PPO. This chapter is divided into two sections the first "Environment methods implementation" and the second "PPO methods implementation". The first chapter explains classes and the respective methods that implement low-level functionalities, the second explains the high-level PPO class that uses the low-level functionalities to implement PPO mechanisms.

7.1 Environment methods implementation

The method *step* includes most of the mechanisms needed for the algorithm to apply the agent choice in the environment and retrieve the observations needed. This method receives through parameters an action, which is executed by the model in the environment and returns the reward and the vehicle state after the action. A vehicle action consists of a speed applied to the vehicle back wheels and an axle angle to the vehicle's front wheels choosing both the direction and the vehicle speed. A state consists of a processed vehicle camera frame. The reward is a value relative to the vehicle's displacement to the center of the lane. The class *AutonomousVehicle* retrieves all the data needed from the simulation (camera frames and vehicle position) and sets the back wheels' speed and axle angle. The class *ComputerVision* receives the frames stored in the *AutonomousVehicle* class and processes them to get the vehicle's state. The *get_reward_2* method retrieves the vehicle's position stored in the *AutonomousVehicle* class and returns the vehicle reward. The *get_reward_2* method also returns the variable *done* value which indicates if the agent has reached a terminal state. The *Environment* class gathers the functionalities from all the other classes and implements the high-level method *step*. A thorough explanation of each class and the methods implementation is carried out in this section.

7.1.1 Simulation drivers

This subsection shows how the simulation drivers are implemented. The drivers are the methods that transmit data such as wheel speed and axle direction and receive data such as camera frames and vehicle's position. This communication is carried out using ROS nodes that receive data (node publisher) and transmit data (node subscriber) to and from the simulation environment. The implementation is carried out using python3 and the sending and receiving of data to and from ROS topics are carried out using the library rospy. The library rospy allows the user to create nodes and then use the nodes to

subscribe and publish to topics. The drivers are implemented inside a class named *AutonomousVehicle*.

a) Init node topics and variables

The point of the constructor is to initiate a node, subscribe the node to Gazebo topics that send message with important data, and set the node as publisher in topics that control the environment. Another important thing is create an object of *CvBridge* this object is important to process frames received from the simulation environment to later be used with OpenCV functions.

```
10  class AutonomousVehicle:
11
12      def __init__(self):
13
14          # Init Node
15          rospy.init_node('autonomous_vehicle_control', anonymous=True)
16
17          # Init node publishers to control angle of axle
18          self.left_axle_controller_pub = rospy.Publisher(
19              '/autonomous_vehicle/left_axle_controller/command', Float64, queue_size=10)
20          self.right_axle_controller_pub = rospy.Publisher(
21              '/autonomous_vehicle/right_axle_controller/command', Float64, queue_size=10)
22
23          # Init node publishers to control back wheels speed
24          self.left_motor_controller_pub = rospy.Publisher(
25              '/autonomous_vehicle/right_motor_controller/command', Float64, queue_size=10)
26          self.right_motor_controller_pub = rospy.Publisher(
27              '/autonomous_vehicle/left_motor_controller/command', Float64, queue_size=10)
28
29          # Init node subscriber to get position of vehicle
30          # This generates a callback to the self.callback_vehicle_state function
31          rospy.Subscriber("/gazebo/model_states",
32                           gazebo_msgs.msg.ModelStates, self.callback_vehicle_state)
33
34          # Define variables for image processing
35          self.cv_image_bridge = CvBridge()
36
37          # Init node subscriber to get image frames from camera plug in
38          # This generates a callback to the self.callback_camera_autonomous_vehicle function
39          # The callback is done for every new frame published in the topic
40          self.ros_image = rospy.Subscriber(
41              "/camera/color/image_raw", Image, self.callback_camera_autonomous_vehicle)
42
43          # Init variables for camera frames
44          self.frame_cnt = 0
45
46          # Init variables for vehicle position in simulation
47          self.vehicle_pose_xy = [None, None]
```

Figure 7.31: Initiate node subscriber publisher variable

- line 15: Initiate the node
- line 17-21: Set node as publisher in the topics that control the axle angle in the simulation environment
- line 23-27: Set node as publisher to topics that control the back-wheels speed
- line 31: Set node as subscriber in topics that publish data about the vehicle's position
- line 35: Create *CvBridge* object that turns the received frames into frames that can be processed using OpenCV functions

- line 40-41: Set node as subscriber to topic that publishes the vehicle's camera frames
- line 44-47: Init variables

b) Control Actuators

Controlling the wheel's speed and axle angle consists of publishing the desired wheel speed value and axle angle value in the respective topics. The library rospy allows the user to publish values to topics by using the *publish* function. When initiating the node as publisher the rospy method returns an object which is used to call the *publish* function, this function publishes the message to the respective topic. The parameter the *publish* function receives is the data to message to the respective topic.

```
49      def drive_autonomous_vehicle(self, autonomous_vehicle_angle, autonomous_vehicle_speed):
50
51          # Publish data to topics to control axle angle
52          self.left_axle_controller_pub.publish(autonomous_vehicle_angle)
53          self.right_axle_controller_pub.publish(autonomous_vehicle_angle)
54
55          # Publish data to topics to control back wheel speed
56          self.right_motor_controller_pub.publish(-autonomous_vehicle_speed)
57          self.left_motor_controller_pub.publish(-autonomous_vehicle_speed)
58
```

Figure 7.32: Publish to topics that control vehicle's actuators

- line 52-53: Publish in the respective topic the axle angle value
- line 56-57: Publish in the respective topic the back wheels speed value

c) Retrieve simulation data

When the node is set as a subscriber to a topic it is associated with a callback function. This callback function is going to be executed every time the topic receives a new message from the publisher node. The received data is passed through the parameter variable. In this case, two callback functions are used, one that receives the frames from the vehicle's camera and one that receives the position of the vehicle in the environment. Both callbacks store the received data in variables, the difference is that the camera callback counts the frames received and processes the frames so that the frames can later be processed using OpenCV functions. The function that retrieves the frames rejects N frames before returning a frame, this is carried out so that when the algorithm receives the next state gets a state with some variation. If no frame was rejected the current state and the next state would be very similar.

```
59      def callback_camera_autonomous_vehicle(self, data):
60
61          try:
62
63              # Increment the frame counter
64              self.frame_cnt += 1
65
66              # Convert the received frame
67              self.camera_frame = self.cv_image_bridge.imgmsg_to_cv2(
68                  data, "bgr8")
69
70          # In case of error converting the frame
71          except CvBridgeError as e:
72              self.frame_cnt = 0
73              print(e)
74
75      def get_frame_reject_N(self, N_frames_to_reject):
76
77          # Reset the frame counter
78          self.frame_cnt = 0
79
80          # Reject N frames
81          while self.frame_cnt < N_frames_to_reject:
82              pass
83
84          # Return N+1 Frame
85          return self.camera_frame
86
87      def callback_vehicle_state(self, data):
88
89          # Store the position of the vehicle to the variable
90          self.vehicle_pose_xy = [
91              data.pose[2].position.x, data.pose[2].position.y]
92
93      def get_autonomous_vehicle_position(self):
94
95          # Return position of the vehicle
96          return self.vehicle_pose_xy
97
```

Figure 7.33: Get position and camera frames from simulation environment

- line 64: Increment variable to count number of frames received
- line 67: Convert frames to OpenCV format

- line 71-73: In case of some error when converting the frame this exception is executed
- line 78: Reset frame counter to start counting number of frames rejected
- line 81-82: Stall until frame counter is equal to the number of frames to reject
- line 85: Return frame after rejecting N
- line 90-91: Store the vehicle position
- line 96: Return vehicle position

7.1.2 Computer vision

The algorithm starts by receiving a raw frame of the vehicle's camera. The frame is then processed to get the important environment features and minimize noise. The frame is first cropped in order to capture only the region of interest, the floor part that shows the road lines. Then the frame is converted to grayscale. The grayscale passes through a down-scaling process, to save memory/CPU resources and besides the algorithm does not need much definition to recognize the road lines. The cropped frame passes through a binary threshold filter. The binary threshold filter receives a threshold value and applies it to the image, setting to zero the pixels with value lower than the threshold and to one the remaining pixels. Finally, the frame is flattened, which means it is turned into an array in order to plug it into the RL algorithm.

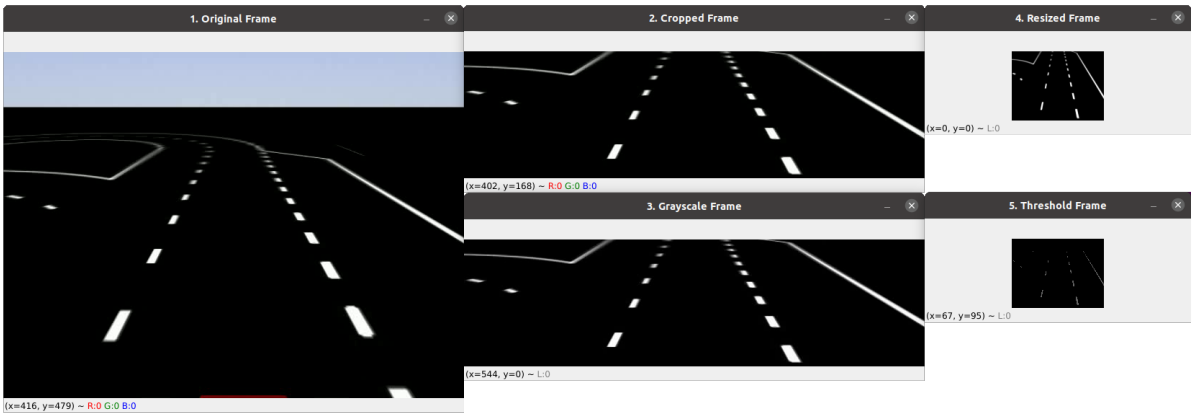


Figure 7.34: Computer vision algorithm

a) Init computer vision algorithm

The computer vision algorithm class constructor sets the variables, which are used to calculate the downscaled frame shape and the region of interest to crop. The value of these variables is defined by the user. The method *get_image_shape* calculates and stores the downscaled frame shape to then downscale the frame during the feature extraction process.

```
4   class ComputerVision:
5
6       def __init__(self, crop_height_up_below):
7
8           # Store crop dimensions
9           self.crop_height_up_below = crop_height_up_below
10
11          # Store the down-scale percentage
12          self.scale_percent = 20
13
14      def get_image_shape(self, frame):
15
16          # Store original frame shape
17          self.original_frame_shape = frame.shape
18
19          # Calculate target downscale dimensions
20          downscaled_width = int(self.original_frame_shape[1] * self.scale_percent / 100)
21          downscaled_height = int(self.original_frame_shape[0] * self.scale_percent / 100)
22
23          # Store target downscale dimensions
24          self.downscaled_dim = (downscaled_width, downscaled_height)
```

Figure 7.35: Init computer vision algorithm

- line 9: Set the region of interest which is the area that will be cropped
- line 12: Set the frame percentage to downscale, this value is used to calculate the downscaled frame shape
- line 17: Store the frame shape before downscaling
- line 20-21: Calculate the downscaled frame width and height, this is calculated dividing the original frame width/height by the percentage of downscaled frame
- line 24: Store the downscaled frame shape

b) Computer vision algorithm

The method *crop_frame* crops the frame to get only the region of interest. The method *flatten_frame* removes the dimension from the frame's matrix and returns an array with the frame's pixels values. The method *computer_vision_algorithm* implements the whole feature extraction process.

```
26    def crop_frame(self, frame):
27
28        # Crop the frame and return
29        return frame[self.crop_height_up_below[0]:self.crop_height_up_below[1],:]
30
31    def flatten_frame(self, frame):
32
33        # Remove frame dimensions
34        return frame.reshape(-1)
35
36    def computer_vision_algorithm(self, frame):
37
38        # Crop Frame
39        cropped_frame = self.crop_frame(frame)
40
41        # Convert to grayscale
42        grayscale = cv2.cvtColor(cropped_frame, cv2.COLOR_BGR2GRAY)
43
44        # Downscale the frame
45        resized = cv2.resize(grayscale, self.downscaled_dim, interpolation = cv2.INTER_AREA)
46
47        # Apply threshold
48        _,thresh1 = cv2.threshold(resized,200,1,cv2.THRESH_BINARY)
49
50        # Return flattened frame
51        return self.flatten_frame(thresh1)
```

Figure 7.36: Computer vision algorithm

- line 29: Crop the figure accordingly to the variables defined in the constructor
- line 34: This function turns the matrix of values into and array
- line 39: Get the cropped frame
- line 42: Convert the cropped frame into grayscale
- line 45: Downscale the grayscale frame accordingly
- line 48: Apply the threshold to the cropped grayscale downscaled frame. The threshold value is 200. Every pixel with value less than 200 is set to 0 all the other pixels are set to 1.
- line 51: Flatten the frame returned by the threshold filter and return it.

7.1.3 Reward

The class *RewardFunction* returns the reward of the vehicle and the variable *done*. The variable *done* notifies whether the vehicle has reached a terminal state or not. When the vehicle reaches a terminal state *done* is 1 otherwise it is 0. To calculate the reward, the reward system receives the vehicle's coordinates in the environment and uses these values to calculate the vehicle's displacement off the lane center. The reward returned is inversely proportional to the vehicle's displacement to the center of the lane.

Calculating the vehicle's displacement to the center of the lane using the coordinates returned by the simulation is one of the challenges of this dissertation. First, the track is divided into sections, to understand how the sections are divided see the background colors of figure 7.37.

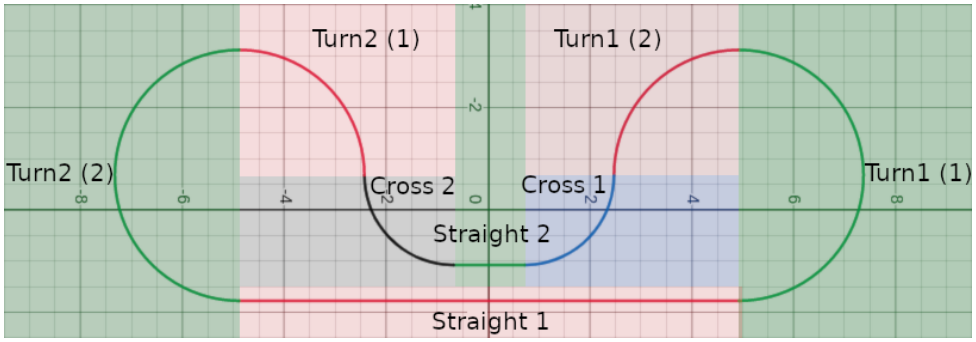


Figure 7.37: Track sections and respective ideal trajectories

Using the vehicle's position coordinates it is possible to understand in which section it is located by comparing it to the limits of each section. After knowing the section in which the vehicle is the next step is to use the parametric equation that describes the ideal trajectory of that particular section and calculate the vehicle's displacement to the ideal trajectory. The track is mathematically simple to describe using parametric equations because it consists of circles with different radius and center points and straight lines. There are two types of equations used, the circle and the straight line. The straight line is a linear equation and because the straight lines are parallel to the y axis then the straight line equation is just x equal to a constant. The circle equation is the following in which the variables h and k correspond to the circle's center coordinates.

$$(x-h)^2 + (y-k)^2 = r^2 \tag{7.21}$$

One thing that is important to point out is that these equations are calculated using the autonomous driving challenge official measurements and coordinates. The only difference is that in simulation the

track's origin is different, so an offset is added to compensate for this difference. The vehicle's displacement in a straight line is given by the difference between the vehicle x coordinate value and the equation's constant. The vehicle's displacement in a turn is calculated using the respective parametric equation values (h,k,r) and the vehicle's coordinates (x,y) in the following formula:

$$e = \sqrt{(x-h)^2 + (y-k)^2 - r} \tag{7.22}$$

With the described system it is possible to calculate the vehicle displacement along the entire track and get the current track section which is most of the data needed to create the reward system. Another reward system parameter is the vehicle speed. The speed penalty is used to motivate the algorithm to navigate the track. If the algorithm did not receive a negative penalty for being still it would never explore the environment because by being still it would get a better reward than navigating and crashing. The vehicle's speed is a value defined by the actor and can be any value between 0 rad/s and 6 rad/s. The rewards and penalties are the following:

- Reward if the vehicle's speed is higher than speed threshold, episode continues.
- Penalty if the vehicle's speed is lower than the speed threshold, episode continues.
- Penalty proportional to the vehicle's displacement relative to the center of the lane, episode continues.
- Penalty if the the vehicle's displacement is higher than the displacement threshold, episode ends.
- Penalty if the vehicle skips a section, episode ends.

a) Init reward class

The *RewardFunction* class constructor sets the variables. The list *track_position_array* stores the name of each track section. The list *driving_line_array* stores the parameters that describe the ideal trajectory for each track section. This list stores two data types, one that has three values and a data type that has only one value. The data types with three values describe a turn, and the data types with only one value describe a straight line. Both lists are sorted, meaning the values stored with index 0 correspond to the first track section, the value stored with index 1 corresponds to the second track section, etc. This is especially useful to check if the vehicle is in the right track section. The variable *current_track_section* stores the current track section the vehicle is at. When the vehicle changes section this variable is used to check whether the vehicle is at the correct track section. The other variables store values relative to the rewards.

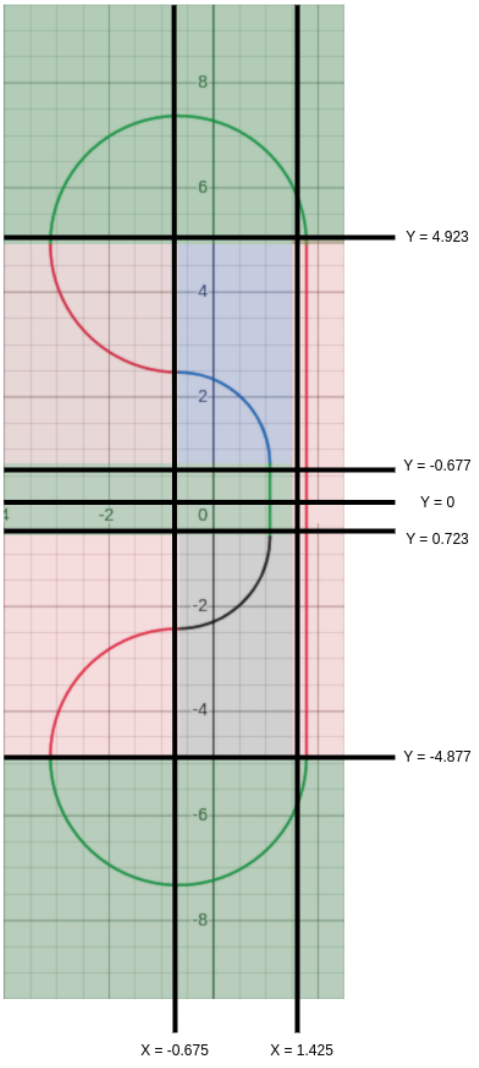
```
3   class RewardFunction:
4
5       def __init__(self):
6
7           # Store name of each track section
8           self.track_position_array = ['Straight 1','Turn 1 (1)','Turn 1 (2)','Cross 1',
9               'Straight 2','Cross 2','Turn 2 (1)','Turn 2 (2)']
10
11          # Store parameters of each section ideal trajectory
12          self.driving_line_array = [[1.775],[-0.675, 4.923, 2.45],[-0.675, 4.923, 2.45],
13              [-0.675,0.723,1.75],[1.075],[-0.675, -0.677, 1.75],[-0.675, -4.877, 2.45],[-0.675, -4.877, 2.45]]
14
15          # Store current track section
16          self.current_track_section = 0
17
18          # Rewards
19          self.penalty_section = -10
20          self.penalty_track_limits = -10
21          self.penalty_driving_line_error = -1
22          self.threshold_track_limits = 0.25
23          self.threshold_vehicle_speed = 3
24          self.penalty_speed = -1
25          self.reward_speed = 1
```

Figure 7.38: Reward function init

- line 8-9: Set list that stores the name of each section
- line 12-13: Set list that stores the ideal trajectory parameters in each track section
- line 16: Set variable that stores the track section the vehicle is at. Variable used to confirm if the vehicle didn't jump track sections
- line 19-25: Set variables that store the agent reward

b) Get vehicle's current track section

The method *vehicle_track_position* identifies the vehicle's current track section. This method receives the vehicle's coordinates relative to the environment's origin and returns the matching track section. This method is a switch case operation that eliminates possible track positions until it finds out the correct current track section. The value returned by this method corresponds to the index of *driving_line_array* that stores the ideal trajectory parameters for the respective track section.




```
42      def vehicle_track_position(self, xy):
43
44          X = xy[0]
45          Y = xy[1]
46
47          # Turn 1 (1)
48          if Y > 4.923:
49
50              return 1
51
52          # Turn 2 (2)
53          elif Y < -4.877:
54
55              return 7
56
57          elif X < -0.675:
58
59              # Turn 1 (2)
60              if Y > 0:
61
62                  return 2
63
64              # Turn 2 (1)
65              else:
66
67                  return 6
68
69          # Straight 1
70          elif X > 1.425:
71
72              return 0
73
74          # Cross 2
75          elif Y < -0.677:
76
77              return 5
78
79          # Cross 1
80          elif Y > 0.723:
81
82              return 3
83
84          # Straight 2
85          else:
86
87              return 4
```

Figure 7.39: Vehicle track section limits

c) Get vehicle's lane displacement

To calculate the vehicle's displacement to the center of the lane first the algorithm gets the vehicle's current track section to get the parameters that define the ideal trajectory. The method *vehicle_track_position* receives the vehicle's coordinates and returns a number, which is the *driving_line_array* list index that stores the ideal trajectory parameters for the current track section. The list returns one value when the ideal trajectory is a straight line and returns three values when the ideal trajectory is a turn. To distinguish a straight line from a turn there is a condition that compares the number of parameters returned by the list *driving_line_array*. To calculate the vehicle's displacement to the center of the lane there are two methods, one that calculates the displacement in a straight line *driving_line_error_straight* and a method that calculates the displacement in a turn *driving_line_error_turn*. The condition checks if the ideal trajectory is a straight line or a turn by analyzing the number of parameters returned and calculates the vehicle's displacement to the ideal trajectory.

The straight lines in this track are parallel to the y axis so to calculate the displacement to a straight line is just calculating the difference between the straight line parameter and the x vehicle coordinate. Calculate the vehicle's displacement in a turn is just applying the formula $e = \sqrt{(x-h)^2 + (y-k)^2 - r}$.

- Line 34-40: Store the vehicle's coordinates (x and y) and the ideal turn parameters (h,k and r) in individual variables for a clearer code.
- line 43: Calculate and return the vehicle's displacement to the turn using the formula $e = \sqrt{(x-h)^2 + (y-k)^2 - r}$
- line 48-51: Store the x vehicle's coordinate and the ideal straight line parameter in individual variables for a clearer code.
- line 54: Return the difference between the straight line parameter and the x vehicle coordinate, this values is the vehicle's displacement to ideal trajectory.
- line 59: Identify the current track section. This method receives the current vehicle coordinates and returns a value that corresponds to the *driving_line_array* list index that stores the parameters that describe the ideal trajectory for the current track section.
- line 62: Use the value returned from *vehicle_track_position* to get the ideal trajectory parameter(s) from the the list *driving_line_array*

- line 65-73: If the list *driving_line_array* returns more than one parameter it means that the current ideal trajectory is a turn and if it returns only one parameter it means the ideal trajectory is a straight line. The respective method is used to calculate the vehicle's displacement to the ideal trajectory.

```
31      def driving_line_error_turn(self, xy, HKr):
32
33          # Store the vehicle's coodinates
34          X = xy[0]
35          Y = xy[1]
36
37          # Store the current section ideal trajectory parameters
38          H = HKr[0]
39          K = HKr[1]
40          r = HKr[2]
41
42          # Calculate and return displacement
43          return np.abs(np.sqrt( (X - H)**2 + (Y - K)**2 ) - r )
44
45      def driving_line_error_straight(self, xy, Xl):
46
47          # Store the vehicle's X coodinate
48          X = xy[0]
49
50          # Store the current section ideal trajectory parameter
51          Xl = Xl[0]
52
53          # Calculate and return displacement
54          return np.abs(Xl - X)
55
56      def driving_line_error(self, xy):
57
58          # Get vehicle's current coordinates
59          vehicle_track_section = self.vehicle_track_position(xy)
60
61          # Get current section idealtrajectory parameters
62          driving_line_coord = self.driving_line_array[vehicle_track_section]
63
64          # Destinguish striaght line vs turn
65          if len(driving_line_coord) > 1:
66
67              # Return displacement when in turn
68              return self.driving_line_error_turn(xy, driving_line_coord), vehicle_track_section
69
70          else:
71
72              # Return displacement when in straight
73              return self.driving_line_error_straight(xy, driving_line_coord), vehicle_track_section
```

Figure 7.40: Calculate displacement

d) Get vehicle's reward

The algorithm calculates the displacement of the vehicle and gets the current track section, and with this the algorithm checks whether it is the same track section as before, if the track section is not the same as before the algorithm checks if the new section is correct. The correct track section order is in the list *driving_line_array*, so checking if the new track section is correct consists of comparing the new track section to the previous track section succeeding index. When the vehicle completes a lap the new track section is set to 0.

When the vehicle changes section and the new section is not correct one, a penalty of -10 is returned and *done* is returned with a value of 1 making the episode end. If the new section is correct the episode continues. If the vehicle's displacement to the ideal trajectory is higher than the threshold a penalty of -10 is returned and *done* is returned with a value of 1 making the episode end. If the vehicle's displacement is lower than the threshold value the episode continues and the vehicle's displacement is used to calculate the reward returned. If the vehicle speed is higher than the speed threshold a positive reward of 1 is stored if not a penalty of -1 is stored. In the end, a reward is calculated by summing the displacement penalty that multiplies with a constant of value -1 and the reward/penalty relative to the vehicle's speed.

- line 186: The method receives the vehicle's coordinates in the environment and returns the vehicle's displacement to the ideal trajectory and the track section.
- line 189: Check if the new track section is the same as the previously stored track section. If the section is the same the process jumps to line 210.
- line 192-194: Check if the new section is correct, the sections have a specific order. This order corresponds to the order in the list *driving_line_array*. To check if the vehicle is correctly driving around the track is just checking if the new section is after the current section. Store the new track section as the current track section.
- line 198-201: Check if the new section is correct. If the vehicle completes a full lap around the track the new section vehicle has to be the first section. Store the new track section as the current track section.
- line 204-207: If the new vehicle didn't complete a lap around the track and the new section is not the next in order it means that the vehicle didn't navigate around the track as it is supposed. What happens is this method returns a penalty of -10 and returns *done* as 1, this makes the episode stop.

- line 210-213: If the vehicle has a higher displacement than the defined threshold this method returns a penalty of -10 and returns *done* as 1 this makes the episode stop.
- line 219-228: If the vehicle's speed is lower than the defined speed threshold a penalty is set in the *speed_val* variable, if the vehicle's speed is higher than defined threshold a reward is set in the *speed_val* variable. The variable *speed_val* is used to calculate the reward returned.
- line 231: Calculate the reward, the reward is given by summing the speed penalty/reward and the displacement penalty. The variable *done* is returned with a value of zero, so the episode continues

```
180     def get_reward_2(self, xy, wheel_speed):
181
182         # The speed of the vehicle clipped
183         wheel_speed = np.clip(wheel_speed, 0, 6)
184
185         # Calulate the displacement of the vehicle and return the current section
186         error, track_section = self.driving_line_error(xy)
187
188         # If the new track section is different from the current track section
189         if track_section != self.current_track_section:
190
191             # If the new track section is the correct
192             if track_section == self.current_track_section + 1:
193
194                 # Update the current track section with the new track section
195                 self.current_track_section = track_section
196
197             # If the new track section is the last track section
198             elif self.current_track_section == 7 and track_section == 0:
199
200                 # Update the current track section with the first track section
201                 self.current_track_section = track_section
202
203             # If the vehicle jumped is not in the correct track section
204             else:
205
206                 # Return the penalty for jumping a track section and end episode
207                 return self.penalty_section, 1
208
209         # If the displacment of the vehicle is higher that the respective threshold
210         if error > self.threshold_track_limits:
211
212             # Return the penalty for exciding track limits and end episode
213             return self.penalty_track_limits, 1
214
215         # If the the displacement did not exceed track limits
216         else:
217
218             # If the speed of the vehicle is smaller than speed threshold
219             if wheel_speed < self.threshold_vehicle_speed:
220
221                 # Get a penalty
222                 speed_val = self.penalty_speed
223
224             # If the speed of the vehicle is higher than speed threshold
225             else:
226
227                 # Get a reward
228                 speed_val = self.reward_speed
229
230             # Return the Reward and continue episode
231             return speed_val + error * self.penalty_driving_line_error, 0
```

Figure 7.41: Calculate reward code

Figure 7.42: Calculate reward flowchart

7.1.4 Environment

The *Environment* class gathers the simulation drivers, computer vision algorithm, and reward system in the *step* method which receives thorough parameters the action to execute. The action is the back wheels speed and the front wheels angle. The *step* uses the simulation drivers implemented to execute the action in the environment. After discarding N frames the algorithm returns a processed frame which is the state. The *step* method also gets the vehicle's coordinates from the simulation drivers and uses the reward function methods to get the reward and *done* value.

The *Environment* class also implements other utility methods. The utility functions return the action/state space dimensions and store evaluation process videos. The *reset* method resets the simulation and then waits for the simulation to reset and return the first state. Since the utility methods are simple or not essential for the program execution these are not explained.

a) Environment Init

The environment class constructor starts by storing the objects of the *ComputerVision*, *RewardFunction*, and *Environment* class to call the respective methods. Then the vehicle's starting position coordinates are stored in lists. This is carried out because each episode the vehicle starts in one of four different positions. To define the vehicle's position a message is sent using ROS service, which has to be of type *ModelState*. The variable *set_position_msg* is a message of type *ModelState* that stores the vehicle's starting position coordinates to be sent through ROS service.

```
14  class Environment:
15
16      def __init__(self):
17
18          # Init Class
19          self.autonomous_vehicle = AutonomousVehicle()
20          self.computer_vision = ComputerVision((144, 320))
21          self.reward_function = RewardFunction()
22
23          # Store Starting position values
24          self.starting_position = 0
25          self.starting_position_array = [0, 2, 4, 3]
26          self.starting_array_position_x = [1.81, -3.09, 1.11, 1.03]
27          self.starting_array_position_y = [-4.39, 4.347, 0.7, 1.12]
28          self.starting_array_position_z = [0.06, 0.062, 0.06, 0.06]
29          self.starting_array_orientation_z = [-0.7,-0.78, -0.73, -0.72]
30          self.starting_array_orientation_w = [0.7, -0.62, -0.69, -0.69]
31
32          # Init variables to set position
33          self.set_position_msg = ModelState()
34          self.set_position_msg.model_name = "autonomous_vehicle"
35          self.set_position_msg.pose.position.x = self.starting_array_position_x[0]
36          self.set_position_msg.pose.position.y = self.starting_array_position_y[0]
37          self.set_position_msg.pose.position.z = self.starting_array_position_z[0]
38          self.set_position_msg.pose.orientation.x = 0
39          self.set_position_msg.pose.orientation.y = 0
40          self.set_position_msg.pose.orientation.z = self.starting_array_orientation_z[0]
41          self.set_position_msg.pose.orientation.w = self.starting_array_orientation_w[0]
```

Figure 7.43: Environment class init

- line 19-21: Set objects of class *ComputerVision*, *RewardFunction*, and *Environment*, these objects allow the class environment to gather methods to implement "step"
- line 24-38: Set lists that store the vehicle's starting position coordinates
- line 33-41: Set variables that store the vehicle's starting coordinate to set the simulation using the respective ROS service.

b) Step method

The *step* method receives the actions returned by the agent neural network. The agent neural network has a last layer normalized with a sigmoid activation function. This means that the neural network's output are values between 0 and 1 with a shape of $f(x) = \frac{1}{1+e^{-x}}$. The wheels speed values are between 0 rad/s and 6 rad/s and the angle value are between -0.27 rad and 0.27 rad so the actions outputted by the neural network have to be processed before being applied to the simulation drivers.

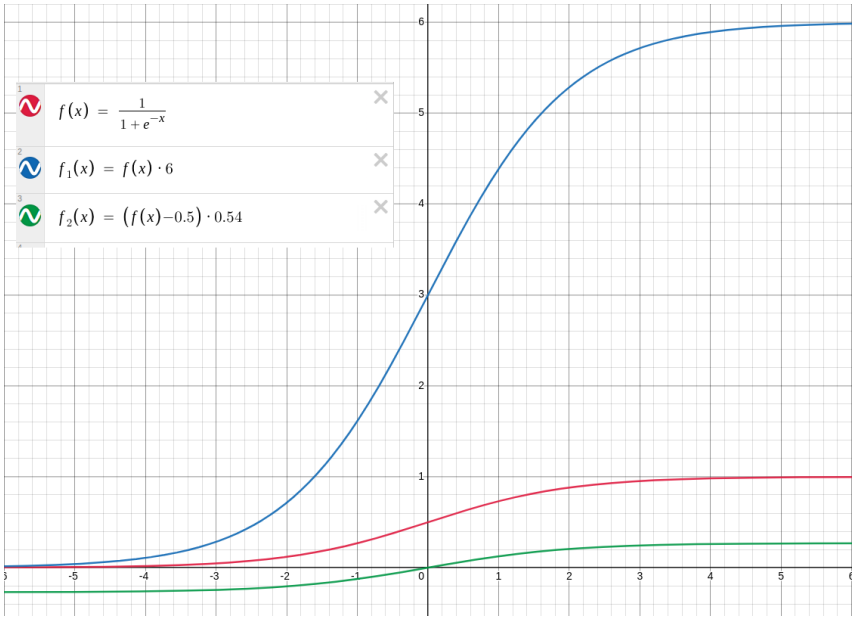


Figure 7.44: Action process

After processing the values returned by the neural network the same are clipped to make sure that the values applied on the actuator are not higher than what they should be. The values are then applied to the actuator and the algorithm retrieves the state, and reward and *done* to return it.

- line 141: Retrieve the vehicle's frame after rejecting N frames
- line 144: Retrieve the vehicle's coordinates in the environment
- line 147: With the vehicle's coordinates calculate the reward and get the *done* value
- line 150: Process the frame retrieved from the environment using the computer vision algorithm implemented to get the state.
- line 153: Return the *state*, *reward* and *done* variables

- line 158-161: Process the action values outputted by the neural network as described in the figure 7.44
- line 164-167: Clip the speed and angle values to make sure the values are not higher or lower than the limits
- 170: Apply the angle and speed values on the wheels and axle.
- 173-176: Retrieve and return the *state reward* and *done* variables

```
138     def get_state_reward(self,wheel_speed):
139 
140         # Get frame after rejecting N frames
141         frame = self.autonomous_vehicle.get_frame_reject_N(self.frames_to_reject)
142 
143         # Get position of the vehicle
144         position = self.autonomous_vehicle.get_autonomous_vehicle_position()
145 
146         # Get reward and done
147         reward, done = self.reward_function.get_reward_2(position, wheel_speed)
148 
149         # Process the frame to get the state
150         flatten = self.computer_vision.computer_vision_algorithm(frame)
151 
152         # Return state reward and done
153         return flatten, reward, done
154 
155     def step(self, action):
156 
157         # Adjust the action of the neural nerwork
158         angle = (action[0] - 0.5) * 0.54
159 
160         # Adjust the action of the neural nerwork
161         speed = action[1] * 6
162 
163         # Clip angle value
164         angle = np.clip(angle, -0.27, 0.27)
165 
166         # Clip speed value
167         speed = np.clip(speed, 0, 6)
168 
169         # Execute the action
170         self.autonomous_vehicle.drive_autonomous_vehicle(angle, speed)
171 
172         # Get the state, reward and done
173         state, reward, done = self.get_state_reward(speed)
174 
175         # Return State reward and done
176         return state, reward, done
```

Figure 7.45: Environment step method

7.2 PPO methods implementation

This section explains the PPO algorithm implementation. There is a class named PPO that implements the high-level algorithm methods which are *learn*, *train* and *test*. This class stores an object of the class *Policy* (actor) and *Value* (critic), *MemoryCollector*, and *ZFilter*. The class *Policy* and *Value* are neural networks implemented using tensorflow2, more specifically Keras, which is included in tensorflow2. The *Policy* neural network has a few specific methods. The policy neural network chooses an action, this action is used to create a normal distribution. After creating the normal distribution a sample is retrieved and the value is the action executed in the environment. The memory collector class collects samples from the environment. To do this it uses the step method to execute actions in the environment and retrieve the state, reward, and *done* variables. It also uses the methods from the neural networks to retrieve the actions and the respective predicted values. After completing the memory collection process the general advantage estimation is calculated and finally, the values can be used to make the gradient update.

7.2.1 Neural networks

An actor-critic algorithm uses two neural networks. An actor (policy) and a critic (value). There are two ways of creating a neural network using Keras. In this dissertation, the neural networks are created by subclassing the model class. To create a model using this process the user has to implement two methods: *call* and *__init__*. Inside the *__init__* method the user specifies the model shape and other parameters/variables. In the method *call* the user specifies the model's forward pass. In this project, *call* does not define the forward pass because the sequential method is being used. The *Sequential* method defines the model's forward pass which simplifies the *call* method. After creating a sequential model object, to get the output, the input is passed as an argument to the object and it returns the model's output. Defining the input shape is carried out by using the method *build*. The *build* method has an *input_shape* argument that allows specifying the input layer shape.

The critic neural network objective is to train a model capable of estimating the states value in the environment, so the input is a state and the output is a value prediction of that state. The critic neural network class has the *__init__* method that defines the critic neural network model and the *call* method that inputs a value in the neural networks and returns the output.

```
5  class Value(tf.keras.Model):
6      def __init__(self, dim_state, dim_hidden=300, activation=tf.nn.leaky_relu, l2_reg=1e-3):
7          super(Value, self).__init__()
8
9          # Store parameters of the neural network
10         self.dim_state = dim_state
11         self.dim_hidden = dim_hidden
12
13         # Define the neural network hidden and output layer
14         self.value = tf.keras.models.Sequential([
15             layers.Dense(self.dim_hidden, activation=activation, kernel_regularizer=tf.keras.regularizers.l2(l=l2_reg)),
16             layers.Dense(self.dim_hidden, activation=activation, kernel_regularizer=tf.keras.regularizers.l2(l=l2_reg)),
17             layers.Dense(1, kernel_regularizer=tf.keras.regularizers.l2(l=l2_reg))
18         ])
19
20         # Define the shape of the input layer
21         self.value.build(input_shape=(None, self.dim_state))
22
23
24     def call(self, states, **kwargs):
25
26         # Input the variable states and get the output of the neural network
27         value = self.value(states)
28
29         # Return the output of the neural network
30         return value
```

Figure 7.46: Critic neural network

- line 10-11: Store parameters that define the model
- line 14-18: Define the neural network model and store the neural network object in the variable value. To get the neural network output for any input is just using the generated object and sending through arguments the model's input

- line 21: Define the input layer size
- line 27: Input the states to the neural network and retrieve the output
- line 30: Return the neural network's output

The actor neural network receives in the input a state and the output is a set of actions. The actor neural network output are two values, one for the vehicle's speed and one for the vehicle's direction. The actor is supposed to explore which consists of choosing actions with a certain level of randomness. This randomness allows the agent to search for the best way of solving the problem. If the agent did not have some randomness in its action choices it would probably never encounter the optimal policy. Exploring is great because by doing a different random action the new action can be compared to actions the policy would choose.

To ensure exploration the agent chooses values, these values are used to set the mean of a normal distribution, the action values are then sampled from the normal distribution created. What this means is that the neural network outputs values, and these values define the mean of a normal distribution, the normal distribution created is used to sample actions that are used to take a step in the environment.

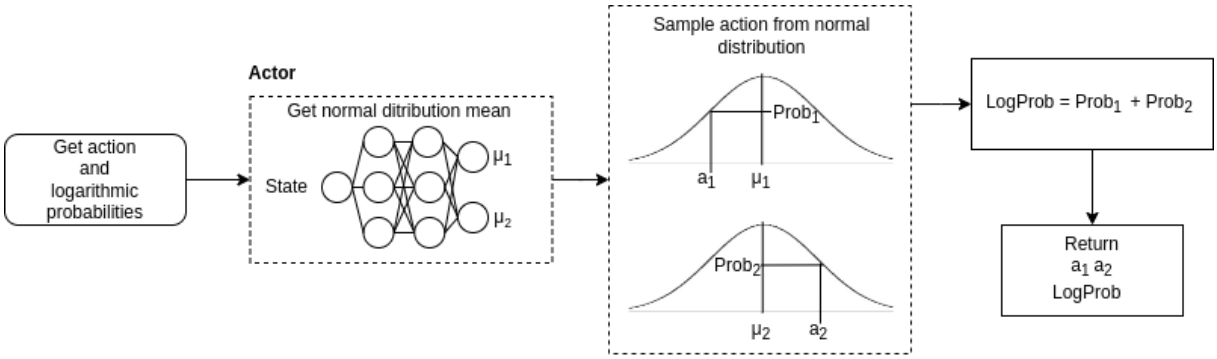


Figure 7.47: Exploration algorithm

A normal distribution is described with two parameters, the mean and the standard deviation. The normal distribution mean is defined by the neural network output, the standard deviation is a *Policy* class parameter. The standard deviation is set as a trainable variable. When a variable is set as trainable it is updated in the gradient update process. The standard deviation defines how random the actions are and the gradient changes the model's parameters accordingly to how big the loss is. The following example helps understand how the standard deviation works in this project. At the beginning of the training, the policy is bad, so if the standard deviation is very small the agent follows a bad policy very strictly, the gradient starts increasing the standard deviation because taking random actions will find better results

than following a bad policy. As the policy gets better the gradient will start decreasing the standard deviation because following the policy is better than taking random actions until the deviation is so small it follows the policy and that's when the algorithm converges to an optimal solution.

```
7   class Policy(tf.keras.Model):
8       def __init__(
9           self,
10          dim_state,
11          dim_action,
12          dim_hidden=300,
13          activation=tf.nn.relu,
14          log_std=1,
15      ):
16          super(Policy, self).__init__(dim_state, dim_action, dim_hidden)
17
18          # Store parameters of the neural network
19          self.dim_action = dim_action
20          self.dim_state = dim_state
21          self.dim_hidden = dim_hidden
22
23          # Define the neural network hidden and output layer
24          self.policy = tf.keras.models.Sequential(
25              [
26                  layers.Dense(self.dim_hidden, activation=activation),
27                  layers.Dense(self.dim_hidden, activation=activation),
28                  layers.Dense(self.dim_action, activation=tf.nn.sigmoid),
29              ]
30          )
31
32          # Define the shape of the input layer
33          self.policy.build(input_shape=(None, self.dim_state))
34
35          # Define the standard deviation variable
36          self.log_std = tf.Variable(
37              name="action_log_std",
38              initial_value=tf.zeros((1, dim_action), dtype=tf.float64) * log_std,
39              trainable=True,
40          )
```

Figure 7.48: Actor Neural Network

- line 19-21: Store parameters that define the model
- line 24-29: Define the neural network model and store the neural network object in the variable policy. To get the neural network output for any input just using the generated object and sending through arguments the model's input
- line 33: Define the input layer size
- line 27: Define the standard deviation variable, which is set as a trainable variable.

- line 47: Set the variable states in the neural network's input and store the output
- line 52: Create an array with the same standard deviation for each output neural network's neuron.
- line 56: Convert the logarithmic standard deviation to a normal standard deviation
- line 63: Get the mean and standard deviation
- line 67: Create a normal distribution object with the values returned from _get_dist

```
43      @tf.function
44      def _get_dist(self, states):
45
46          # Get the output of the neural network
47          mean = self.policy(states)
48          mean = tf.cast(mean,tf.float64)
49
50          # Creates an array of deviation
51          # For each action
52          log_std = tf.ones_like(mean) * self.log_std
53
54          # The deviation is now converted
55          # Converted from logarithic to normal
56          std = tf.exp(log_std)
57          return mean, std
58
59      @tf.function
60      def call(self, states, **kwargs):
61
62          # Gets mean from NN and std value
63          mean, std = self._get_dist(states)
64
65          # Creates distributions to sample a action
66          # With the values retrieved
67          dist = Normal(mean, std)
68
69          # Sample a value from the normal distribution
70          action = dist.sample()
71
72          # Sums all the logarithmic probabilities of all the taken actions
73          log_prob = tf.reduce_sum(dist.log_prob(action), -1)
74
75          # Return action and
76          # Logarithmic probability of executing that action
77          return action, log_prob
```

Figure 7.49: Actor methods

- line 70: Get random normally distributed action
- line 73: Get the returned action probability

7.2.2 Memory collector

The *MemoryCollector* runs multiple episodes in the environment and stores a tuple with the following data:

- state - State received by the environment before choosing an action
- action - Action returned by the agent neural network for the current state
- reward - Reward received after executing the chosen action and getting the new state
- next_state - State received after executing the action chose in the environment
- mask - Stores the value of *done* variable

The stored data is then used to make the gradient updates. Some data needs to be processed before being used. The neural network returns actions in the shape of tensors. Tensors cannot be used directly on the drivers created for the simulation environment so the actions have to be converted for NumPy data

types. The states, on the other hand, need to be inputted into the neural network as tensors and since the state is a python list it needs to be converted to a tensor. The memory collection process ends when the buffer used to make the gradient update is full.

```
34    # Cycle finishes when buffer (min_batch_size) is full
35    while num_steps < min_batch_size:
36
37        # Reset the environment and retrieve first state
38        state = env.reset()
39        episode_reward = 0
40
41        # Apply zfilter to the state
42        if running_state:
43            state = running_state(state)
44
45        # Max iterations in episode
46        for t in range(10000):
47
48            # Periodically data is diplayed to the user
49            if not num_steps %500:
50
51                print("Memory Collector Iter: ", num_steps)
52                print("Episode Iteration Mean: ",mean_iter)
53
54            # Make the state a tensor to input it in neural network
55            state_tensor = tf.expand_dims(
56                tf.convert_to_tensor(state, dtype=TDOUBLE), axis=0
57            )
58
59            # Process action and log_prob to store it
60            action, log_prob = policy.get_action_log_prob(state_tensor)
61
62            # Convert from tensor to numpy value
63            action = action.numpy()[0]
64            log_prob = log_prob.numpy()[0]
65
66            # Use the action returned to step in environment
67            next_state, reward, done = env.step(action)
68            episode_reward += reward
69
70            # Aplly z filter
71            if running_state:
72                next_state = running_state(next_state)
73
74            # Apply mask
75            mask = 0 if done else 1
76
77            # Store tuple
78            memory.push(state, action, reward, next_state, mask, log_prob)
79            num_steps += 1
80
81            # Break if terminal state or buffer is full
82            if done or num_steps >= min_batch_size:
83                break
84
85            # Next state become the current state
86            state = next_state
```

Figure 7.50: Memory collector code

- line 38-39: Reset environment and get first state
- line 49-52: Print data if the iteration is a multiple of 500
- line 55-57: Convert the state to a tensor so it can be inputted to the neural network
- line 60: Get action sampled from the normal distribution generated from neural network output
- line 63-64: Convert values returned by the neural network
- line 67: Step on the environment using the action returned by the neural network

- line 78: Store the data retrieved from the environment

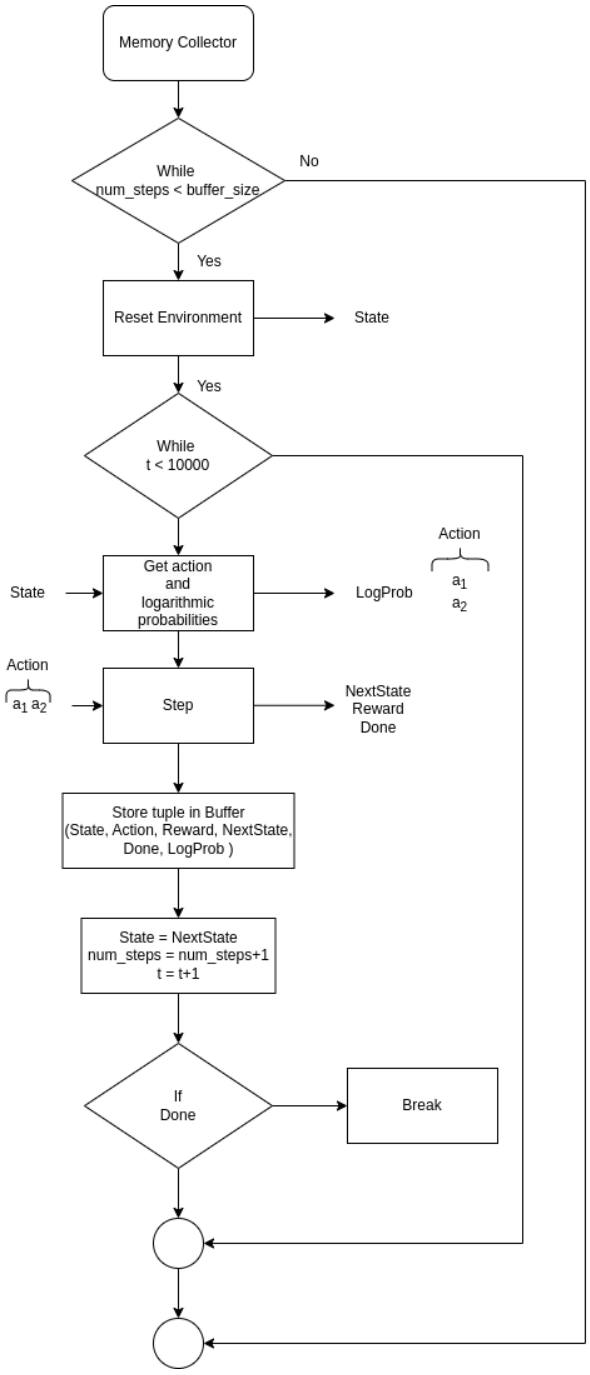


Figure 7.51: Memory collector flowchart

7.2.3 GAE

To calculate GAE the *MemoryCollector* stores the reward, and the critic's value estimation and *done* at each time step. The GAE method creates a list with the advantage at each time step and returns it. The most efficient way of calculating the GAE is by iterating the *MemoryCollector* data backward and calculate the advantage using the GAE formula.

$$\hat{A}_t^{GAE(\gamma,\lambda)}(s,a) = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{7.23}$$

$$\delta_t = r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t) \tag{7.24}$$

During the episode the critic makes prediction of the states values, these prediction are stored and used after the episode to calculate the advantage $\hat{A}(s,a)$. If the advantage is negative it means that the state estimation was optimistic and the state ended being worse than predicted, if the advantage is positive it means that the prediction was pessimistic and the state ended being better than expected, in both cases the critic's prediction is wrong. The following formula is used to calculate the critic's loss.

$$V(s) = \hat{A}(s,a) + \hat{V}(s) \tag{7.25}$$

The critic's value prediction is given by $\hat{V}(s)$, the advantage estimation is given by $\hat{A}(s,a)$ and $V(s)$ is the state value calculated through experience. If the advantage estimation $\hat{A}(s,a)$ is positive it means that the value estimation $\hat{V}(s)$ for that state s is smaller than what it should be. What this means is that the actual value of $V(s)$ is higher than the initial value prediction carried out by the critic. On the other hand if the advantage estimation $\hat{A}(s,a)$ is negative it means that the value estimation $\hat{V}(s)$ for that state s is higher than what it should be. What this means is that the actual value of $V(s)$ is lower than the initial value prediction carried out by the critic. The value calculated $V(s)$ is used in the following loss function formula:

$$L_{critic} = (V(s) - \hat{V}(s))^2 \tag{7.26}$$

To understand this formula it is important to keep in mind the fact that the data retrieved from the episodes is divided into mini batches and successive gradient updates are executed using the different

batches. The critic neural network is different after each gradient update meaning that the value prediction for the same input has different output compared to the neural network before the gradient update. The objective is to make the new value estimations $\hat{V}(s)$ closer to the value calculated through experience.

- line 10: Get the batch size
- line 11-12: Set the lists that store the advantage and the deltas
- line 19: Cycle through the whole batch backwards
- line 23: Calculate the deltas $r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t)$
- line 26: Calculate the advantage $\sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V$
- line 32: Calculate $A(s,a) + \hat{V}(s)$

```
 9  def estimate_advantages(rewards, masks, values, gamma, tau, eps=1e-8):
10      batch_size = rewards.shape[0]
11      deltas = np.zeros((batch_size,1), dtype=NDOUBLE)
12      advantages = np.zeros((batch_size,1), dtype=NDOUBLE)
13
14      # Set the auxiliar variables
15      prev_value = 0
16      prev_advantage = 0
17
18      # Iterate the batch backawards
19      for i in reversed(range(batch_size)):
20
21
22          # Calculate delta
23          deltas[i] = rewards[i] + gamma * prev_value * masks[i] - values[i]
24
25          # Calculate advantage
26          advantages[i] = deltas[i] + gamma * tau * prev_advantage * masks[i]
27
28          prev_value = values[i]
29          prev_advantage = advantages[i]
30
31      # The return is used to calculate the loss of the critic
32      returns = values + advantages
```

Figure 7.52: GAE implementation

7.2.4 Gradient Update

The gradient update is the learning process core. In each gradient update, the algorithm has to store the differential equations of each model parameter. The tensorflow2 library allows this automatic differentiation. The following example shows how to implement a basic gradient update using the tensorflow2 automatic differentiation method.

In this example the gradient update is applied to the variables of a neural network stored in the object *net*. The optimizer is stored in the *optimizer* object. The *loss* function is defined by the user and has to be calculated using a tensor datatype.

```
11  # Gradient tape stores the data needed to calculate gradient
12  with tf.GradientTape() as tape:
13
14    # Calculate loss
15    loss = (...)
16
17  # Get the gradients
18  grads = tape.gradient(loss, net.trainable_variables)
19
20  # Apply gradients using
21  optimizer.apply_gradients(
22    grads_and_vars=zip(grads, net.trainable_variables))
```

Figure 7.53: Basic gradient update

This represents the basic structure of a gradient update using the automatic differentiation of the tensorflow2 library.

- line 12: Begin the gradient update process, *tape* is an object that gives access to all the model's gradients.
- line 15: The *loss* is calculated inside the gradient cycle. The *loss* value has to be a tensor datatype.
- line 18: Retrieve the gradients calculated inside the gradient process
- line 21: Apply the updates to the model variables using the defined optimizer and the calculated gradients.

The automatic differentiation makes implementing the gradient update much easier because the user only needs to implement the loss function, the library uses the loss function to calculate the gradients

and make the gradient update on the neural network variables. The critic uses a mean squared error loss function.

$$L_{critic} = (V(s) - \hat{V}(s))^2 \tag{7.27}$$

```
12      # Create loss object
13      critic_loss_fn = tf.keras.losses.MeanSquaredError()
14
15      # Cycle gradient update optim_value_iternum times
16      for _ in range(optim_value_iternum):
17
18          # Gradient tape stores the data needed to calculate gradient
19          with tf.GradientTape() as tape:
20
21              # Make value estimation for each state
22              values_pred = value_net(states)
23
24              # Calculate critic loss
25              value_loss = critic_loss_fn(returns, y_pred=values_pred)
26
27          # Get the gradients
28          grads = tape.gradient(value_loss, value_net.trainable_variables)
29
30          # Apply gradients using
31          optimizer_value.apply_gradients(
32              grads_and_vars=zip(grads, value_net.trainable_variables))
```

Figure 7.54: Critic gradient update

- line 13: Create an object that calculates automatically the mean squared error $(y - \hat{y})^2$
- line 16: Repeat the gradient update number of times defined by the user
- line 19: Begin gradient update process
- line 22-25: Calculate the mean square error loss
- line 28-32: Get the gradient values and apply them to the model variables using the optimizer defined

The actor loss function used is the defined in the PPO paper:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \tag{7.28}$$

$$L^{CLIP}(\theta) = E_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{7.29}$$

$$L_t^{CLIP+S} = E_t[L_t^{CLIP}(\theta) + c_2 S[\pi_\theta](s_t)] \tag{7.30}$$

```
36      # Gradient tape stores the data needed to calculate gradient
37      with tf.GradientTape() as tape:
38
39          # Get logarithmic probabilities of executing each action
40          log_probs = tf.expand_dims(policy_net.get_log_prob(states, actions), axis=-1)
41
42          # Calculate the ratio
43          ratio = tf.exp(log_probs - old_log_probs)
44
45          # Calculate surrogate objective 1
46          surr1 = ratio * advantages
47
48          # Calculate surrogate objective 2
49          surr2 = tf.clip_by_value(
50              ratio, 1.0 - clip_epsilon, 1.0 + clip_epsilon) * advantages
51
52          # Calculate entropy
53          entropy = tf.reduce_mean(policy_net.get_entropy(states))
54
55          # Calculate the loss
56          policy_loss = - tf.reduce_mean(tf.minimum(surr1, surr2)) - entropy_coeff * entropy
57
58      # Get the gradients
59      grads = tape.gradient(policy_loss, policy_net.trainable_variables)
60
61      # Apply gradients using optimizer
62      optimizer_policy.apply_gradients(
63          grads_and_vars=zip(grads, policy_net.trainable_variables))
```

Figure 7.55: Actor gradient update

- line 37: Begin gradient update process
- line 40: Calculate $\pi_\theta(a_t|s_t)$
- line 43: Calculate $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$
- line 49: Calculate $clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)$
- line 53: Calculate the normal distribution entropy generated to choose the action
- line 56: Calculate the actor loss $L_t^{CLIP+S} = E_t[L_t^{CLIP}(\theta) + c_2 S[\pi_\theta](s_t)]$
- line 59-62: Get the gradient values and apply them to the model variables using the optimizer defined

7.2.5 PPO Learn

The PPO class implements the *learn* method, which gathers all the mechanisms. First, it executes the memory collection method which gathers data from the environment. The gathered data is used to calculate the general advantage estimation. The data needed to execute the gradient update is divided into mini-batches and the mini-batches are then used to make the gradient updates.

- line 143: Run multiple episodes and store data in the buffer until it is full. This returns a memory object that stores the states, actions, rewards, next state, mask and log probability.
- line 145-147: Prints information so the user can check the learning process
- line 150-155: Update data to the tensorboard data base.
- line 157: Convert the memory data.
- line 159-164: Process the data batch
- line 166: Calculate the advantage using GAE
- line 171-172: Calculate how many mini batches are needed to store all the data stored in the batch
- line 174: Repeat the gradient update for all epochs. The number of epochs is an hyperparameter that defines how many times the same data is used in the gradient update.
- line 175: Get a random list of numbers to make the gradient update in random order of mini batches
- line 176: Repeat the gradient update for all the mini batches
- line 177-178: Calculate the current mini batch indexes.
- line 180-188: Convert the tensors to numpy values to slice the mini batch out of the batch and convert it to a tensor again so that the mini batches can be used in the gradient update process

```python
    def learn(self, writer, i_iter):
        """learn model"""
        memory, log = self.collector.collect_samples(self.min_batch_size)

        print(f"Iter: {i_iter}, num steps: {log['num_steps']}, total reward: {log['total_reward']: .4f}, "
              f"min reward: {log['min_episode_reward']: .4f}, max reward: {log['max_episode_reward']: .4f}, "
              f"average reward: {log['avg_reward']: .4f}, sample time: {log['sample_time']: .4f}")

        # record reward information
        with writer.as_default():
            tf.summary.scalar("total reward", log['total_reward'], i_iter)
            tf.summary.scalar("average reward", log['avg_reward'], i_iter)
            tf.summary.scalar("min reward", log['min_episode_reward'], i_iter)
            tf.summary.scalar("max reward", log['max_episode_reward'], i_iter)
            tf.summary.scalar("num steps", log['num_steps'], i_iter)

        batch = memory.sample()  # sample all items in memory

        batch_state = NDOUBLE(batch.state)
        batch_action = NDOUBLE(batch.action)
        batch_reward = NDOUBLE(batch.reward)
        batch_mask = NDOUBLE(batch.mask)
        batch_log_prob = NDOUBLE(batch.log_prob)[:, None]
        batch_value = tf.stop_gradient(self.value_net(batch_state))

        batch_advantage, batch_return = estimate_advantages(batch_reward, batch_mask, batch_value, self.gamma,
                                                            self.tau)

        log_stats = {}
        if self.ppo_mini_batch_size:
            batch_size = batch_state.shape[0]
            mini_batch_num = batch_size // self.ppo_mini_batch_size

            for e in range(self.ppo_epochs):
                perm = np.random.permutation(batch_size)
                for i in range(mini_batch_num):
                    ind = perm[slice(
                        i * self.ppo_mini_batch_size, min(batch_size, (i + 1) * self.ppo_mini_batch_size))]

                    aux_batch_return = batch_return.numpy()

                    aux_batch_return = aux_batch_return[ind]

                    aux_batch_return = tf.convert_to_tensor(aux_batch_return)

                    aux_batch_advantage = batch_advantage.numpy()[ind]

                    aux_batch_advantage = tf.convert_to_tensor(
                        aux_batch_advantage)

                    log_stats = ppo_step(self.policy_net, self.value_net, self.optimizer_p, self.optimizer_v, 1,
                                         batch_state[ind], batch_action[ind], aux_batch_return, aux_batch_advantage, batch_log_prob[ind],
                                         self.clip_epsilon
                                         )
```

Figure 7.56: PPO Learn code

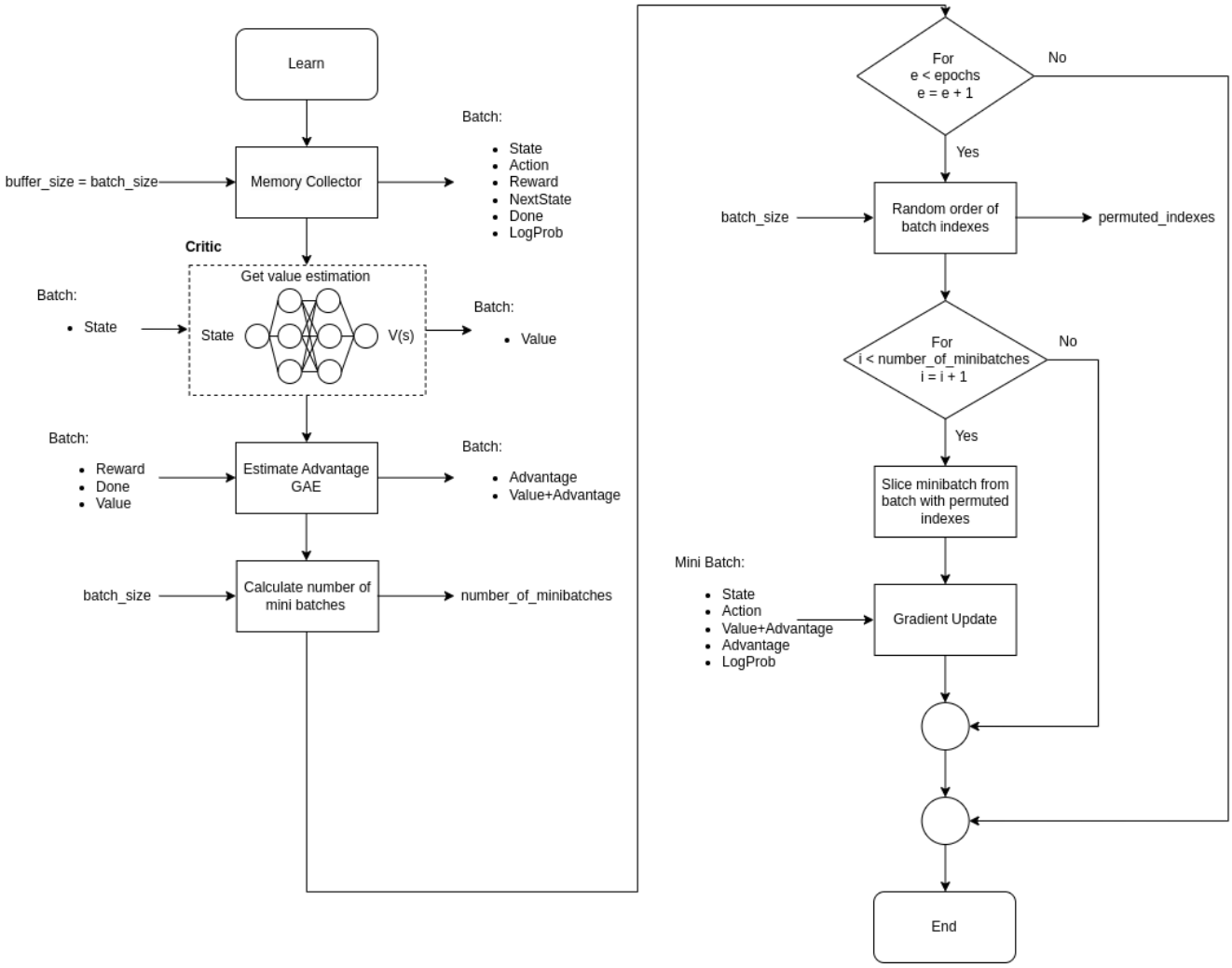


Figure 7.57: PPO Learn flowchart

8 Results

With the algorithm implementation, the simulation environment and the communication between the two the algorithm is ready to start training. This chapters exposes the hyperparameters used to train the model, the problems faced and final results achieved.

8.1 Hyperparameters tuning

This sub section exposes the process of experimenting with different hyperparameters and some conclusions drawn from these experiments. The hyperparameters tuned are the learning rate, and the mini batch size. The main parameter used to compare each learning process is average reward evolution achieved by the algorithm in each learning iteration.

8.1.1 Learning rate

The objective of this experiment is to analyse the learning rate impact in the algorithm's learning process. This experiment consists of running two learning processes during 150 learning iterations with two different learning rates and analyze the development of each. The neural networks have two hidden layers and all the activation functions are ReLU, except for the output layer which is sigmoid. The hidden layers have 250 neurons each. The following table presents the hyperparameters of each learning process. The hyperparameters are all the same in both processes except for the learning rates which are 10 times higher in Learn 1 than in Learn 2.

Hyperparameters	Learn 1	Learn 2
Learning rate	10^{-3}	10^{-4}
Gamma	0,99	0,99
Tau	0,95	0,95
Clip	0,2	0,2
Buffer size	2048	2048
Mini batch size	256	256
Epochs	10	10

After running both learning processes it is possible to conclude that the learning process with lower learning rate achieves a better result than the result achieved by the learning process with higher learning rate. The higher learning rate is unstable and achieves a smaller average reward than the average reward

achieved by the smaller learning rate. Higher learning rate means higher jumps in the gradient, this is what makes Learn 1 unstable, it makes big gradient updates that often are unfounded leading to destructive gradient updates. The algorithm with high learning rate will most likely never achieve an optimal policy and even if it achieves it most likely never converge.

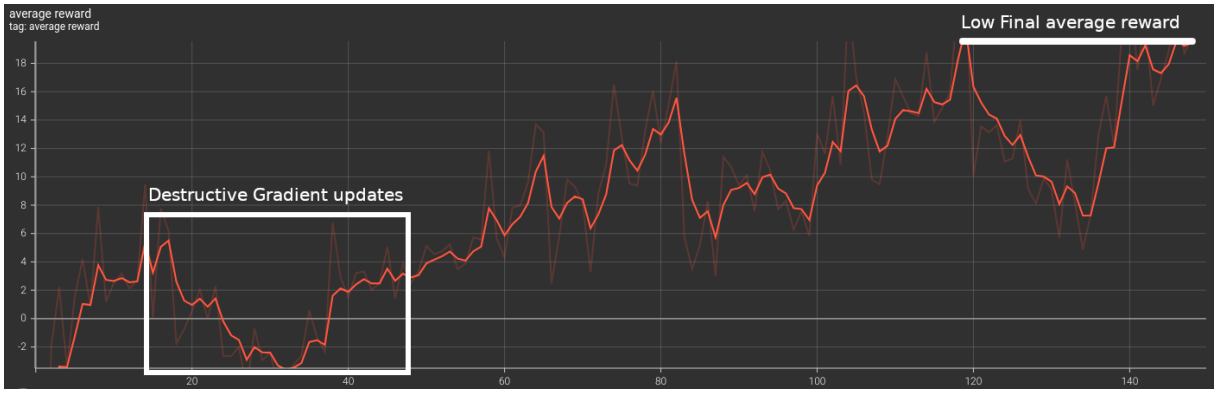


Figure 8.58: Learn 1 - Learning rate 10^{-3}

Figure 8.59: Learn 2 - Learning rate 10^{-4}

8.1.2 Mini batch size

This experiment intends to analyze the mini batch size influence in the learning process. The experiment consists of running two learning processes with different mini batch sizes. The learning process Learn 3 has a mini batch size of 128 and the learning process Learn 4 has a mini batch size of 256. Both learning processes took 300 learning iterations.

Hyperparameters	Learn 3	Learn 4
Learning rate	$5 * 10^{-5}$	$5 * 10^{-5}$
Gamma	0,99	0,99
Tau	0,95	0,95
Clip	0,2	0,2
Buffer size	2048	2048
Mini batch size	128	256
Epochs	10	10

In this test the algorithms achieve similar final results. The learning process Learn 3 seems more unstable than Learn 4 but this is not conclusive because Learn 3 achieves better results earlier than Learn 4, and than its performance drops to lower values very close to the values achieved by Learn 4 at that point. From that point onwards, both learning processes have similar improvement rate.

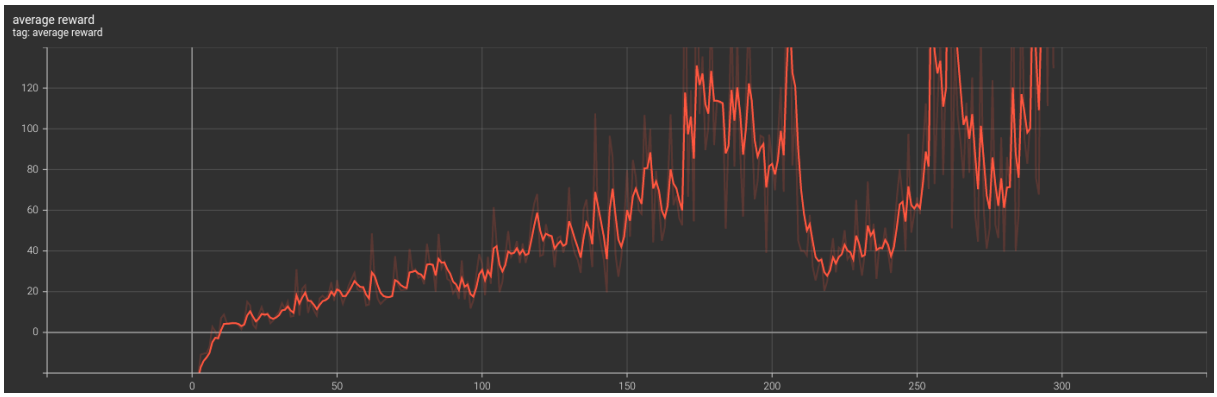


Figure 8.60: Learn 3 - Mini batch size 128

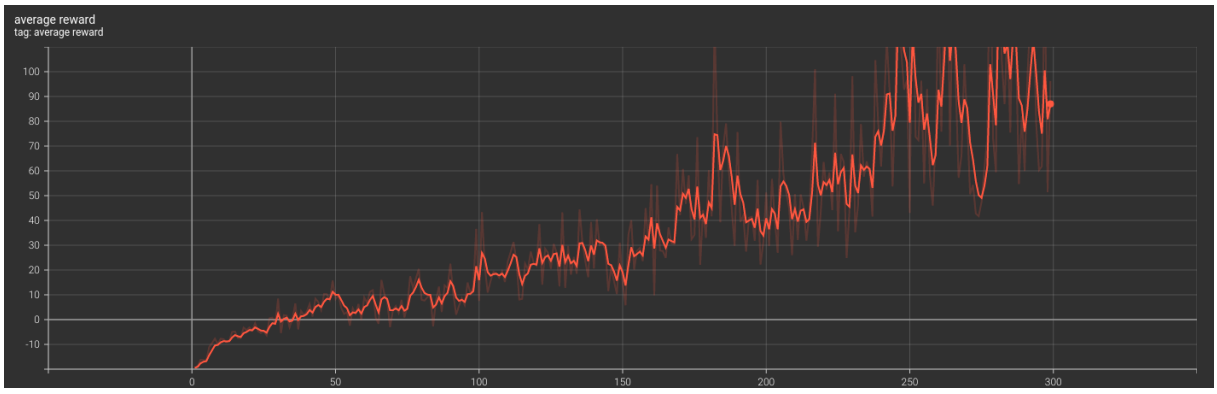


Figure 8.61: Learn 4 - Mini batch size 256

8.2 Final results

This subsection exposes the final results achieved after running the training algorithm. The training consists of running the training algorithm as described in the track described in the figure 8.62. The track shown is a simplified version of the original track, as it does not have the pedestrians crossing or the parking spaces. The pedestrians crossing and the parking spaces are data introduced in the states which is unnecessary and just creates uncertainty. The objective of this track is to test whether the algorithm can create a policy capable of navigating the track in the simplest conditions. The vehicle's starting positions and orientations are described in the figure 8.62 using red squares for the positions and red arrows for directions. The table shows the hyperparameters used in all the final trainings.

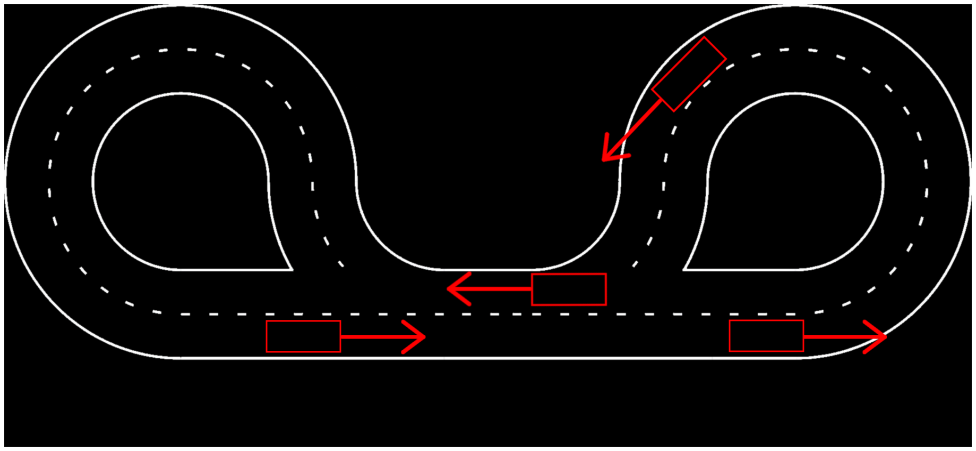


Figure 8.62: Simplified track - Without pedestrians crossing and parking

Learning rate	$1 * 10^{-5}$
Gamma	0,99
Tau	0,95
Clip	0,2
Buffer size	4096
Mini batch size	128
Epochs	10

To save time, this training starts with the policy and critic trained during the hyperparameter tuning stage. This is to avoid wasting time training a policy and a critic from scratch. The training algorithm crashed after four days of training. Another training process starts using the policy and critic trained until the crash. This training runs smoothly until it converges to the optimal policy after 2 days of training. Figure 8.63 and 8.64 show the average reward progression of both training processes.

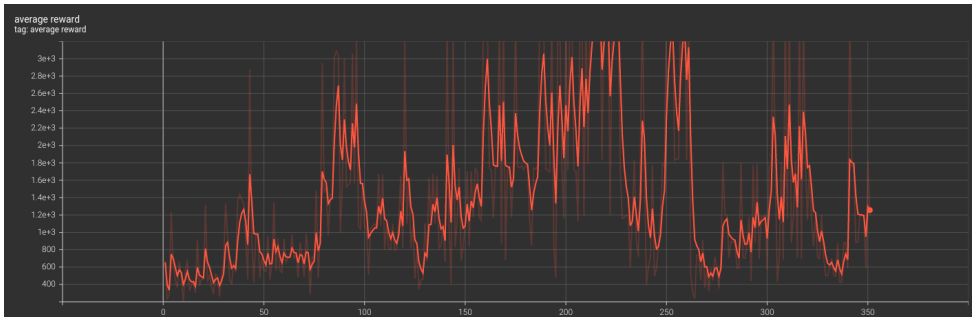


Figure 8.63: Simple training 1

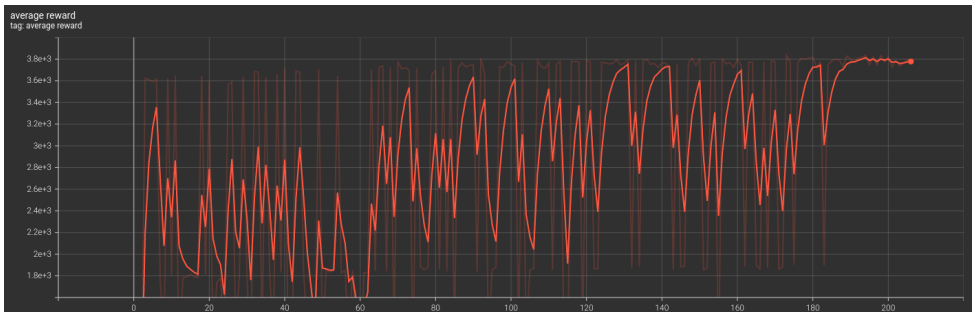


Figure 8.64: Simple training 2

Figure 8.64 shows the policy converging to the optimal solution. After ending the training process a test is conducted to test the policy. The test consists of running episodes and retrieve the states, input the states to the policy neural network, retrieve the output actions and control the vehicle using these actions. The episodes have 2400 iterations each and the vehicle did not crash in any of the 5 episodes. With this test it is possible to conclude that the optimal policy found in this training is capable of consistently navigating the track.

After training a policy capable of navigating the simple track 8.62 the next step is to train the algorithm to navigate the original track with pedestrians crossings and the parking spots. The original track is shown in the figure 8.66. Each episode of the training started from a different spot of the track. The starting spots on the track are the same used in the track 8.62. The starting policy and critic are the result of the training in the simple track.

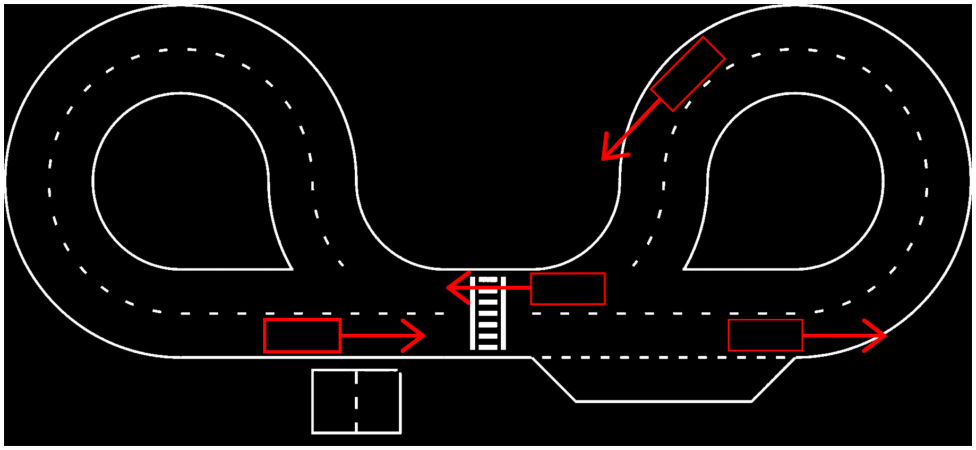


Figure 8.65: Original track - With pedestrians crossing and parking

The results of this training are shown in the Figure. After 8 days of training the process is terminated by the user because during 4 days the learning is unstable and shows no signs of progress. The policy appears to learn the basics of the navigating the track but never converges to a consistent policy. The policy either succeeds following the road lanes and fails to ignore the parking spots/pedestrians crossings or successfully ignores the parking spots/pedestrians crossings and lacks consistency when following the road lanes. Many different training processes failed to complete the challenge of navigating the track with parking spots/pedestrians crossings and after a long time of trying different solutions this task is determined as unresolved.

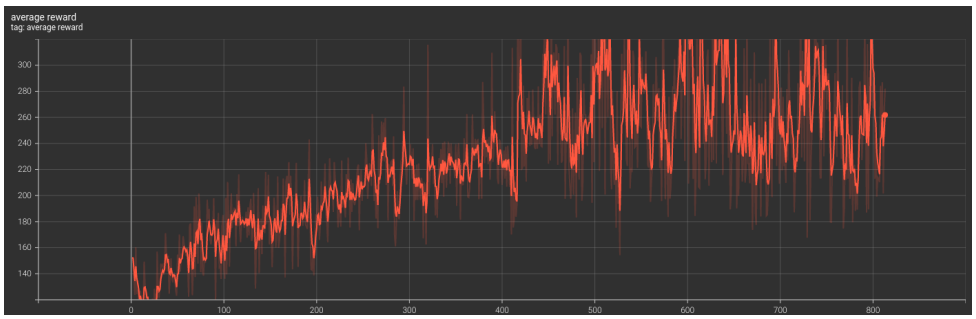


Figure 8.66: Final training

9 Conclusion and future work

The objectives set at the beginning of this dissertation were to develop an RL algorithm capable of creating a policy that can navigate the proposed track. The algorithm has to use as input a camera. After validating the algorithm in simulation the objective is to test it's learning in the real world. The algorithm was not capable of navigating the proposed track on simulation and the learning mechanisms were not tested in the real world. This ended up being a very ambitious project that could not be fully developed within the time limit. Even though the results do not satisfy the objectives proposed at the beginning of the dissertation these results are still very good and it is important to focus on the work actually developed and the amount of research carried out and described in this document. This dissertation describes the basic concepts needed for the development of the most advanced RL algorithms at the moment. Besides the theoretical concepts it shows how to implement an advanced RL algorithm using python3 and tensorflow2, the process of developing a simulation environment with Gazebo+ROS, and basic image processing using OpenCV. The theoretical concepts are applied in the final project that uses all the tools described to create an algorithm capable of training a policy to navigate a simplified version of the original track with consistency. The ROS packages created can be used by anyone who wants to test RL algorithms in computer vision-based autonomous driving.

This dissertation can be the starting point for other projects. Other developers can use the simulation environment to test different algorithms. The next step of this dissertation would be to create a stronger feature extraction algorithm capable of executing line segmentation. This algorithm would most likely get a policy capable of navigating the track with pedestrians crossing/parking with consistency. This belief is justified by the fact that the algorithm can create a policy capable of navigating the track without pedestrians crossing/parking. If the feature extraction makes the line segmentation it eliminates the pedestrians crossing/parking from the input, so it would work like the simplified track in which the algorithm is capable of creating a policy capable of navigating the track with consistency. After implementing the line segmentation in the feature extraction the next step would be to introduce object detection and identification. With this data, the algorithm would be capable of detecting and identifying objects in its trajectory. This functionality could be associated with the depth sensor of the real sense camera. With this data, the policy would be able to detect, identify and locate obstacles in its trajectory.

Bibliographic references

Ackerman, E. (2021, Jun). *What full autonomy means for the waymo driver.* IEEE Spectrum. Retrieved from https://spectrum.ieee.org/full-autonomy-waymo-driver

Agarap, A. F. (2018). *Deep learning using rectified linear units (relu).* arXiv. Retrieved from https://arxiv.org/abs/1803.08375 doi: 10.48550/ARXIV.1803.08375

Alphago. (2017). Retrieved from https://www.youtube.com/watch?v=WXuK6gekU1Y

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-13*(5), 834-846. doi: 10.1109/TSMC.1983.6313077

Bellman, R. (1957). A markovian decision process. *Journal of Mathematics and Mechanics*, *6*(5), 679–684. Retrieved 2023-01-19, from http://www.jstor.org/stable/24900506

Burns, M. (2019, Apr). *'anyone relying on lidar is doomed,' elon musk says.* TechCrunch. Retrieved from https://techcrunch.com/2019/04/22/anyone-relying-on-lidar-is-doomed-elon-musk-says/

Cao, S. (2021, Jun). *Elon musk reflects on tesla's darkest hour: I gave the last of my remaining cash.* Retrieved from https://observer.com/2021/06/elon-musk-recall-tesla-2008-financial-crisis-twitter/

Cohen, J. (2021, Jul). *How radars work.* Think Autonomous. Retrieved from https://medium.com/think-autonomous/how-radars-work-1eb523893d62?readmore=1amp;source=user_profile

CoppeliaSim. (2022, Sep). *Robot simulator coppeliasim: Create, compose, simulate, any robot - coppelia robotics.* Retrieved from https://www.coppeliarobotics.com/

Farama-Foundation, F.-F. (2022, Dec). *Farama-foundation/gymnasium: A standard api for reinforcement learning and a diverse set of reference environments (formerly gym).* Retrieved from https://github.com/Farama-Foundation/Gymnasium

from SAE J3016™, S. I. (2021, Apr). *Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles.* Retrieved from https://saemobilus.sae.org/content/J3016_202104/

Kiran, M., & Ozyildirim, M. (2022). *Hyperparameter tuning for deep reinforcement learning applications.* arXiv. Retrieved from https://arxiv.org/abs/2201.11182 doi: 10.48550/ARXIV.2201.11182

Kocher, L. (2022, May). *Deepmind's open source mujoco is available on github.* Retrieved from

https://www.opensourceforu.com/2022/05/deepminds-open-source-mujoco-is-available-on-github/

Lee, K.-F., & Chen, Q. (2021). *Ai 2041*. Currency.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... Wierstra, D. (2015). *Continuous control with deep reinforcement learning.* arXiv. Retrieved from https://arxiv.org/abs/1509.02971 doi: 10.48550/ARXIV.1509.02971

Mazzari, V., & Mazzari, V. (2021, Jul). *What is lidar technology?* Retrieved from https://www.generationrobots.com/blog/en/what-is-lidar-technology/

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). *Playing atari with deep reinforcement learning.* arXiv. Retrieved from https://arxiv.org/abs/1312.5602 doi: 10.48550/ARXIV.1312.5602

Modayil, J., White, A., & Sutton, R. (2011, 12). Multi-timescale nexting in a reinforcement learning robot. *Adaptive Behavior*, *22*. doi: 10.1177/1059712313511648

Moreno, J. (2021, Jan). *Waymo ceo says tesla is not a competitor, gives estimated cost of autonomous vehicles.* Forbes Magazine. Retrieved from https://www.forbes.com/sites/johanmoreno/2021/01/22/waymo-ceo-says-tesla-is-not-a-competitor-gives-estimated-cost-of-autonomous-vehicles/?sh=480500f9541b

Morris, J. (2022, Nov). *Why is tesla's full self-driving only level 2 autonomous?* Forbes Magazine. Retrieved from https://www.forbes.com/sites/jamesmorris/2021/03/13/why-is-teslas-full-self-driving-only-level-2-autonomous/?sh=3f029dec6a32

Pendleton, S., Andersen, H., Du, X., Shen, X., Meghjani, M., Eng, Y., ... Jr, M. (2017, 02). Perception, planning, control, and coordination for autonomous vehicles. *Machines*, *5*, 6. doi: 10.3390/machines5010006

Reed, E. (2020, Feb). *History of tesla: Timeline and facts.* TheStreet. Retrieved from https://www.thestreet.com/technology/history-of-tesla-15088992

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986, Oct). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. doi: 10.1038/323533a0

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2015). *Trust region policy optimization.* arXiv. Retrieved from https://arxiv.org/abs/1502.05477 doi: 10.48550/ARXIV.1502.05477

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). *High-dimensional*

continuous control using generalized advantage estimation. arXiv. Retrieved from https://arxiv.org/abs/1506.02438 doi: 10.48550/ARXIV.1506.02438

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). *Proximal policy optimization algorithms.* arXiv. Retrieved from https://arxiv.org/abs/1707.06347 doi: 10.48550/ARXIV.1707.06347

Self-driving cars: State of the art (2019). (2019, Feb). YouTube. Retrieved from https://www.youtube.com/watch?v=sRxaMDDMWQQt=4s

Sutton, R. S., McAllester, D., Singh, S., & Mansour, Y. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. In S. Solla, T. Leen, & K. Müller (Eds.), *Advances in neural information processing systems* (Vol. 12). MIT Press.

Swazinna, P., Udluft, S., Hein, D., & Runkler, T. (2022). *Comparing model-free and model-based algorithms for offline reinforcement learning.* arXiv. Retrieved from https://arxiv.org/abs/2201.05433 doi: 10.48550/ARXIV.2201.05433

Synopsys, S. (2022, Sep). *What is an autonomous car? – how self-driving cars work.* Retrieved from https://www.synopsys.com/automotive/what-is-autonomous-car.html

Tellez, R. (2022, Nov). *Ros for beginners: What is ros?* Retrieved from https://www.theconstructsim.com/what-is-ros/

Tesla roadster (first generation). (2022, Jan). Wikimedia Foundation. Retrieved from https://en.wikipedia.org/wiki/Tesla_Roadster_(first_generation)

Wang, Y., Chao, W.-L., Garg, D., Hariharan, B., Campbell, M., & Weinberger, K. Q. (2019, June). Pseudo-lidar from visual depth estimation: Bridging the gap in 3d object detection for autonomous driving. In *Proceedings of the ieee/cvf conference on computer vision and pattern recognition (cvpr).*

WHO. (2022, Jun). *Road traffic injuries.* World Health Organization. Retrieved from https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries