Creation and validation of a simulation environment for machine learning with drones

Master Thesis as part of the Master Biomechanical Design course ME51035

by

Veronika Bojtár

Student Number

5037654

Supervisors

Ole Luhmann / Réka Bérci-Hajnovics - DoBots
Ludo Stellingwerff - Almende
Dr. Wei Pan / Martijn Wisse - TU Delft

June, 2023.

Contents

1 Introduction **5**
1.1 Drone traffic rule generation . . . 6
1.2 Airspace control . . . 6
1.3 Traffic control . . . 7
1.4 Drone types . . . 9
1.5 Incremental complexity . . . 10
1.6 Machine learning . . . 10
1.7 Project goals . . . 11

2 Requirements **12**
2.1 Use case . . . 12
2.2 Requirements . . . 15

3 Theoretical background of Reinforcement Learning **17**
3.1 Reinforcement Learning . . . 17
3.2 Deep Q-learning . . . 19
3.3 Configuration . . . 21

4 Implementation of the simulation environment **26**
4.1 Benchmark test . . . 26
4.2 Main elements . . . 26
4.3 Implementation work . . . 28

5 Reinforcement Learning use-case **43**
5.1 Scenarios . . . 43
5.2 Challenges . . . 50
5.3 Configuration . . . 52

6 Results **56**
6.1 Reinforcement Learning use case . . . 56
6.2 Empty space . . . 56
6.3 Random goal . . . 58
6.4 Obstacle . . . 59
6.5 Two drones empty . . . 60
6.6 Window . . . 61
6.7 Two drones window . . . 62
6.8 Simulation environment . . . 63

7 Discussion **66**
7.1 Resulting behaviour . . . 66
7.2 Incremental complexity . . . 68
7.3 Simulation environment . . . 68

8 Conclusion **70**

Acknowledgements

I would like to thank my supervisors from Almende and DoBots who helped me with the realization of the project and gave me meaningful insight in discussions. I would also like to thank my supervisors from TU Delft for their guidance and advice. I also want to express my gratitude towards my friends and family for their encouragement and my fiancé for all the love and support he gave me.

Abstract

Simulation environments are useful for a wide range of applications and their functionalities continue to improve every year. The aim of this thesis project was to create a simulation environment with high levels of realism and assess its capabilities through the use case of generating distributed drone traffic rules.

This thesis project was executed at the company Almende in cooperation with its sub-company DoBots. The distributed drone traffic rule generation use case was a contribution to the ADACORSA project which Almende takes part in. The goals of the thesis were the following. A simulation environment incorporating elements of realism was created using the software tools Almende has experience with. The capabilities of this environment were tested through the said use case. According to company requirements, this process was realised using an incremental complexity approach, and it utilized machine learning techniques. It was expected that the incremental complexity approach would help speed up the learning process of more complex scenarios, while machine learning would provide the benefit of smart automation and impartiality to solutions. For this process, a model was also devised for creating such rules, incorporating the incremental complexity approach and a standard machine learning technique. Finally, the benefits of the incremental complexity approach were also assessed.

The four building blocks of the implemented simulation environment were Gazebo, PX4, ROS2 and the iris drone model. The code was built upon the OpenAI ROS software package, with extensive modifications. The main work done during the implementation consisted of multiple parts. The original software package was cleaned of unnecessary elements, the PX4 software and the iris drone model were integrated into the environment and the simulation speed was increased. The whole setup was migrated from ROS to ROS2. The use of multiple drones in the same environment was also solved. An implementation of the deep Q-learning algorithm, which was selected as the learning method, was added. Different scenarios were created, as well as some logging and visualization tools. Additionally, the system was set up to run in docker containers and was highly parameterized for future use.

Different elementary traffic scenarios were implemented in the simulation and then executed. Based on these the following points were determined. It was concluded that the created simulation environment was well-fitted for running scenarios for traffic rule generation. It was also suitable for executing the scenarios using the incremental complexity approach, although its usability also depended on how the neural networks of the scenarios were set up. The environment was also feasible to work with machine learning from an engineering viewpoint, however, it did not perform well for its scientific research. This was due to not having statistically well-supported results, which was caused by the long time it took to execute a scenario. The benefits of the incremental complexity approach were also examined. For the simple case tackled in this project, it was found that the incremental complexity approach provided no significant advantages with regard to learning speed.

1 Introduction

Simulation environments are used for a wide range of purposes. Their capabilities are continuously evolving, and they can be used for more and more applications. The main purpose of this thesis project was to create a simulation environment and assess its capabilities for a determined use case.

This thesis project was executed at the company Almende in cooperation with its sub-company DoBots. They were interested in testing the capabilities of a simulation environment with a pragmatically minimal simulation gap using their experience in the various software packages with which such a simulation platform can be created. A small enough simulation gap was important so that the developed products can be transferred to real life without too many difficulties. Their interest lay in assessing the use and benefits of such an environment through different use cases.

At these companies, A 3D simulator called Gazebo [1] is widely used for creating simulations. The company has opted for this software because of its many benefits. From the viewpoint of the current task, the most important is that it provides the physics background that is needed to obtain an environment that has high levels of realism. What is more, external plugins can also be added to the environment which makes it possible to adjust the level of realism in a wide range. Furthermore, Gazebo is open source and has a big community where help can be found for various problems one could encounter when using this software. Of the two main versions, Gazebo Classic was used in this work.

For this thesis project, the generation of distributed drone traffic rules was chosen to be the use case through which the simulation environment was tested. This was due to Almende taking part in the ADACORSA project [2]. The goal of the ADACORSA project is to provide necessary technologies to promote safe beyond visual line of sight drone usage, thus allowing them to be a useful part of the mobility mix. Testing a simulation environment through this use case had the additional benefit of not only gaining information about the capabilities of such a simulation environment but also providing a contribution to the ADACORSA project. This use case will be further detailed in the next sections of this chapter.

It was decided that besides the simulator engine, a realistic drone controller would also be used for the traffic rule use case. This provided the benefit that the drone traffic rules would be created by taking into account the realistic flight capabilities of drones. DoBots also has experience with such a drone controller, called PX4 [3]. It is a well-known industry standard software which is widely used on real-life drones. It also has a Software In the Loop (SITL) implementation, that makes it possible to use the package purely in simulation. This software works well together with Gazebo and is capable of handling sped-up time, which is important to increase the speed of the simulations and provide results faster.

The communication between Gazebo and PX4 is handled using Robot Operating System (ROS/ROS2) [4]. ROS is a collection of software libraries created for the use of robotics and it is widely used for robot control and communication between a robot's different elements.

As was introduced, the use case through which the simulation environment was tested was the generation of distributed drone traffic rules. This was carried out with the help of a standard machine-learning technique. As it will be detailed in Section 1.6, machine learning has many benefits, and this is why Almende was interested in examining how these machine-learning techniques could be used with the created simulation environment.

In the next sections of this chapter, the traffic rule generation use case will be detailed, and in

the last section, the goals of the project will be formulated.

1.1 Drone traffic rule generation

When a new technology surfaces, it is important to not only consider how it can be utilized but to also think about what safety measures should be introduced to avoid it posing any danger to society. New technologies can significantly improve the quality of everyday life, however, they can also lead to dangers. For example, the use of cars shortened travel time significantly, while also bringing new hazards to the roads.

Nowadays, drones are starting to gain ground in our everyday lives. Their purpose is not limited anymore to entertainment or military use. They could be utilized for example for last mile or medical delivery [5] [6], TV and other media broadcasts [7], traffic surveillance [8], and so on. They could be quite effective, especially in denser, urban environments, where they would have the chance to alleviate the load on the road infrastructure. However, having more drones flying around us also poses a negative effect on our safety. Equipping them with physical safety systems is not always enough, safety regulations are also needed. Some regulations are already in place, however, as will be explained in section 1.2, they lack the flexibility to allow many of the possible applications. Various projects aim to improve the existing solutions, and the use case tackled in this thesis project is a contribution to this work.

In the next section, the existing regulations for drone traffic will be presented. It will also be explained why they need improvements to enable widespread drone usage.

1.2 Airspace control

In this section first, the existing regulations on drone traffic are presented. It is important to gain knowledge about these to assess which areas are in need of improvement.

In the European Union, the European Union Aviation Safety Agency (EASA) [9] is responsible for airspace regulations, including drone control. Besides this organization, every country in the EU can have its own institution, for example in the Netherlands, the Ministry of Human Environment and Transport Inspectorate (ILT) [10] is responsible. The regularization introduced by the EASA should be adhered to by every country in the union, but each can add its own rules inside its territory.

In 2020, the EASA introduced a new set of stricter rules regarding drone usage. These rules distinguish three categories for drones, namely open, specific, and certified [11]. The categories are defined based on the risk that comes with flying the aircraft in question. If flying the drone poses high risks, it belongs to the certified category. To use an aircraft like this, both the pilot and the drone have to have certifications. Drones in the specific category come with less risk when flying them, but they still pose some danger. To use these, the pilot needs to submit a risk assessment of their flight and obtain authorization for it. The open category includes drones with the least risk of causing damage, and no prior authorization is needed to use them.

Most countries have their own additional rules for drone traffic control. It is customary that their airspace is regulated by specifying no-fly zones and other types of restricted areas. For example, in the Netherlands, it is not allowed to fly in the airspace of airports without authorization.

In addition to the regulations described above, in the Netherlands, if drones are to be used for commercial purposes, the user has to acquire certain licenses. Moreover, the pilot of the drone has to have a pilot's licence [12]. Drones for recreational use have no such requirements, however, they have to remain below 120m of altitude, are not allowed to fly over groups of people or fly at night [13]. These all serve the purpose of making drone flight safer. The maximization of altitude and the ban on flying at night is important so there is less chance that the drone gets lost or becomes uncontrolled. Also, it is less likely that they get in the way of other aircraft operating at a few hundred meters of altitude. The restriction about flying above people is also self-explanatory: if there is a mechanical or software error with the drone and it crashes to the ground there is less chance of it colliding with people.

While these regulations serve our safety, they are hindering the introduction of drones for a wide range of applications, especially in denser, urban areas. For example, in the Netherlands, a relatively large area of airspace is strictly regulated due to its proximity to airports, which makes it impossible for drones to be used in nearby cities. The rules are also strict regarding the number of aircraft allowed in an area. If there are more drones in a certain space, they, in most cases, fall into the strictest, certified category. The autonomy of drones is also an issue. Due to the defensive nature of the rules, the pilot is always held responsible for the actions of the drone. This, naturally, assumes there is always a pilot, who can at all times take control of the drone, even in the case of beyond visual line-of-sight applications and even if the drone is intended to be autonomous. Although the rules are generally useful to avoid accidents, they are not flexible enough for many applications. For example in the case of logistics, having a fleet of autonomous drones making deliveries is impossible with the current regulations.

To be able to contribute to the improvement of these regulations, it is useful to know the existing projects in this area. There is constant work to improve the existing drone regulations and create meaningful additions to these. There is also significant research effort being made on creating technologies and solutions for safely integrating drones into our everyday lives. According to [14] "U-space is a service-orientated concept to provide air traffic management (ATM) to drones". The aim of this concept is to promote the safe integration of drones into air traffic and make them available to many different applications for which they are not yet widely used. The goal of the SESAR JU project [15] is to promote the applicability of drones in Europe. From their projects, The Metropolis 2 aims to develop solutions that would allow the safe usage of drones in dense urban environments [16]. ADACORSA [2] is also a European project. As was described earlier, its goal is to provide necessary technologies to promote safe beyond visual line of sight drone usage, thus allowing them to be a useful part of the mobility mix.

Other projects tackle creating software environments for development [17], providing conflict management solutions [18], [19] or tackling the problem of optimal 3D path planning [20]

Besides gaining information about these projects, it is also beneficial to understand how traffic control is built up in order to be able to create an effective addition to it. This will be described in the next section.

1.3 Traffic control

In general, traffic control can be realized in two main ways; by either using a centralized or a distributed approach. The two are illustrated in Figure 1.

The centralized way of implementing control means that there is one central controlling system that communicates with all the elements of the traffic in its area. It can issue commands to them and provide the necessary information required for safe and effective participation in traffic. This

solution has the benefits of having an overview of the whole traffic situation and providing solutions that benefit the whole system. The biggest drawback of this approach is the possibility of single points of failure; if there is a problem with the central controller the whole system fails. A good example of this type of traffic control is how airspace is managed, with control towers regulating the traffic.

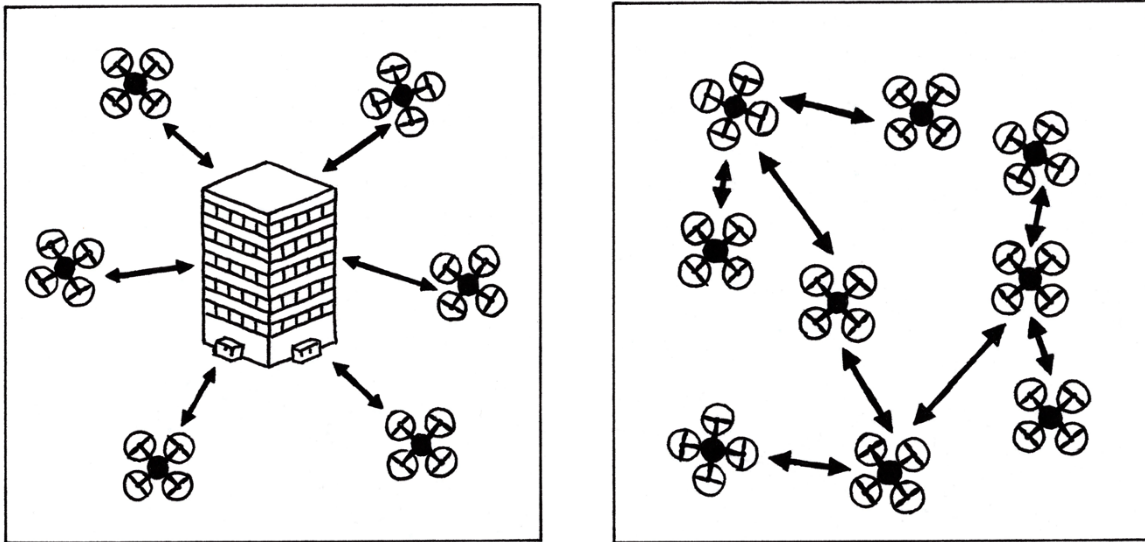


Figure 1: Centralized (left) and distributed (right) control

The distributed control type means that the elements of traffic solve their disputes with each other locally, without relying on a central information source. This is beneficial because it avoids the problems that come with single points of failure, and it also might be faster if the conflicts are resolved locally. This solution might even be cheaper than the centralized one since there is no need to set up reliable central systems. However, its drawbacks can be found in not seeing the whole picture, which may result in creating suboptimal solutions to traffic problems. Additionally, traffic participants are required to use either a common ruleset or a shared method of communication to translate between their different rulesets. If they fail to communicate successfully they will not be able to resolve conflicts between them. A good example of this type of control is how many aspects of road traffic are regulated.

Distributed solutions are especially important in urban environments because of the unpredictability that comes with a dense space. Suddenly appearing unforeseen factors may surprise a centralized system, and it is possible that it cannot provide a solution in time to avoid the accident. Overloading might also be a significant risk. The system can become overburdened if there are too many unpredictable conflicts occurring at once and they all need attention at the same time. It is also possible that some or in extreme cases all of the participants lose connectivity to the central controller. Having access to distributed solutions would help ease these problems.

The most flexible solutions are probably the ones which combine the two control methods. Distribution helps ease the load on centralized control and mitigates the effects of single points of failure, while a central system provides information on the wider traffic situation, and helps avoid predictable dangers. A good example of the combination of the two methods is how the entirety of road traffic is regulated. There are rules for traffic participants to solve their disputes locally,

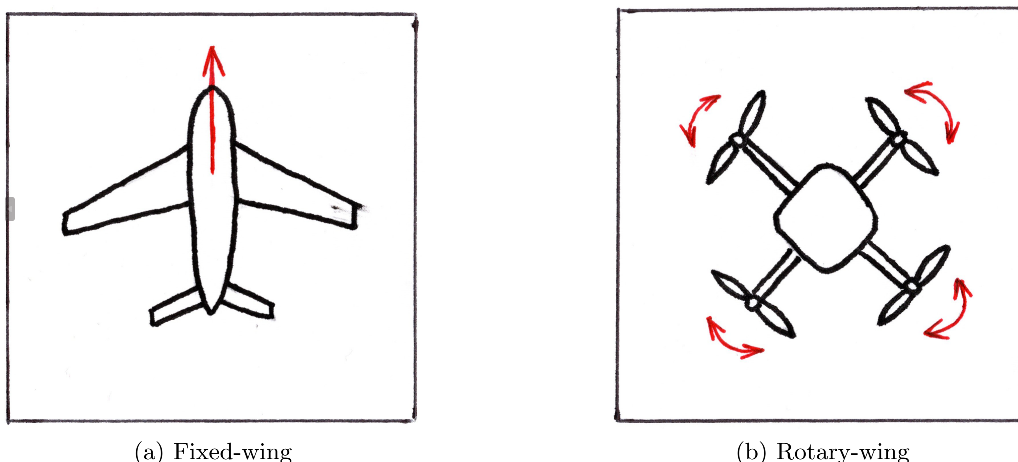
and additionally, traffic lights present centralized control sources. Combining the advantages of centralized and distributed solutions would be also beneficial for drone traffic control. This could be realized, for example, by having a set of distributed traffic rules along with centralized control towers.

A lot of the work described previously is concerned with the control of drones using a centralized method, and less work is being done on the distributed approach. This would mean for example obtaining a ruleset that could be used while the drones are flying around, meeting obstacles both static, like cranes or lamp posts and dynamic, like other drones. This is why the use case of this project was to investigate whether a ruleset for distributed drone traffic could be created with the use of the simulation environment that was described above.

To be able to create simulations it is not only important to know what type of traffic control to work with, but also the capabilities of the drones that will be simulated. The two main types of drones will be described next, along with their advantages and drawbacks, and it will be determined which one is more applicable to this project.

1.4 Drone types

Drones are usually classified into two categories; they can be either fixed-wing or rotary-wing based on the mechanism which keeps them in the air [21]. The differences between the aerodynamical mechanisms of their flight create the differences between their capabilities, advantages and disadvantages. These are important to keep in mind when selecting drone models for traffic rule generation. The two types are depicted in Figure 2. In the case of fixed-wing types, the forward velocity of the aircraft creates an airflow on the wings. This in turn creates a pressure difference between the upper and lower part of the wings which results partly in an upwards force called lift. This lift force is used to get the aircraft into and keep it in the air. Meanwhile in the case of rotary wing types as the rotor blades are rotated, this rotation creates an airflow on the blades which results in the necessary pressure difference and lift force [22].



(a) Fixed-wing (b) Rotary-wing

Figure 2: Drone types

As mentioned, due to the differences between their operation, their capabilities, up, and downsides are also different. Fixed-wing aircraft require some type of runway to take off, which means

they need some amount of open space around them. Contrary to this, rotary types can take off on the spot. Fixed-wing types can generally stay in the air much longer. They are capable of gliding, and in certain conditions use thermals [23] to even increase their altitude. This however comes with the drawback of needing constant forward velocity to stay in the air, thus they are incapable of hovering. In comparison, rotary-wing aircraft constantly use energy to keep themselves in the air, but they are more manoeuvrable and capable of hovering. Due to the above listed, fixed-wing aircraft are more useful in spacious rural areas and for longer-range flights, such as reconnaissance or deliveries between far-away targets. On the other hand, rotary-wing drones perform better in dense urban areas where manoeuvrability is key and the flight time is not as significant, such as last-mile delivery to homes, or short TV broadcasts of events.

As described earlier, the use case of this project aimed to tackle distributed rules, which are especially important in dense, urban environments. This is why for the purposes of this project a rotary-wing drone model was chosen. Having presented the control type for which the rules were generated, as well as the drone type that was used for the simulations, the next two sections describe additional information about how distributed drone traffic generation was carried out.

1.5 Incremental complexity

In addition to what was described above, Almende was also interested in examining the benefits of using an incremental complexity approach for the use case of this project. In this approach, the rules which were developed in less complex scenarios were utilized to promote the process of learning in more complex ones. This method was expected to make the rule-generation process faster and more straightforward. It was inspired by Almende's philosophy of approaching problems. Their vision is that complexity and completeness of solutions emerge from the interaction of simpler elements. This philosophy is applied in many aspects of the company's work, and it was expected that it would also provide an advantage for the purposes of drone traffic rule creation.

Throughout the many scenarios executed in this approach, the increase in complexity is with regard to the simulations, not the rules themselves. Having more complete rules does not necessarily mean that they are more complex. The increase in complexity, however, is required for the scenarios that are executed, because as they incorporate more elements of reality into their worlds, they inherently become more complex.

1.6 Machine learning

As was introduced at the beginning of this chapter, the distributed drone traffic rule generation process was carried out with the help of machine learning. The use of machine learning had been examined to determine its benefits for the chosen use case. First, machine learning algorithms can be useful for developing solutions that humans do not foresee. Moreover, these methods can also be capable of mitigating human preconceptions. A negative bias towards certain solutions might still be introduced in the system by its users, however, with proper care it was expected not to pose too much danger in this case.

Another benefit identified of using machine learning techniques was to help the process of testing the developed solutions of the various scenarios. When using an incremental complexity approach, a large number of simulations are needed in order to check the viability of the created rules. This repeated work of testing can be automated by using various different computer algorithms. In the case of this use case, the problem arose that as the scenarios changed the

system needed to be adapted to these changing environments. However, with the help of machine learning the same solution could be used to test the results, even though the environment was changing.

As described earlier, the simulation environment developed in this project had high levels of realism, in order for the developed solutions to be straightforward to apply in real life. Considering the benefits of machine learning, it was a point of interest of Almende and DoBots to examine how such algorithms can work with the realistic simulation environment created within this project with the use of software packages that these companies have experience with. Due to this interest, the created simulation environment was tested through a use case that incorporates machine learning. Nevertheless, as the main focus of the project was to create and test the simulation environment itself, the machine learning algorithm implemented was chosen to be a standard, well-known one.

So far in this chapter, the company interests were described as well as the need for a set of distributed drone traffic rules applicable to dense urban environments. Background information was also presented about the expected method to be used for obtaining these. The next and final section in this chapter presents the formulated goals of the project.

1.7 Project goals

In the course of this project first, a simulation environment with a pragmatically minimal simulation gap was implemented using the software packages that Almende and DoBots have experience with. The capabilities of this simulation environment were then tested through the use case of distributed drone traffic rule generation. For the process of creating rules, a model was devised, which utilized the incremental complexity approach of Almende. Moreover, the model utilized machine learning to realize rule generation. Finally, there was also an investigation regarding the incremental complexity approach itself, to see whether it would help more complicated learning processes.

The created simulation environment and the scenarios implemented in it had the potential to be utilized in future projects at Almende and DoBots. This is why an additional requirement of the work was that the simulation environment would be modular and integrable into the companies' other works.

In this chapter, the reasons behind the project were described and its goals were presented. In the next chapter the model devised for the traffic rule generation use case will be described and the requirements for the simulation environment will be formulated based on this model.

2 Requirements

As discussed in Chapter 1, the use case through which the simulation environment was examined was the generation of distributed drone traffic rules. In order to obtain such rules a well-formulated method was needed. According to the goals of this project, this method incorporated one of the main philosophies of Almende, the incremental complexity approach. Moreover, it also utilized a standard machine learning algorithm. This chapter describes the theoretical model devised for this purpose, presents two models of its practical implementation and explains which one was chosen for further testing. In the last section, the requirements towards the simulation environment will be formulated.

2.1 Use case

The theoretical model which was devised for obtaining a set of distributed drone traffic rules is shown in Figure 3 and will be explained next. As already described, this model uses simulations with an incremental complexity approach. The incremental complexity in this setup manifests by a mechanism where more complete rules are obtained using information from the results of previous, simpler scenarios. As this model is fairly high-level it does not yet contain a description of machine learning.

The process is as follows. Multiple scenarios (shown in the top left corner of Figure 3) are created, representing different traffic problems to create a solution to. These scenarios are ordered by the complexity of the traffic problem they represent. For example, a situation where a single drone tries to make a delivery in an empty space would be earlier on the list than a situation where the same single drone tries to make a delivery in a space filled with static obstacles. The scenario next in line is selected starting from the easiest and moving towards the more complicated setups. Its parameters are configured, and if required, additional human inputs can be defined to steer the learning process. This is also where the incremental complexity approach comes into play. After the simplest scenarios have been executed, the more complex ones will try to utilize what was learned in the earlier ones. The specific implementation of this solution will be discussed after the practical models are presented. When the configuration is decided, the simulation process starts, and possible strategies are generated for the problem presented in the simulated scenario.

After a good strategy is developed for the scenario, it is extracted in some way and formulated into a traffic rule. This is what the *Extraction layer* represents in Figure 3. Then, this rule is tested to see whether it is feasible also in the earlier, less complex scenarios whose results this one used as building blocks. For example, if it is first learned how two drones should avoid each other in an empty space, then it is learned how these two drones can avoid each other in a space full of obstacles, it is important to test how well the more complete rule performs in the previous, empty scenario. If the tests produce positive results, meaning that applying the more complete rule in the simpler environment is at least as safe as the less complete rule, a new scenario can be set up and run to obtain the next rule. If testing does not provide the required results the scenario is run again with some modifications. These modifications can range from smaller to more extensive ones. For example, it is possible to modify the value of only a few parameters (for more information, see Chapters 3 and 5), but in some cases modifying the environment world itself can become necessary.

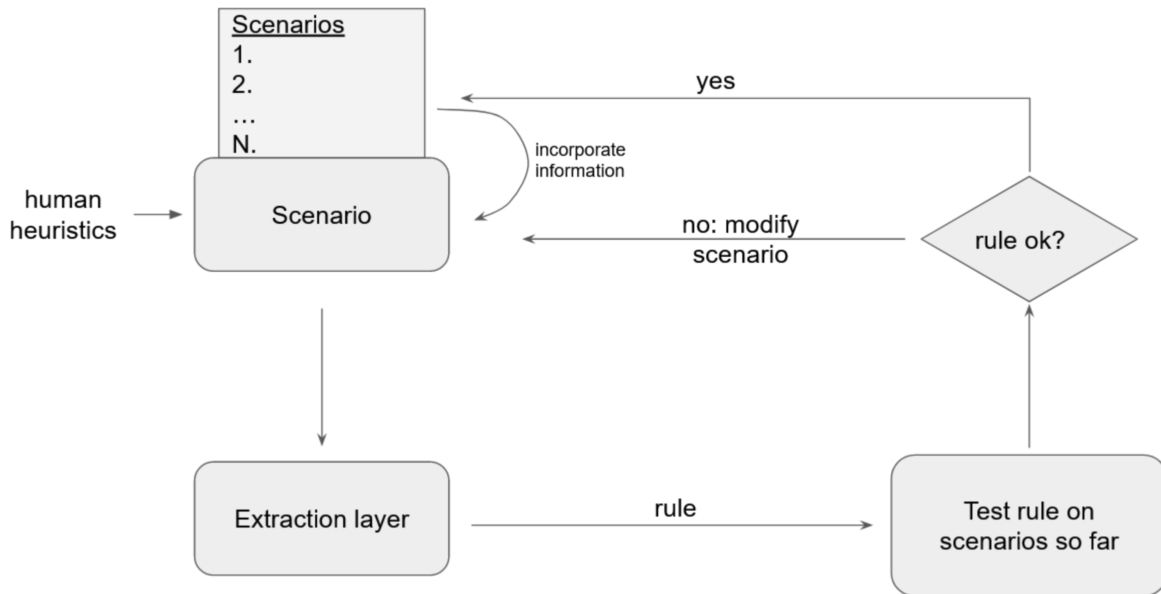


Figure 3: Theoretical model of generating traffic rules

There are many possible different ways to implement the theoretical model of drone traffic rule generation. For this project, two practical models were considered. These are shown in Figure 4 and Figure 5. Both use machine learning in the process of finding good behaviours for the various traffic scenarios that are presented. As will also be explained later, model 2 with the more standard machine learning method was selected. While model 1 presents quite an interesting machine learning challenge, it would shift the focus from the simulation environment itself, and, as explained previously, the goal of this project was to create and test a simulation environment, not to push the state of the art in machine learning.

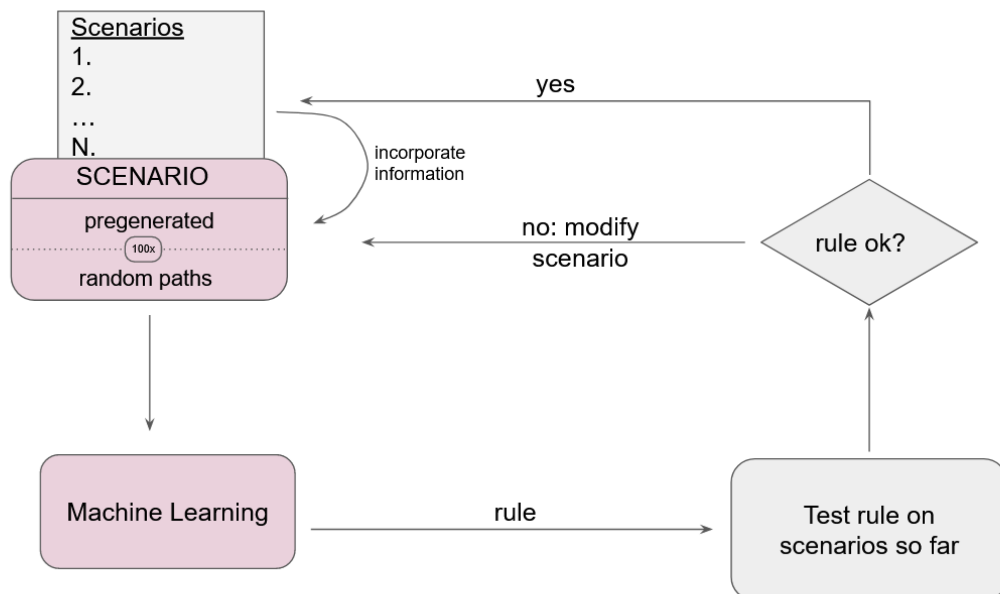


Figure 4: Practical model 1 of generating traffic rules

Figure 4 depicts model 1 for obtaining rules. In this method, the machine learning algorithm implements the *Extraction layer* of the theoretical approach. In the process of using this model, first, many different behavioural paths and strategies, both safe and unsafe, are generated at

the same time. This represents the *Scenario* layer. These are then used as input to a complex machine-learning algorithm. The idea for this model is that first a classifier-like structure would be obtained. This will then be used for the system to learn to generate safe behaviours on its own, similar to how an image generator algorithm would operate.

Even though this model was not used, it is presented here because in future work it would be quite interesting to test its capabilities in the created simulation environment. Moreover, as it will be explained in Chapter 4, the simulation environment was implemented in a highly modular way, therefore adding machine learning to serve as the extraction layer would not result in large modifications of other parts.

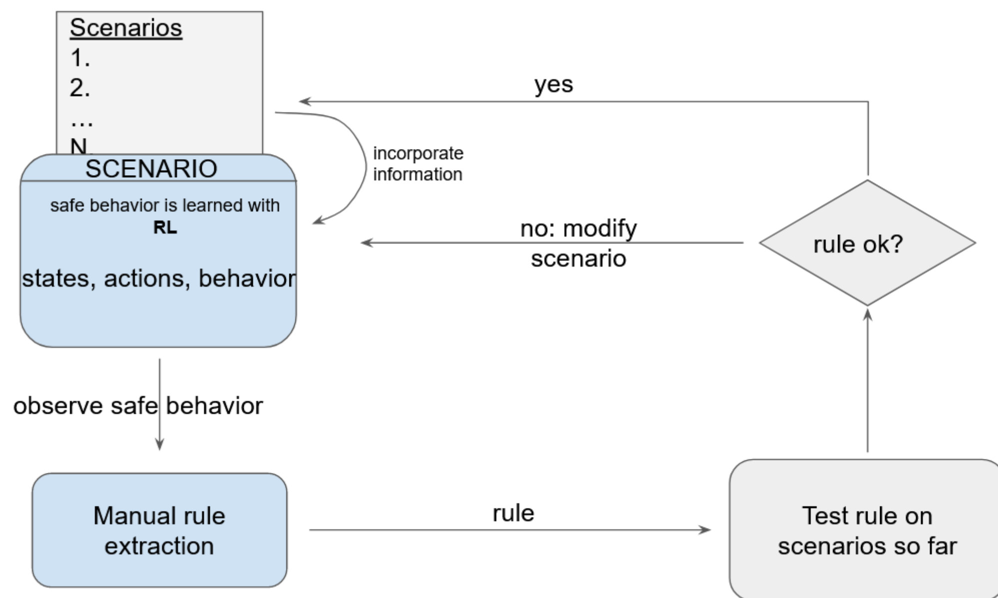


Figure 5: Practical model 2 of generating traffic rules

In the case of model 2, shown in Figure 5, machine learning plays a role in the *Scenario* layer of the theoretical approach. In this implementation, safe behaviours for the various traffic scenarios are generated by the algorithm itself, using reinforcement learning. Reinforcement learning was chosen as the machine-learning algorithm because it is capable of devising appropriate behaviour solutions by itself. The human input appearing in the theoretical model is also included in this layer. This consists of how the various elements of the simulation are defined. In the case of this model, the extraction layer is implemented by hand. This means that the generated good behaviours are observed by an outside party and formulated into rules manually.

As was already mentioned, for the purpose of this project model 2 was chosen. It presented less risk in overrunning the project timeline, while this was harder to assess in the case of model 1. This was due to the nature of the machine learning algorithm. As was already introduced earlier, the machine learning algorithm was decided to be a standard one, with sufficient background information available so as not to take too much focus away from creating the simulation environment. The solution presented in the case of model 1 included more unknowns than in the case of model 2 with regard to both implementation and testing.

While assessing the various capabilities of the simulation environment, the viability of the incremental complexity approach was also assessed. There are various ways how this can be realized. For example, the learned rules of a simpler scenario could be directly implemented into the drone

controller before starting the more complex scenario. However, in this work, a more direct approach was examined, which was starting the more complex scenario with the trained agent of the simpler one.

2.2 Requirements

For building the simulation environment as well as for its verification and validation through the determined use case, the requirements listed in Table 1 were set. The implementation of the environment was carried out by taking these requirements into consideration, and they were also considered in the assessment of the created platform.

As the simulation environment only includes core functionalities, a full MoSCoW analysis was executed for this project. As the reader can see, the requirements formulated in Table 1 are relatively high-level ones. The reason behind this is that for the purposes of the project, the validation of the created environment was chosen to be a more straightforward qualitative, not a more detail-oriented quantitative one. At the end of this report, these requirements will be used to verify and validate the platform.

Use case aspect		**Requirement**
Incremental complexity	1	Adding new scenarios does not result in major modifications in other parts of the environment
	2	Scenarios can start their training by using the trained agent of previous scenarios
Reinforcement learning algorithm	3	Capability to quickly and efficiently reset the simulated world after every episode
	4	Robustness
	5	Scalability
	6	Simulation speed should be significantly faster than realtime
	7	Initial state of the drone can be varied within the scenario
Traffic rule generation	8	Support multiple drones in the same simulation
	9	Support for static and dynamic obstacles
	10	Rules are translatable to real life

Table 1: Requirements

Two requirements were formed with regard to the incremental complexity approach:

1. The first one describes that adding new scenarios to the environment should not result in major modifications of its other components. The reason for this was that since large numbers of different scenarios would be used for the process, the environment should be able to handle integrating and exchanging them without the need for rewriting significant parts of it. Otherwise, too much overhead would be created.

2. The second requirement with regard to the incremental complexity approach describes that scenarios should be able to utilize the previously trained agent of earlier scenarios. This was necessary to realize the core idea behind the incremental complexity approach, which was using information from previous scenarios. Moreover, this requirement is specific to the chosen implementation of the incremental complexity approach in this project, where a new scenario was started with the trained agent of the previous one.

Five requirements were formed with regard to using a reinforcement learning algorithm in this project:

3. As it will be described in Chapter 3, the process required that the simulated world be reset at the beginning of every episode. This requirement describes this necessity.

4. Since many episodes and scenarios would be executed, it was important for the simulation to be robust. This was needed so that after a long time of running simulations results were not lost.

5. This requirement describes that the simulation should also be scalable. This was necessary so that a great many executions can be carried out.

6. Since many simulation runs would be executed, it was important, that the speed of the simulation be significantly faster than realtime. If it were not the case, results could not be obtained fast enough, and not enough simulations could be run to examine the effects of different settings in detail.

7. The last requirement in this category describes that the initial state of the drone can be varied within the scenario. This was important so that the scenarios could be executed with different initial conditions.

Finally, three requirements were formed with regard to the traffic rule generation use case:

8. According to this requirement, the simulation environment should support multiple drones in the same simulation. This was needed because in traffic, in general, more than one participant should be anticipated.

9. The environment should also support both static and dynamic obstacles. This was necessary for describing realistic traffic scenarios, where buildings or trees for example were static obstacles while flying birds were dynamic ones.

10. The last requirement presents the need that the generated rules should be translatable to real life. This was needed so that the generated drone traffic rules could be utilized by human operators.

This chapter provided a description of the theoretical model of obtaining drone traffic rules and the two practical models considered for this project. From these, model 1 was chosen. Based on this model the requirements for the simulation environment were formulated. The next chapter will describe the theory of reinforcement learning and the specific reinforcement learning method chosen for this project.

3 Theoretical background of Reinforcement Learning

As described in the previous chapter, the chosen machine learning method for rule generation was reinforcement learning. In this chapter, a theoretical background will be provided about this algorithm. This description establishes a background for understanding the practical implementation of the simulation environment, described in Chapter 4, and the parameters chosen for the different scenarios, which are detailed in Chapter 5.

In Section 3.1 the general reinforcement learning technique will be described, while in Section 3.2 the specific reinforcement learning method chosen for this project will be detailed. Finally, in Section 3.3 a description of the most important parameters of this method will be presented, providing information crucial for creating a good learning process.

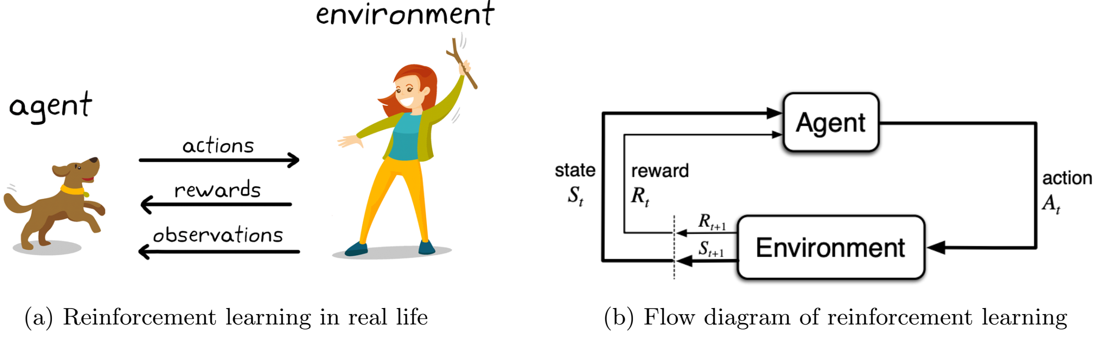
3.1 Reinforcement Learning

For understanding the created simulation environment, including its use and its challenges, it is important to describe the main algorithm behind it. Reinforcement learning is a machine learning technique in which an agent learns a good behaviour from interacting with its environment through an iterative process [24]. The agent gains feedback on its behaviour by obtaining rewards and penalties based on its interactions. Figure 6[1] shows this process. This method can be explained through the example of how a dog learns to play fetch. At first, the dog has no idea what to do when its owner throws away a stick. However, since it is outside where it can do many interesting things, it will try out various actions. It could for example start running around, or go after the stick and smell it. The dog gets encouragement when it goes near the stick, and it receives treats when tries to bring it back. However, it can also be told off when it does something its owner considers bad, for example biting somebody. Eventually, the dog will know what behaviour results in the tastiest treats, and it will learn to play fetch. Reinforcement learning works the same way. In this example, the dog is the agent, its owner and the park are the environment. The treats and telling-offs are the rewards and penalties. The iterative nature of the process can be found in how the dog can try again multiple times. If the owner decides that they should start over, they call back the dog and collect the stick. By doing this they effectively reset the state of the environment to its initial configuration, and by throwing the stick again, a new iteration starts.

In the case of this project the drone is the agent, the scenario world is the environment and the rewards and penalties can be defined based on how safely the drone behaves while also keeping goal achievement in mind. From this point, in this report "drone" and "agent" will be used as equivalents.

Machine learning methods are usually classified into supervised and unsupervised types, however, reinforcement learning is neither [24]. It can be argued that it has some amount of supervision since the reward and penalty structure is defined by the outside user. However, according to [27] supervised machine learning is about a system learning from a dataset where an outside party has shown inputs and their corresponding outputs. The goal of the system is to learn from this dataset and learn to generalize its findings to data so far unseen. It is easy to see that while reinforcement learning uses a reward system, its method is learning from interactions with its environment and exploring, not using a prepared dataset. Reinforcement learning is also not unsupervised, and the reasoning is along the same path as for the supervised. The goal of unsupervised machine learning is to find a structure in a dataset where no target output has been specified [28]. In comparison, the goal of reinforcement learning is to develop a behaviour

[1] Figure 6a is from [25] and Figure 6b is from [26]



(a) Reinforcement learning in real life (b) Flow diagram of reinforcement learning

Figure 6: Reinforcement learning

appropriate to a presented problem.

The reinforcement learning process is often modelled as a Markov decision process [29], [30], which consists of the following elements:

- X - state space
- U - action space
- P - state transition probability
- R - reward

The Markov decision process is then represented by the following touple: (X, U, P, R). The state space of the drone includes all the possible states it can reach, while the action space includes all the possible actions it can carry out. The state transition probability determines the probability of the drone getting from one state to another based on an action, while the reward is the feedback the agent obtains from its environment. In further descriptions, the state transition function $f(x, u) = x'$, which determines the new state of the agent based on the previous state and the action it took from that state will be utilized. Additionally, the reward function $\rho(x, u) = r$, will also be used, which determines the obtained reward based on this transition.

Important elements furthermore are the discount factor γ and the policy π. The discount factor is responsible for lowering the value of future rewards with regard to the present. This means that if one action is worth r_{now} at moment t_{now}, doing it one step in the future would be worth γr_{now} at moment t_{now}. If performed N steps in the future the value of the action would be worth $\gamma^N \times r_{now}$ projecting back to time t_{now}. This parameter can be seen as a see-ahead ability of the agent [30].

Equation 1 describes the sum of discounted future rewards starting from state x_0. This is the value of a series of actions taken by the agent with regard to the present.

$$R^{\pi}(x_0) = \sum_{i=0}^{\infty} \gamma^i r_{i+1} \tag{1}$$

The policy π consists of the actions the agent is set to follow. We talk about optimal policy π^* when by following said policy the agent performs a good behaviour, optimizing R^{π} and thus obtaining a high (or highest) reward. In simple terms, the goal of the reinforcement learning agent is to find this optimal policy.

Equations 2 and 3 show two ways how the worth of a policy can be measured. The Value function is shown in Equation 2, while the Q function is shown in Equation 3. These both consider the obtained reward if the agent follows a policy π. In Equation 3 the state $x_1 = f(x_0, u_0)$ is the result of taking action u_0 from state x_0. The difference between the two ways is that while the value function only considers the states of the agent to evaluate how good a policy is, the Q function also considers the actions the agent takes from its states, incorporating them into the policy evaluation.

$$V^{\pi}(x_o) = R^{\pi}(x_o) \tag{2}$$

$$Q^{\pi}(x_0, u_0) = \rho(x_0, u_0) + \gamma R^{\pi}(x_1) \tag{3}$$

The Bellman equation is often used to describe the cumulative value and Q-function in a recursive manner. These are shown in Equation 4 and in Equation 5, respectively. Here, x' is the new state the drone arrived at by doing action u in state x. Various reinforcement learning methods use these equations to obtain a successful learning process.

$$V^{\pi}(x) = r(x) + \gamma V^{\pi}(x') \tag{4}$$

$$Q^{\pi}(x, u) = \rho(x, u) + \gamma Q^{\pi}(x', \pi(x')) \tag{5}$$

A common reinforcement learning method is Q-learning, which can be used if the state and action space of the agent are finite. It makes use of the Bellman equation for the Q-function. Equation 5 can be rewritten using $Q^{\pi}(x', \pi(x')) = \max_{u'}(Q(x', u'))$ and $\rho(x, u) = r$. Turning this into an iterative update we obtain:

$$Q(x, u) \leftarrow r + \gamma \max_{u'}(Q(x', u')) \tag{6}$$

This equation means that the Q value of the agent taking the action u from state x is calculated by the immediate reward obtained due to that action, and the discounted expected future reward of taking the action that is considered best from x'.

In the case of Q-learning the associated value of taking an action from a certain state is stored in a multi-dimensional table. This table is updated every step based on the observations of the agent. Many episodes are carried out throughout the learning process, and in each episode, the agent is allowed to execute multiple steps. In one step, the agent, starting from a certain state, performs an action. Then the new state, resulting from this action, is observed and the value of the action is calculated. The corresponding value stored in the Q-table is updated using the information obtained in this process. Returning back to the example of the dog learning to play fetch, each episode is one iteration of its learning. Every iteration starts with the owner throwing the stick. Then, each step is some action the dog chooses to do, for example taking a step forward or smelling the grass. Each episode starts with the exact same conditions, thus allowing the dog to utilize what it learned in the previous episodes.

3.2 Deep Q-learning

If the state space is continuous the simple Q-table cannot be used in the solution of the reinforcement learning problem. This is the case for example in this project. To be able to provide an adequate simulation environment for drone traffic rules, the position of the drones as well as their sensor data are not quantized. A solution for this problem is to replace the Q-table with a neural network, which purpose is to estimate the Q function of the system. This algorithm is called

deep Q-learning or deep Q-network (DQN) method [31], and it was chosen as the reinforcement learning technique for this project.

This method optimizes the cost function J, shown in Equation 7, while training the neural network.

$$J = \mathbb{E}\{\big\|r + \gamma \max_{u'}(Q(x', u')) - Q(x, u)\big\|\} \tag{7}$$

The prediction of the neural network $Q(x, u)$ is compared to the target value $r + \gamma \max_{u'}(Q(x', u'))$. This target is calculated by sampling the environment for the current reward r that is obtained by doing action x at state u, and calculating the discounted future reward. This calculation is done by utilizing the neural network in training to predict the reward which is obtained by taking the best action from the new state. Naturally, what the system considers best is not always the real best one, especially early in the learning process. As the learning progresses, the likelihood of this action being the real best will increase.

Having described the general method of deep Q-learning, there are two important issues that need to be addressed. The first one is regarding how the experiences of the agent - in other words, the input data of the neural network - are handled. If the experiences of the agent are immediately discarded after they are utilized, the problem of catastrophic forgetting can arise. This means that the neural network will always try to converge on the last pieces of information it received and forgets what it learned from the older experiences. To tackle this problem an experience replay can be used. This experience replay acts as a buffer, where the experiences of the agent are stored. During the learning process, smaller batches of this data are randomly selected and the network is trained using these. This way past experiences are not forgotten.

The other complication that comes with using neural networks for reinforcement learning is caused by the target value being calculated by the neural network that is trained. This network is continuously changing as it learns, and this makes the learning process unstable. To solve this problem, a separate target network can be introduced, which is updated less frequently. The prediction of the neural network is then compared to the expected output calculated using the current reward and the expected future reward obtained using this target network. In order to still be able to progress in the learning, the weights of this network are periodically aligned with the weights of the continuously learning Q network.

With this solution, the cost function to be optimized can be formulated as shown in Equation 8. $\tilde{Q}$ represents the target network and Q is the continuously learning network that is used for exploitation actions.

$$J = \mathbb{E}\{\big\|r + \gamma \max_{u'}(\tilde{Q}(x', u')) - Q(x, u)\big\|\} \tag{8}$$

An additional problem is the agent's tendency to overestimate the Q-value. As a solution for this, the double learning method can be used [32]. In this method, two neural networks are learning two Q-function approximations. One of these networks will be used to select the action which maximizes the reward, and the other one to estimate that value, see Equation 9 and Equation 10.

$$J = \mathbb{E}\{\big\|r + \gamma Q_2(x', argmax_{u'}(Q_1(x', u'))) - Q_1(x, u)\big\|\} \tag{9}$$

$$J = \mathbb{E}\{\big\|r + \gamma Q_1(x', argmax_{u'}(Q_2(x', u'))) - Q_2(x, u)\big\|\} \tag{10}$$

When choosing the parameters of the neural network it is necessary to know whether the problem is classification or regression. For example, different loss functions are recommended for different types. For an outside observer, DQN can be seen as a classification problem, since it determines the best action that can be taken from a certain state. However, DQN by nature is a regression problem with as many outputs as possible actions. This is due to the fact that it provides a continuous output - the expected reward by doing a certain action - based on a set of inputs, not whether the action is to be chosen or not.

3.3 Configuration

There are many parameters that are important for a successful learning process, and it was necessary to have good knowledge about them for the process of creating the simulation environment of this thesis project. Gaining insight into these parameters, their values and how they influence the learning is also important for effectively configuring learning simulations both in this project and in future works. This section aims to provide a theoretical overview of the most significant parameters that were tackled in this project. In this chapter, it will be described what effect they have on the learning process, while the process of finding appropriate values for them as well as their final settings will be presented later in Chapters 5 and 6. As the reader will see in these chapters, different scenarios might require different settings to perform well.

In this project, two categories were determined for the parameters of the DQN process. Some are concerning the reinforcement learning method itself, and some are parameters concerning the training of the neural network. The most important reinforcement learning-specific ones identified are the exploration-exploitation rate, the number of steps the agent is allowed to take in an episode, the discount factor, and how the rewards and penalties are set. Important parameters determined regarding the training of the neural network are the learning rate, the loss function, the optimizer, the network structure, the activation function, the target alignment period, and the batch size. Additionally, it is also important to decide what information is considered to be part of the state of the agent since this will be the data the neural network learns from.

3.3.1 Exploration-exploitation rate

As explained, the learning process consists of many episodes, all starting with the same configuration, and every episode consists of multiple steps - consecutive actions the drone can take. In each step, the agent can either do a random action or an action that it considers the best based on what it has learned so far. The chance of doing a random action is determined by the exploration-exploitation rate. Exploration means that the agent carries out random actions to obtain information about its environment. Exploitation means following the best action the agent is aware of from its current state. The exploration-exploitation tradeoff is the phenomenon that if the agent is mainly concerned with exploring its environment it will not utilize what it has learned, while if it is mainly concerned about using what it has learned so far it will not gain new useful information about its environment. If the rate of exploration is too low, the system can get stuck in a suboptimal behaviour, thinking it is the best it can do, while with more exploration it could have developed a better strategy. Usually, this problem is tackled by setting the exploration rate high at the beginning of the learning process and then gradually lowering it throughout the episodes.

3.3.2 Number of steps per episode

The allowed number of steps in an episode can be a deciding factor in whether the drone learns a good behaviour or not. The drone needs to be able to discover its environment and obtain infor-

mation about the value of various actions from the different states. However, allowing too many steps can slow down the real-time speed of the learning significantly. Besides the exploration capability, what also needs to be kept in mind is if the allowed number of steps is not enough the drone might never even reach its goal, and it won't be able to utilize this information in its learning. For example, if the goal of the drone is to reach a certain position, which is 15 steps from its starting position, but only 10 steps are allowed in an episode, it will never be able to solve the problem.

3.3.3 Discount factor

In the equations describing how the value of a policy is evaluated, it was shown that the value of the future rewards is discounted for the present. The parameter used for this is called the discount factor. In effect, this determines how important the future rewards are compared to the reward obtained in the present. Its value can be defined between 0 and 1. A low value means that the future rewards are less significant, while a high value close to 1 means that the future rewards are almost as important as the present reward. It is essential to set this value correctly to represent the system's attitude towards future actions. If this value is set too low, the agent will not be able to consider that by doing something less rewarding in the present, it can obtain a much higher reward in the future.

3.3.4 Reward function

The agent needs feedback from its environment to determine the value of its actions. This feedback is defined by the reward function. This reward function is the collection of rewards and penalties the agent can obtain by trying different actions leading to various states. Its values need to be considered carefully since these are what steer the learning in certain directions. If the rewards are not set up correctly the learning can be misdirected to obtain a different behaviour than intended. For example, if the goal of the learning is for a drone to reach a certain position, but reaching said position is not rewarded by a higher value than hovering in place, the drone may never learn to do what is expected of it.

So far the general reinforcement learning specific parameters have been shown. Now the parameters regarding training the neural network will be described. These are all important to obtain a successful learning process, and if they are not configured properly, they can cause the process to fail, even if the general reinforcement learning-specific parameters are set correctly.

3.3.5 Learning rate

As a simplified description, it can be said that training the neural network means changing the weights of the system so that it eventually calculates a required output. This change is carried out every iteration of the training, and the magnitude of this change needs to be properly set. The parameter responsible for this is called the learning rate. It is particularly important to set this parameter correctly because its value can be the deciding factor between success and failure. A too high value may make the learning process to diverge, while a too low value slows down the learning significantly both in real time and with regard to how many episodes it takes to find a stable solution. Figure 7 shows a representation of this effect.

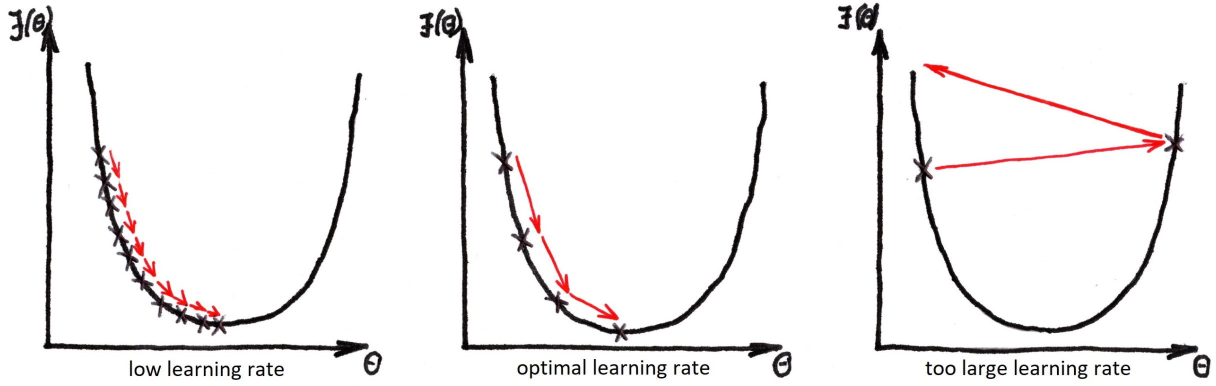


Figure 7: Effect of learning rate size

3.3.6 Loss function

The loss function is also an important parameter. It determines how the difference between the expected output and the output of the neural network is calculated. Different loss functions are more appropriate for different applications, for example, different methods are used for classification problems than for regression ones [33]. As explained earlier, DQN is a regression problem. Three of the most commonly used loss functions for regression problems are the Square loss which calculates the loss based on Equation 11, the Absolute loss which is based on Equation 12, and the Huber loss, shown in Equation 13.

$$L_{Square} = (y_i - \hat{y}_i)^2 \tag{11}$$

$$L_{Abs} = |y_i - \hat{y}_i| \tag{12}$$

$$L_{Huber} = \begin{cases} \frac{1}{2}(y-\hat{y})^2 for |y-\hat{y}| \leq \delta, \\ \delta|y-\hat{y}| - \frac{1}{2}\delta^2 otherwise \end{cases} \tag{13}$$

The Square loss is beneficial because squaring the errors means that it does not matter if the predicted output value is higher or lower than the target, only the difference counts. Its main drawback is that the further the predicted output value is from the target value the more significant will be the increase in the loss. This results in the Square loss being sensitive to outliers [34]. The Absolute loss represents the average of the absolute differences. Sometimes it is used as an alternative to Square loss since it is not sensitive to outliers. However, this function is not differentiable at zero which can create problems for various optimization methods. Using the Huber loss combines the properties two methods. For smaller differences, it acts more similar to the Square loss, while for bigger differences it is more similar to the Absolute loss.

We talk about cost function if the loss is calculated over multiple data points. The Mean Squared Error (MSE) function (Equation 14) is based on the Square loss function, while the Mean Absolute Error (MAE) (Equation 15) function originates from the Absolute loss function.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{14}$$

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{15}$$

3.3.7 Optimizer

The optimizer is responsible for modifying the weights of the network at each step based on the calculated loss value and the defined learning rate. The loss value is calculated by comparing the output of the neural network to the output that is expected. After this calculation, the optimizer modifies the weights of the neural network. Some of the most common ones are Stochastic Gradient Descent (SGD) [35], RMSprop [36], and Adam [37].

3.3.8 State

What is included in the state of the agent also needs to be decided. The information about the state will be used as input for the neural network. If something important is missed, the system won't be able to use that information for its learning process. It can be imagined that in the case of a goal-searching task for example, leaving out the position of the agent will lead to it not being able to utilize it to decide on its actions. This will most likely cause the process to fail. As the states are the input for the neural network, they also have an effect on the network size.

3.3.9 Network structure

The structure of the neural network is also quite important. The number of layers and the number of nodes in them can have a great influence on the effectiveness of the learning. A too small neural network may not be able to capture the target function, while a too big one may take longer to train, both in real-time and by the number of training cycles. Big networks are also prone to overfitting. A too complex structure is also more susceptible to diverging if the learning rate is not set correctly.

3.3.10 Activation function

Having decided on the neural network structure, it is also necessary to choose the activation function for the nodes in the layers. The purpose of an activation function is to create the output signal of a node based on its input signal. This is usually done using a non-linear transformation, although linear activation functions are also used. Popular activation functions are for example Sigmoid, Tanh, and ReLU [38].

3.3.11 Target alignment period

As described earlier, there is a need for a separate target network to calculate the expected future reward in order for the learning process to be stable. The target alignment period determines how many episodes are executed between updating the weights of this target network. Aligning the target network too frequently can make the learning process unstable while aligning it too infrequently can slow down the learning.

3.3.12 Batch size

As explained, the system is trained using smaller batches of data sampled from the experience replay. This is why it is also necessary to choose an appropriate batch size. A too small batch size may make the learning noisier and prevent effective convergence, while it has been shown [39] that a too large batch size can also make the learning suboptimal.

This chapter provided an overview of reinforcement learning and deep Q-learning. The important parameters of the simulation tackled in this project were also presented. The next chapter will describe how the simulation environment was implemented.

4 Implementation of the simulation environment

The main goal of this project was, as described in Chapter 1, to create a simulation environment with relatively high levels of realism using the software packages that Almende and DoBots have experience with, and assess its capabilities through a selected use case. The aim of this chapter is to show the work which was done to implement this simulation environment. The information presented here includes describing how the environment was implemented and what difficulties were encountered while doing so. This description will be useful for both future users of the setup and for those who aim to create a similar environment.

4.1 Benchmark test

As the first step of the implementation work, already realized reinforcement learning solutions were examined. The main focus was put on how much time these take, and how much computational resources they need. The reason for this was to establish a preliminary benchmark for reinforcement learning simulations. Naturally, the simulation time and resource usage are hugely dependent on the task itself, however, it was decided that this examination still provided useful insight.

For the testing, three software packages were used. These were the gym_gazebo package [40] with the task of teaching a rover to navigate in a maze, the openai_gym package [41] with the task of solving the cartpole [42] problem, and the stable_baselines_3 package [43] with the same cartpole problem and the lunar lander [44] example. From these use cases, the rover navigation example had a more realistic world, including a physics engine, while the cartpole and the lunar lander examples were running in a simplified environment without a physics engine.

During the tests, the CPU usage and temperature, the GPU usage and the memory usage were examined. The tests were executed using an ASUS VivoBook S15 laptop, with NVIDIA GeForce MX 150 graphics card and Intel i5-8265U processor. During the rover navigation tests, however, this laptop kept overheating and shutting down. This was the reason behind recording the CPU temperature. After this was discovered, a different laptop with more resources was tried out and it was found that it did not encounter this problem. That is why, for further carrying out this thesis project, an HP OMEN laptop was used with GeForce GTX 1060 graphics card and Intel Core i7-8750H processor.

As it was expected, the learning process in a simulation with realistic physics took considerably more time than learning processes in less realistic environments. While the cartpole and lunar lander tasks were learned in the course of minutes, the rover navigation was in the range of a few hours. It was also seen that running the cartpole problem was extremely slow in case of the openai_gym package, while quite fast in case of the stable_baselines_3 package. Since the cartpole task and its environment is the same in both cases, this most likely originates from a bug in either the downloaded package or in its use. Nevertheless, based on these observations, it was determined that running a reinforcement learning task in an environment incorporating realistic physics would be feasible, however, the simulation speed would need to be significantly increased.

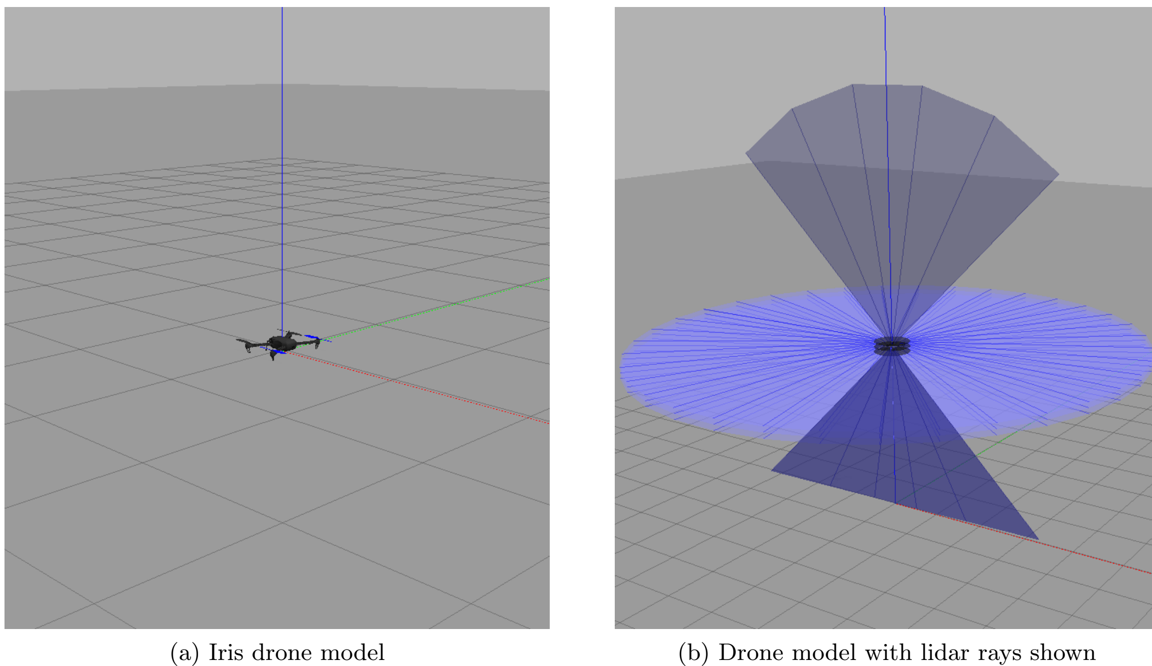
4.2 Main elements

After the examples that are described above were examined and it was determined that running a reinforcement learning simulation in an environment realistic enough to require a physics engine was feasible, the main building blocks of the planned simulation environment were identified.

These were the physics engine, the drone controller, the drone model, and the software which realized communication between the elements. Additionally, the implemented reinforcement learning algorithm was also an important part of the simulation. As the theory behind this algorithm was already described in Chapter 3, in this chapter only its implementation will be detailed.

As was described in Chapter 1, the simulation engine of the environment was Gazebo Classic - first version 9, then 11 - the drone controller was PX4, and ROS/ROS2 was used to implement communication between the various software elements.

With regard to the drone model, it was decided that in this project, the iris drone model would be used. It is a quadcopter model available in the PX4 software package, which made it straightforward to integrate it into the simulation. This model was equipped with lidar sensors to be able to gain information about its environment. Similar to cars, this setup also had blind spots, where the rays were not emitted. The scenarios tackled in this project were simple enough so that this did not cause a problem, but in the course of future work, when creating scenarios to obtain more advanced traffic rules, this limitation will need to be considered. Figure 8 shows a representation of the drone model, and how the sensors were placed.



(a) Iris drone model

(b) Drone model with lidar rays shown

Figure 8: Drone models

In the scenarios that were carried out, the drone was moved with velocity commands. This means that every action of the drone was defined by a velocity in a certain direction and for how long the drone should keep this velocity. For the sake of simplicity, the velocity of the drone was set to be low, 1 m/s. In future work, when more complex rules are to be created, this speed can be increased. The velocity directions were east, west, north, south, up and down. In one step the same velocity was kept for 1 second, measured in simulation time.

An extension to this movement method was also implemented, where the drone kept the defined velocity for 1 second and then the velocity was set to 0 for an additional second. This smoothed

the movement of the drone. This was due to the fact that after the initial 1 second of flying with a certain velocity, the drone had time to slow down and afterwards switch to flying towards a different direction from this slower - close to 0 - speed. Whereas, without the period of 0 velocity, the drone was instantly jerked in another direction, without the possibility to reduce its speed towards the previous direction. Unfortunately, however, this method was not suitable for every scenario. This was the case for example, when the velocity data of the drone at each step was used to gain knowledge about its movement direction in a previous step. This was because if at the end of a step the drone was to be slowed down to have 0 velocity, there was no information that could be extracted based on this about its movement direction.

Table 2 shows what actions the drone was able to do, and what number they were represented in the program with. The actions of the drones were logged throughout the episodes, and the representation shown here was used to process and interpret the resulting behaviours.

code	action
0	move east
1	move west
2	move south
3	move north
4	move down
5	move up
6	move closer to the goal
7	hover

Table 2: Actions of the drone and their code

Unfortunately, the orientation of the internal coordinate system of PX4 could not be changed. It was fixed to the world coordinate system, and if the drone model was turned, these axes would not turn with it. This was why the movement directions were representing the four cardinal directions. This created a problem when, for the purposes of a scenario, two drones needed to face each other. In the case of scenarios like this, a solution was implemented by exchanging the meaning of actions 0 and 1, as well as actions 2 and 3 for one of the two drones.

4.3 Implementation work

Considering the main building blocks of the simulation environment, a search was carried out to see whether an environment already incorporating these elements existed. Eventually, the OpenAI ROS package was found [45]. It is the work of The ConstructCore and has an LGPL licence [46]. A short description of this package [47] was also found, providing help in understanding its main functions.

The OpenAI ROS package uses Gazebo Classic as the simulator engine and uses ROS to control Gazebo from within the program code. The software is created for reinforcement learning applications and also had an example of a drone learning environment implemented in it. However, the part of the system which would include the drone model and control was not found in publicly available sources of The ConstructCore and neither was the actual reinforcement learning

training code included in the package. Nevertheless, this software provided a stable basis for how a simulation environment like this could be elegantly implemented, including how to set up the environment for learning with a drone and how to make the code highly modular.

Figure 9 represents the core idea behind the modularity of this package. Three classes are organized by an inheritance structure, implementing different separate functionalities in the environment. The Gazebo Environment class (the child of the Gym environment, referring to the gym package [48] from openai, which is the highest level and not shown in the figure) implements the necessary functions of the gym class, for example, executing a step or resetting the simulation world. The Robot Environment (the child of the Gazebo Environment) handles the robot-specific tasks of the program, for example taking off, controlling movements and obtaining sensor information. Finally, the Task Environment (the child of the Robot Environment) is responsible for the task-specific details, for example, choosing actions or calculating rewards. This modular structure conforms to requirement (1) in Table 1, namely that adding new scenarios will not result in major modifications of other parts of the environment.

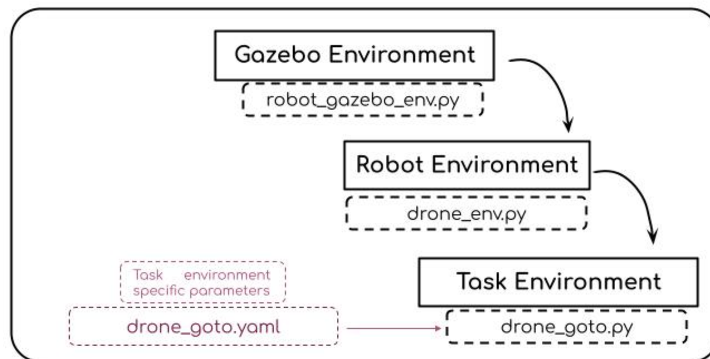


Figure 9: OpenAI-ROS base structure

As the package itself did not have a learning code implemented in it, the publicly available sources of The ConstructCore were searched for such an implementation. An appropriate package was found at [49]. This also has an LGPL licence [46]. The reinforcement learning algorithm used in this package is simple Q-learning. As described in Chapter 3, Deep Q-learning was used for the machine learning use-case of the thesis project. However, it was deemed to present less difficulty if the simulation environment was set up and tested first through a learning code that was expected to work without issues out of the box. Moreover, this package is the product of the same creator as the OpenAI ROS package so it was expected that its integration to the simulation environment would be straightforward. Then, when everything else worked, DQN could be safely added to the system. Since the simulation environment was aimed to be highly modular, exchanging the simple Q-learning code for a DQN implementation was deemed not to present too much difficulty.

After these packages were found, it was assessed what changes and additions were necessary to create a simulation environment that fitted the purposes and requirements of this thesis project. During this assessment, the following challenges were identified:

1. It was necessary to gain a good understanding of the existing code so it would be possible to modify it properly.
2. The learning code was in a separate package from the package containing the OpenAI-ROS environment.
3. The environment needed to be implemented in a way that it would be straightforward and quick to set up on new systems.

4. PX4 was set to be used as the drone controller.
5. The part of the system which would include the drone model and control was not found in publicly available sources of The Contstruct.
6. The simulation time needed to be increased significantly.
7. ROS was getting outdated.
8. A significant element of the use case was multi-drone support, which was not included in the OpenAI-ROS's example drone task environment.
9. The use case required a more advanced learning method than Q-learning.
10. The parameters of the environment needed to be handled in a way that it would be fairly straightforward to change them. This was needed so the simulation could be set up quickly on other systems, especially for Almende and DoBots' other projects.
11. The end user needed access to the output of the simulations.
12. The environment needed to be tested through the selected use case.

To the challenges described above, the following steps were determined to provide appropriate solutions:

1. Understand the existing code and take out the unnecessary elements from it
2. Incorporate the found Q-learning package to the environment
3. Run the environment in docker containers
4. Examine how the PX4 SITL software is structured and integrate it into the code
5. Examine the example environment created for learning with a drone included in the OpenAI-ROS package and integrate the iris drone model based on this solution
6. Speed up the simulation time with the help of PX4's solution
7. Migrate the code from ROS to ROS2 and from python2 to python3
8. Implement a multi-drone usage solution to the system
9. Introduce DQN
10. Make the environment parameterized for future integration to Almende and DoBots' other projects
11. Implement logging and visualization tools
12. Design, implement, configure and execute scenarios

(1) Understand the existing code and take out the unnecessary elements from it
The first challenge of the development process was to understand the existing codebase and reduce the amount of code by taking out unnecessary elements from it. The goal was to start the process of modifications with a simple skeleton structure. By eliminating the elements which would not be used later, making modifications and adding new parts to the code becomes more straightforward. Naturally, it was not a one-time process, as later in the development it turned out that some parts originally thought significant were not necessary.

For executing this cleanup step, it was essential to have a good understanding of how the package worked. Without this knowledge, important parts could have been deleted that would be needed later. Fortunately, understanding the OpenAI-ROS package turned out to be less difficult than expected. The code was well-commented and its structure was straightforward. Additionally, the description found at [47] also helped.

During the development of this simulation environment, a bigger cleaning up was executed first after fully understanding the structure of the original code and before starting the modification process. Another significant cleanup was done before migrating the code from ROS to ROS2. However, smaller cleaning-ups were executed during the other phases too.

(2) Incorporate the found Q-learning code into the OpenAI-ROS package
Since the OpenAI-ROS package had no training code included, first, the package found at [49] was integrated with the OpenAI-ROS package. This package was also the work of The ConstructCore and was based on the same mechanism as the OpenAI-ROS package. This made the integration fairly straightforward. The two most important scripts that were used from it were the *start_training.py* and the *qlearn.py*. The former is the script that controls the learning process, while the latter is the script that includes the Q-learning algorithm.

(3) Run the environment in docker containers
It is quite useful for a simulation environment to be widely available and straightforward to set up on new systems. A good way to achieve this is to containerize the software. This is why the simulation environment of this thesis work was also designed to run in docker containers. This setup had many additional benefits. First, it made the machine learning simulations easy to parallelize. For the different scenarios as many could be run parallel as the user had resources for. This also contributed to the scalability requirement of the environment. The use of docker containers also made it possible to run the software on different operating systems. What is more, this solution made integrating the simulation environment into Almende and DoBots' other projects more straightforward, since many of them also work with docker containers.

The docker container used for this solution was based on what is used for DoBots' other projects. This container included the ROS, the PX4, and the Gazebo installation as well as implementing some additional changes that were necessary for the setup to work.

During the migration from ROS to ROS2, the docker container structure was also changed. Instead of using one container which included every necessary software package, two separate containers were used. Naturally, ROS2 was installed on both. Besides this, one container was set up to incorporate the PX4-Gazebo part of the environment, and the other was used to control the whole learning process. This separation created the possibility to run the two parts even on different computers and use their full resources. In order for the ROS2 topics to be available between the two containers they were set up to belong to the same sub-network with the use of a docker-compose file.

(4) and (5) Integrate the PX4 STIL solution and the iris drone into the codebase
As it was described earlier, the plan for the simulation environment was to include a rotary-wing drone, and for the sake of having realistic flight properties, this drone was to be controlled by a flight controller. As it was also explained, the PX4 SITL implementation was set to be this flight controller. Integrating these two elements into the simulation environment was tackled together, because the iris drone, a rotary-wing model, was already integrated into the PX4 software.

The integration of the drone model and the drone controller into the simulation environment

meant first, implementing spawning the drone at the appropriate place in the code. This was carried out at the beginning of the training process. Then, the drone example provided in the OpenAI-ROS package was modified to work with the PX4 control solution. This meant modifying the Robot, and somewhat the Task Environment class. It was determined which ROS topics were necessary for controlling the drone and gaining information about its state. Then, the appropriate subscribers and publishers were defined, and the solutions were added with which the system of the drone can start up and the drone is able to take off. For this work, the experience of Almende and DoBots with PX4 was quite useful.

The original OpenAI-ROS implementation uses velocity commands to control the drone, as it was explained in Section 4.2. To this, two additional control solutions were implemented. One with adding a 0 velocity phase, as described in Section 4.2, and one which used position commands. Compared to the velocity command solution introduced in Section 4.2, with this one the drone was moved a set amount of meters in one direction in each step. For example, the drone was moved 1 m to the west, then 1 m to the north, etc. It was determined, that while the position commands were more precise with regard to positioning the drone in the world, the velocity commands provided much faster movement. Since in real life, velocity commands are common for a drone controlling task, and the simulation speed is important for the machine learning application, the velocity commands were chosen to be used later in executing the scenarios.

After the PX4 software and the implementation for the iris drone were integrated into the environment, the setup was tested with the Q-learning algorithm to examine whether the behaviour of the drone was correct throughout the learning process. Unfortunately, a problem was encountered when resetting the environment between two learning episodes. In the original OpenAI-ROS implementation, resetting the environment was carried out by resetting the Gazebo world, which resulted in every element being placed back to its original spawn position in the world. This was also true for the drone model. Unfortunately, however, this did not reset the PX4 internal location sensor, as it was running on a separate process. As the world was reset, and the drone moved back to its original spawn position, what the PX4 system detected was a jump in its location and this jump caused the system to settle very slowly. It was soon discovered, that currently there was no functionality in the PX4 SITL system that would allow it to reboot, or reset the location sensor of the drone. Therefore, a workaround was needed.

The first attempt at solving the problem was to delete the drone model from the simulation and respawn a new one at the beginning of the next episode. However, when doing so multiple times, the Gazebo system broke down. The debugging process of this problem consisted of first, updating the Gazebo version to a newer one, 9.19.0 based on the suggestions found at [50], [51] and installing an additional package called ros-melodic-gazebo-ros-control. Unfortunately, this did not solve the issue. After continuing to debug the problem, it was found that the issue was caused by the drone model itself. This was found out by first, writing a script that deleted and respawned a simple basic box model multiple times. When this did not cause any problems to the simulator engine, the same was tried out with the iris model. Using the iris drone model, the Gazebo process always broke down eventually. The number of delete-respawn cycles it took to do so was random, but the system broke usually within one or two, maximum five cycles. After debugging the code for the drone model, three areas were identified which were responsible for the Gazebo software dying. These were the GPS module, the ground-truth plugin, and a part regarding TCP in the mavlink plugin. At this point, it was decided not to debug these issues further, because doing so would have presented a risk to overrun the project timeline. Therefore, a different solution was needed.

The next idea for solving the restart issue was that instead of deleting the drone model, only

kill the PX4 control process, and start a new one. Unfortunately, when this solution was tested it was found that the restarted PX4 process could not reconnect to the existing Gazebo process anymore. A debugging process also started here, however, it did not result in a solution in the allowed timeframe.

Finally, two solutions were devised that were capable of solving the restart issue. In the first one, the world was reset just as in the original implementation, but before doing so the drone was landed at its spawn position. After the world was reset, the process waited for a longer time period so that the location system could settle on the position. This wait was necessary, because due to the realism which was incorporated into the environment, the drone never landed exactly at the startup coordinates, only very close to it. Therefore, resetting the world still caused a small jump in the drone's position.

In the second solution, both the PX4 and the Gazebo processes were killed at the end of every episode and started again at the beginning of a new episode.

As both of these two solutions worked, the time it took to carry them out was measured in order to determine which one was faster. This provided a basis to choose from them.

For the *landing the drone* solution, the speed of the simulation was increased to be faster than real-time. This was implemented by creating different physics profiles in the Gazebo world and switching between them with the *gz physics -o physics_profile_name* command. At this point, a bug in the implemented environment was detected, by encountering the error message "ERROR [mavlink] [timesync] Time jump detected. Resetting time synchronizer". After debugging, the cause of this message was found to be that the value of the use_sim_time parameter was not set to true, causing the simulation to not run in lockstep. After the value was set to true, the error was solved.

For the *killing and restarting PX4 and Gazebo* solution it was also devised how to check whether the Gazebo and PX4 systems have successfully started up. Simply waiting some amount of time was deemed not to be appropriate. The restart process could take a slightly different length every time, and waiting as long to be sure would increase the simulation time significantly. Eventually, for determining whether Gazebo has started up, a function was added to the code which repeatedly tried to unpause Gazebo without limiting the number of tries. It was determined that Gazebo has started up fully when this unpause command was accepted. In the case of the PX4 process, the value of two different variables was checked. These were the *manual state input*, which value had to be true, and the *guided state*, which value had to be false.

After measuring the time of the two solutions it was found that restarting both PX4 and Gazebo was about twice as fast as landing the drone and waiting for the system to settle. What is more, restarting the whole simulation was more robust since the time it took for the drone to settle could change depending on where exactly the drone landed.

After the restarting issue was solved, the simulation along with the learning process seemed to work correctly. However, another issue presented itself, that sometimes at the beginning of an episode, Gazebo did not start up correctly. There were different error messages that all were the signs of this same issue, such as the failure of the unpause service, or that different ROS topics of the PX4 system providing the drone sensor information were unavailable. The Gazebo world was frozen and the Gazebo server did not accept commands. Though this issue occurred rarely, its occurrence was completely random. To try to determine the cause of this problem various things were examined. It was determined that neither the internet connection of the computer

nor the Gazebo model database access was the cause. Multiple different logfiles for both ROS and Gazebo were examined but provided no information that could be used to determine the cause. After these debugging attempts, it was decided that further debugging would lead to an extremely lengthy process and a workaround solution would be needed.

In order to provide a solution to the *Gazebo-breaking* problem, and thus try to fulfil the robustness requirement, a monitoring script was implemented, using a similar mechanism as a watchdog system. At the beginning of every episode, a heartbeat message was sent to a ROS topic from the learning process. An external script was started that monitored the arrival of the heartbeat messages to this topic. If no message arrived in a determined amount of time, the script detected it as the system was stuck and restarted the simulation the same way it was restarted after every episode.

At this point, a warning was encountered that the disk usage of the simulation was over 1GB due to the ROS and Gazebo logs filling up space. To solve this problem while still being able to access information if necessary, the logs were deleted every 100s episodes.

(6) Speed up the simulation time
The PX4 SITL software has the functionality to speed up the simulation time by setting the value of an environment variable called PX4_SIM_SPEED_FACTOR. The value of this variable represents how many times faster the simulation should run compared to real-time. This number, however, is only an indicator of the maximum value that is wanted by the user. The actual speed of the simulation can be less, based on the available computational resources.

As it was shown when describing the episode restart problem, the speed of the simulation can also be increased by changing the physics profile of the world. However, changing the physics profile was found to be much more cumbersome compared to using the PX4_SIM_SPEED_FACTOR variable. With physics profiles, the code of every world would need to be modified, and a separate physics profile would need to be defined for every different speed value. This is why it was decided to use the PX4_SIM_SPEED_FACTOR variable for influencing the simulation speed.

When using the PX4_SIM_SPEED_FACTOR variable, a bug was encountered in the PX4 SITL software. After setting the value of the parameter to anything different than 1, an error message appeared "ERROR [param] not enough arguments. Try ‘param set COM_OBC_LOSS_T 3 [fail]’". During the debugging process, it was found that in the file */usr/px4/etc/init.d-posix/rcS* the system timeout values were calculated based on the value of the PX4_SIM_SPEED_FACTOR parameter, and one of these calculations had an incorrect multiplier. The problematic line of code was
COM_OF_LOSS_T_LONGER=$(echo “$PX4_SIM_SPEED_FACTOR * 0.5” | bc)
Instead of 0.5 the value should have been 1.0 for the system to work correctly in sped-up simulation time. Additionally, the *bc* package also needed to be installed. These changes were made permanent by adding them to the Dockerfile, with an instruction to install the appropriate package and modify the indicated line before building the PX4 package.

It was tested what speed factor would be the highest with which the simulation environment could work on the laptop described in Section 4.1. Due to the higher computational requirements of a sped-up time simulation, if the value of the PX4_SIM_SPEED_FACTOR variable is too high, the simulation speed will start decreasing instead of increasing. It was found that without GUI, a factor of 4 was still good, while with a factor of 5, the simulation speed started to decrease. Turning off the GUI of the simulation is useful for freeing up computational resources. After the migration to ROS2 was carried out and the DQN algorithm was added, a speed factor

of 3 instead of 4 was the best achieved on the laptop which was used to carry out the simulations.

Using a sped-up time was tested by examining the PX4 logs, using the *logs.px4.io* website. It was determined that both the velocity and the position commands worked as required, and no strange behaviour was discovered on the plots.

After the steps described above were carried out it was tested that the simulation was stable for many thousands of episodes. Before starting the process of migrating from ROS to ROS2, a cleaning step was carried out. The code was made more compact by taking out some additional elements, and the Gazebo version was updated to 11.11.0. Additionally, the class structure was also redesigned.

In the new code of the learning process, the drone, which was the agent of the reinforcement learning simulation, was managed through a *robot instance*. Figure 10 shows the structure of this robot instance. Compared to the OpenAI-ROS implementation, it still consists of three classes, however, the Gazebo Environment class was exchanged for the Learn Environment class. Moreover, the inheritance structure between the classes was reversed. In this implementation, the Learn Environment is the child of the Task Environment, while the Task Environment is the child of the Robot Environment. Additionally, this Robot Environment was no longer the child of the Gym class.

Just as it was in the case of the original OpenAI-ROS package, the Robot Environment was set to be responsible for the drone-specific parts of the code, for example taking off, or carrying out movements. The Task Environment was responsible for task-specific elements, like calculating rewards or obtaining state information. Finally, the Learn Environment was responsible for some of the learning-specific tasks, for example how the system is reset after an episode. As separate classes, they were freely exchangeable to other implementations, without changing other parts of the code. It is useful to know this structure so that future users of this environment will be able to modify these and implement their own scenarios, including the task, the type of drone and even the learning algorithm.

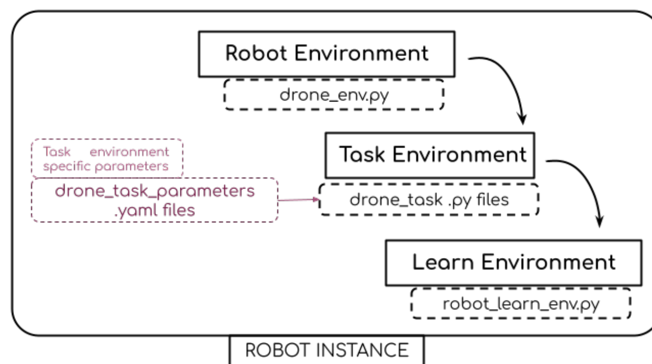


Figure 10: Structure of robot instance

(7) Migrate the code from ROS to ROS2 and from python2 to python3

Another challenge encountered during the development was that ROS was getting outdated. This was especially noticeable when using python libraries for the various solutions. The reason for this was that ROS mainly uses python2, which was fully deprecated in 2020, while ROS2 uses python3. To solve this problem and keep the code of the environment up to date, it was

decided to migrate to ROS2.

For a user to carry out this migration process, the first step should be to understand the main differences between ROS and ROS2. In this work, mainly the ROS2 documentations were used [52], [53] for this goal. The examples in the tutorials were examined, and some of them were carried out locally, including setting up a package, creating launch files, creating nodes, and writing simple publishers and subscribers.

The changes that influenced the code of the simulation environment the most are listed below:

- New package structure
- Different launch system and node handling
- Different python package (rclpy instead of rospy)
- Different parameter handling
- Different plugins for Gazebo

After the main novelties and changes in ROS2 were understood, first, a new ROS2 package was created. The files containing package-specific information were set up, as well as a folder structure inside the package.

ROS2 uses a different launch system and node handling compared to ROS. This meant that instead of initializing and starting nodes separately, as was done in the earlier implementation, every node was started from a common *main.launch.py* file. Each was set to be a class, inherited from the base Node class. The parameters of the nodes were set in this launchfile, and every parameter used was also initialized inside the node's program code.

In the previous implementation using ROS, the subscribers and publishers for the robot environment were initialized in the Robot Environment class. In this new implementation using ROS2, this initialization was placed before starting the spin command, into the main learning control script's initialization function.

After restructuring the code to accommodate ROS2, the drone's Robot Environment was also reimplemented, since PX4 used a different, and more complex, solution for ROS2 than for ROS. The appropriate topics for controlling the drone and gaining information about its state needed to be identified. It needed to be determined how to take off with the drone and how to control its movements through the python3 code. The control example on the PX4 site [54] was utilized for this. Additionally, a discussion found on [55] was used to determine that the position control values needed to be set to NaN in order for the velocity values to be accepted.

The parameter which needed to be set for the drone to be controllable from the program was also identified. For the first few tests, this parameter was set by using the PX4 console with the command "param set COM_RCL_EXCEPT 4", however, later the modified value of this parameter was hardcoded into the PX4 SITL software itself, by changing its value through the Dockerfile before building PX4 system. This ensured that controlling the drone from the program code would work without further modifications to the program.

The first control solutions were tested through the PX4 console and several other terminal windows. Testing this way made sure to focus on the commands themselves and eliminate the layer of uncertainty that would arise from issuing the commands using the new rclpy package.

A drone .sdf model was created to incorporate the lidar structure as shown in Figure 8. This was done by examining an already existing drone model available in the PX4 folder containing the SITL gazebo models, which also had a lidar sensor equipped to it. In the new version, more lidar sensors were added, with different orientations and changed ray numbers. Additionally, since the original model was written for ROS, the plugins used in the model's .sdf file needed to be exchanged for ROS2 plugins so that they would function with the new setup.

A lidar processing node was also created. The purpose of this node was to process the information coming from the lidar sensors and publish the processed data through new ROS2 topics. This processing included adding a moving average filter to smooth inconsistencies and adding two variables for indicating if something was close and if something was so close it can be considered a crash.

At one point during the migration process, a strange error was encountered. The velocity movement commands which were given to the drone were received by it, but the drone's behaviour was erratic and its velocity included components which were not present in the control commands. Many different things were tested to find the cause of this problem. Various ROS2 topics were tried out for issuing control commands and different values were set for these topics, for example setting the acceleration, position and/or orientation to 0 or NaN. The publisher rate was also experimented with, by setting it to many different values from a few Hz to a hundred Hz. The timestamp values in the published control commands were also varied. Unfortunately, none of these solved the erratic behaviour of the drone. In the end, the cause was found to originate from a mismatch between the ROS2-specific PX4 packages and the PX4 SITL code. These packages are available in two separate GitHub repositories. From the PX4 SITL code always the newest version was pulled, however, the ROS2 packages were downloaded one time and made part of the folder that contained the code for simulation control and learning. This folder was mounted to the docker container. As the PX4 libraries are continuously under development, this caused a mismatch between the two packages and the control solution stopped to work. Additionally, these two different GitHub repositories were not version-locked to each other, so it was difficult to determine which versions belonged together. After this issue was found, the fix for this problem was straightforward. The versions which worked together correctly were identified and locked in the setup.

A problem also presented itself with the monitoring tool. After a few hundred episodes, the subscriber used in the monitoring script failed to receive messages which were sent to the heartbeat ROS2 topic from the publishers in the main learning control script. This led to the simulation being restarted mid-learning by the monitoring tool, thus hindering, and eventually making the learning process impossible. Unfortunately, the cause of this error was not found. It was examined that the heartbeat messages did arrive at the topic (this was tested by connecting a subscriber to the topic from a separate terminal window), and the monitoring script was still alive. Due to this problem, instead of sending data through ROS2 topics, this solution was exchanged for a file write-read solution. The learning control script periodically updated a .yaml file with time information, and the monitoring script periodically accessed and read this file. Naturally, this solution was much slower than the ROS2 version, but since it was only used to test whether the simulation is frozen, and it did not control anything time-sensitive, it was accepted. It proved to be sturdy and worked without issues. In future work, this solution could be changed to a more elegant method.

It was also observed, that with the new ROS2 setup, the frequency of Gazebo not starting up correctly at the beginning of an episode increased. Fortunately, the monitoring tool made the simulation robust enough that this phenomenon did not cause much trouble.

In order to safely restart the system, the state of the learning process needed to be saved after every episode. This save mechanism was implemented with the use of the pickle python3 package. Luckily, this did not create any noticeable delay in the simulation time. This solution was also useful by providing the capability to stop the simulation at any episode and resume the process when wanted.

It was also implemented that the experiences of the drone were stored in multiple files, and at the beginning of an episode, the contents of these files were assembled into one experience replay. At the end of an episode, the new experiences of the drone were saved either to an already existing file or, if this file became too large, to a new one. As described in Chapter 3 the experience replay stored the past experiences of the drone. Due to the restarts after every episode, the experience replay needed to be saved after the end of every episode. After a couple of hundred episodes, however, the size of this file became so large, that the pickle package failed to handle it. This is why this solution was implemented.

(8) Solve multi-drone usage
Running the simulation with multiple controlled drones was an essential element of this project, as the use case was with regards to traffic scenarios, and in traffic, more than one participant should be anticipated. This was also what requirement (8) describes in Table 1. However, the original implementation of the environment was only prepared for having one drone in the simulation. This is why multi-drone support was also added.

To implement multi-drone spawning, the following PX4 script was used:
/PX4-Autopilot/Tools/gazebo_sitl_multiple_run.sh.
This script spawns multiple drone instances in the same gazebo world. This solution was examined, and three separate scripts were created based on it. One was responsible for starting up the gazebo server, one was responsible for starting the gazebo client that visualizes the simulation, and the third one was responsible for spawning the drone instance and starting up its PX4 controller.

The drone spawning process used jinja to instantiate a drone model based on a template. The reason for this was that there were various parameters with regard to the drone model that needed to be modified in case multiple drones were spawned in the same world. These parameters defined for example several UDP and TCP ports through which different elements of the system communicate.

Originally, there was difficulty when trying to use the
/PX4-Autopilot/Tools/gazebo_sitl_multiple_run.sh
script. The problem was identified to be that the jinja program code was missing from the specific version of the PX4 build pulled from the GitHub repository of PX4. To solve this problem, earlier versions were searched for this missing code. It was eventually found and copied to its appropriate place in the PX4 package filesystem.

The main structure of the developed code is shown in Figure 11. This will be briefly explained here. The simulation was started through the monitoring tool. As it was explained, the function of this tool was to monitor the simulation throughout the entire process and if any problems occurred, restart it. The monitoring tool started the main launch file which was then responsible for starting three types of nodes: the *microRTPS agent node*, which was responsible for the communication between PX4 and its ROS2 packages for one drone, the *drone learning node*, which controlled the entire learning process of one agent and the *lidar processing node*, which prepro-

cessed the data coming from the lidar sensors of a drone and passed on this data using ROS2 topics. From these types of nodes as many were launched as many drones were participating in the simulation.

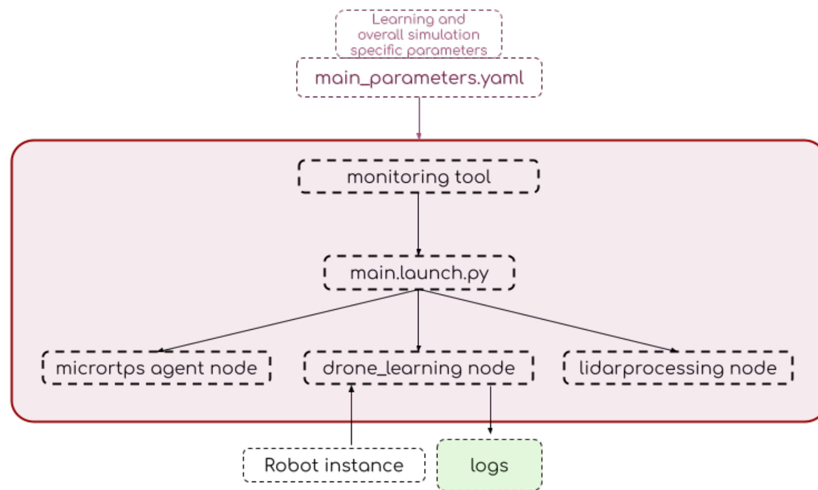


Figure 11: Main structure of the code

If the user of the environment wanted to start the gazebo client script this could be specified in the *main_parameters.yaml* file. Besides this, starting the gazebo server script and the drone spawning script was executed from the *main.launch.py file.* The drone spawning script was parametrized so that the port variables and the spawn positions can be externally added. Initially, the startup was very unstable. During the work to solve this instability, it was discovered that starting the simulation was much more stable if these two scripts were not called directly from the *main.launch.py* file, but from a separate script that was launched from the *main.launch.py* file.

Additionally, a problem was discovered with regard of using multiple drones and sped up simulation time. Based on the error message ("gazebo_mavlink_interface.cpp:397] max_step_size of 0.004 s does not match real_time_update_rate of 500, aborting") the debug process found that the
/PX4-Autopilot/Tools/sitl_gazebo/src/gazebo_mavlink_interface.cpp
file needed to be modified slightly to correct the issue, by only changing the simulation speed if the real-time update rate was in sync with the max step size. If they were not in sync, this indicated that the real-time update rate had been changed previously. In this solution, the possibility of having a badly configured starting world was not considered.

(9) Introduce DQN
As it was described earlier, the chosen machine learning algorithm for the use-case was reinforcement learning, and, due to the realism aspect of the simulation, the specific method was DQN. Since for the initial integration of PX4, iris, and the ROS2 migration, simple Q-learning was used, the DQN algorithm also needed to be added to the simulation environment.

The first step of this process was to gain a good understanding of how the DQN process works. The algorithm was understood, the important parameters were collected and their effects were determined. The theory behind the DQN process and its parameters was described in Chapter 3.

Since DQN uses neural networks, a decision needed to be made on which python3 package to

be used for the training and handling of neural networks. After consideration, TensorFlow was chosen to be this package, more specifically, tensorflow.keras. This package has a big community, therefore if any issue is encountered during its use, it is probable that help could be found online. Moreover, it was deemed to be the most user-friendly of the packages tried out for this simulation environment.

Initially, the description found on [56] was used as a basis for implementing the DQN algorithm for this simulation environment. However, the calculation speed was extremely slow, in the range of an episode taking a few minutes. The problem was identified to originate from how the predictions were calculated. For the target values, every prediction was done separately, while they could have been calculated together at one time. This was solved, by getting all the predictions from TensorFlow at once, instead of one by one. With this solution, the calculation speed returned to normal.

Because of the aforementioned issue, other descriptions of implementing the DQN algorithm were searched for. Eventually, this guide: [30] was found to provide the most comprehensive description of the DQN setup and its improvements. The code was modified based on it and its GitHub repository, which has an MIT license. The code located at [57] was also utilized as well as some information from [58].

During the implementation and testing process, different network visualization tools were also tried out. The two most important were Tensorboard and netron.

At this point, the code was modified to pause the Gazebo simulation process after the drone was finished with a step. This was needed because even though the calculations were relatively fast, they still took some amount of time to carry out. During this time, the program focused on the calculations, and the drone was left uncontrolled. This caused a problem because when detecting that the control was released, the PX4 system immediately started landing the drone. To solve this, it was implemented that the simulation be paused by default, and only be unpaused at the beginning of a movement step.

Afterwards, this pausing system was further modified to support multi-drone usage. As it was explained in the paragraph above, by default, the simulation was in a paused state, and it was only unpaused when a drone was executing its selected action. However, when more than one drone was flying in the same world there was a chance that they unpause the simulation when the calculation process for the training of the other drone was still running. This is why a locking mechanism was implemented that prevented the drones to unpause the simulation before the calculations were finished for all of them. This was implemented through ROS2 messages. As was described in the monitoring tool section, ROS2 sometimes had instabilities. Some problems also occurred in this case, but while in the case of the monitoring tool, speed was not an issue, in this case, communication speed was essential.

Additionally, a strategy was implemented to figure out a good learning rate value to start the scenarios with. The goal of this script was not to determine the best learning rate, nor to make it adaptable, but to determine a learning rate with which the learning process could be safely started with. In this process, after the collection of the first few batches of data, a training with many epochs was performed, using this collected dataset. The training started with a relatively high learning rate (values around 0.1, 0.01). If the losses started to diverge, showing that the learning rate was too high, the value was decreased and the testing process started anew. This method was used for example for the later described *random goal* scenario and in the experimentation with the *static obstacle* scenario. Initially, this strategy proved more effective than trying

to figure out a starting value for the learning rate by hand.

Additionally, a short random waiting time was added before loading or saving the experience replay files for the case when multiple drones were used in the simulation. This was implemented so that the control scripts which ran parallel for the multi-drone solution would not access the same file at the exact same time. The waiting was made conditional to multiple drones being present because even though the time to wait was set to a short value, waiting before accessing every file in the experience replay could increase the length of an episode by a couple of seconds.

(10) Make the environment parameterized

To be able to set up and start scenarios quickly, the code was highly parameterized. The parameters regarding the learning algorithm and the simulation process itself could be set from the
<mounted_folder>/src/drone_learning_ros2/config/main_parameters.yaml
file. Additionally, the parameters regarding the scenario itself could be set from the
<mounted_folder>/src/drone_learning_ros2/drone_learning_ros2/task_envs/drone/config/drone_task_para
file. These are also shown in Figure 11 and Figure 10, respectively.

(11) Implement logging and visualization tools

Throughout the simulation process, the actions of the drones and the cumulative rewards reached in each episode were logged. These were used to evaluate different aspects of the learning. The logged actions could be utilized to examine the behaviour of the drones, while the logged rewards were needed to examine the progress of the training. In this step, a visualization script was also written. This was used for creating the result graphs presented in Chapter 6.

(12) Design, implement and configure scenarios

For the machine learning use case, many different scenarios were implemented. These all depict elementary traffic situations with simple worlds. The work included deciding on a traffic scenario and designing the world and its elements to depict the situation. Then this needed to be implemented in Gazebo and saved. Afterwards, the Task Environment needed to be modified. The states which will be used for the DQN process and what was considered to be the end state needed to be decided. Additionally, for all the scenarios the appropriate learning parameters needed to be set. The implemented scenarios will be described in more detail in Chapter 5.

Besides these scenarios, an example world was created to showcase the possibilities of the simulation environment. It depicted a scene where cars were located along a street, with pedestrians walking on the pavement and houses on each side. Birds were flying in the air. The cars and buildings were static obstacles, while the human and bird models were dynamic. Different movement scripts were also written for the dynamic models. From these, the most important ones were the script that controlled the movement of the human model and the script that controlled the movement of the bird. In the case of the former, the model was moved along a defined line, back and forth. This represented the movement of a pedestrian walking on the street. In the case of the latter, the model was moved along randomly created lines in a defined 3D cuboid, representing the random flying of a bird. It was also implemented for the movement scripts that the models face the direction they are moving towards.

Additionally, collision models were defined for the models used in the simulated world. This was needed because if the collision model was as detailed as the model itself, it would slow down the simulation considerably. The human model had a simple cylinder collision model and the bird had a sphere. The cars had cuboid shapes whose dimensions were carefully designed to closely encompass the model. The houses had their own collision models specifically designed for them in Blender.

At one point, a chopping movement was discovered in the movement of the flying birds. The cause was identified to originate from the fact that the model was set to static in order to "stay in the air". Instead of setting it to static, it was enough to turn off the gravity for the model. This resolved the chopping movement.

In this chapter, the main elements of the simulation were described as well as the work that was carried out during its implementation. The simulation engine was Gazebo Classic, while the drone controller was implemented with the use of PX4 SITL. ROS2 was used for communication between different elements of the simulation. The iris model included in the PX4 codebase was used as drone model, completed with lidar sensors. The main points of the implementation process included incorporating the PX4 code into the system, running the solution in Docker containers, migrating from ROS to ROS2, and adding DQN implementation to the system.

After the implementation was finished, the environment was tested through the execution of various elementary drone traffic scenarios. This will be described in the next two chapters, Chapter 5 and 6.

5 Reinforcement Learning use-case

The goals of this project included testing the simulation environment through the devised model and assessing the benefits of the incremental complexity approach. The simulation environment created for these purposes was described in Chapter 4, while the model was explained in Chapter 2. This chapter presents the scenarios which were created for the distributed drone traffic generation use case. Three challenges that came with using the formulated method will be presented, providing points of caution also for future users. Finally, the configurations of the various parameters of the learning process will be explained.

5.1 Scenarios

The implemented scenarios are summarized in Table 3. Here it is also presented which scenarios were planned to be used for testing the incremental complexity approach. In the course of this work, five scenarios from the seven implemented ones resulted in learning processes that could be accepted from an engineering viewpoint. From the remaining two scenarios, the *two drones window* scenario did manage to develop good solutions to its traffic problem even though its learning process did not stabilize with the attempted configurations.

The description of the scenarios includes introducing the task of the drone for the scenario as well as showing the created world. In the images showing the 2D representation of the world the arrow pointing directly outwards from the drone symbolizes the east direction. The data which was used to train the neural network of the system is also listed. In summary, it can be said that in each case the goal of the drones was to fly to a specified area in space without colliding with anything. For the sake of simplicity, in most cases, this area was a cube. This imitates for example a delivery task. However, in the case of the *window* and *two drones window* scenarios, this area was the other side of the wall on which the window was located.

Name	Shorthand	Short description	Stable results	Builds on
Empty space	E	Static goal in an empty space	done	-
Random goal	ER	Randomly placed goal in an empty space	done	-
Static obstacle	O	Static obstacle in the path of the drone	done	-
Random obstacle	OR	The position of the obstacle is changed between the episodes.	future work	-
Two drones empty	TE	Two drones on a collision course in an empty space	done	ER
Window	W	Enclosed space with a window	done	-
Two drones window	TW	Two drones in neighbouring enclosed spaces with a common window	future work	W

Table 3: Summary of the implemented scenarios

The incremental complexity approach in this work was implemented by copying the resulting neural network of a simpler scenario to be the starting state of a more complex one. Generally, in order to be usable in other scenarios and not just on its own, the trained neural network needs to have a flexible strategy encoded in it. This means that the solution of the system to the pre-

sented traffic problem should not depend on the static elements which build up the world. Since the neural network tends to encode these static elements, a flexible strategy can be taught to it by changing components from episode to episode on which the strategy should not be dependent.

Although what is described above is generally true also for the scenarios of this project, a network with a static strategy can still be utilized if the static encoded elements of the world remain the same also in the next scenario. Moreover, the resulting behaviours are usable for the purpose of rule creation even if the learned strategy is static. This is due to the fact that a rule can be formulated flexible even if it is based on a static strategy, because of the presence of a human observer. This shows the benefits of human heuristics in the process.

5.1.1 Empty space

This was the only implemented scenario that did not aim to provide a solution for a traffic situation or have rules extracted based on its results. Instead, this one was used to test whether the setup and the implemented algorithm work correctly. In this scenario, the drone's task was to learn to fly to a specific point in space. Figure 12 shows the 2D representation of this. Since the space is empty, the 3D representation would not show anything more than what was already depicted in Figure 8b. In every episode, the starting position and the goal position of the drone were placed in the same location. This made the task very easy, and thus appropriate for evaluating whether all components of the system worked as intended.

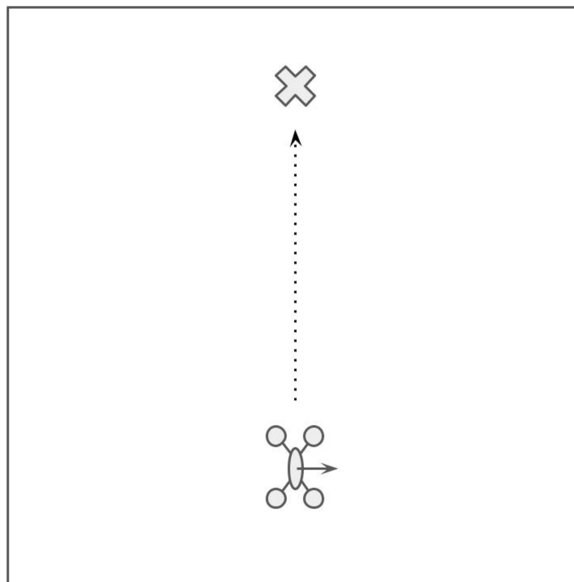


Figure 12: 2D representation of the empty space scenario

The following input data was used to train the neural network:

- rounded x, y, z position (3 data points)
- rounded roll, pitch, yaw orientation (3 data points)
- middle horizontal lidar data (35 data points)

5.1.2 Random goal

This scenario was meant to symbolize that, in case there are no obstructions in the way, the drone should follow the path devised by its trajectory planner. This can be seen as the first

step in rule generation, and also as an important building block for future scenarios using the incremental complexity approach. Compared to the *empty space* scenario, the strategy developed in this case was flexible - what the drone should do did not depend on where it was positioned in the world. In order to develop a strategy like this, the goal position of the drone was changed throughout the episodes, thus not allowing the algorithm to learn a static path. What the drone needed to learn instead, was to use its *move closer to the goal* action, in other words, to trust its trajectory planner. Figure 13 shows the 2D representation of this scenario. Similarly to the *empty space* scenario, the 3D representation of this world shows no more than Figure 8b, the world being completely empty.

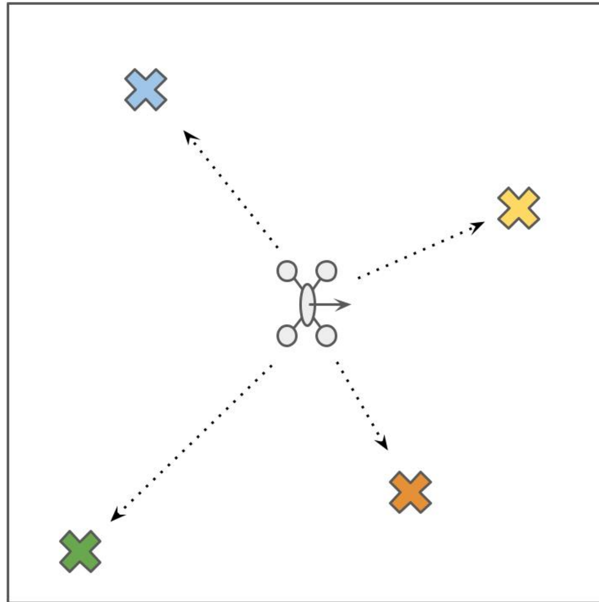


Figure 13: 2D representation of the random goal scenario

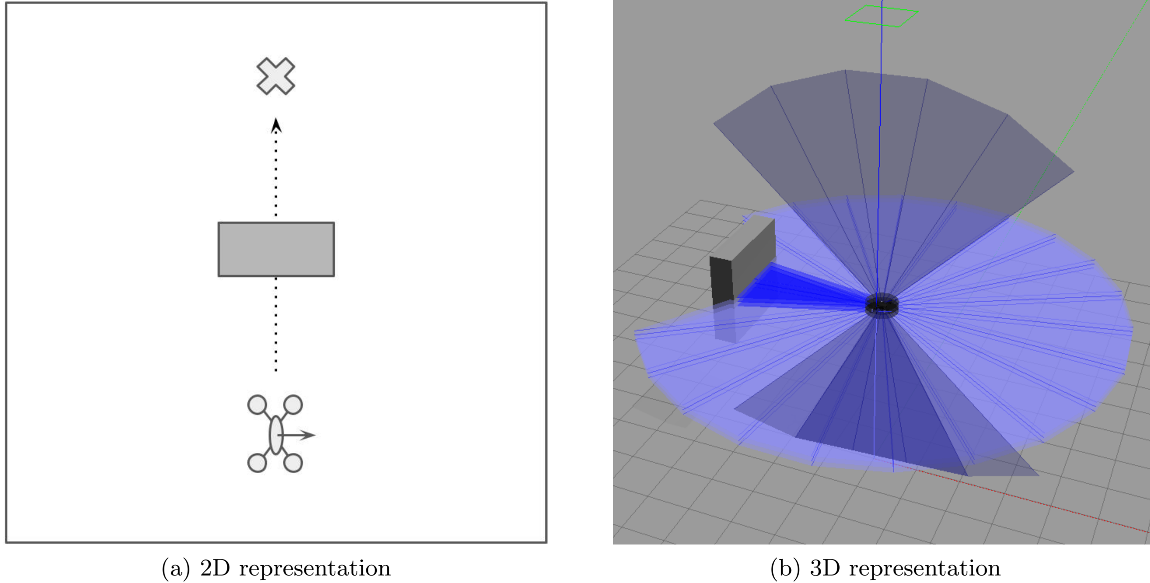
The following input data was used to train the neural network of the system:

- x, y, z position (3 data points)
- roll, pitch, yaw orientation (3 data points)
- middle horizontal lidar data (35 data points)

5.1.3 Obstacle

In this scenario, the drone needed to learn how to avoid an obstacle in its path. The representation of the simulated world is shown in Figure 14. The starting position of the drone, the location and size of the obstacle, as well as the position of the goal, all stayed the same throughout the episodes.

The resulting neural network of this scenario encoded a static path around the obstacle. This was good enough for rule extraction, however, a more flexible strategy would be needed for the incremental complexity approach. For this goal, the elements of the world would need to be varied throughout the episodes, and this is why the *random obstacle* scenario was created.



(a) 2D representation

(b) 3D representation

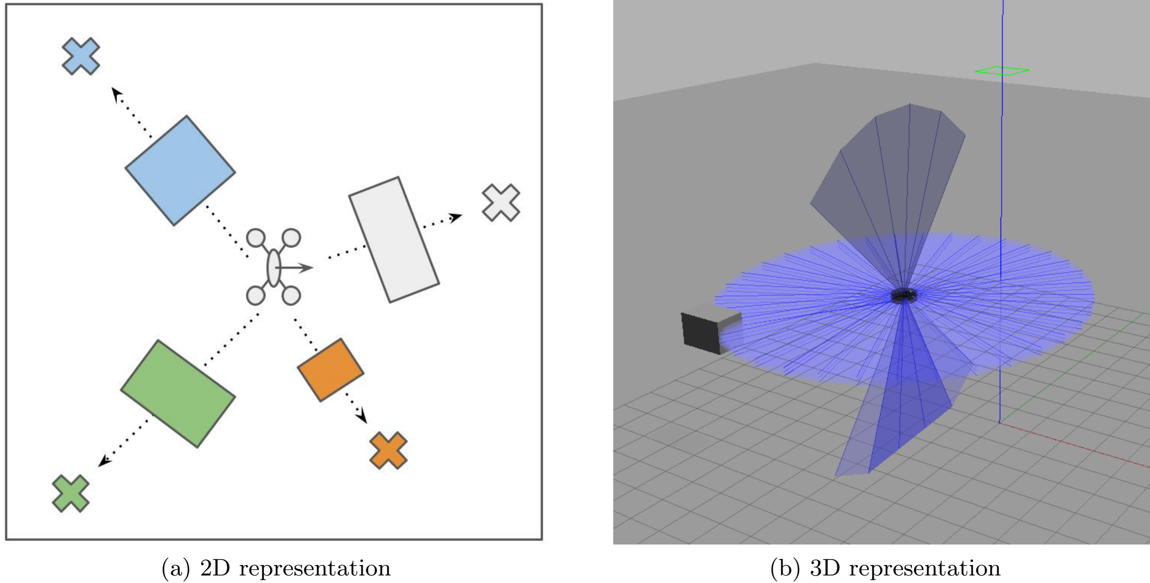
Figure 14: Representation of the static obstacle scenario

The following input data was used to train the neural network of the system:

- x, y, z position (3 data points)
- goal x, y, z position (3 data points)
- middle horizontal lidar data (35 data points)
- vertical upwards lidar data (6 data points)
- vertical downwards lidar data (6 data points)

5.1.4 Random obstacle

Similarly to the previously shown *static obstacle*, in this scenario, the drone also needed to learn to avoid obstacles in its path. However, in this case, the scenario was set up in a way so that the developed strategy would be flexible. Moreover, using the lidar sensors to detect the obstacle would be an essential part of this strategy. This was realized by changing the goal position, the position of the obstacle along the path, as well as its shape, from episode to episode.



(a) 2D representation (b) 3D representation

Figure 15: Representation of the random obstacle scenario

The following input data was used to train the neural network of the system:

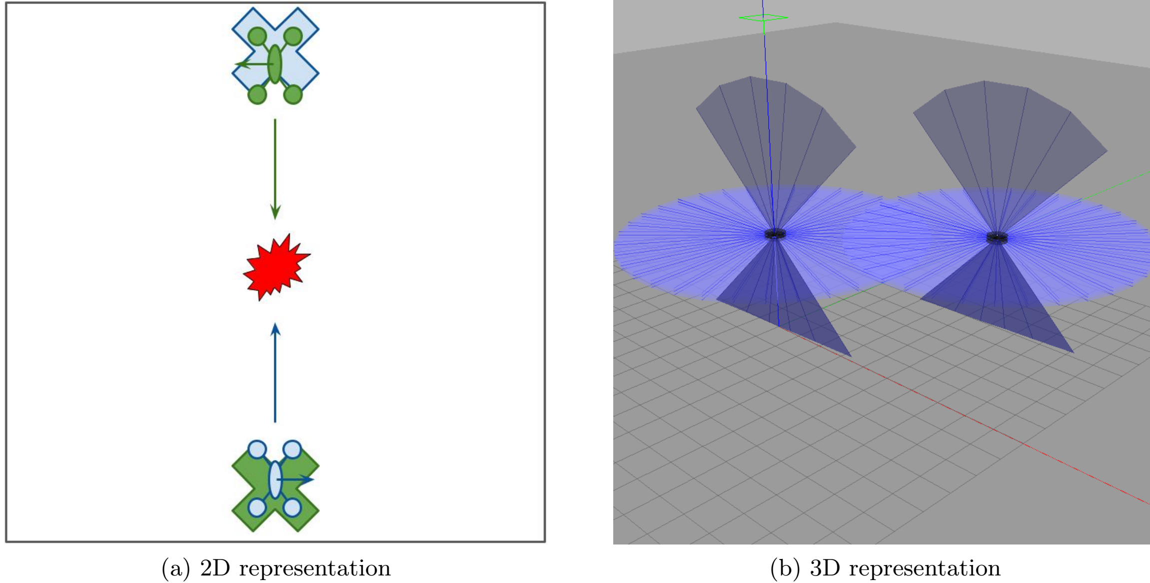
- x, y, z position (3 data points)
- goal x, y, z position (3 data points)
- distance from the straight line connecting the takeoff and the goal position (1 data point)
- middle horizontal lidar data (35 data points)
- vertical upwards lidar data (6 data points)
- vertical downwards lidar data (6 data points)

5.1.5 Two drones empty

This scenario represented the situation when two drones are on a collision course with each other, and they need to avoid crashing. This setup consisted of two drones that were in an empty space. They took off at two different positions, and their goal was the takeoff position of the other drone. If they were to go in a straight line to their goal, as they learned in the *empty random* scenario, they would collide with each other. The task of the drones here was to develop a strategy to avoid collision and safely reach their goal.

Figure 16 depicts this scenario. This setup was also used to test the incremental complexity approach of developing more complete rules by testing the initial use of the network trained in the *random goal* scenario.

Regarding the trained neural network which resulted from the learning process, the same can be said as in the case of the *static obstacle* scenario. While this scenario could be used for traffic rule formulation, problems arose when it was to be utilized as a building block in the incremental complexity approach. Since the neural network was not flexible with regard to the world, it could only be used as a basis for more complex scenarios if the starting and goal positions remained the same.



(a) 2D representation

(b) 3D representation

Figure 16: Representation of the two drones empty scenario

The following input data was used to train the neural network of the system:

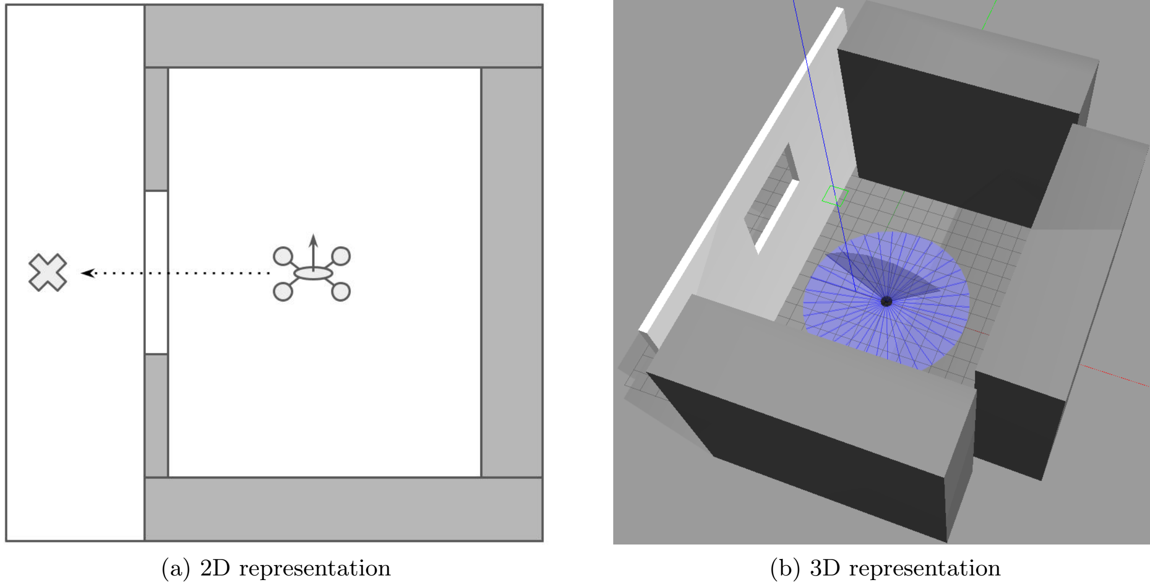
- x, y, z position translated to a common coordinate system (3 data points)
- roll, pitch, yaw orientation (3 data points)
- x, y, z directional velocity (3 data points)
- own identifier number (1 data point)
- transponder data of the other drone (position and velocity information, 6 data points)

In this implementation, the two drones used a shared neural network, for their training and action selection. This will be further discussed in Chapter 7.

5.1.6 Window

It can happen that a drone accidentally gets trapped in a space from where only a smaller opening leads out. This is the case, for example, if the drone flies into an enclosed courtyard or a field surrounded by high trees and bushes. It can even be that the drone's task is to deliver a package through an open window. This scenario aimed to represent this situation. Figure 17 shows how this is represented in the simulation environment.

This scenario was also intended to be the building scenario for the *two drones window* scenario. Since the environment remained the same from the viewpoint of the drones, the resulting neural network of this scenario could be used with the implemented incremental complexity approach even though the network encoded a static strategy.



(a) 2D representation (b) 3D representation

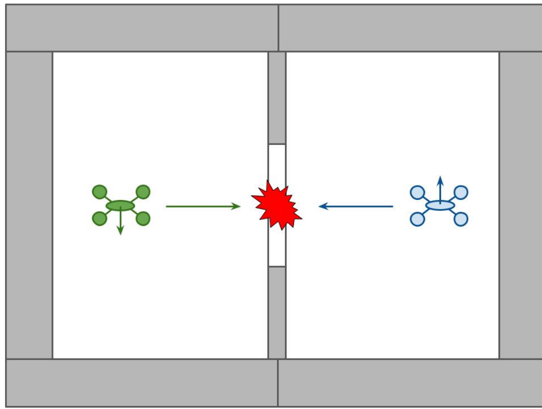
Figure 17: Representation of the window scenario

The following input data was used to train the neural network of the system:

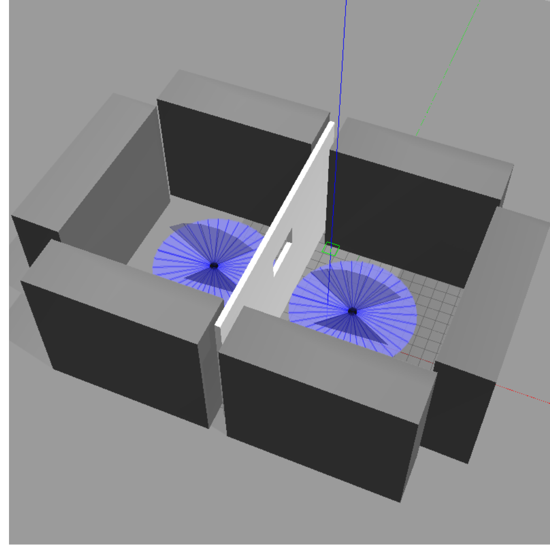
- x, y, z position (3 data points)
- x, y, z directional velocity (3 data points)
- own identifying number (1 data point)
- dummy data preparing for the use in the *two drones window* scenario (11 data points, see next section)
- middle horizontal lidar data (35 data points)
- upwards vertical lidar data (6 data points)
- downwards vertical lidar data (6 data points)

5.1.7 Two drones window

In this scenario, there were two drones located in neighbouring enclosed areas. The two areas were separated by a wall, which had a single window on it. The task of the drones was to go through the other side of the window from their own space, respectively. The most difficult aspect of this setup was that the drones needed to create a strategy for cooperation and priority handling. Figure 18 shows the created world. The tests for this scenario had not resulted in a stable learning process yet, however, two successful strategies were developed. In the implementation that was executed for these tests, the two drones utilized their identity number as input for the neural network. However, this unintentionally introduced an inherent priority into the system, weakening the results of the scenario. This will be further discussed in Chapter 7.



(a) 2D representation



(b) 3D representation

Figure 18: Representation of the two drones window scenario

The following input data was used to train the neural network of the system:

- x, y, z position translated to a common coordinate system (3 data points)
- x, y, z directional velocity (3 data points)
- own identifying number (1 data point)
- own priority (1 data point)
- other drone's priority (1 data point)
- goal x, y, z position (3 data points)
- transponder data of the other drone (position and velocity information, 6 data points)
- middle horizontal lidar data (35 data points)
- upwards vertical lidar data (6 data points)
- downwards vertical lidar data (6 data points)

The priority of the drones was intended to be used in future simulations. For the tests executed in this work, the value of these two inputs was set to be the same number.

Having described the implemented scenarios, it is also important to gain insight into the various challenges that can cause problems when executing these. These challenges will be presented in the next section.

5.2 Challenges

In this section, three challenges are described that came with obtaining results for the use case of this project using machine learning in a simulation environment. Proper care needs to be taken when dealing with these challenges, otherwise, they could significantly deteriorate the results of the simulations.

5.2.1 Interpretable results

Since the goal of the use case in this project was to obtain drone traffic rules, it was important that the results of the reinforcement learning process were interpretable. One way of obtaining results as such would be to train a system to translate its findings like this was done in [59]. Another way of obtaining interpretable results would be to design the problem itself in a way that the results would be easy to interpret. This could be done for example by simplifying the state space like in [60] or using a weight structure, like in [61].

Fortunately, in the case of this project, interpreting the resulting behaviour of the drones was not particularly difficult. The actions of the drones were logged every episode and with the necessary tools, the resulting behaviours were easily decoded. Additionally, if using the logged actions would be too complicated, a graphical visualization of the simulation could also be used.

5.2.2 Bias

Bias is the presence of effects that steer the process in some way. This bias is often harmful. In this case, the simulation is steered in a way that is not intended by its creators and can result in hazardous situations. An example of a harmful and potentially dangerous bias is described in [62]. In any learning process, care has to be taken to mitigate the effects of harmful bias and consider different aspects of it. However, bias can also be beneficial. This is the case for example if the learning is to be steered in an intended direction by a researcher [63]. Furthermore, if the machine learning process does not work as well as expected, intended bias, in other words, human heuristics, can be used to make it more efficient by driving the process in certain ways.

In the case of this thesis work, intended bias was inherent to the process. How the simulations were set up by the user, the created world and the complexity of the scenarios were all intended ways to steer the learning. In the theoretical model which was described previously in Chapter 2 and shown in Figure 3 this bias is explicitly presented by the "human heuristics" expression.

Harmful and beneficial biases are usually both present in a process. For example, if the simulations created for this thesis project are considered, the size of the workspace of the drone can limit its behaviour, steering the results in wanted and unwanted directions. If the workspace is defined too small in the vertical direction, for instance, the drone will be limited to working out strategies by moving in the horizontal plane. This is beneficial if the intended question is what kind of behaviours can be generated which involve only horizontal movements. However, it is harmful, if this restriction is not intended and its effects are not expected beforehand.

5.2.3 Simulation to reality gap

The simulation gap is an important phenomenon to be considered, especially if the results of a learning process are to be used in the real world. A simulation cannot perfectly describe reality, there will always be elements that are left out of it. Unfortunately, this may affect the usability of results that are obtained in a simulator when applied to real-life scenarios. This is the effect of the simulation gap. A good example of this, for instance, is when developing a traction control algorithm, and not including the possibility of slippery roads in the simulated worlds. This results in the algorithm most likely performing well in nice weather conditions but failing in the presence of rain or ice.

Naturally, it is difficult to assess which elements of the real world are considered significant for the goal of the learning. However, there are quite a few actions that can be taken to ease the

effect of this phenomenon. For example what the agent learns can be made more general, not focusing on smaller details. In the case of traffic rules for example this could mean focusing on overall behaviour instead of the exact inputs of the drone controller. Another popular technique, for example in the case of image recognition problems, is domain randomization. In this solution, the agent is presented with environments in which the elements that are not specific to the learning task are varied. For example, if the image of a number is to be recognized, the system is trained using images with different background colours. This method was utilized for example in [64].

In the simulation environment of this project, a virtualized real-life drone controller and simulated sensors were used to mitigate the effects of the simulation gap. There were elements of reality, however, which were not taken into account and it was important to keep these in mind. There was no airflow simulation included, for example, which left out the effects two drones could have on each other when they were in close proximity. Additionally, the effect of wind and other weather conditions were also not included. Imperfect environments, where the sensors can fail, were not simulated either.

5.3 Configuration

One of the difficulties in this project was identifying good settings for all the important parameters described in Chapter 3. While many different settings were experimented with, time constraints imposed by partly the project length and partly the speed of the simulation did not allow an exhaustive test on the parameters. Nonetheless, this section aims to provide insight into what was learned throughout the project about the different configurations of these parameters and how they influenced the learning process.

Initially, two problems were identified regarding the values that were processed as input for the neural network. The first was with regard to how the rewards were defined. At the beginning of the process, a huge negative reward (-1000000000000) was set as the penalty for crashing. However, this skewed the weights of the network so much that eventually, they all became NaNs. The appropriate solution to this was to set a penalty approximately in the range of the other rewards.

The other issue was with the lidar data input. By default, if the rays did not detect any obstacle they returned an inf value. Since this again impaired the weights of the network, it was implemented in the lidar processing node that if the value of a ray was inf it should return its maximum range - in this case, 5m - instead. These values were present to indicate that the lidar sensor had not detected any obstacle in its range.

For most scenario runs, the exploration rate was set to be 100% at the beginning of the learning process, and this was then lowered throughout the episodes. The cause for lowering the exploration rate was, as already explained in Chapter 3, to increase the chance for the drone to utilize what it has learned in its process. The exploration rate was lowered in an exponential fashion, as in the example of [30] using the function described in Equation 16.

$$\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \cdot e^{-\lambda \cdot steps} \tag{16}$$

For the λ parameter different values were used, ranging from 0.00003 to 0.00006 depending on the scenario. In the case of the *random goal* scenario, the exploration rate was lowered faster than in the case of the *static obstacle* scenario for example. This was because, in the case of a more complex scenario, the agent needed more time to explore. The slower decrease in exploration rate let the system gain more information from its environment.

As it was described previously, the incremental complexity approach was tested by using the resulting trained network of a simpler scenario as starting state of the neural network of a more complex one. For these tests, the exploration rate was set to start at a lower value than 100%. This allowed the agent to utilize what was learned in the previous scenario while still having enough chance to explore possible actions. The lowered starting value was chosen from a range between 80%-45%. The goal of a lowered exploration rate was to let the agent instantly utilize the information from the previous scenario. When the value of the exploration rate was chosen to start around the upper limit of this range, the agent was only steered towards the previously learned behaviour sometimes, while focusing more on exploration. While the selected value was at the lower end of this range, the agent was steered more strongly towards the previously learned behaviour. 80% was defined as the highest value because it was believed that significantly higher values would not allow the steering of the drone's behaviour enough to count. 45% was defined as the lowest value because in this case, the drone is doing what it has learned in a previous scenario more than half of its actions, and a significantly lower value was believed not to allow enough exploration so that new behaviours can be developed. The stronger focus on exploration was needed when the task to solve was more complex, for example in the case of the *two drones window* scenario, while the stronger focus on exploitation was needed when the task to solve was quite similar to the previously learned one, for example in case of the *two drones empty* scenario.

The number of steps per episode was set to be roughly 2-3 times the number of steps that are expected to build up a good behaviour. This was thought to provide the agent with enough possibility for exploration, while not making the length of the simulation in real-time needlessly large. For most scenarios this was a good setting, however, the *static obstacle* scenario needed significantly more steps per episode - about 5 times the number of steps building up a successful behaviour - to be able to effectively explore its environment.

The discount factor was set to be high, 0.9 or above, in order to incorporate the fact that future rewards are considered very important. This was especially needed since for many of the scenarios presented in this project an optimal behaviour includes actions which are resulting in some amount of penalty. If the future effects are not considered, the agent will actively avoid these states, thus not being able to reach its goal. This is the case for example for the *window* scenario. Going through the window forces the drone to be close to an obstacle, the solid wall, thus possibly getting penalties for it. If no future rewards were considered, the drone would avoid this. However, looking at the final outcome, getting close to the wall is a logical action.

For training the agents the following rewards and penalties were used:

- Smaller reward (+10, +20) for getting closer to the goal
- Big reward for reaching the goal (+500, +1000)
- Small penalty each step (-5, -10)
- Bigger penalty for being near an obstacle (-10, -50)
- Big penalty for being so close to an obstacle that it can be considered a crash (-100, -500)

In some of the simulations, there was no penalty specified in order to test the effect of this on the results. It was observed that the ratio of the rewards and penalties matters more than their exact values.

Bigger networks generally require smaller learning rates, while smaller systems can make use of bigger values. It is also a common strategy to start the learning with a relatively big learning rate to speed up the beginning of the process and lower it after a while. During this project lowering

the learning rate manually was also beneficial. For example, in the case of the *two drones empty* scenario a good solution could be reached if the learning was started with a learning rate of 0.01, and then decreased to 0.0005 as shown in Table 7.

Initially, the size of the experience replay was set to be in the range of 10000-20000. This was thought to contain enough number of experiences for the learning to progress correctly. However, during the executions, it was found that these values were suboptimal. In case of using experience replays this small, forgetting occurred. This phenomenon means that as the experience replay is filled up, old data is discarded from it, and the agent starts to forget what it learned from those old experiences. If this happens when the agent starts to perform well and always moves on a certain optimal path, the neural network can converge on this, while forgetting what to do if somehow getting further from this path.

In the case of the *random goal* scenario, the effect of forgetting was barely noticeable, while in the case of the *window* scenario, it was quite apparent by the presence of decreasing periods in the learning graph. Moreover, the *static obstacle* scenario did not learn at all with a too small experience replay. It was found that a value of 60000 was needed at least. The *window* scenario was also tested with this number. It was found that initially, this setting resulted in a smoother process, however, when forgetting occurred it was more catastrophic than in the case of using smaller sizes. The problem of forgetting can be mitigated in various ways. In this project, the size of the experience replay was simply increased so much that it won't be filled by when the learning stabilizes. In the case of the *window* scenario this was 120000, in the case of the *obstacle* scenario this was 150000. This can be done because the size of the stored data is not so big as to cause problems. A different method which could be used when a big amount of data cannot be preserved would be to always preserve some amount of the early exploration experience.

After some simple initial tests using MSE, MAE and Huber, MSE was found to be an appropriate cost function for the training. This was used for all the scenarios. Adam and Adamax methods were chosen as optimizers. It is difficult to predict what optimizer would work best with a certain task, and Adam is generally considered a good starting point. Adamax is known as an improvement of Adam, and this is the reason why it was also tried out. Both worked for the simulations and no substantial difference was seen between the results of the two. From this, the *static obstacle* scenario may have been an exception. In the case of this scenario, it is possible that using a different optimizer would result in a smoother learning process. Since the experience replay was set to be as high as to contain all the experiences of the drone, the incorrect choice of the optimizer is the next logical reason why the learning process could deteriorate at some point, while afterwards resuming to improve as will be shown in Figure 21. Since Adam utilizes a momentum solution, it is possible, that after the initial good progress, the optimizer overshot from the optimum, decreasing the result for some time. In future work, it is expected that a different optimizer, for example, SGD or RMSProp, would create a smoother learning process.

As the activation function for the nodes, PReLu was chosen, because compared to ReLU it has the advantage of avoiding the problem of dying nodes. Initially, the size of the neural networks was chosen to contain about 9-11 layers, as it was thought that this many layers would be necessary to capture the target function. However, this was decreased to 3 in the case of the *static obstacle* scenario, and it was found that the process still worked correctly. The networks in the scenarios of this project were not as sensitive to the layer size, however, a bigger network usually requires a smaller learning rate. In order to be able to use a bigger learning rate and make the process faster it is beneficial to use a smaller network structure.

For the scenarios tackled in this project, a target alignment number between 60-160 was used,

the usual value being around 100. The value for the target alignment period came partly from examining other works, such as [30], and partly from experimentation. A too low number can cause the system to be unstable, while a too high number slows down the learning. In a failed test, where the target network was never aligned with the continuously learning network, it was also shown that this error can cause the system to diverge.

For most simulations, batch sizes between 300-500 were used. These were chosen based on [65] and [66]. However, the *static obstacle* scenario was unable to learn a good behaviour with batch sizes this big. After some experimentation, it was found that for this scenario a batch size of 64 was needed. A lower number (32) and higher numbers (128, 400, 600) were also tested but resulted in unsuccessful learning processes. After this observation was made, the *random goal* and *window* scenarios were also tried out with batch sizes of 64. It was found that while they were able to provide a stable learning process, the decreased batch size slowed down their learning - more episodes were needed to stabilize than with bigger batch sizes. Fortunately, they were not overly sensitive to this parameter; while the decrease was noticeable, it was not drastic. The reason why a smaller batch size could be beneficial is that the neural network is trained using a smaller amount of data at once, and in one step it is able to better adapt itself to them. The *random goal* and the *window* scenarios however may not require such fine resolution, and by training on smaller amounts of data at once, the learning is somewhat slowed down.

As will be seen from the result of the scenarios, adding irrelevant data to the state does not necessarily break the effectiveness of the learning. In the case of the *random goal* scenario, for example, the horizontal lidar ray data was included in the state, however, since the learning was performed in an empty space, this provided no useful information. Nonetheless, the agent managed to learn a good behaviour. This phenomenon is useful when applying the incremental complexity approach. When the neural network of a scenario is to be utilized for the neural network of a different scenario, it is important that the base scenario should include the same data points as the new one.

While inspecting the effect of these parameters it is important to note that they are not completely independent from each other. If the value of one is changed care should be taken to examine whether other parameters should be changed also. For example, as already described, if the number of layers is increased, a lower learning rate could be needed.

In this chapter, the scenarios created for the drone traffic rule generation were described. Three challenges that come from using machine learning in this simulation environment were also presented. Finally, the process for finding appropriate values for the parameters explained in Chapter 3 was shown. The next chapter will present the results of the project.

6 Results

This chapter presents the results of the thesis project. These are two-fold. First, the results from executing the created scenarios in the implemented simulation environment will be shown. Next, the results regarding the simulation environment itself will be detailed.

6.1 Reinforcement Learning use case

For each scenario, many trials were needed to reach a successful learning process, which resulted in carrying out several thousand episodes for every one of them. Each of these episodes can be considered a full-scale simulation in which a drone tries to accomplish its mission. Due to the reasons which were described earlier in Chapter 4 executing an episode took several seconds, in extreme cases even a few minutes. This resulted in a scenario attempt to take several hours, in the worst case even one or two days. Eventually, thousands of hours were spent on running the simulations of this work.

In the following sections of this chapter, the results of the scenarios summarized in Table 3 will be presented. The resulting learning graphs of the learning process of the scenario and a table summarizing its most important parameters are shown. Additionally, an example of a developed behaviour is displayed for every scenario, represented by a series of actions. The examples shown were extracted from the log files in which the actions of the drone were saved for every episode. The episodes chosen for extraction are showing the behaviour on which the system stabilized. As the learning of the system progressed, many episodes presented this correct behaviour. From these episodes, the ones with the cleanest series of actions were selected, meaning that they were as free of interfering random actions as possible. The reason behind this choice was that it was easier to extract and formulate rules based on a relatively cleaner series of actions.

On the graphs which show the learning processes of the scenarios the x-axis displays the episodes of the process and the y-axis shows the cumulative reward reached in an episode. In order to show the trend of the learning process better, the cumulative rewards were also averaged out by a number of episodes.

These graphs represent the result of just one or two runs for each scenario. To correctly evaluate a reinforcement learning process and make statistically significant observations about it, several hundreds of repetitions would be needed. These then could be averaged and their results examined. This would be the correct method to show results on learning plots. However, in the case of this project, it was not possible to do so, due to the simulations taking a long amount of time to execute. Nevertheless, even though the graphs cannot be used for assessing the learning process itself, they can be applied - and were successfully used - as showcases for the environment and the implemented DQN algorithm. Additionally, although statistically significant observations cannot be concluded based on these results, the process itself is appropriate for generating behaviours for obtaining drone traffic rules. Even though the representativity of the learning process is mathematically not proven, the successfully found solutions are there and can be used for further engineering purposes.

6.2 Empty space

Figure 19 shows the learning process of this scenario run. It can be seen that the setup is working and the algorithm found a good solution relatively fast. The parameters for this scenario are

shown in Table 4.

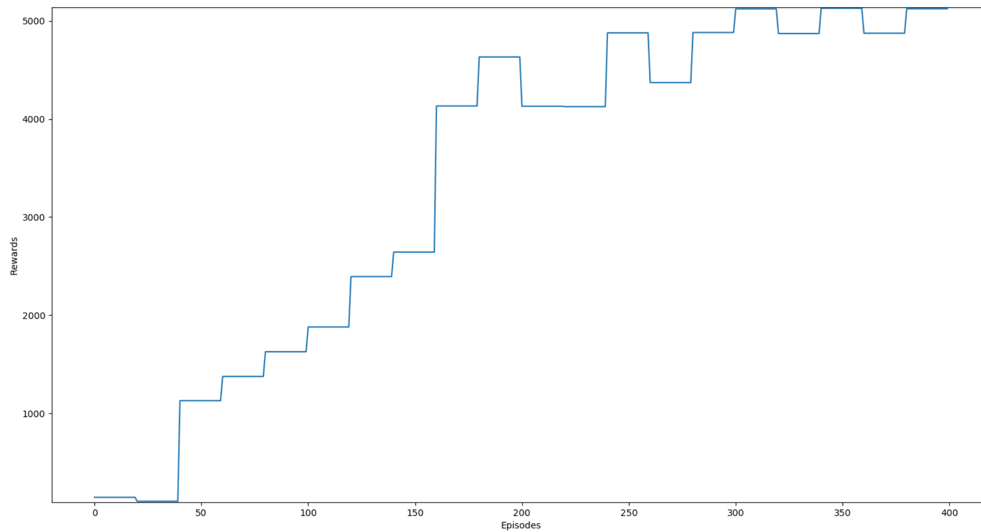


Figure 19: Result graph of the empty space scenario

	E2
gamma	0.8
epsilon max	1.0
epsilon min	0.0
learning rate	0.01
step per episode	20
batch size	300
target alignment period	80
number of layers	4

Table 4: Parameters used for the empty space scenario

Example of the developed behaviour as present in the logfiles:

3 3 6 6 6 6 6 6 6

Referring back to Table 2, action 3 meant going towards north, while action 6 meant going towards the goal. Since the goal area was directly north from the area where the drone took off, this series of actions moved the drone straight to its goal.

6.3 Random goal

Figure 20 shows the learning curve made of this scenario, while Table 5 summarizes the most important parameters used for the simulations.

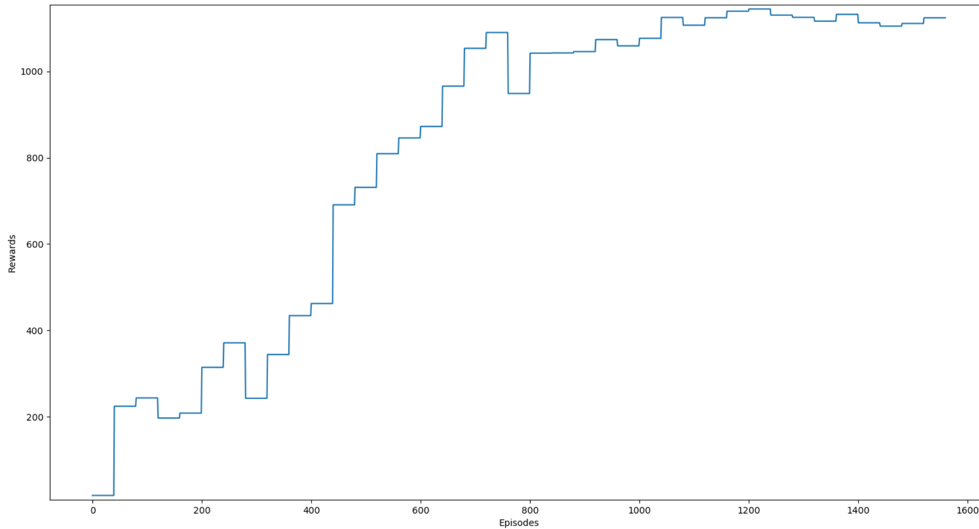


Figure 20: Result graph of the random goal scenario

	ER3	**ER5**
gamma	0.9	0.9
epsilon max	1.0	1.0
epsilon min	0.0	0.0
learning rate	0.00077	0.00077
step per episode	35	35
batch size	400	400
target alignment period	100	100
number of layers	10	10

Table 5: Parameters of the random goal scenario

Example of the developed behaviour as present in the logfiles:

6 6 6 6 6 6 6 6

In this case, the drone only used its *move closer to the goal* action. It could do so safely since there were no dangers ahead.

6.4 Obstacle

Figure 21 shows the learning curve made of this scenario, while Table 6 presents the most important parameters used.

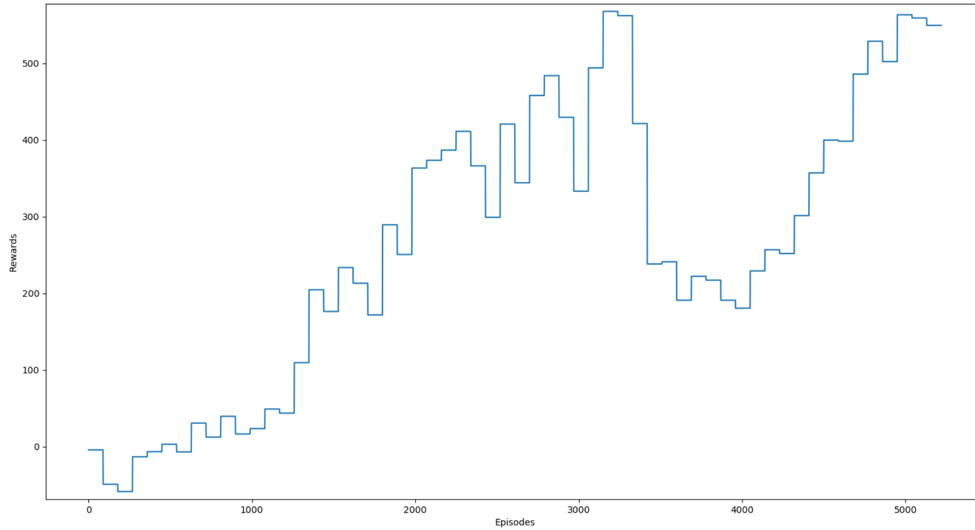


Figure 21: Result graph of the static obstacle scenario

	O7
gamma	0.999
epsilon max	1.0
epsilon min	0.0
learning rate start	0.0001
learning rate end	0.0000001
step per episode	45
batch size	64
target alignment period	100
number of layers	4
experience replay size	150000

Table 6: Parameters of the static obstacle scenario

Example of the developed behaviour as present in the logfiles:

3 5 5 3 3 3 3 4 4

As discussed before, action 3 meant flying north, action 5 meant flying upwards, and action 4 meant flying downwards. In the created world, the goal was positioned directly north from the takeoff position of the drone, with the obstacle located directly between the two. In the course of this developed behaviour, the drone first moved one step closer to its goal. Then, it increased

its altitude for two steps. Afterwards, it continued towards its goal, flying above the obstacle. Finally, the drone descended on the other side of the obstacle, reaching its goal.

6.5 Two drones empty

The result graph of the learning process of this scenario is shown in Figure 22, while the parameters used are shown in Table 7.

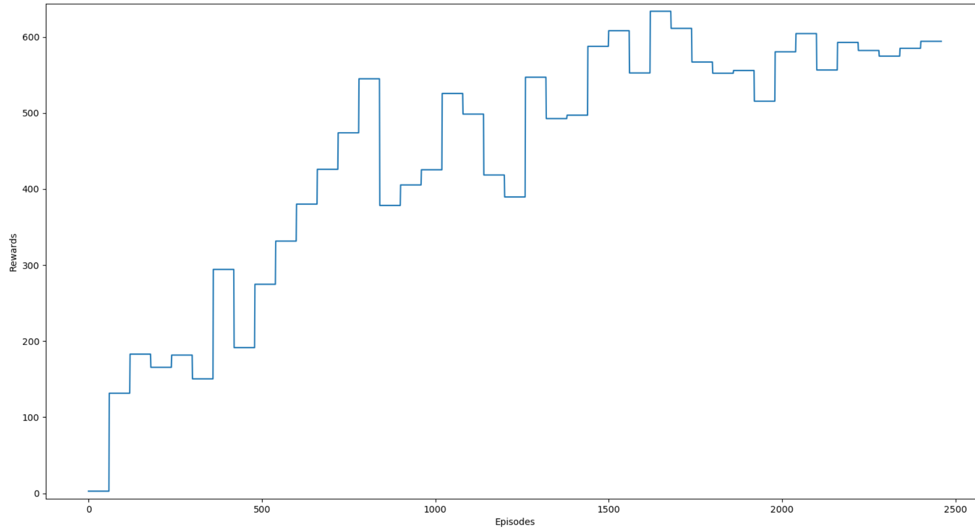


Figure 22: Result graph of the two drones empty scenario

	TE10
gamma	0.9
epsilon max	1.0
epsilon min	0.0
learning rate start	0.01
learning rate end	0.0005
step per episode	25
batch size	300
target alignment period	50(100)
number of layers	11

Table 7: Parameters of the two drones empty scenario

Two examples of the developed behaviour from two different successful runs as present in the logfiles:

drone 0: 6 3 0 0 6 0 6 6 6
drone 1: 6 0 6 0 0 0 6 6 6 6

drone 0: 6 2 2 2 2 6 6 6 6 6
drone 1: 6 6 2 6 6 6 6

Based on Table 2, action 0 represented moving eastward, action 2 represented moving south and action 6 represented moving towards the goal area. As described in Chapter 4 the meaning of actions 0 and 1, as well as actions 2 and 3 were exchanged for one of the drones. Keeping this representation in mind, the presented series of actions can be translated the following way. It can be said that in the first example, the two drones avoided each other by going to their respective lefts, while in the second example, they did so by going to their respective rights.

6.6 Window

Figure 23 shows the learning curve made of this scenario, while Table 8 summarizes the parameters used for the simulation.

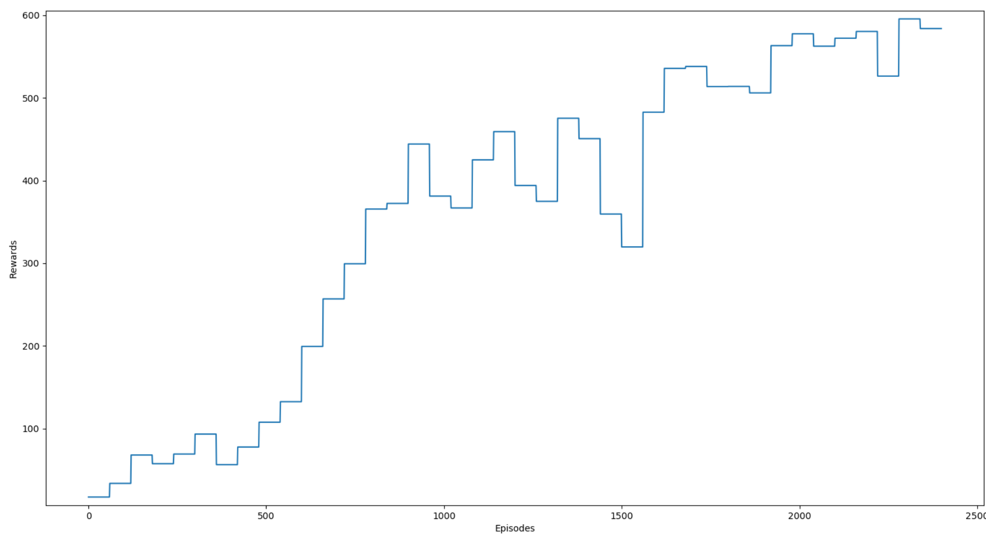


Figure 23: Result graph of the window scenario

	W12
gamma	0.99
epsilon max	1.0
epsilon min	0.0
learning rate	0.0005
step per episode	30
batch size	350
target alignment period	160
number of layers	9
experience replay size	120000

Table 8: Parameters of the window scenario

Example of the developed behaviour as present in the logfiles:

5 5 5 5 3 3 3 3 3 3 3

Just as before, action 3 meant flying north, while action 5 meant increasing altitude. Since the wall with the window was north of the drone, and the window is centred horizontally along this wall, this series of actions meant that the drone first rose to the height of the window and then went straight through it.

6.7 Two drones window

Two examples of good behaviour as present in the logfiles:

drone 0: 5 7 5 5 3 5 3 5 6 3 6 2 3 7 6 3 3 3
drone 1: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 6 6 6 5 6 5 6 6 6 3 3

drone 0: 5 5 5 5 5 6 3 3 5 6 1 6 2 5 6 2 6 6 7 6 6 6 3
drone 1: 5 5 5 5 3 6 6 6 6 2 2 2 6 6 3 2 2 6 5 2 3 2 5 2 2 6 6 6 6 6 6 3 3

Regarding the axes of the two drones, the same applies to this scenario as to the *two drones empty* scenario. As before, action 2 meant moving south, and action 3 meant moving north for one of the drones. For the other drone, the two actions were exchanged. Action 5 represented increasing the altitude of the drone. Action 6 meant moving towards a defined goal position, and action 7 meant hovering in place. For both drones, executing action 3 moved them straight towards the wall on which the window was located.

In the first strategy, one drone waited, hovering in place, while the other one flew through the window. After it reached the other side, the one that was waiting also flew through. In the other solution, first, both drones flew towards the window. Then, one of them flew through it, while the other one flew some distance away from it, and then started to oscillate back and forth. Finally, when the one going first reached the other side, the one waiting also flew through.

6.7.1 Extracted rules

As the goal of the use case was to generate drone traffic rules, it was also examined whether such rules could be created from the scenarios that were executed. Based on the results of the scenarios described above the following set of basic rules was created. Naturally, the created rules are subjective, and there are more ways to interpret the generated behaviours. The rule formulation process will be further discussed in Chapter 7.

0. Don't fly too fast toward the ground.
1. Use the pre-built navigation if you can do so safely.
2. If there is a static obstacle ahead and otherwise empty space, avoid said obstacle by flying above it.
3. If two drones are on a frontal collision course with each other, they should avoid colliding by moving to their respective rights.
4. If there is a solid obstacle in the path of the drone, but with a big enough opening, the drone can go through.
5. If two drones are trying to go through an opening from opposite directions at the same time, the drone coming from the lower GPS coordinates needs to start hovering as soon as the presence and intentions of the other drone are detected. The drone with the higher GPS coordinates can go through instantly. After it went through, the drone that was waiting can also go through.

6.8 Simulation environment

The implementation of the simulation environment was presented in Chapter 4, while the use case through which its capabilities were assessed was presented in Chapter 5. The results of said use case were shown earlier in this chapter. This section presents the assessment of the capabilities of the simulation environment based on what was observed.

It was determined that the created environment with realistic elements is suitable for reinforcement learning simulations from an engineering viewpoint. This means that results for various problems can successfully be obtained by the use of this technique in an acceptable amount of time. Good results are accepted even if it is possible that they were the outcome of a lucky scenario run because the focus is on the developed behaviour itself. Just as it was shown in the use case of this project, good behaviours could be created with the use of this environment.

Unfortunately, however, this environment is not optimal for reinforcement learning research, when the goal is not only to create an acceptable result but to examine the learning process itself. For applications like these, every scenario should be executed hundreds of times. To do so with this environment, either the number of instances running parallel would need to be increased, which would be quite expensive or the time waited to obtain the results would be very long.

Such a platform could also be used for other applications besides reinforcement learning. An environment like this could be used for example testing created rules. Due to the simulation gap, this is a complex task, however, as the realism of the environment can be increased quite much, it is still possible to gain meaningful insight with its use.

Utilizing the observations made throughout this project, a verification process was carried out based on the requirements listed in Chapter 2. For easier reference Table 1 was copied here and shown as Table 9.

Use case aspect		Requirement
Incremental complexity	1	Adding new scenarios does not result in major modifications in other parts of the environment
	2	Scenarios can start their training by using the trained agent of previous scenarios
Reinforcement learning algorithm	3	Capability to quickly and efficiently reset the simulated world after every episode
	4	Robustness
	5	Scalability
	6	Simulation speed should be significantly faster than realtime
	7	Initial state of the drone can be varied within the scenario
Traffic rule generation	8	Support multiple drones in the same simulation
	9	Support for static and dynamic obstacles
	10	Rules are translatable to real life

Table 9: Requirements

Two requirements were formed with regard to the incremental complexity aspect:

1. The first one presents the need that adding new scenarios to the system should not result in major modifications of other parts of the environment. As was described in Chapter 4 the structure of the robot instance was highly modular. This means that the Task Environment could be freely changed without having to change the Robot or the Learn Environment. Due to this modular class-based structure, if a new scenario is created, only the Task Environment needs to be modified. Based on this, it can be said that this requirement is fully satisfied.

2. The other requirement originating from the incremental complexity aspect is that scenarios can start their training by using the trained agent of previous scenarios. In the case of the DQN algorithm, this trained agent is the resulting neural network. This network can be used as the starting state of another one in case the two networks have the same structure. Based on this, it can be stated that the fulfilment of this requirement depends on how the user sets up the consecutive scenarios.

Five requirements were formed with regard to the reinforcement learning algorithm aspect:

3. The environment should have the capability to efficiently reset the simulation between every episode. As it was described in Chapter 4 the resetting part of this requirement is fulfilled, however, the solution for it was less efficient than expected.

4. The simulation should be robust. The presence of the monitoring tool adds robustness to the simulation since if it breaks down due to an issue, it is automatically restarted.

5. Additionally, the simulation should also be scalable. The scalability requirement is fulfilled by its straightforward parallelization capability which is available due to the simulation being run in docker containers.

6. Another requirement was that the simulation speed should be significantly faster than real time. Unfortunately, this requirement is only partially fulfilled. The simulation speed is faster than real life, however, it is only by three times.

7. The last requirement with regard to the reinforcement learning aspect was that the initial state of the drone can be varied within the scenario. It is possible to do so from two parts of the software. As every scenario has a parameter file attached to it, the position of the drone can simply be changed among these parameters. Additionally, the position can also be varied within the scenario's Task Environment itself.

Finally, three requirements were set based on the traffic rule generation aspect:

8. The environment should support multiple drones being controlled in the same simulation. As it was explained in Chapter 4 this functionality is included in the simulation environment.
9. Additionally, it should be possible to add static and dynamic obstacles to the simulated world. To fulfil this requirement, models can be spawned into the simulation environment, and movement scripts were created for dynamic obstacles.
10. Finally, the created rules should be translatable to real life. This inherently means that the simulated worlds should have the level of realism with which this is possible. With the use of plugins in Gazebo, the level of realism can be adjusted in a wide range. What is more, the model chosen for this project has the capability to overcome certain negative obstacles, such as if the learned strategy is static, or if there was a presence of negative bias. This is due to the generated behaviours being formulated into rules by a human observer.

In this chapter, the results of scenarios implemented during the project were shown. Additionally, a verification process was carried out for the simulation environment based on the requirements set at the beginning of the work. In the next chapter, the results presented here will be further discussed.

7 Discussion

In this chapter, the results of the project will be discussed. The resulting behaviour of the scenarios and what was learned about the incremental complexity aspect will be detailed as well as the validity of the simulation environment itself.

7.1 Resulting behaviour

Six of the seven implemented scenarios resulted in behaviours that can be used for traffic rule formulation. In the *random goal* scenario the drone successfully learned to utilize its "move towards the goal" action, while in the *static obstacle* scenario the drone successfully avoids the obstacle. In the *two drones empty* scenario the two drones developed strategies for avoiding each other, while in the *window* scenario the drone was capable of moving through the window. Even the *two drones window* scenario presented two different strategies for going through a common window safely.

It is interesting, that the system encountered significantly more difficulty in case of the *static obstacle* scenario than with some of the others. Initially, it was thought that this scenario would not be more difficult than the *window* scenario for example. However, it turned out to be much more sensitive to the parameters and more complex from the viewpoint of the learning. This scenario was tried out also with an easier goal formulation; the area of space that the drone needed to reach was increased. However, this did not result in a significantly different learning process. The difficulty of this setup can be explained by the fact that in the case of this scenario, the environment needed to be more thoroughly explored than in the case of the other ones. Additionally, the size of the environment that needed to be explored was bigger. This does not necessarily mean the size of the workspace of the drone, but the amount of area that the drone needs to have a good and detailed representation of.

For the purposes of formulating traffic rules, the resulting behaviour and learning process of the *static obstacle* scenario were acceptable. A stable good behaviour was developed, even though the learning process itself might not be as stable as desired. By examining the resulting learning graph it can be seen, that this scenario presented a more unsteady learning process than the other ones shown. After an initial progress in the good direction, the curve suddenly decreased, followed by a longer period when the results of the episodes were far from optimal. After this period, the learning process resumed progressing in the preferred direction. It could happen, however, that if the process was allowed to continue for more episodes, additional oscillation phases would appear. As it was explained in Chapter 5 the cause of the process decreasing mid-learning might be originated from an incorrect choice of optimizer.

In the *two drones empty* scenario, the drones successfully developed strategies to avoid each other. In an initial setup, the movement solution which was created for the two drones when facing each other was not yet developed. In this case, moving to the west, for example, meant that both drones moved towards the same direction. This, along with the implementation that the two drones utilized the same neural network, effectively forced the system to create a solution where the drones avoid each other by vertical separation. This is an example of unwanted bias. This solution also would not be appropriate in real-life scenarios, due to the downwash of the drone which is above. After the corrected movement solution was implemented, the drones developed the strategy of avoiding each other sideways, either to the left or to the right. This is an interesting result since while the downwash of the drones was not simulated, it was observed that moving vertically was easier for the drones than moving horizontally. It is possible that if more simulations were run the vertical separation avoidance would appear again.

The *window* scenario also produced interesting findings. Initially, some problems arose from using a too thick wall, with a too narrow opening. In this case, the learning process was not successful, to the effect that instead of going through the "window" the drone learned to oscillate in place. This behaviour could be explained by the fact that because the drone was unable to reach the other side of the wall, it didn't learn that there was a series of actions that would produce higher rewards than oscillating. Going through the opening without colliding to the side would have needed multiple steps in the correct direction, and there was too little chance of doing so while the drone was randomly exploring.

In the window scenario, a conscious bias was added by limiting the vertical size of the workspace so that it would not be higher than the walls around the enclosed space. This was done so that the drone would not be able to fly over the wall and consider this an appropriate behaviour.

An interesting aspect of the *two drones window* scenario is how the prioritization solution between the two drones emerges. Unfortunately, the two solutions developed in the course of this work, are only partly usable due to how the priority was determined between the drones. As mentioned in Chapter 5, the setup was flawed, due to the presence of an unintentional bias. When two drones were used in the simulation, they were set up to use and train the same neural network. The system was implemented this way to create solutions that would be uniform to every drone. To promote the learning process, the identifier number of the drones was added to the inputs of the training. However, the use of an identifier number in the data also introduced an inherent priority into the system. In the case of the *two drones empty* scenario, this caused no problems since there was no need for priority handling to solve their task successfully. However, in the case of *two drones window*, a fundamental part of the solution was to devise a method for priority handling, and introducing this by hand steered the learning in an unwanted way. Removing the identifier number from the data would solve this issue. Future work should focus on this priority determination task.

Chapter 6 also presents the extracted rules based on the behaviour of the drones. These show that the task of distributed drone traffic rule generation is possible with the created simulation environment and the devised model. When multiple good behaviours are developed, choosing from them is up to the one who formulates the rules. In the case of the *two drones empty* scenario the choice for avoiding each other on the right side was made because, in The Netherlands, road traffic participants avoid each other on the right, which would make this solution feel more familiar to pilots. From the two behaviours of the *two drones window* scenario the one deemed safer was selected. Waiting further from the window presents less danger of colliding than going close and then moving farther. Additionally, the problem of unintentional bias regarding the priority was tackled by selecting a basis for priority which is independent of the added identification number. This is another example where human heuristics provide benefits to the process.

Besides evaluating the resulting behaviour, logging the actions of the drone and decoding them with the list in Table 2 was also quite useful in the process of setting up the scenarios and figuring out good parameters for them. It could be examined how the drones behaved without having to slow down the simulation speed by turning on the GUI for the system. It was also traceable how the behaviour changed throughout the episodes. The state of the learning could also be determined; whether the drone was still randomly exploring, converged on oscillation, or was close to following a good behaviour and only needed some time to find the optimal solution.

Finally, it is interesting, that in some cases, the system presented a different solution than the creator of the scenario would have thought of. For example, in the *static obstacle* scenario, a

horizontal avoidance behaviour was expected. However, the drone developed a behaviour to fly above the obstacle, not around it. Additionally, in the case of the *two drones empty* and *two drones window* scenarios, multiple good solutions were created.

7.2 Incremental complexity

The benefits of incremental complexity were assessed by using the resulting trained network of a simpler scenario as starting state of the neural network of a more complex one. The simpler scenario in these tests was the *random goal*, while the more complex one was the *two drones empty* scenario. As already described in Chapter 5 in these cases, the initial exploration rate was set to be lower than 100% to allow the previously trained neural network to steer the process. What was found in testing this is that while it is possible to use a pre-trained network as a starting point, it does not speed up the learning as significantly as expected. The system found a good behaviour in approximately the same amount of episodes with and without using a pre-trained network. The lack of effect is possibly due to the simplicity of the scenarios used here. It is possible that using this method in a more complex setup would result in bigger differences with regard to how fast a good behaviour is learned. As shown in Table 3, the *window* scenario was also used for the *two drones window* scenario. However, since the *two drones window* process has not yet resulted in a stable learning process, no definite observations can be concluded based on these two.

Using this method also presents a few difficulties. As mentioned before, in most cases the neural network needs to learn a flexible strategy, and this makes creating and executing scenarios more troublesome. Additionally, the neural network structure of the two scenarios needs to match, and if this is not the case, the less complex one needs to be rerun with the structure that is to be used with the more complex scenario. This can occur, for example, if additional state inputs are needed for the new scenario. Luckily, it was found that the learning process was not sensitive to unnecessary input data. An additional problem was that in some cases using a pre-trained network created an initial loss so high, that the system was unable to lower it to realistic values. Moreover, it can occur that the two networks require a different structure to learn properly. If the requirements are strict and different, then the neural network of one cannot be used in the other.

Nonetheless, it is believed that in the case of more complex scenarios, this approach could be a useful method to speed up the learning process. However, this would have to be tested once more complex scenarios are executed.

7.3 Simulation environment

In Chapter 6 a verification process of the simulation environment was carried out based on the requirements that were defined in Chapter 2. These requirements were categorized based on the use case aspect which they originate from. The three aspects were the incremental complexity approach, the reinforcement learning algorithm and the traffic rule generation.

It was shown that one of the requirements for the incremental complexity approach was fully satisfied, while the fulfilment of the other depended on how the user set up the simulation. Based on this it can be said, that the simulation environment is utilizable for this approach if the user has all the information on how to apply it.

Most of the requirements for the reinforcement learning use case aspect were satisfied, however, the two most important ones encountered difficulties. As was shown in Chapter 4 resetting the environment between every episode was solved, but it was more difficult than originally expected, and the solution slowed down the simulation itself. What is more, the simulation speed could only be increased up to three times faster than real-life speed. These issues all originated from the realism aspect of the system. Due to these, while this highly realistic simulation environment can be used for obtaining results from reinforcement learning processes, it makes it impractical for reinforcement learning research.

The three requirements set for the traffic rule generation use case were all fulfilled. The created simulation environment is well-fitted for this use case, and due to the possibility to add external plugins to Gazebo, it will be functional also for more complex scenarios in future work.

In this chapter, the results of the simulations and various findings of the work were discussed. This includes the detected behaviour for the executed scenarios, the results of testing the incremental complexity approach and the validation of the simulation environment. The next chapter concludes this thesis report.

8 Conclusion

The capabilities of simulation environments keep improving, and their use becomes more widespread every year. The goals of this project were the following. A simulation environment with a pragmatically minimal simulation gap was created using the software packages that Almende and DoBots have experience with. Afterwards, the capabilities of this platform were assessed through the distributed drone traffic rule generation use case. For this use case, a model was devised, which also incorporated the incremental complexity approach of Almende as well as machine learning techniques. Furthermore, it was assessed whether approaching the problem in an incremental complexity way - by using the results of simple scenarios to speed up the process of learning more complex ones - was providing the expected benefits.

Since the resulting simulation environment and the created scenarios could be utilized for Almende and DoBots' other projects, an additional requirement was that they were modular and integrable into their other works.

In the course of this project, a simulation environment was created, with the main building blocks of Gazebo Classic, PX4, ROS2 and the iris drone model. The code was implemented in a way that it would be easy to integrate the created scenarios into Almende and DoBots' other projects. A model was also devised for distributed drone traffic rule generation. This model used reinforcement learning for the autonomous development of safe behaviours. After generating these, the behaviours were manually extracted and formulated into traffic rules by a human observer. It was investigated whether a basic set of drone traffic rules could be formed using this setup and it was found that it was possible, though as the complexity of the scenarios increased, they became more sensitive to the parameters of the learning.

It can be said, that the platform with realistic aspects is usable for the engineering side of machine learning. It was capable to generate safe behaviours to the various traffic scenarios that were presented to it, and a human observer was able to formulate traffic rules based on these. Unfortunately, however, the created simulation environment is not practical for scientific research in machine learning applications.

It was also investigated whether an incremental complexity approach was beneficial for the process. This was carried out by using the trained neural network of a simpler scenario as the starting neural network of a more complex setup. With the relatively simple scenarios used in this project, this method had not resulted in a significant advantage. However, it is possible that in the case of more complicated ones, it will provide faster learning. The challenges of this method were also determined, namely that the base neural network needs to include a flexible strategy, copying the neural network can create initial losses too high for effective convergence, more complex scenarios might require different inputs or a different configuration is needed for the neural network.

In future work, the simulation environment could be further developed, and more complex scenarios could be run and tested with it to obtain more sophisticated rules.

References

[1] Open Robotics. URL: https://classic.gazebosim.org/. (accessed: 04.06.2023.)

[2] *ADACORSA*. URL: https://adacorsa.eu/. (accessed: 04.06.2023).

[3] URL: https://px4.io/. (accessed: 04.06.2023.)

[4] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[5] Jean Philippe Aurambout, Konstantinos Gkoumas, and Biagio Ciuffo. "Last mile delivery by drones: an estimation of viable market potential and access to citizens across European cities". In: *European Transport Research Review* 11 (1 Dec. 2019). ISSN: 18668887. DOI: 10.1186/s12544-019-0368-2.

[6] Valerie Homier et al. "Drone versus ground delivery of simulated blood products to an urban trauma center: The Montreal Medi-Drone pilot study". In: *The journal of trauma and acute care surgery* 90 (3 Mar. 2021), pp. 515–521. ISSN: 21630763. DOI: 10.1097/TA.0000000000002961.

[7] Enrico Natalizio et al. "Take the Field from Your Smartphone: Leveraging UAVs for Event Filming". In: *IEEE Transactions on Mobile Computing* 19 (8 Aug. 2020), pp. 1971–1983. ISSN: 15580660. DOI: 10.1109/TMC.2019.2917176.

[8] Muhammad Arsalan Khan et al. "UAV-Based Traffic Analysis: A Universal Guiding Framework Based on Literature Survey". In: vol. 22. Elsevier B.V., 2017, pp. 541–550. DOI: 10.1016/j.trpro.2017.03.043.

[9] *European Union Aviation Safety Agency*. URL: https://www.easa.europa.eu/. (accessed: 04.06.2023.)

[10] *Inspectie Leefomgeving en Transport*. URL: https://www.ilent.nl/onderwerpen/drones. (accessed: 04.06.2023.)

[11] *EASA Civil drones (unmanned aircraft)*. URL: https://www.easa.europa.eu/domains/civil-drones. (accessed: 04.06.2023).

[12] Government of the Netherlands. *Rules for the commercial use of drones*. URL: https://www.government.nl/topics/drone/rules-pertaining-to-the-commercial-use-of-drones. (accessed: 04.06.2023.)

[13] Government of the Netherlands. *Rules for the recreational use of drones*. URL: https://www.government.nl/topics/drone/rules-pertaining-to-recreational-use-of-drones. (accessed: 04.06.2023.)

[14] Cristina Barrado et al. "U-space concept of operations: A key enabler for opening airspace to emerging low-altitude operations". In: *Aerospace* 7 (3 2020). ISSN: 22264310. DOI: 10.3390/aerospace7030024.

[15] *SESAR*. URL: https://www.sesarju.eu/. (accessed: 04.06.2023.)

[16] *Metropolis 2*. URL: https://metropolis2.eu/. (accessed: 04.06.2023.)

[17] Amjed Al-Mousa et al. "UTSim: A framework and simulator for UAV air traffic integration, control, and communication". In: *International Journal of Advanced Robotic Systems* 16 (5 Sept. 2019). ISSN: 17298814. DOI: 10.1177/1729881419870937.

[18] Thomas Dubot and Antoine Joulia. "Towards U-space conflict management services based on 4D protection bubbles". In: *AIAA AVIATION 2021 FORUM* (2021).

[19] Abdulrahman Alharbi et al. "Rule-based conflict management for unmanned traffic management scenarios". In: vol. 2020-October. Institute of Electrical and Electronics Engineers Inc., Oct. 2020. ISBN: 9781728198255. DOI: 10.1109/DASC50938.2020.9256690.

[20] Hao Xie et al. “Connectivity-Aware 3D UAV Path Design with Deep Reinforcement Learning”. In: *IEEE Transactions on Vehicular Technology* 70 (12 Dec. 2021), pp. 13022–13034. ISSN: 19399359. DOI: 10.1109/TVT.2021.3121747.

[21] Moad Idrissi, Mohammad Salami, and Fawaz Annaz. “A Review of Quadrotor Unmanned Aerial Vehicles: Applications, Architectural Design and Control Algorithms”. In: *Journal of Intelligent Robotic Systems* 104 (2 Feb. 2022). ISSN: 0921-0296. DOI: 10.1007/s10846-021-01527-7.

[22] United States. Department of the Army. *Fundamentals of fixed and rotary wing aerodynamics*. Fundamentals of fixed and rotary wing aerodynamics Part 1. U.S. Department of the Army, 1984. URL: https://books.google.nl/books?id=G5GPI78mIrwC. (accessed: 04.06.2023.)

[23] FAA U.S. Department of Transportation. *Glider Flying Handbook*. https://www.faa.gov/sites/faa.gov/files/regulations_policies/handbooks_manuals/aviation/glider_handbook/faa-h-8083-13a.pdf(accessed: 04.06.2023.) FAA government handbooks, 2013.

[24] Richard S Sutton and Andrew G Barto. *Reinforcement learning : an introduction*. Cambridge, Massachusetts: The MIT Press, 2018. ISBN: 9780262352703, 0262352702.

[25] URL: https://nl.mathworks.com/discovery/reinforcement-learning.html. (accessed: 04.06.2023.)

[26] Aysegul Takimoglu. *Reinforcement Learning: What it is, how it works, benefits applications*. URL: https://research.aimultiple.com/reinforcement-learning/. (accessed: 04.06.2023.)

[27] FY Osisanwo et al. “Supervised machine learning algorithms: classification and comparison”. In: *International Journal of Computer Trends and Technology (IJCTT)* 48.3 (2017), pp. 128–138.

[28] Mohamed Alloghani et al. “A systematic review on supervised and unsupervised machine learning algorithms for data science”. In: *Supervised and unsupervised learning for data science* (2020), pp. 3–21.

[29] Tengteng Zhang and Hongwei Mo. *Reinforcement learning for robot research: A comprehensive review and open issues*. 2021. DOI: 10.1177/17298814211007305.

[30] Jaromír Janisch. *Let’s make a DQN*. 2016. URL: https://jaromiru.com/2016/09/27/lets-make-a-dqn-theory/. (accessed: 04.06.2023).

[31] Fuxiao Tan, Pengfei Yan, and Xinping Guan. “Deep reinforcement learning: from Q-learning to deep Q-learning”. In: *Neural Information Processing: 24th International Conference, ICONIP 2017, Guangzhou, China, November 14–18, 2017, Proceedings, Part IV 24*. Springer. 2017, pp. 475–483.

[32] Hado Van Hasselt, Arthur Guez, and David Silver. “Deep reinforcement learning with double q-learning”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[33] Qi Wang et al. “A Comprehensive Survey of Loss Functions in Machine Learning”. In: *Annals of Data Science* 9 (2022), pp. 187–212. DOI: 10.1007/s40745-020-00253-5.

[34] Aryan Jadon, Avinash Patil, and Shruti Jadon. “A Comprehensive Survey of Regression Based Loss Functions for Time Series Forecasting”. In: *arXiv preprint arXiv:2211.02989* (2022).

[35] Léon Bottou et al. “Stochastic gradient learning in neural networks”. In: *Proceedings of Neuro-Nımes* 91.8 (1991), p. 12.

[36] Tijmen Tieleman and Geoffrey Hinton. “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude”. In: *COURSERA: Neural Networks for Machine Learning* 4 (2012).

[37] Diederik P. Kingma and Jimmy Lei Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[38] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. “Activation functions in neural networks”. In: *Towards Data Sci* 6.12 (2017), pp. 310–316.

[39] Nitish Shirish Keskar et al. “On large-batch training for deep learning: Generalization gap and sharp minima”. In: *arXiv preprint arXiv:1609.04836* (2016).

[40] URL: https://github.com/erlerobot/gym-gazebo. (accessed: 04.06.2023.)

[41] URL: https://github.com/openai/gym. (accessed: 04.06.2023.)

[42] URL: https://www.gymlibrary.dev/environments/classic_control/cart_pole/. (accessed: 04.06.2023.)

[43] URL: https://github.com/DLR-RM/stable-baselines3. (accessed: 04.06.2023.)

[44] URL: https://www.gymlibrary.dev/environments/box2d/lunar_lander/. (accessed: 04.06.2023.)

[45] URL: https://wiki.ros.org/openai_ros. (accessed: 04.06.2023.)

[46] URL: https://www.gnu.org/licenses/lgpl-3.0.en.html. (accessed: 04.06.2023.)

[47] Karelics team. URL: https://karelics.fi/using-gazebo-for-reinforcement-learning/. (accessed: 04.06.2023.)

[48] URL: https://www.gymlibrary.dev/. (accessed: 04.06.2023.)

[49] TheConstructCore. URL: https://bitbucket.org/theconstructcore/drone_training/src/master/. (accessed: 04.06.2023.)

[50] URL: https://answers.gazebosim.org//question/24982/delete_model-hangs-for-several-mins-after-repeated-additionsdeletions-of-a-sdf-model-which-sometimes-entirely-vanishes-from-the-scene-too-in-gazebo/. (accessed: 04.06.2023.)

[51] URL: https://github.com/gazebosim/gazebo-classic/issues/2724. (accessed: 04.06.2023.)

[52] URL: https://docs.ros.org/en/foxy/Tutorials.html. (accessed: 04.06.2023.)

[53] URL: https://docs.ros.org/en/foxy/The-ROS2-Project/Contributing/Migration-Guide.html. (accessed: 04.06.2023.)

[54] URL: https://docs.px4.io/v1.13/en/ros/ros2_offboard_control.html. (accessed: 04.06.2023.)

[55] URL: https://discuss.px4.io/t/offboard-control-using-ros2-how-to-achieve-velocity-control/21875. (accessed: 04.06.2023.)

[56] Rubik’s code. “Guide to Reinforcement Learning with Python and TensorFlow”. In: (2021). URL: https://rubikscode.net/2021/07/13/deep-q-learning-with-python-and-tensorflow-2-0/. (accessed: 04.06.2023.)

[57] URL: https://github.com/mswang12/minDQN/blob/main/minDQN.py. (accessed: 04.06.2023.)

[58] Eugenia Anello. “Deep Q-network with Pytorch and Gym to solve the Acrobot game”. In: (2021). URL: https://towardsdatascience.com/deep-q-network-with-pytorch-and-gym-to-solve-acrobot-game-d677836bda9b. (accessed: 04.06.2023.)

[59] Andrea F. Daniele, Mohit Bansal, and Matthew R. Walter. “Navigational Instruction Generation as Inverse Reinforcement Learning with Neural Machine Translation”. In: (Oct. 2016). URL: http://arxiv.org/abs/1610.03164. (accessed: 04.06.2023.)

[60] W. J.M. Probert et al. “Context matters: Using reinforcement learning to develop human-readable, state-dependent outbreak response policies”. In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 374 (1776 2019). ISSN: 14712970. DOI: 10.1098/rstb.2018.0277.

[61] Ioana Bica et al. “Learning "What-if" Explanations for Sequential Decision-Making”. In: (July 2020). URL: http://arxiv.org/abs/2007.13531. (accessed: 04.06.2023.)

[62] Daniel James Fuchs. *The Dangers of Human-Like Bias in Machine-Learning Algorithms*. 2018. URL: https://scholarsmine.mst.edu/peer2peer. (accessed: 04.06.2023.)

[63] Thomas Hellström, Virginia Dignum, and Suna Bensch. “Bias in Machine Learning – What is it Good for?” In: (Apr. 2020). URL: http://arxiv.org/abs/2004.00686. (accessed: 04.06.2023.)

[64] Antonio Loquercio et al. “Deep Drone Racing: From Simulation to Reality with Domain Randomization”. In: *IEEE Transactions on Robotics* 36 (1 Feb. 2020), pp. 1–14. ISSN: 19410468. DOI: 10.1109/TRO.2019.2942989.

[65] URL: https://ai.stackexchange.com/questions/23254/is-there-a-logical-method-of-deducing-an-optimal-batch-size-when-training-a-deep. (accessed: 04.06.2023.)

[66] Adam Stooke and Pieter Abbeel. “Accelerated methods for deep reinforcement learning”. In: *arXiv preprint arXiv:1803.02811* (2018).