אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

Path Planning for Drones using Reinforcement Learning

Final report

Ofir Nagauker, Eyal Elazari

Department of Industrial Engineering and Management

Ben-Gurion University of the Negev

A *final project* presented for

the degree of

Bachelor of Science

Supervised by: Dr. Armin Biess, Prof. Yael Edan

Table of Contents

List of Figures

List of Tables

Abstract

In recent years, the use of autonomous vehicles such as drones has increased thanks to technological development and improvements in control systems. Today, most of the drones' navigation missions in space are based on satellite communications and appropriate peripherals (such as a camera). That is, the drone plans its trajectory to a particular destination by mapping the space around it using satellite data and avoiding obstacles by processing an image from a built-in camera. However, sometimes satellite communications are not available, whether due to a hardware issue or an area without satellite communications. Also, avoiding obstacles based on image processing becomes challenging when the obstacles are dynamic (such as a vehicle or other drone in space). Therefore, with the development of the field of artificial intelligence and machine learning, the classical control algorithms have been replaced by data-based methods.
Reinforcement learning (RL) is a branch of artificial intelligence in which the agent (drone) learns to perform a particular task by “trial and error” while interacting with the environment. The agent receives feedback from his environment through a "reward" for each action. Which it performs in a given situation (state). The agent's goal is to maximize the "cumulative reward" by learning optimal policies.

In our project we implemented a model of Deep Reinforcement Learning that combines the field of reinforcement learning with deep neural networks. The mission is a navigation mission, performed by planning a path to a specific drone destination, in a complex and multi-obstacle environment. To do this, we trained a DQN (Deep Q-Network) model in an Unreal Engine simulation environment. After the training, we tested the model's performance in the training environment and in another environment. We found that the application of the model is indeed possible in another environment.

After analyzing the results and adjusting parameters to the existing model, we implemented another DDQN (Double DQN) model. We also trained the model in the simulation environment and compared the models.

In the last phase of the project, we performed a navigation task for a multi-agent RL. The purpose of the mission is to test the effectiveness of the routes in terms of arrival times and accuracy, when there is an interaction between two agents. This type of task may form the infrastructure for a follow-up project in the field.

1| Introduction

1.1 Motivation

The purpose of this project is to research and develop algorithms in the field of RL and apply them to drones. Moreover, development of an infrastructure for multi-agent tasks, which can be used in future projects. To do so, we had to gain knowledge in the fields of RL, deep learning and robotics. This project can also be useful for several use-cases, such as a complex assembly operation involving industrial robotics or navigation tasks of autonomous cars and more. By working on this project, we gained experience in working with AI, ML, Simulation etc.

1.2 Project Objectives

The main objective of this research is to devise an efficient RL algorithm for creating an optimal collision-free path for drones operating without camera. This was achieved by training and evaluating different models in a simulation environment. As well as comparing different algorithmic approaches. The models were implemented in different scenarios to determine which model performs best in the simulation environment. Furthermore, to develop a fundamental environment for multi-agent systems.

2| Literature Review

2.1 Overview

As a result of high availability of advanced hardware and the technological development of recent decades, autonomous drones are becoming increasingly common in various industries. Particularly, quadcopters are being used for various purposes, including educational, military, and commercial. This decision is because the quadcopter's frame has a very low moment of inertia and six degrees of freedom, so it is very stable.

2.2 Drones- Dynamics & Kinematics

There are three movements that describe all possible combinations of attitude: Roll (rotation around the X axis) is obtained when the balance of rotors 2,3 and 1,4 is changed (speed increases or decreases). Pitch movement (rotation around the Y axis) is obtained when the balance of the speed of the rotors 1,2 and 3,4 is changed. Yaw (rotation about the Z axis) is obtained by a simultaneous change of speed of the pair (1,3) or (2,4).[9]

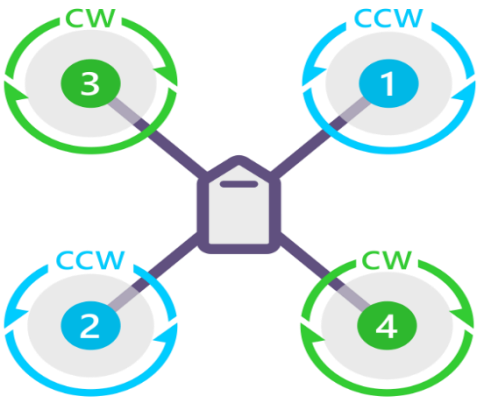


Figure 2.2.1: Rotors direction

The absolute linear position of the quadcopter is defined in the inertial frame x, y, z axes with ξ. The attitude, i.e. the angular position, is defined in the inertial frame with three Euler angles η. Pitch angle ϑ determines the rotation of the quadcopter around the y-axis. Roll angle φ determines the rotation around the x-axis and yaw angle ψ around the z-axis. Vector q contains the linear and angular position vectors.[1]

$$\boldsymbol{\xi} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad \boldsymbol{\eta} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}, \quad \boldsymbol{q} = \begin{bmatrix} \boldsymbol{\xi} \\ \boldsymbol{\eta} \end{bmatrix}$$

Figure 2.2.2: Orientation vectors

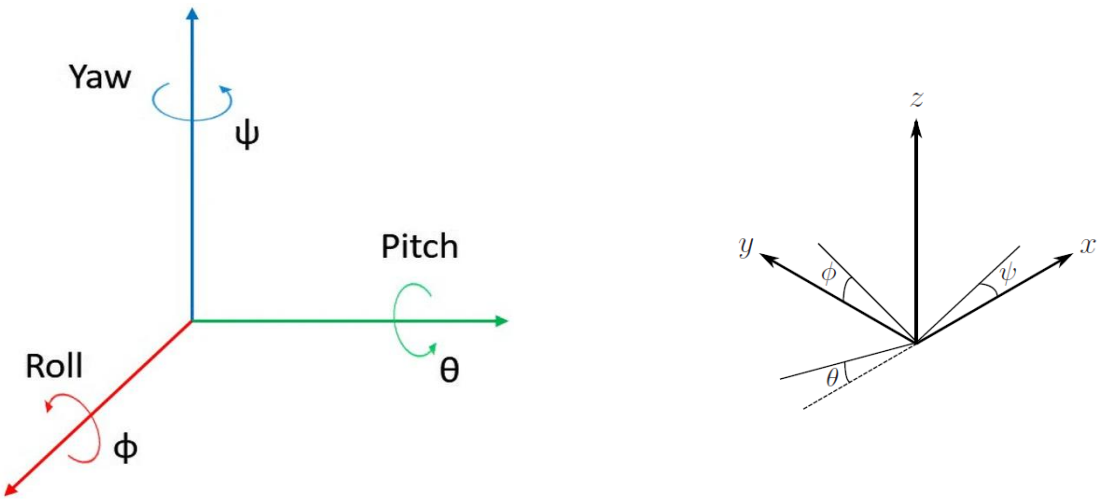


Figure 2.2.3: Quad-Copter orientation

The origin of the body frame is in the center of mass of the quadcopter. In the body frame, the linear velocities are determined by V_B and the angular velocities by V.

$$\boldsymbol{V}_B = \begin{bmatrix} v_{x,B} \\ v_{y,B} \\ v_{z,B} \end{bmatrix}, \qquad \boldsymbol{\nu} = \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

Figure 2.2.4: Velocity vectors

To stabilize the quadcopter, a PID controller is utilized. Advantages of the PID controller are the simple structure and easy implementation of the controller. The general form of the PID controller is:

$$e(t) = x_d(t) - x(t) \quad (2.2)$$

$$u(t) = K_P e(t) + K_I \int_0^t e(\tau)\, d\tau + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}, \quad (2.3)$$

in which u(t) is the control input, e(t) is the difference between the desired state $x_d(t)$ and the present state x(t), and K_P, K_I and K_D are the parameters for the proportional, integral, and derivative elements of the PID controller.[2]

The development of the kinetic model of the quadrotor includes several stages. Description of the degrees of freedom of the drone (as explained in the previous section), finding the rotation matrix from the body system to the inertial system and the rotation matrices from each rotor to the inertial system. Finally with the help of these three steps we will get the angular velocity vector of the tool in the world system.

After doubling the rotation matrix of the three axes in the following order:

$$R = QzQyQx$$

We will get the rotation matrix from the body system to the world system:

$$R = \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\phi - \sin\psi\cos\phi & \sin\psi\sin\phi + \cos\psi\sin\theta\cos\phi \\ \sin\psi\cos\theta & \cos\psi\cos\phi + \sin\psi\sin\theta\sin\phi & \sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix}$$

Once we have found the entire rotation matrix, we can relatively easily find the angular velocity vectors of the body.

After an analytical calculation of the derivative of the rotation matrix, and the multiplication of the matrices, the resulting angular velocity vector is:

$$\overline{\omega}_{body} = \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = \begin{pmatrix} \cos\psi\cos\theta\cdot\dot{\phi} - \sin\psi\cdot\dot{\theta} \\ \cos\psi\cdot\dot{\theta} + \sin\psi\cos\theta\cdot\dot{\phi} \\ \dot{\psi} - \sin\theta\cdot\dot{\phi} \end{pmatrix}$$

After connecting the members, we get the expression for the angular velocity of the rotor in the world system:

$$\overline{\omega}_{rot_i} = \begin{pmatrix} \cos\psi\cos\theta\cdot\dot{\phi} - \sin\psi\cdot\dot{\theta} + (-1)^{i+1}\dot{\alpha}_i(\sin\psi\sin\phi + \cos\psi\sin\theta\cos\phi) \\ \cos\psi\cdot\dot{\theta} + \sin\psi\cos\theta\cdot\dot{\phi} + (-1)^{i+1}\dot{\alpha}_i(\sin\psi\sin\theta\cos\phi - \cos\psi\sin\phi) \\ \dot{\psi} - \sin\theta\cdot\dot{\phi} + (-1)^{i+1}\dot{\alpha}_i\cos\theta\cos\phi \end{pmatrix}$$

Once we have found expressions for the angular velocity of the tool, the angular velocities of the rotors, the linear velocity of the tool and the forces acting on it, we can develop the equations of motion of the quadrotor.[19]

2.3 Markov Decision Processes

Markov Decision Processes (MDP) are an intuitive and fundamental formalism for *decision-theoretic planning* (DTP), reinforcement learning (RL) and other learning problems in stochastic domains. In this model, an environment is modelled as a set of states and actions can be performed to control the system's state. The goal is to control the system in such a way that some performance criterium is maximized. Many problems such as (stochastic) planning problems, learning robot control and game playing problems have successfully been modelled in terms of an MDP. In fact, MDPs have become the *de facto* standard formalism for learning sequential decision making [16].

2.4 Reinforcement Learning

A new age of Internet of Things (IoT) devices and artificial intelligence (AI) are increasingly replacing embedded systems programmed ad hoc and acting in relative isolation with autonomous sensors that acquire information, process, and understand it, and interact with the environment. In the IoT node of the future, IoT autonomously navigates an environment while simultaneously sensing, analyzing, and understanding it.

Reinforcement Learning consists of learning behaviors to achieve a specific goal through interaction. A learner or decision maker is called an "agent", while the thing they interact with, and any environment they exist in, is called an "environment". The interaction takes place at each of a sequence of discrete time steps "*t*". At each time step, the agent receives a state "S_t "from the state space "*S*" and selects an action "A_t from a set of possible actions in the action space" $A(S_t)$". Afterwards, the agent gets a numerical reward $R_{t+1} \subset$ IR from the environment Due to the previous action. The agent has now reached a new state "S_{t+1}" [16].

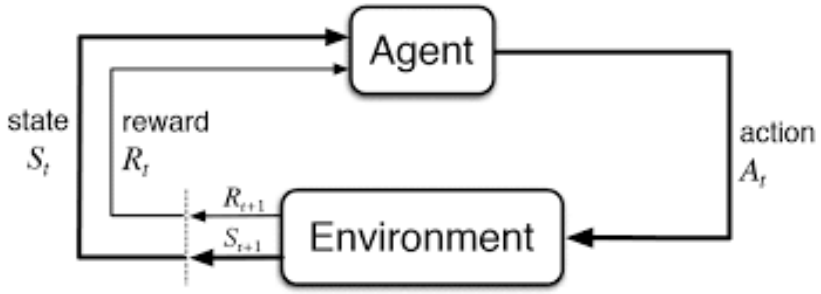


Figure 2.4: Agent-Environment Interaction

Note - every state in an environment comes about as a result of the previous state, which is itself a consequence of the previous state. In reality, storing all this information, even for short-term environments, is not feasible. As a solution, we assume that each state follows a Markov property, i.e., each state is purely based upon the previous state and its transition to the current state. [10][8][18]

2.5 Q-Learning

The Q-learning algorithm is a policy-learning algorithm that explains what action to take to the agent based on its state (Figure 3.4.1), and it doesn't require a model of the environment (i.e. model-free). To maximize profits, the agent will follow a series of actions that will eventually yield the best results. We will formalize our strategy based on this total reward, which is also known as the Q-value (state-action value) as:

$$Q(s,a) = r(s,a) + \gamma * \max_a(Q(s',a)) \quad (3.4.1)$$

As shown above, r is the immediate reward for taking action a in state s, In addition the highest Q-value possible from next state s^0, when γ is discount factor, which determines the amount of rewards to be allocated in future. By this method, we can see that the q-values at each q-state depend on those of its next state, and since the equation is recursive, we can start by making arbitrary assumptions about the q-values at each q-state. (e.g. $Q(s,a) = 0, \forall s,a$).

As it gains experience, it will reach an optimal policy. Updates are implemented in practical situations as follows:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha * [R_{t+1} + \gamma * \max_a(Q(S_{t+1},a) - Q(S_t,A_t))] \quad (3.4.2)$$

Assuming that α is the learning rate/step-size, it determines how much of the newly gained information is overwritten by earlier acquired information.[8][12][11][5]

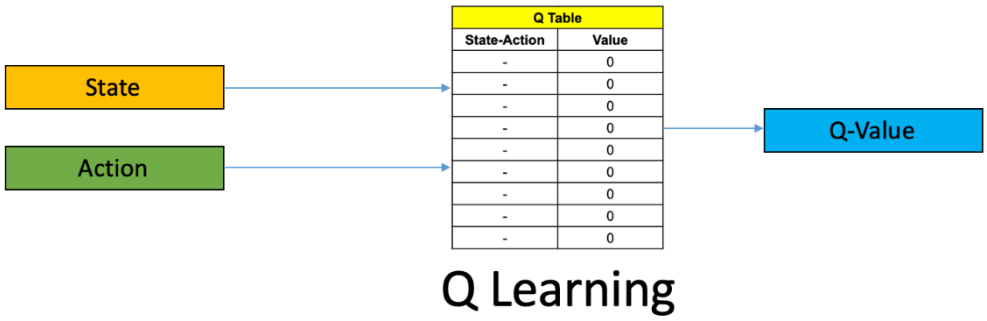


Figure 2.5: Q-Learning Model

2.6 Multi-Agent system

A multi-agent system is a group of autonomous, interacting entities sharing a common environment, which they perceive with sensors and upon which they act with actuators. Multi-agent systems are finding applications in a variety of domains including robotic teams, distributed control, resource management, collaborative decision support systems, data mining, etc. They may arise as the most natural way of looking at a system or may provide an alternative perspective on systems that are originally regarded as centralized.

Multi-agent drone systems are difficult to implement when it comes to synchronization, yet these systems have an advantage when reducing a problem's complexity. For instance, if the task is mapping a huge area with different objectives (Figure 3.5), performing it with just one drone could take a long time and require lots of resources. Using multiple drones, however, would reduce the performance time significantly and save more fixing resources, due to the relatively small area that each drone would map [20].

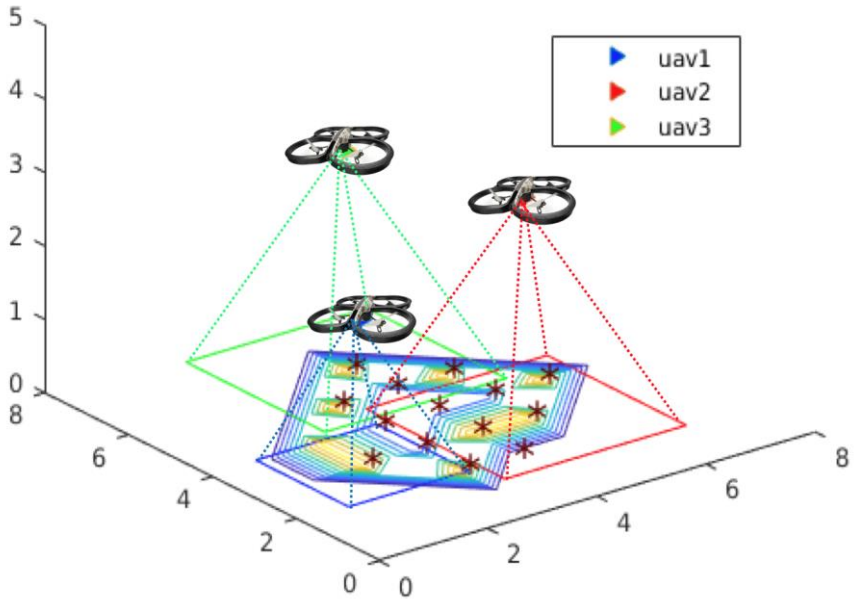


Figure 2.6: Multi Agent System

3| Methods

3.1 Overview

The aim is to develop an optimal model for planning of drone navigation paths. We will perform this task without the aid of a camera and only use a front distance sensor. Our method is comparing and evaluating different algorithms from the RL field in a simulation environment. Once the best algorithm has been discovered, it will be possible to implement the model on a physical drone and perform multi-agent tasks. Due to the limited hardware available, we decided not to test the model on the physical drone, but to evaluate it using different simulation environments.

3.2 Drones

Commercial off-the-shelf (COTS) quadrotors have already started to enter the nano-scale, featuring only few centimeters in diameter and a few tens of grams in weight. However, commercial nano-UAVs still lack the autonomy boasted by their larger counterparts since their computational capabilities, heavily constrained by their tiny power envelopes, have been considered so far to be totally inadequate for the execution of sophisticated AI workloads, as summarized in Table 4.1.[13]

Vehicle Class	⌀ : Weight [cm:kg]	Power [W]	Onboard Device
std-size [11]	$\sim 50 : \geq 1$	≥ 100	Desktop
micro-size [12]	$\sim 25 : \sim 0.5$	~ 50	Embedded
nano-size [13]	$\sim 10 : \sim 0.01$	~ 5	MCU
pico-size [14]	$\sim 2 : \leq 0.001$	~ 0.1	ULP

Table 3.2.1: Quad-copters classes

The system is based on ultra-low-power computing platform, and 27 grams commercial, open-source Bitcraze Crazyflie 2.1 nano-quadrotor. Due to the limited capabilities of the Crazyflie, the sensor payload is limited. The microcontroller does all the processing required in real-time without any external assistance with only 192 KB of RAM for the full flight stack and machine learning model. Table 4.2 shows a comparison between the Nvidia computer (Jetson TX2), which is designed to perform AI tasks and the micro-controller (STM32) used by the Crazyflie.[6]

Name	STM32F405	Jetson TX2
CPU	1-core 168MHz	6-cores 2GHz+
GPU	None	256-core 1300MHz
RAM	192 kB	8 GB
Storage	1 MB	32 GB
Power	0.14 W (max)	7.5 W

Table 3.2.2: Hardware Capabilities

3.3 Simulation

To train and evaluate the models, the "AirSim", a platform from Microsoft for training and simulation of autonomous vehicles including cars and quadcopters was used. It relies on Unreal Engine for the graphics, which provides many choices for building the environment. The team that created "AirSim" published an evaluation of a quadcopter model and found that the flight tracks simulated in the simulator closely match the behavior of real drones. [7][14]

In order to allow the drone to train in a relatively generic and simple environment, we implemented one of the environments from the "AirSim" library and made additional adjustments. It consists of a 20m x 20m room with five cylindrical obstacles at fixed areas (Figure 3.1.1). The drone starts its path at (0,0,0) and the target is randomly chosen, before each episode, by the uniform distribution. Targets are defined in the X- and Y-axes, and flight altitude remains constant during navigation. Once the drone collides with an obstacle or wall, the training session is reset and the drone returns to (0,0,0) and a new episode begins.

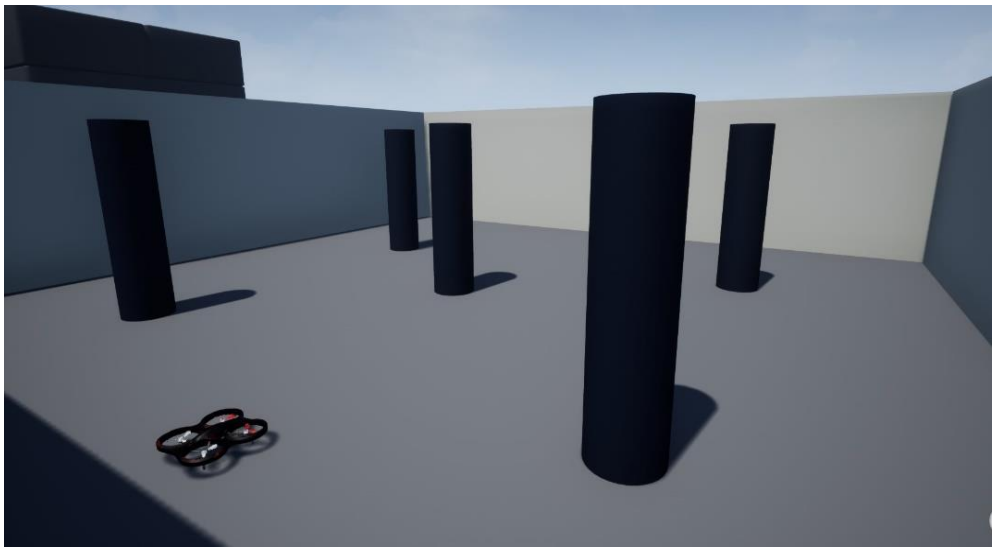


Figure 3.1.1: Training Environment

3.4 Algorithm Implementation

The drone's navigation algorithm is based on Q-learning. For this reason, two elements are to be considered during the application process: 1. "State" - the location of the drone. 2. "Action" - the action that the drone must perform in its current position. To do so, the Q-learning method is used to determine the "optimal action" to perform in each "state". The drone's "state" is determined by data collected by its distance sensor. A learning agent receives the start and destination locations, as well as obstacle locations, at the beginning of each learning phase. Using the distance sensor, the drone estimates its relative positions within the environment. Decisions about which "action" to perform in each location are made based on the future rewards of that action. The agent, which is designed to maximize the reward, will carry out "actions" that bring the drone as close as possible to the destination point, while avoiding obstacles along the way. This sequence of actions produces an optimal path to the destination location in each state the drone is located in the environment.

3.5 Multi-Agent Application

After selecting the algorithm with the best performance for the navigation task, we next developed an infrastructure for a multi-agent navigation task. Using this type of system, it is possible to perform navigation more quickly and efficiently, as several agents map the environment. By this point, we implemented two agents that navigate in parallel to a fixed and predetermined destination, starting from different locations. In fact, each drone implements an optimal navigation algorithm based on the single agent policy. Performance of this task is also improved as the agent located closer to the destination arrives at the destination quicker than the agent farther away. The next step is to set several destinations so each drone will navigate to its closest destination and implement this with more than two agents.

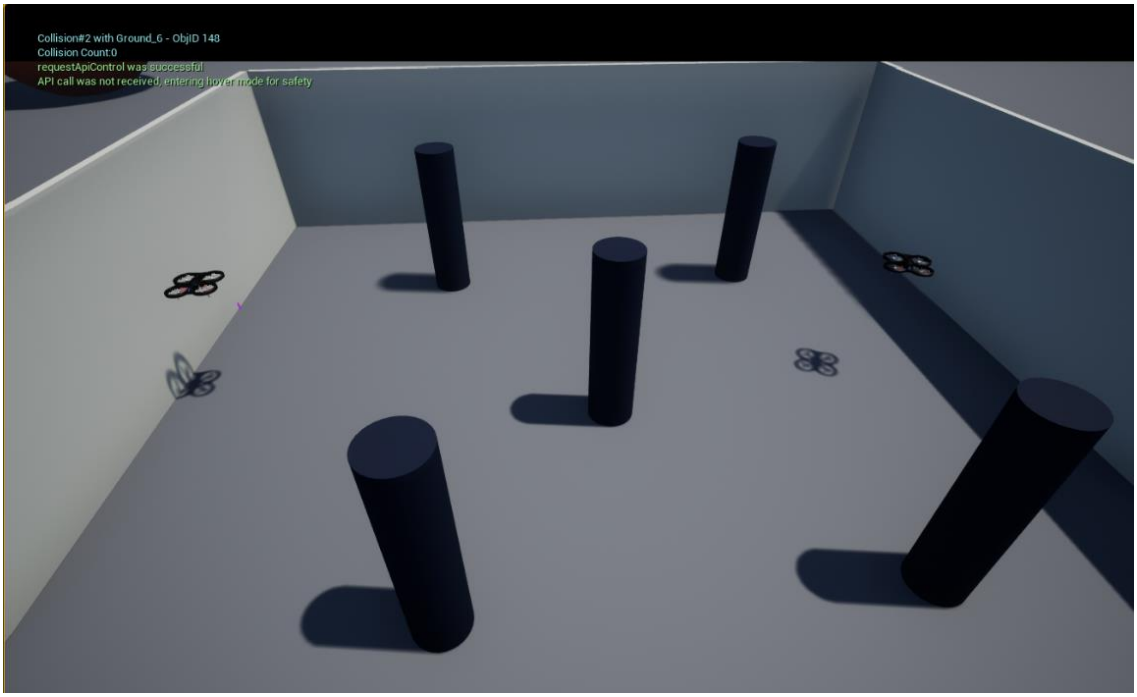


Figure 3.5: Multi-Agent Implementation

3.6 Experiments

3.6.1 Lab Experiments

As a first step to testing our algorithm on the drone (which requires a huge computational power), we had to test simple commands like "Fly straight/left/right" to ensure that the components are functioning properly. Based on these simple commands, we have determined that the drone is operating correctly.

For testing the model in a real environment, commands are transmitted over radio frequency by a computer to the drone. The computer receives a noisy sensory input from the drone, processes it, and sends the output (or the best action for its current state). Flight testing took place in a room that was roughly 4m x 3m, with a flight altitude of 0.6m to prevent severe crashes.

Due to the limited components of the “Bitcraze crazyflie” drone, we encountered a lot of issues implementing the algorithm, specifically crashes of the drone. As ordering new components during the COVID-19 period was rather challenging and replacing the drone was not possible at this stage of the project, we decided to focus on simulation experiments.

3.6.2 Simulation Experiments

The first experiments were performed on the training environment. To do this, we set fixed target locations for the drone when navigating to it. Intentionally, we chose locations that would challenge the drone's navigation operation, such as targets attached to obstacles or the confining walls of the room. We repeated the experiments several times and examined the decision-making processes of the agent.

In spite of the fact that our simulation environment relies on a physical engine (Unreal Engine) that includes natural forces such as gravitation, it still has some disadvantages. As an example, it is rather difficult to determine how well the drone would perform its tasks with real complex obstacles and different ground textures around it. To reduce the gap between the training simulation environment and the real world, and to create a more realistic environment in which to test the drone, we implemented a new “AirSim” environment (Figure 3.6.2).

To avoid overfitting to the training environment, the environment is different in terms of size, and has obstacles that differ from each other. Additionally, the environment looks less generic than the training environment, and may well be comparable to a real-world environment. In this case, the environment can be considered as a kind of operational environment when unique obstacles are involved. As with the training environment, we selected relatively challenging targets and conducted the experiment over a fairly large number of repetitions. As the drone continues to learn more environments of this kind, we expect its performance to improve when it comes to training phases in a physical environment.

Figure 3.6.2: Testing Environment

3.7 Performance Measure & Analysis

The training set is defined using 3000 'Games' where in each game the agent gets the starting point, data from the distance sensor, data on obstacle locations, randomly selected variable target, action and reward for performing this action, and whether the drone was able to reach the target. The testing set was similarly defined for 100 'Games' in each environment, except that the target location is now predefined rather than randomly assigned.

In order to evaluate the performance of the various models we implemented and compared three main indicators during the evaluation phase. Two of them were measured during **training**:

1. **Success rate**: The percentage of times the drone has successfully reached its destination during training.

2. **Score**: The cumulative amount of rewards that the agent has earned during its training

During the **testing** stage, a total of 100 experiments were conducted for each model, and the following indices were measured:

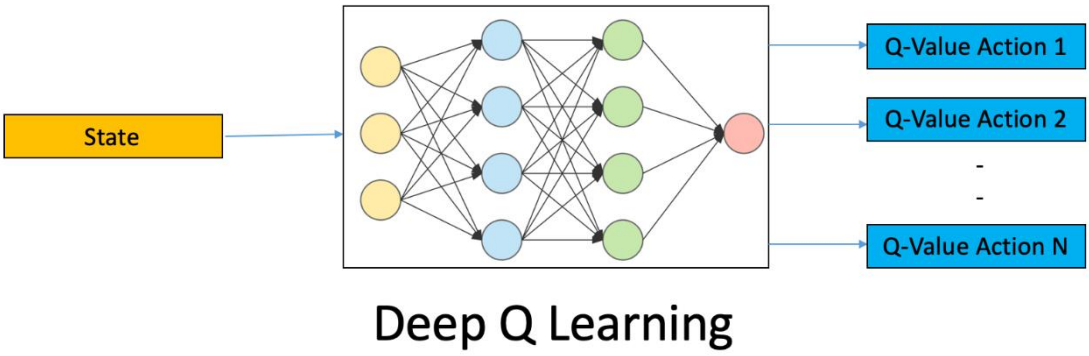
3. ‘Done’ per 100 Games: The number of times the drone successfully reached the target from the number of experiments performed. This was determined by using 100 binary variables called 'Dones', which count each time the agent reaches its destination.

In the results stage, these metrics will be described using suitable plots.

4| Algorithms

4.1 DQN

The Deep Q Network (DQN) algorithm is capable of human level performance on many Atari video games using raw pixels as input. The action-value function was then estimated using deep neural network approximators. Although DQN can handle high-dimensional observation spaces, it is limited to low-dimensional and discrete action spaces. There are several interesting tasks, such as physical control, which involve continuous (real value) and high dimensional action spaces. Since DQN tries to find the action that maximizes the action-value function, it cannot be directly applied to



continuous domains since every step in the optimization process has to take an iterative shape. The most obvious way to apply deep reinforcement learning to continuous domains such as DQN is to discretize the action space. Nevertheless, this approach is bound to have limitations, most notably the curse of dimensionality: the number of actions rapidly increases as the degrees of freedom increase. Quadcopters, for example, have six degrees of freedom with coarse discretization $a_i \in \{0,1\}$ The action space for each motor has a dimension of: $2^6 = 64$. It is even worse for activities requiring fine control of actions since they require a correspondingly fine-grained discretization, leading to a massive increase in the number of discrete actions. In the context of large action spaces, it is likely intractable to train DQN-like networks. Furthermore, naive discretization of action spaces throws away information about the structure of the action domain, which may be crucial for solving many problems. In this study, we describe a model-free, off-Policy actor-critical algorithm that learns policies in high-dimensional, continuous action spaces using deep function approximations.[10][8]

Figure 4.1: DQN Model

4.2 Reward Function

Basically, the quality of each action is determined by the reward that the agent gets by doing that action. The reward function is presented in Equation 4.2. At each step the drone receives data from the environment and distance sensor regarding its location relative to the target location and whether it has collided with one of the obstacles in the environment. Accordingly, it sets up the relevant values in the reward function. The reward from performing this "action" is calculated and stored in a dedicated data structure in memory, so when the drone returns to the same location the next time, it can predict the reward from performing each of the possible actions, and then choose the option which yields the greatest reward.

Whenever a goal has been reached, α is set to 1. In the event that the agent collides with an obstacle or wall, then β is set to 1. It also sets to 1 if the agent stays at its place for 8 steps or more, or if the agent takes 300 steps in one episode. The Δd is the difference between the agent's distance from the goal in time step t and time step t-1, to promote actions that get the agent closer to the goal and penalize actions that bring it further away. The −1 exists in order to penalize delays [10].

$$R = 1000 \cdot \alpha - 100 \cdot \beta - 20 \cdot \Delta d - 1$$

$$\alpha = \begin{cases} 1 & if\ reached\ target \\ 0 & else \end{cases}$$

$$\beta = \begin{cases} 1 & if\ collided\ or\ exceeds\ step\ limits \\ 0 & else \end{cases}$$

$$\Delta d = distance\ now\ - distance\ before$$

Equation 4.2

4.3 DQN- model improvements

A supervised learning procedure involves independent and identical distributions of inputs - samples are randomized, thus each batch has the same distribution, and samples within a batch are independent. Otherwise, the model might be over-fitted, meaning the solution cannot be generalized.

According to equation 3.4.2, the target values for Q depend on Q itself, for this reason we chase a non-stationary target [18]. To address this issue, we have upgraded our model by implementing two main features:

1. Replay Memory- Transitions are inserted into a buffer, and a mini-batch of samples are then used to train the deep network.

2. Target Q Network- Two deep networks are created with the same architecture ϑ and ϑ^{-}. The first one is used to retrieve updates in the training and the second one is used to retrieve Q values from st+1.

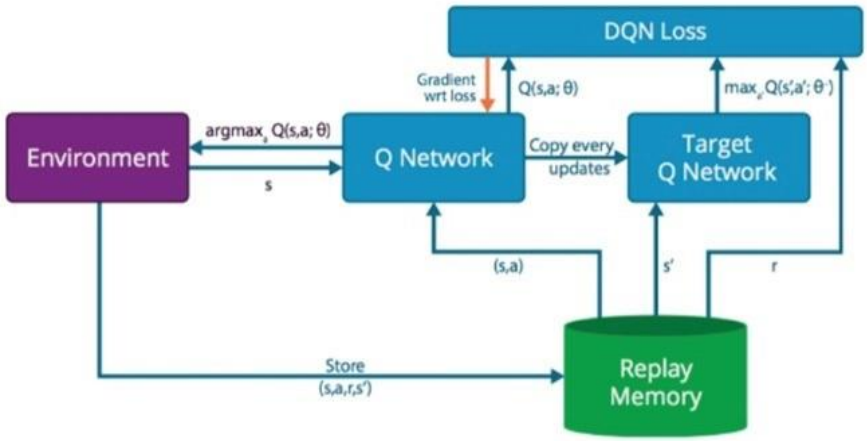


Figure 4.3.1: DQN Model Architecture

In order to update the target network, there are two possible methods:

1. Hard update- Every 10 episodes we synchronize ϑ^{-} with ϑ
2. Soft update- - Every time step the target network is updated according to the Equation 4.1.2:
   $\vartheta^{-} = \tau\vartheta + (1 - \tau)\vartheta^{-}, \qquad \tau \in [0,1]$

τ- Initialized to a number that tends to zero (e.g. 0.01)

It is intended to fix the Q-value target so that we do not chase it. The target network and experience replay provide more stable input and output, so the network behaves more like supervised learning.

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-)$$

Figure 4.3.2: DQN Target Value

4.4 DQN- The training method

In RL, one of the biggest difficulties is balancing exploration with exploitation [14]. There is the need for the agent to explore its environment, that is to try different actions in a given state, so that it won't get stuck in a local minimum, however, letting the agent choose randomly each time will cause the model to fail to converge. Therefore, we let it explore at first, and over time it will explore much less and begin to exploit the Q-values. This is in order to satisfy the GLIE properties [14]:

1. Each state-action pair is explored infinitely many times:

$$\lim N_k(s,a) = \infty \; k \to \infty$$

2. Policy converges on a greedy policy

Equation 4.4.1 presents the GLIE function:

$$\epsilon = \begin{cases} \epsilon \cdot \epsilon_{decay}, & if \quad \epsilon > \epsilon_{min} \\ \epsilon_{min}, & else \end{cases} \tag{4.4.1}$$

Hyperparameters	Value	Notes
episodes	3000	
epsilon start	1.1	
epsilon end	0.01	
epsilon decay	0.98	
batch size	64	
replay memory	100,000	
discount factor γ	0.99	
learning rate α	0.0025	Adam optimizer
target update period	10	relevant for hard update

Table 4.4: Hyperparameters for the training

In order to prevent the agent from staying at a local minimum at every time step, the agent performs an action according to the network (Figure 4.4.1) or chooses a random action with a certain probability. After the agent has performed the action, the environment outputs: 1. the reward it received, 2. the new state it has reached, 3. whether the agent has completed the current episode or not (reach the goal or collide with an obstacle). At every step performed by the agent, this information is stored and forms a database for the neural network. The network should converge after a large enough number of steps, so the algorithm knows - for each state of the agent - which actions will bring it the most value and ultimately lead it to the target.

Similarly, the Testing environment uses the neural network in the same way, and in fact the network receives the agent's state from it and outputs an action that should be taken.

Observations are modified in a way that will facilitate neural network learning. Observations are defined as follows: The position of the X axis, the position of the Y axis, the Euclidean distance to the goal, the angle to the goal, and the distance to the goal as read by the front distance sensor (IR sensor). Finally, the network decides which action to take straight, right or left.

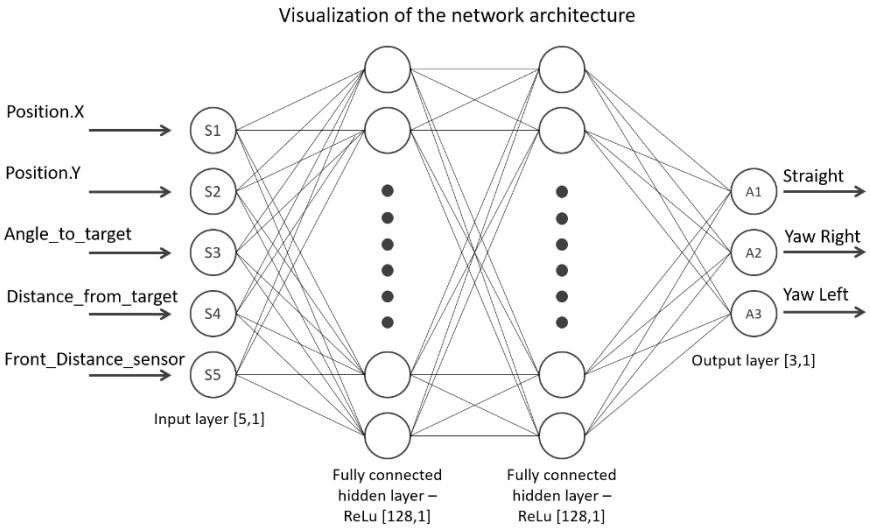


Figure 4.4: Network Architecture

The 'ReLu' activation function is used. Activation function transforms the summed weighted input from the node into the activation of the node or output for that input. A rectified linear activation function is a piecewise linear function that outputs the input directly if it is positive and zero otherwise. Because neural networks with this activation function are easier to train and generally perform better, it has become the default activation function for many types of neural networks [22].

4.5 DDQN (Double DQN)

As part of improving algorithm performance, we explore some additional approaches for using DQN models. This is how we discovered the Double DQN (DDQN) method.

The idea of Double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation. Although not fully decoupled, the target network in the DQN architecture provides a natural candidate for the second value function, without having to introduce additional networks. We therefore propose to evaluate the greedy policy according to the online network but using the target network to estimate its value. In reference to both Double Q-learning and DQN, we refer to the resulting algorithm as Double DQN. Its update is the same as for DQN, but replacing the target Q(s,a) (Figure 3.4.1) with:

$$Q(s,a) = r(s,a) + \gamma\, Q(s', \operatorname{argmax}_a Q(s',a))$$

Figure 4.5.1: Double DQN Target Value

The concept behind this solution is by decoupling the selection of the actions from the generation of the target Q value, we compute the Q target by using two networks [23]:

- Choose the best action for the next state using our DQN network (the action with the highest Q value).

- Calculate the target Q value for taking that action at the next state using our Target Q Network.

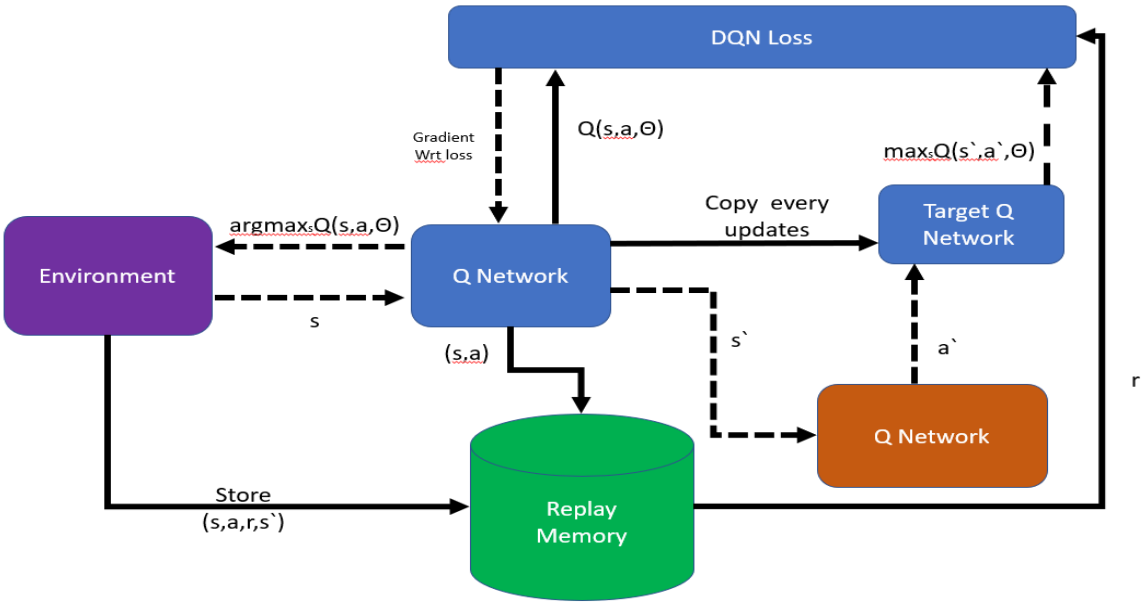


Figure 4.5.2: Double DQN Architecture

5| Results

5.1 Training Environment

For the purpose of evaluating the models in the training environment, two types of results were produced. One is during the training phase, and the other is during the testing phase. Each chart describes a different metric (Section 3.7) for each of the models:

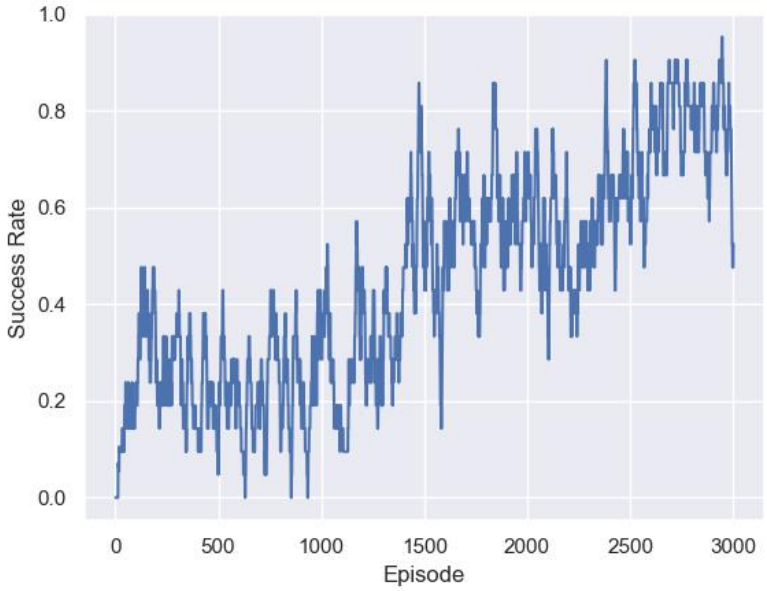


Figure 5.1.1: DQN Success rate

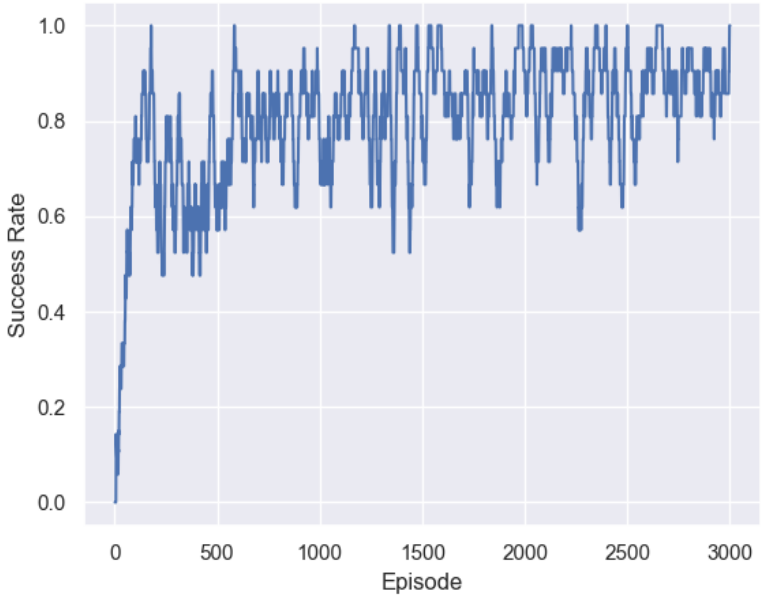


Figure 5.1.2: Double DQN Success rate

These data were collected during training and describe how fast each model reaches high success rates. According to these graphs, the DDQN model converges to a higher success rate faster than the DQN model, which means it learns the environment faster.

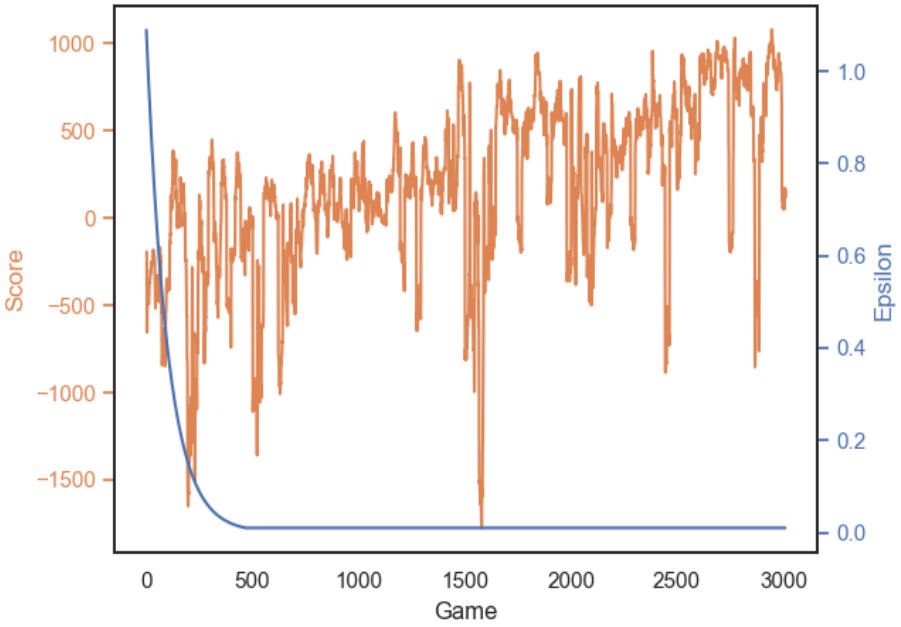


Figure 5.1.3: DQN Score

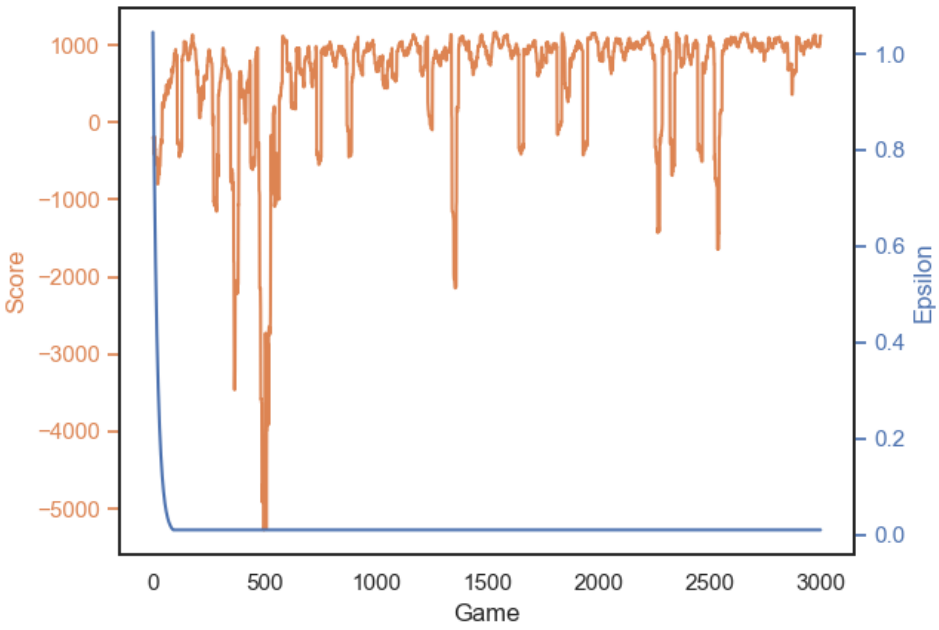


Figure 5.1.4: Double DQN Score

On the basis of the score metric, it is apparent that in the DQN model, the trend has been upward. The DDQN model exhibits a higher success rate excluding exceptions. Another interesting point is that the DQN ranges are almost exactly the same as the DDQN ranges in spite of the upward trend. Based on the graphs, the DDQN model has indeed improved the performance.

We will now present the results of the testing phase

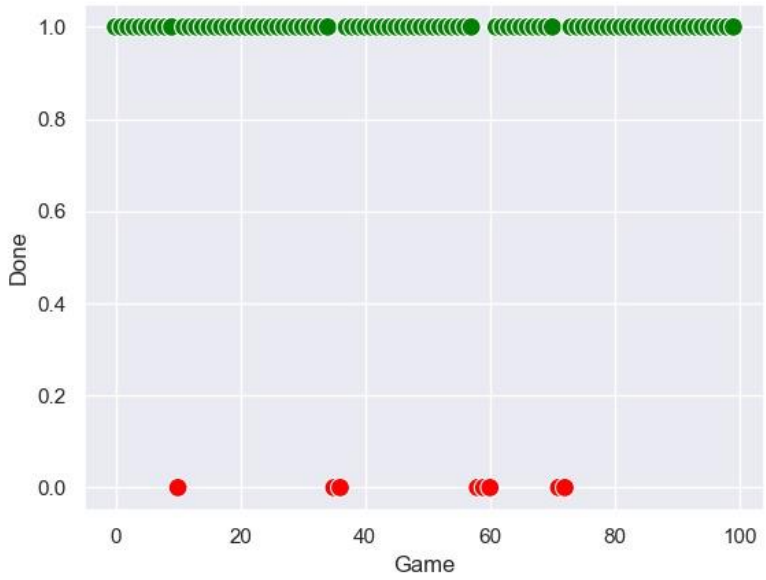


Figure 5.1.5: Training: DQN Done of 100 Games

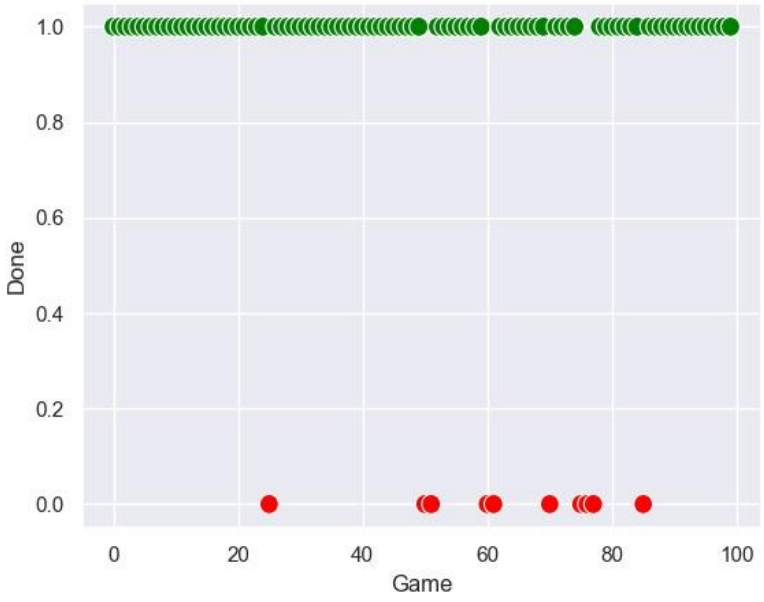


Figure 5.1.6: Training: DDQN Done of 100 Games

Based on the two graphs that show the number of times the drone has reached the target within 100 experiments, it is clear that both models have high success rates.

DQN reaches its target 92 times out of 100 experiments, meaning 92 % of games are successful. The DDQN performed 90% of the games successfully

5.2 Testing Environment

We will now examine the results for each of the models in the testing environment, which is an environment where the agent (drone) is navigating for the first time.

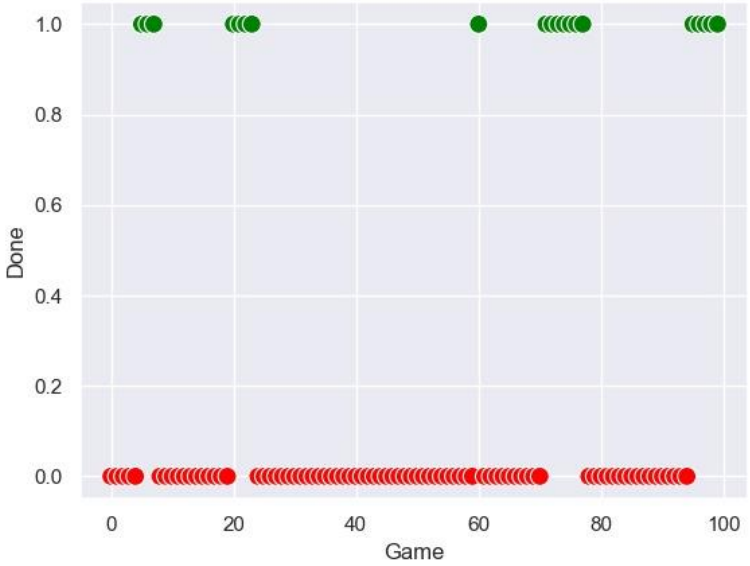


Figure 5.2.1: Testing: DQN Done of 100 Games

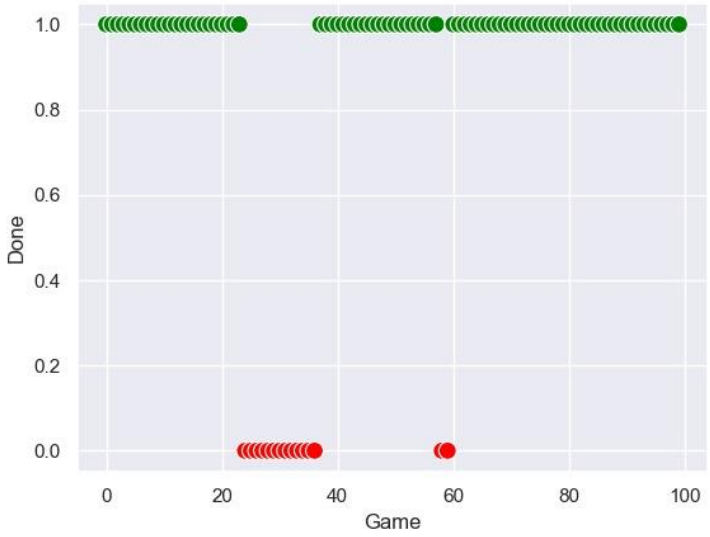


Figure 5.2.2: Testing: 2.1DDQN Done of 100 Games

According to the DQN graph, the percentage of reaching the target position in the test environment is significantly lower (40%) than the success rates achieved in the training environment. It can be concluded that this model is over-fitted to the training environment. Conversely, the DDQN model maintained a relatively high success rate (85%). The over-fitting problem can be addressed by training the models in as many different environments as possible. The DDQN model, based on the results we obtained in this chapter, should learn faster and perform better than the DQN model.

6| Conclusion & Future Work

This project presents an autonomous path planning strategy utilizing a deep reinforcement learning policy that is applied to simulations. The simulation platform we used, 'AirSim', has only a front distance sensor, while CrazyFlie, the physical drone has a five-direction distance sensor. This gap led us to use the grid in a different way than we initially intended - reading front, rear, left, right distance sensors, and the distance to the target and the angle to the target. However, the 'AirSim' platform simulates the environment in a very realistic way, making the simulation gap a very small one, so it wasn't necessary to fine-tune the model for the real drone.

Due to the limited hardware capabilities of the drones, it wasn't possible to fully apply the neural network model to the drone, as it required large computational power. Therefore, we focused on the simulation environment. In order to evaluate the model developed in an environment that better simulates real-world situations. For this environment, we examined the performance of several algorithms in the field of DRL and concluded that the best model for performing the task is the DDQN (Double DQN) model that generated the highest success rate in the exam. Finally, we have developed a preliminary model for performing a multi-agent task, which may serve as the foundation for future work in this area.

This project integrated a Python API, a simulation engine, and theoretical knowledge and comprehension - https://github.com/Ofirnag/RL_Project

As for future work, our project products can be used to develop additional tasks. First of all, we understand the importance of training the model in real world conditions, so that research can be done for additional drones that can apply the models we have developed in real world settings. Additionally, our multi-agent system infrastructure can be used for more complex tasks that require the simultaneous work of multiple drones.

7| References

[1] L. Argentim, W. Rezende, P. Santos, and R. Aguiar. Pid, lqr and lqr-pid on a quadcopter platform. In *2013 International Conference on Informatics, Electronics and Vision (ICIEV)*, pages 1– 6. IEEE, 2013.

[2] K. Åström and T. Hägglund. *PID controllers: theory, design, and tuning*, volume 2. Instrument society of America Research Triangle Park, NC, 1995.

[3] A. Choudhary. A hands-on introduction to deep q-learning using openai gym in python. *Retrived from https://www. analyticsvidhya. com/blog/2019/04/introduction-deep-q-learningpython*, 2019.

[4] B. Duisterhof, S. Krishnan, J. Cruz, C. Banbury, W. Fu, A. Faust, G. de Croon, and V. Reddi. Learning to seek: Deep reinforcement learning for phototaxis of a nano drone in an obstacle field. *arXiv*, pages arXiv–1909, 2019.

[5] Epic Games. Unreal engine. https://www.unrealengine.com.

[6] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, Tom Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with Ben-Gurion University of the Negev Industrial Engineering & Management

[7] T. Luukkonen. Modelling and control of quadcopter. *Independent research project in applied mathematics, Espoo*, 22, 2011.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[9] G. Muñoz, C. Barrado, E. Çetin, and E. Salami. Deep reinforcement learning for drone delivery. *Drones*, 3(3):72, 2019.

[10] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, and L. Benini. A 64-mw dnn-based visual navigation engine for autonomous nano-drones. *IEEE Internet of Things Journal*, 6(5):8357–8371, 2019.

[11] D. Palossi, J. Singh, M. Magno, and L. Benini. Target following on nano-scale unmanned aerial vehicles. In *2017 7th IEEE international workshop on advances in sensors and interfaces (IWASI)*, pages 170–175. IEEE, 2017.

[12] S. Shah, D. Dey, C.Lovett, and A. Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer, 2018.

[13] S.Y. Shin, Y. W. Kang, and Y.G. Kim. Obstacle avoidance drone by deep reinforcement learning and its racing with human pilot. *Applied Sciences*, 9(24):5571, 2019.

[14] S.Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine learning*, 38(3):287–308, 2000.

[15] S.L. Sun and Z.L. Deng. Multi-sensor optimal information fusion kalman filter. *Automatica*, 40(6):1017–1023, 2004.

[16] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[17] S.L. Sun and Z.L. Deng. Multi-sensor optimal information fusion kalman filter. *Automatica*, 40(6):1017–1023, 2004.

[18] R.S Sutton and A. G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[19] D. Warren. Trajectory Generation and Control for Quadrotors, Scholarly Commons, 2012

[20] H. X. Pham, H. M. La, D. Feil-Seifer, and A. Nefian, *Cooperative and Distributed Reinforcement Learning of Drones for Field Coverage,* 2018

[21] J. Brownlee. A gentle introduction to the rectified linear unit (relu). *Machine Learning Mastery. https://machinelearningmastery. com/rectifiedlinear-activation-function-fordeep-learning-neural-networks*, 2019.

[22] H. van Hasselt and A. Guez and D. Silver. Deep Reinforcement Learning with Double Q-learning, arxiv.org ,2015