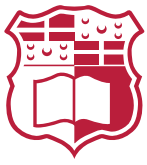
Autonomous Drone Control with Reinforcement Learning

Kian Parnis

Supervisor: Dr. Josef Bajada

June 2025

Submitted in partial fulfilment of the requirements
for the degree of Bachelor of Science in Information Technology (Honours)
(Artificial Intelligence).



L-Università ta' Malta
Faculty of Information &
Communication Technology

Abstract

This project aims to develop a system for autonomous drone control that focuses on the problem of drone obstacle avoidance. The successful development of such a system is crucial for ensuring the safe and efficient deployment of drones across various industries, including search and rescue operations, package delivery, and infrastructure inspections. The specific problem this thesis is addressing is developing a reinforcement learning-based solution for unmanned aerial vehicles (UAVs) that enable drones to safely navigate through unmapped cluttered environments, including obstacles that are either static or moving.

To achieve this, AirSim was used to simulate drone physics, and the UnReal engine was used to construct its simulated environment. As part of the RL approach, the project incorporated a depth sensor to capture environmental data of the drone's surroundings. This data was utilized as the state input of the reinforcement learning algorithm to learn and make decisions about the surrounding environment. The agent was provided with various observations representing the state of the environment. The most significant observation was the depth imagery that was captured at every step using the drone's depth sensor. This state was processed using a Convolutional Neural Network (CNN), which extracted and learned relevant features from these images. In addition to depth imagery, the agent also received information on its current velocity, its current distance from the goal, and a history of its previous actions. These actions are passed through an Artificial Neural Network (ANN) before being flattened and combined with the processed imagery to be fed to the agent. This framework was utilized to train four RL algorithms to navigate environments with static obstacles and train the best two RL algorithms on environments with dynamic obstacles. The two discrete models were trained using the Deep Q-Network (DQN) and Double Deep Q-Network (DDQN) algorithms, while the two continuous models were trained using the Proximal Policy Optimization (PPO) and Trust Region Policy Optimisation (TRPO) algorithms.

This ultimately resulted in successful policies that could avoid obstacles and reach their destination in complex environments. The best result obtained was with the Double Deep Q-Network algorithm which reached its target goals 93% of the time in an environment with static obstacles and an average of 84.5% target goals reached in an environment with dynamic obstacles.

Acknowledgements

I dedicate this dissertation to my mother, whose unwavering support and encouragement have driven me to pursue my passion for Artificial Intelligence. Without her guidance, I would not have reached this point on the path to academic success. To my sister Khaila, whose creative aspirations inspire me always.

I'm sincerely grateful to my supervisor Dr. Josef Bajada, whose patience, kindness, and invaluable advice have kept me motivated, confident, and proud of the work put into this project which led to achieving my aims and objectives.

My appreciation extends to all my friends, including my course-mates Mario, Evangeline, Max, Gabriel, and Kieron who supported me throughout this wonderful undergraduate experience, as well as my long-time friends Roberto, Jamie, Nik, Rodney, Rouen, Levi, Kayjay, John, and David who never fail to make me smile.

Lastly, my heartfelt thanks to my best friend Aimee, who has shown unwavering interest in my endeavors and has provided constant support during difficult moments. Your encouragement keeps me moving forward.

"The struggle itself toward the heights is enough to fill a man's heart.
One must imagine Sisyphus happy."

—'The Myth of Sisyphus,' Albert Camus

Contents

List of Figures

List of Tables

List of Abbreviations

UAV Unmanned Aerial Vehicles
RRT* Rapidly Exploring Random Tree*
RL Reinforcement Learning
DQN Deep Q Network
LiDAR Light Detection and Ranging
Radar Radio Detection and Ranging
CNN Convolutional Neural Networks
Deep RL Deep Reinforcement Learning
ANN Artificial Neural Networks
ReLu Rectified Linear Unit
MDP Markov Decision Process
DDQN Double Deep Q-Network
TRPO Trust Region Policy Optimisation
PPO Proximal Policy Optimisation
MMW Millimeter Wave
JNN Joint Neural Network
D3QN Double Dueling Deep Q-Network
ACKTR Actor-Critic using the Kronecker-Factored Trust Region
DRQN Deep Recurrent Q-Network
LSTM long short-term memory network
TP True Positive
TN True Negative
FP False Positive
FN False Negative

1 Introduction

Drones, also known as unmanned aerial vehicles (UAVs), are remotely operated aircraft that can be controlled manually or autonomously. Autonomous drone control refers to the capability of UAVs to perform tasks and navigate through their environment without direct human intervention. Instead of relying on manual control, autonomous drones utilize sophisticated onboard systems, sensors, and algorithms to make intelligent decisions and execute predefined missions. However, the problem lies in developing self-governing solutions that empower UAVs to fulfill essential tasks effectively. Currently, UAVs heavily rely on manual control by human pilots, which becomes impractical in scenarios where the pilot is unable to maintain a connection with the vehicle, such as in search and rescue operations or large-scale healthcare delivery. Extensive research [1–5] is being conducted to enable UAVs to operate autonomously, with various approaches being explored in literature.

1.1 Problem Definition

The specific problem this thesis is addressing is developing a solution for unmanned aerial vehicles (UAVs) to navigate unmapped, dynamic, and static environments. The UAV must rely on onboard sensors to gather real-time information due to the absence of prior knowledge. The solution should effectively handle uncertainties, variations, and unpredictable obstacles, including both dynamic elements (e.g., moving objects) and static elements (e.g., fixed structures). Considering the presence of pre-existing static objects, such as buildings, enhances the UAV's situational awareness and allows for informed decisions to ensure safe and efficient flight. This becomes particularly crucial in situations such as natural disaster scenarios, where the lack of prior knowledge is prominent, as events like avalanches or earthquakes can introduce new static elements or modify existing ones.

1.2 Motivation

The need for autonomous drone control is motivated due to the sheer number of different applications that can utilize such technology. In [6] the use of Reinforcement Learning with drone control is highlighted in different problems such as path planning, navigation, and control. The following is a breakdown of some of the different applications highlighted by [6] which are Altitude control, Obstacle avoidance, Drone delivery, and Unmanned Rescue. Respective research in these fields can be found in [1–4]. This project focuses on one of the most noticeable tasks UAVs need to

overcome which is Obstacle Avoidance. The motivation behind focusing on Obstacle Avoidance is due to its overall importance in its involvement in various critical tasks. When carrying out a search and rescue operation for example. The autonomous drones' knowledge of navigating an adaptive environment is paramount to the overall success of the operation.

1.3 Challenges

This solution depends on the agent being able to navigate through static or dynamic environments that the agents had no prior knowledge of, depending on the limited data such as its sensor data, as well as its GPS signals to reach its targeted goal. The traditional path-planning algorithms, such as A* and RRT*, face several challenges in the context of the problem being addressed [7, 8]. A* relies on graph-based approaches and requires the drone to navigate the environment to gain knowledge about its surroundings. This can be problematic as it may involve flying large distances and backtracking, which is inefficient and risky in hazardous environments. Additionally, A* would need to recalculate its trajectory from scratch if the environment shifts, such as when obstacles move or wind interference occurs. On the other hand, RRT* is unable to handle dynamic environments and needs to reconstruct its mapped tree if the agent encounters wind interference.

To overcome these challenges, Reinforcement Learning (RL) can be employed. RL enables the agent to learn from its surroundings and make informed decisions based on acquired knowledge. By interacting with the environment, RL algorithms like Q-Learning [9] and Deep Q Network (DQN) [10] can learn which behaviors yield favorable outcomes. In the case of path planning for a drone in a dynamic environment, an RL agent can adjust to changes such as moving obstacles or wind interference by gathering data from its sensors and modifying its policies accordingly. Unlike A* or RRT* algorithms, this approach eliminates the need for backtracking or recalculating trajectories from scratch. Instead, the RL agent can continuously update its policy to adapt to the changing environment.

1.4 Aims and Objectives

The aim of this project is to develop algorithms for autonomous drone navigation that are effective in obstacle avoidance in unmapped environments. More specifically the goal is to train agents using different state-of-the-art Reinforcement Learning Techniques which are capable of navigating through environments that contain both static and dynamic obstacles. This will be accomplished by fulfilling the following

objectives:

1. **Objective (O1):** Integrate a drone environment simulator within a Reinforcement Learning framework such that the drone can be programmatically controlled.
2. **Objective (O2):** Identify what sensor information to use for the observations and evaluate which algorithms and configurations perform obstacle detection accurately.
3. **Objective (O3):** Train four RL algorithms to navigate environments with static obstacles, with two algorithms using discrete actions and two algorithms using continuous actions, and evaluate them by the number of successful runs without collision.
4. **Objective (O4):** Train the best two RL algorithms on environments with dynamic obstacles, using adapted observations and rewards to capture obstacle movement information and evaluate them by the number of successful runs without collision.

1.5 Report Structure

The remainder of this report is organised as follows:

Chapter 2 provides background information to help the reader better understand the project's various components.

Chapter 3 reviews the latest research to identify the most appropriate algorithms and data to use in this project.

Chapter 4 describes the approach taken to meet the project's objectives.

Chapter 5 uses the metrics reviewed in Chapter 3 to compare the different algorithms and sensor architectures used in this project and evaluate their performance.

Chapter 6 revisits the project's aims and objectives to determine if they have been achieved, together with suggestions of future work that can be carried out to build upon the achievements of this project.

2 Background

This chapter aims to aid readers' comprehension of essential terms and techniques frequently used in this field. We begin by examining the Kinematic and Dynamic state of a Drone. Next, we provide a high-level explanation of the sensors that UAVs can use, along with a discussion of how Convolutional Neural Networks can aid these sensors. Finally, we provide an overview of Reinforcement Learning, explaining various concepts, and delving into technical explanations of some of the state-of-the-art algorithms.

2.1 Kinematics and Dynamics

Figure 2.1 shows the different variables of the kinematic and dynamic state of the drone. These states are the drone's position and orientation in space, represented by Cartesian coordinates (x, y, z) for position and Euler angles (ϕ, θ, ψ) for orientation. The dynamic state of the drone includes variables such as $v(x), v(y), v(z)$ for linear velocity along the x, y, and z axes, respectively. The drone's angular velocity is represented as the Roll(ϕ) rate, Pitch(θ) rate, and Yaw(ψ) rate. These states provide information about the drone's position, orientation, velocity, and angular velocity.

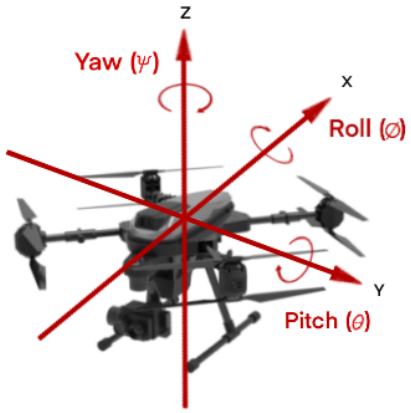


Figure 2.1 Kinematic and Dynamic State Variables of a drone

The drone's forward and backward motion is represented by the x-plane, while its side-to-side motion is represented by the y-plane. The drone's up-and-down or vertical movement is represented by the z-plane. A positive velocity in the x-plane, for instance, would mean that the drone is traveling forward at a constant speed. We would argue that it has a negative acceleration in the x-plane if it then starts to slow down. If for example, the drone starts to move up, it would have a positive velocity in the z-plane. ϕ signifies tilting from side to side, θ represents tilting forward or

backward, and ψ indicates the heading or direction of the drone. The drone's Roll(ϕ) rate, Pitch(θ) rate, and Yaw(ψ) rate capture the angular velocities around the x, y, and z axes.

2.2 Sensors

In this section, an overview of each sensor considered shall be given, this will be followed up in Section 3.1, where different works will be examined to determine which sensor best fits this project's needs. UAVs require sensors that can supply precise and trustworthy data to accomplish the critical task of directing their navigation and decision-making processes in accomplishing successful obstacle avoidance. The sensors which are considered are LiDAR, Radar, Camera, and Depth Sensor.

Light Detection and Ranging sensor (LiDAR) uses lasers to measure distances and create 3D maps of the surrounding environment. It works by emitting laser pulses and measuring the time it takes for the pulse to bounce back from objects in the environment. LiDAR sensors are typically used in autonomous vehicles, drones, and robotics for navigation, obstacle detection, and mapping.

Radio Detection and Ranging (Radar), is a similar remote sensing technology that uses radio waves instead of laser beams to detect objects. The sensor emits a radio wave that reflects off the object and returns to the sensor, allowing the sensor to calculate the distance based on the time it takes for the wave to return. Radar sensors can be used for speed measurement, obstacle detection, and object tracking.

Cameras capture visual images and videos of the surrounding environment. It works by using lenses to focus light onto a photosensitive surface, which converts the light into electrical signals that can be processed and stored. Cameras are commonly used in drones, robotics, and surveillance systems for visual inspection, object recognition, and tracking.

Depth sensors, also known as range sensors, are devices that measure the distance between the sensor and objects in the surrounding environment. They can be based on various technologies, such as infrared, ultrasound, or time-of-flight, and can be used for obstacle avoidance, indoor mapping, and robotics.

2.3 Artificial Neural Networks

Artificial Neural Networks (ANN) is a collection of perceptrons and activation functions. The Network is comprised of an input layer, an output layer, and several hidden layers (see Figure 2.2) that map the input to its output [11].

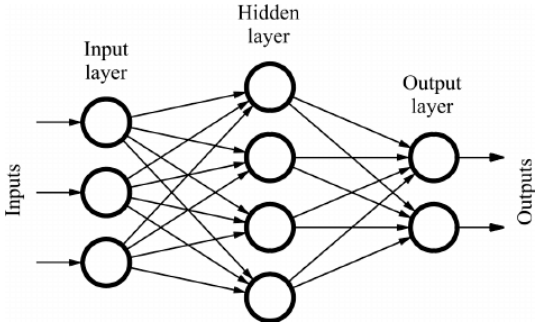


Figure 2.2 Artificial Neural Network

A Perceptron is a fundamental unit in ANNs, they receive a number of real values with each input given a weight, with the weight determining the contribution of the input to the output. Its computations are the linear combination of these inputs plus a certain bias to produce a single output Y. (This is shown in Equation 2.1).

$$Y = \sum(weight * input) + bias \tag{2.1}$$

Perceptrons are limited to linear functions [11], so neural networks use activation functions to introduce nonlinearity. The simplest activation function is the binary Step Function [12], which fires a value of 1 when the summation Y exceeds a threshold and 0 otherwise, which makes this activation function piece-wise linear. The Sigmoid Function [12] is a smoothened version of the step function that's used to represent non-linear functions between 0 and 1. The Sigmoid function is denoted as:

$$\sigma(Y) = \frac{1}{1 + e^{-Y}} \tag{2.2}$$

The Tanh Function [12] is a scaled version of the Sigmoid Function, this function maps the values between -1 and 1 which makes its gradients more stable. The Tanh function is denoted as:

$$\tanh(Y) = \frac{2}{1 + e^{-2Y}} - 1 \tag{2.3}$$

The problem with the Sigmoid and Tanh functions is that they fire all the time making an ANN heavy. The Rectified Linear Unit (ReLu) [13] is a less computationally expensive activation function that maps the input Y to the max of either 0 or Y. This lets large numbers pass making few perceptrons stale. The ReLu is denoted as:

$$\sigma(Y) = max(0, Y) \tag{2.4}$$

2.4 Convolutional Neural Networks

As described in Section 2.2, drone imagery can be used as one of the observations that can be fed to a Reinforcement Learning algorithm to receive information about its surrounding environment. However, If a regular feed-forward neural network were to be used for this imagery, its large size would result in a huge number of neurons being needed. For example, an image with size 300x300 px and 3 color channels would result in the network having 270,000 weights (300x300x3), which is not only computationally expensive but leaves the network prone to over-fitting (learning to memorize its data set) due to a large number of parameters.

Convolutional Neural Networks (CNN) are based on the idea of Artificial Neural Networks but aim to use the idea of mathematical convolution to perform well with image, speech, or audio signal inputs. CNNs gained popularity based on the ImageNet architecture during 2012 [14].

Convolutions layers utilize Kernels (Also referred to as Filters and Feature Detectors) which are small matrices known as Edge Detectors that stride across the image starting from the top and progressively going downward per stride. For each stride, a dot product between the Kernel and the current position on the image is taken and saved into a Feature Map. This Feature Map would then reduce the image (depending on the kernel size) with specific patterns of information in an image being highlighted.

Pooling Layers are used to down-sample the feature map along its spatial dimensionality [15] to further reduce parameter size while maintaining important features which helps with the previously described over-fitting problem. In [16], Reinforcement Learning, Deep Learning, and Convolutional Neural Networks were successfully integrated together which used sensor data and game scores to learn successful control policies. The modified RL algorithm Deep Q-Learning agent [10] managed to beat several Atari Games such as Space Invaders, Pong, and Breakout while achieving a level comparable to that of professional human players.

2.5 Reinforcement Learning

Reinforcement Learning (RL) is the process of learning what to do, by interacting with the environment. This involves the recognition of similarities between the characteristics of different situations and mapping these situations to actions, with the overall goal being to maximise some utility that is of interest. RL is typically broken down into an interaction loop (see Figure 2.3), this involves our agent taking some action in an environment based on some policy that transitions its current state to a

new state while collecting some reward as well as an observation.

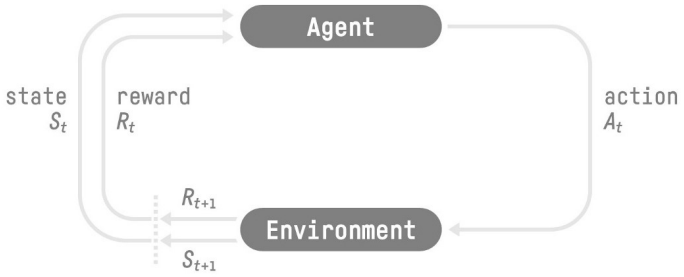


Figure 2.3 RL Interaction Loop [17]

The policy, typically denoted as π, is a function that maps a state s_t to an action A_t (i.e., how the agent acts in each different environmental state, where t is the current time step). Rewards, R_t, are represented as numerical values given back to the agent as a result of the action taken in a state. Rewards are used to improve the agent's policy in deciding what action to take in a particular state to reach its target goal. The optimal policy π^* and is achieved by maximising an expected cumulative reward (the sum of all rewards achieved after a particular time step) [18]. The environment models the rules of the world and is typically modeled around aiding the agent to train efficiently to reach its desired objectives/goal. The environment could either be the real world or a simulation of it, with the latter being a cheaper, faster, and safer way to conduct training.

Deep Reinforcement Learning (Deep RL) is a sub-field of machine learning that combines Reinforcement Learning (RL) and Deep Learning. Deep RL is often used when the state space is not feasible to engineer due to space constraints. For example, the game of chess has 10^{47} states while continuous actions are considered to have an infinite state space. Deep Learning aids by utilizing an approximation function which is parameterized by a vector of weights and is typically characterized as Artificial Neural Networks (which includes CNNs and other variants of Deep Neural Networks).

2.6 Value Based Methods

Value-Based Methods are a class of Reinforcement Learning (RL) algorithms that aim to approximate the optimal value function which is denoted by $V_\pi(s)$, which maps states to the expected cumulative reward when following a certain policy. The main idea behind these methods is to estimate the value of being in a given state and then choose the action that maximises the value function. Value-Based methods are based on the Bellman equation, which expresses the relationship between the value of a state and the values of the next states and the rewards obtained by transiting them.

The Bellman equation for the value function is:

$$V_\pi(s) = \mathbb{E}_\pi(R(s,a) + \gamma V_\pi(s')) \tag{2.5}$$

where s is the current state, a is the current action, $R(s,a)$ is the immediate reward, s' is the next state and $\gamma \in [0,1]$ is the discount factor which determines the importance of future rewards. In order to solve the above equation, we need to know the transition function denoted as $P(s^{'}|s,a)$ which gives the probability of going to state s' from a state s taking action a.

2.6.1 Deep Q-Network (DQN)

Deep Q-Network (DQN) [10] is a variant of the Q-learning algorithm [9] that uses a neural network to approximate the Q-function. The Q-function represents the expected cumulative future reward for a given action in a given state. The goal of Q-learning is to find the optimal Q-function that maps states to actions such that the expected cumulative reward is maximised. In Q-learning, the Q-function is typically represented as a table of values, where each entry corresponds to the expected cumulative reward for a given state-action pair. However, in real-world problems, the state space can be very large and complex, making it infeasible to represent the Q-function as a table. DQN addresses this issue by using a neural network to approximate the Q-function. The full Deep Q-Network algorithm is given in Appendix A labeled as Algorithm 1.

As DQN extends Q-learning, it also follows an off-policy learning strategy, meaning that it will explore the environment using an exploratory policy while optimizing a different target policy with respect to the estimated q values $\hat{Q}(s,a|\theta)$ where θ is the network's parameters. To update the network's parameters θ, DQN uses the following loss function:

$$L(\theta) = \frac{1}{|K|}\Sigma_{i=1}^{|K|}[R_i + \gamma max_a \hat{q}(S_i^{'}, a_i|\theta_{targ}) - \hat{q}(S_i, A_i|\theta)]^2 \tag{2.6}$$

where: $L(\theta)$ is the loss, K is the size of the batch, R_i is the current batch reward, $\gamma \in [0,1]$ is the discount factor, $max_a \hat{q}(S_i^{'}, a_i|\theta_{targ})$ is the maximum approximated q value from all the possible actions applied to $S_i^{'}$ according to the target policy (parametrized by weights θ_{targ}) and finally, $\hat{q}(S_i, A_i|\theta))$ is the current expected Q-value. The idea is to keep the weights of the target Q-network fixed for a certain number of steps and then update them to the weights of the Q-network being trained. The algorithm uses this fixed target network to prevent the Q-values from oscillating or diverging.

To improve the stability of the training process, DQN uses a technique called experience replay. In experience replay, the agent stores a dataset of past experiences (i.e., state, action, reward, next state) in a replay buffer. The agent then samples random mini-batches of experiences from the replay buffer to use in the training process, rather than using the most recent experience.

2.6.2 Double Deep Q-Network (DDQN)

Double Deep Q-Networks (DDQN) [19] is a variant of the Q-learning algorithm that addresses the problem of overestimation of Q-values that can occur in traditional Q-learning. The huge overestimation [20] of action values is the root cause of this subpar performance. Because Q-learning approximates the highest expected action value using the maximum action value, it suffers from maximization bias, leading to these overestimations. The principle behind DDQN is to use two separate neural networks—one for selecting actions (the 'online' network) and one for evaluating actions (the 'target' network). In Appendix A, Algorithm 2 shown in [19] illustrates these two distinct neural networks.

As explained, DDQN maintains two separate Q-value networks: Q^A and Q^B. At each step of the algorithm, an action is chosen based on the average of the Q-values in Q^A and Q^B for the current state s. The algorithm then selects either the Q^A or Q^B network to update based on some criterion (e.g., random selection). If Q^A is selected for update, the algorithm updates the Q-value for the current state-action pair using the following formula:

$$Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)(r + \gamma Q^B(s', \arg\max_a Q^A(s',a)) - Q^A(s,a)) \tag{2.7}$$

where: $Q^A(s,a)$ is the current Q-value estimate for taking action a in state s according to the Q^A network, $\alpha(s,a)$ is the step size (or learning rate) used to update the Q-value, r is the reward obtained after taking action a in state s, $\gamma \in [0,1]$ and finally, $Q^B(s', \arg\max_a Q^A(s',a))$ is the estimated value of the best action a in state s' according to the Q^B network. Similarly, if Q^B is selected for update, the algorithm updates the Q-value for the current state-action pair using the following formula:

$$Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s', \arg\max_a Q^B(s',a)) - Q^B(s,a)) \tag{2.8}$$

The algorithm continues to select actions and update the Q-value tables until some stopping criterion is met (e.g., a maximum number of iterations or convergence of the Q-value estimates).

2.7 Policy Based Methods

Policy Based Methods are also a class of Reinforcement Learning (RL) algorithms whose goal is to learn a policy function, denoted as a stochastic policy $\pi(a|s)$, which is the probability of choosing a in a state s, $P(a|s)$. This can also be extended to approximate π using parameter θ which represents the parameters of our policy as $\pi(a|s,\theta) = P(A_t = a|S_t = s, \theta_t = \theta)$. Given this policy approximation function, we would want to find the best parameters for θ. This is achieved with the use of Stochastic Gradient Accent which utilizes the score function $\nabla \ln \pi_\theta(s,a)$ and combines it with the return to adjust the parameter vector θ as follows:

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla \ln \pi_{\theta t}(S_t, A_t) \tag{2.9}$$

Where G_t is the return for state S_t and α is the step size hyper-parameter. Policy-based methods offer several benefits over value-based methods. Firstly, Policy-based methods can handle stochastic policies more naturally, avoiding the risk of getting stuck in suboptimal solutions. Secondly, policy-based methods support exploration implicitly, without requiring additional exploration strategies such as ϵ-greedy. This feature makes them suitable for environments with high-dimensional state spaces or sparse rewards, where exploration is particularly challenging. Lastly, policy-based methods can handle continuous action spaces more efficiently than value-based methods, which often require discretization of the action space. This is not to imply that value-based methods are worthless, policy gradients' high variance estimates of the gradient updates are one of their main drawbacks. This results in extremely noisy gradient estimations and can make learning unstable.

2.8 Actor-Critic Methods

Actor-Critic methods combine both value-based and policy-gradient algorithms together. These methods have two main components: an actor and a critic. The actor represents the policy, and the critic represents the value function. The actor learns to select actions, while the critic learns to evaluate the actions taken by the actor. The two components are trained together and influence each other.

Value-based methods are generally more sample efficient and converge faster than policy-based methods because they only need to estimate the value of a state or state-action pair, rather than the full policy [21]. However, they can be sensitive to the choice of function approximator and may have difficulty dealing with large or continuous action spaces. Policy-based methods, on the other hand, can handle large or continuous action spaces more easily, but they tend to have high variance estimates

and converge more slowly than value-based methods [20].

The combination of these two allows the agent to balance the trade-off between exploration and exploitation. Actor-Critic methods generally have a lower variance than pure policy-based methods and are more able to handle large or continuous action spaces than pure value-based methods.

2.8.1 Trust Region Policy Optimisation

In 'vanilla' policy gradient methods two key limitations are faced [21]. Firstly, selecting appropriate stepsizes becomes challenging due to the nonstationary nature of input data caused by changing policies, observations, and reward distributions. In contrast to supervised learning, a single bad step can have a severe impact on the visitation distribution, making it difficult to recover from. If the step is too drastic, it can lead to a poor policy, which affects the subsequent batches collected under that policy, resulting in a collapse in performance. Secondly, since, a drastic step size can collapse the policy the sample efficiency is harmed. Trust Region Policy Optimisation (TRPO) [22] is an actor-critic style algorithm that can be used in infinite state-space environments with continuous as well as discrete action spaces. TRPO handles the addressed issues 'vanilla' policy gradient methods have by imposing a constraint on the KL divergence to limit the magnitude of the policy update leading to a quicker and monotonic improvement in performance. The prescribed update for TRPO is as follows:

$$\theta_{k+1} = argmax_\theta \, \mathbb{E}_{s\sim p_{\theta_{old}} a\sim\pi_{\theta_{old}}}[\frac{\pi_\theta(a|s_n)}{\pi_{old}(a|s_n)} A_{\theta_{old}}(s,a)] \; s.t. \; \bar{D}_{KL}(\theta_{old},\theta) \leq \delta \tag{2.10}$$

where $\mathbb{E}_{s\sim p_{\theta_{old}} a\sim\pi_{\theta_{old}}}[\frac{\pi_\theta(a|s_n)}{\pi_{old}(a|s_n)} A_{\theta_{old}}(s,a)]$ is the surrogate advantage, a measure of how policy π_θ performs with respect to the old policy π_{θ_k} with the use of the old policies data. $\bar{D}_{KL}(\theta_{old},\theta)$ represents the mean KL divergence between policies for the states visited by the previous policy. Due to the complexity of this update, TRPO employs certain approximations to expedite the calculation process. These approximations are made using Taylor series expansion which is used to expand the objective and constraint as follows:

$$L(\theta) \approx g^T(\theta - \theta_{old}), D_{KL}(\theta) \approx \frac{1}{2}(\theta - \theta_{old})^T H(\theta - \theta_{old}) \tag{2.11}$$

where g is the derivative of the surrogate function w.r.t θ and H is the Hessian matrix of the second derivative of the KL divergence w.r.t the policy parameters. These approximations then result in a constrained approximation problem which the authors [22] solved by using the Lagrangian duality [23]. Resulting in the following

backtracking line search solution:

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_g^T \hat{H}_k \hat{x}_g}} \hat{g}_k, \tag{2.12}$$

where $\alpha \in (0, 1)$ and $j \in \{0, 1, 2, ...K\}$ satisfies the sample KL-divergence constraints. The full TRPO algorithm is given in Appendix A labelled as Algorithm 3.

2.8.2 Proximal Policy Optimisation (PPO)

The Proximal Policy Optimisation (PPO) algorithm [24] is an improvement over the vanilla policy gradient method. PPO, similar to TRPO [22] considers better data efficiency and reliability in taking data steps which classifies them under Trust Region Methods. The goal of PPO is to reduce the amount of change made to the policy at each training epoch in order to increase the training stability of the policy. This is achieved by measuring $r_t(\theta)$ a ratio that the probability of comparing the current policy to the old policy. It is used to measure the change in the policy network during the update step. The current policy, π_θ is represented by the probability distribution of actions given the current state, which is estimated by the policy network with the current set of parameters θ. The old policy, $\pi_{\theta_{old}}$ is represented by the probability distribution of actions given the current state, which is estimated by the policy network with the old set of parameters θ_{old}. $r_t(\theta)$ defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \tag{2.13}$$

The PPO algorithm uses the ratio $r_t(\theta)$ to ensure that the update of the policy network is not too large, by limiting the values of $r_t(\theta)$ between $1 - \epsilon$ and $1 + \epsilon$ where ϵ is a hyper-parameter called the clipping factor. This helps to prevent the algorithm from overreacting to the samples collected in a single episode, making the algorithm more stable and less likely to oscillate. This is then used to define the objective function as follows:

$$J^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)] \tag{2.14}$$

However, the original PPO algorithm uses the Monte Carlo method to estimate the advantage function, which can lead to high variance and bias in the estimation. The Generalized Advantage Estimation (GAE) [25] (an enhancement of the PPO algorithm) addresses this issue by introducing a parameter λ that controls the trade-off between bias and variance in the advantage function.

3 Literature Review

In this section, we first review the various sensors that can be used by UAVs that address the outlined problem ***O2*** and determine which sensor is best suited for the task. We will then examine various research pertaining to autonomous UAV obstacle avoidance to determine which reinforcement learning algorithms are best suited to solve objectives ***O3*** and ***O4***. Finally, we will take a look at the metrics used to evaluate performance for all our objectives.

3.1 Observation Data

LiDAR (Light Detection and Ranging) is the first sensor to be examined. A number of academics have looked into the application of LiDAR-based UAV obstacle avoidance systems. One such example is [26], this paper presented a navigation solution for unmanned aerial vehicles (UAVs) navigating inside a forest. The proposed solution was based on two main systems, called localization and motion control. The localization algorithm combined LiDAR-based odometry and Global Navigation Satellite System (GNSS) information using a robust adaptive sensor fusion algorithm (RAUKF). LiDAR-based odometry relied on the fact that trees were easily identified with a laser scan. Another example [27] described a system for Safe Landing Area Determination for small UAVs using a 2D LiDAR-based ground system. A servo motor was used to rotate the sensor so that the system could produce a point cloud of the 3D surveillance volume from 2D laser distance data. As soon as the UAV reached the landing volume, the system could recognize it and track its descent. To find and alert obstacles to the oncoming aircraft, a basic clustering-based obstacle detection algorithm was applied. Even in missions where the UAV was required to land on a moving platform, the system was able to deliver safe landing information. However, LiDAR comes with a limitation in how computationally expensive it is. LiDAR calculates tens of thousands of points per second into actions in order to provide an accurate 3D model of the surroundings. As a result, LiDAR demands a large amount of processing power. Moreover, there is a high cost in order to implement a set of LiDAR sensors, as is to be expected given the complexity of the software and the processing resources required.

Another common sensor that can be used is the traditional camera. Cameras can provide visual data that can be used for tasks such as tracking, and classification. Unfortunately, cameras have well-known limitations in terms of their ability to perceive depth and achieve low performance in dimly lit environments making them unsuitable for obstacle avoidance. Yet another sensor that can be used in autonomous UAVs is radar. In [28] authors use millimeter wave (MMW) radar for vehicle obstacle avoidance.

With their system, the distance between the object and the vehicle is computed for object detection and object tracking by observing the echoes created by radar signals. Also, the performance is assessed at various distances and weather conditions. Despite its appeal, similar to LiDAR radar-based systems are either too expensive or too heavy to be a payload for smaller robots like battery-powered UAVs [29].

In recent years, depth imagery has emerged as a promising sensor for autonomous UAVs. One such example is [30] where depth imagery was used in an RL task in order to teach drones how to act independently in a suburban neighborhood setting. Using Q-Network and Joint Neural Network (JNN), the drone was taught to detect obstacles and accomplish its task. The setting contained both fixed and moving obstacles as well as a geo-fence, a virtual wall. The drone's front camera could capture depth images, and scalar performance metrics such as elevation angle, angle to goal, and distance to geo-fence were employed to boost efficiency. The results of training and testing revealed encouraging success rates in avoiding obstacles and getting to the desired location. Another study was conducted [3] to train drones to deliver packages in a neighborhood environment with obstacles using only a stereo-vision front camera and a geo-fence to prevent leaving the area. The study found that the use of depth information obtained from the stereo-vision front camera contributed to the improved performance of the trained model in navigating the drone around obstacles during package delivery.

3.2 Reinforcement Learning Algorithms

In [2] a study was carried out for drones with obstacle avoidance capabilities. The study aimed to compare different Reinforcement Learning techniques and split their uses for both discrete actions and continuous actions.

For discrete actions, a comparison was done between DQN [10] and its variants, Double DQN [31], Dueling DQN [32], and Double Dueling DQN (D3QN). Along with these different algorithms, the actual input was also tested for the difference between RGB and Depth imagery. Three main environments were used to test these algorithms. A woodland that had a random number of trees being distributed in a path, a block world that had 3D Objects such as cubes, cones, and spheres distributed, and finally, the third environment built upon a block world to introduce a curved trajectory that utilizes yaw control. The action space utilized by these algorithms where split into five: up, down, forward, left and right. The result of this first experiment showed that DQN's varients gave better performance in each environment. The reason why this is the case is that each of these algorithms features improvements on the vanilla DQN.

The second experiment utilized continuous actions to improve upon the

unnatural movement caused by discrete actions. The algorithms used were from the family of policy gradient algorithms which consisted of Trust-Region Policy Optimization (TRPO) [22], Proximal Policy Optimization (PPO) [22], and Actor-Critic using the Kronecker-Factored Trust Region (ACKTR) [22] that all utilize actor-critic networks. Within this section of the study, it was highlighted that a U-Net-Segmentation model [33] is used. This type of model used to be done through supervised learning, but the labeling of all this data was not only time-consuming but was not perceived as optimum. Thus, the structure of this experiment utilized the policy gradient methods to solve the manual labeling of data. The outcome of this would be a vision-generated segmentation map as well as Actor-Critic RL for drone control. The result of this second experiment highlighted two important findings, firstly for the third environment, unlike discrete actions the smoother flight achieved by continuous actions would lead to a better completion rate for the final course. The second result showed that agents trained with PPO outperformed a fully tuned PID controller on almost every metric.

3.2.1 Dynamic environments

So far, research that has been mentioned has only dealt with environments that are static. In dynamic environments, where the state of the environment changes over time, RL algorithms face several challenges such as non-stationarity and partial observability. Therefore, further investigation is needed to evaluate the effectiveness of RL algorithms in dynamic environments.

In [34] a proposal was made using a Deep Reinforcement Learning approach for path planning of Unmanned Aerial Vehicles (UAVs) in dynamic environments with potential threats. The approach was based on the global situation information and employed the dueling double deep Q-networks (D3QN) algorithm to approximate the Q-values corresponding to all candidate actions. The proposed method's performance was demonstrated under both static and dynamic task settings using the STAGE Scenario software, considering the UAV survival probability under enemy radar detection and missile attack. The way dynamic enemy threats were handled by the D3QN network instead of passing just one singular RGB map, a set of situation map stacks are instead passed as observations to the algorithm. These maps are stacked sequentially along the channels of the image with the original dimensions going from 84 x 84 x 3 for one image to 84 x 84 x 12 for four stacked images.

This approach of stacking image frames along the channel was also used in the paper which proposed the Deep Q-Network (DQN) [10]. The authors performed image frame stacking by taking the current and previous three frames of the Atari game screen and concatenating them together along the channel dimension to create a

single input image for the deep neural network. By stacking the frames, the agent was able to observe how objects in the game screen moved and interacted over time, which provided it with important contextual information for making decisions and achieving human-level performance on several Atari games.

An alternative approach to this method has been developed [35], which by utilies recurrent neural networks in Deep Q-Networks to improve their ability to perceive and act in partially observed environments. By replacing the first fully-connected layer with a recurrent LSTM, the resulting Deep Recurrent Q-Network (DRQN) was able to successfully integrate information through time and perform well on standard Atari games with flickering game screens. DRQN's performance scaled with observability and was better able to adapt at evaluation time when the quality of observations changed compared to DQN. Recurrency proved to be a viable alternative to stacking a history of frames in the DQN's input layer, however, no systematic advantages were achieved between one method and the other.

3.3 Evaluation Metrics

In this section, we will delve into the metrics utilized in various literature to assess the performance of an RL drone obstacle avoidance system. Evaluation will be divided into two parts. Firstly, we will test the efficacy of sensors combined with a CNN independently, going through the standard metrics used with regard to a trained classifier's capabilities to predict collisions. Secondly, we will evaluate the system as a whole determining suitable metrics that will allow us to compare the various state-of-the-art algorithms in terms of their performance in completing the various objectives set out by this project.

3.3.1 CNN Classification

In literature, several standard metrics are commonly used to evaluate Convolutional Neural Networks (CNNs). One such metric is the confusion matrix, which summarizes the performance of a classification model by comparing predicted and actual classifications. It provides detailed information on True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). TP represents cases where the model correctly predicted the positive class, TN represents cases where the model correctly predicted the negative class, FP corresponds to cases where the model predicted the positive class but the actual class was negative (also known as a "Type I Error"), and FN corresponds to cases where the model predicted the negative class but the actual class was positive (also known as a "Type II Error"). These values are used to

calculate other commonly used crucial evaluation metrics, including accuracy, precision, recall, and F1-score:

Accuracy: Accuracy is the most commonly used metric to evaluate the performance of a CNN. It measures the proportion of correctly classified samples from the total number of samples. It is defined as the ratio of the number of correct predictions to the total number of predictions made by the model.

$$Accuracy = \frac{TP + TN}{(TP + TN + FP + FN)} \tag{3.1}$$

Precision: Precision is a metric that measures the proportion of correctly predicted positive samples from the total number of positive predictions. It is calculated as the ratio of true positives to true positives plus false positives. Precision is a useful metric in scenarios where false positives have a high cost.

$$Precision = \frac{TP}{(TP + FP)} \tag{3.2}$$

Recall: Recall is a metric that measures the proportion of correctly predicted positive samples from the total number of actual positive samples. It is calculated as the ratio of true positives to true positives plus false negatives. Recall is a useful metric in scenarios where false negatives have a high cost.

$$Recall = \frac{TP}{(TP + FN)} \tag{3.3}$$

F1 Score: F1 score is the harmonic mean of precision and recall. It combines both metrics to give a balanced measure of the model's performance. F1 score ranges from 0 to 1, where 1 represents the best possible performance.

$$F1\text{-}score = 2 \ * \ \frac{Precision * Recall}{(Precision + Recall)} \tag{3.4}$$

From these metrics, it is noted that when evaluating a drone's obstacle avoidance system, recall is one of the most important metrics considered. Recall is a measure of how many actual obstacles the drone properly identified and avoided. Avoiding collisions with objects is the goal of drone obstacle avoidance, which can be crucial for the safety of people, property, and the drone itself. Therefore, it is crucial to ensure that the drone's obstacle avoidance system can accurately detect and avoid obstacles in its path. High recall means that the drone can successfully identify and avoid most obstacles in its environment, which is essential for safe operation. If the recall is low, it means that the drone may miss obstacles or not detect them at all, leading to a potential collision. Therefore, a high recall rate is necessary for reliable obstacle avoidance.

3.3.2 Navigation and Obstacle Avoidance Metrics

While a number of standard metrics exist for measuring the performance carried out by a convolutional neural network, evaluating whether a particular reinforcement learning policy's behavior completes a certain task effectively requires metrics that are designed around the specific system. In terms of obstacle avoidance, papers such as [36][34][37] visualize the efficiency of the system in terms of the agent's crash and accomplishment rate. This is typically done by running the agent through a trained section a given number of times and calculating on average how many times the UAV has successfully reached the target destination. This measurement can also be extended if a given number of obstacle 'levels' are determined beforehand and evaluated on average how many of these 'levels' the UAV has successfully passed through (also known as 'leveling-up'). One such example is [37] where each particular 'level' corresponded to a wall with different opening positions the drone had to pass through and upon a successful passage i.e. every time the drone crosses an obstacle would then be considered a successful 'level-up'.

However, it is important to note how the specific environment is set up. Two approaches can be used to verify if the policy has achieved generalization, which is the ultimate objective: teaching the policy to adapt to new and unfamiliar situations. The first typical approach is running this test to determine the crash and accomplishment rate in an environment that the policy has never been seen before. Achieving a high completion rate in such an environment would ensure that the agent is learning the actual task of obstacle avoidance rather than the environments themselves. Another approach is to add randomness to the environment. At each episode, the positions of each obstacle as well as the starting position of the agent within a specific starting grid. This would then ensure that every episode of the environment is unseen.

Apart from crash and accomplishment rates, testing how quickly and to what degree an agent has learned a specific policy can also be represented by plotting the episodic returns the agent has managed to achieve versus a given number of episodes. This visualization can then show the growth rate as well as the peak reward obtained by each algorithm when conducting training. Relying on this visualization alone, however, may be problematic if an inappropriate reward function is not aligned with the goal. This may be the case when a particular agent may gain rewards from unintended behaviors, showing good improvements while in reality, the intended goals wouldn't be met. This may be the case where for example, the drone may gain a net-gain amount of rewards while performing loops through certain maneuvers which may out-way the rewards given by reaching the goal. Thus, it is important to couple this with the previously defined crash and accomplishment metric to ensure that such behaviors can be accounted for.

4 Methodology

This chapter presents the approach used to implement the objectives outlined in Chapter 1.4. Each objective will be addressed chronologically, with implementation details provided in separate sections. Objective (**O1**) focuses on integrating a drone environment simulator within a Reinforcement Learning framework. Objective (**O2**) involves identifying relevant sensor information and evaluating algorithms for accurate obstacle detection. Objective (**O3**) includes training four RL algorithms (two using discrete actions, two using continuous actions) to navigate environments with static obstacles. Objective (**O4**) entails training the top two RL algorithms on environments with dynamic obstacles.

4.1 O1: Constructing the RL framework

One of the major challenges in RL-based drone navigation is the risk of collisions with obstacles, which can lead to significant damage to the drone and endanger people and property in the vicinity. Although training an RL-based drone obstacle avoidance system in the real world is feasible, it can be a time-consuming, expensive, and a hazardous process. However, simulating the environment and drone behavior can offer a safer, more cost-effective, and more efficient alternative. By doing so, the agent can learn the specified task of obstacle avoidance in a controlled environment before being deployed in the real world. To achieve this goal, we are utilizing Air-Sim and UnReal Engine to simulate our environment. This section provides the details of how these were constructed.

4.1.1 Air-Sim and Unreal Engine

Unreal Engine is a game engine developed by Epic Games, used to create video games, virtual reality experiences, and other interactive content. It features a visual editor called the Unreal Editor, which enables developers to design environments using a wide range of tools, including 3D modeling, animation, physics simulation, and more. For the integration of the drone, Microsoft's AirSim [38] offers a multi-rotor model with realistic physics that can either be controlled by using an RC controller or programmatically. AirSim also offers methods for different levels of control. At the lowest level, the drones Yaw, Roll, and Pitch can be controlled. At a higher level, the drone can be directed to move in different directions. It can move horizontally along the x and y-axis, and vertically along the z-axis.

4.1.2 Static environments

A closed corridor was constructed as the environment the drone would navigate through. The corridor had a length of 61.3 m, width of 21.8 m, and height of 15 m. Obstacles were placed every 10.8 m for a total of four levels, starting at a position length of 18.5 m. The obstacles were columns with a diameter of 1m and were positioned both vertically and horizontally with a 3 m gap in between. Every time the drone crosses an obstacle in a single test, it is considered a 'level-up'. We consider the drone to take off from one end of the corridor and avoid all obstacles (passing through all the openings between the horizontal and vertical columns) as passing the test. The environment was designed to be randomized at each episode to achieve generalization. The middle column for both the vertical and horizontal columns was designated as the parent of the other sets of obstacles on the same level, which could be shifted uniformly between -3m and +3m programmatically from their original location every time the environment reset. Figure 4.1 shows a sample of these generated environments:

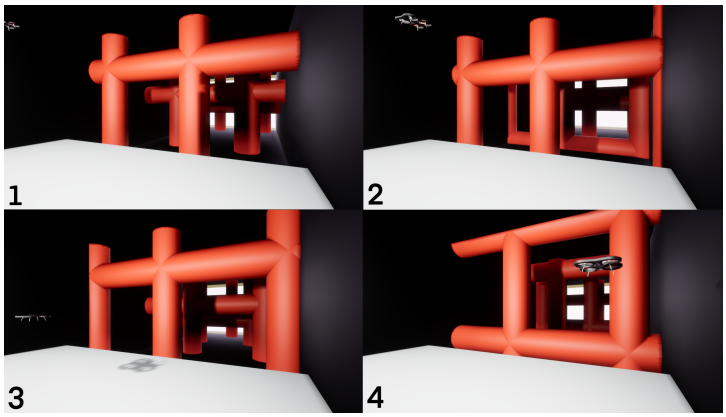


Figure 4.1 Four randomly generated static environments

4.1.3 Dynamic environments

For the dynamic environment, the dimensions of the closed corridor were kept the same, with the same approach of having obstacles being placed starting at a position length of 18.5 m with a subsequent obstacle being placed every 10.8 m for a total of four levels. Instead of columns, the obstacles used are cubes and squished cylinders of various sizes. For the dynamic obstacles to achieve movement, the Unreal engine's blueprint system was used to create points where these obstacles would be able to move from one point to another at a constant speed. The speed was set so that each obstacle would need to take 20 seconds in real time to arrive from one predetermined destination to another. A different approach was taken in this environment to test for generalization instead of having obstacles randomized. Two varied approaches were

taken in order to test the drone's generalization when dealing with randomization as well as dealing with a familiar training environment and in an entirely separate environment. This separate environment was specifically designed, where the obstacle order was randomly shuffled. Figure 4.2 shows these two environments:

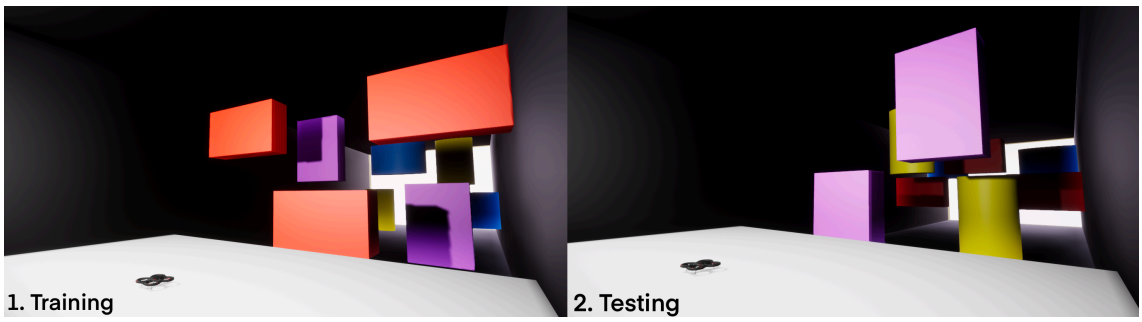


Figure 4.2 Original Training and Shuffled Testing Environments

4.1.4 Reinforcement Learning Framework

To implement the various Reinforcement Learning algorithms, a bridge between Python and Unreal Engine was established with the use of Air-Sim's Python API[1]. This enabled the ability to control the drone and receive feedback data in real-time. With this, each algorithm could be successfully implemented in Python while achieving the Reinforcement Learning-interaction loop between Python and Unreal Engine. The following are tools and packages utilized within Python that allowed for baseline implementations, performance monitoring, and metric visualization for our algorithmic frameworks. Additionally, the hardware and software specifications for training our Reinforcement Learning agents are listed in Appendix B as Table B.1.

4.1.5 Stable-Baselines3 and Gym

Stable-Baselines 3 [39] is a popular open-source library for reinforcement learning (RL) that provides a set of state-of-the-art RL algorithms implemented in Python, using the Torch library. This library is used due to its allowance for the possibility of quick prototyping and experimenting with various RL algorithms and configurations, without having to implement them from scratch, as well as offering a high-level API that makes it easy to train and evaluate RL agents on a wide range of tasks, including classic control problems, robotics, and games. Open AI's Gym package [40] is used as a wrapper around the AirSim API to feed the necessary information into Stable-Baselines3. Open AI's Gym enables the creation of a custom gym class that is used to define the state space, action space, reward function, and termination conditions for our task.

[1] https://microsoft.github.io/AirSim/api_docs/html/

4.1.6 Early Stopping and TensorBoard

Reinforcement learning uses early stopping to avoid overfitting by stopping the training process before the maximum number of epochs is reached. Stable-Baselines3 has an Early-Stopping callback that can be used to implement this functionality. The callback requires specifying a metric to monitor, a patience value, and whether to maximise or minimise the metric. In our case, the monitored metric is the average return in order to maximize the total reward obtained over time. By tracking the mean return, we can assess the overall performance of our system to measure its success by focusing on strategies that consistently generate higher average returns. TensorBoard is a web-based tool developed by TensorFlow, which can also be used with PyTorch that allows users to visualize and analyze their machine-learning models. Tensor-Board can be used to visualize various metrics such as the agent's reward over time, the loss, and the number of epochs.

4.2 O2: Sensor Information and Algorithm Configurations

As discussed in Section 3.1, Depth imagery uses active sensors such as stereo cameras or time-of-flight cameras to create 3D point clouds that represent the depth information of the environment. In conjunction with Convolutional Neural Networks (CNNs), depth information can be coupled to improve the accuracy of object detection and recognition tasks. Depth information provides accurate data even in low light conditions compared to the traditional camera, and its cost in terms of computation and the overall price is relatively low compared to both LiDAR and Radar, making it the preferred sensor for autonomous UAVs that require accurate depth perception for the task of obstacle avoidance. Moreover, as detailed in Section 2.4, a CNN architecture can be used to process this imagery to lower the computational expense as well as avoid over-fitting that occurs when passing imagery straight through an ANN. In order to solely focus on the classifier's capability to predict obstacles the problem was broken down and changed to a classification problem which was trained via Supervised Learning. Breaking down the whole problem into individual parts helps better verify if, for example, the retrieved sensor data is sufficient for Obstacle Avoidance.

4.2.1 Data Collection

In order to train a classification model to predict collisions, obtaining sufficient training data is crucial for evaluating the classifier. In order to collect the training imagery data, we conducted multiple episodes of running the UAV forward through our environments. This necessitated the implementation of Section 4.1. For this scenario,

an episode is triggered when a collision flag is returned by the AirSim API which is saved at every time step. This flag was then used to label the data into a binary classification of Collision (1) and Non-Collision (0). An episode would terminate once a collision is met or the goal was reached and would reset the drone's position back at the start of the environment whilst randomizing the environment as well as the drone's starting position.

Since we want the drone to take evasive action before it is too close to the obstacle, using just the last frame before collision is not sufficient. We want our data to indicate instead, a collision a number of time steps before the actual collision occurs. To achieve this, a stack was maintained by adding an image taken by the UAV at every time step. These images would be processed and adjusted to a size of 150x150 before being pushed onto the stack to allow for the CNN architectures to process these images. Once an episode ends with a collision, the last k images are popped and labeled as collisions, while the rest are popped and labeled as non-collision. After testing, k was set to three since training our classifier with this set parameter indicated it was enough time steps taken before considering collision. Since our drone navigates differently in static and dynamic environments, we collected two sets of labeled data. In static environments, single imagery at each step was sufficient, while in dynamic environments, stacked frames were necessary for the agent to predict object movement.

A total of 500 episodes were conducted in both environments, resulting in a dataset of 10,392 images from the static environment (1,445 collisions and 8,947 non-collision) and 11,949 images from the dynamic environment (1,371 collisions and 10,578 non-collision). This data set was then split into three data sets, Training, Validation, and Test sets. This was done in order to evaluate the performance and the generalization ability of our classification model. The validation set is used to fine-tune the model's hyperparameters and prevent overfitting. The test set is kept separate from training and is tested at the end to fully evaluate the performance achieved on data with an unbiased estimate of how well the model performs on new, unseen data. This split was taken as 80% Training, 10% Testing, and 10% Validation.

4.2.2 CNN Architecture

Once our datasets were constructed comprising of Collision and Non-Collision Images, the model could be trained via a Convolutional Neural Network. This kind of architecture was implemented with the help of the TorchVision library. TorchVision is a computer vision library built on top of PyTorch, a popular open-source machine learning framework. TorchVision provides a number of features and utilities for building and training computer vision models, as well as for data preprocessing and

augmentation. In total three different computer vision models were trained on various parameter complexity. The goal was to aim for a lower parameter CNN, for due the following reasons:

1. **Low processing power:** Drones often have less memory and processing capability than traditional computers. With fewer layers, a CNN with lower parameters would run more efficiently. This implies that a drone with constrained computing power can implement a lower parameter CNN.
2. **Faster processing:** A lower parameter CNN would also have faster processing time compared to a larger network. This is due to the fact that there are fewer layers overall would result in fewer computations. This implies that in a drone application, the CNN can process images or data more quickly, enabling the drone to respond to environmental changes more hastily.
3. **Reduced memory usage:** A lower parameter CNN requires less memory to store its parameters compared to a larger network. This is crucial given that drones frequently have a limited amount of memory. A lower parameter CNN can be used to save memory, allowing the drone to retain more data or perform additional jobs at the same time as the CNN.

The tables showcasing each of the tested architectures are presented in Appendix C, with the final table, Table C.4 highlighting the different sizes between each architecture in terms of parameters and total size (MB). From the tested architecture's, Table 4.1 showcases the CNN architecture that was ultimately used for our agents.

Table 4.1 CNN Architecture used by Agents

Layer	Input Dims	Transform	Kernel Size	Stride	Activation	Output Dims
Input	150 x 150 x 1	Conv2D	8	2	ReLU	36 x 36 x 32
H1	35 x 36 x 32	Conv2D	4	2	ReLU	17 x 17 x 64
H2	17 x 17 x 64	Conv2D	3	2	ReLU	15 x 15 x 64
H3	15 x 15 x 64	Flatten	N/A	N/A	None	14400
H4	14400	Linear	N/A	N/A	ReLU	256
Output	256	Linear	N/A	N/A	None	2

4.2.3 Algorithm Configurations

The Deep Q-Network (DQN) and Double Deep Q-Network (DDQN) algorithms were used for training on discrete actions, while the Proximal Policy Optimization (PPO) and

Trust Region Policy Optimization (TRPO) algorithms were employed for continuous actions. These algorithms are widely used and have shown good performance in Drone Navigation, as discussed in Section 4.1.4. PPO and TRPO are considered state-of-the-art actor-critic algorithms that employ a trust regional approach to policy optimization. It is worth noting that the performance of RL algorithms can be highly sensitive to their hyperparameters, thus proper configuration is important to optimize the performance of these algorithms and improve their learning efficiency. Hyper-Parameters were tuned slowly as different observation spaces, and rewards were considered.

Each algorithm ran for $50,000$ time steps, with early stopping at $40,000$ if there was no improvement with the mean return metric (which is calculated at a frequency of 500 time steps) after 10 evaluations. A maximum of $50,000$ time steps was sufficient for each algorithm since each managed to successfully converge within this time frame. The hyper-parameter setting for the Deep Q-Network (DQN) and Double Deep Q-Network (DDQN) included an exploration fraction of 0.6ϵ this decreased the exploration rate steadily from 0.9ϵ to 0.01ϵ by $30,500$ time-steps to encourage more exploration before taking exploitation on there acquired knowledge. A learning rate of 0.0001α was chosen for the agent to make smaller and more precise updates to their weights which facilitated better convergence and more accurate results. Finally, a batch size of 64 and a buffer size of $50,000$ were chosen for balanced training speed, generalization, and memory constraints. With regards to PPO and TRPO, the best hyperparameters were found when a batch size of 128 was used, and the learning rates were kept similar to the previous two algorithms at 0.0001ϵ. PPO has additional hyper-parameters that were tuned which are the maximum value for gradient clipping (which was set to 0.5) which specifies the maximum magnitude of the gradients during the optimization step, and a clip range (set at 0.10) which refers to the maximum allowed deviation between the old and updated policy during each iteration.

4.3 O3, O4: Navigation in Static and Dynamic Envrionments

For our final two objectives, we train four RL algorithms (two using discrete actions, two using continuous actions) in a static environment and the best-performing two in dynamic environments. Time constraints necessitated this approach, as each algorithm required extensive tuning and substantial training to attain optimal performance. During the training process, we made adaptations to the action space, observation space, and rewards based on whether the agent operated in a static or dynamic environment and whether the action space was discrete or continuous. Thus in this section, the action spaces, observations spaces, and rewards shall be outlined.

4.3.1 Actions

Discrete actions involve a limited set of choices, while continuous actions offer an infinite range of possibilities. In the context of drone obstacle avoidance, discrete actions include specific movements like ”move left,” ”move right,” ”move forward,” and ”move backward.” On the other hand, continuous actions pertain to controlling the drone’s speed and direction. For encoding discrete actions, the agent is given six options: move right, move down, move back, move left, move up, and hover. These actions correspond to movements along the x-axis, y-axis, and z-axis. When executing a discrete action, the drone’s speed is set to 1 m/s along the chosen axis and 0 m/s for the remaining axes during hovering. Regarding continuous actions, the agent can select velocities within a floating-point range of -1.0 m/s to 1.0 m/s for each axis. This allows the drone to move along the x-axis, y-axis, and z-axis with varying degrees of velocity or even stay stationary along a specific axis by choosing velocities within this range.

4.3.2 Observations

Regarding the observation space, in the case of discrete actions, an additional observation is included to retain a record of prior actions taken. During training, it was observed that the agent would occasionally become trapped in an action loop directly in front of an obstacle, resulting in poor performance. This looping behavior was caused by the agent’s insufficient prior event history. To address this issue, a queue containing the 10 most recent actions was implemented. This enabled the agent to develop a more comprehensive understanding of previous events, thereby preventing it from becoming trapped in the aforementioned loop. On the other hand, when training using continuous actions, since the agent wasn’t limited to movement along the x, y, and z axis this problem was not encountered.

Another difference in observation spaces occurred when dealing with static and dynamic environments. This is highlighted in Section 3.2.1, whereas for static environments one frame observation per step is sufficient, on the other hand, a stack of observation frames along the channel frames is required for the agent to be able to observe how obstacles are moving. These frames are collected between each step, starting at the beginning before an action is taken, in the middle of the action being taken, and a final frame when the action concludes. The agent received supplementary observations that are used across various action and obstacle types, including the agent’s velocity, its relative distance to the goal, and its prior relative distance to the goal. Relative Distance is calculated using Euclidean distance for each axis as shown before:

$$d(x, y, z) = \sqrt{(x_{agent} - x_{goal})^2}, \sqrt{(y_{agent} - y_{goal})^2}, \sqrt{(z_{agent} - z_{goal})^2}, \tag{4.1}$$

To summarize Table 4.2 showcases the different observations taken.

Table 4.2 Agent Observations

Observation Spaces		
Static Environment (Discrete Actions)	Static Environment (Continuous Actions)	Dynamic Environment (Discrete Actions)
Depth Frame: 150 x 150 x 1	Depth Frame: 150 x 150 x 1	Depth Frame: 150 x 150 x 3
Agent Velocity	Agent Velocity	Agent Velocity
Relative Distance	Relative Distance	Relative Distance
Previous Relative Distance	Previous Relative Distance	Previous Relative Distance
Action History		Action History

When these observations are fed into their respective Reinforcement Learning Algorithms, as mentioned in Section 2.4 it is important that the observation imagery is passed through a convolutional neural network. This is the CNN architecture presented in Section 4.2. The rest of the observations are passed through a linear layer of size 16, followed by a ReLu activation function. Each observation is then concatenated together before being fed to the feature extractor class.

4.3.3 Rewards

In this project, we experimented with different reward functions to identify the optimal one for the agent to achieve its objective successfully. We iteratively modified the reward function until we arrived at the best set of rewards that enabled the agent to perform the task successfully.

While the structure of rewards in the environments remained mostly consistent, the rewards varied slightly when changing actions spaces from discrete to continuous. The primary objective of the agent is to maximise its cumulative rewards over time. To achieve this objective, negative rewards were employed as a penalty to discourage undesirable behavior, while positive rewards were used to reinforce desired behavior. The largest rewards and penalties were given either when the agent collides (-100 penalty) or when the agent reaches the desired objective (100 reward). An additional -100 penalty is given if the agent fails to reach the objective after a substantial amount of time. Once any of these conditions are met the episode ends, resetting the whole

environment.

The agent receives a interim reward of 20 every time it 'levels-up'. Level-ups are achieved every time an agent successfully crosses an obstacle level as defined in Section 4.1.2. This reward encourages the agent to avoid obstacles and move closer to the goals. Furthermore, euclidean distance is also used to encourage the agent to move towards the goal on a more frequent basis, giving minor penalties or minor rewards at each step depending on whether the agent is moving towards or away as follows: This reward is calculated by the following:

$$reward = (d_{previous} - d_{current}) - d_{pos} \tag{4.2}$$

where $d_{previous}$ is the euclidean distance of the agent's previous position from the target goal, $d_{current}$ is the distance of the agent's current position from the target goal, d_{pos} is the distance between the agent's previous position and current position. The only difference in rewards for each action space is a penalization if the agent hovers for too long to discourage staying idle. For discrete actions, hovering is designated as a specific action, which incurs a penalty of -0.5 when selected. On the other hand, for continuous actions, the agent hovers if it selects a specific range of values for each axis, this range is specified between a speed of -0.2 m/s and 0.2 m/s for all three axes incurring a penalty of -1.5. Table 4.3 presents a summary of these reward values.

Table 4.3 Agent Rewards

State Type	Reward Value
Collision	-100
Level-Up	$+20$
Goal Reached	$+100$
Time Limit	-100
Movement	$(d_{previous} - d_{current}) - d_{pos}$
Hovering (Discrete)	-0.5
Hovering (Continuous)	$-1.5\ if\ speed\ is\ between\ -0.2\ m/s\ \wedge\ 0.2\ m/s\ for\ all\ three\ axis$

5 Evaluation

In this chapter, we will use the evaluation metrics described in Section 3.3 to measure performance and compare the results of the objectives set out for this project. Our first step is to utilize confusion matrices to determine the ability of our trained classifiers to detect collisions accurately. By utilizing confusion matrices, we can effectively evaluate the performance of different CNN architectures to select an architecture that can achieve a high classification rate accurately while minimizing the utilization of processing power.

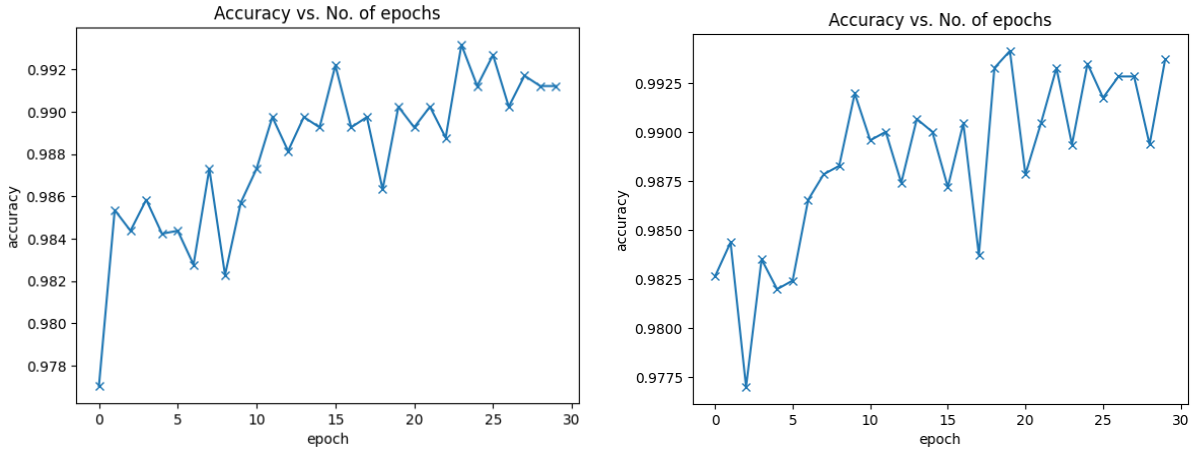
Next, we will compare the performance of four agents (two discrete and two continuous) that have been trained to navigate environments containing static obstacles. We will evaluate these agents based on the number of successful runs without collisions and examine the differences encountered when training them using different kinematics. The use of fully randomized environments ensures that a high rate of successful runs validates their ability to generalize effectively. Lastly, we will evaluate the performance of the two top-performing agents in navigating environments with dynamic obstacles. Their evaluation will be based on the number of successful runs without collisions. To assess their ability to generalize, we will validate their high success rate not only in the environment they were trained in but also in an unseen environment.

5.1 Obstacle Detection Evaluation

In our initial experiment, we evaluated the classification accuracy of our trained classifiers in detecting collisions. We collected training data by flying a UAV through our environments multiple times from different starting positions. This data was used to train CNN models of varying complexity. We assessed the effectiveness of each model in classifying collisions. Our goal is to enable our agents to navigate through both static and dynamic environments. For this purpose, we created two datasets. The first dataset consisted of 150 x 150 x 1 images, suitable for accurately classifying predictions in static environments. The second dataset included stacked 150 x 150 x 3 images, necessary to address the non-stationarity challenge in dynamic environments.

In this section, we present the performance of the CNN architectures utilized in our system. For the metrics of the other architectures, please refer to Appendix D. All the architectures were trained effectively and achieved high accuracy. Even the most complex architecture, which had 86 million parameters and a size of 401 MB, showed only a negligible drop in performance when compared to the least complex architecture with 4 million parameters and a size of 15 MB. This was true for both

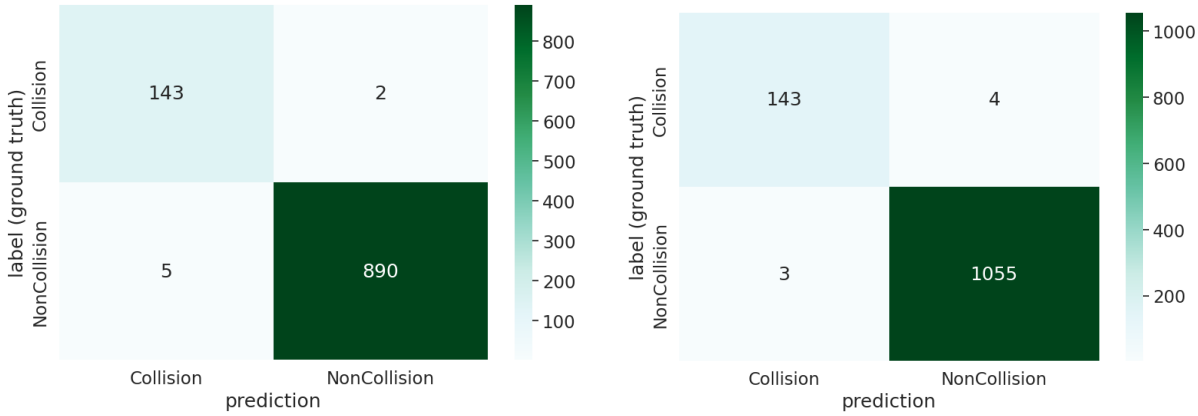
single and stacked imagery. Therefore, we were able to integrate the architecture with the least computational expense while still maintaining high classification accuracy. In Figure 5.1a and Figure 5.1b the accuracy achieved on this architecture over the validation set for thirty epochs is showcased. The CNN trained on single imagery achieved at best 99.17% accuracy over our validation set, while the same architecture trained on stacked imagery achieved an accuracy of 99.37%



(a) Static Dataset Accuracy (b) Dynamic Dataset Accuracy

Figure 5.1 Training Accuracy of the Third CNN Architecture on Dynamic and Static Environments

After training, the trained models were run on an unseen test set, and with these predictions, the confusion matrices were formed. From these matrices, using the equations established in Section 3.3.1 we can then infer our model's Accuracy, Precision, Recall, and F1-Score. These formed matrices for the architecture used are illustrated in Figures 5.2a and 5.2b.



(a) Static Confusion Matrix (b) Dynamic Confusion Matrix

Figure 5.2 Confusion matrix of the Third CNN Architecture on Dynamic and Static Environments

The results in Figures 5.2a and 5.2b demonstrate the effectiveness of our CNN architecture in obstacle detection for both static and dynamic environments. In static environments, we achieved an adequate 99.3% accuracy, 96.6% precision, 98.6% recall, and 97.6% F1-score. Meanwhile, in dynamic environments, we achieved an accuracy of 99.4%, with 97.9% precision, 97.2% recall, and 97.5% F1-score. As discussed in Section 3.3.1, recall is the most important metric for our obstacle detection system. With a recall of 98.6% in static environments and 97.2% in dynamic environments, we have demonstrated the high performance of our depth sensor-equipped drone in detecting obstacles. These results indicate that our CNN architecture, combined with a depth sensor, is well-suited for obstacle detection in various environments.

5.2 Navigation and Obstacle Avoidance Evaluation

In our second experiment, we trained two continuous and two discrete algorithms in a Static Environment. Each algorithm ran for 50k time steps, with early stopping at 40k time steps. In terms of real-time performance, the DDQN algorithm achieved the fastest convergence in 10 hours. The DQN algorithm required 11 hours and 30 minutes to converge, followed by the TRPO algorithm requiring 11 hours and 40 minutes. Lastly, the PPO algorithm took 12 hours to reach convergence. However, if instead, the number of time steps is considered, the PPO algorithm demonstrated the quickest convergence, requiring only 42k time steps. TRPO followed with 45,500 time steps, while DDQN took 46,500 time steps. Interestingly, the DQN algorithm reached the maximum allotted time steps of 50k, indicating that early stopping was not triggered. This difference between real-time convergence and time-step convergence is due to the mean episode length between each episode. Continuous algorithms took longer to reach their destination, throttling forward at a speed of approximately 0.5m/s which was deemed safer by the policy, while discrete algorithms had to stick at a forward speed of 1m/s. Figure 5.3 illustrates the disparity in episode length between the four algorithms.

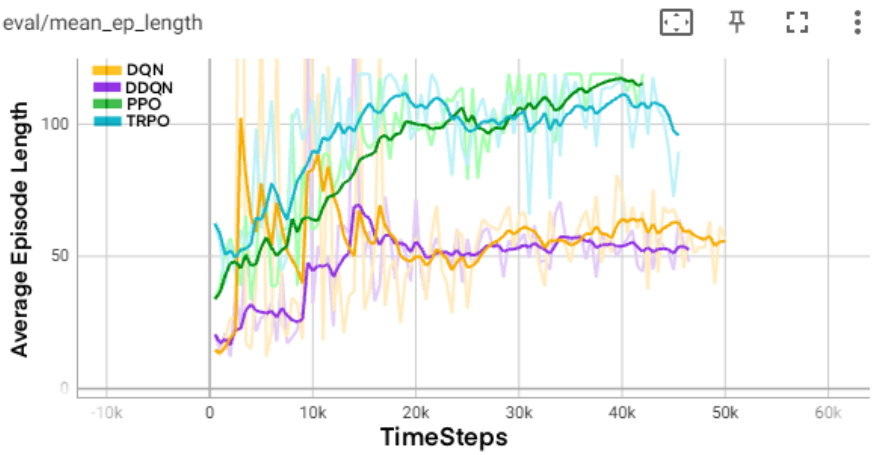


Figure 5.3 Mean Episode Length between the two continuous and two discrete agents

Regarding rewards, a direct comparison between Discrete and Continuous actions is challenging due to differences in their reward structures. Nevertheless, we can still compare the algorithms within each set. Among the discrete algorithms (Figure 5.4), DDQN exhibited a slight improvement over DQN, with a higher average reward over time. However, it is worth noting that DQN achieved a slightly higher maximum reward of 186.2 compared to DDQN's 183.2. For the continuous algorithms, (Figure 5.5), both showed a steady increase in rewards during training, but at around 20k time steps, their policies experienced a temporary drop in performance. This could be due to factors such as policy updates causing divergence from the optimal solution or a trade-off between exploration and exploitation, where the agent starts effectively exploring the learned policy over time. In the first half, TRPO outperformed PPO in terms of average reward, but PPO recovered with a steep increase in average reward and achieved a higher maximum reward of 152.4, while TRPO showed a gentler increase in average reward with a slightly lower average reward of 151.1.

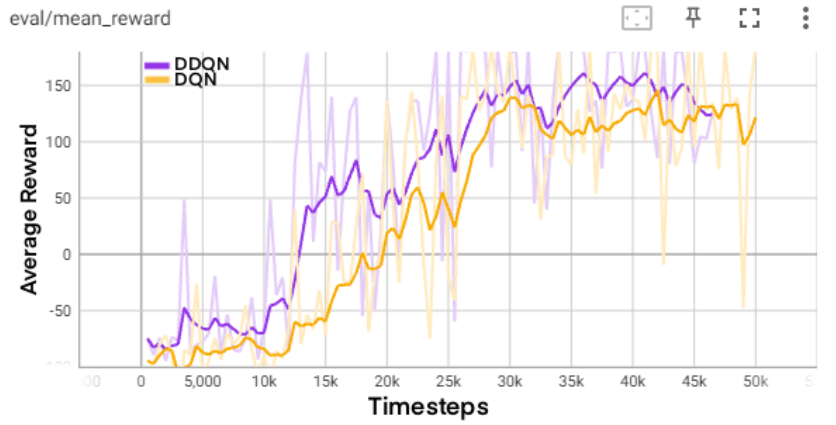


Figure 5.4 Mean Reward vs Time steps for DQN & DDQN in a Static Environment

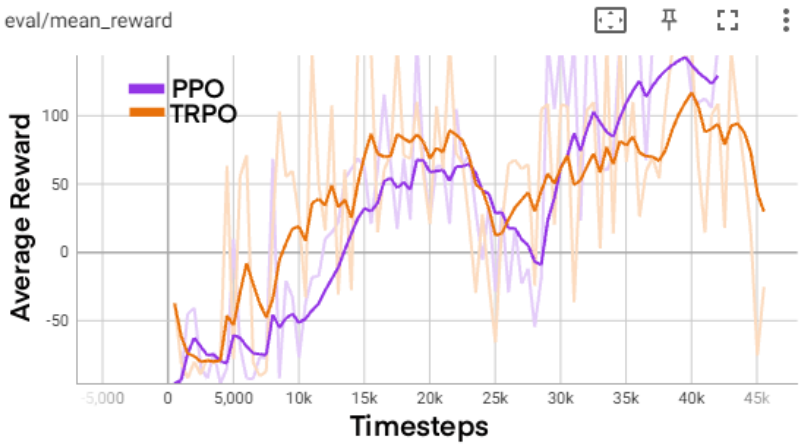


Figure 5.5 Mean Reward vs Time steps for PPO & TRPO in a Static Environment

After these models were fully trained, we then proceeded to evaluate their performance by running them through the randomized environment a hundred times. Then, the averages were computed by how many times the agents could surpass the presented obstacles and reach the goal while maintaining generalizability. As explained in Section 4.1.2, the environment used in this study was fully randomized, resulting in an unseen environment for each run. The obstacles were divided into four levels, corresponding to the four levels of obstacles encountered by the agent. If the agent successfully passed the final obstacle, it was considered to have reached the goal.

For the discrete algorithms, the results revealed that the DDQN algorithm achieved the highest completion score of 93%, indicating the best performance in obstacle avoidance within a generalized static environment. In comparison, the DQN algorithm achieved an average score of 86%, which was 7% lower than the DDQN algorithm. For a detailed breakdown of the results, please refer to Table 5.1.

Table 5.1 Testing discrete algorithms' avoidance rate on static environments

Average Obstacle Avoidance Rate (Static Environment)		
Obstacle Sequence	Double Deep Q-Learning	Deep Q-Learning
Level: 1	97%	95%
Level: 2	94%	93%
Level: 3	93%	91%
Goal Reached	93%	86%

On the other hand, for the continuous algorithms, the results revealed that the PPO algorithm achieved a completion score of 79%, whilst, in comparison, the TRPO algorithm achieved an average completion score of 71%, 8% lower than the PPO algorithm. Table 5.2 showcases these results. A drop in 14% completion score is notable between the best-performing discrete algorithm and best-performing

continuous algorithms. During navigation, the agents operating in the continuous space encountered difficulties when traversing tighter areas. Instead of moving forward steadily, these agents tended to oscillate gently between the y-axis and z-axis. As a result, they occasionally made contact with obstacles, preventing them from reaching the goal. This observation aligns with the average passing rates, where the PPO agent successfully passed through the first level 97% of the time, but only accomplished reaching the goal in the final level 79% of the time. In contrast, the discrete algorithms, with their fixed action space, demonstrated superior performance in maneuvering through narrower gaps. The DDQN agent, for instance, experienced a mere 4% decrease in success rate between passing through the first level and reaching the goal in the final level.

Table 5.2 Testing continuous algorithms' avoidance rate on static environments

Average Obstacle Avoidance Rate (Static Environment)		
Obstacle Sequence	Proximal Policy Optimization	Trust Region Policy Optimization
Level: 1	97%	91%
Level: 2	92%	86%
Level: 3	89%	77%
Goal Reached	79%	71%

For our final experiment, we trained the two best-performing algorithms, which are the DQN and DDQN algorithms, and trained them in a Dynamic Environment. We yet again ran these algorithms for 50k time steps, with early stopping at 40k time steps. Both algorithms didn't trigger early stopping and trained for a full 50k timesteps both taking 11 hours to complete training. In terms of episode length, both algorithms took roughly the same amount of time to reach the goal. In terms of reward, DDQN again exhibited a slight improvement over DQN, with a higher average reward over time. However, it is again worth noting that DQN achieved a higher maximum reward of 188.5 compared to DDQN's 156.2. This is shown in Figure 5.6.

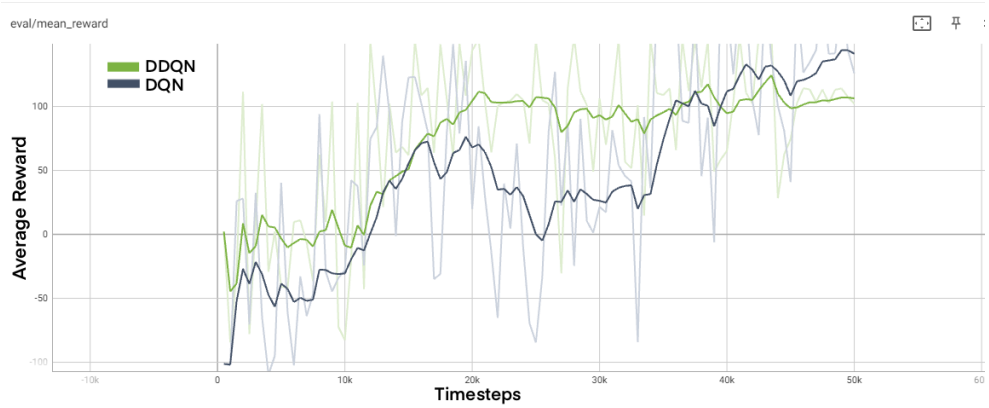


Figure 5.6 Mean Reward vs Time steps for DQN and DDQN in a dynamic environment

After the models were fully trained, we measured their average completion rate by running the agent's policies through the same environment used during training. Next, we tested the models on a shuffled version of the environment, where the order of obstacles was randomized. This was done in order to verify the agent's ability to generalize in an environment it hasn't seen before. Similar to the previous experiment, we performed 100 run-throughs for each environment in this experiment. The obstacles were again divided into four levels, with the successful passage of the final obstacle considered as reaching the goal. We computed the average number of times each obstacle was overcome, which is shown in Table 5.3.

Table 5.3 Testing discrete algorithms' avoidance rate on dynamic environments

Average Obstacle Avoidance Rate (Dynamic Environments)				
	Double Deep Q-Network		Deep Q-Netwrok	
Obstacle Sequence	Training Environment	Testing Environment	Training Environment	Testing Environment
Level: 1	99%	94%	95%	93%
Level: 2	93%	89%	91%	89%
Level: 3	87%	86%	88%	84%
Goal Reached	85%	84%	85%	81%
Average Goal Reached	84.5%		83%	

By comparing the results presented in Table 5.1 and Table 5.3, we can observe that the best-performing algorithm in a static environment achieved a completion rate of 93%, while the best-performing algorithm in a dynamic environment achieved a completion rate of 84.5%, which represents a drop of 8.5%. Furthermore, when comparing the performance of the two algorithms in the dynamic environment, we can see that DDQN achieved a completion rate of 84.5%, while DQN achieved a completion rate of 83%, showing only a slight advantage of 1.5% for DDQN. Finally, when evaluating the model's performance in a shuffled environment, where the order of obstacles was randomized, both algorithms demonstrated only a marginal decrease in completion rate. Specifically, the DQN algorithm showed a completion rate drop of 4%, while the DDQN algorithm showed a negligible drop of 1%. These results indicate the models' ability to generalize well to previously unseen environments.

6 Conclusion

The aim of this thesis was to develop a reinforcement learning-based solution for unmanned aerial vehicles (UAVs) that enable drones to safely navigate through unmapped cluttered environments, including obstacles that are either static or moving. To reach this aim. the following objectives were set and achieved.

The first objective, **O1**, involved constructing the Reinforcement Learning Framework to allow the drone to be programmatically controlled. To achieve this, we utilized Air-Sim and Unreal Engine to simulate our environment. We bridged this environment with Python using OpenAI's Gym as a wrapper around the AirSim API, which was implemented through the Stable-Baseline3 model. The Unreal Engine was employed to create a simulation environment that encompassed both static and dynamic components. In the static environment, the positions of the drone and obstacles were randomized, whereas, in the dynamic environment, the drone's positioning was also randomized, accompanied by a separate dynamic environment with shuffled obstacles. This approach ensured a comprehensive evaluation of the agents' generalization capabilities across both familiar and unfamiliar scenarios.

Our second objective, **O2** involved identifying the best sensor for obstacle detection. After reviewing relevant literature, we determined that the depth sensor was most suitable for obstacle detection due to its accurate depth information and affordability. In order to solely focus on whether the observation data was sufficient for the agents to predict obstacles collisions, the problem was broken down and changed to a classification problem with classifiers being trained via Supervised Learning. We trained these classifiers using three Convolutional Neural Network (CNN) architectures of various sizes. These architectures were trained using labeled imagery, which had been collected by the agents using the depth sensor from our environments. Among the three tested architectures, the smallest one was chosen for its lower computational requirements and high recall rates, with the trained classification model achieving a recall of 98.2% and 97.2% using observation data from a static environment and a dynamic environment, respectively.

In our third objective, **O3**, we employed four agents in an environment with static obstacles. The first two algorithms, the Double Deep Q-Network and the Deep Q-Network utilized a discrete action space, while the last two algorithms, the Proximal Policy Optimization and the Trust Region Policy Optimization utilized a continuous action space. After these agents were trained, each agent was run through the randomized environment a hundred times and the average number of times each obstacle was overcome and the goal had been reached was computed. From the four trained algorithms, the DDQN algorithm achieved the best performance achieving a

completion rate of 93%, followed by DQN with 86%, PPO with 79%, and lastly, TRPO achieving the lowest completion rate of 71%.

Finally, for our fourth objective, **O4**, we employed the two best-performing algorithms, which were the Deep Q-Network and the Double Deep Q-Network, in an environment with dynamic obstacles. Both algorithms had their observations adapted from single imagery to stacked image frames to capture obstacle movement information. After training, these models had their average completion rate measured by running them through two environments. The first environment was the environment the agents were trained in, and the second environment was a shuffled version of the training environment, that had the order of obstacles randomized. On average, between the training and testing environment, the DDQN yet again achieved the best performance with a completion rate of 84.5%, with the DQN algorithm being close behind with a slight decrease of 1.5% at 83% completion.

6.1 Future Work

In this project, the primary focus is on the higher level of control provided by Air-Sim. It has been shown that this level of movement can be used for drone navigation through complex environments, where the agent has discrete and continuous action spaces to choose from within the x, y, and z-axis. Moving forward, the problem can be made significantly more complex by utilizing a lower level of control which is the drone's Yaw, Roll, Pitch, and Throttle. This would require the action space to consider a negative to the positive range for Yaw, Roll, and Pitch in radians and a throttle range between 0.0 and 1.0. This would include a new dimension of control in terms of rotation. With rotation, environments can be made reasonably more complex and can include curved trajectories. This new level of control would need to be accompanied by adding rewards and observations pertaining to the agent's orientation and body frame relative to the goal, as described in [41]. Additionally, in this project, we limited the evaluation of our policies to the AirSim Simulation. In future work, this can be extended with real-life scenarios which have been replicated and tested within the simulation. For instance [42] proposed a high-level controller that can direct a UAV to track and pursue another UAV in a real-life pursuit-evasion scenario. This approach could be leveraged to test the effectiveness of our policies in a more realistic and challenging setting.

References

[1] W. Koch, R. Mancuso, R. West, and A. Bestavros, "Reinforcement Learning for UAV Attitude Control," *ACM Trans. Cyber-Phys. Syst.*, vol. 3, no. 2, Feb. 2019, ISSN: 2378-962X. DOI: 10.1145/3301273. [Online]. Available: https://doi.org/10.1145/3301273.

[2] S.-Y. Shin, Y.-W. Kang, and Y.-G. Kim, "Obstacle Avoidance Drone by Deep Reinforcement Learning and Its Racing with Human Pilot," *Applied Sciences*, vol. 9, no. 24, 2019, ISSN: 2076-3417. DOI: 10.3390/app9245571. [Online]. Available: https://www.mdpi.com/2076-3417/9/24/5571.

[3] G. Muñoz, C. Barrado, E. Çetin, and E. Salami, "Deep in Reinforcement Learning for Drone Delivery," *Drones*, vol. 3, no. 3, 2019, ISSN: 2504-446X. DOI: 10.3390/drones3030072. [Online]. Available: https://www.mdpi.com/2504-446X/3/3/72.

[4] W. Cao, X. Huang, and F. Shu, "Unmanned Rescue Vehicle Navigation with Fused DQN Algorithm," in *Proceedings of the 2019 International Conference on Robotics, Intelligent Control and Artificial Intelligence*, ser. RICAI 2019, New York, NY, USA: Association for Computing Machinery, 2019, pp. 556–561, ISBN: 9781450372985. DOI: 10.1145/3366194.3366293. [Online]. Available: https://doi.org/10.1145/3366194.3366293.

[5] J. E. Scott and C. H. Scott, "Drone Delivery Models for Healthcare," in *Hawaii International Conference on System Sciences*, 2017.

[6] A. T. Azar *et al.*, "Drone deep reinforcement learning: A review," *Electronics*, vol. 10, no. 9, p. 999, 2021.

[7] C. Zammit and E.-J. Van Kampen, "Comparison between A* and RRT Algorithms for UAV Path Planning," Proceedings of the 2018 AIAA Guidance, Navigation, and Control Conference, Nov. 2018. DOI: 10.2514/6.2018-1846.

[8] S. Karaman and E. Frazzoli, "Sampling-based Algorithms for Optimal Motion Planning," *CoRR*, vol. abs/1105.1186, 2011. [Online]. Available: http://arxiv.org/abs/1105.1186.

[9] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[10] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[11] S. Haykin, "Neural Networks and Learning Machines," Pearson Education India, 2010.

[12] T. M. Mitchell and T. M. Mitchell, *Machine learning*. McGraw-hill New York, 1997, vol. 1.

[13] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

[14] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Neural Information Processing Systems*, vol. 25, Nov. 2012. DOI: 10.1145/3065386.

[15] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[16] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[17] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018, ISBN: 0262039249.

[18] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[19] H. Hasselt, "Double Q-learning," in *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23, Curran Associates, Inc., 2010. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

[20] R. Stekolshchik, *Some approaches used to overcome overestimation in Deep Reinforcement Learning algorithms*, 2022.

[21] D. Ghosh, M. C. Machado, and N. Le Roux, "An operator view of policy gradient methods," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 3397–3406. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/22eda830d1051274a2581d6466c06e6c-Paper.pdf.

[22] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, 2015, pp. 1889–1897.

[23] F. Giannessi, "On the theory of Lagrangian duality," *Optimization letters*, vol. 1, no. 1, pp. 9–20, 2007.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal Policy Optimization Algorithms*, 2017. DOI: 10.48550/ARXIV.1707.06347. [Online]. Available: https://arxiv.org/abs/1707.06347.

[25] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[26] E. Aldao, L. M. González-de Santos, and H. González-Jorge, "LiDAR Based Detect and Avoid System for UAV Navigation in UAM Corridors," *Drones*, vol. 6, no. 8, p. 185, 2022.

[27] G. Ariante, S. Ponte, U. Papa, A. Greco, and G. Del Core, "Ground Control System for UAS Safe Landing Area Determination (SLAD) in Urban Air Mobility Operations," *Sensors*, vol. 22, no. 9, 2022, ISSN: 1424-8220. DOI: 10.3390/s22093226. [Online]. Available: https://www.mdpi.com/1424-8220/22/9/3226.

[28] C. Blanc, R. Aufrère, L. Malaterre, J. Gallice, and J. Alizon, "Obstacle detection and tracking by millimeter wave RADAR," *IFAC Proceedings Volumes*, vol. 37, no. 8, pp. 322–327, 2004, ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)31996-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667017319961.

[29] M. P. Owen, S. M. Duffy, and M. W. M. Edwards, "Unmanned aircraft sense and avoid radar: Surrogate flight testing performance evaluation," in *2014 IEEE Radar Conference*, 2014, pp. 548–551. DOI: 10.1109/RADAR.2014.6875652.

[30] E. Çetin, C. Barrado, G. Muñoz, M. Macias, and E. Pastor, "Drone Navigation and Avoidance of Obstacles Through Deep Reinforcement Learning," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, 2019, pp. 1–7. DOI: 10.1109/DASC43569.2019.9081749.

[31] H. v. Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16, AAAI Press, 2016, pp. 2094–2100.

[32] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16, JMLR.org, 2016, pp. 1995–2003.

[33] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*, 2015, pp. 234–241.

[34] C. Yan, X. Xiaojia, and C. Wang, "Towards Real-Time Path Planning through Deep Reinforcement Learning for a UAV in Dynamic Environments," *Journal of Intelligent & Robotic Systems*, vol. 98, Mar. 2020. DOI: 10.1007/s10846-019-01073-3.

[35] M. J. Hausknecht and P. Stone, "Deep Recurrent Q-Learning for Partially Observable MDPs," in *AAAI Fall Symposia*, 2015.

[36] O. Bouhamed, H. Ghazzai, H. Besbes, and Y. Massoud, "Autonomous UAV Navigation: A DDPG-Based Deep Reinforcement Learning Approach," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, Oct. 2020, pp. 1–5, ISBN: 978-1-7281-3320-1. DOI: 10.1109/ISCAS45731.2020.9181245. [Online]. Available: https://ieeexplore.ieee.org/document/9181245/.

[37] Z. Xue and T. Gonsalves, "Vision Based Drone Obstacle Avoidance by Deep Reinforcement Learning," *AI*, vol. 2, no. 3, pp. 366–380, 2021, ISSN: 2673-2688. DOI: 10.3390/ai2030023. [Online]. Available: https://www.mdpi.com/2673-2688/2/3/23.

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

[39] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-Baselines3: Reliable Reinforcement Learning Implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html.

[40] G. Brockman *et al.*, "OpenAI Gym," *CoRR*, vol. abs/1606.01540, 2016. [Online]. Available: http://arxiv.org/abs/1606.01540.

[41] Y. Song, M. Steinweg, E. Kaufmann, and D. Scaramuzza, "Autonomous Drone Racing with Deep Reinforcement Learning," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021, pp. 1205–1212. DOI: 10.1109/IROS51168.2021.9636053.

[42] M. A. Akhloufi, S. Arola, and A. Bonnet, "Drones Chasing Drones: Reinforcement Learning and Deep Search Area Proposal," *Drones*, vol. 3, no. 3, 2019, ISSN: 2504-446X. DOI: 10.3390/drones3030058. [Online]. Available: https://www.mdpi.com/2504-446X/3/3/58.

Appendix A Algorithms Pseudo-code

Algorithm 1 Deep Q-Learning

Input: α learning rate, ϵ random action probability, γ discount factor,
Initialise q-value parameters θ and target parameters $\theta_{targ} \leftarrow \theta$
b $\leftarrow \epsilon$-greedy policy w.r.t $\hat{q}(s, a|\theta)$
Initialize replay buffer B
for episode $\in$ 1..N **do**
    Restart environment and observe the initial state S_0
    for t $\in$ 0..$T-1$ **do**
        Select action $A_t \sim b(S_t)$
        Execute action A_t, and observe S_{t+1}, R_{t+1}
        Insert transition ($S_t, A_t, R_{t+1}, S_{t+1}$) into Buffer B
        $K = (S, A, R, S') \sim B$
        Compute loss function over the batch of experiences:

$$L(\theta) = \frac{1}{|K|} \sum_{i=1}^{|K|} [R_i + \gamma max_a \hat{q}(S_i^{'}, a_i|\theta_{targ}) - \hat{q}(S_i, A_i|\theta)]^2$$

        Update the NN parameters θ via SGD:

$$\theta = \theta - \alpha \nabla L(\theta)$$

    end for
    Every k Episodes synchronize $\theta_{targ} \leftarrow \theta$
end for
Output: Optimal policy π and q-value approximations $\hat{q}(s, a|\theta)$

Algorithm 2 Double Q-Learning

Initialize Q^A, Q^B, s
repeat
    Choose a, based on $Q^A(s, .)$ and $Q^B(s, .)$, observe r, s'
    Choose (e.g random) either UPDATE(A) or UPDATE(B)
    if UPDATE(A) **then**
        Define $a^* = argmax_a Q^A(s', a)$
        $Q^A(s, a) \leftarrow Q^A(s, a) + \alpha(s, a)(r + \gamma Q^B(s', a^*) - Q^A(s, a))$
    else if UPDATE(B) **then**
        Define $b^* = argmax_a Q^B(s', a)$
        $Q^B(s, a) \leftarrow Q^B(s, a) + \alpha(s, a)(r + \gamma Q^A(s', b^*) - Q^B(s, a))$
    else if UPDATE(B) **then**
    end if
until you die.

Algorithm 3 Trust Region Policy Optimization

Input: initial policy parameters θ, initial value function parameters ϕ
Hyperparamaters: KL-divergence limit, δ, backtracking coefficient α, maximum number of backtracking steps K.
for $k = 0, 1, 2, ...$ **do**
    Collect set trajectories $\mathcal{D}_k$ = $\{r_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}_t$
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
    Estimate advantages $\hat{A}_t$ using the generalized advantage estimation algorithm
    Estimate policy gradient as

$$\hat{g_k} = \frac{1}{|\mathcal{D}_k|} \sum_{r \epsilon \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t$$

    Use the conjugate gradient algorithm to compute

$$\hat{x}_k \approx \hat{H}_k^{-1} \hat{g}_k,$$

    where, $\hat{H}_k$ is the Hessian of the sample average KL-divergence.
    Update the policy by backtracking line search with

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{\hat{x}_k^T \hat{H}_k \hat{x}_k}} \hat{x}_k,$$

    where $j \in \{0, 1, 2, ...K\}$ is the smallest value which improves the sample loss and satisfies the sample KL-divergence constraints.
    Fit value function by regression on the mean-squared error:

$$\phi_{k+1} = argmin_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{r \epsilon \mathcal{D}_k} \sum_{t=0}^{T} (V_\phi(s_t) - \hat{R}_t)^2,$$

    , typically via some gradient descent algorithm
end for

Algorithm 4 PPO with Clipped Objective

Input: initial policy parameters θ, clipping threshold ϵ
for $k = 0, 1, 2, ...$ **do**
    Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
    Estimate advantages $\hat{A}_t^{GAE(\gamma,\lambda)}$ using the generalized advantage estimation algorithm
    Compute policy update

$$\theta_{k+1} = argmax_\theta J^{CLIP}(\theta)]$$

    by taking K steps of minibatch SGD (via Adam), where

$$J^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

end for

Appendix B Specifications and Versions

Table B.1 Hardware and Sofware Specifications and Versions

Hardware / Software	Versions/Specifications
CPU	AMD FX(tm)-8350
GPU	NVIDIA GeForce GTX 1660 Ti
RAM	32.0 GB DDR3
Operating System	Windows 10
Python	3.10.11
Gym Library	0.21.0
Stable-Baselines 3	2.0.0a0
TensorBoard	2.11.2
sb3-contrib	1.8.0
Torch	1.13.1+cu116
TorchVision	1.14.1+cu116
OpenCV	4.6.0.66
AirSim Simulator	1.7.0
Pillow	9.2.0
Unreal Game Engine	4.27.2

Appendix C Tested CNN architectures

Table C.1 First CNN Architecture to be tested

Layer	Input Dims	Transform	Kernel Size	Stride	Padding	Activation	Output Dims
Input	150 x 150 x 1	Conv2D	3	1	1	ReLU	150 x 150 x 32
H1	150 x 150 x 32	Conv2D	3	1	1	ReLU	150 x 150 x 64
H2	150 x 150 x 64	MaxPool2D	2	2	0	None	75 x 75 x 64
H3	75 x 75 x 64	Conv2D	3	1	1	ReLU	75 x 75 x 128
H4	75 x 75 x 128	Conv2D	3	1	1	ReLU	75 x 75 x 128
H5	75 x 75 x 128	MaxPool2D	2	2	0	None	37 x 37 x 128
H6	37 x 37 x 128	Conv2D	3	1	1	ReLU	37 x 37 x 128
H7	37 x 37 x 128	Conv2D	3	1	1	ReLU	37 x 37 x 128
H8	37 x 37 x 256	MaxPool2D	2	2	0	None	18 x 18 x 256
H9	18 x 18 x 256	Flatten	N/A	N/A	N/A	None	82944
H10	82944	Linear	N/A	N/A	N/A	ReLU	1024
H11	1024	Linear	N/A	N/A	N/A	ReLU	512
Output	512	Linear	N/A	N/A	N/A	None	2

Table C.2 Second CNN Architecture to be tested

Layer	Input Dims	Transform	Kernel Size	Padding	Activation	Output Dims
Input	150 x 150 x 1	Conv2D	3	1	ReLU	150 x 150 x 32
H1	150 x 150 x 32	MaxPool2D	2	0	None	75 x 75 x 32
H2	75 x 75 x 32	Conv2D	3	1	ReLU	75 x 75 x 64
H3	75 x 75 x 64	MaxPool2D	2	0	None	37 x 37 x 64
H4	37 x 37 x 64	Conv2D	3	1	ReLU	37 x 37 x 128
H5	37 x 37 x 128	MaxPool2D	2	0	None	18 x 18 x 128
H6	18 x 18 x 128	Flatten	N/A	N/A	None	41472
H7	41472	Linear	N/A	N/A	ReLU	512
H8	512	Linear	N/A	N/A	ReLU	64
Output	64	Linear	N/A	N/A	None	2

Table C.3 Third CNN Architecture to be tested

Layer	Input Dims	Transform	Kernel Size	Stride	Activation	Output Dims
Input	150 x 150 x 1	Conv2D	8	2	ReLU	36 x 36 x 32
H1	35 x 36 x 32	Conv2D	4	2	ReLU	17 x 17 x 64
H2	17 x 17 x 64	Conv2D	3	2	ReLU	15 x 15 x 64
H3	15 x 15 x 64	Flatten	N/A	N/A	None	14400
H4	14400	Linear	N/A	N/A	ReLU	256
Output	256	Linear	N/A	N/A	None	2

Table C.4 The different architecture sizes

Architecture	Parameter Size	Total Size (MB)
Table C.1	86,587,010	401.39
Table C.2	21,359,810	103.40
Table C.3	3,759,010	15.67

Appendix D CNN architectures metrics

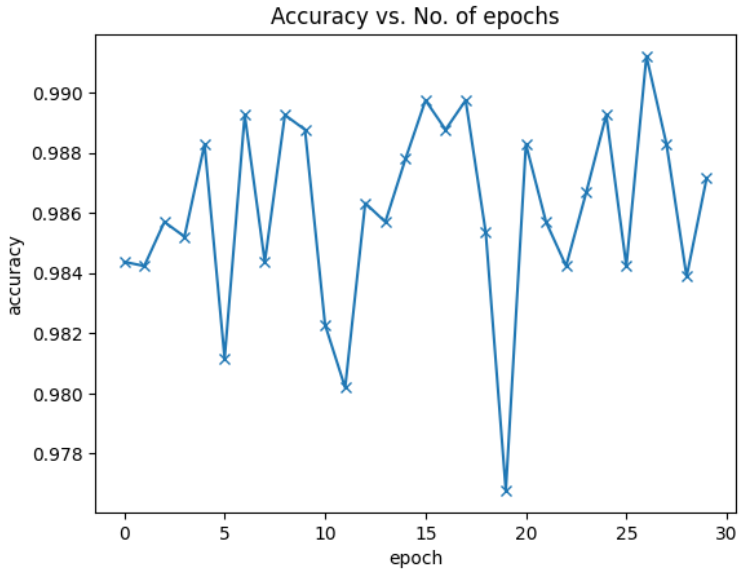


Figure D.1 Training Accuracy of the First CNN Architecture on a Static Environment

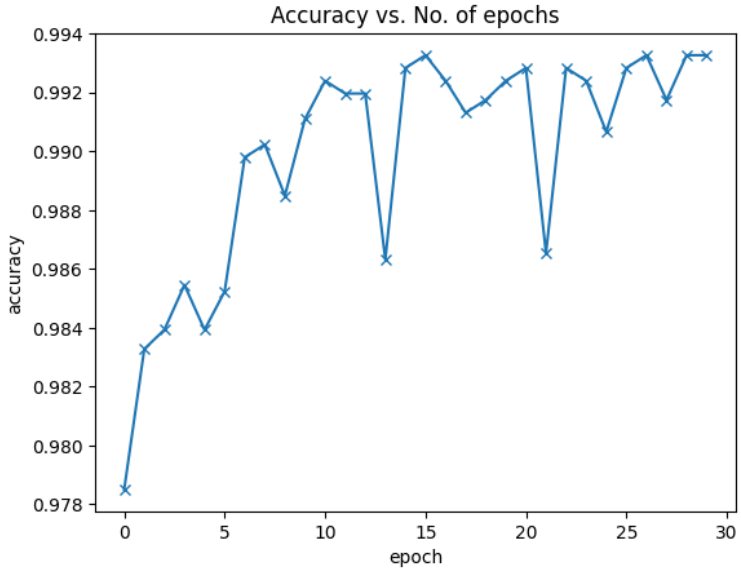


Figure D.2 Training Accuracy of the First CNN Architecture on a Dynamic Environment

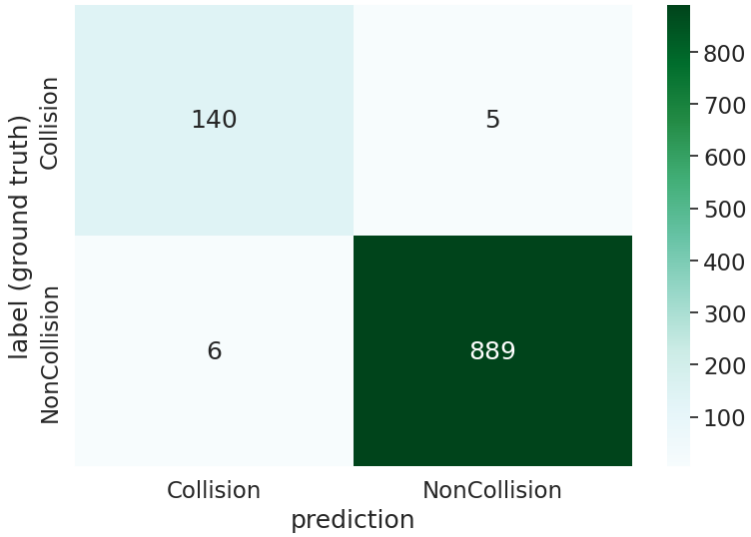


Figure D.3 Confusion matrix of the First CNN Architecture on a Static Environment

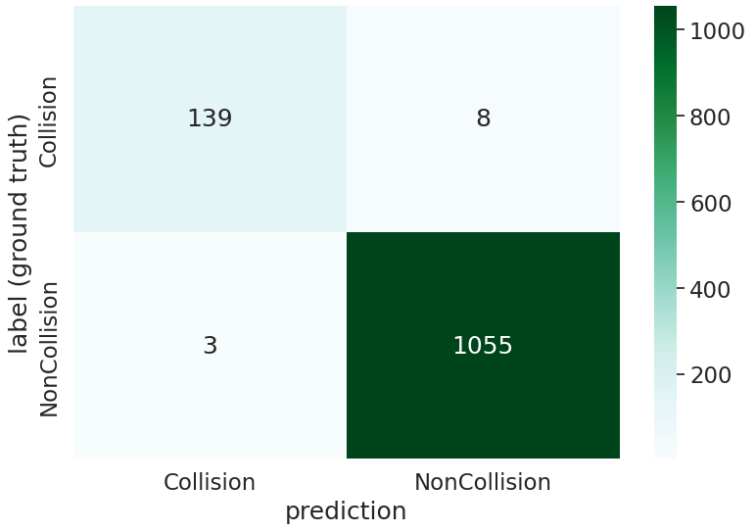


Figure D.4 Confusion matrix of the First CNN Architecture on a Dynamic Environment

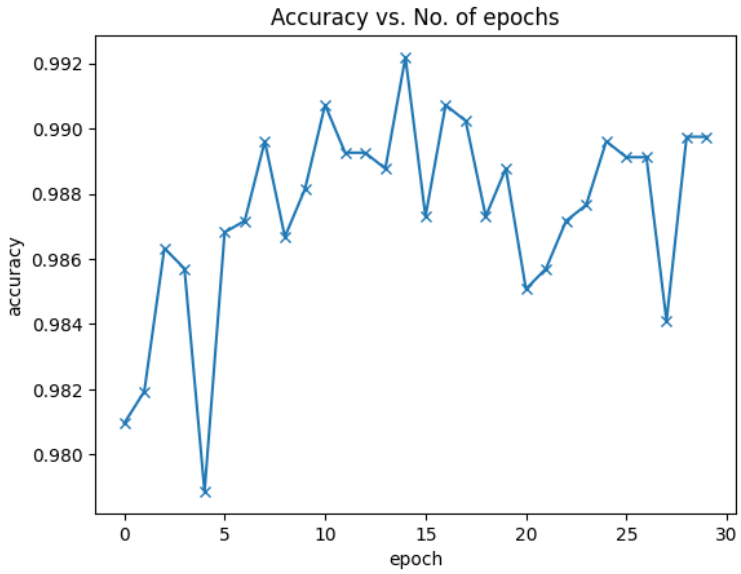


Figure D.5 Training Accuracy of the Second CNN Architecture on a Static Environment

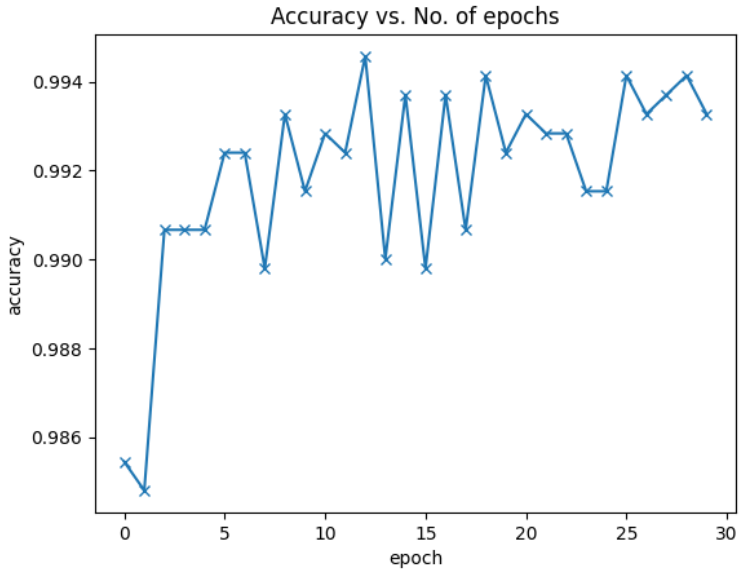


Figure D.6 Training Accuracy of the Second CNN Architecture on a Dynamic Environment

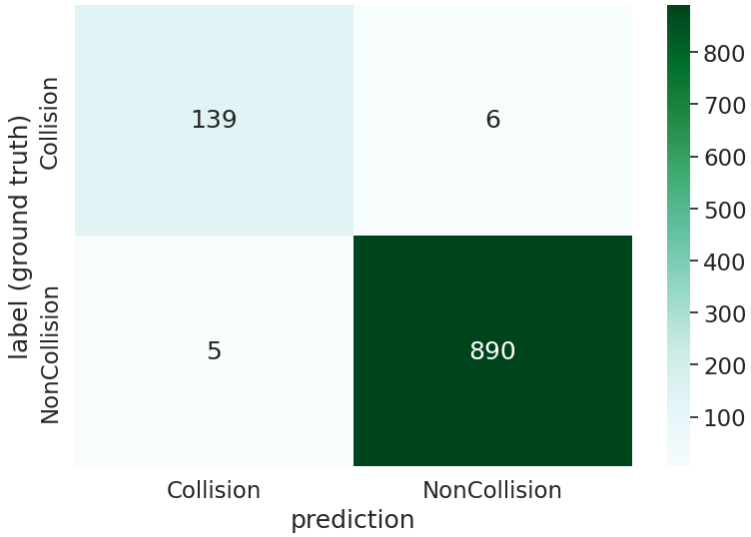


Figure D.7 Confusion matrix of the Second CNN Architecture on a Static Environment

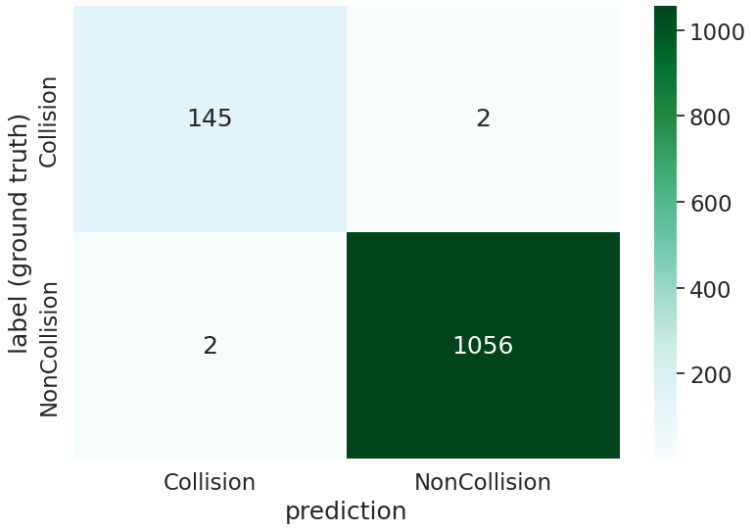


Figure D.8 Confusion matrix of the Second CNN Architecture on a Dynamic Environment