Technical University of Crete
School of Electrical & Computer Engineering

Diploma Thesis

Autonomous Drone Navigation for Landmark Position Estimation using Reinforcement Learning

Michalis Galanis

Thesis Committee

Associate Professor Michail G. Lagoudakis (School of ECE)
Professor Michalis Zervakis (School of ECE)
Associate Professor Panagiotis Partsinevelos (School of MRE)

Chania, September 2021

Πολυτεχνείο Κρήτης
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Διπλωματική Εργασία

Αυτόνομη Πλοήγηση Drone για Εκτίμηση Θέσης Διακριτικών με χρήση Ενισχυτικής Μάθησης

Μιχάλης Γαλάνης

Εξεταστική Επιτροπή

Αναπληρωτής Καθηγητής Μιχαήλ Γ. Λαγουδάκης (Σχολή ΗΜΜΥ)
Καθηγητής Μιχάλης Ζερβάκης (Σχολή ΗΜΜΥ)
Αναπληρωτής Καθηγητής Παναγιώτης Παρτσινέβελος (Σχολή ΜΗΧΟΠ)

Χανιά, Σεπτέμβριος 2021

Acknowledgements

An endless amount of hours and effort was devoted to this thesis. Learning about machine, deep and reinforcement learning was very informative, experimenting with ROS and Gazebo was really interesting and mastering Linux was painful but worth it. Therefore, i would like to express my acknowledgements to my professors, close friends and family.

Personal

First of all, my greatest appreciation and gratitude to my supervisors in my thesis committee and especially to associate professors Panagiotis Partsinevelos and Michail G. Lagoudakis who guided me throughout the course of this project.

I would also like to thank my colleagues Dimitris and Angelos, for providing me a significant head start on my thesis, as well as Antheas for his hyperparameter suggestions.

I cannot appreciate enough the shared memories with my friends. They were always by my side and without them i would not have made it.

Last, but not least, my honest and special thanks to Ifigenia and my parents Despina and Nikos for the enormous support and unlimited love they have shown me for the past five years of my studies.

Institutional

I feel very fortunate for being part of *SenseLab*, Panagiotis Partsinevelos's multi-award-winning and world-leading research team operating in several fields, including Geographic Information Systems, Remote Sensing, Unmanned Aerial Vehicles and Artificial Intelligence. Their incredibly high team expertise, keeps pushing the boundaries of UAV applications by designing, constructing and developing high-performance systems that outperforms worldwide related efforts.

Last but not least, I am very grateful for accomplishing my studies at the Technical University of Crete, one of the best technical universities in Greece. The ECE School provides in-depth education and high-level training for engineers in cutting-edge technologies and is deeply involved in basic and applied research activities. Every professor, associate professor, assistant professor and lecturer in this school is incredibly experienced and committed with outstanding careers.

Abstract

Unmanned aerial vehicles (UAVs) have been increasingly used for critical and challenging applications, which often require a substantial level of autonomy. Several approaches have been investigated to create autonomous navigation systems such as *Simultaneous Localization and Mapping* (SLAM) using real-time mapping and position estimation. *Reinforcement leaning* (RL) is a promising alternative that focuses on learning to perform a task through a trial-and-error procedure, in which an agent interacts with its environment and receives continuous feedback based on the actions taken, with no access to any information about the environment itself. Eventually, the agent's objective is to find the best possible sequence of actions that lead to the maximum total reward in the long term. This thesis explores a mapless approach to UAV autonomous navigation in completely unknown 3D environments using *deep reinforcement learning* (DRL), a reinforcement learning approach that incorporates deep learning techniques (deep neural networks) to overcome dimensionality limitations. The goal of the agent is to safely navigate through this unknown environment, so as to detect and approach a predefined set of ArUco markers (landmarks) placed within the environment. The unknown environments are dynamically created and contain a number of procedurally generated obstacles. We evaluate our agent in five different environment profiles with increasing difficulty level and observe how environment complexity affects training performance. Results show that deep reinforcement learning can be effective and can be successfully used for autonomous navigation missions. The entire project was implemented using the *Robot Operating System* (ROS) platform within the *Gazebo* robot simulator environment.

Περίληψη

Τα μη επανδρωμένα αεροσκάφη (Unmanned Aerial Vehicles, UAVs) χρησιμοποιούνται ολοένα και περισσότερο για κρίσιμες και απαιτητικές εφαρμογές, οι οποίες συχνά απαιτούν ένα σημαντικό επίπεδο αυτονομίας. Έχουν διερευνηθεί διάφορες προσεγγίσεις για τη δημιουργία συστημάτων αυτόνομης πλοήγησης, όπως ο *ταυτόχρονος εντοπισμός και χαρτογράφηση* (SLAM) που υλοποιεί σε πραγματικό χρόνο χαρτογράφηση και εκτίμηση θέσης. Η *Ενισχυτική Μάθηση* (Reinforcement Learning, RL) θεωρείται μια πολλά υποσχόμενη εναλλακτική λύση που επικεντρώνεται στη μάθηση κάποιου έργου μέσω μιας διαδικασίας δοκιμής και σφάλματος, στην οποία ένας πράκτορας αλληλεπιδρά με το περιβάλλον του και λαμβάνει συνεχή αξιολόγηση εξαρτώμενη από τις ενέργειες που επιλέγει, χωρίς ωστόσο να έχει πρόσβαση σε πληροφορίες για το ίδιο το περιβάλλον. Εν τέλει, ο στόχος του πράκτορα είναι να βρει την καλύτερη δυνατή ακολουθία ενεργειών που θα εξασφαλίσουν τη μέγιστη συνολική ανταμοιβή μακροπρόθεσμα. Η παρούσα διπλωματική εργασία διερευνά μια προσέγγιση αυτόνομης πλοήγησης αεροσκαφών (χωρίς χάρτη) σε εντελώς άγνωστα τρισδιάστατα περιβάλλοντα χρησιμοποιώντας *βαθιά ενισχυτική μάθηση* (Deep Reinforcement Learning, DRL), μια προσέγγιση ενισχυτικής μάθησης που ενσωματώνει τεχνικές βαθιάς μάθησης (βαθιά νευρωνικά δίκτυα) για να αντιμετωπιστούν οι περιορισμοί διαστατικότητας. Ο στόχος του πράκτορα είναι να περιηγηθεί με ασφάλεια στο άγνωστο περιβάλλον, ώστε να εντοπίσει και να προσεγγίσει έναν προκαθορισμένο αριθμό διακριτικών δεικτών ArUco που είναι τοποθετημένοι μέσα στο περιβάλλον. Τα άγνωστα περιβάλλοντα δημιουργούνται δυναμικά και συμπεριλαμβάνουν έναν πλήθος από εμπόδια παραγόμενα με αυτοματοποιημένο τρόπο. Αξιολογούμε τον πράκτορας μας σε πέντε διαφορετικά προφίλ περιβαλλόντων με αυξανόμενο επίπεδο δυσκολίας και παρατηρούμε πως η πολυπλοκότητα του περιβάλλοντος επηρεάζει την απόδοση της μάθησης. Τα αποτελέσματα δείχνουν ότι η βαθιά ενισχυτική μάθηση μπορεί να είναι αποτελεσματική και μπορεί να χρησιμοποιηθεί επιτυχώς σε αποστολές αυτόνομης πλοήγησης. Η εργασία στο σύνολό της έχει υλοποιηθεί μέσω της πλατφόρμας *Robot Operating System (ROS)* στο περιβάλλον ρομποτικής προσομοίωσης *Gazebo*.

Contents

List of Figures

Acronyms

UAV . . . . . Unmanned Aerial Vehicle (Drone)
SAR . . . . . Search and Rescue
AI . . . . . Artificial Intelligence
ML . . . . . Machine Learning
DL . . . . . Deep Learning
RL . . . . . Reinforcement Learning
IMU . . . . . Inertial Measuring Unit
DRL . . . . . Deep Reinforcement Learning
ANN . . . . . Artificial Neural Network
ROS . . . . . Robot Operating System
SL . . . . . Supervised Learning
UL . . . . . Unsupervised Learning
MDP . . . . . Markov Decision Process
TD . . . . . Temporal Difference
NN . . . . . Neural Network
DQN . . . . . Deep Q-Network
DDQN . . . . Double Deep Q-Network
ROS . . . . . Robot Operating System
SLAM . . . . . Simultaneous Localization And Mapping
DOF . . . . . Degree Of Freedom
JSON . . . . . JavaScript Object Notation

1 Introduction

Contents

1.1 Introduction

Over the past decade, the use of unmanned aerial vehicles (UAVs) has been increasingly popular. They have shown promise in a variety of practical applications including construction sites, agricultural remote sensing, landmark estimation, search and rescue (SAR), surveillance and offensive military operations. Drones offer incredibly high flexibility with minimal operational costs, which renders them ideal for exploration in unknown, normally inaccessible and potentially hazardous-for-humans environments. In most scenarios, the UAV must safely navigate among various locations to perform specific tasks. Therefore, an effective and efficient navigation system is considered necessary in order to successfully accomplish these missions.

When these devices were initially developed, navigation was performed manually and remotely controlled by humans. In the following years, semi-autonomous safety

features were developed to actively assist human pilots. Lately, huge advancements in information technology and artificial intelligence (AI) lead to the development of flight mechanisms which can completely take over and perform autonomous flights in an environment. These mechanisms are able to collect important information about the environment from the attached sensors and therefore enable the drone to successfully avoid collisions and locate and track targets by processing this information and creating observational patterns.

By observing the natural learning process of the animals (since nature is a continuously proven source for human inventions), researchers developed one of the most emerging subfields of artificial intelligence, designed for solving control-related challenges: Reinforcement learning (RL), a technique that focuses on training an algorithm by attempting actions in an environment and carefully observing and evaluating the feedback it receives after each step. Similar to every animal's childhood, positive feedback is a reward, while negative feedback is a punishment for making a mistake. This continuous trial-and-error interaction ultimately allows the agent to successfully learn from its past experiences.

1.2 Thesis Contribution

This thesis proposes a deep reinforcement learning approach to autonomous navigation. According to this approach, an agent (UAV) randomly spawns inside a three-dimensional (3D) environment with obstacles and targets and its objective is to proceed towards every target in the environment. Specifically, the DQN algorithm is implemented, which aims to discover the best sequence of actions in order to receive the most overall positive feedback from the environment. DQN follows the traditional pipeline of a reinforcement learning problem, but is influenced by recent advancements of the deep learning subfield, particularly through the use of deep neural networks. DQN is extensively analyzed in Section (2.1.8.2).

The entire project is implemented using Robot Operating System (ROS), a platform that allows the design of functional and realistic robotic models with sensors and motors. To visualize our UAV, the Gazebo robot simulator was used. Regarding the

environment, configurable, lightweight and dynamic 3D worlds were constructed with ranging difficulty to serve as a testbench for the reinforcement learning algorithm. This thesis uniquely combines autonomous navigation control using reinforcement learning on a ROS platform. The results show that deep reinforcement learning techniques can be very effective. Since this is a ROS project and the model's elements are identical copies of already manufactured parts, these results can be easily reproduced in the real world, with almost no coding modifications.

1.3 Thesis Structure

Moving forward, this thesis is categorized into five distinct chapters.

- Chapter 2 presents all the background information required, spanning from the foundations of reinforcement learning to the tools and frameworks that were utilized throughout this project.
- Chapter 3 explicitly states the problem that this thesis attempts to resolve and compares our approach to other investigated solutions.
- Chapter 4 extensively analyzes the executed approach, including the environment creation, UAV model design and algorithm implementation.
- Chapter 5 sets up experiment configurations and benchmarks our approach with multiple difficulty levels to evaluate its performance.
- Chapter 6 underlines some limitations of our approach and indicates possible extensions that could produce improved results.

2 Background

Contents

This chapter introduces all the required background for this thesis. Initially, an overview is presented about the traditional, mathematically formulated approach of reinforcement learning. We explain the goal of any reinforcement learning problem and analyze the equations of achieving this goal. Later, a brief

discussion will be made about the categorization of reinforcement learning algorithms with a particular focus on Q-Learning and DQN. Additionally, an outline will be provided about the Robot Operating System platform, the Gazebo simulation software and several frameworks used for this project.

2.1 Reinforcement Learning

2.1.1 Overview

Machine learning is a branch of artificial intelligence which studies computer algorithms that improve over time through experience using large amounts of data. It is generally broken down into three main categories: supervised learning (SL), unsupervised Learning (UL) and reinforcement learning (RL).

In supervised learning, the learning algorithm attempts to learn the dependencies between data points. The learning process is "supervised" by matching the calculated results with the original output (ground truth) which is provided in advance. Supervised algorithms often solve problem categories, such as regression and classification.

In unsupervised learning, ground truth labels are not provided with the data, therefore the learning algorithm attempts to determine patterns dynamically with unknown initial relationships between data. This approach needs larger amounts of data in order to successfully create an accurate model. Clustering and dimensionality reduction are the most typically used unsupervised learning algorithms.

In reinforcement learning, an agent tries to learn to behave in an unknown environment through trial-and-error. Unlike supervised learning, it does not depend on a supervisor (labeled data), instead, the agent learns from its own experience created during the interaction with the environment. The objective of the agent is to find an optimal sequence of actions that would lead to the maximum cumulative reward in search for a goal.

The three main categories of machine learning are shown in Figure (2.1).

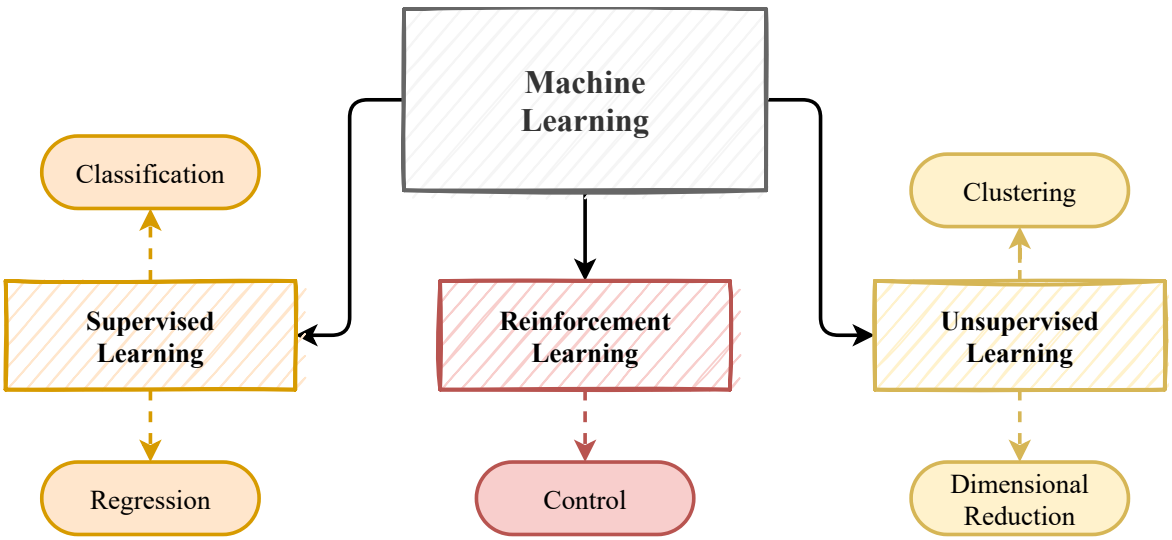


Figure 2.1: Machine learning and its three main subfields.

2.1.2 General Terms

This subsection focuses on some basic definitions of RL.

- **Step (time-step)** - A time-step is the smallest possible unit of time in a reinforcement learning problem. Every state corresponds to a specific time-step.
- **Episode** - An episode is a collection of time-steps starting from an initial state all the way to a terminal state. After reaching a terminal state, a new episode emerges.
- **Agent** - The agent is the algorithm itself, the learning part of the reinforcement learning system. It has the power to act on an environment and receive a reward from it.
- **Environment** - The environment is the world through which the agent acts. It takes the current state of the agent as input and returns the reward and next state to it. Environments follow a set of predefined laws. They may change over time thus can be described as either static or dynamic.
- **State** (s) - A state is a collection of variables and conditions that describe a situation of the environment at a specific point in time. Every action ever taken

by the agent at each time step can be represented by a specific state. Terminal states are states in which the agent is in a fatal condition and therefore the environment needs to be reset.

- **State Space (S)** - A state space is the set of every possible state of the environment such that $s \in S$. Depending on the application, it can be either discrete (e.g. grid world) or continuous (e.g. robotic arm).

- **Action (a)** - An action is every possible move the agent is allowed to make. The action can be represented as an output of the agent.

- **Action Space (A)** - An action space is the set of every possible action the agent can make such that $a \in A$. Similarly to the state space, the action space can be either discrete (e.g. left/right in a grid world) or continuous (e.g. 75% throttle in a driving situation). In continuous action spaces, the agent must output some real-valued number, possibly in multiple dimensions. Continuous action spaces are used in rarer cases due to their increased complexity and poorer support by reinforcement learning algorithms. In those scenarios, promising results can be achieved by discretizing the action space into fewer, discrete actions.

- **Reward (R)** - The reward is a feedback returned to the agent by the environment, functioning as an evaluation that indicates the success or failure of its previous action. Rewards can be tricky to design, because they are tied up to a specific environment.

- **Policy (π)** - In mathematical terms, the policy $\pi(s)$ is a probability distribution over actions given states.

$$\pi(s) \rightarrow a$$

  In practice, it's the strategy that the agent obeys and improves over time to determine the next action based on the current state. RL algorithms are tasked to learn an optimal policy that achieves a specific goal.

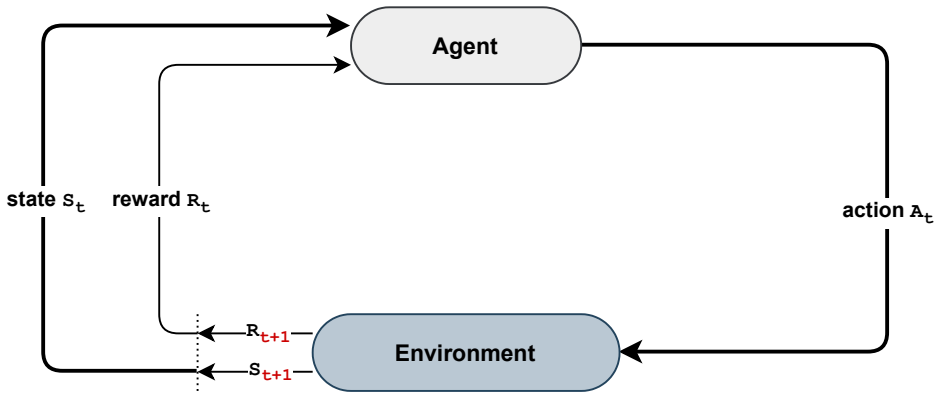


Figure 2.2: A simple reinforcement learning model. The agent performs an action in the current state, then receives from the environment the next state as well as a reward (evaluating its previous action).

- **Value (V), Q-Value (Q)** - Value $V_\pi(s)$ is the expected discounted reward for an agent that obeys a policy π at the current state s. Q-Value $Q_\pi(s, a)$ is similar to Value, but it also takes into account the current action a. While reward is an immediate score received in a given state, both Value and Q-Value represent long-term expectations, which every RL algorithm tries to maximize.

Figure 2.2 shows the interaction cycle between an agent and the environment. The next subsections will provide a detailed explanation of reinforcement learning, mathematically modeled as a Markov Decision Process (MDP).

2.1.3 Markov Decision Process (MDP)

In order to mathematically represent a decision-making problem, researchers constructed a framework that can fully describe environments using state-transition probabilities. Therefore, at every time-step, given a state and an action, we can predict the reward and the next state. The outcomes may or may not be stochastic (contain randomness). The original configuration of an MDP consists of a tuple with the following five elements:

$$MDP : (S, A, P, R, \gamma) \tag{2.1}$$

where S is the *state space*, A is the *action space*, $P(s, a, s')$ is the *transition model* which defines the probabilities of transitioning from every state s to every successor state s' given the action a. $R(s)$ (or alternatively $R(s, a)$ or $R(s, a, s')$) is the *reward*

that the agent receives in its current state s, after performing an action a and leading to the state s'. γ is the *discount factor* required in order to differentiate the importance between short and long-term rewards.

MDPs contain an important property, the *Markov Property*, which indicates that each current state is only dependent on its immediate previous state (previous time-step). Essentially, this means that a single state contains all the information required, rendering it memoryless. This convention is respected throughout our custom environment.

2.1.4 Environment

The environment is the world in which the agent learns and behaves. There are no restrictions of how an environment can be designed. Depending on the requirements of the problem, it can range from a simple grid world, to an ATARI game emulator, or even complicated 3D highly realistic virtual worlds. Each environment has its own state space (S) and action space (A). As already mentioned in subsection (2.1.2), both spaces can be either discrete or continuous. Continuous state spaces are fairly common, since most real-life problems are continuous by nature. They can be generally used in various RL algorithms, but they can also be discretized with methods, such as tile coding.

A state is defined by an N-dimensional space vector, where N always depends on the problem. Higher-dimensional states greatly increase the amount of information that the agent can utilize, but at the same time, complexity rises exponentially, thus affecting training time and convergence. Ideally, the minimum amount of information is preferred (to decrease complexity), which is enough for the agent to learn efficiently. Additionally, the type of information included needs to be relevant to the design of the reward system in order for the agent to discover behavioral patterns. For example, in an autonomous driving situation, where we penalize the agent for colliding with obstacles, a state naturally needs to include information, such as velocity and distance measurements from the vehicle's sensors.

2.1.5 Rewards

Rewards are an integral part of a reinforcement learning problem. Positive rewards motivate the agent to repeat an action in a similar situation, while negative rewards teach the agent to avoid a certain behavior. The general goal is to maximize the expected cumulative reward, that is, the sum of rewards over each time-step:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \dots \tag{2.2}$$

However, there are two problems with this approach. Firstly, the *trajectory*, that is, the sequence of states starting from an initial state leading all the way to a terminal state, may contain an infinite amount of states (without ever reaching a terminal state), thus an infinite amount of rewards. This causes the expected cumulative reward to "explode" to infinity. Secondly, future rewards are not as concrete as the present ones, which instead hold more accurate reward information. This is due to uncertainty introduced in the environment as time passes.

To force an infinite sum to converge to a finite number and to reduce the importance of future states, a *discount factor* γ is introduced, in which each reward (state) is multiplied by a factor of γ per time-step. The discount factor has a range of $[0, 1)$. Values close to one (1) render future rewards equally important as the reward in the current state, while values close to zero (0) considers rewards near the present state much more valuable. Thus, after tweaking our initial equation, our goal is to maximize the expected **discounted** reward:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i} \tag{2.3}$$

Designing an efficient reward system

Typically, the hardest part of a reinforcement learning problem is the design of the reward system, since there are no rules, no absolute restrictions and it is tied up with how the state space is defined. Different reward systems for the same problem can affect the agent's behavior, training speed (therefore convergence), and the ability to prevent local optima scenarios.

It is basically an optimization problem, which is solved by trial and error. Nevertheless, there are some known guidelines, which can be very useful. Firstly, it is not recommended to build sparse reward functions, even though they are easier to define (e.g return +1 if you win the game, else -1). This can dramatically slow down the learning process, because the agent needs to explore much further and perform many actions before getting any reward. Secondly, in continuous state spaces, where it is rather inefficient to create a table representation for rewards, continuous and differentiable reward functions, such as polynomial functions are desired, because they produce a much more gradual reward path over states. Last, but not least, in real world problems, where time is an important factor, rewarding the distance to the goal is probably not the most efficient choice, instead, higher dimensional variables, such as velocities, can be included to incorporate the sense of time.

2.1.6 Bellman's Equations

As a reminder, the agent's goal is to find an optimal policy π^* that determines the best sequence of actions for the agent and therefore maximizes the expected discounted cumulative reward, as stated in Equation (2.3). We can rewrite this equation with a recursive relationship:

$$\begin{aligned} G_t &= \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i} \\ G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \\ G_t &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \dots) \\ G_t &= r_{t+1} + \gamma(r_{t+1+1} + \gamma r_{t+1+2} + \dots) \\ G_t &= r_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.4}$$

Now that the recursive property is apparent, we can proceed to the definition of the value function. The *value function* $V_\pi(s)$ assigns values to states and is a measurement for our states. Mathematically, the value is an expected ($\mathbb{E}$) discounted total reward starting from a particular state s when the agent is following a policy π:

$$V_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i} \mid S_t = s \right] \tag{2.5}$$

We can extend the definition of the state-value function to include state-action pairs. This function is also known as *quality function* or *action-value function* $Q_\pi(s, a)$.

$$Q_\pi(s,a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i} \mid S_t = s, A_t = a \right] \tag{2.6}$$

We can relate the value function (2.5) with the quality function (2.6) in just a few steps. The sum of probabilities of all possible actions $a \in A$ from a state $s \in S$ equals 1:

$$\sum_a \pi(a \mid s) = 1$$

where $\pi(a \mid s)$ is the transitional probability of the policy selecting the action a given a state s. The value function, then, is essentially the sum of the transitional probability multiplied by the Q-value function over each action a:

$$V_\pi(s) = \sum_a \left(\pi(a \mid s) \cdot Q_\pi(s,a) \right) \tag{2.7}$$

We can now reach our initial goal by solving (2.5). But, instead of summing over multiple time steps, we can break down our complex value function into two simpler recursive subproblems to find the optimal solution using the recursive property of (2.4). This is exactly what the famously known *Bellman Equation* achieves.

- **Bellman's Equation for V_π:**

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\ &\overset{(2.4)}{=} \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} P\left(s' \mid s, a\right) \left(R\left(s,a\right) + \gamma \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s' \right] \right) \\ &= \sum_a \pi(a \mid s) \sum_{s'} P\left(s' \mid s, a\right) \left(R\left(s,a\right) + \gamma V_\pi\left(s'\right) \right) \end{aligned} \tag{2.8}$$

- **Bellman's Equation for Q_π:**

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] \\ &\overset{(2.4)}{=} \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a \right] \\ &= \sum_{s'} P\left(s' \mid s, a\right) \left(R\left(s,a\right) + \gamma \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s' \right] \right) \\ &= \sum_{s'} P\left(s' \mid s, a\right) \left(R\left(s,a\right) + \gamma \sum_{a'} \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s', A_{t+1} = a' \right] \right) \\ &= \sum_{s'} P\left(s' \mid s, a\right) \left(R\left(s,a\right) + \gamma \sum_{a'} \pi\left(a' \mid s'\right) Q_\pi\left(s', a'\right) \right) \end{aligned} \tag{2.9}$$

Although the above equations assume a reward function which depends on the current state and action taken $R(s, a)$, any simpler or general form of reward can be used. Bellman's equations are linear systems and can thus be expressed in a matrix form. This is convenient, since both values can be estimated either directly or recursively in a tabular form.

Optimal Policy

Although we learned how bellman's equations are defined for a given MDP, the estimation of the optimal policy is unknown and still our main objective.

A value function is **optimal**, if it yields the maximum value compared to any other value functions, such that:

$$V_{\pi}^{\star}(s) = \max_{\pi} V_{\pi}(s)$$

Similarly, the optimal state-action value function (Q-function) denotes the maximum reward received starting from a state s and taking action a:

$$Q_{\pi}^{\star}(s, a) = \max_{\pi} Q_{\pi}(s, a)$$

Between two policies π_1 and π_2, the better policy is the one whose value function is greater throughout its states. Mathematically, this can be written as:

$$\forall s, \pi_1 \geq \pi_2 \text{ if } V_{\pi_1}(s) \geq V_{\pi_2}(s)$$

A policy however, can be optimal, only if it yields an optimal value function. Multiple optimal policies can also exist, leading to the same optimal values.

In case of optimality, Bellman's Equations (2.5) & (2.6) are slightly tweaked, such that, instead of averaging over the agent's action, we select the action with the maximum value.

$$V^{\star}(s) = \max_{a} \left(\sum_{s'} P(s' \mid s, a) \left(R(s, a) + \gamma V_{\pi}(s') \right) \right) \tag{2.10}$$

$$Q^{\star}(s, a) = \sum_{s'} P(s' \mid s, a) \left(R(s, a) + \gamma \max_{a'} Q_{\pi}(s', a') \right) \tag{2.11}$$

Equations (2.10) and (2.11) are also known as *Bellman Optimality Equations.*

Lastly, the optimal policy is estimated through the value function by picking the greediest action, since in every state there is at least one optimal action leading to the maximum value.

$$\begin{aligned}\pi^{\star}(s) &= \arg\max_{a}\left(\sum_{s'} P\left(s' \mid s,a\right)\left(R\left(s,a\right)+\gamma V^{\star}\left(s'\right)\right)\right)\\ &= \arg\max_{a} Q^{\star}(s,a)\end{aligned} \tag{2.12}$$

Unfortunately, the linearity of this system breaks down in Bellman optimality equations, due to the introduction of the max operator. As a consequence, both values can only be estimated using iterative methods and techniques, such as linear programming.

2.1.7 Algorithms Taxonomy

There is currently a large variety of reinforcement learning algorithms in existence, each with its own characteristics and approach to solving a RL problem. These algorithms can be classified from different perspectives, such as model-based and model-free methods, value-based and policy-based or on-policy and off-policy methods.

Figure (2.3) presents a structured map of the most popular reinforcement learning algorithms.

2.1.7.1 Model-Based vs Model-Free Algorithms

MDP-based algorithms can be roughly divided into Model-free and Model-based methods. Generally, maximizing the rewards for our actions, depends on the policy and the model which needs to be known.

In *model-free* RL, we can ignore the model, since we depend on sampling and simulation to estimate rewards. Essentially, this means the agent learns from experience, performing actions directly in the real world (or simulation) and collecting reward live from the environment in order to update its value function. Since the interaction is performed live, these algorithms have irreversible access to the MDP. If an action is performed in a given state, it permanently affects the environment. Model-free algorithms are usually slower to learn, but also more reliable.

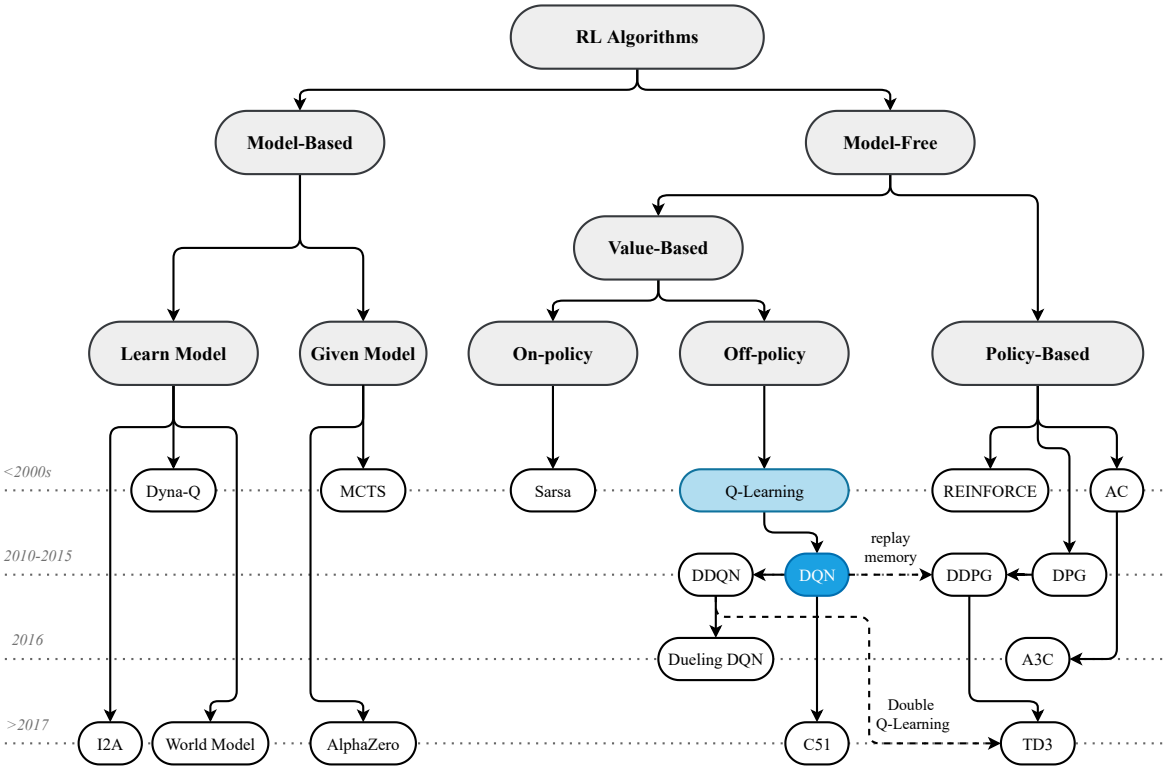


Figure 2.3: Taxonomy of reinforcement learning algorithms. Q-Learning and DQN algorithms will be analyzed and are highlighted in blue.

In *model-based* RL, we can calculate the optimal actions from the model directly, using a custom-defined cost function. This family of algorithms uses a reduced number of interactions with the real environment during the learning process in order to construct an model. It then uses this model (either known or learned) in order to **internally** simulate the subsequent episodes and not by applying them to the real environment. This is often referred to as *planning*. Since planning is performed internally, it has reversible access to the MDP, which allows the algorithm to repeatedly plan forward from the same state (evaluating different future paths each time). Another point worth noting, is that in this case we optimize the trajectory for the least cost instead of the maximum rewards. Model-based algorithms generally have the advantage of speeding up the learning, since no real interaction with the environment is required. There is a risky trade-off though, if the model is not accurate, it might lead the agent to learning undesired behavior. Popular model-based algorithm examples include Monte Carlo Tree Search, I2A and state-of-the-art AlphaZero.

2.1.7.2 Policy-Based vs Value-Based Algorithms

Model-free algorithms can be further categorized to *value-based* and *policy-based*, depending on which optimal element we are trying to estimate.

In policy-based methods, a representation of a policy is explicitly built and kept in memory during learning. The optimal policy is then computed by updating the policy directly. These algorithms support high dimensional and continuous action spaces and are also able to learn stochastic policies.

In value-based methods, no policy is defined. Instead, only a value function is stored and learned. The policy is used indirectly by picking the action with the best value. Value-based methods can be further categorized to *on-policy* and *off-policy*, depending on how the Q-Value is calculated.

There are also algorithms that combine policy-based and value-based methods, such as Actor-Critic which has both a value function and a policy function.

2.1.7.3 On-policy vs Off-policy Algorithms

There are two phases of an RL algorithm. The *learning phase*, which refers to the process of training the algorithm and the *inference phase* in which a trained algorithm is used to make a prediction. The distinction between on-policy and off-policy only affects the learning phase.

An off-policy algorithm has two different policies, one for each of the two phases (behavioral policy and optimal policy). The behavioral policy is used to select actions, while the latter is the policy which the agent tries to estimate (e.g Q-Learning, DQN and its variants).

On-Policy algorithms evaluate and improve the same policy, which is being used to select actions (e.g SARSA).

2.1.7.4 Exploration vs Exploitation

When performing the learning process, agents can have either of the following behaviors:

- **Exploration** - The agents chooses a random action (sub-optimal). This is useful to explore the environment and gain experience by visiting unexplored states.

- **Exploitation** - The agent chooses the greediest action, the best action it can take according to its current experience level.

As already mentioned, during the training phase, the agent tries to estimate the optimal value (or policy) function. To maximize the cumulative discounted reward, the agent theoretically needs to always exploit in order to reach the maximum sum of rewards. However, at the beginning, the agent has a very poor experience level. By exploiting very early in the learning process, the agent reaches a sub-optimal behavior, since it never had the opportunity to visit other unexplored states of the problem. This challenge is known as the *exploration-exploitation trade-off*.

To overcome this challenge, researchers introduced the *epsilon* (ϵ) parameter. This parameter has a range of $[0, 1]$ and it denotes the probability of the agent exploring (performing a random action). There are known policies, which use this parameter with various approaches, with a common purpose of mitigating this problem:

- **Greedy-Epsilon Policy** - The agent exploits $(1 - e) \cdot 100\%$ of the time, and explores $e \cdot 100\%$ of the time (assuming the randomness follows a uniform distribution between time-steps).

- **Linear-Decaying Greedy-Epsilon Policy** - This policy is similar to the greedy-epsilon policy, but the epsilon parameter slowly decays from a start value to an end value over k steps.

- **Exponential-Decaying Greedy-Epsilon Policy** - This policy is similar to the linear-decaying greedy-epsilon policy, but the decay is exponential and controlled by a *decay factor*.

- **Stretched Exponential-Decaying Greedy-Epsilon Policy** - This policy is similar to the exponential-decaying greedy-epsilon policy, but there is a larger opportunity for exploration at the initial part of the training and more room for exploitation towards the end of the training session. This is done by narrowing the transition between exploration and exploitation.

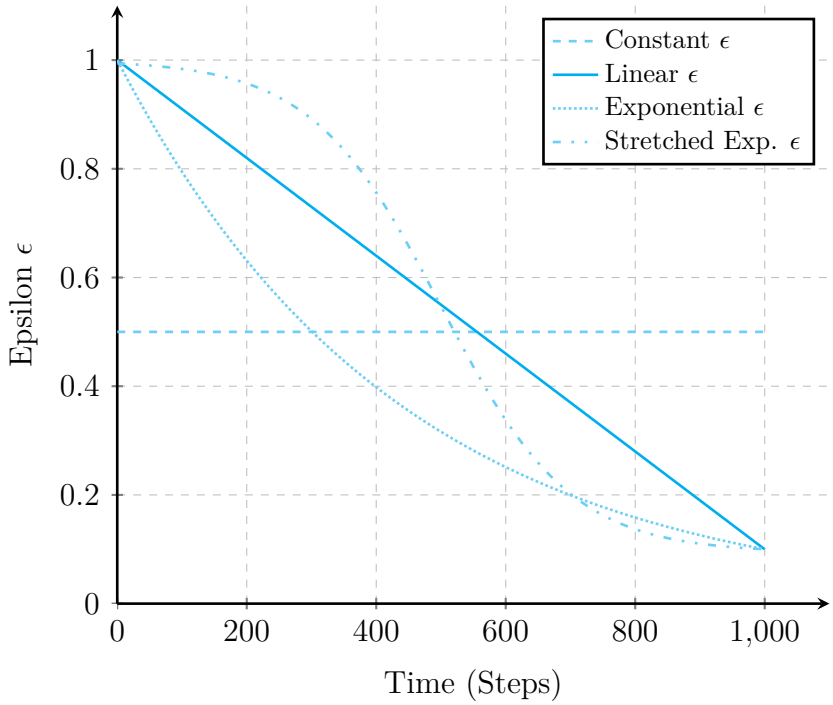


Figure 2.4: Plot showing how the ϵ-parameter is affected (decayed) by different epsilon-greedy policies with a starting value of 1 and a terminal value of 0.1 over 1000 steps.

- **Softmax Policy** - The agent mainly explores, but instead of sampling the actions from a uniform distribution, it samples from a custom distribution biased for more preferable actions. However, this policy is more complicated and is out of the scope of this thesis.

Decaying greedy-epsilon policies are practically useful, since they offer the opportunity for exploration at the initial part of the training process (in order to increase the experience level of the agent), so it can later reach the optimal behavior by mostly exploiting. Figure (2.4) shows how the epsilon parameter is affected over time for each mentioned policy.

2.1.7.5 Temporal Difference Learning

Temporal Difference (TD) learning is a popular class of model-free algorithms that estimates the policy by gradually updating the estimate until it converges. Unlike a dynamic programming approach, TD algorithms are able to learn directly from raw experiences without the knowledge of the environment's model. TD algorithms do not

estimate the Q-value from scratch at every time-step. They instead follow a general update rule to gradually update the Q-function which has the following structure:

$$Estimate_{new} \leftarrow Estimate_{old} + StepSize \underbrace{(Estimate_{target} - Estimate_{old})}_{\text{error}} \tag{2.13}$$

The *error* is defined as the deviation between the target and the actual value and is continuously reduced by taking small steps towards the target estimate. The StepSize however, which will be later referred to as the *learning rate*, is a hyperparameter that controls the amplification of the response to error estimates. In Equation (2.13) for example, a larger StepSize would push the estimate closer to the target. Estimating the correct learning rate can be challenging. A value too small may result in a long training process, while a value too large can cause instability issues. It is often recommended to decay the learning rate as time passes, since large correction steps are required at the beginning, with smaller fine-tuning steps needed towards the end.

2.1.7.6 Introduction to Q-Learning

Q-Learning [1] is a TD off-policy algorithm, which is used to find the optimal policy using a Q function. Essentially, it is the process of iteratively updating Q-Values for each state-action pair, using the Bellman Equation until the q-function eventually converges to the optimal Q-value function $Q^\star$. It is considered off-policy, since the approximation of the optimal Q-value does not depend on the current policy.

This iterative update method uses a *Q-table*, a large table of states and actions in which Q-values for each state-action pair are stored. All values are firstly initialized to zero, though upon playing, the agent continuously observes the reward and state transition and estimates the updated Q-value. The TD update step is the following:

$$Q : S \times A \rightarrow \mathbb{R}$$
$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left(r_{t+1} + \gamma \max_a Q\left(s_{t+1}, a\right) - Q\left(s_t, a_t\right)\right) \tag{2.14}$$

Q-learning is further analyzed in Algorithm (1).

Algorithm 1: Q-Learning (TD, off-policy)

Data:
$\alpha \in (0, 1] \rightarrow$ learning rate (intensity of value updates)
$\epsilon \in (0, 1] \rightarrow$ greediness (probability of random action)
$\gamma \in (0, 1] \rightarrow$ discount factor (importance of future rewards)

$Q(s, a) = 0, \forall s \in S, \forall a \in A$ `// initialize Q-Table`

foreach *episode* **do**
  $s_t \leftarrow s_0$ `// initialize state`
  foreach *step of episode* **do**
    $a_t \leftarrow \pi(s_t)$ `// select action` $a \in A$ `according to current policy (e.g epsilon-greedy policy)`

    Apply action a_t, observe reward r_{t+1} and next state s_{t+1}
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$
    $s_t \leftarrow s_{t+1}$ `// next state is now the current state`

  end foreach
end foreach

2.1.7.7 Function Approximation

So far, Q-Learning estimates the values in a tabular form, gradually updating each state-action pair in the Q-table. In a way, we memorize every single combination of states and actions in our environment. This is acceptable for cases where our environment only consists of a few states and actions. However, as already mentioned in Section (2.1.4) on most occasions, the state space is concerningly large and complex. This is a well-known challenge in reinforcement learning, referred to as *curse of dimensionality* and indicates the exponential growth of state complexity over the number of dimensions. In fact, real-world problems naturally also contain continuous state spaces, which require massive amounts of memory and time to "support" the Q-table, thus rendering the tabular approach incredibly inefficient.

To overcome this challenge, researchers came up with a mathematical technique, known as *Value Function Approximation*, which aims to generalize the estimation of the value at states that have similar features. Although "generalizing" usually means some loss of information, it is surprisingly effective and far more efficient than any

tabular method. There are a variety of function approximation methods used, both linear (e.g polynomials) and non-linear, such as neural networks (NNs).

2.1.8 The Deep Reinforcement Learning Family

As already mentioned, reinforcement learning algorithms, such as Q-learning, rely on tabular methods to store and update the Q-values. This is satisfactory in problems where the state space of the environment is relatively small, but fails miserably as the environment becomes more and more complicated. This inefficiency is caused by the fact that tabular methods essentially keep duplicate information about the environment, in the sense that they cannot correlate similar states and generalize past experiences. Humans and other animals, on the other hand, overcome this problem by combining incredibly optimized reinforcement learning along with state-of-the-art image processing systems.

Thus, a more efficient representation of the environment from high-dimensional sensory inputs is required. Early solutions proposed the use of linear function approximators to generalize the available information (as mentioned in Section 2.1.7.7), but their implementation requires careful study of each specific problem. As it turns out, non-linear function approximators and especially neural networks generally offer far greater feature generalization, which leads us to investigate the subfield of Deep Learning.

2.1.8.1 Deep Learning Overview, Neural Networks and Concepts

Although deep learning's basic concepts are traced back to the early 1960s, their impact wasn't evident enough until the early 2010s where large-scale industrial applications of deep learning began to formulate. This delay occured for a few reasons. As we will later find out, deep learning techniques require huge amounts of data to process. By observing the timeline of the digital revolution, previously there just wasn't enough available data to utilize. In addition, processing this data proved to be incredibly time-consuming and simply unfeasible, considering the computational power at the time. Advances in hardware in the following years had driven renewed interest in deep learning, especially

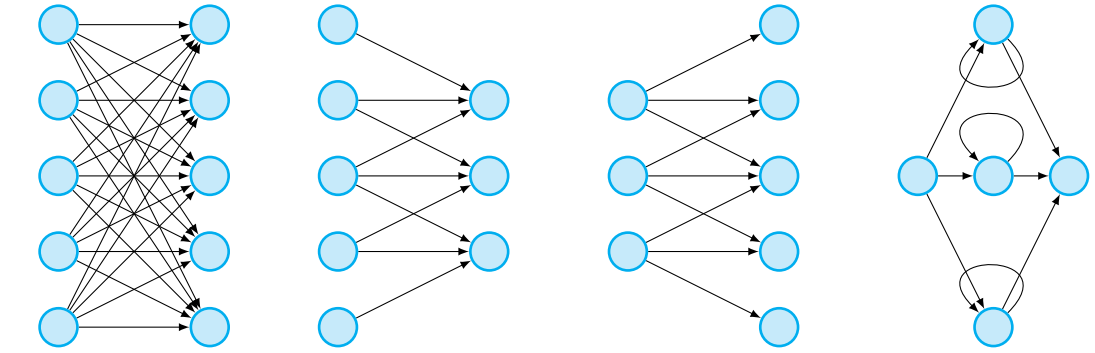
after NVIDIA's breakthrough in graphics processing units (GPUs) in 2009, which enabled training of deep-learning systems and saw increased speeds of 100 times due to their ability for fast matrix/vector computations required for machine and deep learning.

The deep learning field is considered as the succession of artificial neural networks (ANNs), which were inspired by neuroscientist researchers. An *artificial neuron* is the fundamental block of deep learning, which simulates the neurons of our biological brain. Each articifial neuron has several inputs, along with a bias input that functions as a weighted sum. These inputs are then passed through an activation function and thus determine the output of the neuron. By arranging a large number of these neurons in a chain structure, we are able to construct a *neural network*. A neural network with a single artificial neuron is called *perceptron*.

The neural network is used to approximate non-linear functions. The network's performance is dependent on the weight parameters used, which is an iterative process of continuous updates that we refer to as *learning*. The neural network consists of multiple *layers*, namely, groups of neurons that exist at the same level. The connection between two consecutive layers can vary. Four of the most popular layer connection types are explained below.

- **Fully Connected Layer** - Fully connected layers connect every neuron from a layer to every neuron of the next layer. Since the amount of connections are generally large, they can be computationally expensive. They are mainly used in CNN's for image classification in computer vision or machine learning.

- **Convolutional Layer** - Convolution layers are the basic ingredients of any CNN or FCN. It uses a *filter* or *kernel* to scan an image and perform *convolutions*, a linear operation which multiplies (dot product) a set of weights (filter) with the input. The filter is intentionally smaller than the input in order to be multiplied by the input array multiple times at different points on the input. This is generally useful to detect features in images. It should be noted that the output layer is smaller than the input layer, since we are essentially downsampling information.



(a) Fully Connected Layer **(b)** Convolution Layer **(c)** Deconvolution Layer **(d)** Recurrent Layer

Figure 2.5: Popular types of neural network layer connection

- **Deconvolutional Layer** - Deconvolution layers are the opposite of convolutional layers. It is mainly used to upsample data to a higher resolution. It can also be found in a neural network next to proceeding convolutional layers, to restore the information to match the input size.

- **Recurrent Layer** - The main differentiator of recurrent layers is their ability to support neuron looping, a feature that enables each neuron to set as an input its own output. This essentially provides *memory*, which is especially useful in cases involving sequential data, such as natural language and time series.

Figure (2.5) shows an overview of the mentioned types of layer connections.

Multiple consecutive layers can be connected with different combinations, eventually forming an *architecture*. There are many known architectures, such as multi-layer perceptrons (MLPs), stacked autoencoders (SAEs), convonutional neural networks (CNNs), deep belief networks (DBNs), recurrent neural networks (RNNs) and generative adversarial networks (GANs), each of which is optimized for a specific set of tasks.

Before proceeding to the deep reinforcement learning subfield, some deep neural network concepts require explanation.

- **Feed Forward** - Feed Forward is a neural network type in which information only travels forward, from the input nodes (x), all the way to the output nodes. This network can approximate a function: $\hat{Y} = f(x, \theta)$.

- **Activation Function** - In order to approximate the non-linear function, an activation function is used in the forward propagation phase. Activation functions can be either linear or non-linear and are used to control the neuron's outputs. Sigmoind, TanH and Rectified Linear Unit (ReLU) are three of the most widely used activation functions.

- **Loss** - Loss is a metric which evaluates a model's prediction by comparing the estimated value $\hat{Y}$ with the target value Y. There is a variety of loss functions available, each optimized for a specific task. The most popular loss function is *mean squared error*, which calculates the squared distance of the target and the predicted value.

$$MSE\left(\hat{Y}, Y\right) = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{y}_i\right)^2 \tag{2.15}$$

- **Back-Propagation** - After we calculate the loss, we need to carefully "correct" our network weights in order to eventually minimize it. Back-propagating solves the first part of the problem. Assuming a neuron has two inputs x, y and an output z, we can define it as $z = f\left(x, y, \theta\right)$ and its local derivatives (determined during the forward pass) are $\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}$. The local gradient of the loss is estimated during the back-propagation phase, by multiplying the derivatives with the local gradient $\frac{\partial loss}{\partial \theta}$.

- **Update** - After successfully estimating the gradients, the second step is to update all the parameters. The intensity of correction depends on how large is the deviation of the predicted value to the target value. This estimation is performed by an *optimizer*.

- **Optimizer** - Optimizers are algorithms used to minimize an error function. They focus on helping change the weights or learning rate of a neural network. Some popular optimization algorithms include Gradient Descend, Momentum, Adam, AdaGrad, RMSProp, etc.

- **Regularization** - Not all the information that enters a neural network is reliable. Some input data does not really represent the true state of an environment, causing the model to overfit, considering it's trying hard to construct a pattern. This challenge can be mitigated by the regularization technique, which decreases the importance (weight) of "problematic" data. A small sacrifice to flexibility is made to counter the possibility of overfitting.

2.1.8.2 DQN Algorithm

Deep learning has shown promise in extraction of high-dimensional features from raw sensory data, such as images, audio and video. In this case, different types of neural network architectures are used including convolutional networks, fully connected networks and recurrent neural networks. Therefore, incorporating deep learning techniques in reinforcement learning problems is a very intriguing approach in order to manage large state spaces.

This is exactly what Google's DeepMind research team managed to accomplish in 2013, where they achieved human-equivalent or even superior playing performance on a variety of Atari 2600 games by only using visual pixel information [2, 3]. The Atari games were part of the Arcade Learning Environment (ALE) platform also launched in 2013 aiming to speed up development of AI agents for this specific type of environments. DQN is now considered a successful expansion of Q-learning. Multiple variants of DQN have also emerged, such as Double DQN (DDQN) [4], Dueling DQN [5] and Prioritized DQN [6]. Each of these variants have slighlty different implementation methods, focusing on optimizing some aspects of the original version, but are out of this scope of this thesis.

During the training of our model, the aim is to construct a robust and generalized agent that does not depend on a specific environment. Though a challenge arises, if the network learned only from consecutive samples of experiences as they occured sequentially in the environment, the samples would be highly correlated and therefore lead to inefficient training. To surpass this issue, the team used a known technique called *experience replay*. During training, we define the agent's *experience*, that is,

a tuple containing the current state s_t, the action taken a_t, the reward received by taking this state-action pair r_{t+1} and next state it reaches s_{t+1}:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

This experience is then stored in a buffer called *replay memory*, which holds a particular amount of the agent's experiences. New experiences initially fill up the replay memory, then they overwrite older, poorer experiences. The network is now trained by randomly sampling from this replay memory, thus avoiding correlated sequential samples.

Deep Q-Learning with Experience Replay is shown in Algorithm 2

At the beginning, the network is initialized with random weights to break symmetry between different units. If two units with the same activation function and the same inputs have also the same initial parameters, it would cause the model to update both units equally. For every episode and time-steps in each episode respectively, an action needs to be selected,according to our current policy. Epsilon-greedy policies are typically used (see Section 2.1.7.4 for more information), which determine when to explore or exploit. After the action is selected, this action is applied to the environment. As a quick refresher, this is a model-free algorithm, meaning our actions are taking place in real time, without the ability to revert to a previous state. To complete the experience tuple, the reward and the next state are extracted from the environment.

After a random batch of experiences is sampled from the replay memory, we need to perform an essential pre-processing for the states to reduce the network's input size. In the original DQN paper for example, states represent entire RGB image frames of the ATARI game. Each frame consists of 160 horizontal frames, 210 vertical frames and a 128-color palette, thus a single state would contain over 100.000 inputs for the neural network. Instead, the RGB frame is converted into a grayscale version, downsampled to 110×84 pixels and cropped to a 84×84 region that captures the playing area, reducing the total number of inputs to 7056 (14 times smaller). An additional part of the pre-processing stage, is to keep the m most recent frames and stack them to

produce the final input of the neural network. Having multiple frames provides to the network useful information about the flow of time.

At this point, the training phase has begun. The preprocessed state is passed as input to the policy network, for the purpose of approximating the optimal policy (finding the optimal Q-function). The state data is forward- propagated through the network itself and the model then outputs an estimation of our Q-value for every available action. To calculate the loss, the model compares this estimation to a *target Q-value* for the same action.

$$\begin{aligned} loss &= Q^{\star}(s,a) - Q(s,a) \\ &= \underbrace{\mathbb{E}\left[R_{t+1} + \gamma \max_{a'} Q^{\star}(s',a')\right]}_{\text{optimal Q-value}} - \underbrace{\mathbb{E}\left[\sum_{i=1}^{\infty} \gamma^{i-1} R_{t+i}\right]}_{\text{estimated Q-value}} \end{aligned} \tag{2.16}$$

To calculate the optimal value, we need to compute the following term:

$$\max_{a'} Q^{\star}(s',a')$$

The Q-value of the next state s' along with the next action a' are unknown, which is why we need to perform a second pass to the neural network with the next state as input (this is why an experience e_t also holds the next state and action pair). With the second pass, we obtain the maximum Q-value among the next actions, just to compute the loss of our original state-action pair.

However, this is where another potential danger can emerge. The second pass is performed using the same network as the first pass, thus the same network weights. At each iteration the weights of the network are updated to eventually minimize loss, causing our estimated Q-values to approach closer to the target values, but then the target Q-values are also updated and pushed further away. This dog-chasing-its-own-tail problem creates instability issues for our training. The team introduced a second network called the *target network*, which is a clone of the policy network. Its weights are frozen with the original policy network's weights and are only being updated every C amount of steps. The second pass is now performed on the target network, greatly contributing to a converging training session. The final step of the pipeline is to update the weights using a gradient descent in order to minimize the loss.

Algorithm 2: Deep Q-Learning with Experience Replay

```
Initialize replay memory D with capacity N
Initialize the policy network with random weights θ
Initialize the target network with weights θ⁻ = θ
foreach episode do
    Initialize the starting state
    Initialize the preprocessed history sequence of states
    foreach step of episode do
        Select action a_t according to policy (explore or exploit)
        Apply action a_t in the environment in current state s_t
        Observe reward r_{t+1} and next state s_{t+1}
        Perform pre-processing of state and next state
        Store experience tuple (s_t, a_t, r_{t+1}, s_{t+1}) in replay memory
        Sample random mini-batch (s_t, a_t, r_{t+1}, s_{t+1}) from replay memory
        Pass batch of preprocessed states to policy network
        Calculate loss between Q-values and target Q-values
        Update network weights θ using gradient descent
        Every C steps, target network weights catch up to the policy network
    end foreach
end foreach
```

The ALE environment is capable of rendering images at high frame rates. If the reinforcement-learning-defined time-step lasted only as long as a frame, it would cause the agent to take actions very frequently, which is redundant. DeepMind's solution was to use *frame skipping*. By skipping frames, the algorithm only needs to perform these calculations every m frames, which reduces computational costs and helps the agent collect more experience.

One last optimization recommended is *clipping rewards*. As already mentioned in Subsection (2.1.8.1), regularization is a critical part of a neural network. Having unbounded rewards popes a similar threat, since it would produce inconsistent loss values and lead to a divergent training. A typical reward range would be $[-1, +1]$, where -1 denotes the worst possible reward, while +1 denotes a perfect situation.

2.2 Tools and Frameworks

This section analyzes all the frameworks and tools that were required throughout this thesis.

2.2.1 Robot Operating System (ROS)

The Robot Operating System (ROS) platform is a flexible collection of libraries, tools and conventions aiming to standardize the development of robotic applications. Right from the beginning, it was built with a modular ecosystem in mind, thanks to a unified communication system and its open source community that encourages collaborative work. ROS is based on a distributed architecture, in which a master *node* can communicate with secondary nodes and exchange real-time information about sensors and actuators.

ROS provides out-of-the-box several useful facilities, which can be used to speedup the robotic development. Its communication system involves a publishing/subscribing approach transferring asynchronous data, which enables for recording and playback of messages. The message-passing design pattern can offer automation dynamics, reduce the development effort and can improve the debugging experience. Although the messages are mostly asynchronous, there is support for synchronous *service* calls, which enhances the real-time interaction of its nodes. Additionally, there is a global parameter system which allows for parameterized configuration setup between tasks. Lastly, the ROS team built several specialized high-level robot-specific features in order to offload some of the effort by developers:

- Standard Robot Messages (template messages for communication)
- Geometry libraries, such as TF (links, joints, frames)
- Robot description language, such as Unified Robot Description Format - URDF which is an XML document containing important properties of each model (dimensions, sizes, location, sensors, visual appearance)
- Diagnostic tools, such as Rviz for visualization and debugging, as well as RQT for plotting.

The *middleware*, that is, ROS's internal core, achieves communication through the following components:

- **Node** - A node is an executable process within ROS. It can represent sensors and actuators, or control different part of the robot's decision-making. Nodes can communicate with each other through *messages* by publishing or subscribing to *topics*. Due to ROS's modular nature, each node should have a very specific purpose to improve reusability.

- **Message** - A message is a container of information transferred between nodes. It is tied to a specific data type and holds all the important data, such as sensor measurements of the robot. Developers can either utilize existing standard robot messages or create custom messages with simple or complex structure (messages containing other messages).

- **Topic** - A topic is the place where messages are being published. Nodes can publish a message to a topic to post information, or can otherwise subscribe to a topic to become informed about its messages. This system is especially flexible, allowing multiple nodes to be subscribed to multiple topics concurrently (by running on different threads).

- **Service** - A service is an additional communication method. It is synchronous and bidirectional, since it allows a node to send a request and receive a response. While topics and messages are used for continuous data streams, services should be used for remote procedure calls that terminate quickly, ideal for specific single tasks.

2.2.2 Gazebo Simulator

Robot simulation is an essential tool for the development of robotic applications. It is used to represent a physical robot, but in a virtual world, without depending on the actual machine. Gazebo is a well-known, fully-ROS-integrated simulator, capable of high-accuracy representation of robots in complex environments. Gazebo supports multiple high-performance physics engines, including ODE, Bullet, Simbody and DART, as well as realistic environments using the OGRE graphics rendering engine. Since it has an open source community, users can choose between a variety of predefined

Figure 2.6: A Gazebo world.

robot models or design their own custom models using SDF. There are several types of sensors available, including laser rangefinders, cameras, kinect, contact and force-torque sensors. Noise models can be optionally added on each sensor, further simulating real world scenarios. A Gazebo world can be seen in Figure (2.6).

2.2.3 OpenAI Gym

OpenAI is a research lab focusing on Artificial Intelligence, founded by Elon Musk in 2015. As a competitor to Google's DeepMind, OpenAI's research is primarily focused on Reinforcement Learning. In late 2016, they released Gym, a reinforcement learning toolkit with a modular structure, which enables the development and comparison of RL algorithms in a variety of environments. The provided environments implement a similar interface despite spanning from Atari games to classic control and 2D/3D worlds.

2.2.4 Tensorflow and Keras

Tensorflow is an open source platform that specializes in machine learning development. It provides a flexible ecosystem of tools and libraries, enabling researchers to easily build and train machine learning models.

Keras is a python-oriented deep learning API. It is powered by Tensoflow, meaning it still provides its powerful features, but focuses on delivering fast and easy access

to deep learning experimentation. Keras is designed to be simple and efficient, by providing essential abstractions and building blocks for machine learning projects to offload significant development effort.

2.3 Sensors

Every robot needs sensors in order to have situational awareness. Every sensor category specializes in a specific task. Camera and distance sensors provide important information about the robot's surrounding, while localization and motion detection can be achieved through satellite systems and internal measuring units respectively.

2.3.1 Optical Camera

What began as a live feed for remote control, has now evolved into an essential machine vision instrument, capable of applications, such as object recognition, classification, target tracking and depth estimation. An *optical camera* is a sealed box with a small hole (aperture) which allows light to pass through and capture an image on a light-sensitive digital sensor.

Cameras can be placed on the robot's fixed frame or can be mounted on a *gimbal*, a support mechanism, which can either provide additional stabilization (passive mechanical gimabls) or extend the object's rotational capabilities to provide more degrees of freedom (electronic motorized gimbals).

2.3.2 Distance Measurement Sensor

Another very popular category of sensors used in robotic application is laser sensors. They essentially emit pulses of light and can estimate a target's distance depending on the reflected beam off the surface. While single point laser rangefinders are relatively simple, to create perception on higher dimensional areas, the *Light Detection and Ranging (LIDAR)* technique is used. LIDAR emits laser pulses that move outwards in various directions reaching objects and then reflecting the light back to the receiver. This eventually creates a two- or even three-dimensional map of estimated ranges providing a richer source of information. LIDAR sensors have large measurement ranges, impressive

accuracy and extremely fast response times, since waves travel at the speed of light, making it incredibly useful in time-sensitive applications. It is worth noting however, LIDAR sensors are extremely expensive and are mainly used in high-end applications.

Sound Navigation and Ranging (SONAR) or ultrasonic technology is similar to LIDAR, but emits sound waves in order to estimate distances. It is a cost-effective solution for detecting and identifying objects and excels in marine applications, since sound waves are able to easily penetrate seawater. However, some accuracy and measurement range are definitely sacrificed.

2.3.3 Inertial Measurement Unit (IMU)

An Interial Measurement Unit (IMU) device is a complete package used in robotic applications that consists of *accelerometers, gyroscopes* and potentially *magnetometers*. Accelerometers are sensors responsible for measuring inertial acceleration, and change in velocity over time. Gyroscopes are devices that sense angular velocity, that is, the change in rotational angle over time. Magnetometers are types of sensors which can measure the strength and direction of a magnetic field. An IMU can efficiently combine the information of all three sensors and provide accurate motion sensing in virtually every robot's axis.

2.3.4 Global Navigation Satellite System (GNSS)

The *Global Navigation Satellite System (GNSS)* is a group of artificial satellites used for proving position and timing data. GNSS receivers are widely used in robotic applications, mainly for *localization* purposes, that is, the estimation of the absolute robot's position in an environment.

3
Problem Statement

Contents

After providing all the necessary background information, we can now state the problems which this thesis will attempt to resolve, comparing our approach to other widely implemented methods.

3.1 Problem Statement

Let's assume a three-dimensional environment filled by a number of randomly generated obstacles and a predefined number of *aruco markers*, which are representing our targets. More information about aruco markers can be found in Appendix A. Then, a UAV is spawned in a random location inside the environment. The goal for the drone is to detect and approach every hidden marker in the environment without crashing into any obstacle. No rules have been given to the agent about its objective or its environment whatsoever. Therefore, at the beginning, random actions are selected. After each action, the environment returns to the agent an evaluation of its selection, thus highlighting if actions were profitable or setbacks to the mission. After many

experiments, possibly leading to many collisions and dead ends in the process, the agent gradually begins to form a pattern over its actions, resulting in its improvement over time. Eventually, the agent learns how to safely navigate through the environment, successfully avoiding obstacles and reaching its targets. The output of this project can be a list containing the positions of every detected ArUco tag.

The drone used has in its possession an optical camera, a 2D LIDAR rangefinder for horizontal situational awareness, as well as two sonar sensors for vertical measurements.

3.2 Related Work

Several approaches have been used for UAV navigation tasks. The most common and well-known technique is Simultaneous Localization and Mapping (SLAM). This approach uses the UAV's onboard sensors to construct a real-time map (map-building) of the environment, while simultaneously estimating its relative position (localization). Multiple variations of SLAM have been investigated in research studies, including camera and laser scanner based SLAM [7, 8], ultrasonic based SLAM [9], landmark-based stereo-vision SLAM [10, 11] and even vision-only SLAM in GNSS-denied regions [12, 13]. This information can be later used in conjunction with several path-planning algorithms to determine the best route of the UAV to reach a specific goal.

While this is a complete and proven procedure, challenges arise when the environment becomes too complex. Firstly, the mathematical calculations required increase significantly which lowers the onboard real-time capability. Additionally, these algorithms only work when the environment is fully explored, namely, every decision of action is based on known pre-calculated signals.

Using "mapless" approaches, such as reinforcement learning can be very effective, where an exact model of the environment may not be available. These methods are offline, do not rely on GNSS systems to operate. Some studies [14, 15] implemented reinforcement learning to navigate UAVs in unknown environments. However, the curse of dimensionality problem mentioned in (2.1.7.7) becomes an apparent issue, when the environment becomes very large and complex. Instead, combining deep learning techniques and reinforcement learning has been repeatedly proven successful [16, 17, 18].

4
Approach

Contents

Based on previously presented concepts and our problem's statement, we can now introduce our autonomous UAV navigation approach. Initially, a detailed description of both our drone's models will be provided, as well as an overview of the creation of dynamically generated environments. Next, from a reinforcement learning perspective, this chapter will thoroughly examine our approach of integrating deep learning techniques into an MDP, by explaining the algorithm implementation,

observation extraction, states and actions definition and the concept behind our reward system. Finally, an overview will be presented regarding the neural network's composition, hyperparameter selection and several performance optimizations.

4.1 UAV Setup

In this section we introduce the basic principles of a drone's coordinate system and its sensor components.

4.1.1 Coordinate Systems and Transformations

Robotic models can be very complicated, consisting of a series of mechanical connections, such as *joints* and *links*. A joint provides relative motion between two links of a robot, unlocking a certain degree-of-freedom (DOF). Figure (4.1) depicts a robot's composition of two (2) joints and three (3) links. However, a dilemma arises when two sensors are placed on two different, non-parallel links regarding the way in which the data is mathematically represented. Obviously, a unified coordinate system is required in order to provide consistent representations of the data.

ROS's answer to this confusion, inspired by John Craig's *Introduction to Robotics* book [19], is the use of *frames* and *transformations*. Every sensor can provide measurements on its own frame, namely, its own local coordinate system (consisting of a set of orthogonal axes that can define positions and orientations relative to the attached object). A transformation, then, is a mathematical operation describing the offset (translational and rotational) between the two frames. Any frame can be described with respect to another through the use of transformations. There is also a *global frame*, typically referred to as the "world" frame, which represents the environment around the robot. Figure (4.2) shows a drone with its frames along with their correlations. The robot's frames can be generally represented in an expanding structure called *tf tree*. Figure (4.3) shows a TF tree containing the structuralized relationship of the drone's frames.

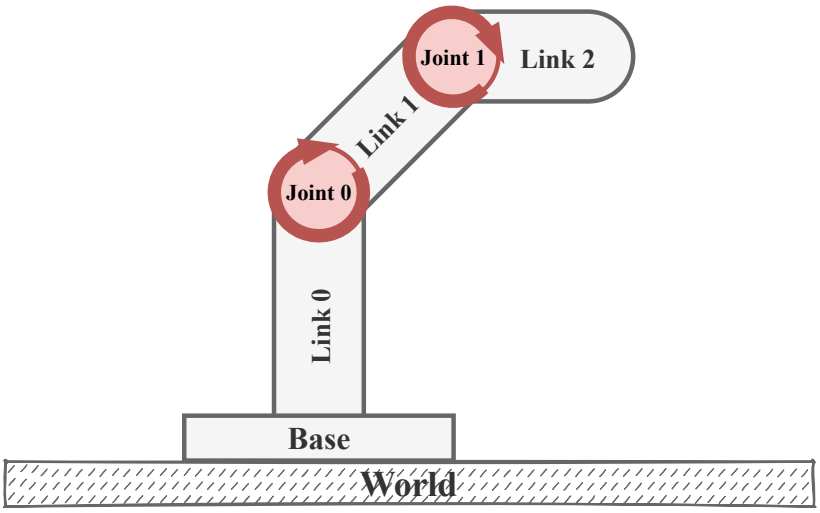


Figure 4.1: Very Simple representation of a robot containing links and joints.

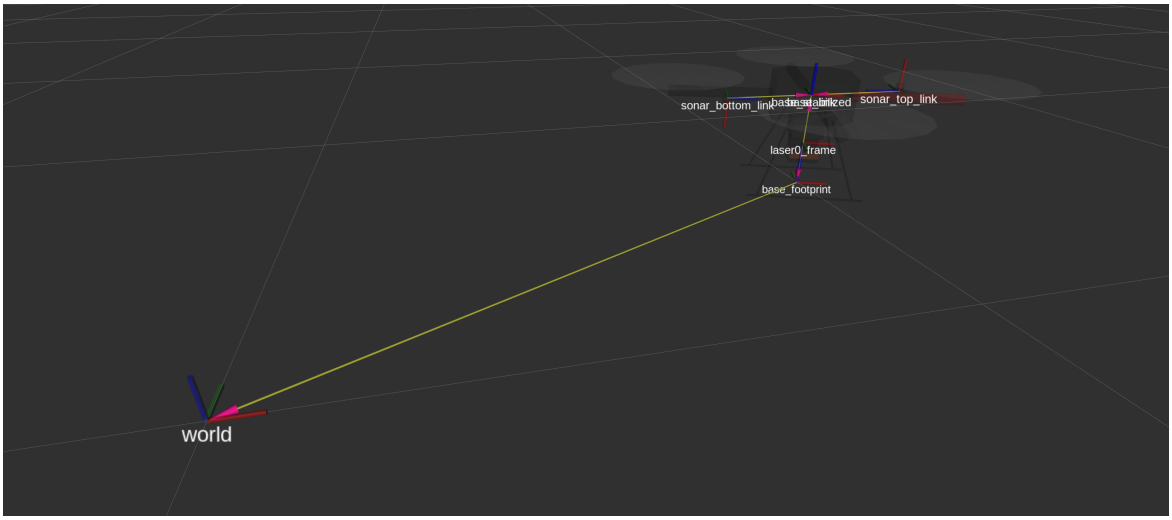


Figure 4.2: Each drone component has its own frame. Frames can be translated to each other using transformations, as appeared in RViZ visualization tool.

4.1.2 Basic Aircraft Principles

Drones and other aerial vehicles all obey to a fundamental aircraft principal rotational system consisting of three (3) axes. This configuration is derived from the right-hand rule, which serves as a common mnemonic to remember orientation of axes in a 3D space. Each axis is briefly explained below and graphically illustrated in Figure (4.4)

- **Yaw Axis (Normal)** - An axis drawn from top to bottom, allowing nose left/right rotation.

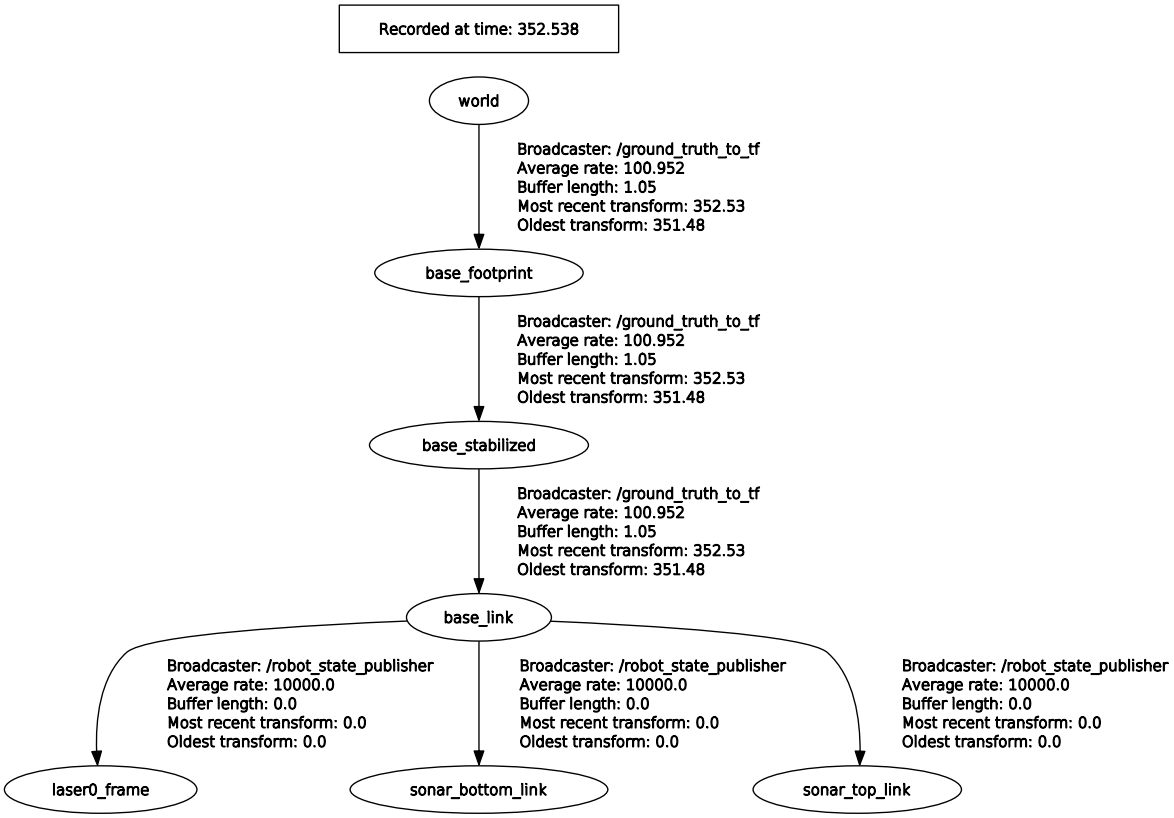


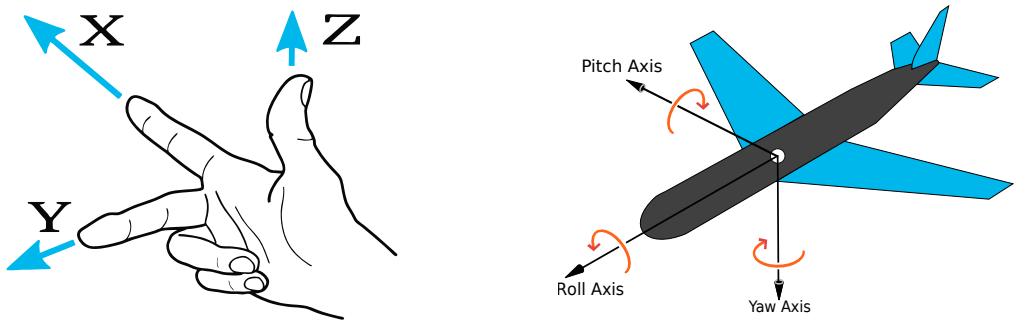
Figure 4.3: A TF Tree of the hector quadrotor drone consisting of multiple correlated frames. Both sonar and laser sensors have their own frames relative to *base_link*, a frame fixed to the robot's rigidbody. *base_footprint* indicates the projection of the robot's *base_link* to the ground. *world* is the global frame.

- **Pitch Axis (Lateral)** - An axis drawn parallel to an aircraft's wings, allowing nose up/down rotation.
- **Roll Axis (Longitudinal)** - An axis running through the aircraft's body from nose to tail, allowing nose clockwise/anti-clockwise rotation.

4.1.3 Sensor Modules

As already mentioned, every ROS package is based on manufactured counterparts, allowing projects to be physically ported to the real world. In this subsection we present the exact sensor modules used in this thesis.

[1]modified version of `https://en.wikipedia.org/wiki/Aircraft_principal_axes`
[2]modified version of `https://i.stack.imgur.com/0hxY1.png`



(a) Right Hand Rule [1]

(b) An aircraft's three rotational axes (roll, pitch, yaw) [2]

Figure 4.4: Basic aircraft principles

4.1.3.1 Optical Camera Module and 3D Gimbal

The camera module used is a pinhole generic camera sensor with configurable specifications. The camera's resolution was set to 1280×768 that captures RGB-format images, which mostly represents today's small-scale camera sensors. The image sensor reports its image planes to the *camera_optical_frame* which is translated into real coordinates through the *camera_frame*.

The camera unit was mounted on a custom gimbal mechanism, based on the *dji_m100_ros* package [20]. The gimbal is motorized and composed of three gimbal joints (roll, pitch, yaw), allowing for 3-DOF rotations. According to the study, each gimbal joint has a separate frame, connected and translated using the necessary transformations. Eventually, with a gimbal system, the optical camera becomes orientationally indipendent of its host, allowing for flexible object detection.

Camera Module Specifications

- **Camera Resolution** : 1280×768
- **Color Format** : RGB
- **Frame Rate** : 30 frames per second

4.1.3.2 Ultrasonic Module

Sonar sensors were used in this project for vertical obstacle avoidance (floors and ceilings). Sonars in ROS and Gazebo can be simulated with the *gazebo_ros_range*

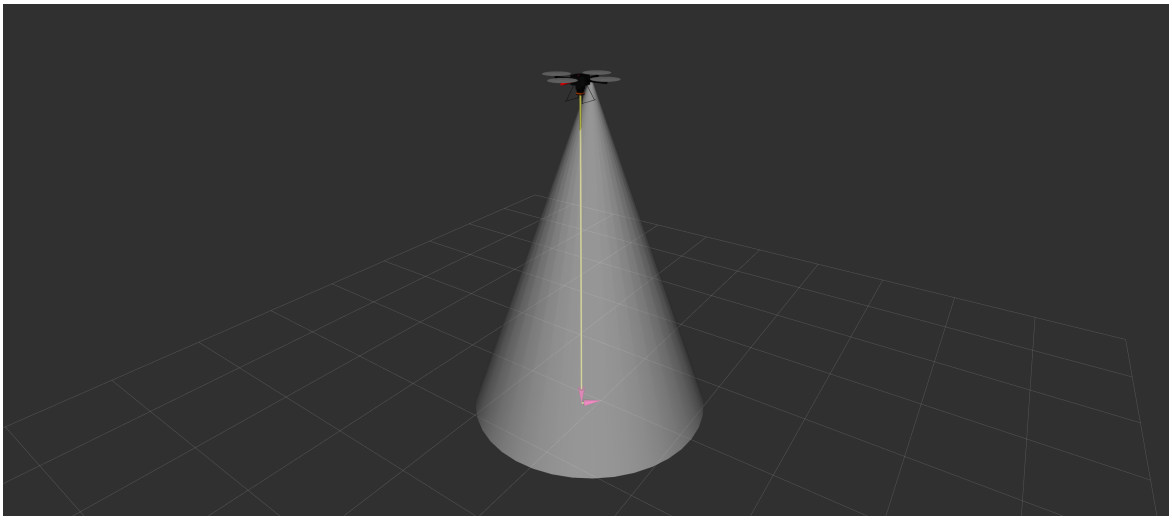


Figure 4.5: A sonar (ultrasonic) sensor is used to provide vertical obstacle avoidance. A cone-shaped structure represents the sonar's detection area as shown in RViZ.

plugin. Using sonar sensors for vertical measurements is convenient, thanks to the cone-shaped detection area, which offers increased field of view, as shown in Figure (4.5). Naturally, each sonar has its own frame, mounted directly above or below the UAV's *base_link* fixed frame. The plugin was configured using the following specifications, which typically represents real-world usage:

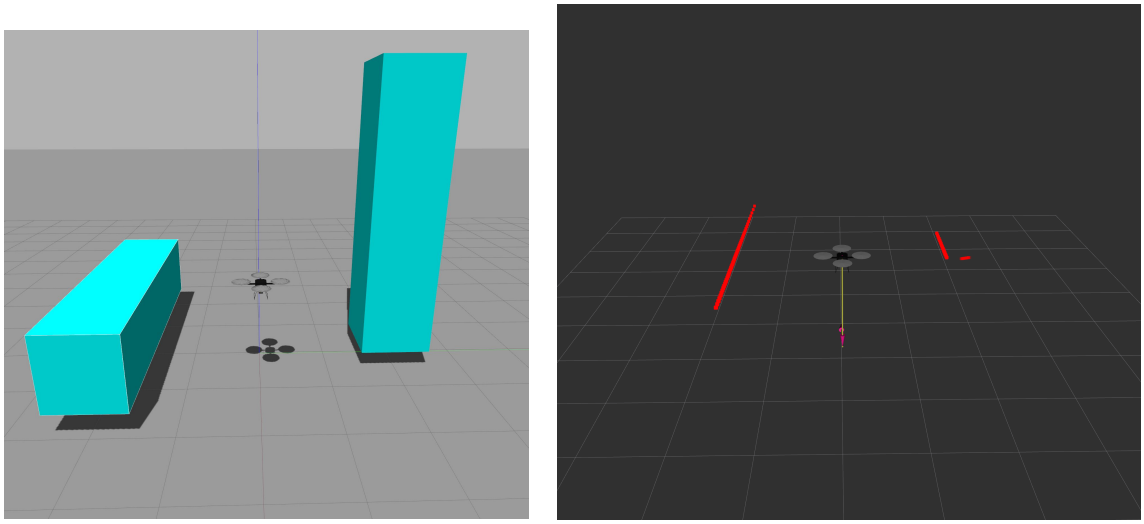
Sonar Module Specifications

- **Update Rate** : 30 measurement updates per second
- **Minimum Range** : 0.03 meters
- **Maximum Range** : 5 meters
- **Field Of View** : $^{40\pi}/_{180}$ degrees
- **Ray Count** : 3

4.1.3.3 2D Laser Rangefinder Module

To achieve horizontal obstacle avoidance, a two-dimensional LIDAR sensor was used (Figure 4.6). Specifically, we utilized the Hokuyo UTM-30LX-EW, which is one of the most popular LIDAR rangefinders for drone applications. The laser rangefinder's frame is *hokuyo_frame*. Considering the required transformations, surrounding distances can

Figure 4.6: A LIDAR sensor is used to provide horizontal obstacle avoidance in a two-dimensional plane.



(a) Gazebo world with a drone and two obstacles

(b) The distances measured can be represented in a local point cloud map using RViZ.

Figure 4.7: World with drone and two obstacles shown in both Gazebo and RViZ.

be easily represented in a local point cloud map of the area as shown in Figures (4.7a) and (4.7b). Our configured LIDAR sensor is presented on the following specsheet.

LIDAR Module Specifications

- **Update Rate** : 30 measurement updates per second
- **Operating Distance Range** : 0.3 meters to 30.0 meters
- **Operating Angle Range** : $[-135, +135]$ degrees, temporarily increased range to $[-180, +180]$ during training for increased data
- **Ray Count (Resolution)** : 1081 points, temporarily decreased to 180 points during training for faster processing

4.1.4 UAV Model Overview

Two UAV models are presented. The first drone will be solely used for training purposes, while the second drone can provide the final testing results after the training is complete.

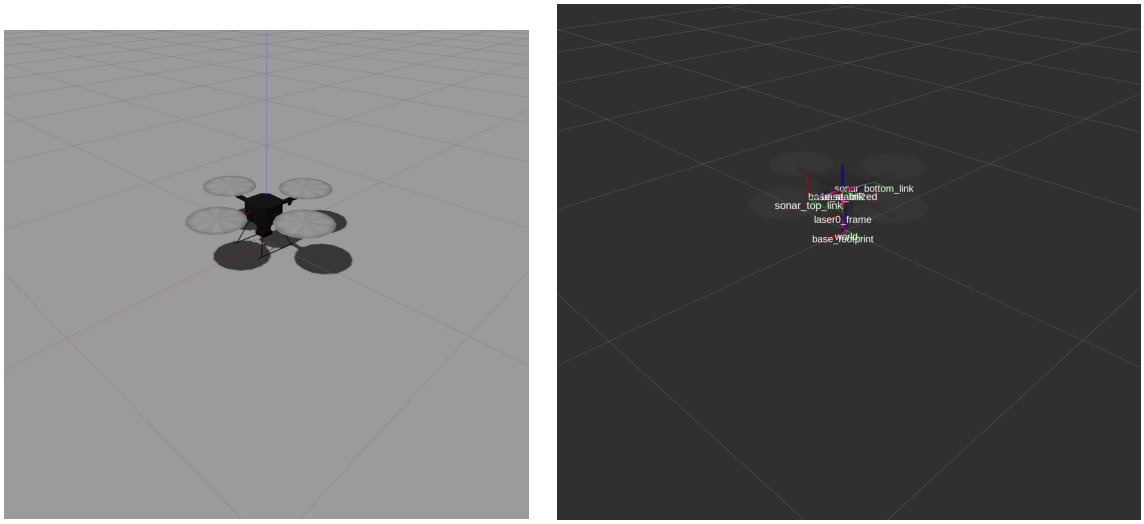
4.1.4.1 Hector Quadrotor as Training Drone

hector_quadrotor [1] is a category of general-purpose ROS packages related to modeling, control and simulations of UAV quadcopter systems. For this thesis, a modified version of the *quadrotor_hokuyo_utm30lx* was used. This UAV model consists of:

- 2D laser rangefinder (LIDAR)
- Top facing ultrasonic sensor
- Bottom facing ultrasonic sensor
- IMU
- GNSS receiver

Figure (4.8) shows the Hector Quadrotor in an empty environment from both the world's and the drone's perspective. Subsequent sections will explain why the training drone is "missing" a camera setup.

[1] `https://github.com/tu-darmstadt-ros-pkg/hector_quadrotor`



(a) Empty gazebo world with a hector quadrotor model **(b)** Empty gazebo world with a hector quadrotor model from RViZ point of view.

Figure 4.8: Empty world with hector quadrotor model shown in both Gazebo and RViZ.

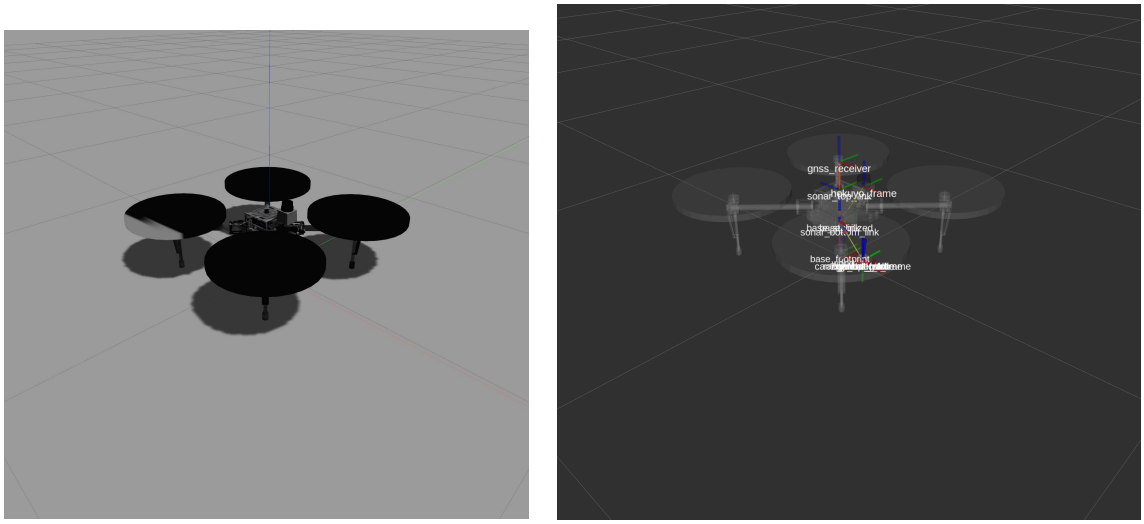
4.1.4.2 DJI Matrice 100 as Testing Drone

dji_m100_gazebo [2] is a ROS package for simulating the real Matrice 100 by DJI, a quadcopter by DJI designed for developers. This UAV model consists of:

- 2D laser rangefinder (LIDAR)
- Top facing ultrasonic sensor
- Bottom facing ultrasonic sensor
- 3D gimbal
- Optical camera mounted on 3D gimbal
- Single-point laser rangefinder mounted on 3D gimbal
- IMU
- GNSS receiver

Figure (4.9) shows the DJI Matrice 100 drone model in an empty environment from both the world's and the drone's perspective.

[2] `https://github.com/dji-m100-ros/dji_m100_gazebo`



(a) Empty gazebo world with a dji matrice 100 model

(b) Empty gazebo world with a dji matrice 100 model from RViZ point of view.

Figure 4.9: Empty world with dji matrice 100 model shown in both Gazebo and RViZ.

4.2 Environment Setup

In this section, we explain the concept of our dynamically generated environment, define the state and action space of our UAV and ensure compatibility with the OpenAI Gym framework by following its guidelines and maintaining its internal structure.

4.2.1 World Generation

The training process of a deep reinforcement learning algorithm requires many iterations in order to converge. To avoid model overfitting, we need a robust enough environment, which offers a variety of situations for our agent to experience, without being overly complicated and thus affecting training speed. Therefore, a highly-configurable custom world generation system was created to comply with our requirements. As mentioned in the problem statement, the environment consists of an indoor 3D space, containing obstacles and ArUco markers.

Firstly, we define our 3D indoor space as a *training_box*, a $(10 \times 10 \times 7.5)\, m^3$ semi-transparent hollow cube, in which all of our experiments will be conducted. An image of this box is presented in Figure (4.10).

Regarding the obstacles, a set of eleven (11) obstacle models were created with

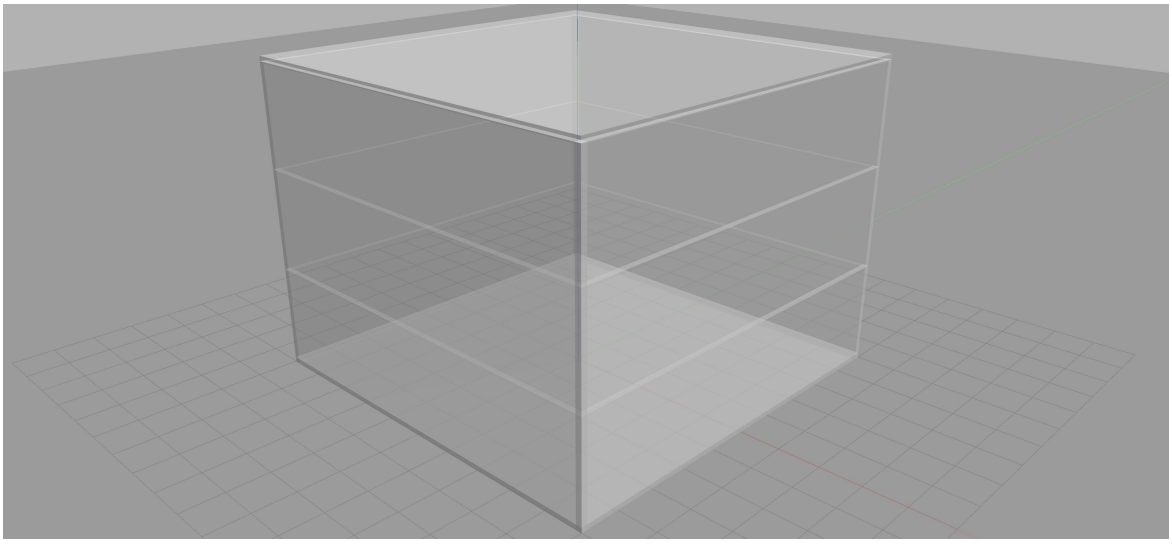


Figure 4.10: Gazebo world containing the *training_box*, a 3D space in which the RL agent will be trained.

Gazebo Model Name	Model Category	Size (meters)	Color
obstacle_cube_small	Cuboid	0.5×0.5	Green
obstacle_cube_medium	Cuboid	1×1	Yellow
obstacle_cube_large	Cuboid	2×2	Red
obstacle_tall	Cuboid	1×5	Cyan
obstacle_wide	Cuboid	3×2	Purple
obstacle_cylinder_small	Cylinder	0.5×0.5	Navy Blue
obstacle_cylinder_medium	Cylinder	1×1	Yellow
obstacle_cylinder_large	Cylinder	2×2	Orange
obstacle_sphere_small	Sphere	0.5×0.5	Salmon
obstacle_sphere_medium	Sphere	1×1	Pink
obstacle_sphere_large	Sphere	2×2	Blue

Table 4.1: Specifications of the obstacle family

different shapes and sizes to approximate real world scenarios. The obstacle family specifications are presented in Table (4.1). A Gazebo snapshot is also shown in Figure (4.11).

Lastly, a set of twelve (12) ArUco markers were also included, containing three (3) different IDs with four (4) sizes each. Table (4.2) presents the specs of the markers and Figure (4.12) shows the created models in the Gazebo simulator.

With this world generation system, custom specific configurations of markers and obstacles can be built with random or predefined poses (positions and orientations). Each time the world is launched, a random set of obstacles and markers is sampled

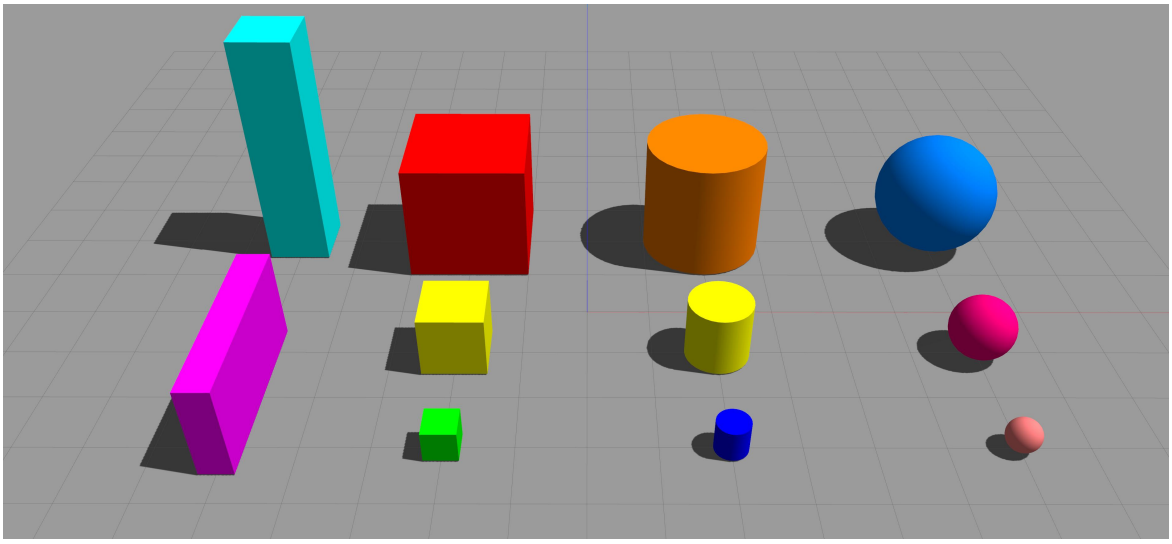


Figure 4.11: The obstacle family with different shapes and sizes for increased variety.

Gazebo Model Name	ArUco ID	Matrix Size (Blocks)	ArUco Scale (centimeters)
Aruco_Marker26_9cm	26	5×5	9×9
Aruco_Marker26_11cm	26	5×5	11×11
Aruco_Marker26_15cm	26	5×5	15×15
Aruco_Marker26_20cm	26	5×5	20×20
Aruco_Marker27_9cm	27	5×5	9×9
Aruco_Marker27_11cm	27	5×5	11×11
Aruco_Marker27_15cm	27	5×5	15×15
Aruco_Marker27_20cm	27	5×5	20×20
Aruco_Marker28_9cm	28	5×5	9×9
Aruco_Marker28_11cm	28	5×5	11×11
Aruco_Marker28_15cm	28	5×5	15×15
Aruco_Marker28_20cm	28	5×5	20×20

Table 4.2: Specifications of the ArUco marker family with 3 different IDs and 4 different sizes

from the corresponding family pool and spawned inside the training environment. Configurations can also be exported in JavaScript Object Notation (JSON) files for future utilization. This high level of adjustability allows the creation of worlds with ranging difficulties, defined by the number of markers and obstacles, as Figure (4.13) demonstrates.

This system also supports reinforcement learning setups with multiple training boxes and drone swarms for faster and parallel training, as shown in Figure 4.14.

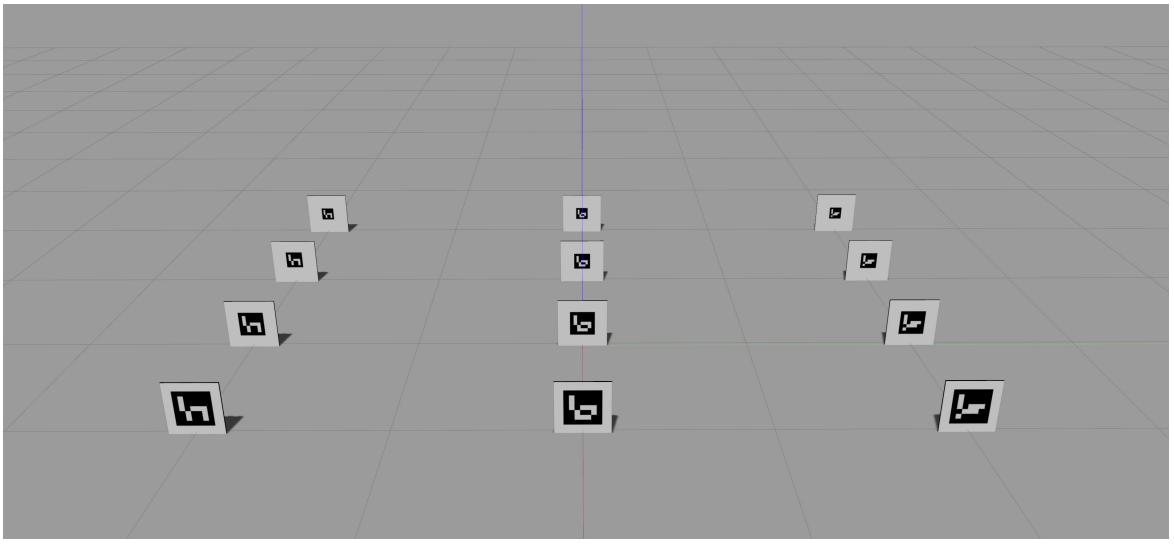
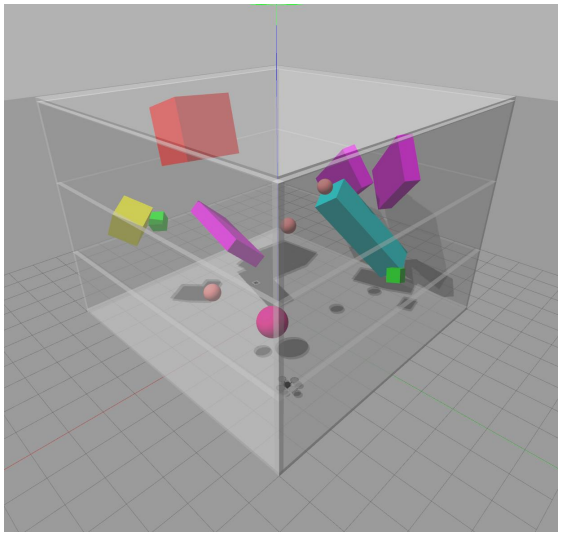
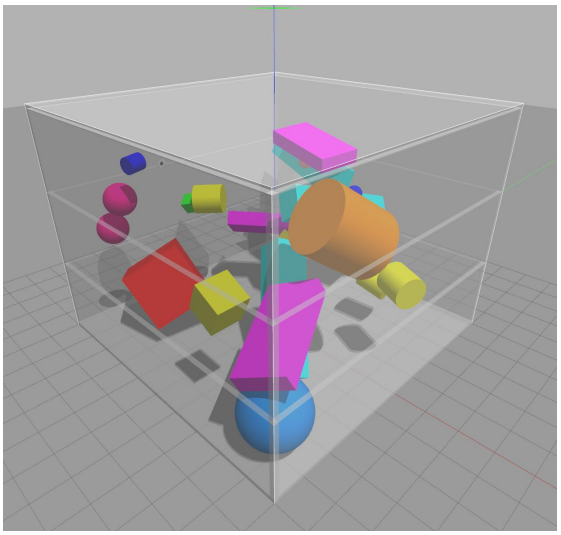


Figure 4.12: The ArUco Marker family with different IDs and sizes for increased variety. Each column of markers contains a specific ID while each row contains a specific marker size.



(a) Dynamically generated world with twelve (12) obstacles and one (1) marker



(b) Dynamically generated world with twenty-four (24) obstacles and three (3) markers

Figure 4.13: Two dynamically generated gazebo worlds with different parameters

This feature was not used for our training at the present time, since the DQN segment needed significantly more effort than anticipated, but it is definitely the first action point of future work.

4.2.2 Compatibility for OpenAI Gym

During development, compatibility was maintained with the OpenAI Gym framework. Gym environments require a specific code structure and strict definitions of state,

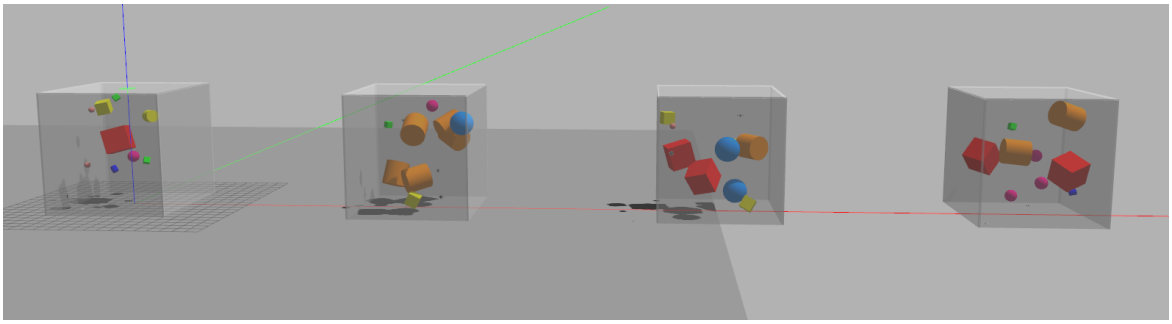


Figure 4.14: A gazebo world with multiple sets of training boxes.

action and reward spaces. Additionally, two main operations have to be implemented.

- **step** - The *step* function runs one time-step in the environment and collects the observation, action and reward of this specific time-step.
- **reset** - If the agent ever results in a fatal condition, the *reset* function is called to reset the environment back to its initial state.

4.3 Deep Reinforcement Learning Pipeline

This section provides in-depth analysis for every aspect of our deep reinforcement learning implementation. Initially, the sensor data extraction methods along with the UAV's action space and the designed reward system are explained. Eventually, focus will be shifted towards the neural network, analyzing its structure and selection of hyperparameters.

In Section (2.1.8.2) we introduced the DQN algorithm as an alternative promising approach to other reinforcement learning methods using deep neural networks. It also makes use of two powerful features, *Experience replay* and *Target Networks* to overcome several instability problems. A diagram form of DQN can be shown in Figure (4.15).

4.3.1 States and Observations

Section (2.1.3) established the *state space* and *action space* as part of the definition of a Markov Decision Process, denoting that each space can be either discrete or continuous. The state space of our environment is continuous, since the information we extract consists of decimal numbers. On every time-step (state), the agent observes

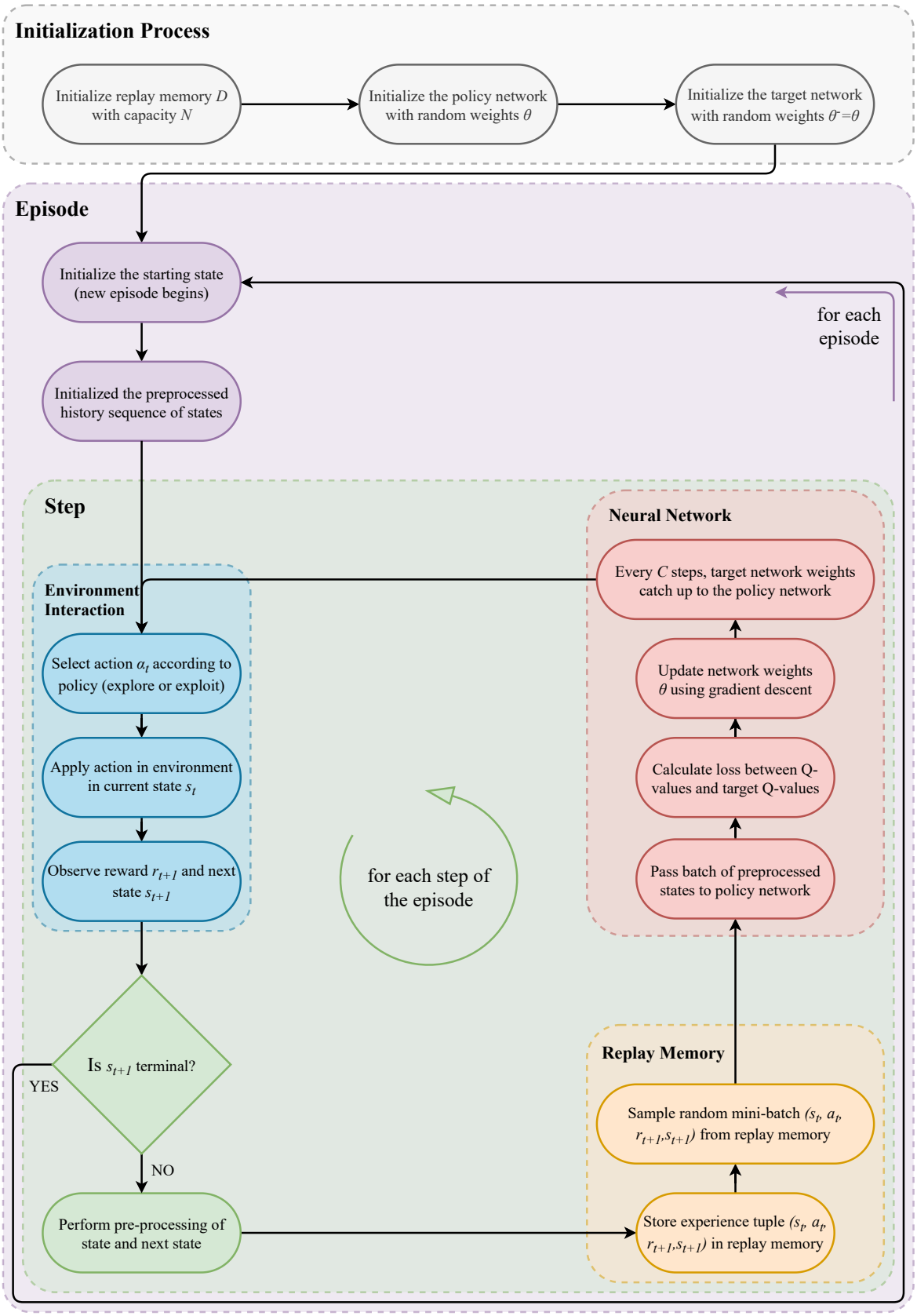


Figure 4.15: Diagram representation of the DQN algorithm

the environment and selects an action in order to receive a reward. An *observation* is a collection of measurements made by the robot's sensors, which describes a specific state. Real-time sensor information can be transmitted using ROS messages and topics.

In our case, there are two types of observations required to solve our problem:

- **Target Information** - to find and approach the target
- **Surroundings Information** - to avoid collisions with obstacles and walls (situational awareness).

4.3.1.1 Target Information

Target information can be achieved in two ways. Surely, we can use the optical camera, scan the image plane for any ArUco markers, using the OpenCV detection algorithm. This algorithm then estimates the marker's *pose*, that is, position and rotation with respect to the drone's camera. Lastly, using transformations, we can obtain the relative pose between the UAV and the marker.

Alternatively, and this is where things get interesting, we can skip the optical camera and gimbal setup altogether and just obtain the marker's known position and rotation right from the simulator itself. One could argue that the latter method is "cheating", because in the real world we do not have access to the real marker pose information. Strangely, it does not matter, since from a reinforcement learning perspective, we are only interested in training our neural network. This greatly reduces processing time, thus increasing training speed. It is important to note, however, that this method can only be used during training. When the training process is complete, we can then use the optical camera to estimate the marker's position.

Another relevant matter to discuss, is that our mission is considered complete, when the UAV successfully approaches every ArUco marker in the environment. By "approaching", we imply reaching a *target point*, located 2 meters (arbitrarily selected) away and in front of the marker. This way, the final marker's pose estimation would be the most accurate.

The resulting array for our target information is:

$$T = \begin{bmatrix} x_{delta} & y_{delta} & z_{delta} & yaw_{delta} & yaw_{global} \end{bmatrix} \tag{4.1}$$

4.3.1.2 Surroundings Information

Both drones can achieve situational awareness by using a 2D LIDAR sensor and a pair of vertical opposite-facing SONAR sensors.

Extracting SONAR data is relatively easy and minimal processing is required. The LIDAR sensor on the other hand, produces 180 measurements each time and only six (6) values are chosen to represent our observation array. Therefore, a dimensionality reduction processing pipeline is required. Firstly, we split the 180 measurements into 6 zones (each zone is responsible for a 60° area). Then, each zone is represented by its minimum value, resulting in 6 values. The "minimum" operation is chosen to consider the worst possible case. The resulting array for our surroundings information is:

$$D = \begin{bmatrix} L_1 & L_2 & L_3 & L_4 & L_5 & L_6 & S_1 & S_2 \end{bmatrix} \tag{4.2}$$

where L_i are the LIDAR measurements and S_j are the SONAR measurements..

The complete observation array is produced by combining the target information (4.1) and the surroundings information (4.2) leading to a 13-dimensional state vector. Figure (4.16) graphically represents this entire procedure, while Figure (4.17) shows a snapshot of the training process, in which the UAV is trying to approach the target point (blue dot), in order to optimally estimate the marker's pose (2D white square). The drone is also aware of its surroundings, by analyzing the SONAR data (two opposite cones) and LIDAR data (red dots).

Terminal States

As the DQN diagram indicated in Figure (4.15), if a terminal state occurs, the environment is reset and a new episode begins.

In our implementation, there are two conditions, which can cause a terminal state:

- **Mission Fatal** - if any distance measurement is less than 0.4 meters, we presume a collision has occured:

$$Fatal = \exists D_i < 0.4m, D_i \in D \tag{4.3}$$

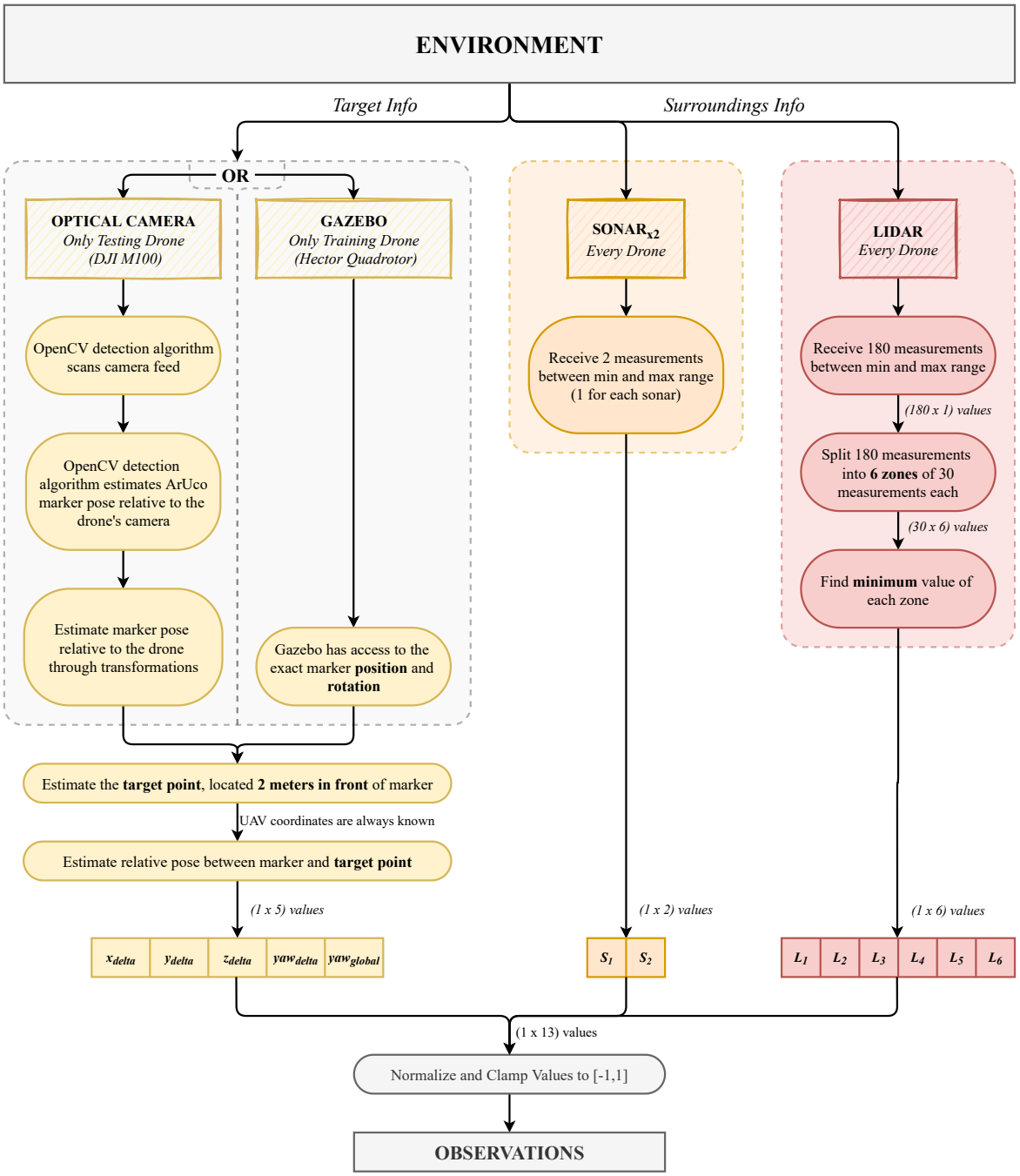


Figure 4.16: Diagram representation of the observation system

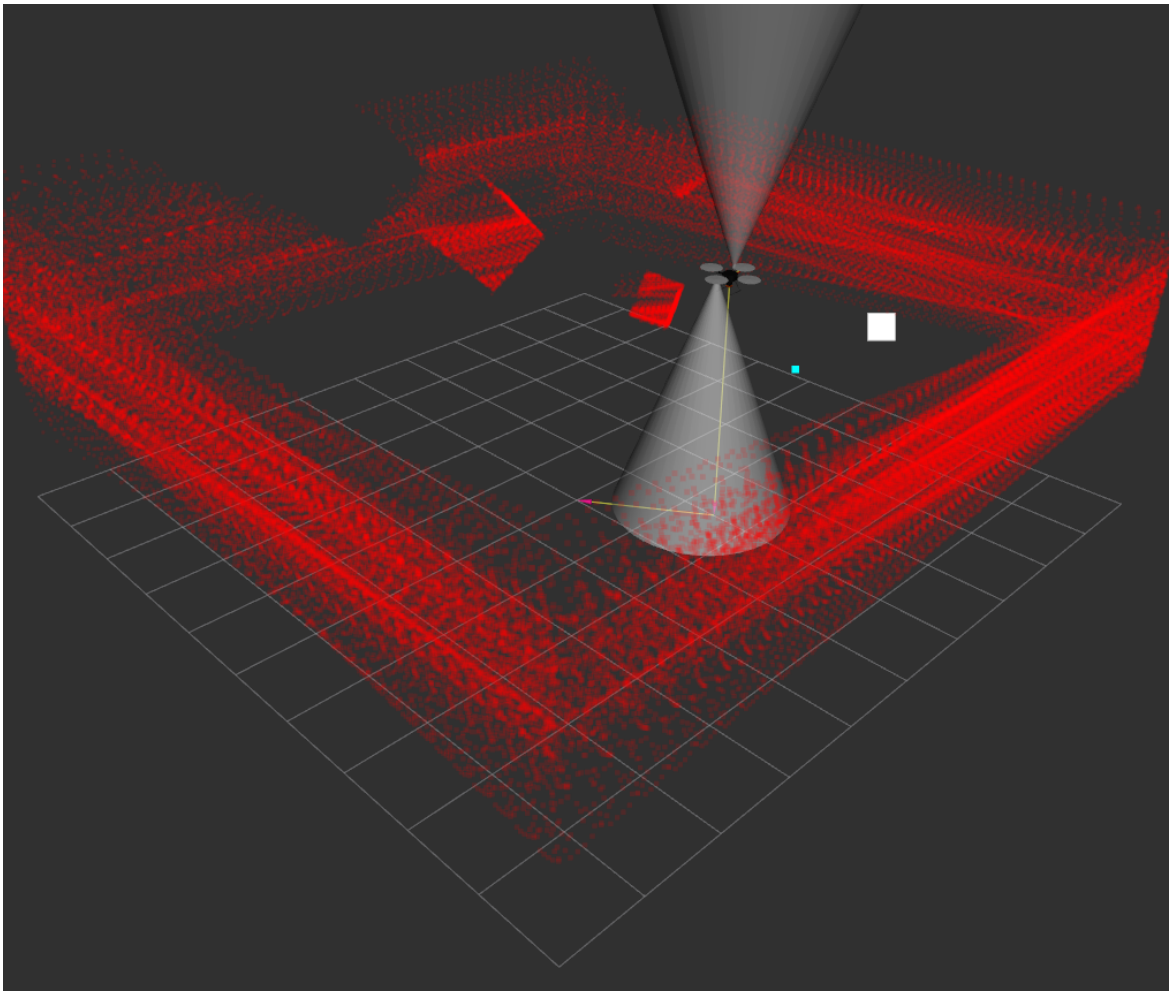


Figure 4.17: Snapshot of a training session. The UAV's goal is to reach the target point (blue dot) in order to estimate the marker's pose.

- **Mission Complete** - If the maximum distance of every dimension between the drone and the target point is less or equal than 0.8 meters, we presume a successful target point approach:

$$Complete = \max\left(x_{rel}, y_{rel}, z_{rel}\right) \leq 0.8m \tag{4.4}$$

A state is terminal, if the drone's mission is either fatal or complete:

$$Terminal = Fatal \vee Complete \tag{4.5}$$

It should be noted that a cap of 5.000 maximum steps for each episode was implemented in order to avoid endless episodes.

Action number	Action Title	Action Description
0	*increase pitch (positive pitch)*	moves forward
1	*decrease pitch (negative pitch)*	moves backward
2	*increase roll (positive roll)*	moves right
3	*decrease roll (negative roll)*	moves left
4	*increase throttle (positive throttle)*	moves higher
5	*decrease throttle (negative throttle)*	moves lower
6	*increase yaw (positive yaw)*	rotates clockwise
7	*decrease yaw (negative yaw)*	rotates anti-clockwise

Table 4.3: Simple representation of all actions available

Axis Name	Velocity increment	Min Value	Max Value	Total Stages
pitch	$\pm 1^{m}/_{s}$	$-2^{m}/_{s}$	$2^{m}/_{s}$	4
roll	$\pm 1^{m}/_{s}$	$-2^{m}/_{s}$	$2^{m}/_{s}$	4
throttle	$\pm 0.75^{m}/_{s}$	$-1.5^{m}/_{s}$	$1.5^{m}/_{s}$	4
yaw	$\pm {}^{\pi}/_{2}{}^{rad}/_{s}$	$-\pi^{rad}/_{s}$	$\pi^{rad}/_{s}$	4

Table 4.4: Specifications of each axis's incremental value and range

4.3.2 Actions

Actions define the behavior of a reinforcement learning agent. The action space of this environment is naturally continuous, since the UAV is constantly moving and acting in a 3D space. However, since DQN and other RL algorithms do not usually support continuous action spaces, we classified our actions into eight (8) discrete options, effectively controlling each vehicle axis' velocity. The actions are presented in Table (4.3).

In fact, a two-level velocity system was integrated. Each axis has a minimum and a maximum velocity. Every action results in an increase/decrease in velocity of a particular axis by a specific increment. There is a total of four (4) increment stages for each axis. This allows the drone to delicately fine-tune its velocity when it's near the target for increased stability. Axes specifications are presented in Table (4.4). A more advanced diagram is shown in Figure (4.18), explaining how the increments work.

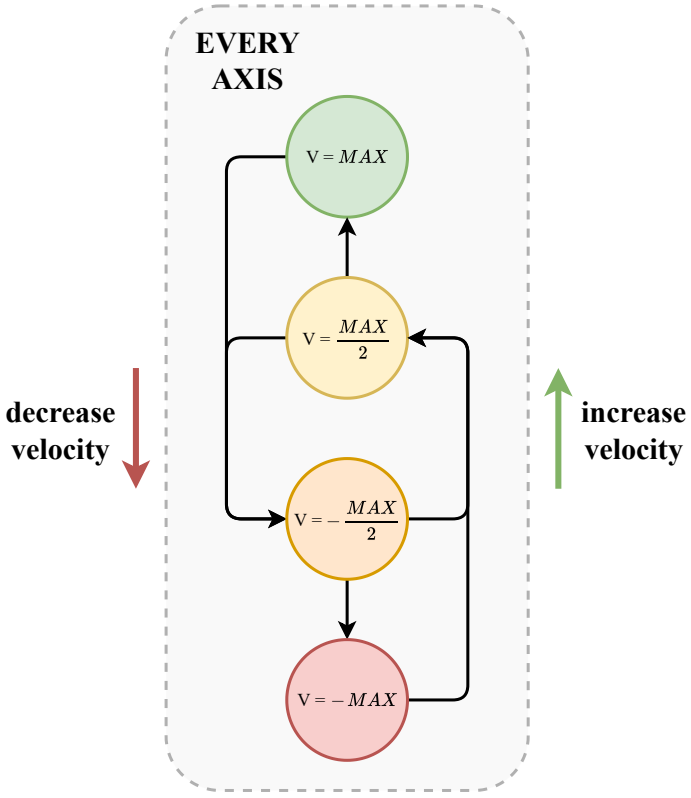


Figure 4.18: Diagram showing the different increment stages of each velocity axis.

4.3.3 Reward System

An effective reward system is crucial for training the agent and usually tricky to design. Even after finding the reward function concept, the fine-tuning required is similar to an optimization problem, which is solved by trial and error. Based on several guidelines indicated in Section (2.1.5), the main requirements were to build simple and continuous reward functions that offer dense rewards for the agent.

Many variations were investigated throughout our testing, including distance to target, looking direction and more. Our last iteration involves two (2) reward functions, which are analyzed below. It's important to remember that rewards are clipped to a range of $[-1, +1]$ to avoid instabilities in our neural network.

- **Wall Distance Reward** - This function penalizes the agent for being close to obstacles and walls. Although it is an exponential function, values are gradually

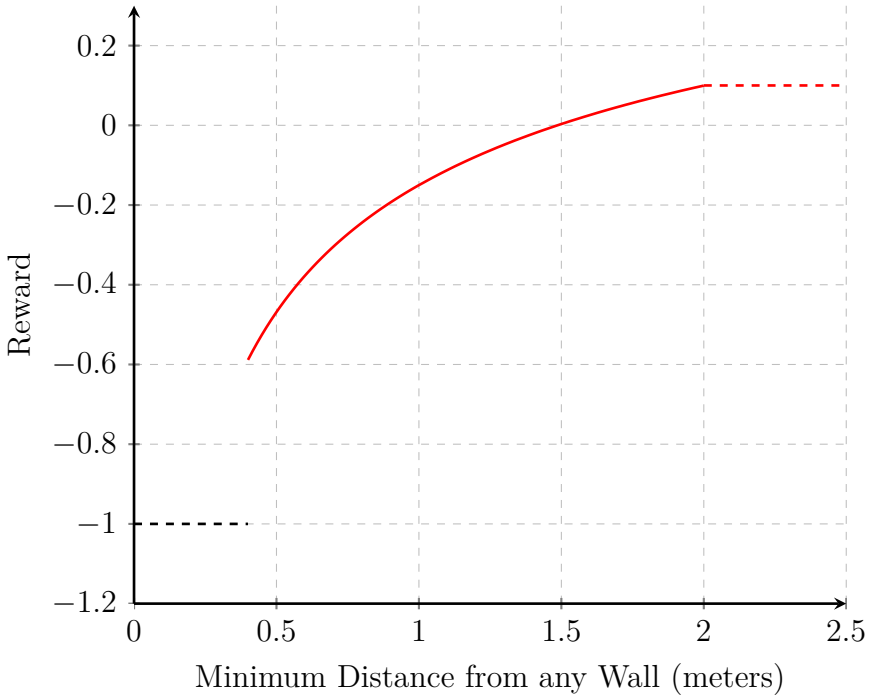


Figure 4.19: Plot showing how reward is affected given the minimum distance from a particular wall or obstacle. The reward becomes exponentially worse as the UAV comes closer to an obstacle.

distributed throughout its range.

$$WDR = \left\{ \begin{array}{ll} 0.1 & \text{for } x > 2 \\ -1.16x^{-0.35} + 1.01 & \text{for } 0.4 \leq x \leq 2 \\ -1 & \text{for } x \leq 0.4 \end{array} \right\} \tag{4.6}$$

where:

$$x = \min(D) \tag{4.7}$$

and:

$$D = \begin{bmatrix} L_1 & L_2 & L_3 & L_4 & L_5 & L_6 & S_1 & S_2 \end{bmatrix} \tag{4.8}$$

The wall distance reward is clipped to a range of $[-0.6, 0.1]$.

- **Velocity Direction Reward** - This rewards the agent for heading towards the target, by subtracting the current goal distance from the previous goal distance over two (2) consecutive time-steps.

$$VDR = 50\left(d_{delta,prev} - d_{delta,curr}\right) \tag{4.9}$$

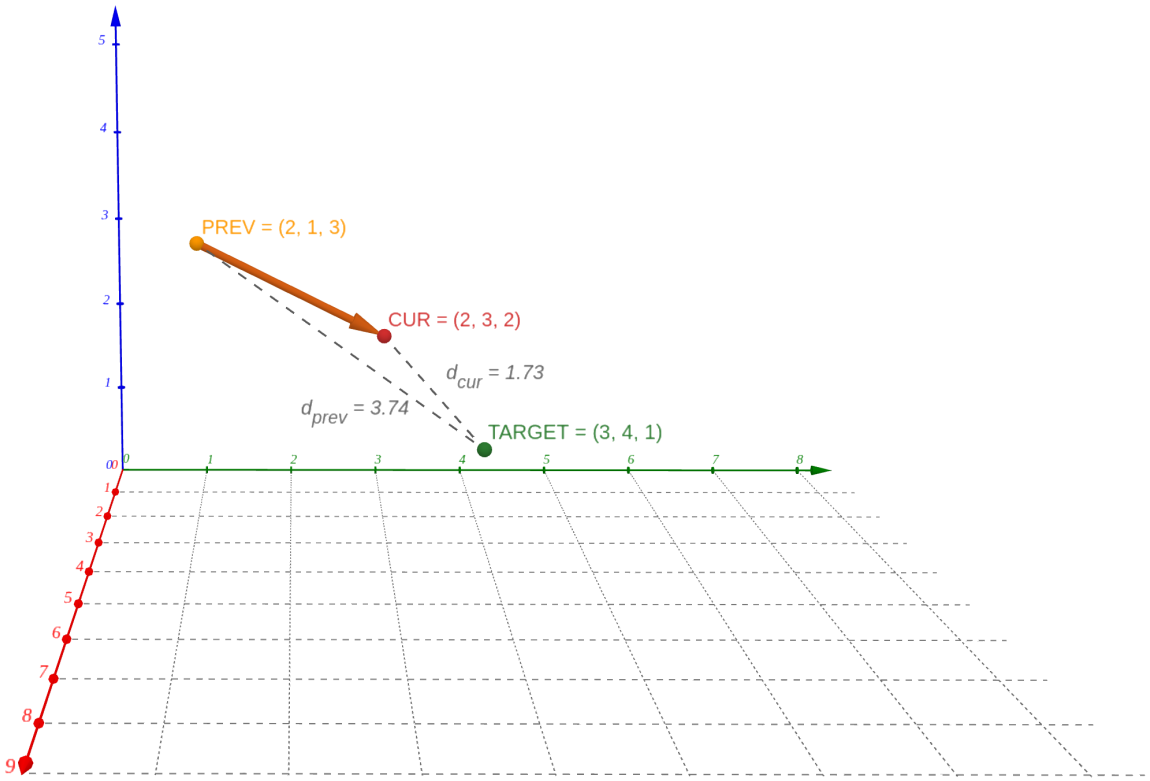


Figure 4.20: Plot showing the general concept of approaching a target by subtracting the current distance from the previous distance.

where:

$$d_{delta,prev} = \sqrt{x^2_{delta,prev} + y^2_{delta,prev} + z^2_{delta,prev}} \tag{4.10}$$

$$d_{delta,curr} = \sqrt{x^2_{delta,curr} + y^2_{delta,curr} + z^2_{delta,curr}} \tag{4.11}$$

The coefficient of 50 was added to balance the reward values to a clipped range of $[-1.2, 1.2]$, since the value of the distance traveled over 2 time-steps is relatively small. Figure (4.20) shows the concept of approaching the target.

The total reward is produced by just summing up both reward functions (4.6) and (4.9):

$$R_t = WDR + VDR \tag{4.12}$$

The worst possible reward value for our agent is $R_{min} = -0.6 - 1.2 = -1.8$ and it is earned when the UAV almost collides with an obstacle and heading towards the opposite side of the target. The best possible reward for our agent is $R_{max} = 0.1 + 1.2 = 1.3$,

achieved if the UAV is perfectly heading towards the target with no obstacles around. To achieve a total range of $[-1, 1]$, the total reward was clamped (clipped) to the smaller range without normalizing it, since normalizing would dramatically decrease the importance of both rewards just to accommodate the rarer cases $R_t \in [-1.8, -1) \cup (1, 1.2]$.

It should be noted that when the environment is reset, there is a lot of movement involved. In order to catch possible reshuffling errors, a safety mechanism was implemented, which ignores the first five (5) steps of each episode. These steps are defined as *NO_OP_STEPS*. The UAV takes no action, the reward is not taken into account, no samples are stored to or extracted from the replay memory and no training is performed.

Distance Interpretation

In machine and reinforcement learning, there are several well-known distance interpretation methods, two of the most popular being the *Euclidean Distance* and the *Manhattan Distance* (also known as the *taxicab geometry*).

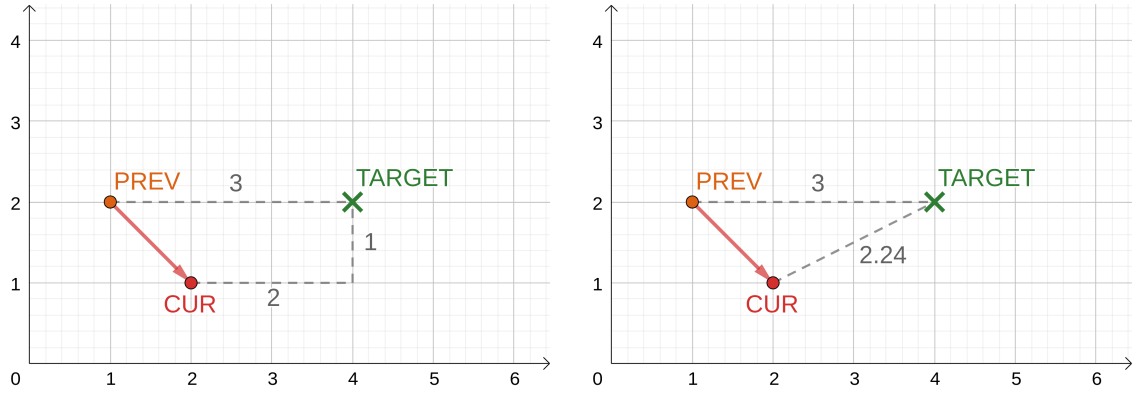
In Euclidean systems, distance is defined as the length of a line segment between two points.

$$d(x, y) = \sqrt{\sum_{i=1}^{N} (x_i - y_i)^2} \tag{4.13}$$

In Manhattan systems, it is defined as the sum of the absolute differences of each axis between two points.

$$d(x, y) = \sum_{i=1}^{N} |x_i - y_i| \tag{4.14}$$

The Euclidean distance was more suitable for our purpose and it was used to estimate the distance traveled in the VDR function (4.9), since as Figure (4.21) indicates, the Manhattan definition equates the previous-target and the current-target distances, when traveling diagonally.



(a) Manhattan distances of previous-target and current-target

(b) Euclidean distances of previous-target and current-target

Figure 4.21: Manhattan vs Euclidean distances between target and previous/current coordinates

4.3.4 History Preprocessing

Recall that during a DQN step, we sample a mini-batch from the replay memory. One last pending operation before proceeding to the neural network is the preparation of the network's input. For each experience in this batch, we create a stack of four (4) consecutive frames. This effectively gives the DQN algorithm a measure of velocity for moving objects. Since the experience tuple has access to the next state, consecutive frames can be easily found. Figure (4.22) shows the process of frame stacking for a single experience.

4.3.5 Deep Neural Network (DNN)

The original DQN paper [2, 3] used entire pre-processed RGB image frames as inputs. The most efficient method of classifying image information is by using convolutional layers, since the partial connections (convolution and other pooling layers) are used for feature extraction. Thus, DeepMind implemented a deep convolutional neural network, where each node contained raw image pixels.

In our case, our input data only consists of a small number of parameters. Specifically, each state observation contains 13 scalar values and we stack 4 consecutive frames, resulting in 52 total input parameters. Since our data is tabular and generally unrelated, the best option is to use fully connected layers, also referred to as *Dense* layers. Dense

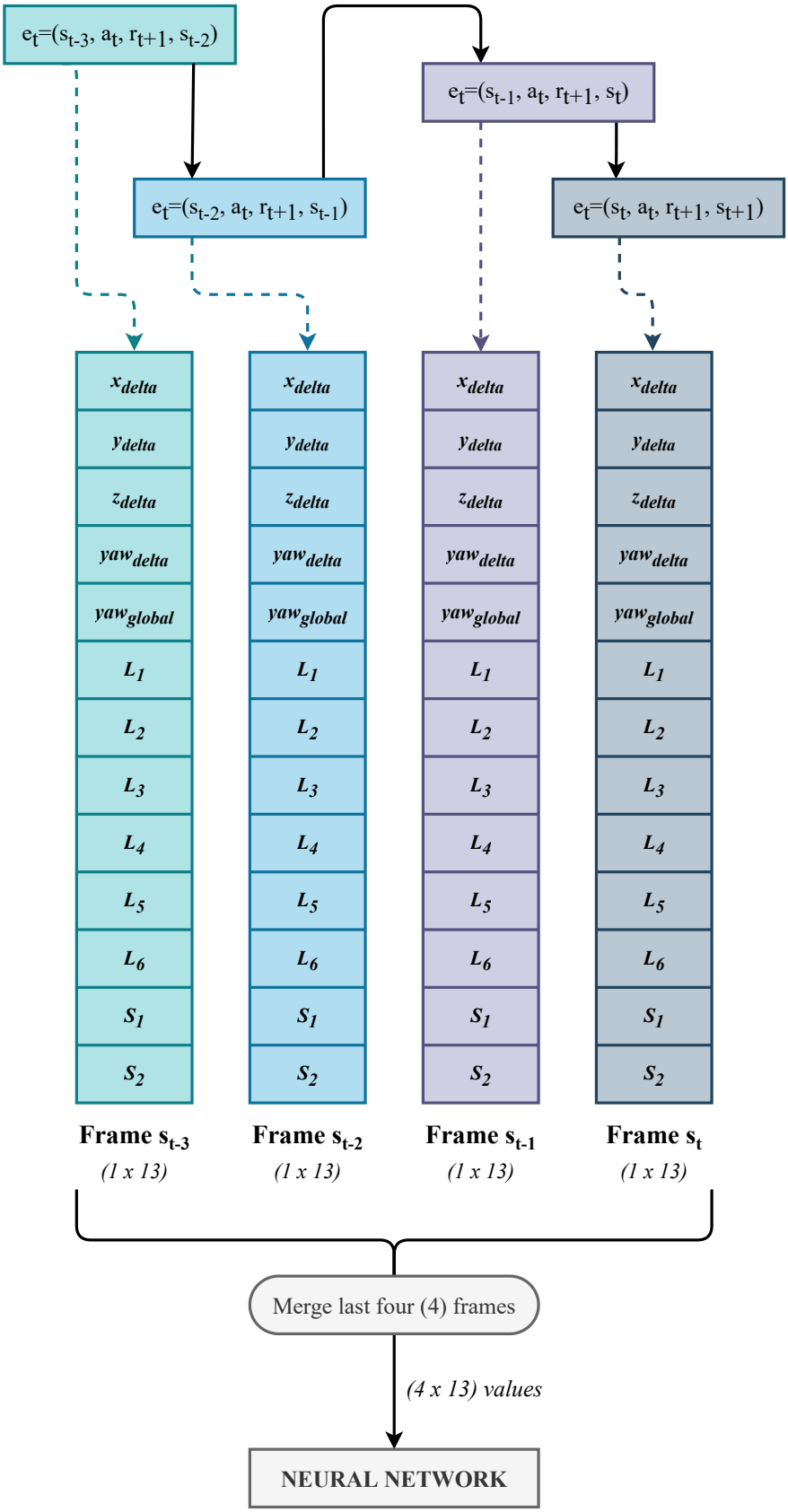


Figure 4.22: Diagram showing the process of frame stacking before training.

Layer Name	Layer Type	Input Shape	Output Shape	Activation Function	Initializer	Params
input_1	InputLayer	(-, 4, 13)	(-, 4, 13)	-	-	0
flatten	Flatten	(-, 4, 13)	(-, 52)	-	-	0
dense	Dense	(-, 52)	(-, 256)	relu	he_uniform	13.568
dense_1	Dense	(-, 256)	(-, 512)	relu	he_uniform	131.584
dense_2	Dense	(-, 512)	(-, 128)	relu	he_uniform	64.664
dense_3	Dense	(-, 128)	(-, 8)	linear	he_uniform	1.032

Table 4.5: Table showing specifications of every layer of our neural network architecture. The dash (-) denotes the batch size used.

layers are used whenever there the input data has a lack of structure that can taken into advantage. Therefore, the implemented neural network architecture consists of only fully connected layers. Specifically, it contains 1 input layer, 4 hidden layers and 1 output layer. Every node in each layer has the same activation function. Hidden layers used the *ReLU* activation function, a linear function that outputs the input directly if positive, a zero value otherwise. ReLU has become the default activation function for a variety of neural networks and is known to achieve great performance. Regarding weight initialization, *he_uniform* was used, which draws samples from a uniform distribution within $\left[-\sqrt{6/n}, \sqrt{6/n}\right]$, where n is the number of input units.

The specifications of each layer are denoted in Table (4.5). It should be noted that the Flatten layer unpacks the network's shape into a single dimension. It does not affect the size of the batch.

A saving/loading system has also been implemented, allowing for resumable training sessions. Each save operation includes the policy and target network weights, every single current hyperparameter and optimizer state.

4.3.6 Hyperparameter Optimization

In machine learning we often use the term *hyperparameter* to denote a parameter whose value affects the learning performance. Hyperparameters are static, meaning they don't change during training. Finding the optimal tuple of hyperparameters which yield the best training results can be time-consuming. In our case, hyperparameter optimization was performed manually, following some generally known guidelines.

There are many types of hyperparameters, some of which are permanent, while others depend on training time, in order to accomodate several built training configurations used in Chapter (5). Table (4.6) presents every single existing parameter in this project.

4.3.7 Relevant Information

This subsection provides some additional related information.

4.3.7.1 ROS and Gazebo Interaction Cycle

This project makes heavy use of ROS messages and topics. Figure (4.23) shows the active nodes and topics used in a training session. *train* is the main node responsible for training the agent. The interaction cycle can be easily noticed between the agent and the environment. *gazebo* node has the role of the environment, feeding information into the drone's sensors by publishing messages to their topics (*sonar_height_bottom* for the bottom sonar, *sonar_height_top* for the top sonar, *scan* for LIDAR sensor, *ground_truth_state* for positional and rotational information). The train node, being subscribed to these topics, repeatedly reads incoming messages and process the received information.

As Figure (4.15) indicated, once every step, we select an action based on our policy and we immediately apply the action to the environment. Then naturally, an undefined amount of time passes until the next interaction, since a series of processing operations take place. During this time, the simulator's clock is paused using a developed pause-resume system. If the agent needs to interact with the environment, the simulator is briefly unpaused and then paused again. This allows for consistent time-steps without losing any sequential information. Pausing, unpausing, resetting and object spawning and manipulation operations are achieved through the use of ROS services.

4.3.7.2 Gazebo Performance

Gazebo, being a fully-fledged robotic simulator is a computationally demanding software. Since one of the thesis's targets is to train a reinforcement learning agent as fast and efficiently as possible, several considerations were made during this project's development.

Parameter Name	Parameter Category	Dependency	Value	Parameter Description
gamma (γ)	network	fixed	0.99	importance of future rewards
learning_rate (α)	network	fixed	0.00025	network reaction intensity to errors
batch_size	network	fixed	1024	size of sample to be trained
train_freq	network	training time dependent	-	frequency of network training
target_update_freq	network	training time dependent	-	frequency of target network update
epsilon_init	policy	fixed	1	initial value of random action probability
epsilon_min	policy	fixed	0.05	final value of random action probability
epsilon_decay_steps	policy	training time dependent	-	number of steps to reach final value
memory_size	replay memory	training time dependent	-	amount of experience tuple capacity
start_train_steps	replay memory	training time dependent	-	training-delay steps
total_steps	other	training time dependent	-	total number of training steps
max_episode_steps	other	fixed	5000	max number of steps in each episode before resetting
eval_max_episodes	other	fixed	10	max number of episodes when validating
eval_freq	other	training time dependent	-	frequency of evaluations

Table 4.6: Hyperparameters used in our training. The dash (-) denotes a value range instead of a fixed value, which will be presented along with the training configurations.

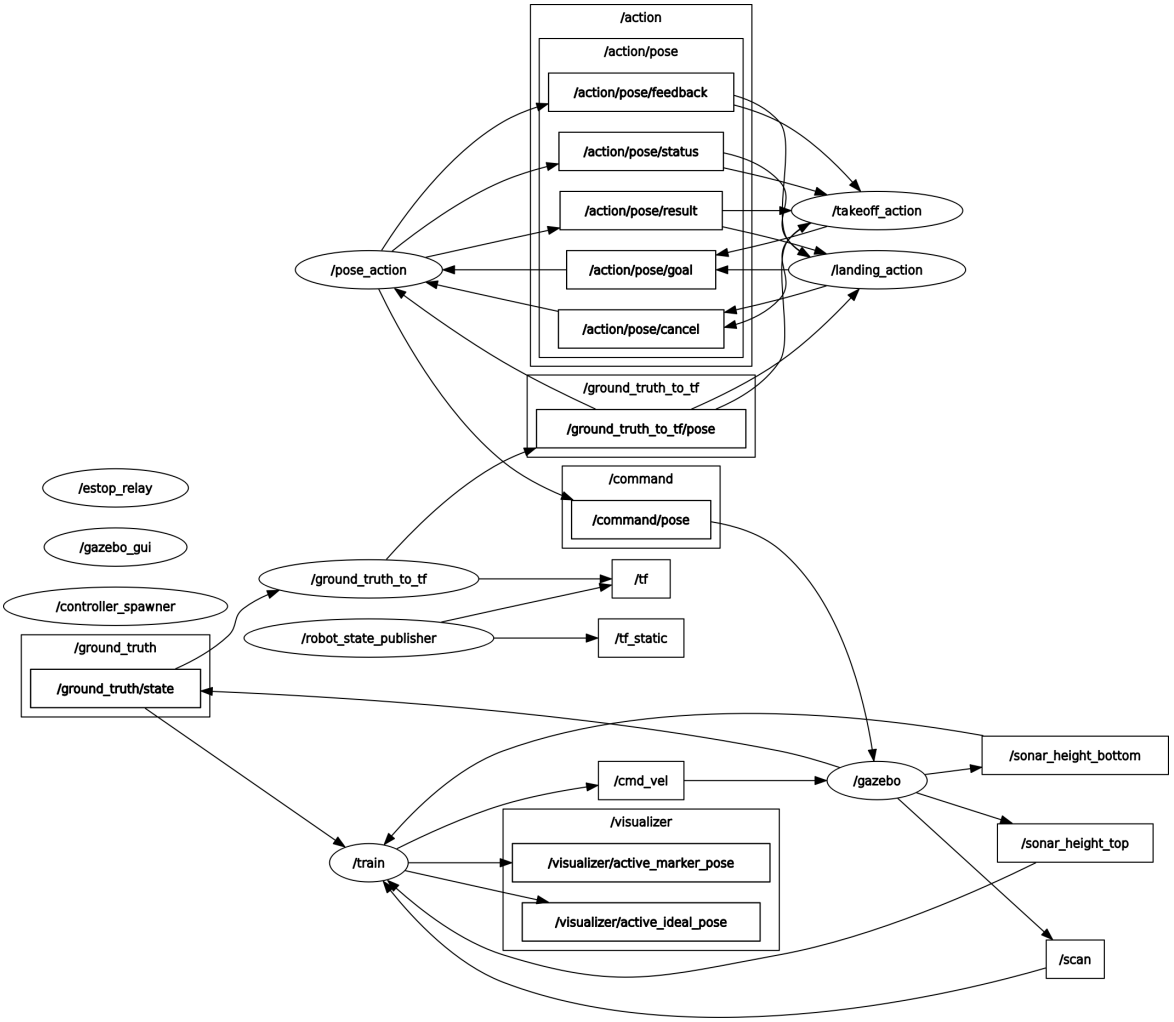


Figure 4.23: Diagram showing the nodes and topics used in a training sessions.

Firstly, the environment was designed to be as lightweight as possible, containing 3D obstacles with primitive shapes. Basic shapes allow for fast computation of physics and collision operations. Secondly, when the environment is reset, obstacles are simply rearranged, instead of being destroyed and recreated. This is a popular design pattern known as *object pooling*, which offloads computational resourcess off the CPU.

Regarding the neural network architecture, several optimizations were also made in order to speed up the training process. For starters, the network's input is relatively small, only consisting of 52 parameters. This is considered very small for a neural network, but results in faster training. While convolutional layers can be easily parallelized in GPUs, fully connected layers are more CPU-intensive.

The training was performed using a computer running an Intel Core i5-6600K (4 cores, 4 threads) paired with 16GB or RAM and accelerated by an NVIDIA Geforce GTX 1070. Every optimization made resulted in a Gazebo *Real Time Factor* of 14-20 while training, which denotes the amount of speed increase over the real time. Each RL training configuration session lasted an average of 32 hours. CPU usage was always at 100%, while GPU usage only peaked at 15-20%.

5
Experiments and Results

Contents

This chapter will present the results of the UAV's trained behavior. Initially, an examination will be made regarding the used experimental setup, including time and world difficulty configurations. Lastly, we compare the UAV's performance in various timing and difficulty circumstances.

5.1 Experimental Setup

In order to evaluate our agent's behavior, a concrete configuration structure needs to be prepared. The two following sections will focus on how training sessions can be parameterized with different timing and world difficulty settings respectively.

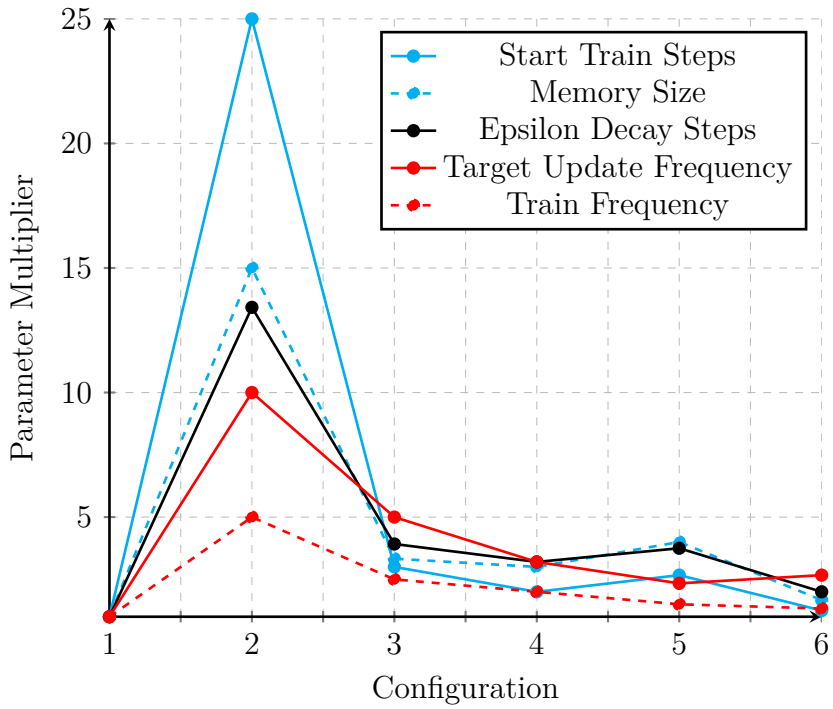


Figure 5.1: Plot showing parameters multiplier scale with respect to each configuration.

5.1.1 Training Time Configurations

For the scope of this thesis, six (6) training configurations were created based on training length, since linear decaying epsilon-greedy policies obligate a fixed pre-defined number of total training steps (before training starts). Several parameters are time-dependent and require adjustments in order to become suitable. For example, a training session with 1000 total steps, probably does not require a replay memory capacity of 1 million samples.

The six timing configurations are presented in Table (5.1), in which, previously introduced time-dependent parameters (TDPs) are now fully revealed. TDPs were empirically determined and were optimal considering a discrete number of options. Figure (5.1) shows the incremental change of each parameter over each longer training session.

It should be noted however that, not every time configuration is used. For final training purposes, the two largest configurations “Large” and “Marathon” were utilized, meanwhile quicker, simpler configurations were preferred for internal use and debugging. Nevertheless, having discrete timing options can help standardize benchmarking results.

Configuration Name	Demo (1)	Instant (2)	Quick (3)	Standard (4)	Large (5)	Marathon (6)
Total Steps	*5K*	*30K*	*125K*	*400K*	*1.5M*	*3M*
Total Duration	5m	25m	1h, 30m	5h	16h	32h
Time Multiplier	1	6	25	80	300	600
Epsilon Decay Steps	1.9K	25.5K	100K	320K	1.2M	2.4M
Memory Size	1K	15K	50K	150K	600K	1M
Start Train Steps	200	5K	15K	30K	80K	100K
Train Frequency	20	100	250	500	750	1K
Target Update Frequency	500	5K	25K	80K	187.5K	500K

Table 5.1: Training configurations based on length with corresponding parameters

5.1.2 World Difficulty Profiles

The ultimate goal of this project is to evaluate the agent's behavior over a number of scenarios of varying difficulty. For this reason, five (5) different world profiles of increasing difficulty were created depending to stillness of uav/marker initialization and obstacle quantity.

As Table (5.2) suggests, experimentation is initially performed using fixed, permanent spawn coordinates for the drone and markers. "Easy" and "Medium" configurations eventually replace each fixed aspect with randomness, while "Hard" and "Extreme" profiles include obstacles at random locations.

World Profile	UAV Initialization	Marker Initialization	Obstacle Initialization	Obstacle Number
Ridiculous	fixed	fixed	-	0
Easy	random	fixed	-	0
Medium	random	random	-	0
Hard	random	random	random	6
Extreme	random	random	random	12

Table 5.2: Specifications of world configurations with increasing difficulty level.

5.2 Training Results

This section provides and justifies the results of each training session. Specifically, we experiment on each world profile and observe how stillness and obstacle quantity affect the agent's performance.

Training in worlds with obstacles was performed using the "Marathon" train length, while worlds without obstacles used the "Large" configuration due to a lower world complexity.

In order to make multifaceted comparisons, as much data has been captured as possible during training. Each training session generates the following charts:

1. **[S] Total reward** : total accumulated reward during training (blue)
2. **[S] Loss** : loss occured during training (red)
3. **[S] Parameters** : epsilon (cyan) and memory filled (gray)
4. **[S] Average sample reward** : (green)
5. **[E] Episode outcome rates** : rate of collisions (CR) ($\frac{total\ collisions}{total\ episodes}$, red), rate of markers found (MFR) ($\frac{total\ markers\ found}{total\ episodes}$, cyan), rate of expiring episodes ($\frac{total\ expired\ episodes}{total\ episodes}$, orange)
6. **[E] Episode outcome rates of last 100 episodes** : rate of collisions (CR100) ($\frac{collisions,\ last\ 100}{100}$, red), rate of markers found (MFR100) ($\frac{markers\ found,\ last\ 100}{100}$, cyan), rate of expiring episodes ($\frac{expired\ episodes,\ last\ 100}{100}$, orange)
7. **[E] Relative marker approach to target (RMA)** : shows how close the drone approached the target before colliding ($\frac{distance\ from\ target,\ after}{distancefromtarget,before}$, blue)

8. **[E] Total markers found** : (brown)
9. **[E] Episode length** : shows how many steps are performed in each episode (black)
10. **[E] Mean max q-value** : mean max Q-values over each episode (purple)

Charts 1 and 2 are the most popular metrics in a reinforcement learning problem. Loss is used to provide an insight of the neural network's performance, denoting the deviation of its weights from the optimal weights. Total reward on the other hand, is supposed to evaluate the agent's behavior, but it is not always an accurate indicator. For example, if a reward system is designed to always provide the highest reward (e.g +1), the accumulated reward chart would show an incredible performance, but in reality, the agent would learn nothing.

Therefore, more practical measurements are required in order to effectively measure our training performance (5,6,7,8), such as marker detection rates and average approach distance from target.

Relative secondary information is also provided in charts (3,4,9), providing several details about the training sessions.

5.2.1 Ridiculous World, Large Training

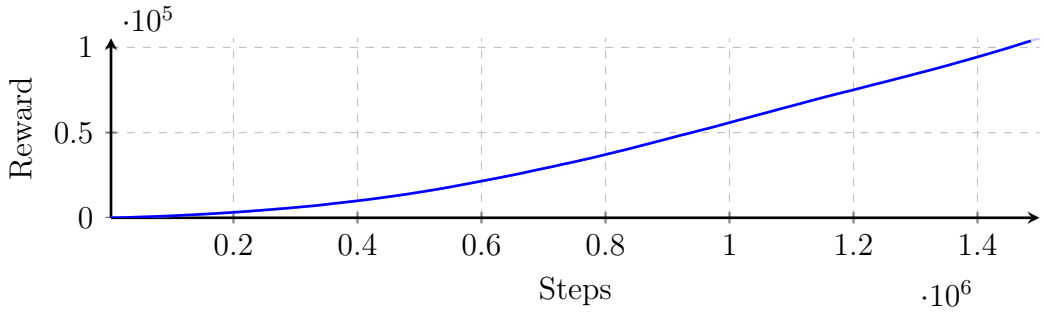


Figure 5.2: Total Reward

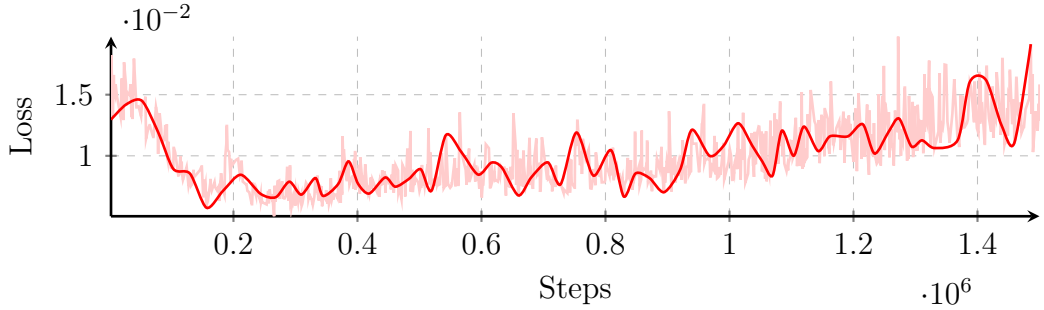


Figure 5.3: Network Weight Loss

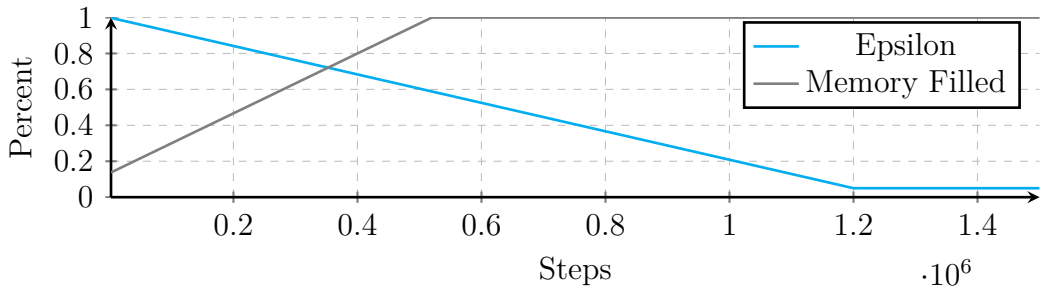


Figure 5.4: DQN Parameters (Epsilon and Memory Filled)

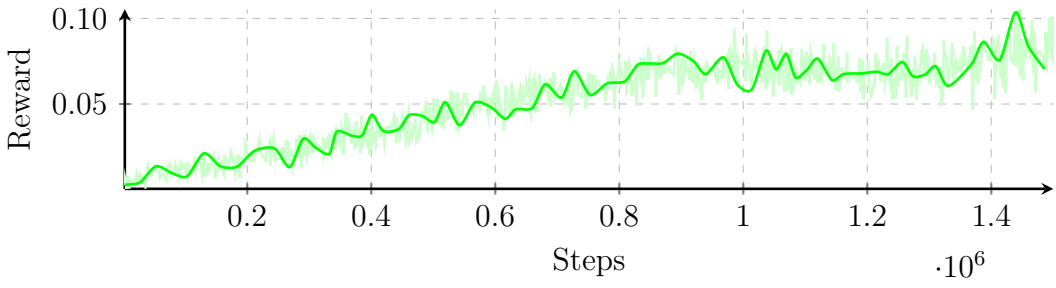


Figure 5.5: Average Sample Reward

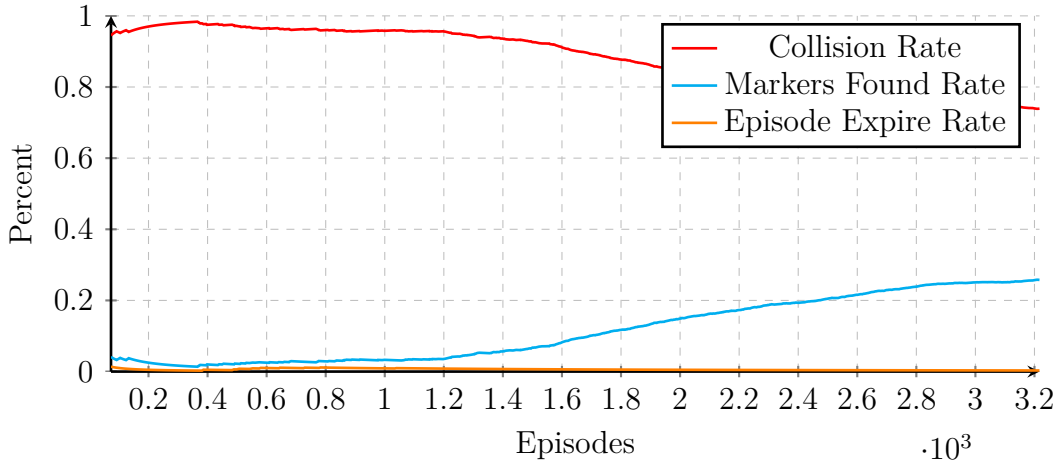


Figure 5.6: Episode outcome rates: collision rate, marker found rate, time expired rate

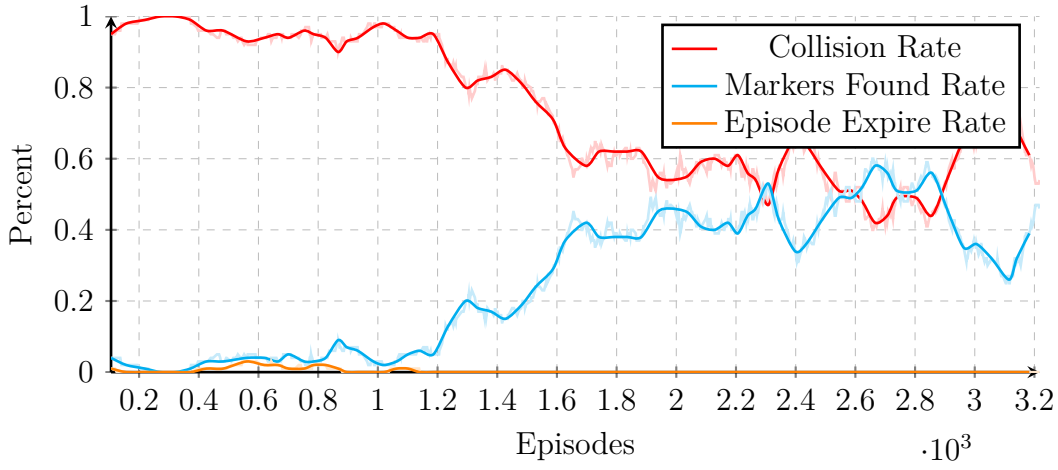


Figure 5.7: Episode outcome rates in last 100 episodes: collision rate, marker found rate, time expired rate

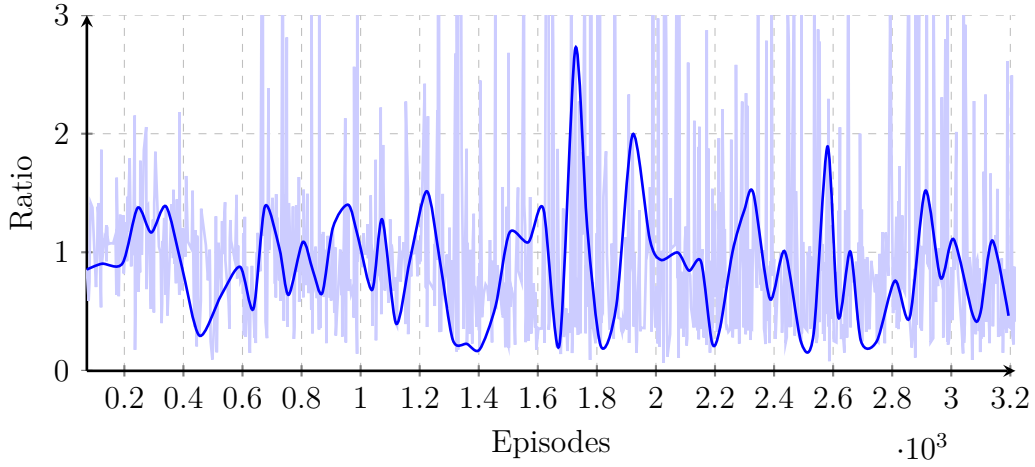


Figure 5.8: Relative Marker Approach from initial to termination point

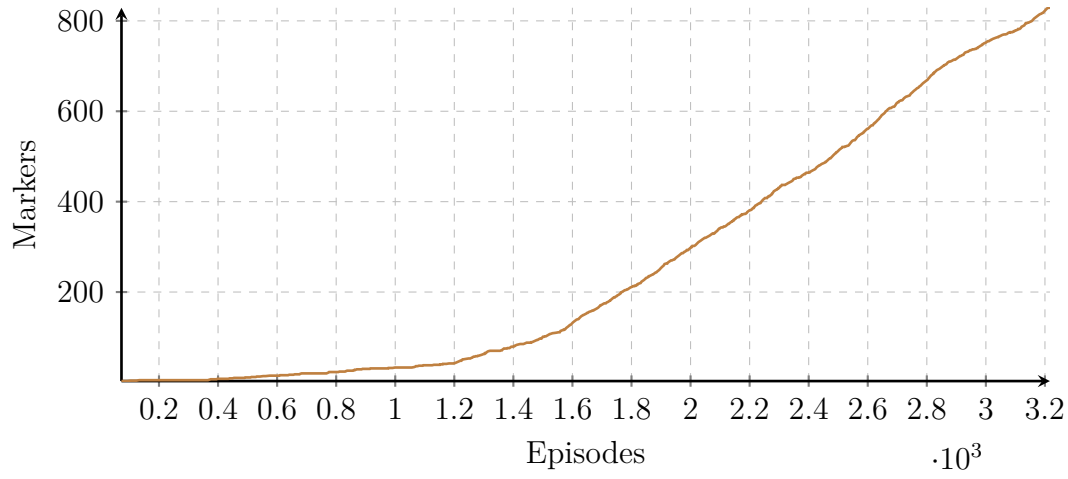


Figure 5.9: Total Markers Found

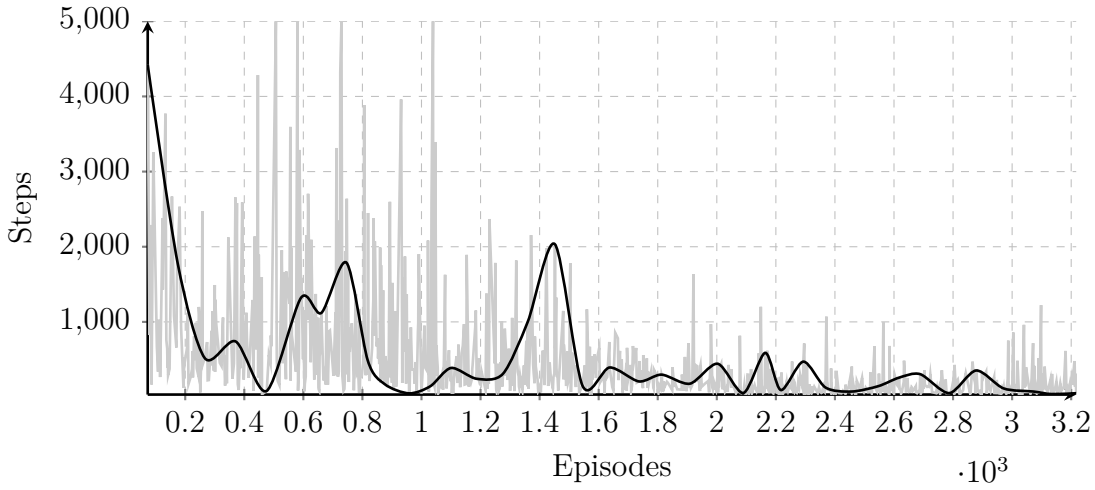


Figure 5.10: Episode Length

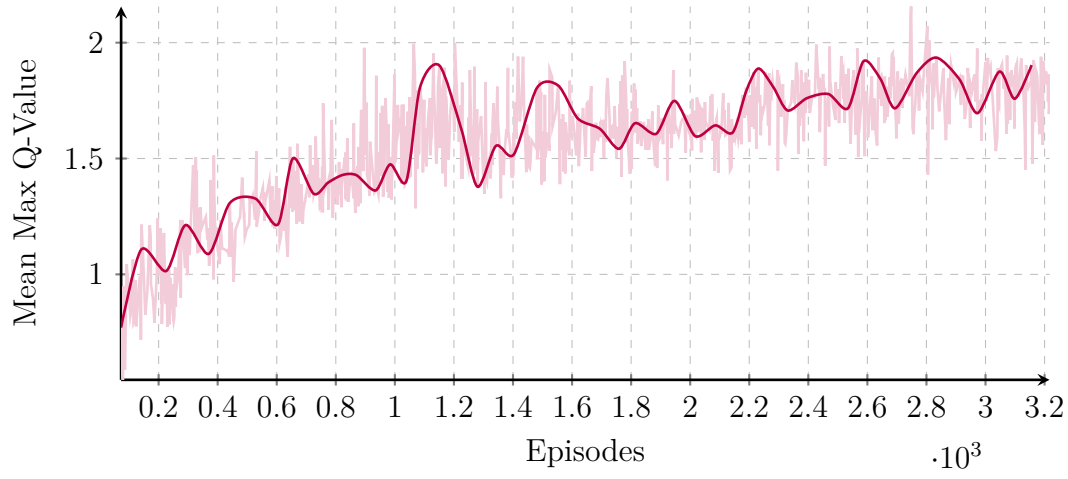


Figure 5.11: Mean Max Q-Value

5.2.2 Easy World, Large Training

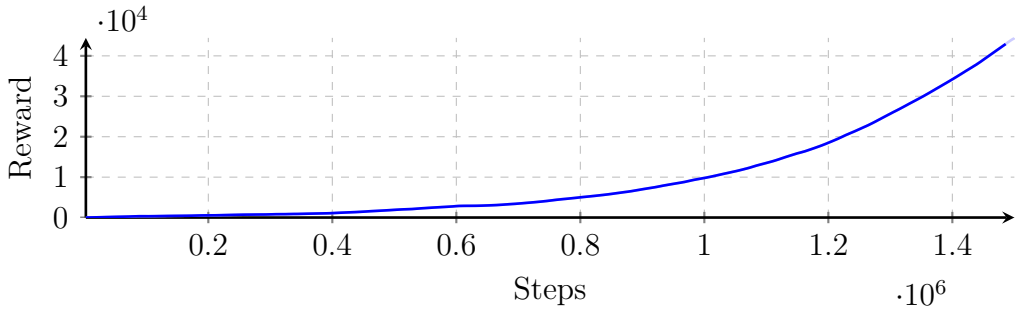


Figure 5.12: Total Reward

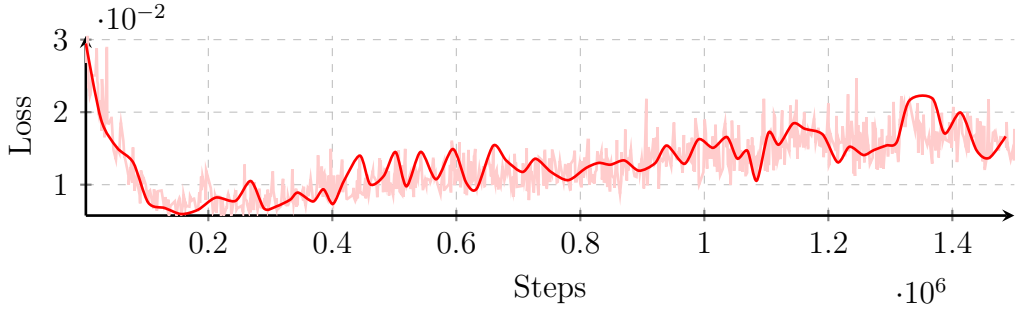


Figure 5.13: Network Weight Loss

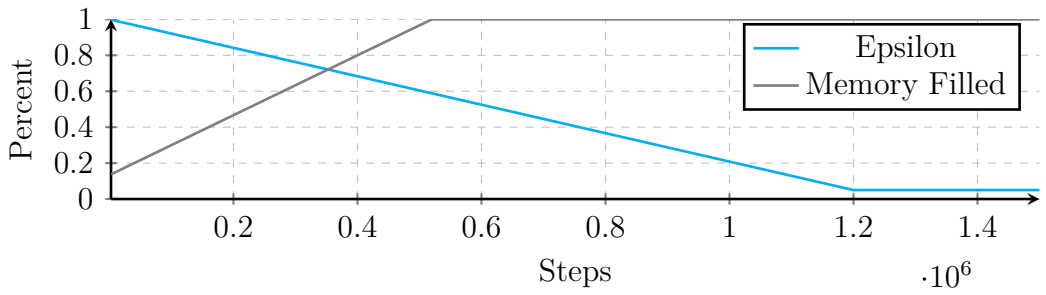


Figure 5.14: DQN Parameters (Epsilon and Memory Filled)

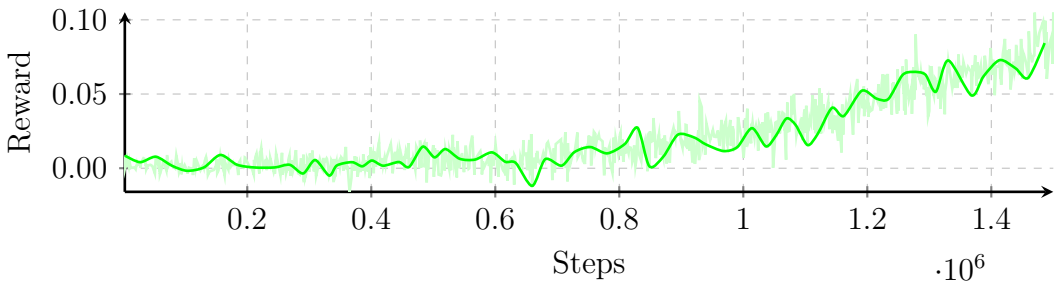


Figure 5.15: Average Sample Reward

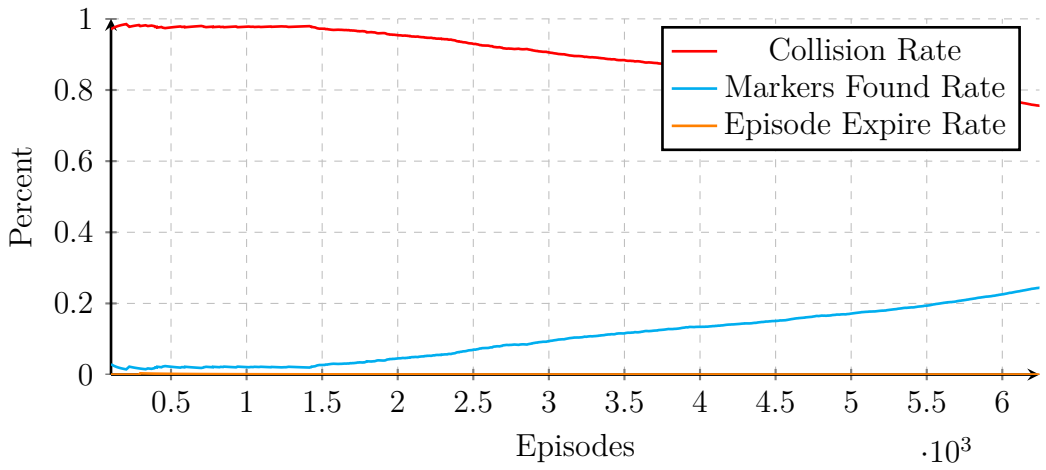


Figure 5.16: Episode outcome rates: collision rate, marker found rate, time expired rate

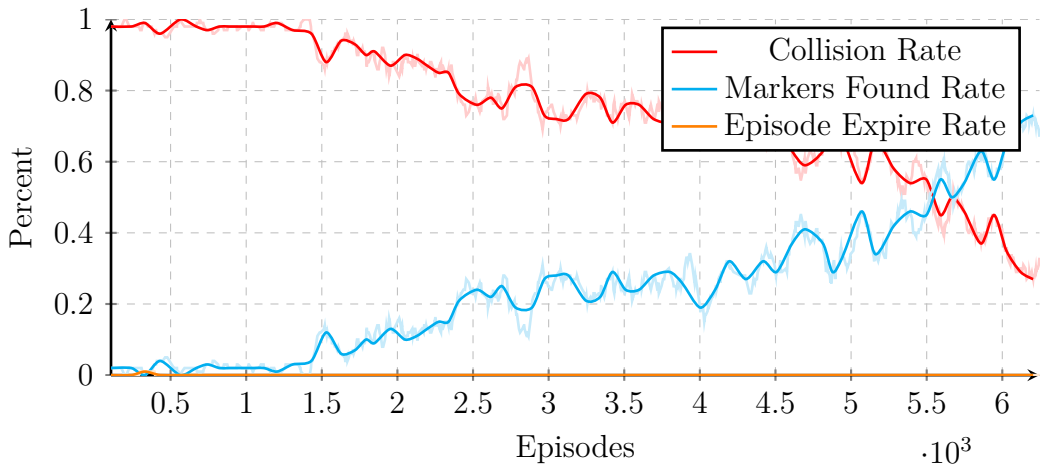


Figure 5.17: Episode outcome rates in last 100 episodes: collision rate, marker found rate, time expired rate

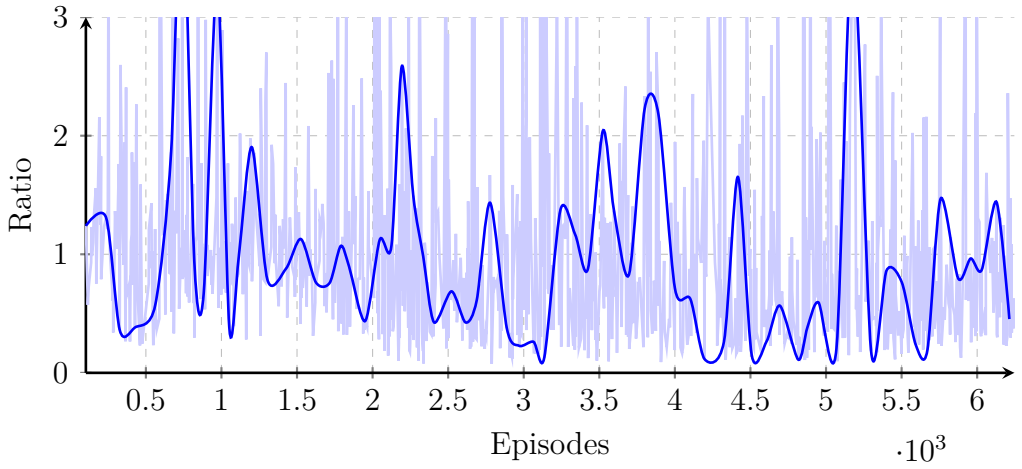


Figure 5.18: Relative Marker Approach from initial to termination point

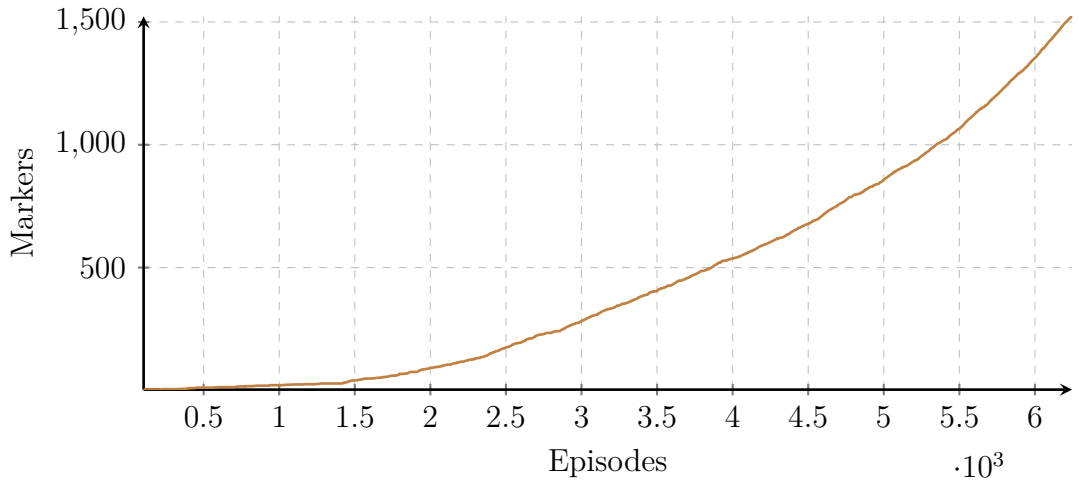


Figure 5.19: Total Markers Found

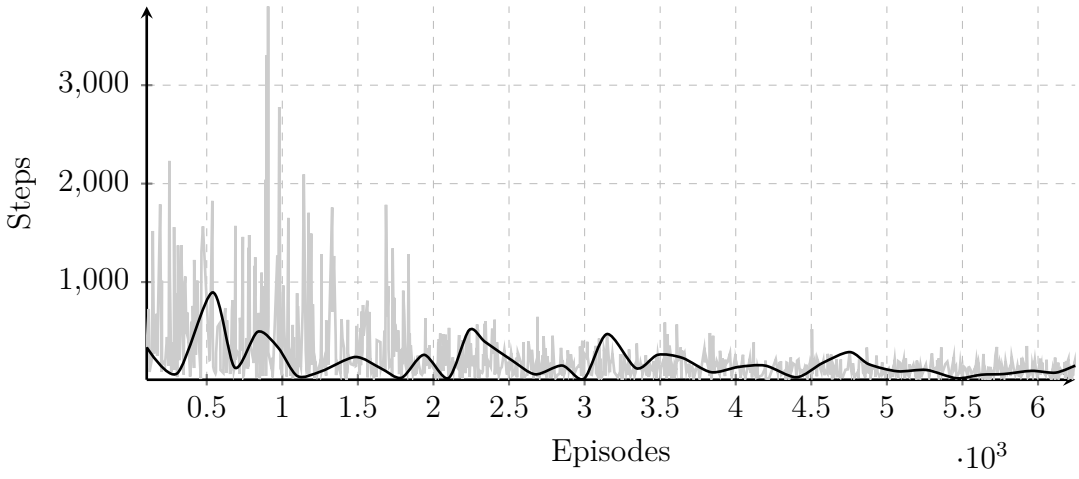


Figure 5.20: Episode Length

Figure 5.21: Mean Max Q-Value

5.2.3 Medium World, Large Training

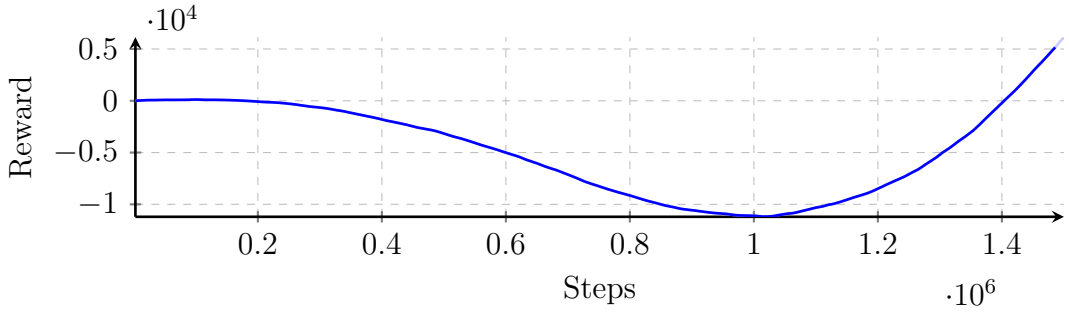


Figure 5.22: Total Reward

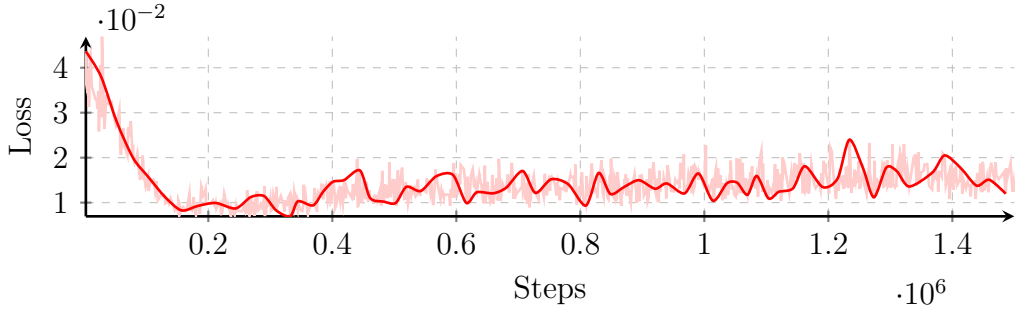


Figure 5.23: Network Weight Loss

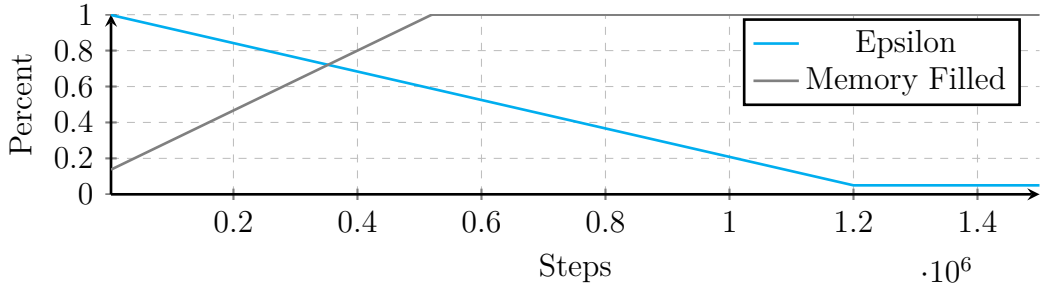


Figure 5.24: DQN Parameters (Epsilon and Memory Filled)

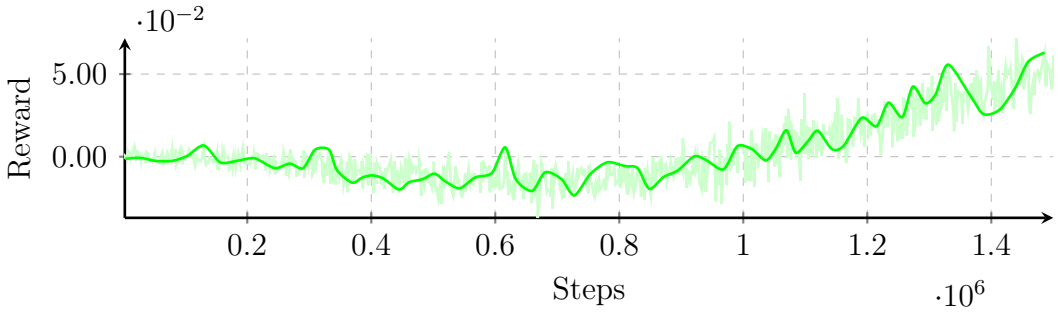


Figure 5.25: Average Sample Reward

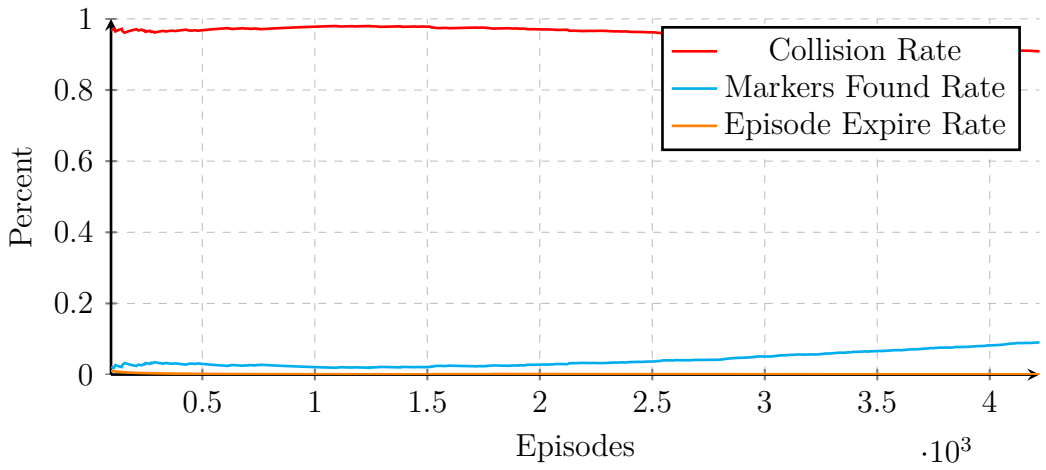


Figure 5.26: Episode outcome rates: collision rate, marker found rate, time expired rate

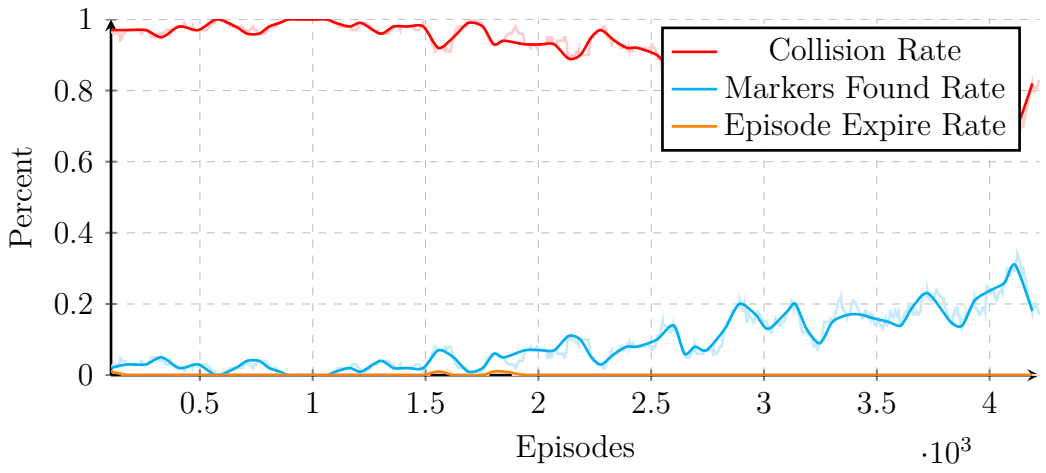


Figure 5.27: Episode outcome rates in last 100 episodes: collision rate, marker found rate, time expired rate

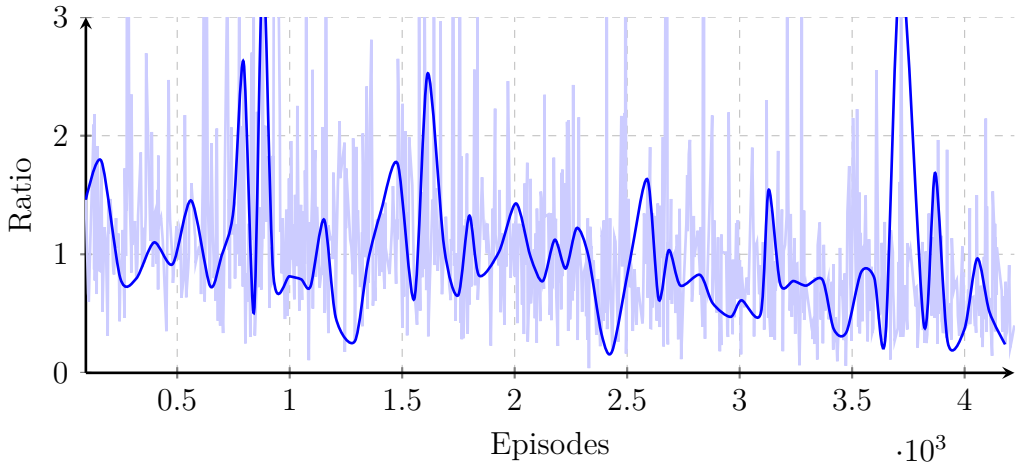


Figure 5.28: Relative Marker Approach from initial to termination point

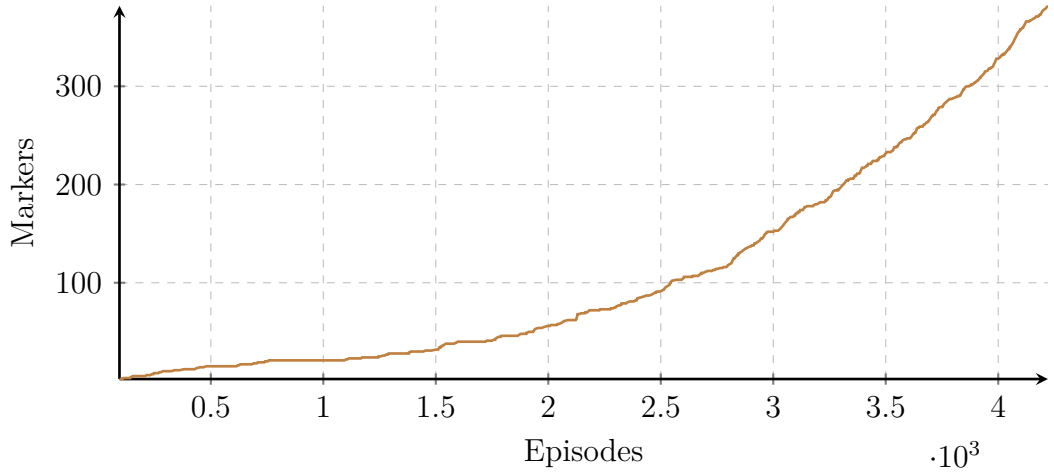


Figure 5.29: Total Markers Found

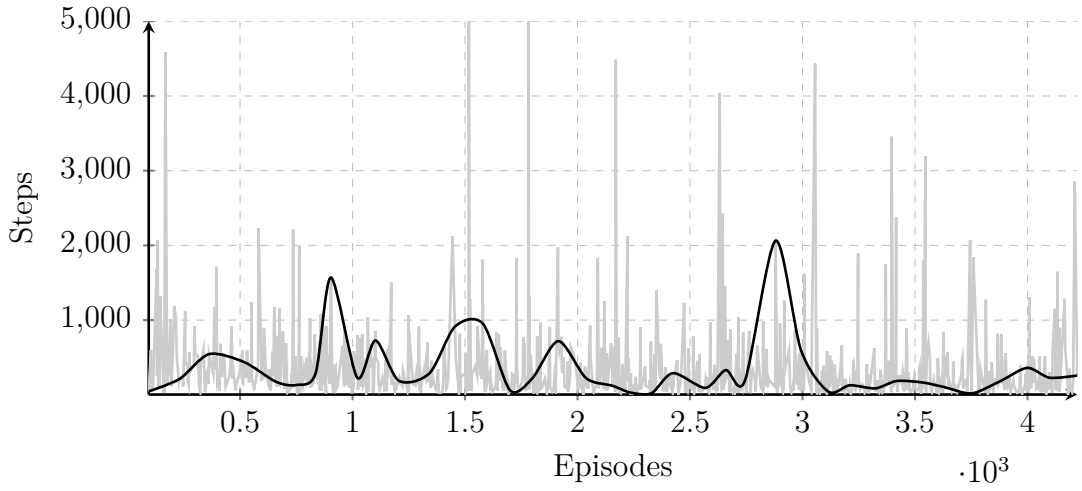


Figure 5.30: Episode Length

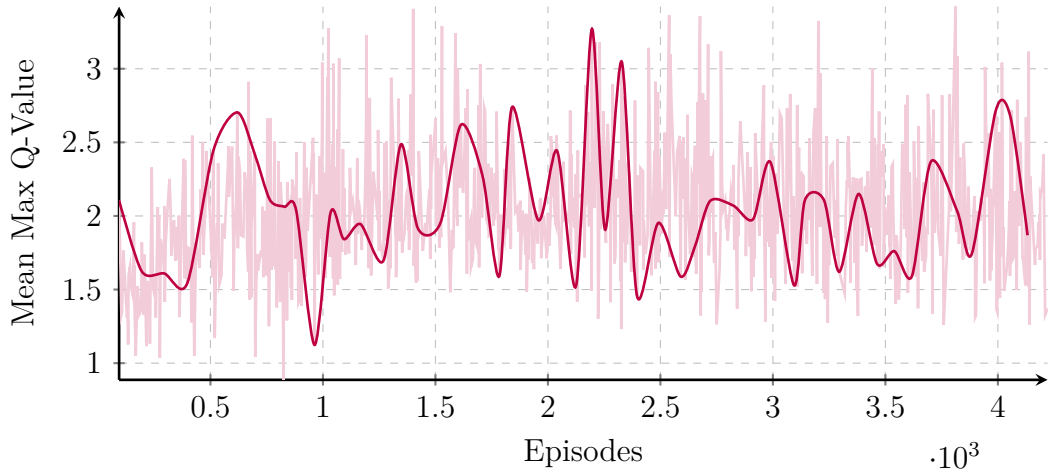


Figure 5.31: Mean Max Q-Value

5.2.4 Hard World, Marathon Training

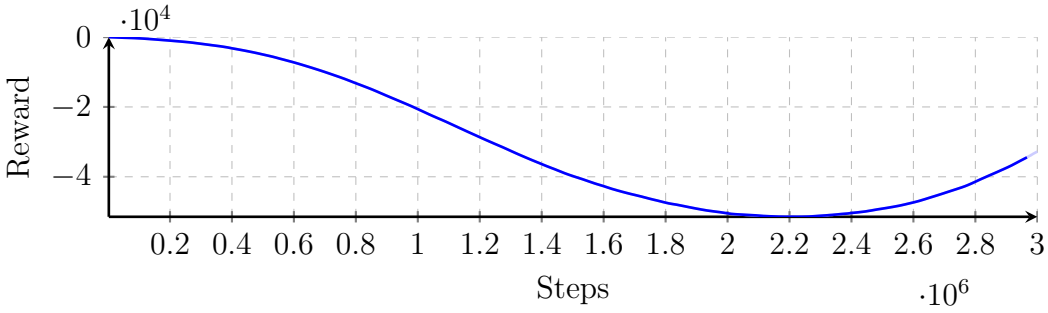


Figure 5.32: Total Reward

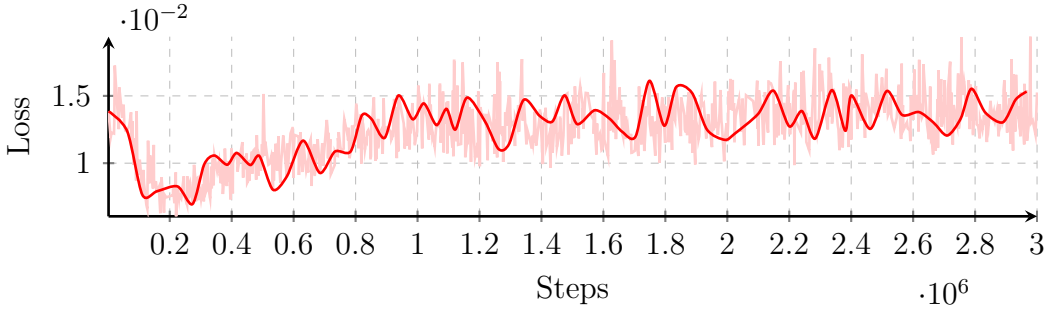


Figure 5.33: Network Weight Loss

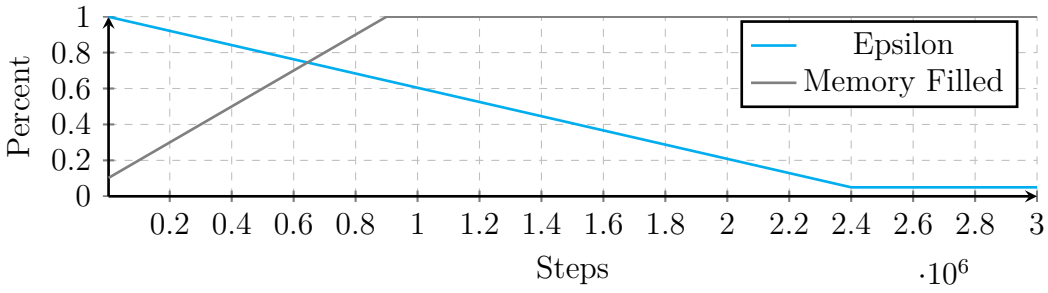


Figure 5.34: DQN Parameters (Epsilon and Memory Filled)

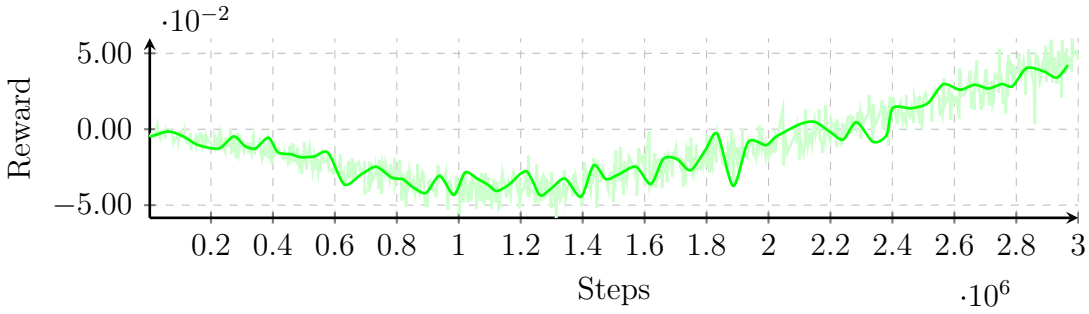


Figure 5.35: Average Sample Reward

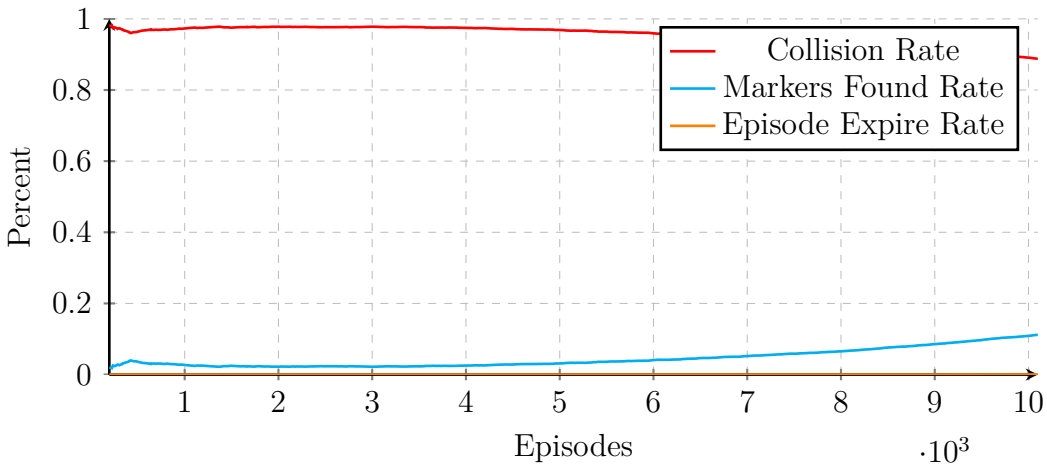


Figure 5.36: Episode outcome rates: collision rate, marker found rate, time expired rate

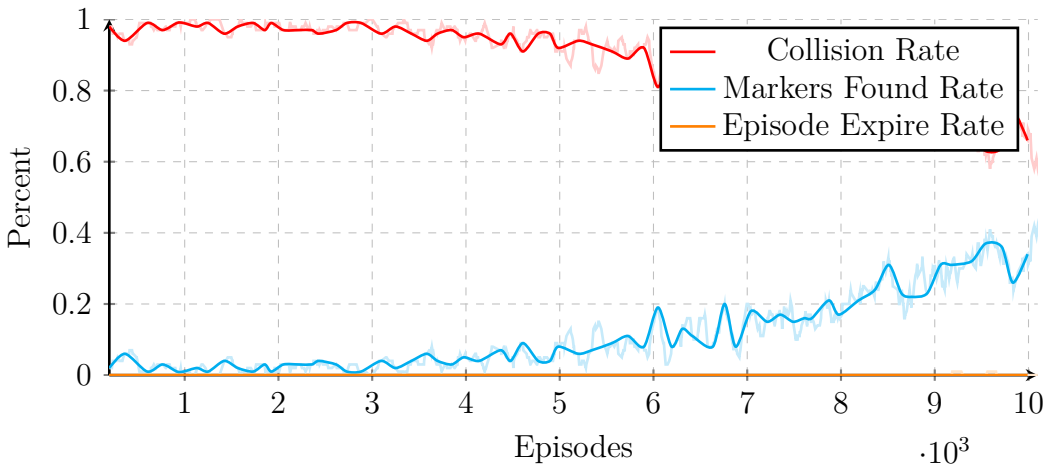


Figure 5.37: Episode outcome rates in last 100 episodes: collision rate, marker found rate, time expired rate

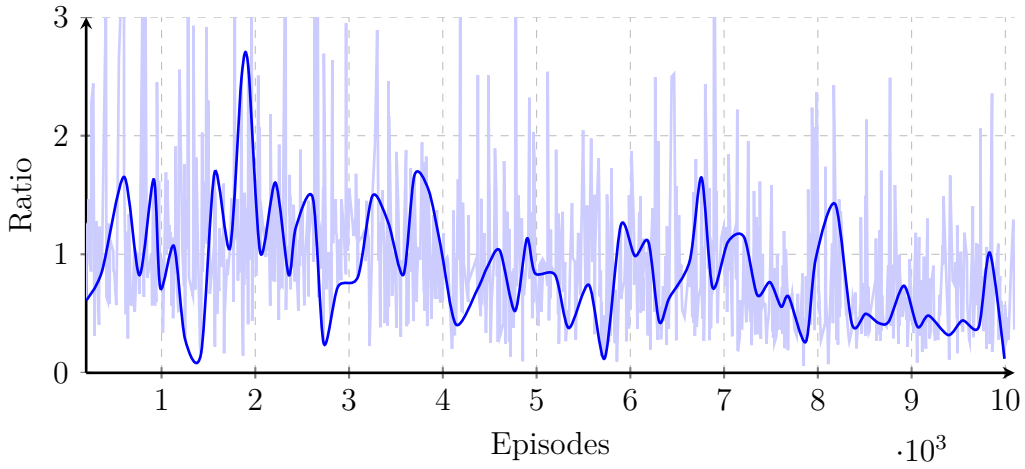


Figure 5.38: Relative Marker Approach from initial to termination point

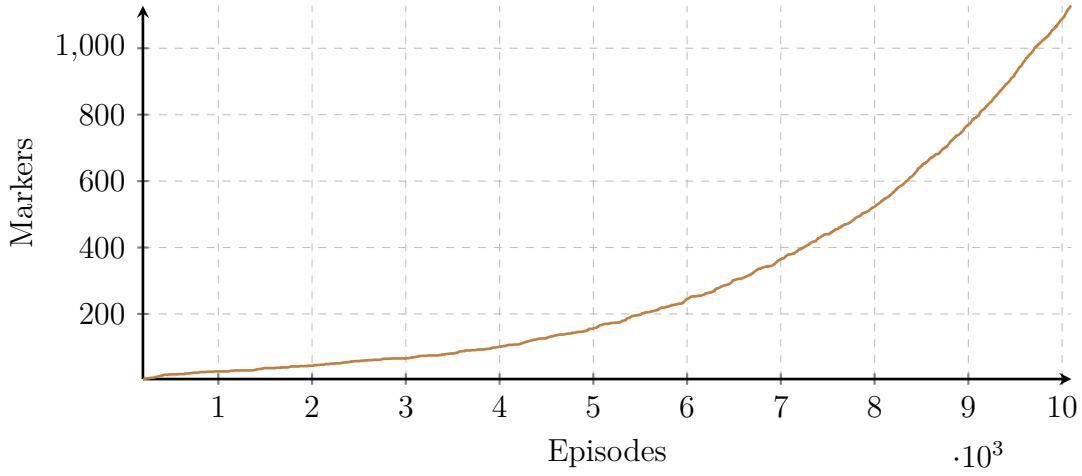


Figure 5.39: Total Markers Found

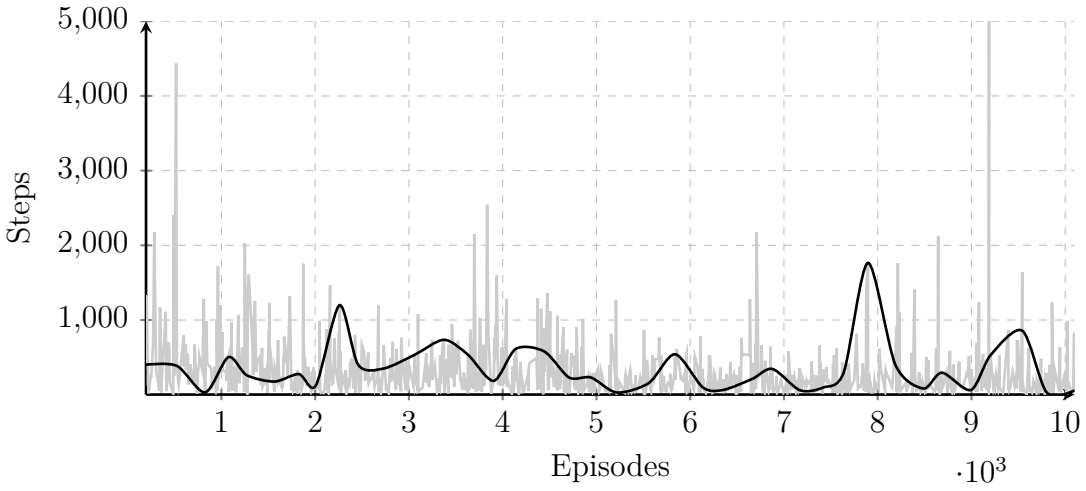


Figure 5.40: Episode Length

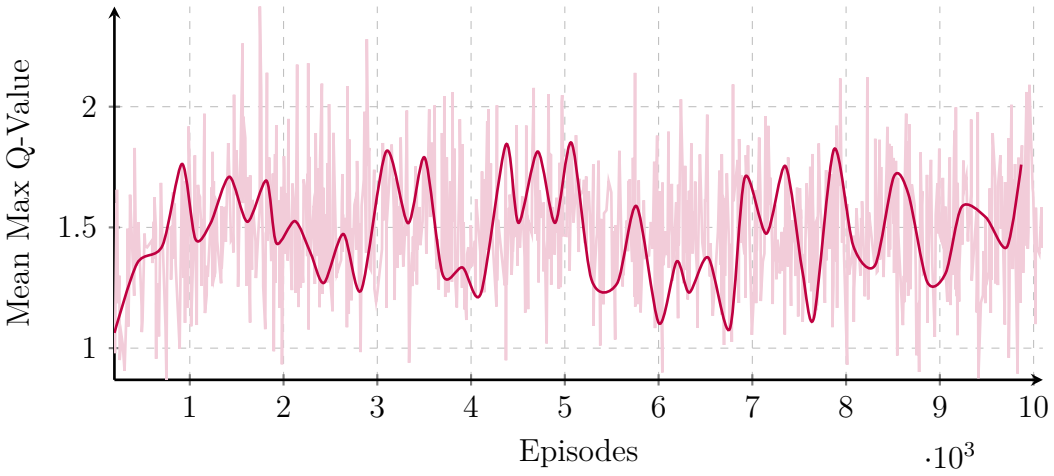


Figure 5.41: Mean Max Q-Value

5.2.5 Extreme World, Marathon Training

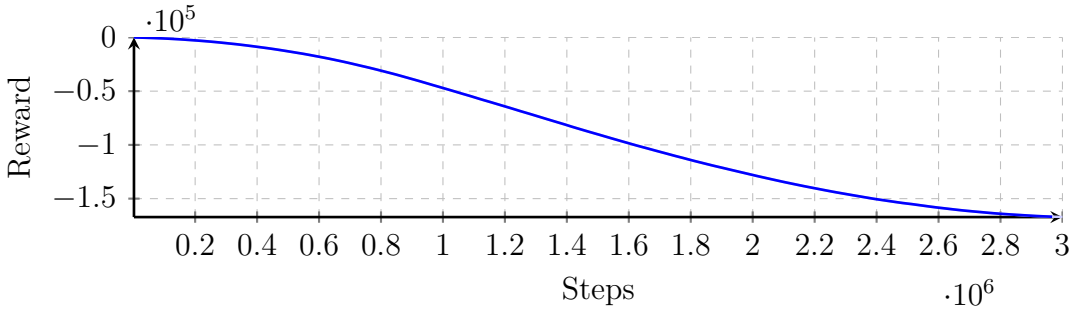


Figure 5.42: Total Reward

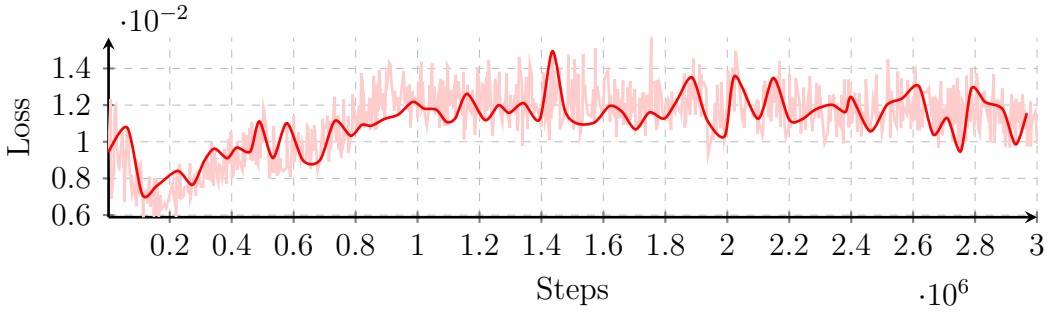


Figure 5.43: Network Weight Loss

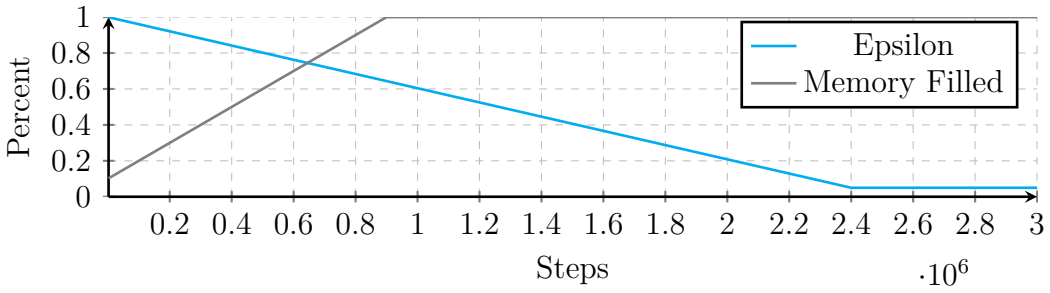


Figure 5.44: DQN Parameters (Epsilon and Memory Filled)

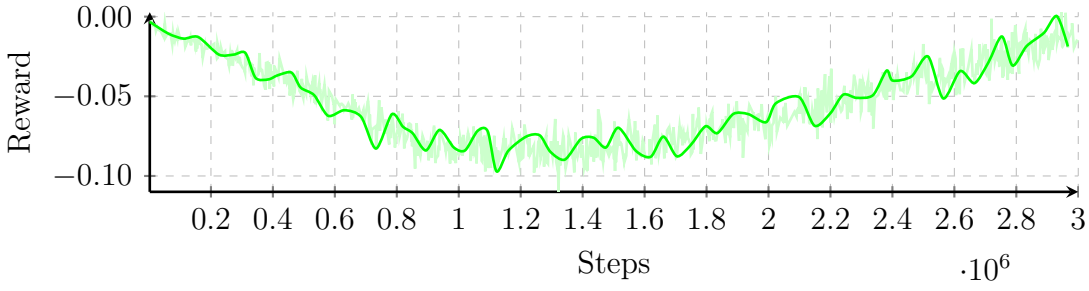


Figure 5.45: Average Sample Reward

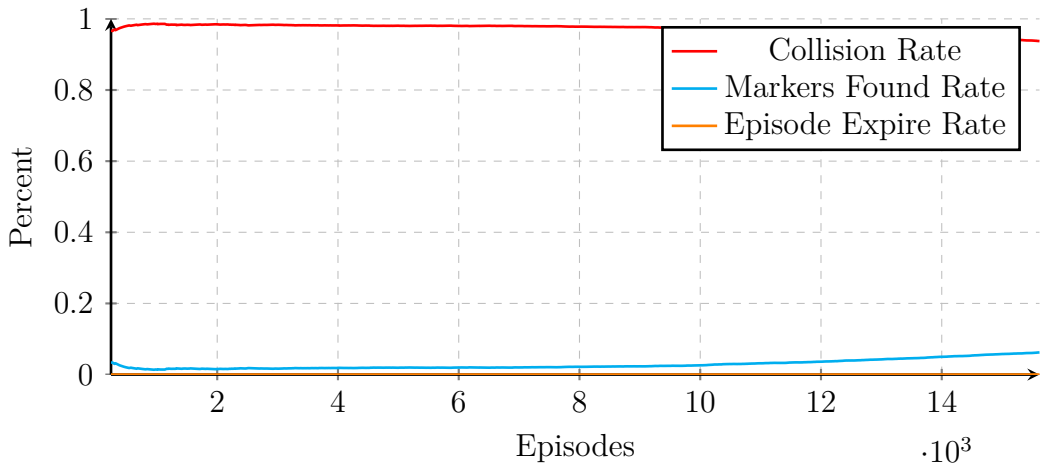


Figure 5.46: Episode outcome rates: collision rate, marker found rate, time expired rate

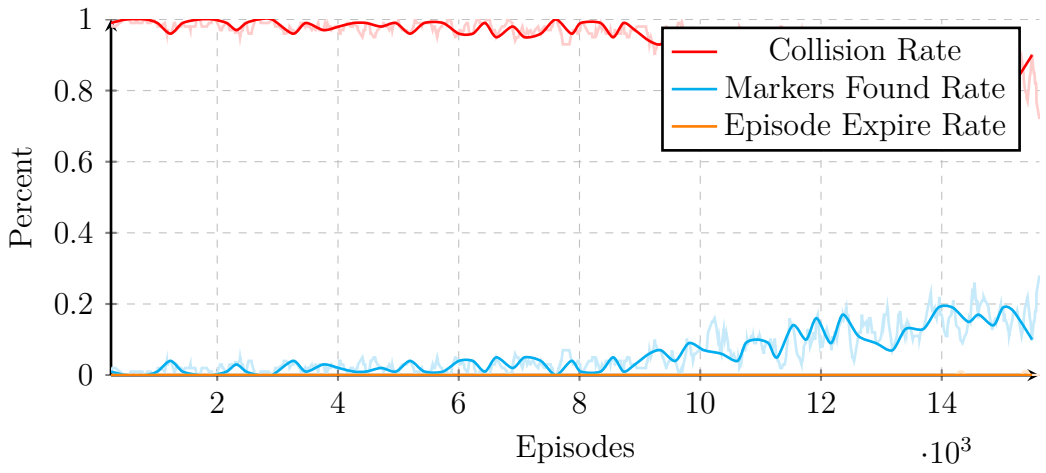


Figure 5.47: Episode outcome rates in last 100 episodes: collision rate, marker found rate, time expired rate

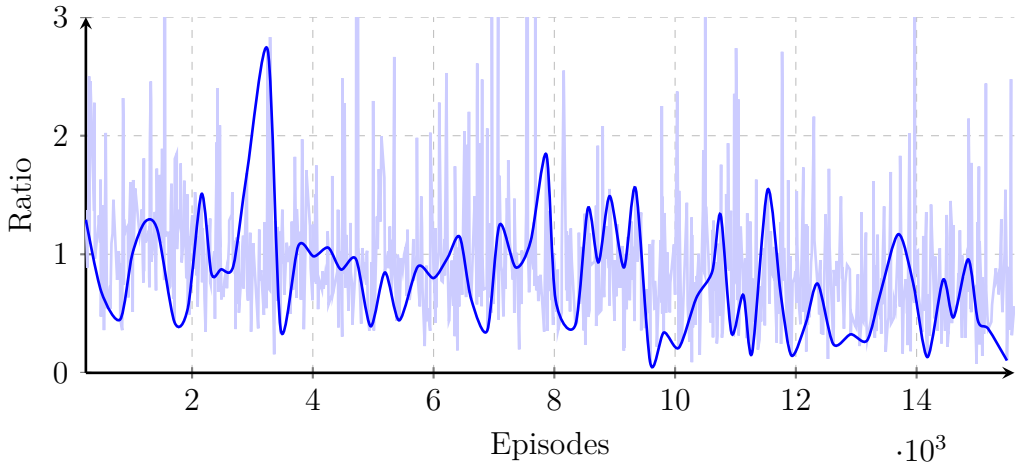


Figure 5.48: Relative Marker Approach from initial to termination point

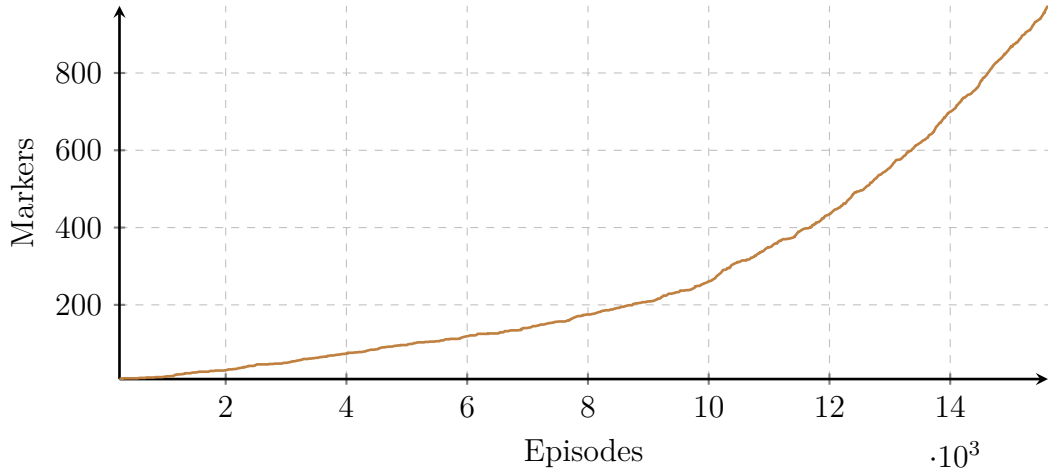


Figure 5.49: Total Markers Found

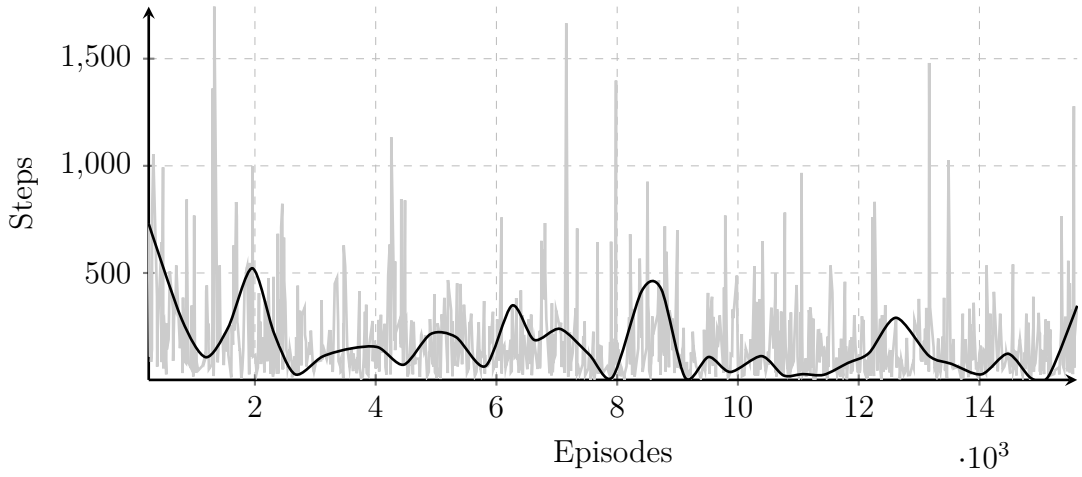


Figure 5.50: Episode Length

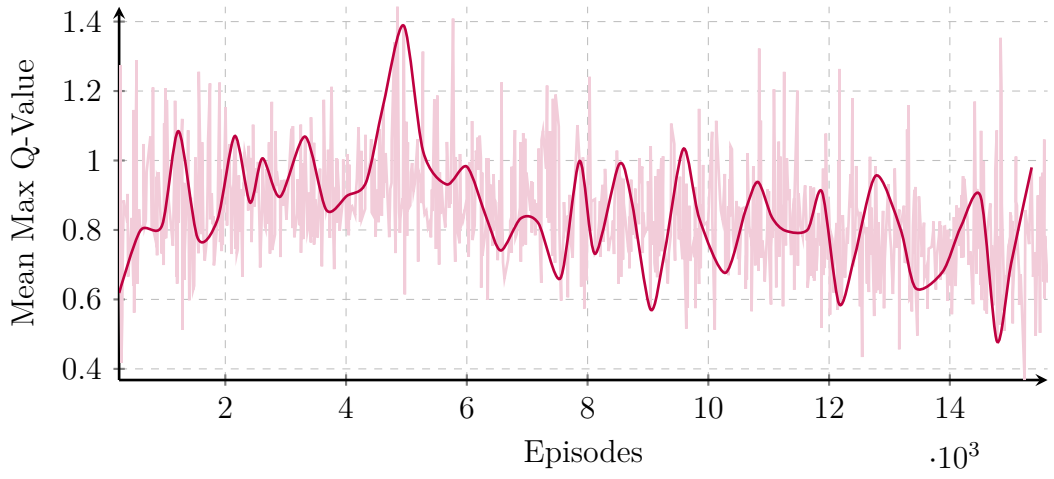


Figure 5.51: Mean Max Q-Value

5.2.6 Highlights

Figure 5.52: Comparison of marker found rates (last 100 episodes) between worlds

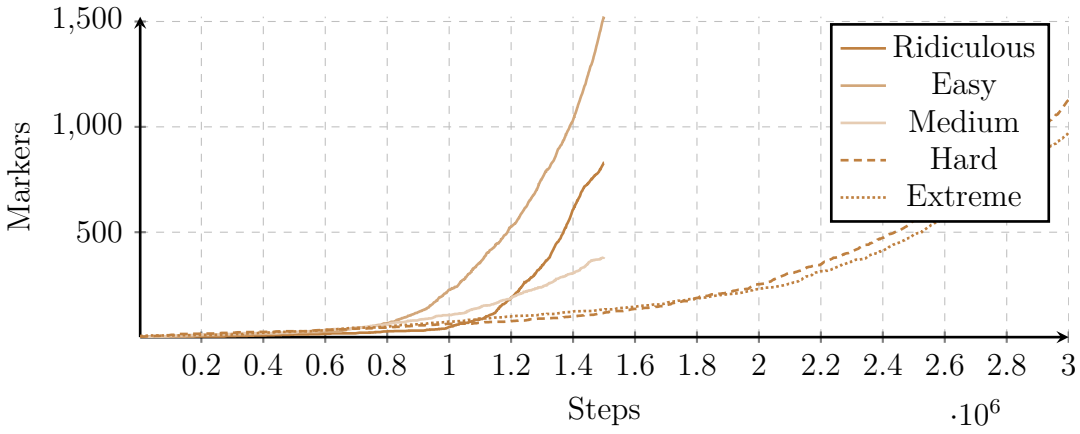


Figure 5.53: Comparison of total markers found between worlds

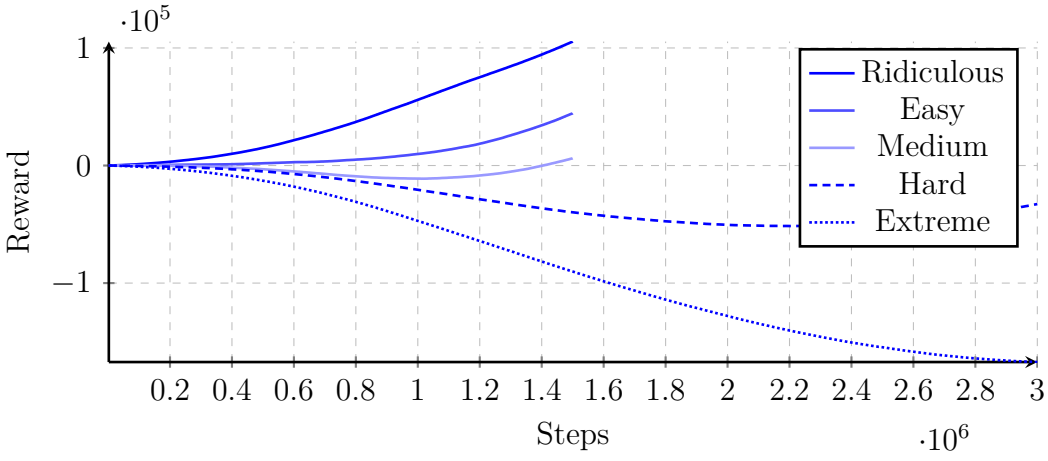


Figure 5.54: Comparison of total rewards between worlds

World Difficulty Level	Ridiculous	Easy	Medium	Hard	Extreme
Training Steps	1.5M	1.5M	1.5M	3M	3M
Marker Found Rate (Last 100)	0.46	0.71	0.22	0.4	0.28
Collision Rate (Last 100)	0.54	0.29	0.78	0.6	0.72
Relative Marker Approach (Average)	0.79	0.76	0.64	0.65	0.77
Total Reward	105.450	44.446	6.145	-32.784	-167.170

Table 5.3: Comparison of agent's performance between world profiles

5.2.7 Results and Observations

Training results appear to be promising at a first glance. For every scenario, metrics show a similar behavioral pattern, indicating a poor initial agent performance and a significantly improved version by the end of each training session.

Total reward decreases, as the world becomes more difficult (Figure 5.54), since the absence of obstacles in easier worlds, cause the reward system to provide positive rewards more frequently. Even if the total reward is negative, the rate increase is always positive, indicating that longer training sessions eventually lead to positive total rewards. Similarly, the average sample reward gradually decreases at the beginning, as the replay memory is being filled for the first time, but then increases during the second and third pass.

Loss always drops instantly as shown in Figures (5.3, 5.13, 5.23, 5.33, 5.43), but then slightly increases over time. This may be due to a relatively high learning rate which causes the network to diverge and fail to estimate the minimum of the loss function.

The most important metric of this experiment is the rate at which the drone is approaching targets and colliding with objects (Figures 5.6, 5.16, 5.26, 5.36, 5.46 and 5.7, 5.17, 5.27, 5.37, 5.47 and 5.52). In every case, collision rate always decreases while the rate of approaching targets always increases, which means the total markers found is increasing exponentially.

Another important metric to consider is how far the UAV moved towards its target with the relative marker approach (RMA) metric, in cases where a collision occurred. A value of 1 indicates that the drone collided at the same distance as its spawn location, while a value of 0.2 means the drone was five (5) times closer than before. Every training session concluded with an average RMA of <0.8 which indicates that the agent was consistently approaching the target before colliding.

Generally, a decrease of the agent's performance is expected as the world difficulty rises. The *ridiculous* profile performed surprisingly poor, acheiving a final MFR100 metric of 46% with an all-time-high of 56%. This may be due to the lack of robustness in a training session with no random elements (fixed marker position, fixed UAV spawn location). In contrary, the easy profile managed an impressive MFR100 of 71%, completing its mission more than 1500 times. *Medium* was slow to catch up and clearly required more training time, since it was outperformed by both upcoming configurations with rates of 40% and 28% respectively, as shown in Table (5.3).

On every occasion, the agent's behavior significantly improves, proving that DQN can be beneficial for this task. Results could be even greater with additional training, since the above charts indicate that the agent has not yet reached its peak performance. Training length was instead fixed to a total number of steps, which was required by the policy's epsilon decay rate. The goal was to observe how world difficulty increase affects the agent's performance over the same number of steps.

6 Conclusion

Contents

This final chapter summarizes the research of this thesis, underlying the potential of deep reinforcement learning. Several limitations are also presented, along with possible solutions and enhancements.

6.1 Summary

This thesis investigated a mapless approach to UAV autonomous navigation tasks in fully unknown 3D environments by incorporating deep learning techniques into a well-defined reinforcement learning problem (MDP). Specifically, the DQN algorithm was implemented, which integrates several key features, including a neural network architecture and a replay memory. Navigation is performed in dynamically generated Gazebo environments, which interacts with the agent (UAV) through the ROS framework.

Five experiments were conducted in order to evaluate the agent's performance. Results suggest that the agent can successfully learn to nagivate in the environment

and avoid obstacles. This also proved that the DQN algorithm with a number of tweaks can also be applied in a large variety of custom environments, not just ATARI games, which the original creators intended for.

6.2 Limitations and Optimizations

Unfortunately, this thesis has a few limitations and bottlenecks worth mentioning.

Firstly, the world does not accurately represent reality, with obstacle variation being limited to cubes, spheres and cylinders, instead of complex objects, such as trees and cars. This was mainly a performance measure, as upgraded object quality with complicated collision models would result in increased training times.

Additionally the obstacles are static, meaning their position is immutable throughout each episode. The model generally should be able to handle at least low-velocity moving obstacles with minimal sacrifice in the agent performance, albeit this thesis did not provide explicit testing of this configuration. Avoiding high-speed obstacles may require several modifications to both the model and the algorithm to confront high dynamic environments. A study [16] was able to overcome this challenge by using Long Short-Term Memory (LSTM) based DRL networks.

Several optimization measures can be applied in order to improve training speed and quality. In the currently implemented DQN algorithm, experience tuples are uniformly sampled from the replay memory. A study [6] managed to achieve superior performance by prioritizing samples based on their temporal-difference (TD) error. Picking samples with higher loss values will cause the neural network to faster minimize the error. Therefore, prioritized experience replay is an important improvement to consider. Another valuable change would be an automated hyperparameter optimization process. Current selection was performed empirically and there may exist other combinations providing even greater training results.

There are also several performance-improving methods to examine. The most beneficial is the use of swarm drones for parallel training. Each UAV would separately send input parameters to a shared neural network, allowing for faster exploration and sample extraction. Using multiple drones concurrently can multiply the agent's

training pace. A more efficient pause-resume system for Gazebo is also deemed necessary. The current implementation relies on services in order to send and receive requests, introducing significant delays to the pipeline and slowing down the training process by 30%.

Finally, as a cost-saving measure, the LIDAR sensor could be replaced by six ultrasonic sensors around the UAV's frame. Since we already utilize a neglegible percentage of the LIDAR's full potential, we could dramatically lower the drone's cost for an insignificant accuracy sacrifice.

Appendices

A
Aruco Marker

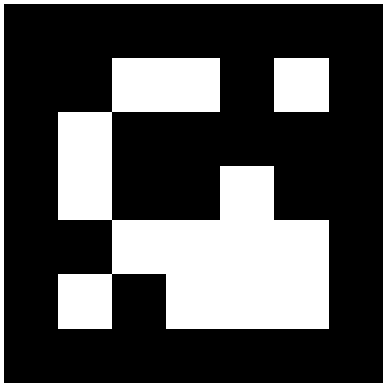
This appendix chapter will explain the basic functionality of an aruco marker.

A.1 Overview

An Aruco Marker [21, 22] is a square fiducial structure that consists of a $N \times N$ binary matrix enclosed in a black color border. Aruco markers are widely used in robotic applications, especially in pose estimation challenges, but they can also be utilized in augmented reality tasks.

Aruco markers consist of black and white blocks. A white block indicates a value of one (1), while a black block denotes a value of zero (0). Each marker, can act as a unique identifier depending on the composition of the internal binary matrix. Figure (A.1a) presents a 5×5 marker, with an ID of 42.

In practice, these markers can be found rotated in the environment, however the detection process needs to be able to determine its original rotation (corner-sensitive). Detection of aruco markers can be performed by the ArUco Library, which is based exclusively on the Open Source Computer Vision (OpenCV), the most popular machine vision framework, containing over 2500 optimized algorithms.



(a) Aruco Tag (id = 42)

(b) QR Code

Figure A.1: Figure showing difference between an aruco marker and a QR code

A.2 Aruco Marker vs QR-Code

Although an aruco marker and a QR code seem to share similar concepts, they are different at encoding information and thus have separate use cases. QR code is a general-purpose tool that is used to encode information, such as websites, usernames, passwords, text data, booking tickets and more. Aruco markers, on the other hand, are primarily used in research areas to help a camera estimate its relative position and rotation.

References

[1] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (May 1992), pp. 279–292. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.

[2] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (Dec. 2013).

[3] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (Feb. 2015). DOI: 10.1038/nature14236.

[4] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG].

[5] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. 2016. arXiv: 1511.06581 [cs.LG].

[6] Tom Schaul et al. *Prioritized Experience Replay*. 2016. arXiv: 1511.05952 [cs.LG].

[7] P. Newman, D. Cole, and K. Ho. "Outdoor SLAM using visual appearance and laser ranging". In: *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.* 2006. DOI: 10.1109/ROBOT.2006.1641869.

[8] Chenglu Wen et al. "Three-Dimensional Indoor Mobile Mapping With Fusion of Two-Dimensional Laser Scanner and RGB-D Camera Data". In: *IEEE Geoscience and Remote Sensing Letters* 11.4 (Apr. 2014), pp. 843–847. DOI: 10.1109/LGRS.2013.2279872.

[9] Christopher R. Hudson and Leighanne Hsu. "Simplistic Sonar based SLAM for low-cost Unmanned Aerial Quadrocopter systems". In: 2013.

[10] Matthew N. Dailey and Manukid Parnichkun. "Landmark-based Simultaneous Localization and Mapping with Stereo Vision". In: *In Proc. Asian Conference on Industrial Automation and Robotics*. 2005.

[11] Rodrigo Munguía et al. "Vision-Based SLAM System for Unmanned Aerial Vehicles". In: *Sensors* 16.3 (2016). DOI: 10.3390/s16030372. URL: https://www.mdpi.com/1424-8220/16/3/372.

[12] Mostafa Rizk et al. "Real-Time Slam Based on Image Stitching for Autonomous Navigation of UAVs in GNSS-Denied Regions". In: Mar. 2020. DOI: 10.1109/AICAS48895.2020.9073793.

[13] Stephan Weiss, Davide Scaramuzza, and Roland Siegwart. "Monocular-SLAM–based navigation for autonomous micro helicopters in GPS-denied environments". In: *Journal of Field Robotics* 28.6 (2011), pp. 854–874. DOI: https://doi.org/10.1002/rob.20412. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.20412. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.20412.

[14] Anna Guerra et al. "Reinforcement Learning for UAV Autonomous Navigation, Mapping and Target Detection". In: *2020 IEEE/ION Position, Location and Navigation Symposium (PLANS)* (Apr. 2020). DOI: 10.1109/plans46316.2020.9110163. URL: http://dx.doi.org/10.1109/PLANS46316.2020.9110163.

[15] Nursultan Imanberdiyev et al. "Autonomous navigation of UAV by using real-time model-based reinforcement learning". In: *2016 14th International Conference on Control, Automation, Robotics and Vision (ICARCV)*. Nov. 2016, pp. 1–6. DOI: 10.1109/ICARCV.2016.7838739.

[16] Tong GUO et al. "UAV navigation in high dynamic environments: A deep reinforcement learning approach". In: *Chinese Journal of Aeronautics* 34.2 (2021), pp. 479–489. DOI: https://doi.org/10.1016/j.cja.2020.05.011. URL: https://www.sciencedirect.com/science/article/pii/S1000936120302247.

[17] Chao Wang et al. "Autonomous Navigation of UAVs in Large-Scale Complex Environments: A Deep Reinforcement Learning Approach". In: *IEEE Transactions on Vehicular Technology* 68.3 (Mar. 2019), pp. 2124–2136. DOI: 10.1109/TVT.2018.2890773.

[18] Lei Tai and Ming Liu. "A robot exploration strategy based on Q-learning network". In: *2016 IEEE International Conference on Real-time Computing and Robotics (RCAR)*. June 2016, pp. 57–62. DOI: 10.1109/RCAR.2016.7784001.

[19] John J Craig. *Introduction to robotics*. 3rd ed. Addison-Wesley Publishing Company, 1986, 2005.

[20] Dimitrios Chatziparaschis. "Machine Learning for Enhancing Robotic Perception and Control". Master's Thesis. Technical University of Crete, 73100: Electrical and Computer Engineering, Technical University of Crete, Nov. 2020.

[21] Sergio Garrido-Jurado et al. "Generation of fiducial marker dictionaries using Mixed Integer Linear Programming". In: *Pattern Recognition* 51 (Oct. 2015). DOI: 10.1016/j.patcog.2015.09.023.

[22] Francisco Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. "Speeded Up Detection of Squared Fiducial Markers". In: *Image and Vision Computing* 76 (June 2018). DOI: 10.1016/j.imavis.2018.05.004.