1.5em 0pt

Professorship of Distributed Information Systems

Unmanned Aerial Vehicle Path Planning Optimization: A Comparative Study of DDQN, PPO, and MuZero Algorithms

Masterarbeit von

Bipin Kumar Chaudhary

Matriculation Number: 87715

1. Prüfer	2. Prüfer
Prof.(FH) PD Dr. Habil Mario Döller	Prof. Dr. Harald Kosch

November 13, 2023

Contents

Abstract

Reinforcement Learning (RL) has recently gained significant popularity as the Artificial Intelligence (AI) field continues to evolve and advance rapidly. Extensive research has been conducted to explore the application of Deep Reinforcement Learning (DRL) in various domains. Within DRL, several algorithms have appeared as suitable options for achieving favourable results in specific environments. In this thesis research, the city image maps named "Manhattan32" and "Urban50" are utilized, featuring obstacles, No-Fly-Zones (NFZs), take-off zones, and landing zones. The objective is to train a drone to fly from the take-off zone to the landing zone while avoiding obstacles such as buildings and NFZs. Secondly, in Unmanned Aerial Vehicles (UAVs) with limited power sources, one of the main concerns is how long we let the UAV fly out in the field before it gets back to the landing zone safely. Due to this, the movement budget is considered while training to have enough battery left to return safely to the landing zone. Thirdly, we have considered a global-local map combination to help the agent to converge faster and exploit the environment. This allows the agent to process less information data, faster computation and still able to achieve the coverage area. At last, three DRL algorithms are employed to achieve efficient path planning for the UAV. The algorithms used in this study are Double Deep Q-Network (DDQN), Proximal Policy Optimization (PPO), and MuZero. The algorithms are trained with different hyperparameter settings for a single map and one common environment. Finally, the results of all three algorithms are compared, and the algorithms are evaluated based on their coverage rate, cumulative rewards, successful landings, and other relevant metrics.

Acknowledgements

In pursuing my master's degree, I am indebted to numerous individuals whose support and encouragement have played an integral role in my academic journey. Among them, there are key figures to whom I owe special recognition.

I am incredibly fortunate to have had Mr. Julian Bialas as my supervisor over the past year. His continuous guidance, insightful advice, and the freedom he provided for me to explore my imagination and research interests have been invaluable. Mr. Julian Bialas's unwavering readiness to assist me whenever needed has significantly contributed to my academic progress, and I am deeply grateful in many ways for his tremendous supervision.

I also thank Prof. Dr. Mario Döller, my first supervisor, whose support was instrumental in completing my experiments. His encouraging words gave me added strength and motivation, especially when the experiments took longer than anticipated. Even amidst the challenges of a pandemic, Dr. Mario Döller's guidance and responsiveness eased my mental pressures. I extend my thanks to Prof. Dr. Harald Kosch, my second supervisor, for his role in supporting my academic endeavours.

Additionally, I appreciate the precious friendships I have cultivated during my studies. These friends supported me academically and became a source of strength and companionship. My heartfelt thanks go out to all my friends and family in Passau and those who have wished me well and expressed their love and support.

Finally, I sincerely thank my parents for their unwavering support and sacrifices to ensure I received the best possible education. Their continuous encouragement has paved the way for me to pursue my master's degree at the University of Passau.

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AI ..... Artificial Intelligence

ML ..... Machine Learning

RL ..... Reinforcement Learning

DRL ..... Deep Reinforcement Learning

MP ..... Markov Property

MDP ..... Markov Decision Process

UAV ..... Unmanned Aerial Vehicle

USV ..... Unmanned Surface Vehicle

GPS ..... Global Positioning System

TSP ..... Travelling Salesman Problem

SAR ..... Search and Rescue

DQN ..... Deep Q-Network

ACO ..... Ant Colony Optimization

LSTM ..... Long Short-Term Memory

D3QN ..... Dueling Double Deep Q-Network

PPO ..... Proximal Policy Optimization

DDQN ..... Double Deep Q-Network

NFZs ..... No-Fly-Zones

POMDP ..... Partially Observable Markov Decision Process

GAE ..... Generalized Advantage Estimation

MCTS ..... Monte Carlo Tree Search

CR ..... Coverage Ratio

CRAL ..... Coverage Ratio And Landed

BC ..... Boundary Counter

LA ..... Landing Attempts

MR ..... Movement Ratio

SL ..... Successful Landing

1 Introduction

1.1 Machine Learning

A crucial step in expanding Machine Learning (ML) into the industry involves shifting from data modelling to algorithm modelling [4]. This evolution of ML highlights the transition from statical modelling based on discovering the data-generating model for a problem to statistical modelling focused on finding the best algorithm to reproduce or exploit data. This shift has significantly impacted various fields, benefiting both industry and research. For example, in medicine, ML has been used to learn from image data to identify cancerous cells or detect skin cancer [37]. In finance, it has been applied to option pricing [7], sentiment analysis [31], and meta-data-based systematic review [51]. ML has also had implications for management, healthcare, and agriculture [29] and the legal field to explore social controversies [46].

Previously, when we thought about ML, the terms that often came to mind were supervised learning and unsupervised learning, which played a main role in the field. In recent years, RL has gained admiration and proven to be advantageous. RL is the third pillar of ML and has successfully generated and exploited data using algorithmic methods, leading to increased expectations in various industries [8].

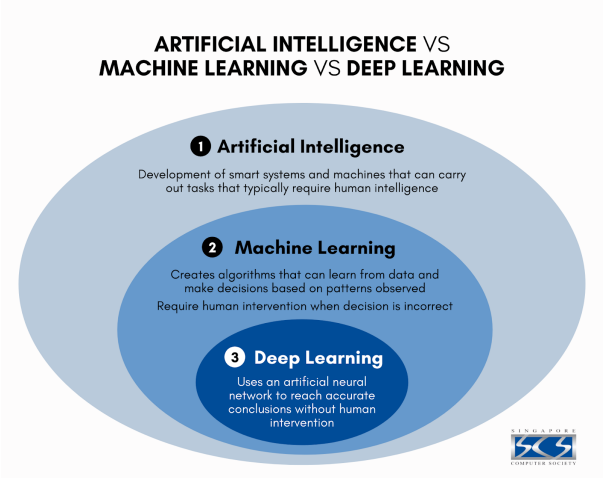

Figure 1.1: Venn diagram to distinguish artificial intelligence, machine learning and deep learning concepts [35].

1.2 Reinforcement Learning Motivation

RL is an area of ML aimed at maximizing cumulative rewards [52]. It can be defined as a framework for solving controlled tasks or decision problems [10]. In RL, an agent makes decisions within an environment to accumulate the maximum cumulative rewards while avoiding negative rewards or punishments. The agent learns by repeatedly performing tasks and learning from the rewards received. This sets RL apart from other branches of ML. In supervised learning, the model learns from labelled data provided by experts, while in unsupervised learning, the model learns patterns from unlabeled data. In contrast, RL learns from the environment, where the agent has knowledge of its current state, takes actions, receives rewards, and decides its future actions based on those rewards. There is also a trade-off between short-term and long-term rewards in RL.

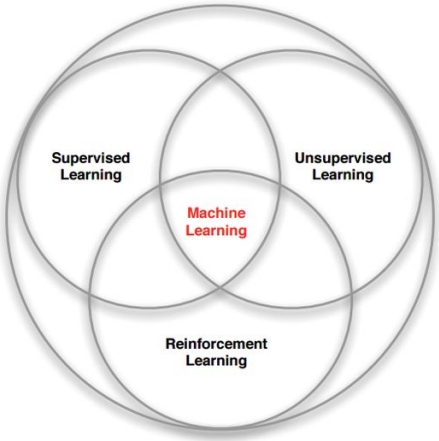

Figure 1.2: Machine Learning branches venn diagram [20].

The best way to explore and research RL is by simulating and analyzing RL within a game world. Games provide an ideal environment for RL as they involve actors playing games, aiming to receive more rewards. Initially, the actors do not know the game environment but learn over time by making good or bad moves, thereby mastering the game. RL has been successful getting several popular games such as Go with AlphaGo [40][41], chess and Shogi with AlphaZero [39], Atari[23] and more advanced games like DOTA2 [25] and StarCraft II with AlphaStar [45].

Given these achievements, it is impossible to ignore the powerful capabilities of DRL and its ability to solve complex tasks. DRL has found applications in various fields, including robotics (e.g., pancake flipping, bipedal walking, archery-based aiming) [16], autonomous vehicles (e.g., learning intersection handling behaviour) [14], healthcare [6], natural language processing (e.g., generating dialogue and extracting information) [48], and many others.

This thesis optimises coverage path planning for UAVs using DRL algorithms. Our experiment created an urban city map with obstacles like buildings and red zones. In this thesis, we considered the combination of flying time and avoidance of obstacles while covering the targeted coverage area. Our drone agent has to cover the target area, avoid obstacles, and land back in the landing zone with the given movement budget. One of the main problems with UAVs is the high power consumption while computing the complex

data of the environment. Therefore, the combination of global-local maps is considered where a combination of two maps is fed via the channel to the neural network. The global map will need less detailed information, so a compressed map of the whole environment will be provided to the agent for information on distance features and directions. For a local map, detailed information is required to take action, resulting in collision avoidance for the drone. That's why cropped (providing only surrounding information concerning the agent's location) and uncompressed maps are fed to an agent as a piece of state information. This combination of global-local maps will also address the scalability, and due to this feature, this research work can also be applied to bigger maps. We also considered the flying time of the drone because the drone has to return to the landing zone before its battery is empty to avoid crashes. We will compare the performance of three different DRL algorithms in the same environment. Previously, only DDQN was employed in such an environment.[44]. However, this case study will show how algorithms like PPO and MuZero perform with different hyperparameter settings. This research has potential applications in various aspects of our daily lives, such as UAVs [53][59][11], intelligent sweeping robots [56], watering tractors [57], cleaning robots [22], tile robots [19], electrical self-control lawn mowers [2], search and rescue robots [43], and more. In the next section, we provide an overview of this thesis structure.

1.3 Overview

The thesis is structured in the following chapters.

- In Chapter 2, we introduce some basic mathematics related to RL, like the Markov Decision Process (MDP) and its optimality theorem. In addition, there is an introduction to DRL and some insight overview of a deep neural network and its training process. We also address the classical RL algorithm and the advantages and limitations of classical RL, resulting in the need for modern deep reinforcement algorithms.
- In Chapter 3, we provide an overview of previous works in UAVs and RL. We also explore research on coverage area and path planning using UAVs.
- In Chapter 4, we introduce the methods employed to compare CPP approaches, outlining three DRL algorithms utilized for training the UAV in the given environment. We employ various hyperparameter combinations for different algorithms to achieve the best possible results in each setting.
- In Chapter 5, we explain each algorithm's results and performances, providing a comparative analysis of their outcomes.
- Finally, Chapters 6 and 7 summarize the discussions and conclusions drawn from the study. We also discuss future works and explain the intended purpose of the algorithm to be used.

2 Technical Background

Before diving into the core method, this chapter focuses on some basics related to this thesis that we need to understand.

2.1 Reinforcement Learning

RL is an area of ML concerned with how intelligent agents should take action in an environment to maximize cumulative rewards. In simple terms, RL involves learning in an environment to make better and wiser decisions over time, ultimately aiming to maximize the total rewards earned during the training process. The fundamental concept of RL is depicted in Figure 2.1. Here are some basic technical terms related to this topic.

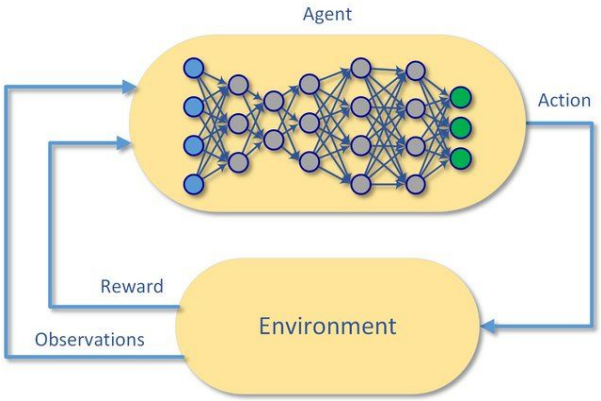

Figure 2.1: Basic reinforcement learning concept [47].

2.2 Agent and Environment

In RL, it is essential to understand two fundamental concepts: the agent and the environment. The agent, also known as the learner or decision maker, is the entity that takes actions within the environment. On the other hand, the environment represents the context in which the agent operates and interacts. Essentially, the agent performs actions within the environment, and as a result, it receives rewards based on those actions. Various terms, such as feedback-based, reward-based, or interactive learning, often refer to RL. The interaction between the agent and the environment can occur in both discrete and continuous time steps. However, this discussion will focus solely on discrete time steps. These terms are used to describe specific characteristics of the learning process. For instance, the agent interacts with the environment and receives feedback on

the quality of its actions. Furthermore, it is referred to as reward-based learning due to the assignment of rewards for its actions. Figure 2.2 illustrates the relationship between the agent and the environment, highlighting how rewards are accumulated based on the performed actions.

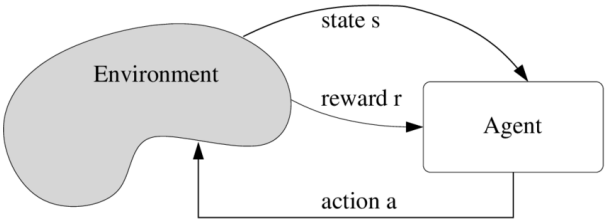

Figure 2.2: Agent-environment interaction in reinforcement learning [36].

Based on the previous statements, it is evident that distinguishing the agent and the environment is often a crucial aspect of RL. In specific scenarios, there are situations where the agent has no control over certain parts which become part of the environment. For example, when multiple agents are learning to play soccer, one agent cannot dictate the actions of the others. In such cases, the other actors become part of the environment for the agent initiating the actions. In DRL, most interactions are episodic, meaning they start at an initial stage and conclude at a terminal state. Here are some categories of environments and actions:

- Fully observable (e.g., Chess) vs. Partially observable (e.g., Poker)
- Single-agent (e.g., Atari) vs. Multiple-agent (e.g., Deep traffic)
- Deterministic (e.g., Cart pole) vs. Stochastic (e.g., Deep traffic)
- Static (e.g., Chess) vs. Continuous (e.g., Cart pole)
- Discrete (e.g., Chess) vs. Continuous (e.g., Poker)

2.3 Markov Property

If the present state contains all the information needed to choose the action leading to the next state, it exhibits the Markov Property (MP). This means that, in terms of drones, if the current position of a drone encapsulates all the necessary information about the states that will help it move to the next steps in the environment, it can be defined as having the Markovian property. In other words, MP implies that future decisions depend only on the present state and are independent of past history. A Markov Decision Process (MDP) consists of a tuple of states, actions, state transitions, and rewards (S, A, T, R). We will provide a brief explanation of these components in this section. Typically, MDP involves infinite time steps due to their inherent complexity. However, this discussion will focus solely on finite time steps.

2.3.1 State

A state can be defined as the instance of the agent at a particular time 't'. We denote the set of states as S, consisting of individual states s_1, s_2, ..., s_N. The total number of states in S is $|S|$, equal to N. Each state s is characterized by a collection of features or properties that uniquely describe the condition of the environment in that specific situation.
At the given time t, the State S_t is Markov if and only if

$$\mathbb{P}(S_{t+1}|S_t) = \mathbb{P}(S_{t+1}|S_1, \ldots, S_t). \tag{2.1}$$

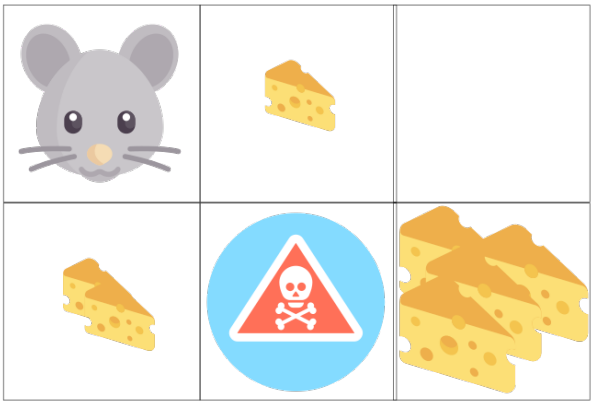

Figure 2.3: States with respect to Rat in RL Environment [42].

From the diagram 2.3, we can deduce some of the states as follows:

- State_1: The rat is at the initial top left corner of the square.
- State_2: The rat will be in a square with one cheese.
- State_3: The rat will be in a square with danger.
- State_4: The rat will be in a two-cheese square.
- State_5: The rat will be in a square with a pile of cheese.

In this scenario, each state represents specific information about the rat and cheese's position and any obstacles in their proximity. The features of each state may include the position of the rat, the position of the cheese, the distance between them, the presence of any obstacles, and all the past scenarios it has been in.

2.3.2 Action

We define $A = \{a_1, a_2, \ldots, a_N\}$ as the set of actions that can be performed from the current state within the environment to the next. In general, not all actions are applicable in all states to reach the next state. Therefore, we define $A(s) \subseteq A$ as the set of actions applicable in state s, a subset of the set of all states, S.

2.3.3 Transition of State

When the action $a \in A$ is applied in a state $s \in S$, an agent transitions from the initial state s to the new state $s' \in S$ based on its probability distribution over the set of possible transitions. We define the given transition function T as:

$$T : S \times A \times S \to [0, 1]$$

where $(s, a, s') \mapsto P[s'|s, a]$

For a Markov state s and successor state s', the state transition probability is defined as:

$$P_{s,s'} = P(S_{t+1} = s'|S_t = s) \tag{2.2}$$

The probability matrix P is defined as:

$$P = \begin{bmatrix} P_{11} & P_{12} & \dots & P_{1n} \\ P_{21} & P_{22} & \dots & P_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ P_{n1} & P_{n2} & \dots & P_{nn} \end{bmatrix} \tag{2.3}$$

where the sum of each row of the matrix is equal to 1. [38]

There are certain conditions for the transition of state T being a proper probability distribution over possible next states, as listed below:

- For all $s, s' \in S$, $a \in A$, $0 \leq T(s, a, s') \leq 1$
- To account for certain actions not applying to certain states, we define a set $T(s, a, s') = 0$ for all triples (s, a, s') with $s', s \in S$
- For all $s \in S$, $a \in A$, $s' \in S$, $T(s, a, s') = 1$

2.3.4 Reward

The reward function is one of the vital aspects of RL that drives its agent to learn the desired behaviour. The reward function assigns a scalar value to each state-action pair, providing immediate feedback on the quality of the action taken in that state[38]. Rewards can be defined as:

$$R : S \times A \times S \to \mathbb{R}(\text{set of real numbers})$$

where $(s, a, s') \mapsto R(s, a, s')$

In the MP, the rewards function is a tuple consisting of (S, P, R, γ).

where the reward function

$$R_s = \mathbb{E}[R_{t+1}|S_t = s] \tag{2.4}$$

- S_t is a finite sets of states.

- P is a state transition probability matrix.
- γ is a discount factor, $\gamma \in [0, 1]$

If we are in a given state s at time t, we get an immediate reward as R_{t+1} for the next state. In RL, we mostly maximise the accumulated cumulative rewards obtained over time with a series of actions.

In this context, R is a scalar function that can be understood as follows: if R is positive, it indicates a good reward and action, whereas a negative value signifies a lousy action and is considered a punishment. To illustrate this, consider the earlier example of the rat and cheese. Obtaining the cheese is associated with a positive reward, while getting trapped in the danger square corresponds to a negative action, resulting in a negative reward. Similarly, in this research, flying a drone into a no-fly zone or colliding with obstacles is considered a negative action, leading to negative rewards. Conversely, covering the coverage path is seen as a positive action, resulting in positive rewards.

2.3.5 Returns

Returns can be defined as the sum of total accumulated rewards from the time step t and are discounted by the multiple of γ, so the rewards decrease in each subsequent time step[38].
Mathematically, it can be defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.5}$$

- γ is a discount factor, $\gamma \in [0, 1]$
- The value of γ decides if our model is focused on immediate or delayed rewards.
- γ close to 0 means it's short-sighted.
- γ close to 1 means it's far-sighted.

However, there is the possibility to use an undiscounted Markov reward. For example, if you know there are specific finite steps and want to assign some rewards for each step, then we can make $\gamma = 1$, and it will be considered an undiscounted Markov reward.

2.3.6 Value

Value function can be defined as the value of being in a particular state or the probability of getting the expected return from that state onwards. It shows how important it is to be in a specific state. Value function v(s) provides the long-term value of being in state s. We are taking the random sample returns starting from the particular state(s) at the time(t) till the terminal state and taking the average returns of all the samples to consider the value of that state. It can be referred to as a state-value function when it provides

the value associated with being in that particular state.
Mathematically, It can be defined as:

$$v(s) = \mathbb{E}[G_t | S_t = s] \tag{2.6}$$

- G_t is the total returns we get from state t onwards
- $\mathbb{E}$ is because the environment is stochastic.

2.3.7 Bellman Expectation Equation

The Bellman expectation equation is the sum of the expected immediate reward and the discounted value of the following state. It can be derived from the previously mentioned value function in equation (2.6).

$$\begin{aligned} v(s) &= \mathbb{E}[G_t | S_t = s] \\ &= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \,\middle|\, S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \,|\, S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} \,|\, S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) \,|\, S_t = s\right] \end{aligned}$$

Here, G_t is expanded as immediate reward R_{t+1} and discounted reward for subsequent next state $\gamma v(S_{t+1})$.

Alternatively, we can define the Bellman expectation equation as:

$$v = R + \gamma P v \tag{2.7}$$

where P is a transition probability matrix [38].

2.4 Markov Decision Process

MDP can be defined as for the given current state s_t, we take action a_t, and a reward r_t, the probability of transitioning to the next state s_{t+1} and receiving the next reward r_{t+1} depends solely on the current state s_t and action a_t. It does not depend on any previous states $(s_{t-1}, s_{t-2}, ..., s_0)$, actions $(a_{t-1}, a_{t-2}, ..., a_0)$, or rewards $(r_{t-1}, r_{t-2}, ..., r_0)$ [27]. The primary difference between the MP and the MDP is that, in the MDP, we finally start to make decisions based on action and reward. Another way to understand the difference is that in MDP, there is a cost associated with the action/decision the agent makes. In the MP, we are not making any decisions.

Mathematically, it can be written as:

$$\begin{gathered} Pr(\mathrm{s}_{t+1} = s_0,\, r_t = \mathrm{r} |\; s_t, a_t, r_t,\, s_{t-1},\, a_{t-1},\, r_{t-1}, ...r_1,\, s_0,\, a_0) \\ \Pr(s_{t+1}{=}s_0,\, r_{t+1}{=}\, r_t, s_t, a_t) \end{gathered} \tag{2.8}$$

2.4.1 Policy Function

In the MDP tuple (S, A, T, R), we need to fix the deterministic route for the agent, called the policy function, denoted as $\pi : S \rightarrow A$. This policy function maps a state to the possible action in that state.

Policy π can be defined as mapping the current environment state to the probability distribution of the action to be taken from that state[38]. It can be represented as:

$$\pi(a|s) = P[A_t = a|S_t = s] \tag{2.9}$$

In simple words, it can be described with examples like if you are in state s and if you have the option to go right and left, the policy maps you with the probability of taking action to go to the left and the probability to go to right from that state. This denotes that agents have control over the action, given that policy always depends on its current state rather than the past states in which the agent has been. It can be recognized as Morkovian property.

2.4.2 Value Function

The value function in the MDP can be categorized into two types.

- State-value function
- Action-value function

State-value function $v_\pi(s)$ can be defined as how good the value of the state(s) is if we follow the policy π[38]. It can be written as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.10}$$

where $\mathbb{E}_\pi$ is the expected return by following policy π.

Action-value function $q_\pi(s, a)$ can be defined as how good it is to take a particular action(a) from the state(s) and then follow the policy π[38]. It can be written as:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.11}$$

In other words, if we are in the state(s), take action (a) and by following policy π, we are considering how much expected return or rewards can be achieved from that time.

2.4.3 Optimal Value Function

Optimal state-value function $v_*(s)$ can be defined as the maximum value that can be achieved in all the states in the system. Several different policies can be followed in the system. If the value function $v_*(s)$ is maximum over all the available policies, it can be defined as an optimal value function[38]. Mathematically, it can be written as:

$$v_*(s) = \max_{\pi} v^{\pi}(s) \tag{2.12}$$

Optimal action-value function $q_*(s, a)$ can be defined as the maximum rewards that can be gathered over all the policies in the system. Therefore, given the state(s), if we take action(a) from that state onwards, the maximum rewards that can be achieved will give the optimal action-value function[38]. It is also called a Q-function.
Mathematically, it can be defined as:

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a) \tag{2.13}$$

The main goal is to find the optimal value function, considered the performing step in the MDP.

2.4.4 Optimal Policy Function

In a given system, there are several different policies that an agent can follow. However, we are interested in determining the optimal or best policy that can maximize the extraction of rewards. Certain conditions must be met for one policy to be considered better. It can be written as:

$$\pi \geq \pi' \quad \text{if } v_{\pi}(s) \geq v_{\pi'}(s), \quad \forall s \tag{2.14}$$

- There are always at least one policy π_* which is equal to or better than all other policies π in the given system.
- Optimal policy satisfies optimal state-value function.
- Optimal policy satisfies optimal action-value function.

We can achieve optimal policy function by following the formula below:

$$\pi_*(a|s) = \begin{cases} 1, & \text{if } a = \arg\max_{a \in A} q_*(s, a) \\ 0, & \text{otherwise} \end{cases} \tag{2.15}$$

2.4.5 Bellman Optimality Function

Bellman optimality function is almost the same as Bellman expectation function. The only difference in the Bellman optimality function is that instead of taking an average of all the actions from a particular state, we take the maximum action from that state.

To understand the Bellman optimality value function, let's understand 2.4 in detail.

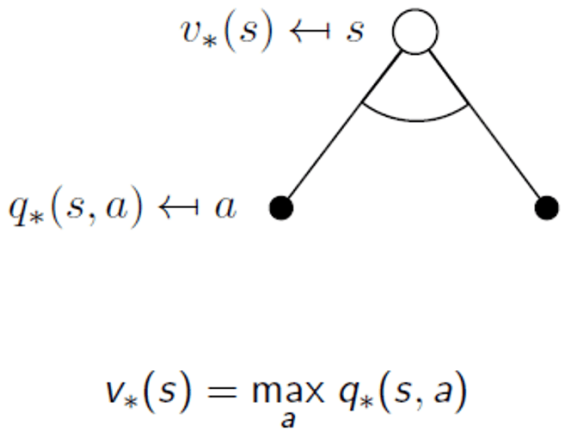

Figure 2.4: Bellman optimality function for value function [3].

From the figure 2.4, the agent is in the state(s) where it can take two actions of either going left or right. Instead of taking an average of both actions and deciding which way to go, in the Bellman optimality function, we take the action with greater value, represented in the equation below.

$$v_*(s) = \max_a \quad q_*(s,a) \tag{2.16}$$

Similarly, we can define the state-action function for the Bellman optimality function from figure 2.6.

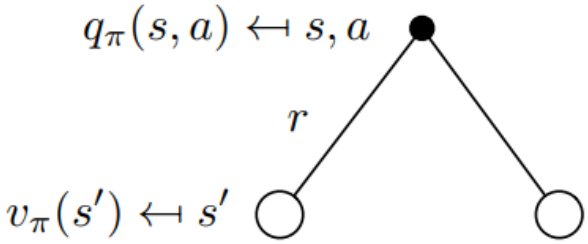

Figure 2.5: Bellman optimality function for action function [3].

From Figure 2.5, we consider an agent in states taking action a. The agent does not decide which state it will end up in; instead, the environment determines this. In this scenario, the agent calculates the average value of both possible states. The critical difference is that we already know the optimal value for both potential next states in this state. Mathematically, we can write this with an equation given below.

$$q_*(s,a) = R_a^s + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \tag{2.17}$$

Finally, we get the state-value function if we attach the above two figures. In figure 2.6, if the agent is in state s and takes action a where a is determined by the policy of the agent at that state and from there, as a result of action a, the agent gets blown away in an environment which is decided by the environment taking an average of the optimal value of states.

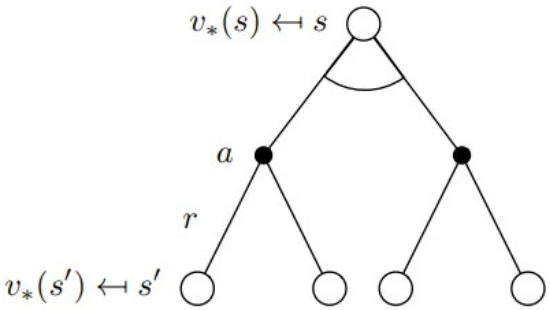

Figure 2.6: Bellman optimality function for state-value function [3].

Now, using 2.17 in 2.16, we can rewrite the equation as follows

$$v^*(s) = \max_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \tag{2.18}$$

Similarly, we can write the equation for the state-action function as well.

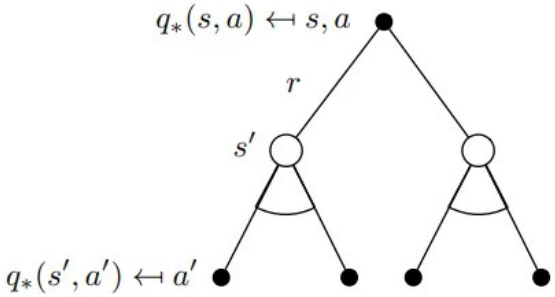

Figure 2.7: Bellman optimality function for state-action function [3].

Figure 2.7 shows the optimal state-action function. It can be described as if the agent takes the action a from the state s it was in, and the environment blows the agent inside the environment and lands it to any state s' from where the agent decides to take maximum q_* value.
The equation for this condition can be expressed as:

$$q_*(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s', a') \tag{2.19}$$

2.5 Model-based and Model-free Algorithm

There are two different approaches to solving the problem in RL. Model-based RL involves creating a model that learns about the environment, its transition probabilities and the rewards before training. Later, when the agent interacts with the environment, it specializes in optimizing the model or its policies. So, having an explicit representation of the environment helps the agent plan and map possible future trajectories before making any decisions. Model-based RL facilitates finding optimal policy and optimal value based on the learned model.
However, on the other hand, a Model-free RL algorithm does not involve learning the model of the environment. Still, it depends on directly interacting with the environment for its transition probability and the rewards it gains from taking the action in the environment. It observes the state-action pair, maps the rewards function to each state and updates its policy and value function during these processes without any knowledge of the dynamics of the environment model. A model-free RL algorithm mostly focuses on optimizing the agent's behaviour based on the observed interactions in the environment and updating the policies and values over time.
Comparatively, Model-free RL needs more interaction and training to converge and learn an optimal policy than model-based algorithms. Model-based is mostly used in a stable environment where rewards are known, like chess. In contrast, model-free is used in many real-world scenarios where the environment is partially observable or changes frequently.

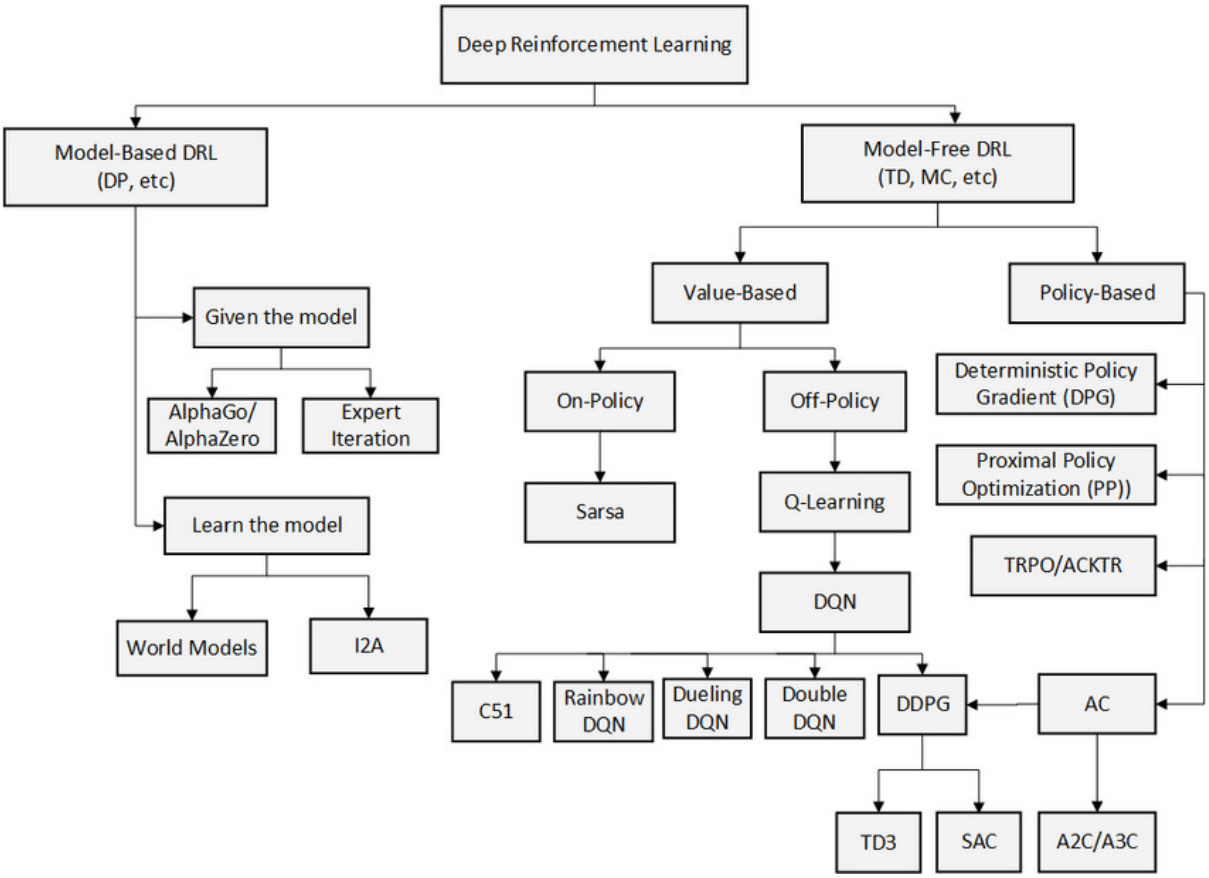

Figure 2.8: Taxonomy of reinforcement learning algorithms [36].

Figure 2.8 shows the categories of the DRL algorithm based on model-based and model-free algorithms. This thesis will primarily focus on algorithms like DDQN, PPO and the MuZero algorithm. DDQN and PPO fall under the Model-free category, whereas MuZero is model-based. However, some parts of MuZero still use model-free categories as it directly learns a policy and value from interacting with the environment.

2.6 Exploration and Exploitation

In RL, an agent learns about the environment by interacting with it, taking action, and knowing its state. Exploration can be described as when agents try to explore the environment in search of a reward that has long-term benefits. This means it helps agents explore the environment and improve their knowledge. On the other hand, exploitation can be defined as the greedy approach of an agent taking action based on the current estimated value of rewards from the current observation and taking action with the highest value, which is called greedy action. It should be noted that the greedy action is taken based on the estimated value but not the real value. The action taken is called greedy action, and the policy obtained from it is called greedy policy.
The exploration-exploitation challenge is about dealing with uncertainty. Exploitation involves embracing established, effective strategies and navigating familiar paths to quickly attain rewards based on acquired knowledge and experience. Conversely, exploration is about trying new ways, like taking different routes to find potentially better options and

learning in the long run. Finding the right balance is essential. Going too much with what might make you miss new chances while trying too many new things might not pay off immediately. In this project, we used three algorithms, each using different ways for the exploration vs. exploitation trade-off. In DDQN, we use the e-greedy function for exploration vs exploitation purposes. The PPO algorithm uses the "clipped surrogate objective" to balance exploration and exploitation. The MuZero algorithm uses Monte Carlo Tree Search (MCTS) and Upper Confidence Bound (UCB) formulas to balance exploration and exploitation.

2.7 Neural Network

A neural network is a computational model inspired by the human brain's structure. It consists of numerous nodes called neurons or units that are interconnected to each other to process complex computations. The neural network consists of many layers to form a complete model. Figure 2.9 shows the primary comparison of the biological and artificial neurons.

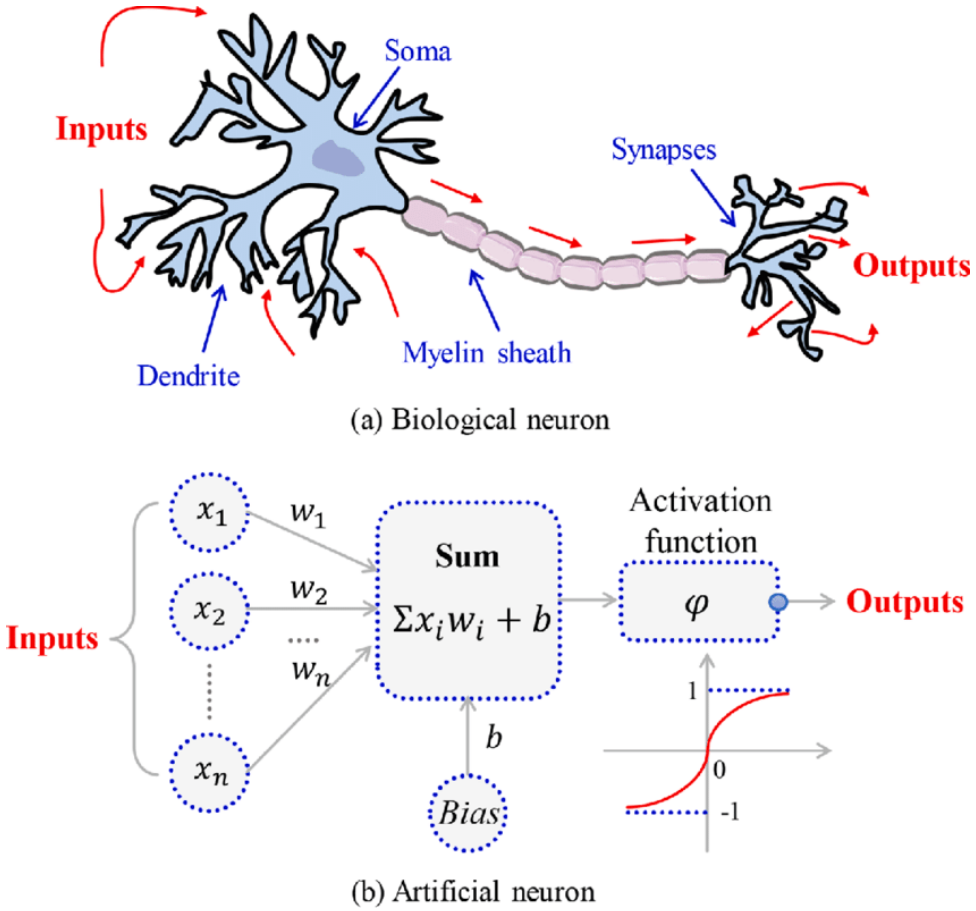

Figure 2.9: Comparison between biological neuron and artificial neuron [49].

The neural network consists of the following key components.

- Neurons/Units: Neurons are the basic building components of neural networks. They receive the inputs process, compute them, and provide the output signals.

They are inspired by the biological neurons present in the human brain shown in figure 2.9.

- Layers: Neurons are arranged in layers in the neural network model. There are three layers in a neural network: The input layer, the hidden layer and the output layer. The input layer takes the initial data as input, which is passed to the hidden layer, where the data is processed, and then the output layer produces the final output of the model.
- Weight: Weights can be described as the number assigned to each edge's connection from one neuron to its subsequent layer neurons. All the neurons in the network are connected to its subsequent layer network through which the data is passed. The weight provides the value which determines the importance or strength of data passing through the connections. In the figure 2.9, $w_1, w_2...w_n$ are weights.
- Activation Function: Each neuron in the subsequent layer applies the activation function to the weighted sum for the inputs it receives and adds the bias. The main functionality of the activation function is to produce the non-linearity into the output of the neuron. It decides whether the current neuron should be activated or not. There are several types of activation functions. Some are rectified linear unit (ReLU), hyperbolic tangent (tanh), softmax, sigmoid, etc.
- Bias: Bias can be defined as the constant which is added to the product of inputs $x_1, x_2...x_n$ and the weight. In figure 2.9, bias is denoted by b. Adding the bias to the input features provides the offset result, helping the activation function to shift towards either the positive or negative side.
- Connections/Edges: Edges of neural networks are the one which connects two neurons. The information inside the neural networks flows through the edges, and each edge is assigned its weight, which defines the strength of the connections.
- Architecture: Architecture consists of a complete model of all the above-mentioned components in a specific pattern. The architecture consists of the input layer of neurons connected to the hidden layer and the output layer, where each layer is assigned its weights, bias and activation functions, completing its complete model.
- Loss Function: Loss function can be defined as the difference between the actual output and the model's predicted output. The loss function helps to adjust the network weights according to the difference between the actual and predicted values. The network's performance is quantified by its loss function during the training process, which helps to converge the network's output towards the true value.
- Optimization Function: The optimization function helps reduce the model error by iterative changing the weights and tuning other hyper-parameters inside the network architecture. Optimization helps the model improve the network's accuracy by helping to minimize the cost of function. Gradient descent is a common optimization technique used in neural architecture to find the local minima of a differential function.

As illustrated in Figure 2.10, the neural network architecture is derived from the information provided above. The figure illustrates the representation of inputs x_1, x_2, x_1, x_2, x_5.

Each hidden layer neuron processes its output by summation of multiplication of the inputs by the weights of each corresponding neuron's edges. However, bias is not added in the figure shown. The output function is the summation of the multiplication of previous inputs to the corresponding weights of the neurons.

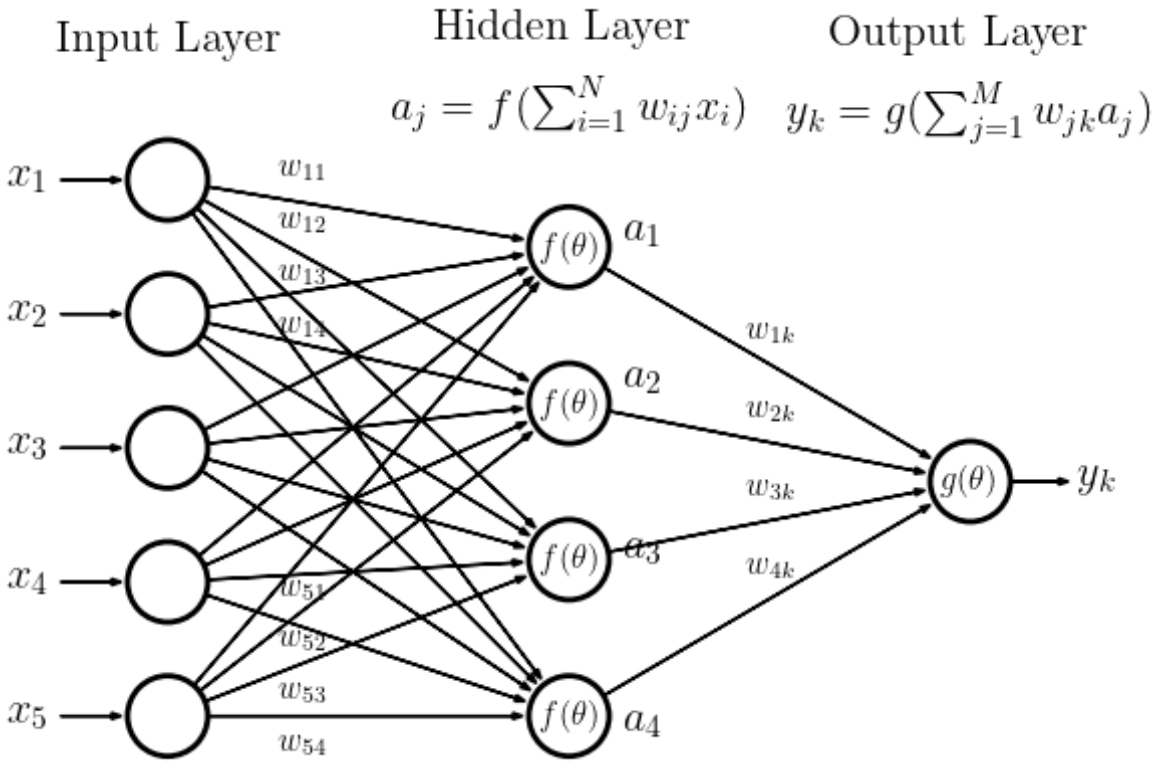

Figure 2.10: Neural network architecture [5].

2.7.1 Backpropagation

One of the most important topics of the neural network is how neural networks actually learn. It can be understood by understanding the backpropagation process inside the neural network. The learning for the model refers to minimising cost function by learning which weights and biases are responsible for it in that neural network. The cost can be defined as the sum of the square difference between the predicted output and the network's actual output, which gives the average cost function of the network. The basic process of backpropagation involves two steps:

- Forward pass: Each neuron performs two operations: computing the weighted sum and then processing the sum via an activation function. The output of the activation function decides if it should be activated or not. In the forward pass, the inputs are provided to input layers, passed to the hidden layer, and finally to the output layer. After getting the output, the error is calculated using a loss function which compares the actual output to the predicted output of the neural network.
- Backward pass: The main functionality of backward pass is to distribute the total error we get from calculating the differences between predicted and actual errors.

The distribution of the total error takes place so that all the network architecture weights and biases can be updated to minimise the network's cost function. The process of the backward pass takes place so that the next forward pass can utilize the updated weights to find the minima of the network.

Backpropagation is a very efficient way to calculate the gradient in neural network architecture with multiple layers where weights are updated, helping the network converge towards the actual output. Hence, it allows the model to learn the task.

2.8 Classical Reinforcement Learning

Before we dive into the complex RL algorithms discussed in this thesis work, we need to learn the simpler algorithms out there and their limitations, because of which we need these complex algorithms. When considering simpler RL algorithms, Q-learning and State-Action-Reward-State-Action (SARSA) are considered some of the earliest and most influential better-performing algorithms for more straightforward tasks. Also, on the other hand, these algorithms introduced fundamental concepts and techniques that remain relevant in modern RL research. This section will briefly discuss Q-learning to understand the need for complex algorithms like DDQN, PPO and MuZero.

2.8.1 Q-Learning

Q-learning is a model-free RL that targets learning the optimal action-value function known as Q-values of the given MDP. The algorithm with the iterative approach focuses on learning and improving over time and solving the task with higher accuracy. Q-learning is model-free, meaning no environment model helps the algorithm during its RL process. The learning process can be described with the help of the following components involved during the learning process.

- Agent: The operator that takes actions within the environment.
- State: The variable that describes the agent's current position in an environment.
- Actions: The legal, behavioural act of agents in the environment.
- Rewards: The numerical prizes or punishments given to the agents according to their actions.
- Episodes: The distinct sequences of interactions between the agent and the environment, concluding when the agent reaches a terminal state or when it ceases to take further actions from that point onward.
- Q-values: The metric values provided for the state-action measurement.
- Q-tables: They are the action-value table in Q-learning and the way in the Q-learning algorithm to keep track of the value of the state in the environment and update it gradually as the actor learns more about the environment. The Q-table is updated for Q-values using the MDP for each action-value pair. The Q-values represent the expected cumulative reward an agent can achieve by taking a specific

action in a particular state and following the optimal policy thereafter. The Q-tables are typically two-dimensional tables with rows representing states and columns representing the number of actions that can be taken in the environment. The dimensionality of the Q-tables depends on the number of states and possible actions in the environment.

Learning Q-values is done by temporal difference or using the Bellman equation. The temporal difference works by comparing the Q-values we got in the current state and action and the values from the previous state and action. The working theory of Bellman has been explained and mentioned in the equation (2.20). The following section outlines the procedural steps in employing Q-tables within the Q-learning algorithm.

- Initialization: The Q-table is initialized with arbitrary values or set to some predefined initial values. The value initialized at the beginning of the table can influence the exploration vs exploitation process for the beginning steps of the algorithm.
- Learning: During learning, the agent interacts with the environment, takes the action, obtains some rewards as feedback for the action and then transitions from the old state to the new state. After each episode, the Q-values in the table are updated based on the rewards and transitions.
- Q-value Update: The Bellman equation is used to update the Q-value.

$$\text{Q(s', a)} \leftarrow (1-\alpha)\cdot Q(s,a) + \alpha\cdot(r + \gamma\cdot \max_{a'} Q(s',a')) \tag{2.20}$$

  where:

  - α(alpha) is the learning rate, which controls the step size updates.
  - r is the immediate reward received after action a in states s.
  - γ(Gamma) is a discount factor determining future rewards' value.
  - s' is the next state after taking the action a from state s.
  - a' is the action which is taken from state s that maximizes the Q-value in state s'.
- Exploration vs Exploitation: The actor must balance their approaches between exploration(taking new actions) and exploitation(choosing the action with the highest Q-values). Some different strategies can be used for this process, such as the epsilon-greedy function or softmax to interact with the environment and decide what to opt for between exploration and exploitation.
- Convergence: After repeating the above-mentioned steps of updating the Q-values. Q-tables converge towards the best possible steps that can be made to achieve maximum awards from the environment.
- Policy extraction: Finally, once the Q-table is converged and the process of Q-learning is complete, the best policy can be extracted by choosing the actions with the highest Q-values in each state.

The algorithms of Q-learning and the steps involved in acquiring the optimal Q-value are mentioned below.

Algorithm 1 Q-learning Algorithm: $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
States $\mathcal{X} = \{1, \ldots, n_x\}$
Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
Learning rate $\alpha \in [0, 1]$, Usually $\alpha = 0.1$
Discount rate $\gamma \in [0, 1]$
procedure Q-Learning($\mathcal{X}$, A, R, T, α, γ)
    Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
    while For each episode **do**
        Start in state $s \in \mathcal{X}$
        while s is not terminal **do**
            Calculate π function based on Q and exploration strategy like epsilon-greedy(e.g. $\pi(x) \leftarrow_a Q(x, a)$)
            $a \leftarrow \pi(s)$
            $r \leftarrow R(s, a)$ ▷ Get the reward
            $s' \leftarrow T(s, a)$ ▷ Move to the new state
            $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$
            $s \leftarrow s'$
        end while
    end whilereturn Q
end procedure

Despite being one of the earliest and most foundational algorithms in RL and contributing the initial algorithms for many types of research in the RL field, there are some limitations to the Q-learning algorithm. Before talking about the limitations of Q-learning, we have mentioned a few advantages of the Q-learning algorithm below.

Advantages of Q-learning

- Model-free learning: Q-learning falls in the category of model-free learning. Which means it does not require an extensive knowledge model of the environment beforehand. Therefore, it does not require any information about the environment's transition probabilities and reward functions, making it easier to apply to a wide range of problems without extensive knowledge of similar states.
- Generalization: Q-learning can generalize the knowledge of one state to its other similar states, which means that Q-learning can learn knowledge of optimal actions of one state. Then, it can be applied to other similar states.
- Convergence guarantees: Q-learning almost always converges towards the optimal Q-values and policy under certain conditions. This makes it one of the most valuable for ensuring the algorithm's effectiveness over time.

- Off-policy learning: Q-learning, being off-policy learning, can learn from the data generated from the different policies. This makes the algorithm more flexible and easier to learn, and it uses the old data from other policies to generate the new policy.

Limitations of Q-learning

- Exploration challenge: In RL, it's essential to balance exploration and exploitation while learning. However, it can be difficult to discover new optimal actions while exploiting the learned knowledge in the Q-learning algorithm.
- Cruse of dimensionality: The main feature of Q-learning is using the table to a Q-values for all the state-action pairs. However, using this algorithm in higher dimensional states or action spaces can get complicated and impractical.
- Slow convergence: Converging in large environments can take many iterations, resulting in high demand for resources and slow computation.
- Requires discretization: For the continuous environment, it often needs discretization so that Q-learning can be applied. Due to this, it is not a favourable choice for a continuous environment. In addition, there are chances of error because discretization introduces quantization of the space.
- Delayed rewards problem: Q-learning struggles to comprehend the conditions of delayed rewards-based environments where the long-term consequences of actions are not immediately shown in effect. That can lead to suboptimal policies and difficulty obtaining the best overall policy.
- Instability and divergence: In the learning process, Q-values might oscillate or even diverge due to the update process. It requires a delicate balance of its learning rate schedule to mitigate this problem.

Due to the limitations mentioned earlier, it is not the optimal choice to use classical RL like Q-learning, SARSA, etc. Even though these provided the essential foundations for the advanced DRL algorithms we use for many higher dimensionality projects and applications, this classical algorithm cannot be preferred due to its various limitations. There are several benefits to using DRL algorithms like DDQN, PPO and MuZero over classical RL, such as:

- Handling high-dimensional state spaces: Due to using neural networks instead of tabular versions, DRL algorithms can easily handle the higher dimensionality problems for the function approximation. Classical RL struggles for higher dimensions problems because of the exponential growth of the Q-table.
- Generalization: DRL algorithms better generalise the learned knowledge from one state to similar states. This means they can handle unseen similar states which need similar actions.
- Feature extraction: DRL is better at feature extraction from the raw data.

- Continuous action spaces: Unlike classical RL, DRL algorithms can be used for continuous actions without discretization. As many real-world problems are based on continuous actions, DRL algorithms are more favourable to use as they can directly output the continuous actions through neural networks.
- Function approximation: DRL algorithms use the neural network for approximating the Q-values or policies, representing the value function and policies more efficiently. This makes it more suitable to use for complex problems.
- Complex policies: It can handle complex policies due to the deep neural network used for data processing.
- Handling delayed rewards: DRL algorithms like MuZero can candle significantly delayed rewards over time and visualize the importance of action even for future steps.
- Experience replay: Algorithms like DDQN use experience replay, where it stores the value of experiences(state, action, reward, next state) and samples randomly for later updates. This helps in reducing the correlations and improves stability.
- Adaptive batch sizes: An algorithm like PPO utilizes the advantages of mini-batches for each update step, allowing flexibility in using collected data efficiency.
- Model-based learning: DRL, like MuZero, leverages model-based learning, enabling it to simulate future trajectories without requiring direct interaction with the real environment. It can efficiently explore and use data, resulting in cost-effectiveness and less time consumption.

The basic categorization of the popular RL algorithms based on their environment type is shown in figure 2.11.

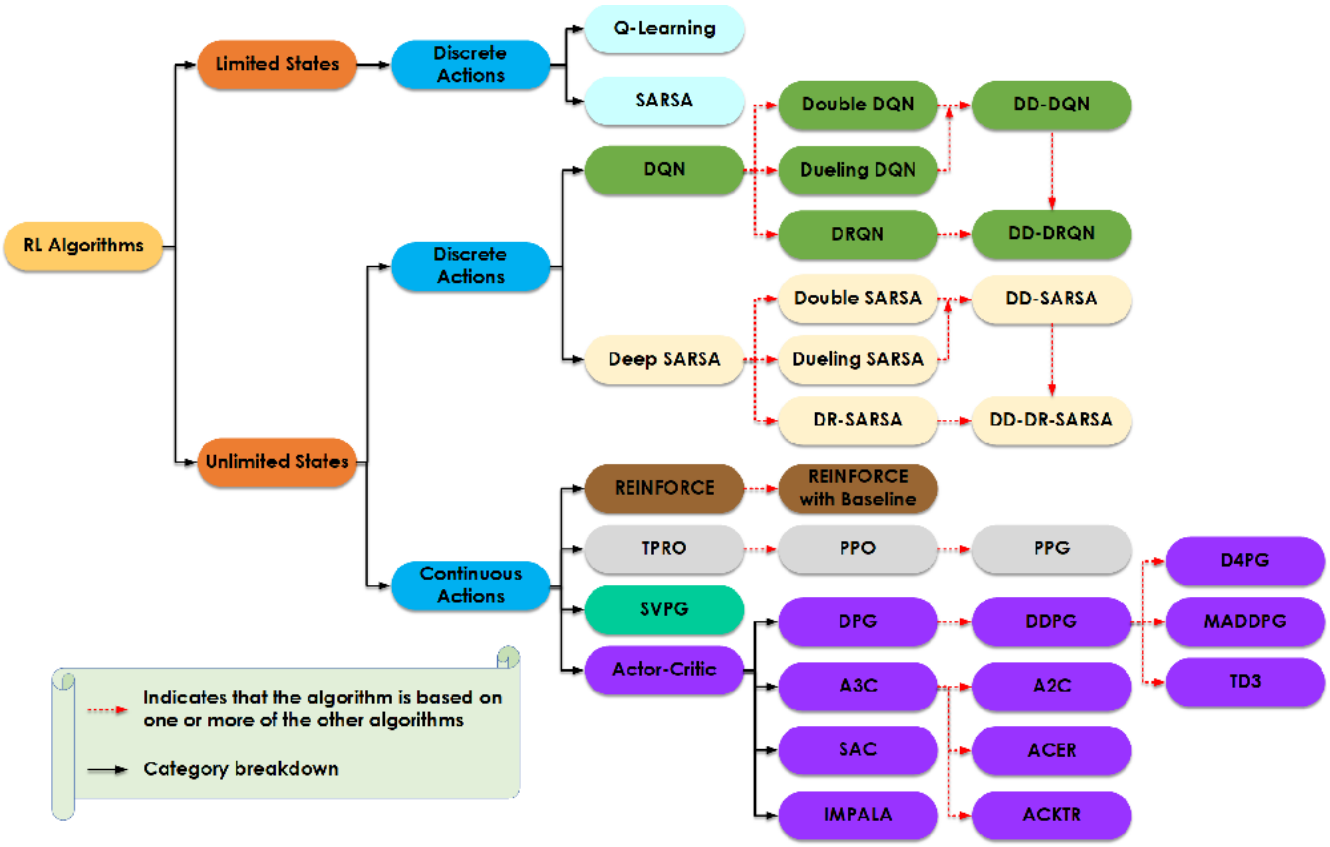

Figure 2.11: Reinforcement algorithms classification based on the environment type [1].

However, figure 2.11 does not have the algorithm MuZero mentioned, which is the model-based algorithm. MuZero combines model-based learning with model-free learning techniques to improve its performance in complex environments like chess, shogi, etc.

3 Related Work

Several types of research are carried out for Coverage Path Planning (CPP), which has a wide range of applications in industrial and daily life settings. Efficient and optimized path planning can be beneficial in various scenarios. RL has emerged as a popular topic in AI and is often the preferred choice in the automation industry. Many coverage path planning tasks have been successfully addressed using RL, outperforming conventional algorithms. This approach has been applied to machines requiring path planning, such as UAVs and Unmanned Surface Vehicles (USVs). Below are some related research works with brief descriptions of why and how they were conducted and their efficiency.

3.1 Literature Review

Noh *et al.* in [24] have tried an adaptive policy in 2D environments where cleaning robots generate the minimum energy path. They employed a basic divide-and-conquer approach by dividing the entire area into sub-segments, connecting the sub-regions, and forming a global path.

Maciel-Pearson *et al.* in [21] have employed the double state input strategy, combining knowledge from raw images with a map containing positional information obtained through the use of a Global Positioning System (GPS). The Extended Double Deep Q-Network (EDDQN) was utilised for DRL. This approach enabled navigation through multiple unseen environments, even with several obstacles.

Yao *et al.* in [55] have utilized a cooperative air-sea targeted search approach, employing UAVs and USVs for Search and Rescue (SAR) missions. The research uses a DRL algorithm called PPO, which has proven efficient. This research's primary concept revolves around operating two cooperative vehicles, where UAVs assist USVs. Each vehicle employs its DRL, resulting in two distinct reward functions. The use of two reward systems facilitates faster convergence and improved efficiency. However, the research did not consider the power and energy-saving constraints specific to UAVs or USVs.

Ouahouah *et al.* in [28] have focused on designing algorithms to prevent collisions in both urban and rural populated environments for UAVs. The objective was to enhance the adaptability and usability of UAVs in situations where operators cannot maintain a direct line of sight to control the vehicles. Therefore, there was a need for UAVs that could autonomously plan and operate Beyond Visual Line Of Sight (BVLOS). The author employed two techniques for collision avoidance. The first technique utilized a Probability Distribution-Based Collision Avoidance (PICA) framework. The second technique involved a Dubbed-RL-Based Collision Avoidance Framework (RELIANCE), which leveraged the Deep Q-Network (DQN) for collision avoidance.

Qi *et al.* in [30] have utilized an improved version of PPO for coverage path planning, addressing limitations observed in classical PPO algorithms such as low efficiency and poor adaptability. This research has employed a modified version of PPO called the Frequency-Decomposition PPO (FD-PPO) algorithm. This algorithm is based on frequency decomposition and incorporates heuristic reward functions to solve the path-planning problem for UAVs. The research paper also compares classical PPO and FD-PPO, revealing that FD-PPO demonstrates better convergence than PPO, leading to improved rewards.

Le *et al.* in [18] have discussed the utilization of PPO, a deep learning technique, for achieving optimal coverage of an area with a tiling robot while minimizing the required actions. The research considered the classical Travelling Salesman Problem (TSP) as the basis for solving this problem. Using complementary learning, TSP-based reinforcement learning was applied to optimize and train the robot. Furthermore, the researchers have also compared it with the classic TSP based on the Ant Colony Optimization (ACO) approach.

Zhou *et al.* in [58] have discussed the multi-robot CPP to cover an area while avoiding obstacles efficiently. They have utilized multiple robots and have introduced communication between them to enable coordinated operation and information sharing when they are within communicative range. Authors have employed the DRL algorithm known as the actor-critic algorithm to accomplish this task.

Xing *et al.* in [53] have introduced preprocessing steps for the raster map, effectively removing blank areas that the USVs do not easily cover. They have utilized an improved version of the DQN algorithm for coverage area and path planning after coverage. They have mentioned that the enhanced version of DQN is faster and achieves a higher coverage rate. However, it is worth noting that the dynamic constraints and performance of USVs have not been considered.

Kiemel *et al.* in [15] have discussed an efficient method for fast and scalable painting using DRL. The authors have employed a PaintRL simulator, which utilizes RL for optimized industrial spray painting on various industrial objects. They have utilized proximal policy gradients based on DRL to accomplish this task. However, it is essential to note that this research does not necessarily cover optimal path planning.

Rückin *et al.* in [32] have used a method for informative path planning for aerial robots based on DRL. They have combined classic tree search algorithms with offline-learned neural networks that predict informative sensing actions. They have used an improved AlphaZero algorithm that addresses and overcomes the computational expense of Monte Carlo rollouts and the sample inefficiency in action selection. Furthermore, they have validated its performance multiple times, achieving significantly better results. The method has been applied to real-world surface temperature data.

Silver *et al.* in [41] have proposed the AlphaZero DRL algorithm, the successor to the previously well-known algorithm AlphaGo. The objective of AlphaZero was to master the popular Chinese game Go without human supervision. The algorithm achieved this by learning through self-play, where it played multiple games against itself randomly. AlphaZero utilized a single neural network instead of separate networks for policy and value.

Deng *et al.* in [9] have used the Asymmetric Travelling Salesman Problem (ATSP) to solve the coverage of a large area using UAVs. The authors have partitioned the total area into sub-areas and deployed USVs to find the globally optimal trail and transition paths.

Yan *et al.* in [54] have used the Dueling Double Deep Q-Network (D3QN) to obtain optimal actions and plan the optimal path in dynamic environments with numerous threats and obstacles. The authors have employed two separate networks, with one network outputting value functions and the other network outputting advantage functions. These two components are combined to approximate the Q-value, enabling the selection of better actions in the environment. Additionally, an epsilon-greedy function is utilized with heuristic search rules to determine the action to be taken.

Kyaw *et al.* in [17] have mentioned DRL to solve the CPP problem based on the TSP. The authors have divided the map into subdivisions and applied TSP to obtain an optimal path. Recurrent Neural Networks (RNN) with Long Short-Term Memory (LSTM) were used as one of the network layers in the architecture. To verify its effectiveness, they compared it with conventional methods in terms of path length, execution time, and overlapping rate on four different maps with obstacles. The results demonstrated its superior performance. Authors have also claimed that it can be utilized in larger grid maps to achieve efficient coverage with a generated coverage path.

Wang *et al.* in [50] have used RL to solve the coverage path planning problem for the Kiwi fruit-picking robot. Initially, LIDAR technology was employed to generate a two-dimensional map indicating the approximate locations of the kiwi fruits on the farm based on their coordinates. Subsequently, the grid maps were divided, creating a suitable scenario for the TSP. Finally, they applied an improved traditional DQN called re-DQN RL to collect kiwi fruits efficiently. They claimed that the convergence rate of this approach is significantly faster, resulting in shorter paths and navigation periods compared to previously used algorithms.

3.2 Motivation

This research aims to use superior feature algorithms that could enhance the training of UAVs compared to traditional RL algorithms. Various methods are available for solving the CPP problems. However, traditional CPP methods typically involve isolating individual target areas through segmentation and later joining them using distance costs to form a graph. Each segment is then covered using a boustrophedon path. This simplifies the CPP challenge into the problem of the TSP, which is NP-Hard. Classical techniques can solve the TSP at the cost of significantly increased time complexity as the number of target areas grows. In addition, one of the main issues of UAV training is the big map being fed to the neural networks, generating vast numbers of hyperparameters and resulting in a lengthy training process. Due to this, the global-local map is considered for scalability purposes. DRL is widespread for this kind of research due to its prior knowledge and assumptions about the environment. However, most of the conventional DRL algorithms have not proved very efficient regarding the complexity of UAV control tasks and computational efficiency. The test cases in this project are done with three algorithms: DDQN, PPO and MuZero. Each of these algorithms has advantages that can

be beneficial to train the UAV with faster convergence and more accuracy in coverage area and landing in an environment like "Manhattan32". For example, DDQN has better generalization capability, handling non-stationary environments, and availability of training data, outperforming conventional algorithms like DQN. Meanwhile, PPO's trust region constraints help to prevent large policy updates, resulting in robustness as it is less sensitive to hyperparameters. Similarly, PPO directly optimizes its policy functions, resulting in a smoother policy. We chose MuZero to test the environment because unlike DDQN and PPO, which are model-free algorithms, MuZero operates as a model-based algorithm, acquiring an understanding of the internal state of the environment. This capability allows it to leverage state-of-the-art methods for training, enabling the anticipation of potential future steps. This, in turn, contributes to improving actions and refining the policy. Another advantage of MuZero is its better sample efficiency capability. It leverages and builds the learned model of the environment. It interacts less with the environment, which can be beneficial when data collection is costly, like UAV.

4 Comparison of CPP Approaches Utilizing DRL

In this chapter, we will discuss the methodology experimental setup, including the choice of environment or scenarios we made in our thesis along with the utilization of DRL for CPP approaches.

4.1 Assumptions and Model

The project's main goal is to find the best path possible to complete the targeted coverage area with the help of a drone flying over or near it. We initially generated the static map containing buildings, NFZs, take-off and landing zones. Buildings are represented with brick-like yellow blocks, take-off and landing zones are represented with blue blocks and NFZs are represented in red. The area that needs to be covered is represented in green on the map, which can be seen in the image of the results.

4.1.1 Environment and Unmanned Aerial Vehicle(UAV) Model

The image of the environment map is shown in figure 5.42. Initially, the square grid environment of size $M \times M \in \mathbb{N}^2$ with the cell size c, where $\mathbb{N}$ is the set of natural numbers. The map can be explained as tensors. The map $M \in \mathbb{B}^{M \times M \times 3}$ has three layers, where $B = 0{,}1$. The starting/landing zone in the map lies in the 1-layer, the union of the NFZs and obstacles are located in the 2-layer, and obstacles are in the 3-layer of the map. The motion of the UAV is limited to ensure it avoids collisions with obstacles and avoids entering NFZs. Moreover, the UAV is required to initiate and conclude its task within cells designated as takeoff and landing zones. It must remain within maximum flight duration constraints, so a movement budget is assigned.

4.1.2 Target Area and Mission

In our project of coverage path planning, we focus on covering the designated area with the drone, which will have a field-of-view camera-like sensor directly underneath the UAV and fly above or near the target area. The designated target area is $T(t) \in B^{M \times M}$, where each cell indicates whether it needs to be covered. The current field of view of the camera can be explained with the $V(t) \in B^{M \times M}$, which indicates whether each cell is in the current view. The field of view covers 5 cells from the current position of the UAV. In addition, the Building obstructs the UAV's line of sight, making it unable to see around

the counter, which is also included in the calculation field of view. As a result, the target area can be described as:

$$T(t+1) = T(t) \wedge \neg V(t) \tag{4.1}$$

where, $\wedge$ is cell-wise *and* logical operation and $\neg$ is cell-wise *negation* logical operation[44]. In addition to that, obstacles cannot be the coverage area in our environment. Meanwhile, the start/landing zone and NFZs can be the coverage area. The mission is to try to cover as much coverage area possible in a given possible movement budget.

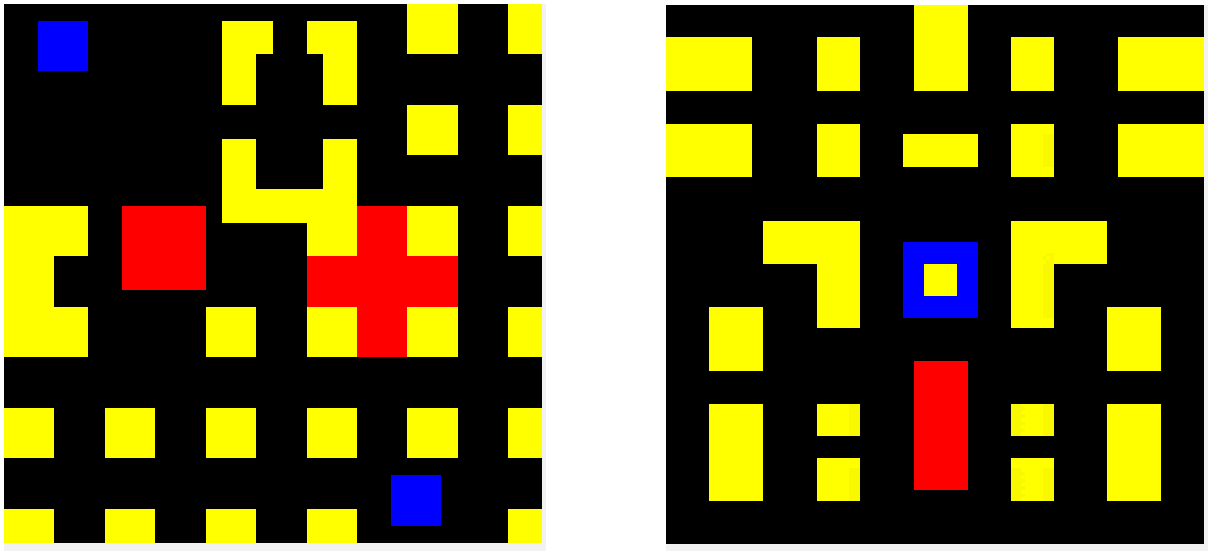

Figure 4.1: Environment maps for "Manhattan32" and "Urban50" [44].

4.1.3 Partially Observable Markov Decision Process

The problem of viewing the states and covering the visible area without segmenting the map can be addressed using a Partially Observable Markov Decision Process (POMDP). We formulate our problem into POMDP, represented as a tuple of $(S, A, P, R, \Omega, O, \gamma)$. where,

- S is state space.
- A is action space.
- $P : S \times A \times S \mapsto \mathbb{R}$ is the transition probability function.
- $R : S \times A \times S \mapsto \mathbb{R}$ is the reward function which maps state, action and the next state to the real value reward.
- Ω is the observation space.
- $O : S \mapsto \Omega$ is the observation function.
- $\gamma \in [0, 1]$ which controls the long and short term rewards.

Finally, from all the above functions, we can write the state space for our problem as:

$$S = \mathbb{B}^{M \times M \times 3} \times \mathbb{R}^{M \times M} \times \mathbb{N}^2 \times \mathbb{N} \tag{4.2}$$

Where $\mathbb{B}$ part is the environment map, $\mathbb{R}$ part is the target Map, $\mathbb{N}^2$ part is the position of UAV and $\mathbb{N}$ is the flying time.

The state $s(t) \in S$ at time t consists of:

$$\mathbf{s}(t) = (\mathbf{M}, \mathbf{D}(t), \mathbf{P}(t), b(t)). \tag{4.3}$$

where

- $\mathbf{M}$ is the environment with start and landing zone, NFZs, and obstacles.
- $\mathbf{D}(t)$ is the target map representing the remaining cells to be covered at time t.
- $\mathbf{P}(t)$ is the UAV's current location at time t.
- $b(t)$ is the UAV's remaining movement budget at time t.
- The action $a(t)$ of the UAV at the given time t belongs to the set A of possible actions within the environment. These actions include $\{\text{north}, \text{south}, \text{east}, \text{west}, \text{land}\}$.

The rewards function consisting of $R(s(t), a(t), s(t+1)$ for the behaviour of the UAV are explained in the following ways:

- $r_c(positive)$ is assigned to the agent for covering the new cells in the environment, compared to previous states. i.e., comparing $s(t+1)$ to $s(t)$.
- $r_{sc}(negative)$ is the safety controller (SC) for the UAV. It penalises the drone while colliding with the buildings and when it enters the NFZ areas.
- $r_{mov}(negative)$ is the movement penalty for the drone, which is given for every step it takes without completing the mission.
- $r_{crash}(negative)$ is another penalty for the drone because of the limited movement budget. So when the movement budget turns to zero, and if it crashes without landing safely inside the landing zone, it results in a huge penalty as a negative reward for a crash.

4.1.4 Network Architecture

In this project, we have compared three different DRL algorithms. By now, we already know that the most important part of DRL is the neural network involved in learning the algorithm. So, in this section, we have mentioned some common network basics, which we will use in all three algorithms as we apply the concept of using a global-local map, where our global map will carry information about the whole map in a compressed form. In contrast, the local map carries the information of the agent's nearby surroundings in uncompressed form. The architecture in figure 4.2 shows how global-local maps are combined before being passed to our neural network. Up to the flattened layer, the

common architecture used by all three algorithms, namely DDQN, PPO, and MuZero, is illustrated in figure 4.2.

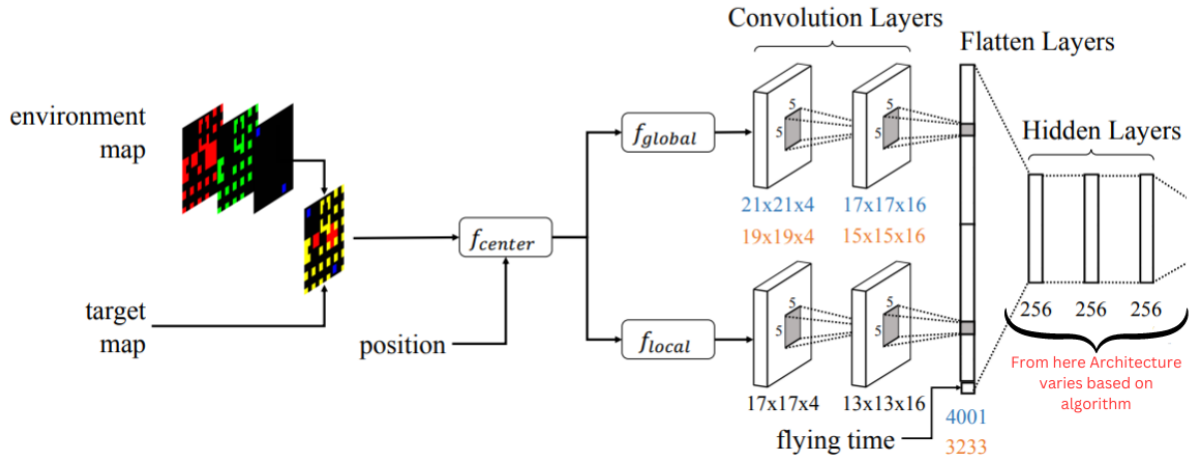

Figure 4.2: The architecture, featuring map centring and global and local mapping, demonstrates variations in layer size. The blue colour denotes the "Manhattan32" scenario, while the orange colour represents the "Urban50" scenario [44].

Although we have used the common algorithm for the first section of the network architecture, where some preprocessing techniques involve getting the image data ready to feed into the network, the second part of the network is based on the algorithms we will train on. In the figure, two colours have been used to denote the shape of the architecture. Blue represents the "Manhattan32" map, whereas orange represents the "Urban50" map.

4.1.5 Neural Network Model and Data Preprocessing

To feed our image data into the neural network, we need to preprocess it to make it accessible to the neural network. At first, the drone's position is encoded into the 2-dimensional one-hot representation. This means encoding the position inside the whole grid. The position is encoded inside the grid to be stacked with the three-channel map and the coverage grid to make it a five-channel input for the network's initial convolutional layer, as shown in Figure 4.2. The kernels of the convolutional layers can create direct geometric connections between the current position and nearby cells. The flying time or movement budget, which helps the drone determine how much time is left to return to the landing zone on time, is fed into the network after the convolutional layers. After that, the convolutional layers are padded so there is no mismatch between the input and output shapes of the map. All the layers are zero-padded, helping them preserve their spatial dimensions and aiding in the proper application of convolutional operations. However, the NFZs have been padded with the one-padded so that there is an extra layer on the NFZs, which helps with enhanced safety margin, avoids close calls, and, in addition, helps to minimize the risk of breaching. Finally, all the convolutional layers are flattened to be

fed into the fully connected layer of our algorithms. In addition to that, the movement budget is also added to the flattened layer. The convolutional layer is passed through the Rectified Linear Unit(ReLu) activation function, which finally connects to the fully connected layer. The remaining architecture of the fully connected layers varies according to the algorithm we use to train the model.

4.2 Algorithms used to train the model

This section will discuss all three algorithms in detail and how they can be used to train our model.

4.2.1 Double Deep Q-Network

Background

DDQN algorithm is an extension or improvement upon the original DQN algorithm. The basic requirement of the algorithm is that it helps to address the main problem of DQN, which is the overestimation bias problem. DQN is the DRL that uses the neural network to estimate the Q-function(action-value function), representing the cumulative future rewards for different actions from the particular given state. While training the model using DQN, the next state is selected by choosing the maximum Q-value. However, due to this, there is the possibility of achieving the overestimation of Q-values, resulting in sub-optimal or unstable learning. Let's see the DQN equations once again to understand this overestimation problem.

Deep Q-Network

The basic objective of DQN learning is to minimize the distance between Q(s, a) and the target Q-value, which is TD-target. Hence, the minimization of the error loss function can be defined as:

$$L_i(\theta_i) = \mathbb{E}_{a\sim\mu}\left[(y_i - Q(s, a; \theta_i))^2\right] \tag{4.4}$$

where,

$$y_i = \mathbb{E}_{a'\sim\mu}\left[r + \gamma \cdot \max_{a'} Q_{\text{target}}(s', a'; \theta_{i-1}) \mid s, a\right] \tag{4.5}$$

where,

- r is the immediate reward obtained action a in state s and moving to next state s'.
- γ is the discount factor.
- $\max_{a'} Q_{\text{target}}(s', a')$ represents the maximum Q-value among all possible actions a' in the next state s', which are estimated by the target Q-network.

- $\mathbb{E}_{a\sim\mu}$ denotes the expectation operator, representing the expected value over possible outcomes.

The TD-target y_i and $Q(s,a)$ are estimated with two separate neural networks. The network calculating y_i and $Q(s,a)$ are called target-network and Q-network, respectively. The parameter θ_i belongs to the Q-network whereas θ_{i-1} belongs to the target network. The action taken by the actor in the environment is determined according to the policy $\mu(a|s)$. In contrast, the policy of the target network will be calculated by following the greedy policy, which means that $\pi(a|s)$ selects a' that helps to maximize $Q(s,a)$. Figure 4.3 shows a deep Q-network using two networks for TD calculation.

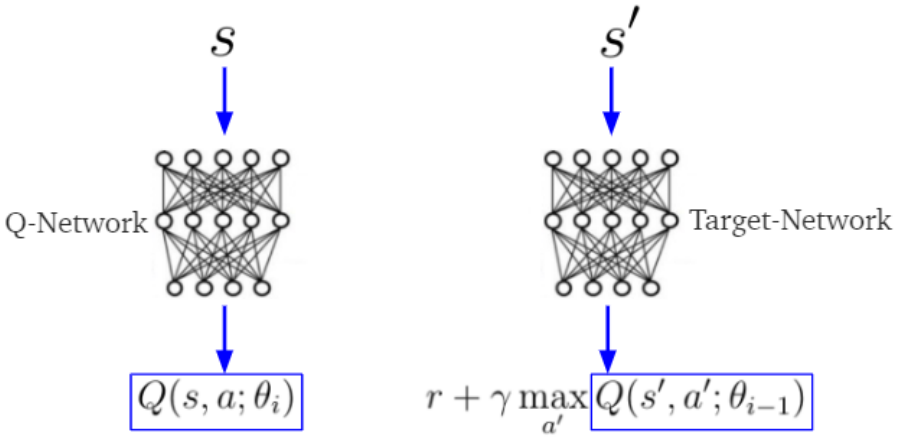

Figure 4.3: Deep Q-Network using Q-Network and target network separately [26].

Problem with Deep Q-Network

From the equation (4.5) equation, we notice that DQN focuses on choosing the maximum Q-value values while updating the network.

specially,

$$\max_{a'} Q_{\text{target}}(s', a') \tag{4.6}$$

Here, the main issue of maximizing the Q-value function is that when we have the model with slightly more noisy Q-values or the target fluctuation is large in the learning process; there will be chances of overestimation. It is not proven that overestimation is necessarily the problem and affects the agents negatively while learning the agents. For example, if all values are higher on average, the selection of relative actions is saved, and there won't be a much worse policy. However, the main problem of overestimation is not being uniform. When we deal with neural networks in algorithms like DQN, our neural networks are prone to overgeneralizing. They can lead to unrealistic high Q-values for specific state-action pairs, easily resulting in an unstable learning process and potentially

slower learning. To address this problem, we chose the DDQN, which potentially performs better than DQN with overestimation issues.

Double Deep Q-Network

DDQN algorithm addresses overestimation in the target equation of the DQN algorithm by decoupling the action selection and value estimation steps. In DQN, the selection and evaluation of the actions are coupled. Using a target network to select the action and simultaneously evaluating the action selected results in chasing its own tail scenario, causing the overestimation problem. In DQN, the target network gives the $Q(s, a_i)$ from where all the actions a_i are decided in the particular state s. Then, the greedy policy chooses the $Q(s, a_i)$ with the highest value for action a_i. This shows that the target network is not only choosing a_i but also evaluating its quality using $Q(s, a_i)$.
The main concept of DDQN is to break down the maximum operation in the target into action selection and action evaluation. However, we will still use the target network from the DQN to calculate the value function. We will use a separate online network to calculate the greedy policy and the estimates of its value using the target network. We are using everything similar to DQN but replacing the target of DQN with a new target equation, as shown in the equation (4.7)

$$y_i^{\text{DoubleDQN}} \equiv \mathbb{E}_{a' \sim \mu}\left[r + \gamma Q\left(s', \arg\max_a Q\left(s', a; \theta_i\right); \theta_{i-1} \right) \middle| S_t = s, A_t = a \right] \tag{4.7}$$

Here, we notice that the maximum Q-value used in the target has been removed. Finally, we can see the target network with parameter $i-1$ deals with the quality of the actions, whereas, on the other hand, the action is determined by the Q-network with the parameters i. The steps involved in the DDQN can be summarized in the following steps.

- Initially, relying on some preceding calculations, the Q-network computes the qualities of $Q(s', a)$ for every potential action in the next states s'.
- The action selected for the next state s' is determined by the primary Q-network (i.e., $\arg\max_a Q(s', a)$).
- The Q-value of this selected action in the next state s' is then estimated using the target Q-network (i.e., $Q(s', \arg\max_a Q(s', a))$.

The idea here is that while the primary Q-network is more prone to overestimating Q-values, the target Q-network is updated less frequently, providing a more stable and reliable estimate of the Q-values. This separation helps to reduce the overestimation bias, leading to more accurate Q-value estimates and improving the learning stability of the algorithm.
The final Mean squared error (MSE) loss function of DDQN is now calculated in the equation (4.8).

$$\text{Loss} = \frac{1}{2}\mathbb{E}_{a' \sim \mu}\left[\left(y_i^{\text{DoubleDQN}} - Q(s, a; \theta_i) \right)^2 \right] \tag{4.8}$$

where,

- $y_i^{\text{DoubleDQN}}$ is the target value based on the double DQN target equation.
- $Q(s, a; \theta_i)$ is the Q-value predicted by the primary Q-network with parameters θ_i for the action a taken in state s.
- $a' \sim \mu$ represents that a' is sampled from a distribution μ and is calculated by taking action with the highest Q-value chosen by the primary Q-network.
- The expectation $\mathbb{E}_{a' \sim \mu}$ is taken over the distribution of a' based on the agent's experience.

From all the above equations and steps explained, now we can look at the DDQN algorithm.

Algorithm 2 Double Deep Q-Network (DDQN)

1: Initialize main network Q and target network Q_{target} with random weights
2: Initialize replay buffer D
3: **for** episode $n = 1$ to N_{max} **do**
4: Initialize state s
5: **for** $t = 1$ to T **do**
6: Choose action a using an exploration strategy based on Q
7: Take action a, observe reward r and next state s'
8: Store transition (s, a, r, s') in D
9: Sample a random batch of transitions from D
10: Set target $y = r + \gamma Q_{\text{target}}(s', \overset{\text{argmax}}{a} Q(s', a))$
11: Update main network weights by minimizing $(y - Q(s, a))^2$
12: **if** t mod target_update_interval $== 0$ **then**
13: Update target network weights: $\theta_{\text{target}} \leftarrow \tau\theta + (1 - \tau)\theta_{\text{target}}$
14: **end if**
15: Update state: $s \leftarrow s'$
16: **end for**
17: **end for**

Steps 12 and 13 in the above algorithm describe how the online network parameter updates the target network at a certain interval. In step 13, θ_{target} represents the parameters of the target network, θ represents the parameters of the main network, and τ is the small value between 0 and 1, which controls the rate of parameters update in the Q-network. The equation for the updating parameters is given by

$$\theta \leftarrow (1 - \tau)\bar{\theta} + \tau\theta \tag{4.9}$$

The target network's parameters are updated to be a weighted average of its current and main network parameters. This update helps the target network to keep track of the changes in the online network.

Double Deep Q-Network in Coverage Path Planning

DDQN was employed to train the model, enabling it to learn the task of covering target areas within the map, including obstacles such as NFZs and buildings. For this, the map with 5 channels is created and fed to the convolutional network. The first 3-channels belong to the map, one channel to the coverage grid, and one to the position of the UAV flying inside the map. Which is fed to the fully connected network. Convolutional and fully connected networks are connected through the ReLu activation function. The final layer of a fully connected network has the output size of the number of actions our UAVs can take with no activation function. This gives us the action-value function called $Q - values$ concerning each action for the given state input. From here, we follow two options to choose the actions for balancing the exploitation-exploration of the UAV in the environment. The exploitation, meaning exploiting the options of learned knowledge, can be done by choosing the greedy policy where we choose the index of the maximum Q-value, which corresponds to the action to be taken of the $Q - values$ we got as an output from the network. The greedy policy for the UAV is given in (4.10)

$$\pi(s) = \arg\max_{a \in A} Q_\theta(s, a) \tag{4.10}$$

However, during the training process of the UAV, it needs to explore the environment to decide which path is better for the drone to take for overall efficiency. To explore the environment, the agent needs to take random actions instead of taking the argmax of the Q-value. For this purpose, we use a softmax policy in the network's final output. The formula of the exploration for the UAV is defined as:

$$\pi(a_i|s) = \frac{e^{Q_\theta(s,a_i)/\beta}}{\sum_{\forall a_j \in A} e^{Q_\theta(s,a_j)/\beta}} \tag{4.11}$$

where $\beta \in \mathbb{R}$ is the temperature parameter and helps balance exploration and exploitation. Initially, the UAV explores more with higher β, resulting in more exploration. As $\beta \to 0$, the exploration tends to move towards the exploitation function described in the (4.11). The steps involved in DDQN are explained in detail with the pseudo algorithm below. Initially, we assigned the replay buffer and random parameters to the target network from the online network. Then, the UAV is assigned to a random position inside the map, which will be the initial position of the UAV, and the movement budget for the drone time limitation of flying is sampled. After that, the UAV will take steps to move inside the environment until the movement budget is 0 or the drone is landed, which is the condition for the terminal state. A new action $a \in A$ is sampled during each step based on the exploration-exploitation strategy. A reward and next state are observed based on the selected action, and the tuple of state, action, reward and next state is stored in the replay buffer D. Then the tuple of minibatch m containing state, action, reward and next state is sampled uniformly from the replay buffer for all the experiences stored in the buffer, where a reward is taken if the next state is terminal state; otherwise, the target value is calculated based on the equation (4.7). Loss is calculated using MSE loss, and parameters are updated with a gradient called Adam optimizer for the primary network. The soft update is performed for parameters so that small changes in the primary network are kept on track with the target network, which is done by using the equation (4.9). Finally, the movement budget is decremented for each episode, keeping

track of the maximum UAV movement steps. The training process continues through a series of episodes until the maximum number of episodes, denoted as N_{max}, is reached to train the model effectively. In addition, we save the best result in each episode to keep track of the learning process during the training process.

Algorithm 3 DDQN for Coverage Path Planning

Initialize replay buffer D, initialize θ randomly, $\bar{\theta} \leftarrow \theta$
for $n = 0$ to $N_{\max}$ **do**
    Initialize initial state s_0 with random starting position and sample initial movement budget b_0 uniformly from B
    while $b > 0$ and not landed **do**
        Sample action a based on (Equation (4.11)).
        Get reward r and observe next state s_0
        Store (s, a, r, s') in replay buffer D
        for $i = 1$ to m **do**
            Sample (s, a_i, r_i, s'_i) uniformly from replay buffer D
            $Y_i = \begin{cases} r_i, & \text{if } s'_i \text{ terminal} \\ \text{according to (Equation (4.7))}, & \text{otherwise} \end{cases}$
            Compute loss $L_i(\theta)$ according to (Equation (4.8))
        end for
        Update θ with gradient $\frac{1}{m} \sum_{i=1}^{m} L_i(\theta)$
        Soft update of $\bar{\theta}$ according to (Equation (4.9))
        $b = b - 1$
    end while
end for

4.2.2 Proximal Policy Optimization

Background

PPO optimizes policies in DRL with stability and efficiency. The main goal of the PPO algorithm is to find the policies that lead to higher rewards while limiting the huge changes in its policy with respect to the previous policy. It is designed to find better policies operating where actions lead to rewards. It used the iterative process to update its policy based on the collected experiences from the environment and gradually improve the decision-making process. The collected experiences are done iteratively and contain trajectories (sequences of states, actions, and rewards) using the current policies. The advantage function is estimated using the value functions and optimizing the policy through gradient-based methods. One of the main distinguishing features of PPO is its "clipped surrogate objective", which controls the magnitude of the policy updates, ensuring that the policy changes don't deviate too far from the previous policy. Due to this feature, the PPO algorithm contributes to stability in learning and prevents a policy collapse.
Here are some points describing the significance of the PPO algorithm in RL:

- Stability: PPO focuses on addressing concerns related to policy optimization instability seen in other algorithms. It introduces the clipped objective function, which helps prevent drastic changes in policy updates, resulting in smoother convergences and better stable training.
- Sample efficiency: PPO uses the collected data efficiently, and its update rule makes good use of collected data and avoids unnecessary updates, which prevents policy degradation.
- Compatibility with Complex Environments: PPO is one of the suitable algorithms for easy and complex environments. PPO can be implemented for various environments varying from simple discrete to complex high-dimensional continuous action domain environments. It makes PPO a versatile algorithm to be used in various real-world applications.
- Easy to implement: PPO is straightforward compared to alternate algorithms. It provides the right balance between simplicity and effectiveness.

Some fundamentals of PPO

The model learns through the PPO algorithm employing a combination of a policy network, a value network, and an advantage function. A brief description of the following topic is as follows:

- **Policy network**: The policy network in PPO is responsible for the agent's strategy for selecting the actions in different environment states. It takes the current state as an input and outputs the probability distribution over all the actions available for the environment. The policy network can be used for both discrete and continuous action space.
  In this project, we are dealing with discrete action spaces where the output of the policy network will be the probability distribution over the available action spaces, and the agent samples the action over this distribution. The goal of the

policy network is to map from the state to the actions that yield the maximum cumulative rewards.

- **Value network**: The value network is also another network used in PPO, where the network takes the state as an input and outputs the estimated value, which is also the value of the estimated cumulative reward that can be accumulated from being in that particular state. This estimation of the value can help the agent understand the long-term reward for the different states in the environment, helping the agent make a better decision.

  1. The state-value function $V(s)$ helps to estimate the expected cumulative rewards from being in the state s and following the agent's policy.
  2. The action-value function $Q(s, a)$ estimates the expected cumulative reward from taking the action a in state s and then following the agent's policy.

  The value function provides the value later used to calculate the advantage function, which is essential for policy updates in the PPO algorithm.

- **Advantage function**: The advantage function shows the benefit of taking the action in a particular state with respect to the expected value of that state. It shows how good or bad the action is compared to the average action in that state. The advantage function helps to update the policy more efficiently and guide the agent towards action that leads to a higher reward. The advantage function can be calculated using the Generalized Advantage Estimation (GAE), which considers immediate and future estimated rewards to provide a balanced measure of the advantage function.
  In PPO, the advantage function is calculated using the surrogate objective function, which helps in policy updates. This function ensures that the policy updates do not have huge updates preventing drastic policy changes.

Figure 4.4 shows the policy network, value network and the advantage function of current policy obtained from the Value function.

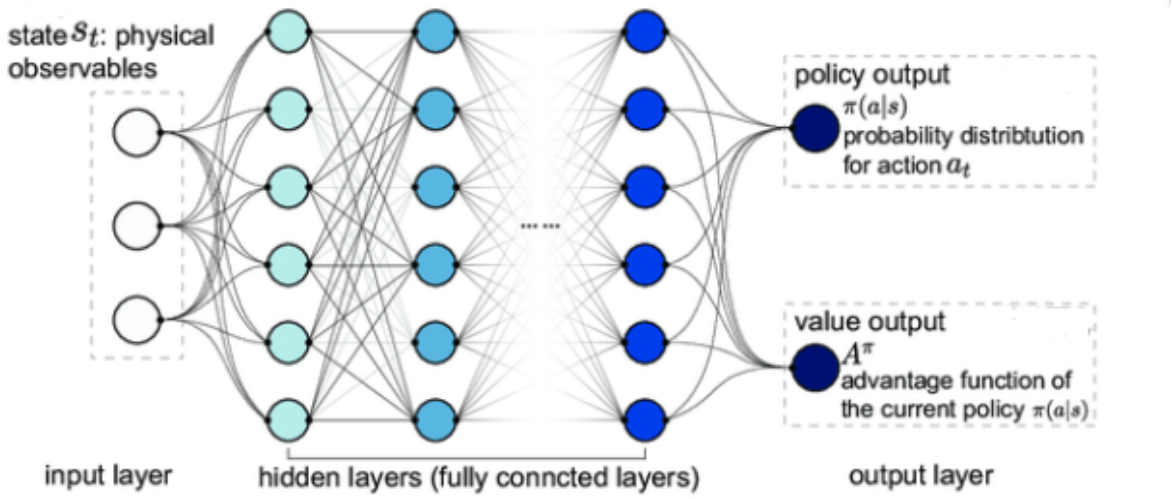

Figure 4.4: Schematic of the neural network structure for PPO algorithm [13].

The Introduction of Clipped Surrogate Objective Function

The clipped surrogate objective function is a pivotal component within the PPO algorithm, serving two fundamental purposes.

- We want to keep our policy updates smaller to increase the probability of converging towards the optimal solution.
- If we make big changes during the policy update, there are chances of falling "off the cliff", meaning getting the bad policy and taking a long time to converge and leading towards the policy, which can be hard to recover from.

Due to these reasons, we use the clipped surrogate objective function to update our policy function conservatively in the PPO algorithm. This helps to provide more stability and effectiveness to the training model. The basic idea is to measure how much policy changes in each update compared to the previous policy, and we will clip this ratio accordingly in the range of [1 - ϵ, 1 + ϵ]. Because of this, we can remove the incentive part if the currency policy update goes too far from the former one.
To understand the PPO algorithm, we need to understand its policy objective function, which is defined in the equation as:

$$L^{PG}(\theta) = \mathbb{E}\left[\log \pi_\theta(a_t|s_t) \cdot A_t\right] \tag{4.12}$$

where,

- $\log \pi_\theta(a_t|s_t)$ is the Log probability of taking that action at that state.
- A_t is the Advantage function, and if $A > 0$, this action is better than the other possible action at that state.

From the above equation, we can understand that the main idea is to take the actions that push our agent to accumulate higher awards and avoid the actions that are not good for the higher awards. But in this, there can be problems with step size, meaning how big updates we can make to change our policy so that it won't be too low to slow our training process or too high to "Jump off the cliff", meaning updating our policy in the way that it affects our policy negatively and brings too much variability in training. To address this problem, the use of the clipped surrogate objective function is used to constrain the policy change in a small range between [1 - ϵ, 1 + ϵ] as shown in equation (4.13)

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\right) \cdot \hat{A}_t\right] \tag{4.13}$$

where, $r_t(\theta)$ is the ratio function given by

$$r_t(\theta) = \frac{\pi_{\theta_{\text{old}}}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \tag{4.14}$$

where,

- $\pi_{\theta_{\text{old}}}(a_t|s_t)$ represents the probability of taking action a_t given state s_t according to the old policy parameterized by θ_{old}.

- $\pi_\theta(a_t|s_t)$ represents the probability of taking action a_t given state s_t according to the current policy parameterized by θ.
- The fraction of both gives the ratio probabilities between the old and current policies for action a_t in state s_t. This ratio replaces the logarithmic function we use in the policy objective function.
- If the value of $r_t(\theta) > 1$, the action a_t at state s_t is more likely to choose current policy action.
- If $r_t(\theta)$ value is between 0 and 1, the action a_t at state s_t is more likely to choose the action from the older policy.

Now, let's understand the concept of the above equation by dividing two different parts. i.e., clipped objective and unclipped objective.

By using equation (4.14) in (4.13) while excluding the clipped part, we get the **unclipped part** of the above equation:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\left(\frac{\pi_{\theta_{\text{old}}}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \right) \cdot \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right] \tag{4.15}$$

There are no constraints in equation (4.15). Therefore, we cannot limit the range of policy updates. Here, we will add the constraints part from the clipped part of the equation, which will penalize the extra changes in the range between [1 - ϵ, 1 + ϵ], which gives the complete equation for **clipped part**.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \right) \cdot \hat{A}_t \right] \tag{4.16}$$

The value of the ϵ is considered 0.2, according to the paper. The minimum function in the equation determines that we take the minimum of clipped and unclipped functions and establish the conservative estimate (a pessimistic approximation) for the unclipped objective function. Due to this value, we ensure that we are not considering too big changes in our policy updates because the new updated policy can not vary too much from our former policy. Finally, we can visualize it by using the graph model where the L^{CLIP} is drawn against the probability ratio r at a single term i.e., a single t in figure 4.5.

	$p_t(\theta) > 0$	A_t	**Return Value of *min***	**Objective is Clipped**	**Sign of Objective**	**Gradient**
1	$p_t(\theta) \in [1-\epsilon, 1+\epsilon]$	+	$p_t(\theta)A_t$	no	+	✓
2	$p_t(\theta) \in [1-\epsilon, 1+\epsilon]$	−	$p_t(\theta)A_t$	no	−	✓
3	$p_t(\theta) < 1-\epsilon$	+	$p_t(\theta)A_t$	no	+	✓
4	$p_t(\theta) < 1-\epsilon$	−	$(1-\epsilon)A_t$	yes	−	**0**
5	$p_t(\theta) > 1+\epsilon$	+	$(1+\epsilon)A_t$	yes	+	**0**
6	$p_t(\theta) > 1+\epsilon$	−	$p_t(\theta)A_t$	no	−	✓

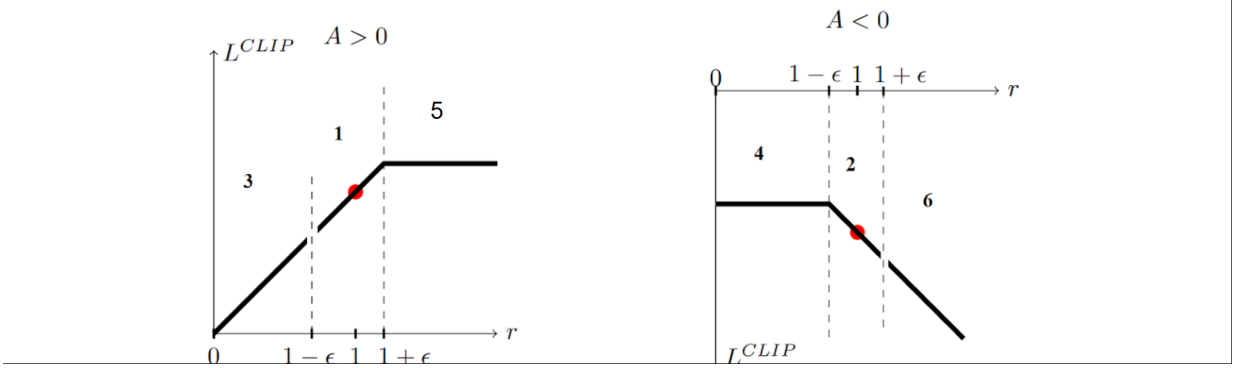

Figure 4.5: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., r = 1. Note that L^{CLIP} sums many of these terms [34].

From the above figure, we can explain six scenarios of the PPO clipped function at time t. Following are the six different conditions that can exist for this clipped function. NOTE: $A > 0$ means the current action function is better than the average of all actions in that state. $A < 0$ means the current action is worse than the average of all actions in that state.

- Case 1 and Case 2: Probability ratio within the range [1 - ϵ, 1 + ϵ]
  - Case 1: The advantage function is greater than 0. i.e., $A > 0$ determines the better action, and in this situation, we want to incentivize the probability of our policy of taking action in that state as the ratio is within the intervals range of [1 - ϵ, 1 + ϵ] in the graph.
  - Case 2: Here, the advantage function is less than 0. i.e., $A < 0$ determines the action is worse, and we don't want our policy to follow that action. Hence, we must discourage our current policy from taking such action in that state. As the ratio is inside the interval of [1 - ϵ, 1 + ϵ], we can decrease the probability of taking that action in our current policy.
- Case 3 and Case 4: Probability ratio below $[1-\epsilon]$
  - Case 3: As we can see from the graph, the probability ratio is less than $[1-\epsilon]$. However, the advantage function is also greater than 0. In this case, we need to use this action. Due to this, we would like to increase the probability of taking that action in that state.

- Case 4: In this case, we have the advantage function of less than 0. However, the probability ratio is already in the flat line area. So, we don't want to further decrease the probability of taking action in that state. Hence, we don't update the policy in such cases, and the gradient becomes 0 with no weight updates.

- Case 5 and Case 6: Probability ratio greater than $[1 + \epsilon]$
  - Case 5: In this state, the probability ratio is higher than the former policy (shown in the red dotted spot). The advantage function is higher than 0, meaning the action is also better. Meanwhile, we have a better policy as the probability function is already greater. However, we don't want to be too greedy to update too much as it's already in the flat line area. Hence, the gradient is 0 with no weight updates.
  - Case 6: The probability ratio is above the $[1 + \epsilon]$ range, and the advantage function is smaller than 0, meaning the action is worse than the average action in that state. So, in this case, the probability of taking that action in that state should be decreased.

We can now write this algorithm for the PPO from all the information above.

Algorithm 4 Proximal Policy Optimization (PPO)

Initialize policy π_θ with parameters θ
Initialize value function V_ϕ with parameters ϕ
Set hyperparameters: K (number of optimization epochs), N (batch size), ϵ (clipping parameter), γ (discount factor)
for $k = 1$ to K **do**
  Collect N trajectories using current policy π_θ
  Compute advantages $\hat{A}(s_t, a_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i} - V_\phi(s_t)$
  Optimize value function by minimizing: $\mathcal{L}_V(\phi) = \frac{1}{N} \sum_{t=1}^{N} (V_\phi(s_t) - \sum_{i=0}^{\infty} \gamma^i r_{t+i})^2$
  for each optimization step **do**
    Sample mini-batch of trajectories from collected data
    Compute surrogate objective:

$$L^{CLIP}(\theta) = \frac{1}{N} \sum_{t=1}^{N} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}(s_t, a_t), \text{clip} \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1 - \epsilon, 1 + \epsilon \right) \hat{A}(s_t, a_t) \right)$$

    Optimize policy by maximizing $L^{CLIP}(\theta)$ using gradient ascent
    Update old policy parameters: $\theta_{\text{old}} \leftarrow \theta$
  end for
end for

The hyperparameters are involved in the above algorithm, which is critical to properly tuning the PPO network for better performance. A brief explanation of these hyperparameters is as follows:

- K (Number of optimization epochs): This value denoted the number of times the algorithm repeats its optimization process. For each optimization, the algorithm

collects the experience for each policy obtained from the new data and then updates the value and policy based on the new data. A higher K value leads to more comprehensive policy updates but may also extend computation time.

- N (Batch size): The batch size N describes the number of experiences(states, actions and rewards) are collected in each optimization. The larger value of N can estimate a more accurate value of the policy gradients and increase the computation time.
- ϵ (Clipping parameter): The PPO clipping parameter prevents the policy updates from deflecting too far from the policy in the previous state. It checks policy updates and prevents large updates that might destabilise the learning process. The surrogate objecting function of the clipping range between [1 - ϵ, 1 + ϵ] and the value of ϵ is commonly used as 0.2, according to the paper.
- γ (Discount factor): The discount factor is used in many RL algorithms that determine the importance of future rewards compared to immediate rewards. When the agent in the environment receives a reward, it discounts the value of the reward by a factor of γ and adds it to the accumulated rewards. This helps the agent adjust its policy choices, giving more importance to immediate rewards for smaller γ, while a higher γ makes the agent consider long-term cumulative rewards more significantly.

Proximal Policy Optimization in Coverage Path Planning

This section will explain how the PPO algorithm has been implemented in our CPP environment. As in DDQN, we will first pass the image data of the map along with its movement budget and the drone's position inside the environment map, which becomes a 5-channel map. The 5-channel map is then passed to the convolution network and fully connected to the network with ReLu as an activation function between them. The PPO algorithm has two main neural networks: the actor and critic networks. The actor-network output is the probability distribution of the possible actions in the environment. It generates actions based on the current state and outputs a probability distribution over possible actions. This distribution encourages exploration by allowing the agent to try different actions and explore the environment. During the initial stages of learning, exploration is crucial for the agent to discover the optimal or near-optimal policy. Therefore, the actor network is responsible for exploration. Initially, the buffer memory is allocated where experiences of states, actions, rewards, next state and predictions are stored during the exploration part of the algorithm. After the initialization of the memory, random parameters for both networks are initialised. After that, the agent inside the environment takes the steps. The actor network predicts the probability distribution of the available actions, and the action is selected according to the probability. Based on the action, the drone moves inside the environment and reaches the next state. The trajectories of collected experiences are then stored in the memory buffer. Once the memory buffer is full, the stored trajectories are retrieved to train the model. Once we retrieve the trajectories of states, actions, rewards, next states, predictions and done flag, the states and next states are passed to the critic network to calculate the estimated value of the being in the particular state and next state, offering the agent an assessment of the expected cumulative reward from that states. Using the critic's value estimates, the agent

can exploit its current knowledge to make more informed decisions. The critic network helps the agent differentiate between states expected to lead to high rewards (exploitation) and states that might lead to unknown or potentially high rewards (exploration). It is important to note that while the actor network encourages exploration through its action sampling process, the critic network doesn't directly explore or generate actions. Instead, it provides the agent with valuable information about the potential value of different states, helping them make more informed decisions about which actions to take. Once the values of states and the next states are calculated using the critic network, the advantage function is calculated using GAE values. The GAE is calculated in two steps for better comprehension of the method. At first, we calculate the temporal difference. i.e., deltas and then GAE.

- **Calculate Temporal Differences (deltas):**
  - The temporal differences (Deltas[t]) are computed for each time step t using the following formula:

$$Deltas[t] = r_t + \gamma \cdot (1 - d_t) \cdot V_{t+1} - V_t \tag{4.17}$$

    where,

    - r_t is the reward at time step t.
    - d_t is a binary variable indicating if the episode terminates at time step t.
    - V_t is the estimated value of the state at time step t.
    - $V_t + 1$ is the estimated value of the state at time step $t + 1$.
    - γ is the discount factor.
- **Calculate Generalized Advantage Estimations (GAE):**

  The GAE values (GAE[t]) are computed in a backward recursive manner using the formula from (4.17) as follows:

$$GAE[t] = Deltas[t] + (1 - d_t) \cdot \gamma \cdot \lambda \cdot GAE[t + 1] \tag{4.18}$$

  Starting from the last time step and moving backwards through time, this formula recursively updates the GAE values based on the temporal differences and the GAE value of the next time step.
- **advantages[t] = GAE[t]**

  The advantages (advantages[t]) are obtained directly from the GAE values for each time step t. The calculated advantages are then used in the policy update step of PPO. The key idea is that GAE provides a more informed estimation of advantages by considering the temporal relationships between rewards and value estimates. This can lead to more stable and effective policy updates during training, ultimately improving the performance of the RL agent.

The algorithm for PPO for Coverage Path Planning is given below:

Algorithm 5 PPO Algorithm for Coverage Path Planning

Initialize buffer memory D, initialize parameters θ for policy and ϕ for value function
Set hyperparameters: N_{max} (number of steps), M (batch size), ϵ (clipping parameter), γ (discount factor), b (movement budget)
for $n = 0$ to $N_{\max}$ **do**
  Initialize initial state s_0 with random starting position and sample initial movement budget b_0 uniformly from B
  while $b > 0$ and not landed **do**
    Sample action a based on prediction probability distributions of actions
    Get reward r, prediction p and observe next state s'
    Store (s, a, r, s', p) in buffer memory D
  end while
  for $i = 0$ to m **do**
    Collect M trajectories for current policy π_θ
    Compute deltas $Deltas[t] = r_t + \gamma \cdot (1 - d_t) \cdot V_{t+1} - V_t$
    Compute Generalized Advantage Estimation function(GAE):

$$GAE[t] = Deltas[t] + (1 - d_t) \cdot \gamma \cdot \lambda \cdot GAE[t + 1]$$

    Compute advantages $\hat{A}[t] = GAE[t]$
    Optimize value function based on ((4.19))
    for each optimization step **do**
      For each trajectory from the buffer memory
      Compute surrogate objective function based on ((4.20))
      Optimize policy by maximizing $L^{CLIP}(\theta)$ using gradient ascent
      Update old policy parameters: $\theta_{\text{old}} \leftarrow \theta$
    end for
  end for
end for

Once the advantages function is calculated, update the value function in the critic network by calculating the loss between predicted and true cumulative rewards to approximate the state-value function better.

$$\mathcal{L}_V(\phi) = \frac{1}{N} \sum_{t=1}^{N} (V_\phi(s_t) - \sum_{i=0}^{\infty} \gamma^i r_{t+i})^2 \tag{4.19}$$

Then, the surrogate objective function is calculated using equation (4.14) and (4.16).

$$L^{CLIP}(\theta) = \frac{1}{N}\sum_{t=1}^{N} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}[t], \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}[t]\right) \tag{4.20}$$

After that, each trajectory is selected, and the policies for each trajectory in the actor network are updated using the clipped surrogate function. Finally, the cycle of collecting experiences, calculating advantages, updating the policy and value function, and evaluating performance is repeated with a specified number of steps, helping the policy to converge or reach a desired level of performance.

4.2.3 MuZero

MuZero is an advanced RL algorithm developed by DeepMind. It was built as an advanced version of the AlphaZero, which DeepMind also built. MuZero is built upon the success of AlphaZero and is designed to excel in environments with incomplete information, making it suitable for a wide range of real-world applications. MuZero combines model-based RL, MCTS, and neural networks to learn and plan effectively in complex, uncertain domains. Its versatility and ability to adapt to various scenarios make it a significant advancement in AI.

A brief history about MuZero

DeepMind, known for pioneering AI and ML work, introduced AlphaGo in 2016. It was the first AI software to defeat professional world champion Go player. The game occurred between Lee Sedol and AlphaGo, where AlphaGo managed to defeat Lee Sedol in four out of five games. This event is a great feat in the AI world because this is the first time AI has been able to evaluate all the possible moves. Their value and importance were able to calculate long-term future effects in the game. AlphaGo used the policy network to evaluate the next move and the value network to predict the winner of the game. AlphaGo, being RL, learned its own mistake by playing several times with itself, learning from its mistake, and finally being able to defeat the world champion. During game 2, move 37, played by AlphaGo against Lee Sedol, surprised the world because it was understood as a bad move at that time by AlphaGo. Still, it turned out to maximise the value and be a better policy to achieve better cumulative rewards. Hence, it proves AlphaGo's ability to think in depth about future steps in the game. The algorithm AlphaGo was fed with the standard rules of the Go game, its domain knowledge, and the human data to learn from, making it possible to train and achieve top-level performance that can even beat the best in the game. However, providing a set of rules for many real-world environments is difficult, making it hard to implement in real-world problems. Later on, in 2017, DeepMind introduced a better version of AlphaGo as an AlphaGo Zero, and in 2018, AlphaZero. AlphaZero was able to play against itself without any human data or domain knowledge and was also able to master other games like chess and shogi in addition to the game Go. The ability to master the three very complex games with one algorithm was a huge achievement. However, in 2020, DeepMind introduced a very sophisticated algorithm named MuZero. Figure 4.6 illustrates the developmental progression of MuZero, tracing its evolution from AlphaGo to the current state of MuZero. MuZero mastered games like go, chess, shogi and atari. The big thing about MuZero was the ability to master these games without any provided human data, domain knowledge, or known rules. That means MuZero, after playing against itself, learns the game's rules. After playing several times, depending on the game's environment, it masters the game and can maximise the awards. This hints at MuZero's ability to be used as a general-purpose algorithm, as it doesn't need a set of rules to understand the environment and achieve the task for better results.

Figure 4.6: Evolution of MuZero from AlphaGo [8].

Background

MuZero is a groundbreaking RL algorithm introduced by the DeepMind. It is designed to learn how to play and make decisions in complex environments without prior knowledge of the game rules or dynamics. It is also a general-purpose algorithm due to its versatility in adapting to various domains, including board games, video games, and many real-world problems. To achieve this high level of performance on different platforms, MuZero combines several techniques to enable one to learn effectively in an unknown environment. The techniques used in MuZero are briefly described below:

- **Model-based RL:** MuZero is a model-based RL algorithm because of its feature of learning a model of the environment's dynamics, which includes how actions affect the system's state and how rewards are received. This contrasts with model-free learning, where an agent directly learns a policy or value function from interacting with the environment.
- **MCTS:** Like its predecessor algorithm AlphaZero, MuZero also uses MCTS for planning and decision-making. MCTS is a popular algorithm in AI for game decision-making, planning and optimizing problems. MCTS builds a search tree that explores possible sequences of actions, evaluates their consequences using the learned model, and then selects actions that lead to better results. MCTS can balance between exploration and exploitation to gradually improve its decision-making process.
- **Value and policy networks:** In addition to MCTS, MuZero uses a deep neural network for two critical functions. i.e., the value network and the policy network. The value network helps estimate the expected future rewards from the given state, helping MuZero evaluate the importance of different states inside the environment. The policy network predicts the probability distributions of actions that should be taken in a given state. Both networks are updated through the RL process to improve their accuracy over time.
- **Rollouts with the learned model:** Another important feature of MuZero is using the rollouts for its learned model to simulate possible futures. Rollouts in MuZero provide additional data for training the value and the policy networks. These simulated trajectories are generated using the MuZero algorithm's internal model. After going through these rollouts; the simulated internal model helps MuZero decide the better actions to take that will result in promising rewards in the future. Hence, this allows MuZero to help achieve the optimal policy.
- **Self-play** In MuZero, self-play plays a critical role where the model learns by interacting in the environment itself. It begins with an untrained model and engages in simulated scenarios, making decisions, predicting future states and updating its understanding of the environmental dynamics. During this process, MuZero uses the MCTS algorithm to explore potential actions and outcomes, striking a balance between exploration and exploitation. Data is collected as the agents interact, covering game states, actions and outcomes. This data is then used to train MuZero's neural networks, including the prediction network, value network, and policy network. Through iterative cycles of self-play, data collection, and training, the model improves its decision-making abilities and predictive accuracy. This self-play helps

MuZero perform well in complex environments and master various scenarios autonomously.

Advantages of MuZero Algorithm

Muzero has several notable benefits and advantages. A few of them are mentioned below:

- Generalization: Because of its feature of requiring no human data, domain knowledge and known rules, MuZero can adapt to different environments and tasks without prior knowledge, making it suitable for real-world applications where learning from scratch is required.
- Sample efficiency: MuZero is Model-Based RL and uses search-based planning like MCTS. This combination allows it to simulate and plan in the environment using a learned internal model. Due to this, fewer interactions with the environment exist to achieve strong performance and planning without depending on trial-and-error exploration.
- Versatility: MuZero's framework can be applied to many unknown environments in various domains, from classic board games to complex video games and even real-world problems like path planning and coverage areas and many other fields.
- Adaptive exploration: MuZero uses the combination of value estimates, policy networks and MCTS to explore the environment effectively. It can adapt its exploration strategy based on the uncertainty in different states, reducing the risk of getting stuck in suboptimal solutions.

Fundamental Process of MuZero

MuZero's working process happens in several iterative steps. At the beginning of the MuZero algorithm, the agent interacts with the environment and tries to model the environment by taking several steps and storing that data in a replay buffer. The steps can terminate if it reaches the terminal point. This can be known as self-play in the MuZero algorithm term. After self-play, MuZero starts its training process by updating the neural network weights, producing an improved neural network version for better learning.

The Three Neural Networks Used in MuZero

MuZero consists of three primary neural networks. A breif explanation of the network is given below:

- Representation network(h):
  1. Representation is also denoted as 'h' and takes in the current state of the environment as input and produces an internal representation of the state.
  2. This network helps in encoding the current state information, including relevant objects, positions, and other important features, into a compact and informative form that can be used for planning and decision-making.

3. The representation network typically performs feature extraction from input data and lowers the dimensionality compared to the input data. This dimensionality reduction is essential to create a more compact and manageable representation that can be used efficiently by MuZero's planning and decision-making components.

- Dynamic network(g):
  1. The dynamic network is also denoted by 'g' and is responsible for modelling the environment's transition dynamics. It is responsible for predicting the next step in the environment and the immediate reward.
  2. It takes the current state representation produced by the representation network and action as inputs and predicts the next state and immediate reward.
  3. The dynamic network helps MuZero simulate the consequences of different actions without interaction with the real environment.
- Prediction network(f):
  1. The prediction network, denoted as 'f', is used to predict the value(V) and the policy(Π) from the internal state representation.
  2. The prediction network takes in the current internal representation produced by the representation network and produces two sets of predictions:
     a) **Value function(V):** This predicts the agent's expected long-term reward from the current state.
     b) **Policy(π):** This predicts the probability distribution over possible actions from the current state, indicating that actions are most likely to lead to favourable outcomes.

These three networks work together to enable MuZero to learn how to make decisions to get better results through a combination of model-based planning and model-free RL. The algorithm alternates between collecting experience data through self-play and using that data to train and update these networks. Over time, MuZero becomes capable of making increasingly better decisions in a wide range of environments. Figure 4.7 shows the basic difference between its predecessor AlphaZero and the MuZero algorithm.

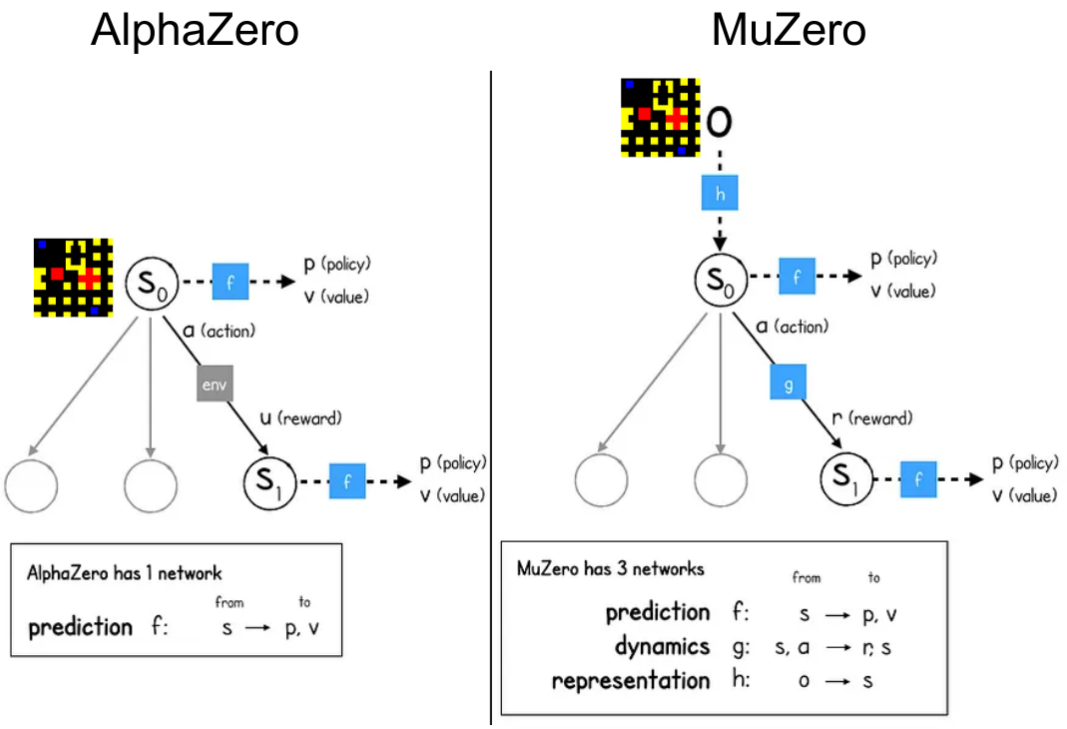

Figure 4.7: Difference between AlphaZero (on the left) and MuZero (on the right). This figure illustrates AlphaZero using one neural network for prediction. In contrast, MuZero uses three combinations of three neural networks for its internal mapping, taking action and predicting value and policy [12].

The algorithm AlphaZero uses one network to predict value and policy during its MCTS. It finds the policy with the probability distribution over all the moves and the value which estimates the future rewards. This is done for all the unexplored leaf nodes and, as a result, assigns the estimated value to all the positions with respect to each action. Then, it is backfilled up in the whole tree until the root node, giving the root node an idea about future value for being in that state.
In MuZero, the representation network(h) maps the environment's current state. Because MuZero doesn't have the luxury of having a set of rules in the environment, it works in hidden representation to plan ahead in tree search. The dynamic network(g) takes the current hidden state s, chooses action a, and rewards r. Meanwhile, the prediction network(f) of MuZero takes the current state in each hidden state and predicts the value and policy. This is done for each unexplored lead node and backfilled until the root node.

Learning Process of MuZero in Coverage Path Planning

The steps involved in learning the agent inside the environment for the CPP take place with the iterative process of MCTS. Then, the immediate rewards are received after taking action for each episode and training the networks to learn based on the loss functions. The schematic diagram of MCTS in MuZero is shown in figure 4.8.

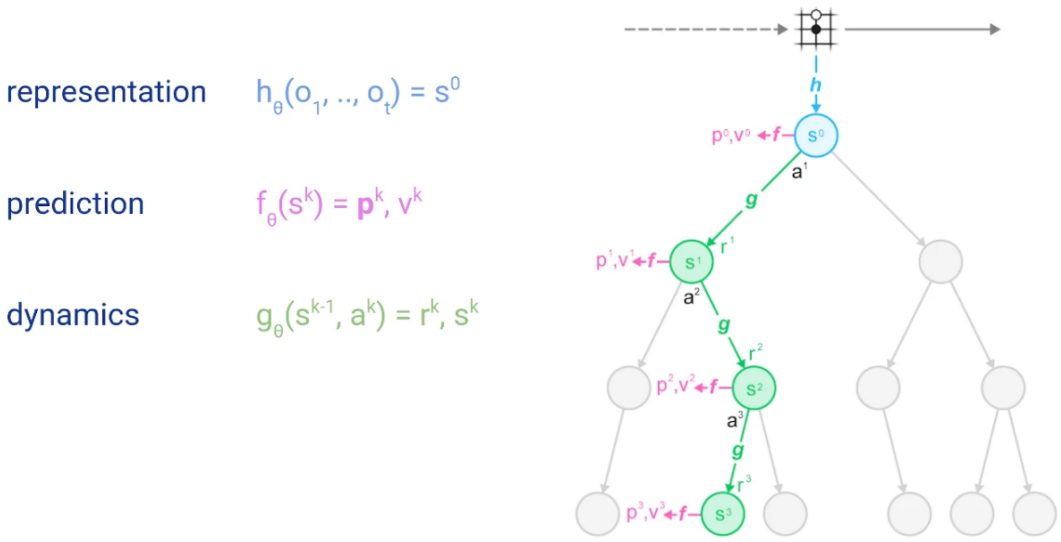

Figure 4.8: Schematic representation of the steps involved in MCTS in the MuZero algorithm. The square dot with a black dot in the middle represents the environment mapping with the current state [33].

There are three basic steps involved in the MCTS process.

- Simulation: From the above figure, we start at the root node. The root node can be the initial start of the tree where the representation network maps the current state of an agent or, in our case, the drone. Before sending the data to the root node, we preprocess our map, creating a 5-channel input described in section 4.1.5 of this thesis. Each circle in the figure 4.8 represents the corresponding state. During the MCTS process, the agent visits each state to analyze the most promising action to take. Once the representation network(h) maps the current state, it is shown in the figure as a blue circle. Each state's scoring function $U(s, a)$ is assigned to compare the actions a. The value of $U(s, a)$ is calculated by adding the prior estimation value and the estimated value function. The formula of $U(s, a)$ is given as:

$$U(s, a) = v(s, a) + c \cdot p(s, a) \tag{4.21}$$

  The variable c represents a scaling factor, which is used to ensure that the impact of the prior information decreases as our value estimate becomes increasingly precise. Every time we move inside the tree by taking action from one state to another, the visit count $n(s, a)$ is increased for the given state and the action, which is later used to calculate the "Upper Confidence Bound (UCB)" c which will help for the selection of the action. The expansion process continues until the leaf node can no longer be expanded, and prior estimates and value estimates are updated every time and stored in the node.

- Expansion: After evaluating the node, meaning calculation of prior and value estimate, the node can be marked as expanded. We can add a child node to that node if possible. Once the expansion is done, we move to the next step, backpropagation.

- Backpropagation: In the backpropagation of MCTS in MuZero, the estimated value is evaluated until the root node. The value of each node is obtained by calculating the mean value of all the nodes below it. Due to this averaging process, the UCB formula keeps getting accurate over time, helping the MCTS converge eventually.

During the MCTS process, we also calculate the transitional rewards. The transitional rewards are calculated based on the prediction network output of policy and value estimation. The reward is observed in every transition of the state. The UCB score with rewards parameter is shown in the equation as:

$$U(s,a) = r(s,a) + \gamma \cdot v(s') + c \cdot p(s,a) \tag{4.22}$$

where $r(s,a)$ is the reward during the transition from the state s by choosing action a with the discount factor γ. The hyperparameter γ describes the importance of the rewards in future states. Once we calculate the value of $U(s,a)$, we further normalize the value of UCB to ensure all the different nodes are in comparable scale and evenly distributed. It helps balance exploration and exploitation aspects and improves numerical stability during the selection phase of MCTS. Normalizing UCB values helps MCTS maintain a balanced exploration strategy, ensuring that both promising and less-explored nodes have a fair chance of being selected for further exploration. The updated equation of UCB with normalization is given in the equation:

$$U(s,a) = \frac{r(s,a) + \gamma \cdot v(s') - q_{\min}}{q_{\max} - q_{\min}} + c \cdot p(s,a) \tag{4.23}$$

where,

1. q_{min} is the minimum observed Q-value among the child nodes of the current node in the MCTS search tree. It represents the worst-case scenario for an action based on the available information.

2. q_{max} is the maximum observed Q-value among the child nodes of the current node in the MCTS search tree. It represents the best-case scenario for an action based on the available information.

Each complete MCTS cycle gives the action, the value estimate of that state and the policy distributions of action from the state from which actual action a_t can be applied to reach state s_{t+1} from state s_t. This process can be repeated until the environment terminates, generating a series of episodes. Each episode has policies from where the action is selected to go to the next state and so on. The visual representation is given in figure 4.9.

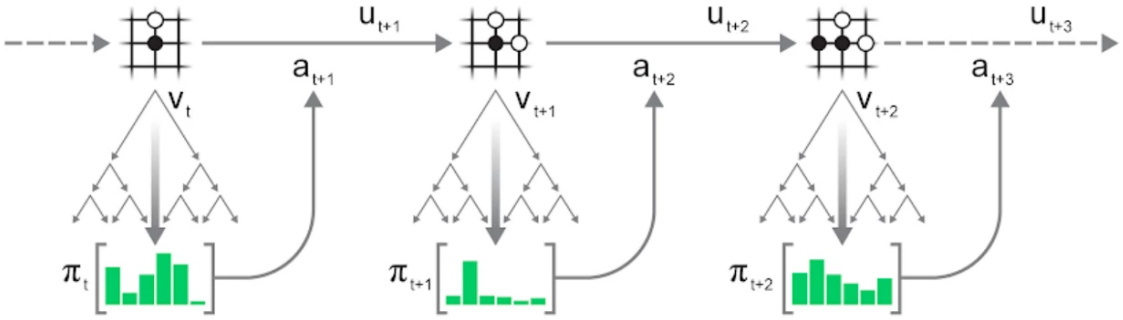

Figure 4.9: Schematic representation of Episode generated by each MCTS cycle giving the action to choose and obtaining the actual reward in the environment after taking the action obtained from the MCTS process [33].

The temperature function is used to balance between the exploration and exploitation in the action selection. The temperature function decides if the selection of action should be greedy, meaning the exploitation of the action with the most visits, or the action should be taken by sampling to its visit count n(s, a), where it will explore the unvisited nodes. This degree of exploration is controlled by applying some temperature parameters, which are shown in the equation:

$$p(a) = \left(\frac{n(s,b)}{\sum_b n(s,a)} \right)^{\frac{1}{t}} \tag{4.24}$$

where the value of t = 0 results in greedy action and t = inf results in uniform sampling actions. Therefore, as the number of iterations increases, temperature decreases, and actions become greedier.

Training Phase of MuZero

Once the agent encounters the terminal phase inside the environment, the training phase starts with the collected experiences or trajectories generated by the MCTS cycle. During the MuZero model learning, all the trajectories generated by the MCTS are sampled and unrolled along with the MuZero model. During this process, all three neural networks from the MuZero play their parts, shown in figure 4.10.

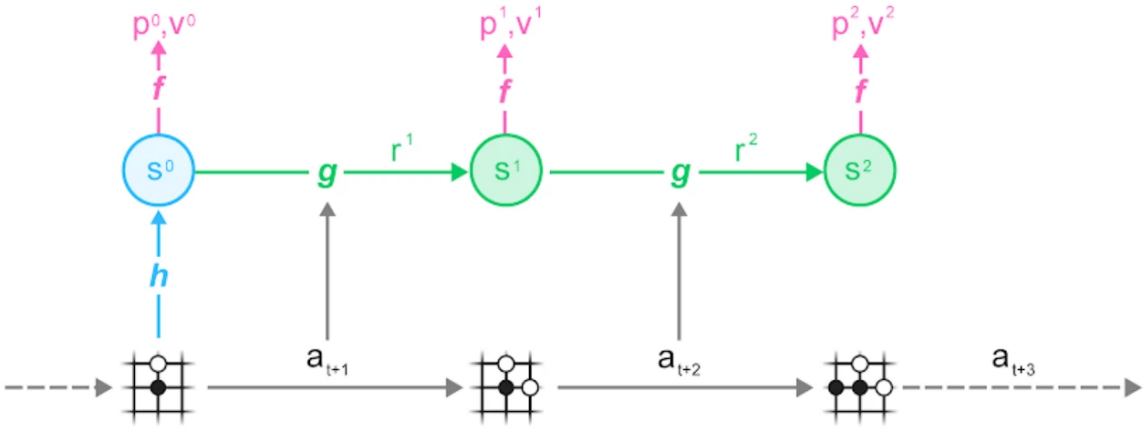

Figure 4.10: Sampling the trajectories created by MCTS during the training phase and comparing them with the true values to train the model concerning their losses [33].

Figure 4.10 shows that the representation network h uses the maps of our environment where a drone flies to create the hidden state s. Once there is the hidden state s^0, the dynamic network g uses the state mapped by the representation network to go from state s^t to s^{t+1} based on the action a_{t+1}. During this transition, there is also reward r^t achieved by the agent and helps to move forward, unrolling all the states involved in that trajectory. At the same time, during the unrolling, each state generated, the prediction network f calculates the policy p^t and value v^t based on the state s^t. These generated values and policies are used by the UCB formula and aggregated by the MCTS.

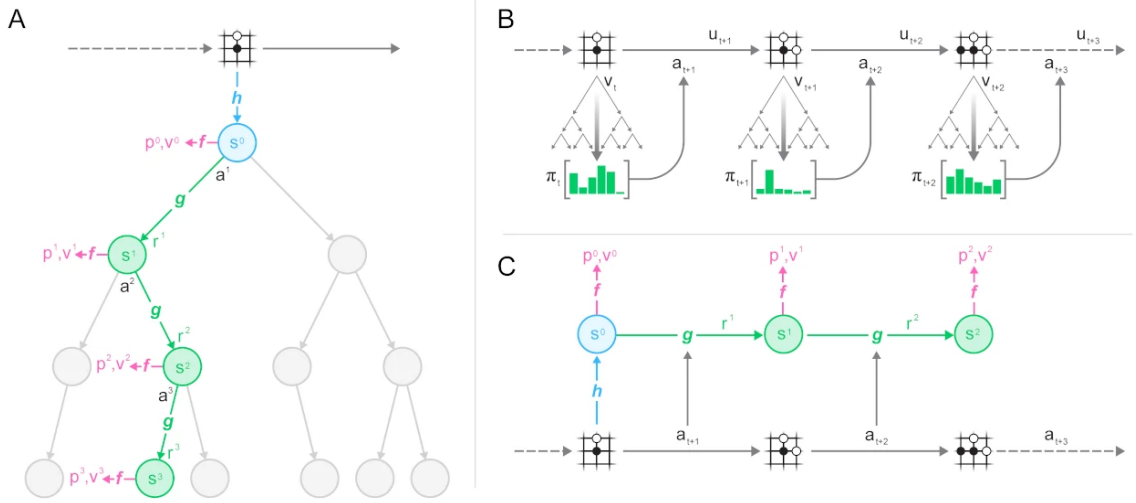

Figure 4.11: The complete diagram of the MuZero algorithm. The diagram shows the major steps involved in the MuZero algorithm. i.e., [A]. Monte-Carlo Tree Search, [B]. Episode generation and [C]. Training phase [33].

Figure 4.11 gives the overall representation of the MuZero process, shown in three different states. In the above figure, part A represents the MCTS, part B represents the episode

generation by each MCTS cycle, and part C represents the unrolling of the trajectory created. From Part B and Part C of the figure, it can be better understood where each episode is compared to its training part, where the corresponding losses are calculated as all the neural networks trained by comparing the results of episodic generations of the MCTS factors. The losses for each factor are calculated by considering the following parameters.

- Policy: Policy loss are calculated by considering the visit count inside the MCTS and the predicted policy loss from the prediction Neural network.
- Value: Value loss is calculated using MSE between a value from the discounted sum of rewards and target network estimate and the value from the prediction neural network.
- Reward: Reward loss is calculated between the trajectory estimation of reward and the dynamics function estimation rewards.

This process of running MCTS and saving them in a replay buffer for all the trajectories until they encounter the terminal state generates many trajectories. Using that trajectory to train the neural network for the training part and running the MCTS for fresh data to retrain the neural network to minimize losses continues until the network converges to provide better-expected results.

5 Evaluation

In this thesis work, we are trying to explore three different algorithms and find one of the most suitable algorithms for a UAV environment for the coverage of the grid in the fastest way, considering the limited time we have to fly our UAV drone. Learning the optimal policies in RL is still a non-deterministic approach. Much trial and error must be done with the hyperparameters setting in the RL algorithm. Finding the right hyperparameter setting for these algorithms is crucial; it takes extensive work and resources to find the best results. For this environment, reinforcement algorithms like DDQN have shown very promising results. However, we will still try to figure out more hyperparameter settings for the DDQN algorithm and the other two algorithms i.e., the PPO and MuZero algorithms. Hence, the following results are conducted to explore, understand and find the optimal hyperparameter setting for respective algorithms and compare their results.

5.1 Simulation Setup and Metrics

The UAV is flying in grid environments. The "Manhattan32" scenario has a $32 * 32$ grid size with the starting and landing zones at the map's top left and bottom right corners. In addition to that, some irregular-sized buildings are present, as well as a no-fly zone. The random targets are created with 3-8 shapes covering around 20-50% of the total map. The drone is assigned to cover the target area in a limited time, controlled by the movement budget. The movement budget is reduced by 1 for each drone step, and the movement budget being zero before landing means the battery is empty and the drone fails to land on time. As the time constraints are considered, full coverage of the target area is unlikely. Hence, it makes less sense to consider the traditional metric like path length. Instead, the Coverage Ratio (CR) is considered, which is the ratio of covered target cells to the initial target cells at the end of the episode. Coverage Ratio And Landed (CRAL) is considered, which becomes zero if the drone fails to land and equals the CR if it lands. The main idea behind CRAL metrics is to combine the drone's high coverage ability with its successful return to the landing zone within flight time constraints. Standardizing performance to a value between 0 and 1 allows for performance comparisons across varying scenarios featuring randomly generated target zones. In addition, the movement ratio, which is the ratio of how much of the movement budget is used to achieve the target area coverage compared to the initial budget, is considered for the metrics. We have also shown Boundary Counters (BC), Landing Attempts (LA), Cumulative rewards, and Successful Landings (SL) for the result analysis.

5.2 Experiments

This section will start experiments for the algorithms individually and compare their results within different hyperparameter settings. Before beginning the experiments, hyperparameters involved in optimizing the algorithms in these experiments are briefly discussed in the following section. Information about the system configurations on which these experiments are carried out is given in the table below.

No.	Components and Packages	Configurations
1	Processor	11th Gen Intel(R) Core(TM) i9-11900K @ 3.50 GHz
2	Installed RAM	32.0 GB (31.9 GB usable)
3	System Type	64-bit operating system, x64-based processor
4	Windows	Windows 11 Pro
5	Installed GPU	NVIDIA GeForce RTX 3060 - 8GB
6	CUDA	8.1
7	CuDNN	cuDNN version 8100
8	Python	3.10.9
9	Tensorflow	2.10.0
10	Tensorboard	2.10.1
11	Keras	2.10.0
12	Numpy	1.22.3

Table 5.1: Information about System Configurations and Packages Used

5.2.1 Double Deep Q-Network(DDQN): Experiments and Results

In the DDQN algorithm, several hyperparameters are involved. They can be tweaked to optimise the algorithm and the behaviour of the algorithm to have a better understanding of DDQN on this project. However, due to the limited resources available and the huge time consumption to run the project, it is impossible to thoroughly study all the hyperparameter effects. Due to this, we have only checked the effects of batch size, learning rate, and global and local scaling with time in our experiments.

Experiments with different learning rates

In this section, we will run the DDQN algorithm in our coverage area path planning environment with the following hyperparameter values, keeping most of them constant except the learning rate value to find the best suitable learning rate for this project in our results.

No.	Parameter Type	Value
1	Batch size	128
2	Discount Factor	0.95
3	Global Map scaling	3
4	Local Map scaling	17
5	Flatten layer size	4001
a	Learning rate	3e-3
b	Learning rate	3e-4
c	Learning rate	3e-5

Table 5.2: Parameters used in DDQN architecture for the training and testing coverage path planning on "Manhattan32" with varying learning rates.

Learning Rate	CR		CRAL		BC		LA		MR		SL		Rewards	
	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test
3e-3	0.03	0.03	0.01	0.01	80.0	82.6	0.28	0.28	0.91	0.94	0.12	0.10	−266.37	−233.47
3e-4	0.70	0.69	0.66	0.67	1.13	3.05	1.04	1.06	0.92	0.93	0.94	0.95	0.70	0.69
3e-5	0.69	0.69	0.67	0.67	0.68	1.69	1.12	1.11	0.91	0.91	0.97	0.97	45.37	43.13

Table 5.3: DDQN results for CPP with three different learning rates

Below is the graph plotted for various metrics for uses of the DDQN algorithm on coverage path planning.

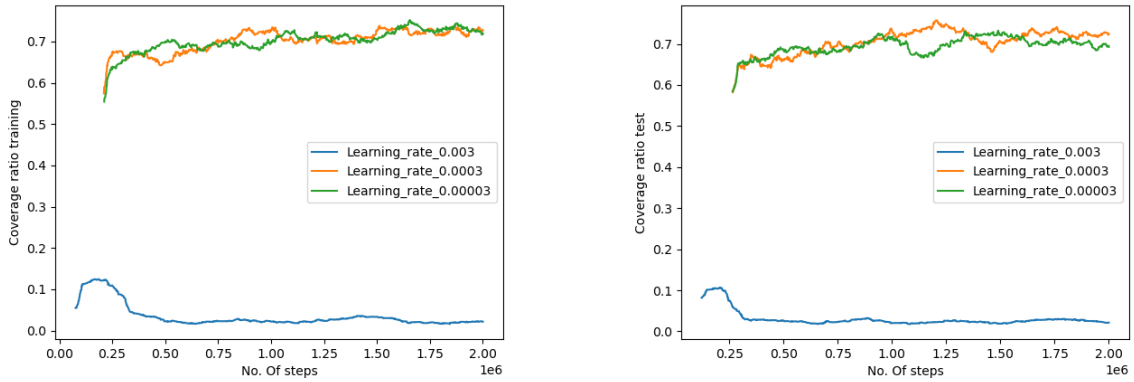

Figure 5.1: CR for DDQN algorithm for three different learning rates.

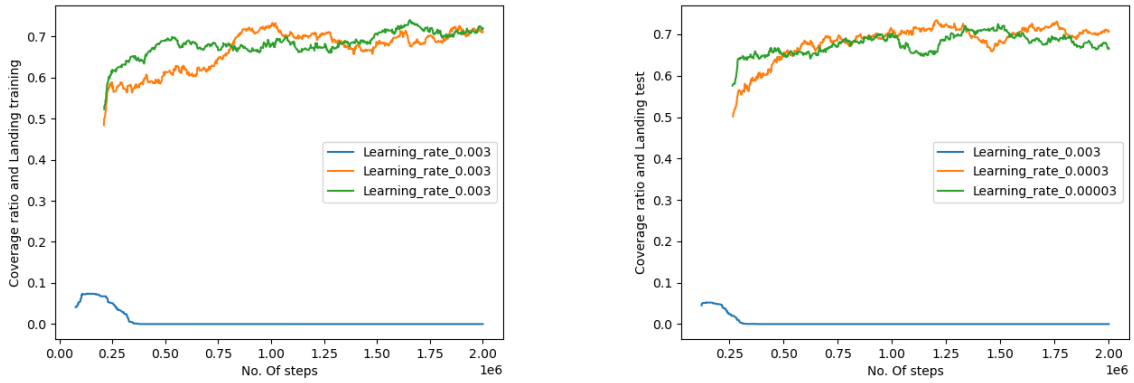

Figure 5.2: CRAL for DDQN algorithm for three different learning rates.

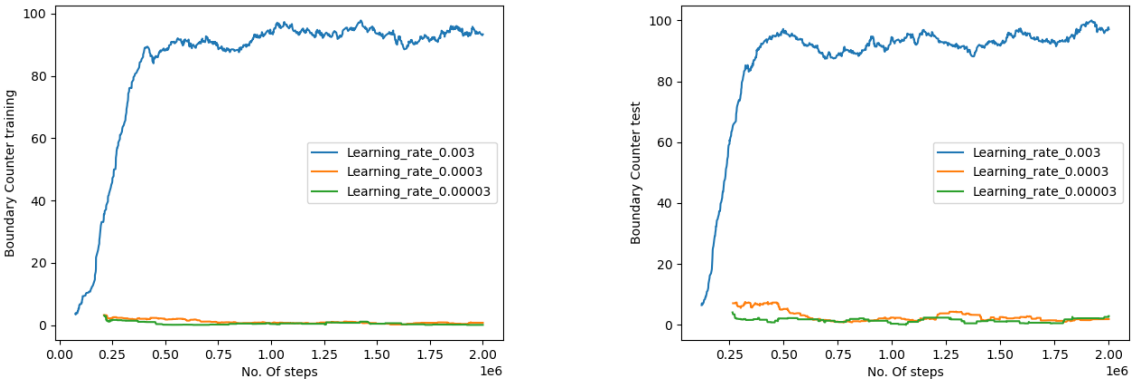

Figure 5.3: Boundary counters for DDQN algorithm for three different learning rates.

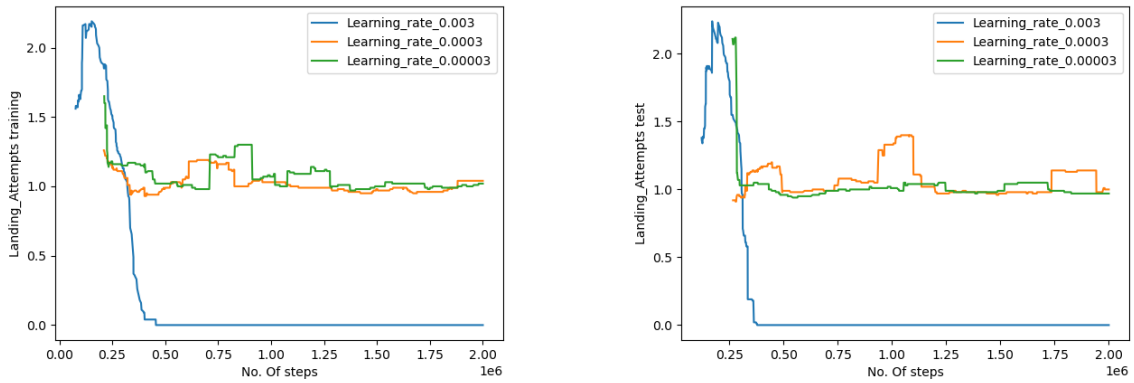

Figure 5.4: Landing attempts for DDQN algorithm for three different learning rates.

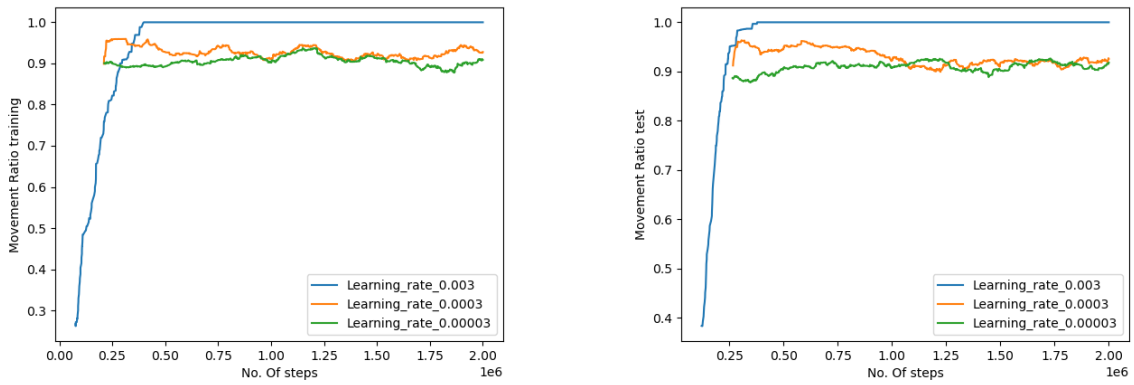

Figure 5.5: Movement ratio for DDQN algorithm for three different learning rates.

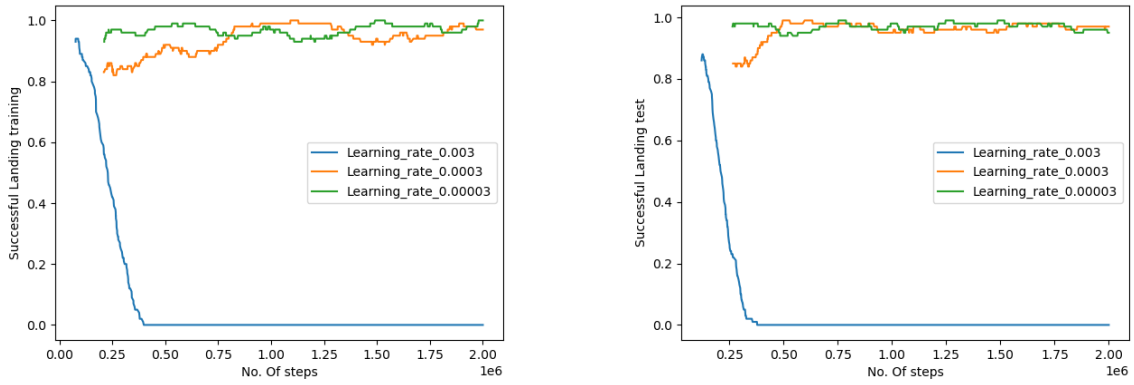

Figure 5.6: Successful landing for DDQN algorithm for three different learning rates.

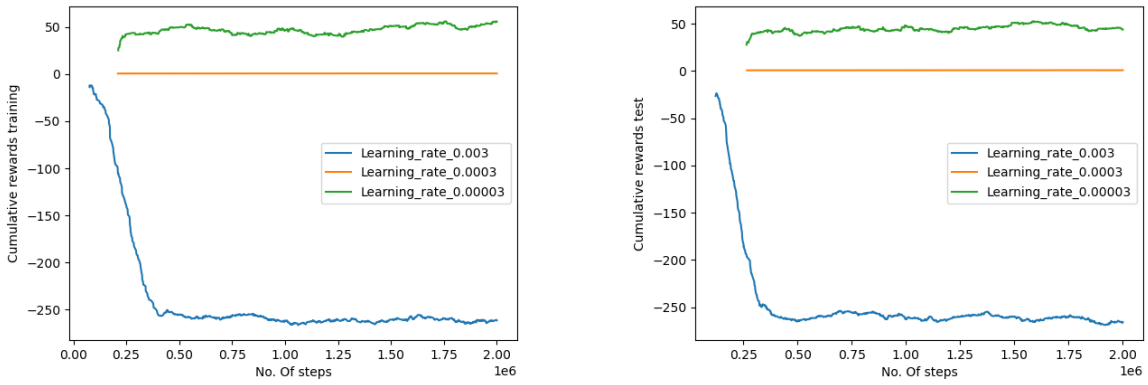

Figure 5.7: Cumulative rewards for DDQN algorithm for three different learning rates.

These graphs cover the results of DDQN conducted for 2 million steps in the coverage path planning environment. In these results, we tried to analyse the effect of learning rate to find the best possible learning rate to obtain better rewards, coverage rate and the ratio between coverage rate and landing attempts. The three learning rates applied are $3e-3$, $3e-4$, and $3e-5$.

From the results of these graphs, we can conclude that the learning rate with a value of $3e-3$ significantly impacts results such as CR and CRAL, showcasing a substantial reduction compared to the other two learning rates, which exhibit similar and closely aligned outcomes. Learning rates $3e-4$ and $3e-5$ give the coverage area and landing ratio around 70%, which is very good compared to results for learning rate $3e-3$. In addition, the results of $3e-3$ have the highest boundary counters with the lowest and most successful landing attempts. Finally, even if it is evident that the learning rate $3e-3$ is not suitable for this environment, we have to look into the cumulative rewards and choose between $3e-4$ and $3e-5$ and from the graph, it shows that $3e-5$ accumulates more rewards and hence is better suitable for this environment. Due to this, we can have a better clarification about the learning rate $3e-5$ providing the best results. The movement ratio of the learning rates $3e-4$ and $3e-5$ indicates values below 1, showing that our drone is flying within our movement budget range and landing successfully. Meanwhile, the learning rate $3e-3$ is 1 in most graphs, showing the drone couldn't be on time and gets terminated.

Experiments with different batch sizes

In this section, different batch sizes have been experimented with to see the effect of batch size on the algorithm and to find the most suitable batch size for the coverage path planning environment. We also choose the best learning rate from the above results as $3e-5$. In this section, three different batches are applied to the algorithm, i.e., 32, 64 and 128, and results are observed as batch sizes should be chosen carefully because of their effect on the algorithm's stability, variances, sample efficiency, training time and memory consumption.

No.	Parameter Type	Value
1	Learning rate	3e-5
2	Discount Factor	0.95
3	Global Map scaling	3
4	Local Map scaling	17
5	Flatten layer size	4001
a	Batch size	32
b	Batch size	64
c	Batch size	128

Table 5.4: Parameters used in DDQN architecture for the training and testing of coverage path planning on "Manhattan32".

Batch Size	CR		CRAL		BC		LA		MR		SL		Rewards	
	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test
32	0.68	0.66	0.66	0.66	0.65	0.95	1.07	1.05	0.87	0.87	0.97	0.98	45.41	45.46
64	0.67	0.68	0.66	0.66	0.62	1.14	1.12	1.17	0.88	0.88	0.98	0.97	46.22	44.92
128	0.69	0.69	0.67	0.67	0.68	1.69	1.12	1.11	0.91	0.91	0.97	0.97	45.37	43.13

Table 5.5: DDQN results for coverage path planning with three different batch sizes

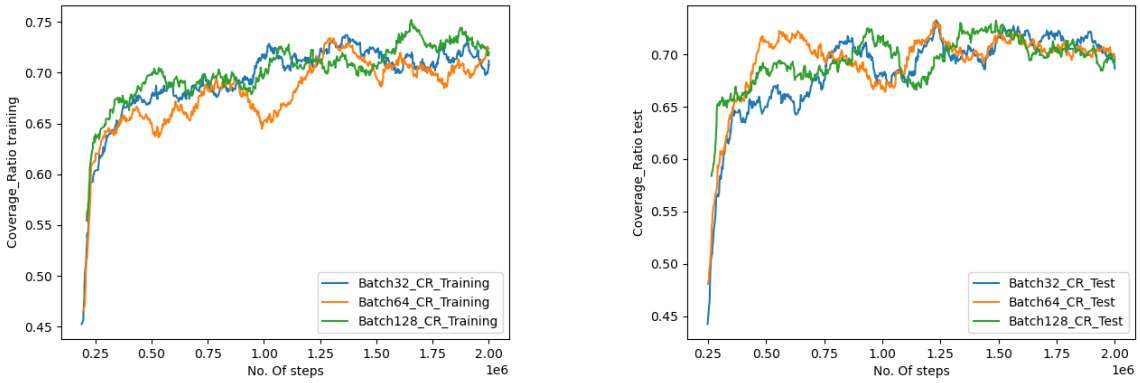

Figure 5.8: CR for DDQN algorithm for three different batch sizes.

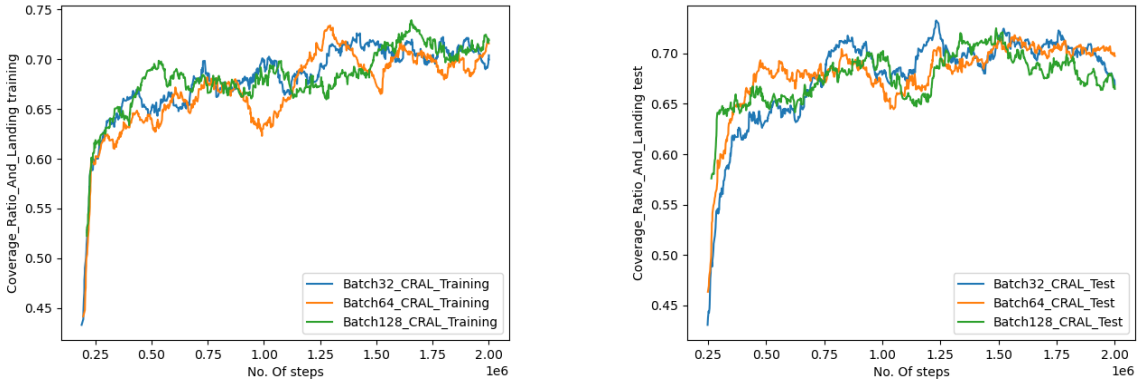

Figure 5.9: CRAL for DDQN algorithm for three different batch sizes.

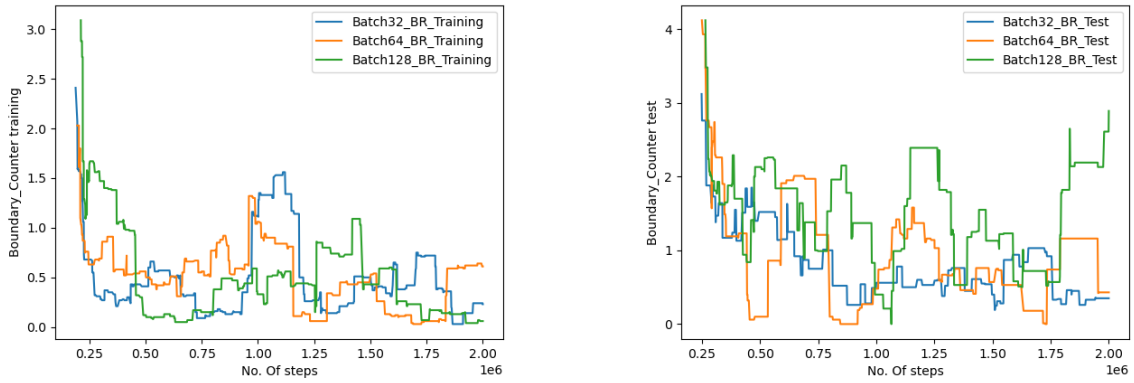

Figure 5.10: Boundary counters for DDQN algorithm for three different batch sizes.

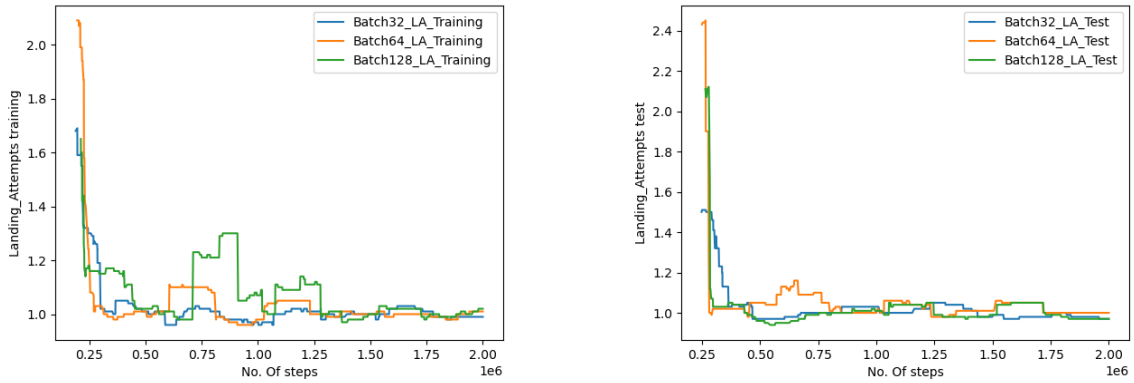

Figure 5.11: Landing attempts for DDQN algorithm for three different batch sizes.

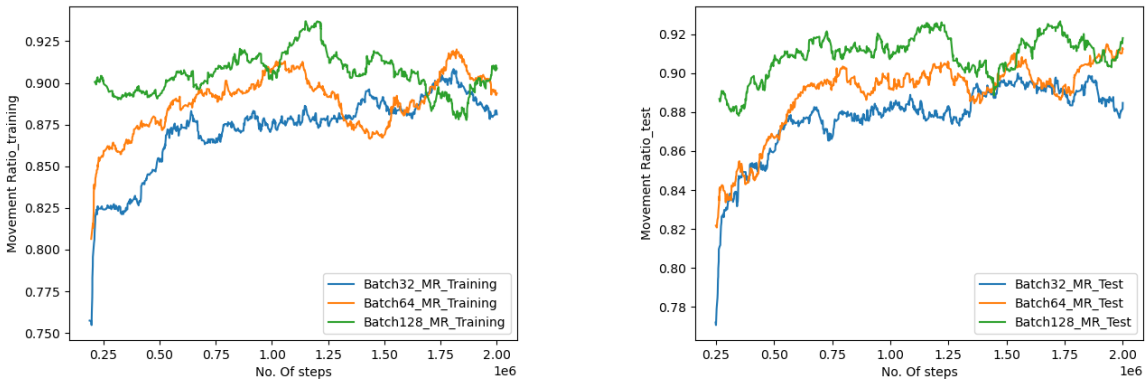

Figure 5.12: Movement ratio for DDQN algorithm for three different batch sizes.

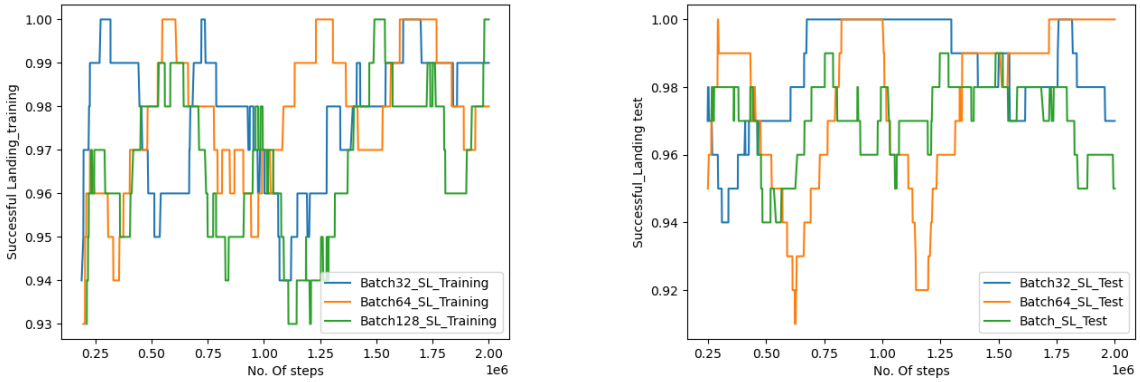

Figure 5.13: Successful landing for DDQN algorithm for three different batch sizes.

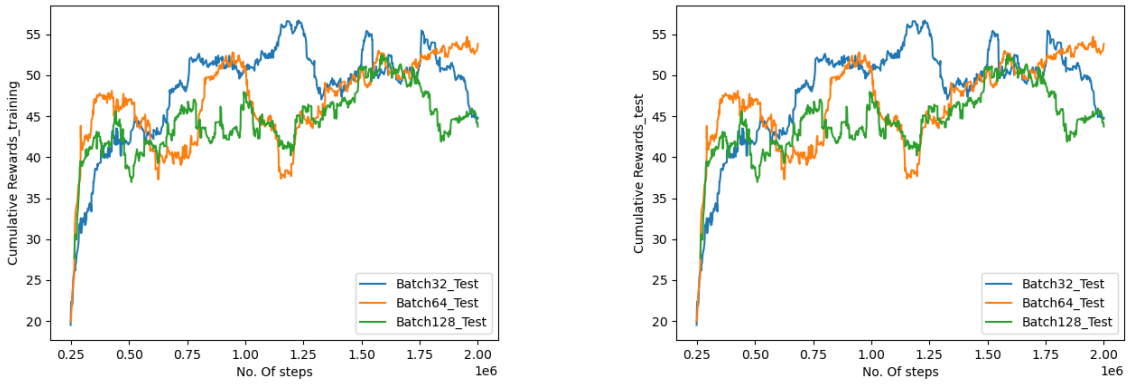

Figure 5.14: Cumulative rewards for DDQN algorithm for three different batch sizes.

From these above results, we can observe that in most scenarios, the results are very close to each other for all three different batch sizes. This suggests the robustness of the DDQN algorithm against the coverage path planning environment. However, batch size 128 is still slightly better than the other two. It shows that batch size doesn't significantly impact the algorithm's ability to learn and make decisions effectively for the hyperparameters above. In this situation, smaller batch sizes can be advantageous from a computation efficiency perspective as they require less memory and computation, making them potentially more suitable for resource-constrained environments. On the other hand, selecting a larger batch size helps faster convergence as each batch contributes more to the gradient estimate. The movement ratio is also less than 1, showing that our drone is landing within its movement budget. Hence, 98% of the time, the drone lands successfully before its battery is finished with a coverage area of approx. 67%.

In summary, the performance of the DDQN in this particular environment for the coverage path planning for "Manhattan32" is very effective and yields optimal results. The algorithm DDQN, having the replay buffer, is stable and robust as it samples the data from the replay buffer. It can also address the over-estimation of Q-Values as it separates the action selections from the evaluation of its values. Training DDQN in a complex environment helps to overcome challenges associated with the high dimensionality of states and action spaces. This enables the algorithm to learn efficiently and accurately helping the model to represent complex relationships within the environment.

5.2.2 Proximal Policy Optimization(PPO): Experiments and Results

In the PPO algorithm, several hyperparameters are involved. They can be tweaked to optimise the algorithm and the behaviour of the algorithm to have a better understanding of PPO on this project. However, due to the involvement of high dimensional state and action spaces, storing and manipulating large amounts of data in RAM was resource-intensive. In addition to that, due to the many layers involved in the neural network model, the complexity of the model increases. Hence, we cannot thoroughly train the two million steps that were initially intended. As limited resources are available, all the PPO algorithms are trained in around 125k steps in this experiment, and the results are compared. Instead of tweaking many hyperparameters, a few are tweaked, like batch sizes and learning rates. We perform these experiments after understanding that the remaining hyperparameters are nearly optimal. The results of the experiments with their corresponding hyperparameters are shown in the following section.

Experiments with different Learning rates

In this section, we will run the PPO algorithm in our coverage area path planning environment with the following hyperparameter values, keeping most of them constant except the learning rate value to find the best suitable learning rate for this project in our results.

No.	Parameter Type	Value
1	Batch size	64
2	Discount Factor	0.95
3	Global Map scaling	3
4	Local Map scaling	17
5	Flatten layer size	4001
a	Learning rate	3e-04
b	Learning rate	3e-05
c	Learning rate	3e-06

Table 5.6: Parameters used in PPO architecture for the training and testing coverage path planning on "Manhattan32" with varying learning rates.

Learning Rate	CR		CRAL		BC		LA		MR		SL		Rewards	
	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test
3e-04	0.14	0.12	0.00	0.00	56.37	50.70	0.62	0.87	0.95	0.96	0.07	0.05	−186.35	−187.56
3e-05	0.31	0.32	0.00	0.00	10.97	12.91	0.95	0.93	0.96	0.95	0.05	0.05	−109.33	−109.90
3e-06	0.09	0.08	0.01	0.01	12.06	10.85	3.16	3.71	0.74	0.72	0.33	0.36	−109.37	−113.35

Table 5.7: PPO results for coverage path planning with three different learning rates

Below are the graphs plotted for the results in the table above, visually illustrating the results for the different metrics.

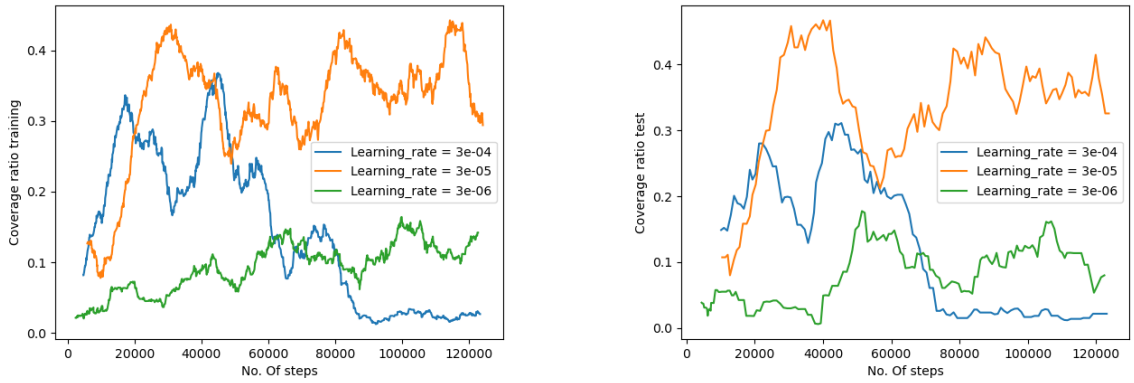

Figure 5.15: CR for PPO algorithm for three different learning rates.

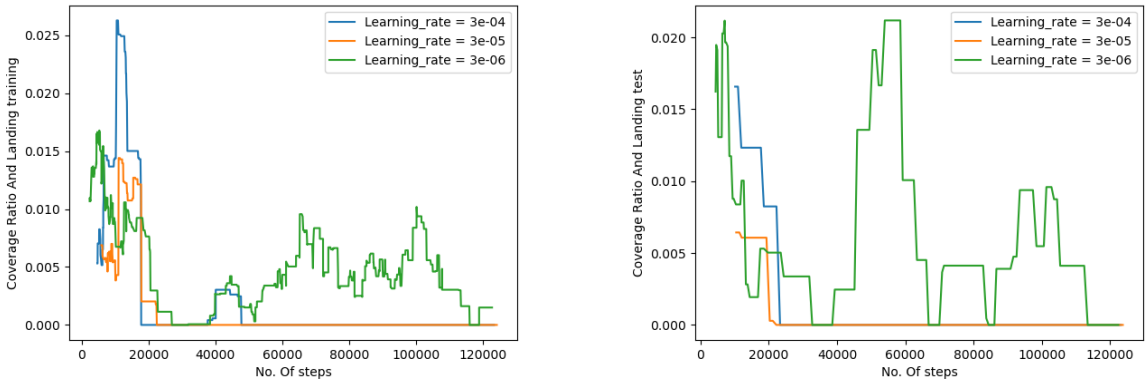

Figure 5.16: CRAL for PPO algorithm for three different learning rates.

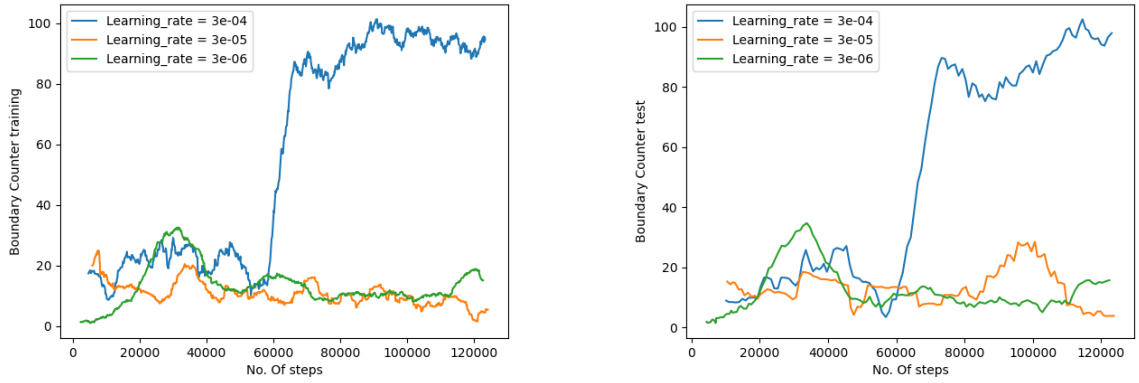

Figure 5.17: Boundary counters for PPO algorithm for three different learning rates.

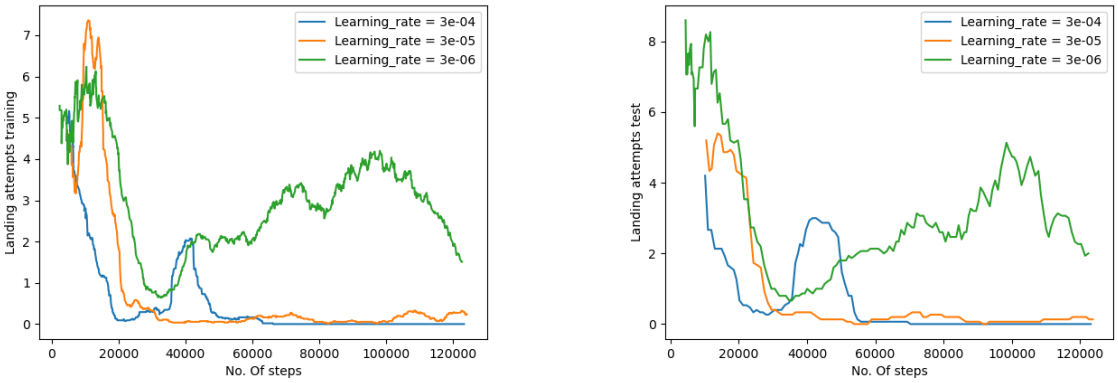

Figure 5.18: Landing attempts for PPO algorithm for three different learning rates.

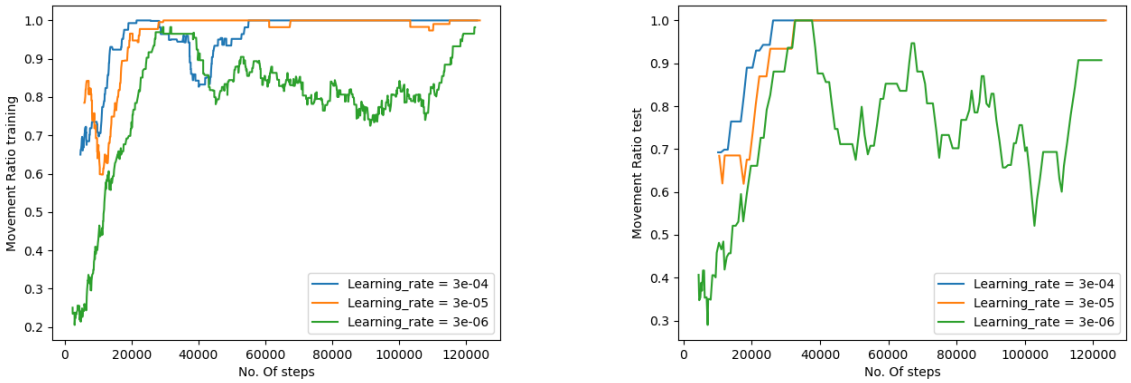

Figure 5.19: Movement ratio for PPO algorithm for three different learning rates.

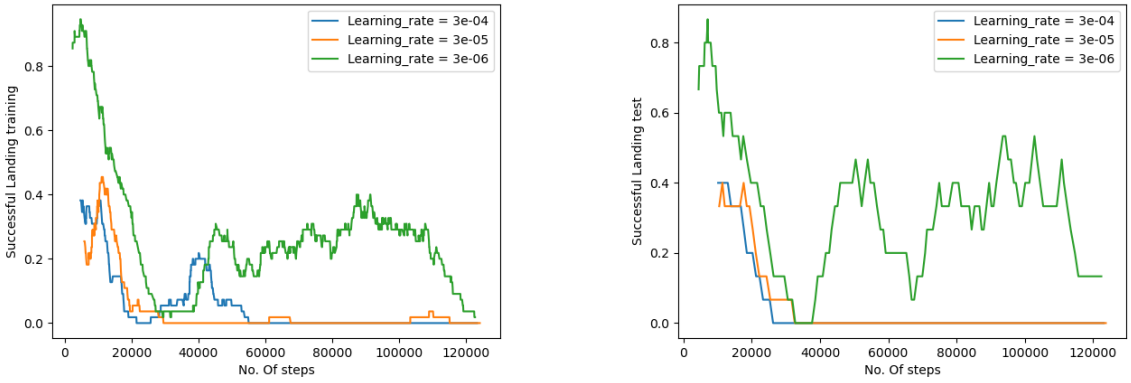

Figure 5.20: Successful landing for PPO algorithm for three different learning rates.

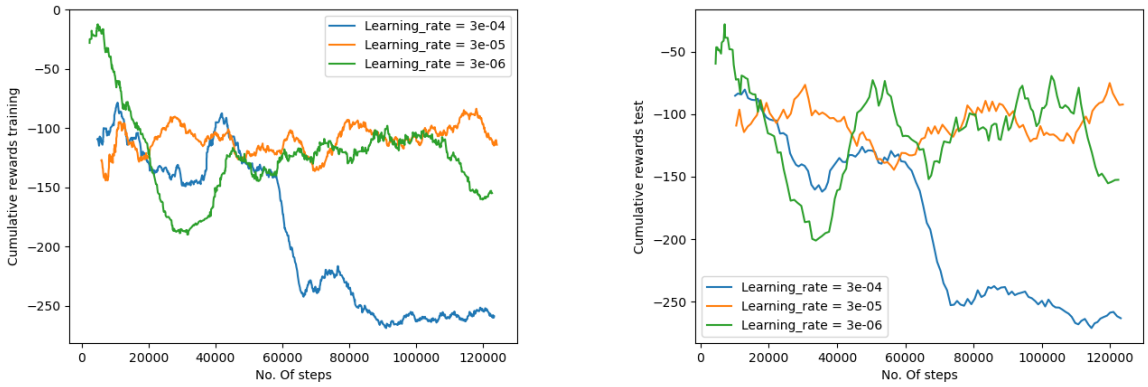

Figure 5.21: Cumulative rewards for PPO algorithm for three different learning rates.

In the above graph plots, we plotted different metrics for the PPO algorithm against the number of steps. Different parameters are tweaked and finally adapted before trying these experiments for different learning rates i.e., 3e-04, 3e-05 and 3e-06. However, since we could run only up to 120K, the plotted results do not show a significant effect of the PPO algorithm on the coverage path planning algorithm. Despite the limited number of steps, we can still derive some conclusions from the results we obtained. It can be seen that the learning rate 3e-05 has the best CR among other learning rates, which covers up to 32% of the randomly generated target during the test run of the algorithm.
Given that the CR is still acceptable, the drone fails to land within the given budget. However, the CRAL is not acceptable for the limited training steps. In addition to that, it can be observed that the movement ratio is very high and successful landing is very low. This shows that the algorithm needs to train more steps before it learns to fly. It does not have a high value for boundary counters and movement ratios nor a low cumulative reward. However, it still started to reverse back into collecting better rewards, and among them for learning rates, 3e-05 has to have the best rewards performance, which shows that all learning rates 3e-05 are best-suited learning rates for this experiments for given hyperparameters combinations.

Experiments with different Batch Sizes

This section will run the PPO algorithm in our coverage area path planning environment with the following hyperparameter values. We will keep most of them constant except the batch sizes to find the best suitable for this experiment.

No.	Parameter Type	Value
1	Learning rate	3e-05
2	Discount Factor	0.95
3	Global Map scaling	3
4	Local Map scaling	17
5	Flatten layer size	4001
a	Batch size	32
b	Batch size	64
c	Batch size	128

Table 5.8: Parameters used in PPO architecture for the training and testing coverage path planning on "Manhattan32" with varying Batch sizes.

Batch Sizes	CR		CRAL		BC		LA		MR		SL		Rewards	
	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test
32	0.13	0.12	0.01	0.01	18.09	16.76	3.58	3.65	0.77	0.73	0.27	0.32	−119.28	−113.07
64	0.31	0.32	0.00	0.00	10.97	12.91	0.95	0.93	0.96	0.95	0.05	0.05	−109.90	−109.33
128	0.21	0.21	0.00	0.00	15.07	14.52	1.51	1.65	0.91	0.90	0.12	0.13	−123.20	−123.22

Table 5.9: PPO results for coverage path planning with three different batch sizes

Below are the graphs plotted for the results in the table above, visually illustrating the results for the different metrics.

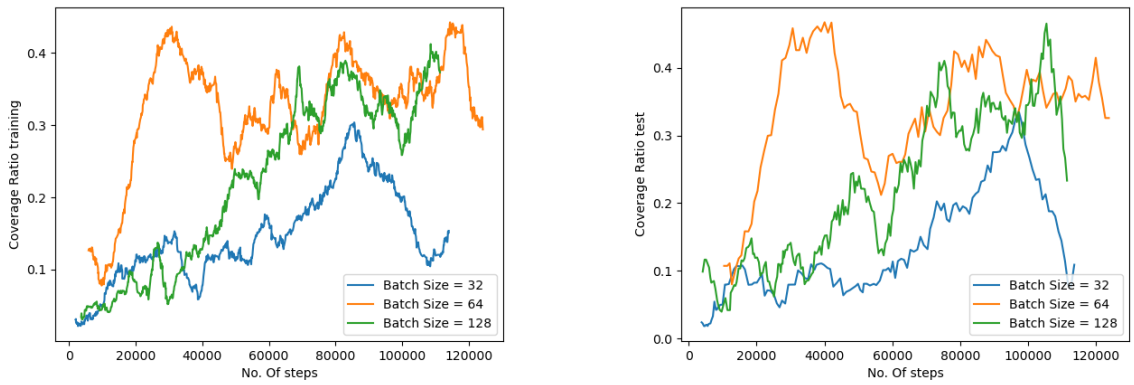

Figure 5.22: CR for PPO algorithm for three different batch sizes.

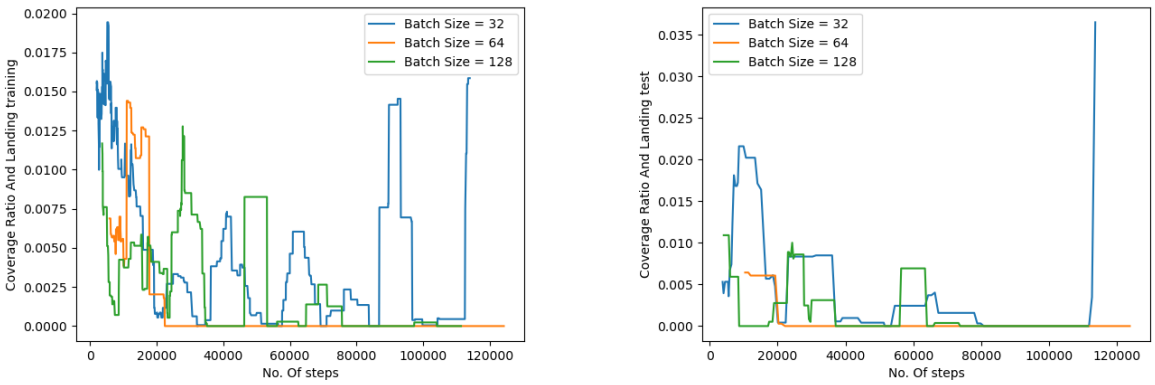

Figure 5.23: CRAL for PPO algorithm for three different batch sizes.

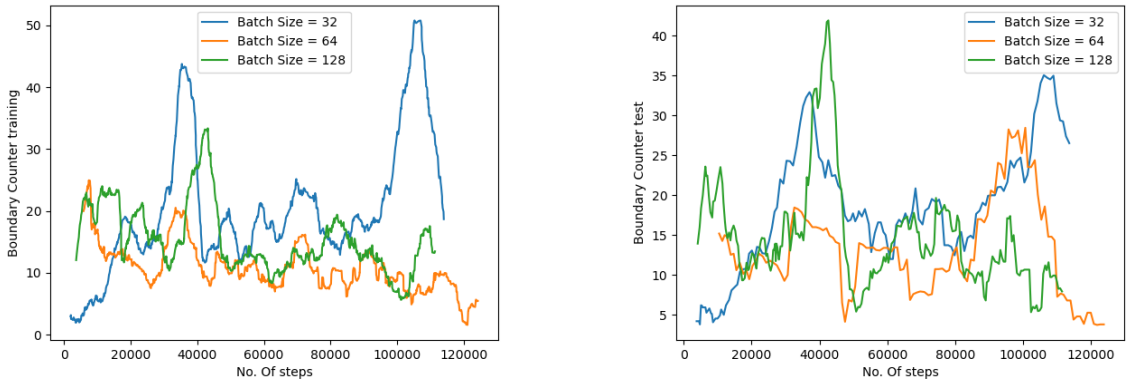

Figure 5.24: Boundary counters for PPO algorithm for three different batch sizes.

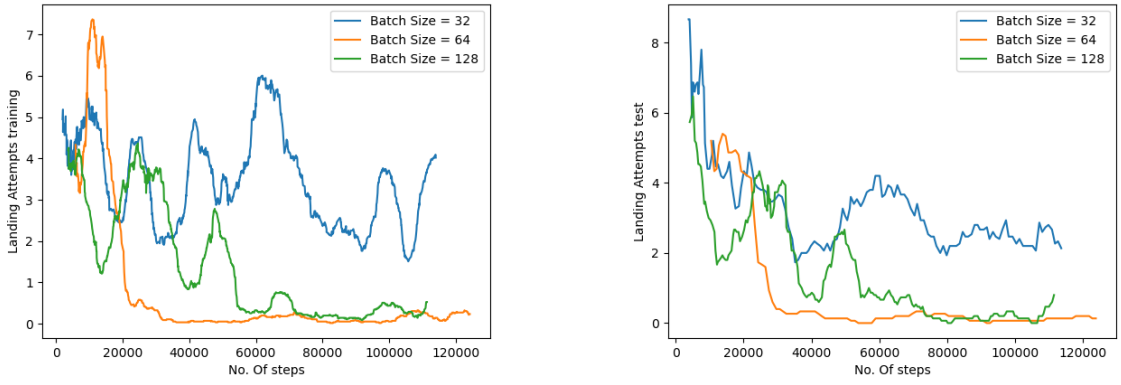

Figure 5.25: Landing attempts for PPO algorithm for three different batch sizes.

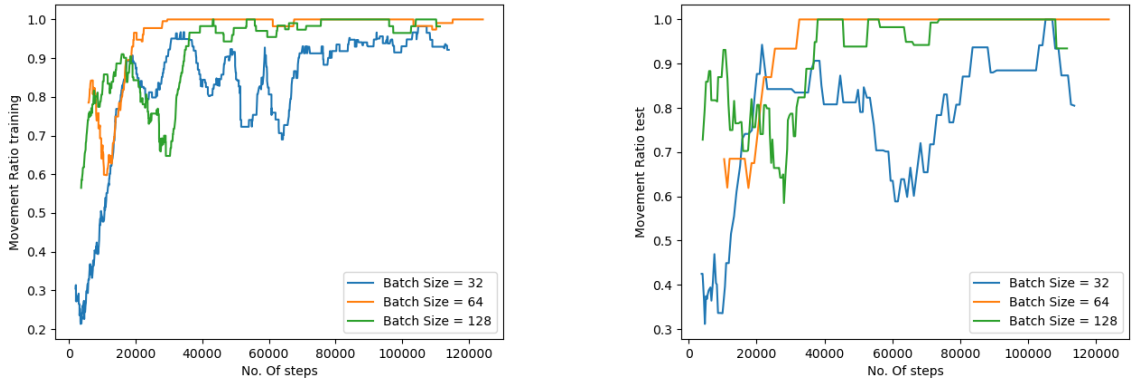

Figure 5.26: Movement ratio for PPO algorithm for three different batch sizes.

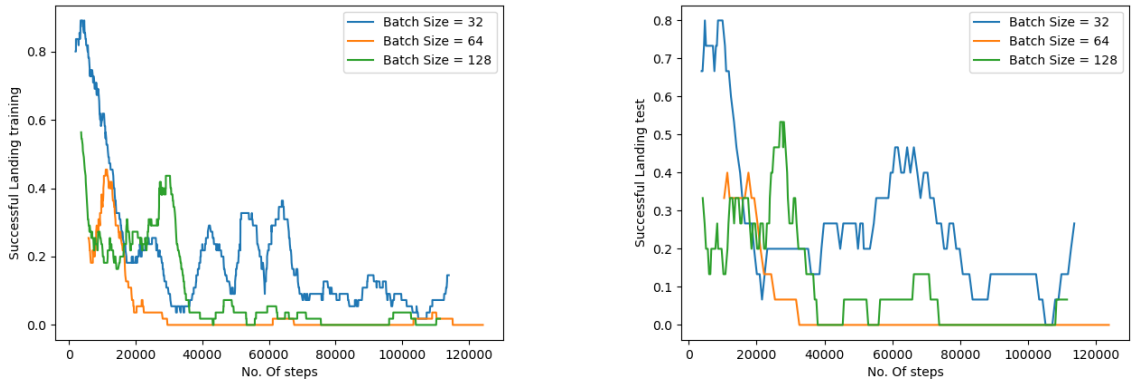

Figure 5.27: Successful landing for PPO algorithm for three different batch sizes.

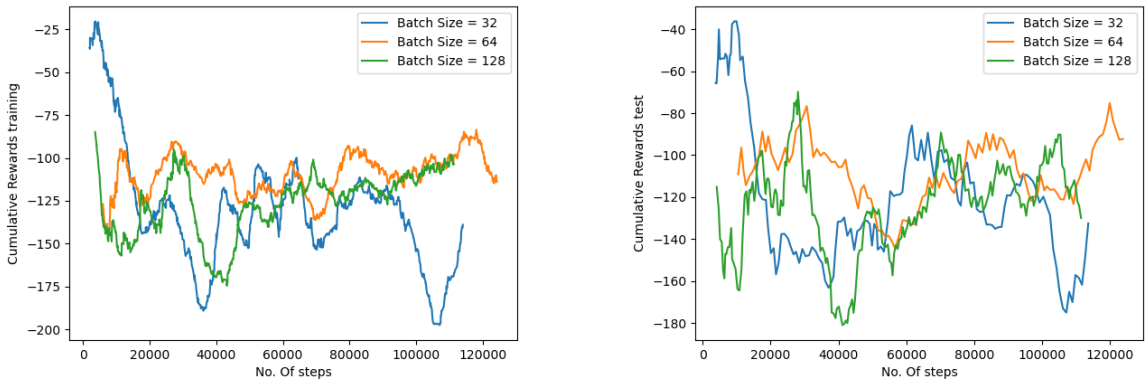

Figure 5.28: Cumulative rewards for PPO algorithm for three different batch sizes.

In the above results of the PPO algorithm for the environment of coverage area path planning for the "Manhattan32", we obtained the results where different batch rates are used to get three different results and to compare which batch rate fits the best for this hyperparameters setup. We have plotted various metrics for the performance measurement: CR, CRAL, landing attempts, successful landing, cumulative rewards, and more. Among the results we obtained, we can see that batch 64 provides the best results, giving a CR of around 32% for the randomly generated shapes. These experiments reveal a CRAL of 0.00. This indicates that when drones depart to cover the target area, they fail to return and land or reach the other side landing zones. This is attributed to the limited movement budget, preventing them from exploring further and causing them to get trapped in local minima. This suggests it needed more training steps to explore more and learn from the environment to function inside the environment properly. In addition, we can see the boundary counters are high, indicating that instead of properly navigating the target area based on the incentive rewards, it keeps hitting the boundary to explore more about the environment. When we see the graph plot, batch 64 has the lowest landing attempts and is exploring faster towards the target area. It might have better probabilities for exploring maximum percentages of the target area and then learning to land in the landing zones on time without exhausting the movement ratio. In addition, there are also cumulative rewards, which are hard to draw conclusive results from, but batch 64 shows promising results of cumulative rewards getting better over time without getting too many negative rewards.

In summary, the optimal performance of PPO in this particular environment is observed when the algorithm is trained for approximately two million steps. This training duration results in effective learning, showcasing superior cumulative rewards, a notable increase in CR, and a successful landing. This suggests that a careful balance in training duration significantly contributes to PPO's effectiveness within this specific environment, enhancing overall performance. These experiments indicate that a batch size of 64, coupled with a learning rate $3e-05$, produces optimal results by achieving high rewards, superior performance, and a rapid convergence rate, thereby saving considerable time.

5.2.3 MuZero: Experiments and Results

In this section, we are going to do extensive training on the MuZero algorithm and try to find the best possible combinations of the MuZero's hyperparameters and, at last, analyze the results and see how it performs compared to the other two algorithms like PPO and DDQN. It's crucial to acknowledge that MuZero is a complex algorithm, demanding substantial computational power for effective training. The resources outlined in Table 5.1 have limited capability, restricting our ability to conduct extensive environmental research. Consequently, compromises in various sectors of MuZero hyperparameters are necessary to obtain results that can be reasonably compared.

Experiments with different hyperparameter combinations

In this section, we will train the MuZero algorithms with varying combinations of hyperparameters in our environment coverage path planning and see how the MuZero algorithms behave in those scenarios. The various combinations of the MuZero algorithms hyperparameters are mentioned in the table below.

No.	Parameter Type	1st	2nd	3rd
1	Learning rate	3e-06	3e-5	3e-06
2	Discount Factor	0.95	0.98	0.99
3	Buffer size	2e3	1e3	1e3
4	Sample size	200	100	125
5	Number of simulations	25	5	5
6	Number of Bootstrap Timesteps	5	2	1
7	Number of Unroll Steps	5	2	1
8	Local Map scaling	17	9	17
9	Global Map scaling	3	3	3
10	Number of steps	2e6	2e6	2e6
11	Dirichlet alpha	0.25	0.25	0.25
12	Exploration Fraction	0.25	0.25	0.25
13	C1	1.25	1.25	1.25
14	C2	19652	19652	19652

Table 5.10: Parameters used in MuZero architecture for the training and testing coverage path planning on "Manhattan32".

In the following section, we compared the results of the MuZero algorithm with three different hyperparameter combinations. We plotted the graph of the various metrics against the number of steps for the algorithm.

Experiment No.	CR		CRAL		BC		LA		MR		SL		Rewards	
	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test
1st	0.03	0.06	0.01	0.00	36.81	16.36	4.96	2.98	0.45	0.75	0.65	0.29	−55.05	−107.18
2nd	0.06	0.07	0.00	0.00	62.03	27.12	6.27	0.86	0.73	0.96	0.34	0.05	−80.22	−128.71
3rd	0.03	0.05	0.01	0.00	90.85	2.43	5.58	0.00	0.30	1.00	0.82	0.00	−36.62	−255.56

Table 5.11: MuZero results for coverage path planning with three different hyperparameter configurations.

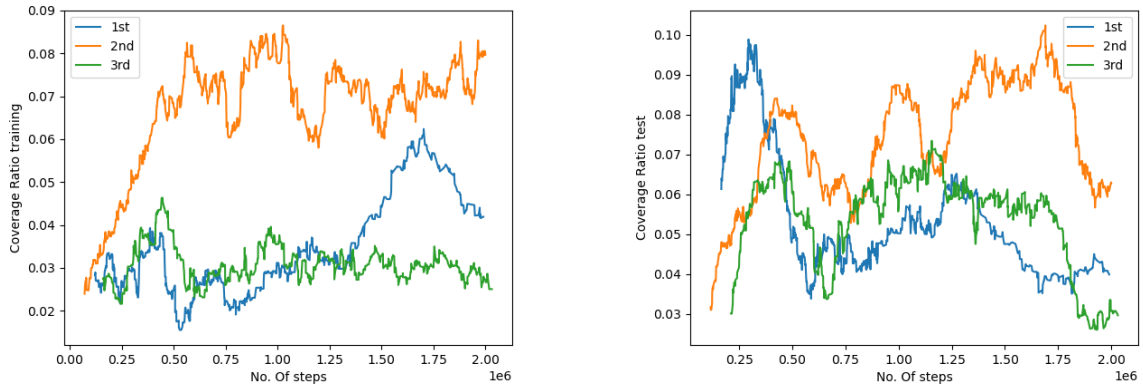

Figure 5.29: CR for MuZero algorithm for three different hyperparameter configurations.

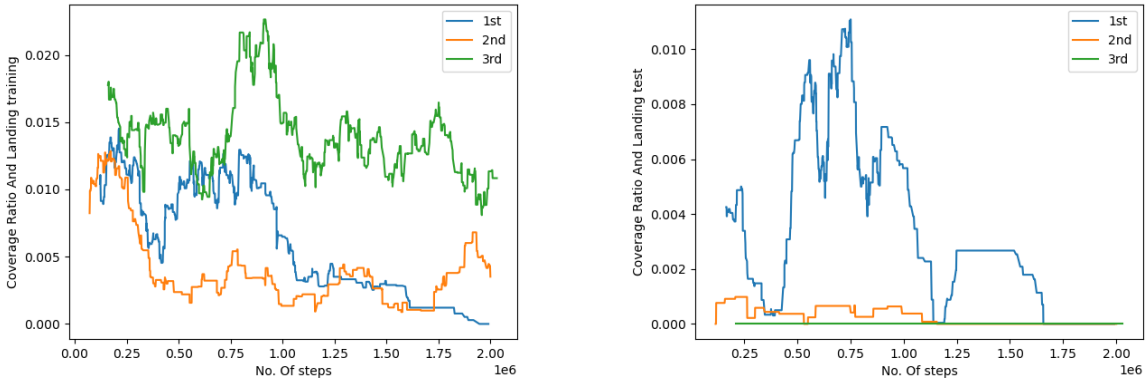

Figure 5.30: CRAL for MuZero algorithm for three different hyperparameter configurations.

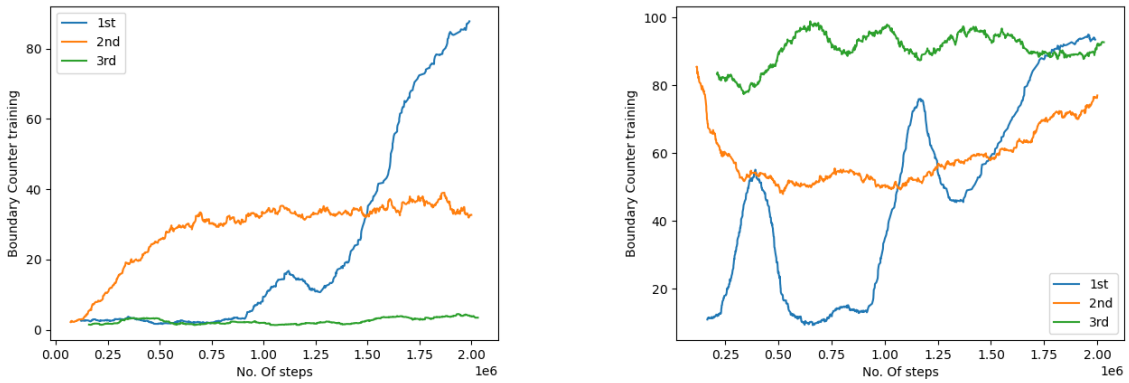

Figure 5.31: Boundary counters for MuZero algorithm for three different hyperparameter configurations.

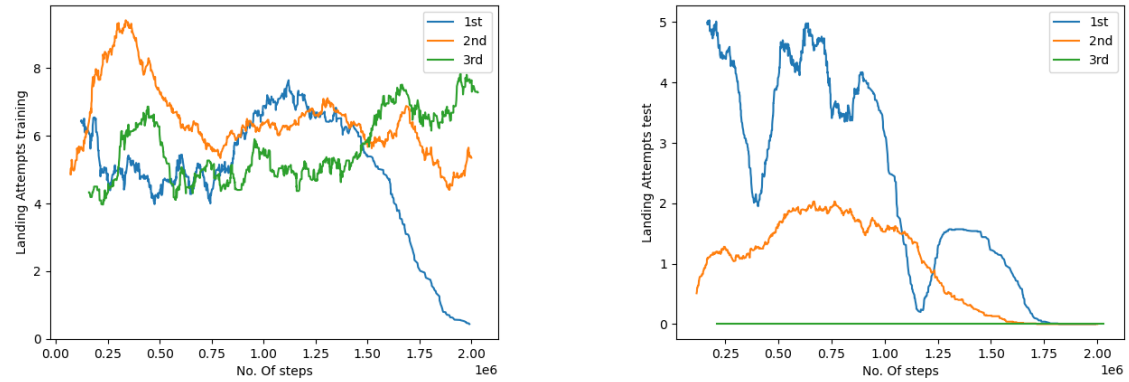

Figure 5.32: Landing attempts for MuZero algorithm for three different hyperparameter configurations.

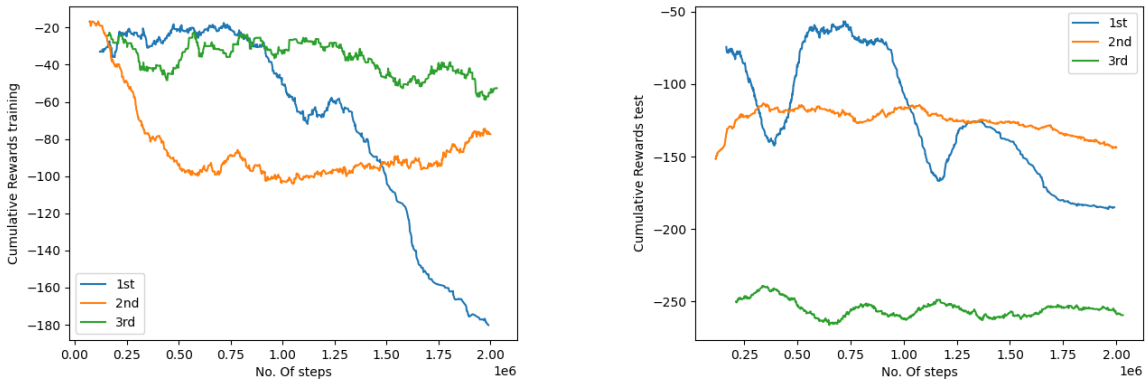

Figure 5.35: Cumulative rewards for MuZero algorithm for three different hyperparameter configurations.

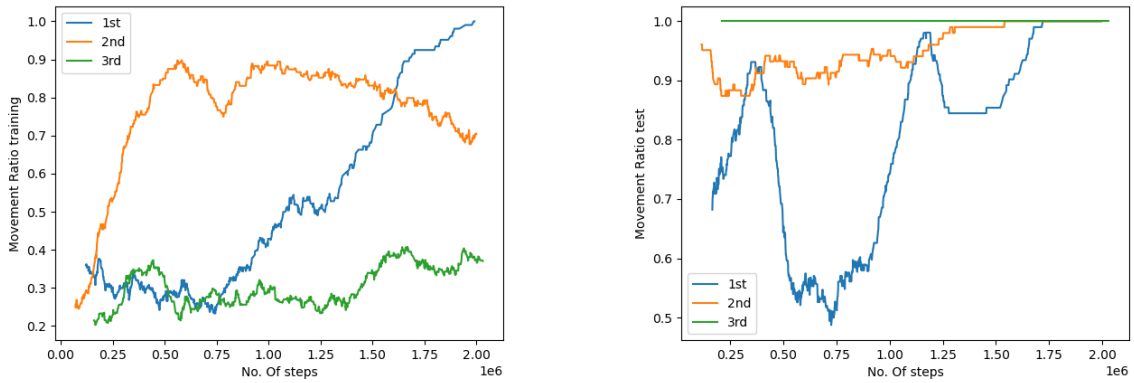

Figure 5.33: Movement ratio for MuZero algorithm for three different hyperparameter configurations.

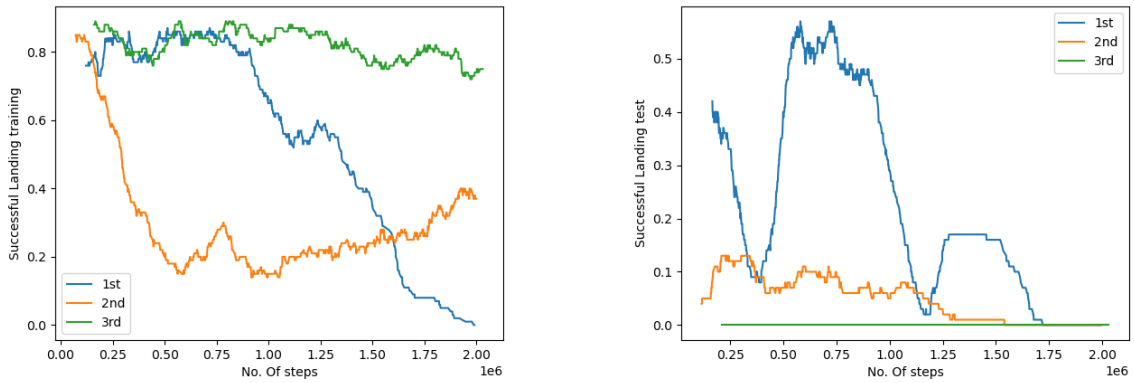

Figure 5.34: Successful landing for MuZero algorithm for three different hyperparameter configurations.

From the graph above, it can be seen the coverage area of the drone is about 10% to 13% in best-case scenarios. MuZero exhibits optimal performance in the hyperparameter configuration of the second experiment. However, these results are not good enough if we compare them with the results of the DDQN algorithm. This is due to our limited resources to train the MuZero algorithm. The hyperparameters, like the number of simulations, buffer size, and sampling size for the training from the memory of experienced samples and others, are highly reduced to obtain some results for this environment. Even though the results are not good enough, we can still derive some understanding from these results, such as the agent having minimal CRAL values. This means the drone was successfully covering the coverage area, but it failed to land inside the landing zone after it took off.

No.	Parameter Type	1st	2nd	3rd
1	Learning rate	5e-06	3e-05	3e-05
2	Discount Factor	0.98	0.95	0.99
3	Buffer size	1e3	1e3	1e3
4	Sample size	125	100	125
5	Number of simulations	20	25	25
6	Number of Bootstrap Timesteps	15	5	5
7	Number of Unroll Steps	25	5	5
8	Number of steps	165K	200K	500K
9	Local Map scaling	17	9	17
10	Global Map scaling	3	3	3
11	Dirichlet alpha	0.25	0.25	0.25
12	Exploration Fraction	0.25	0.35	0.25
13	C1	1.25	1.25	1.25
14	C2	19652	19652	19652

Table 5.12: Parameters used in MuZero architecture for the training and testing of coverage path planning on "Manhattan32".

Experiment No.	CR		CRAL		BC		LA		MR		SL		Rewards	
	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test	tr.	test
1st	0.04	0.09	0.01	0.00	6.14	36.27	6.49	1.78	0.45	0.94	0.68	0.08	−113.41	−38.10
2nd	0.03	0.04	0.02	0.00	3.64	48.71	5.32	2.04	0.35	0.83	0.79	0.20	−183.06	−43.34
3rd	0.03	0.04	0.01	0.00	3.10	44.66	5.16	4.16	0.28	0.70	0.82	0.34	−11.63	−25.12

Table 5.13: MuZero results for coverage path planning with different hyperparameter configuration

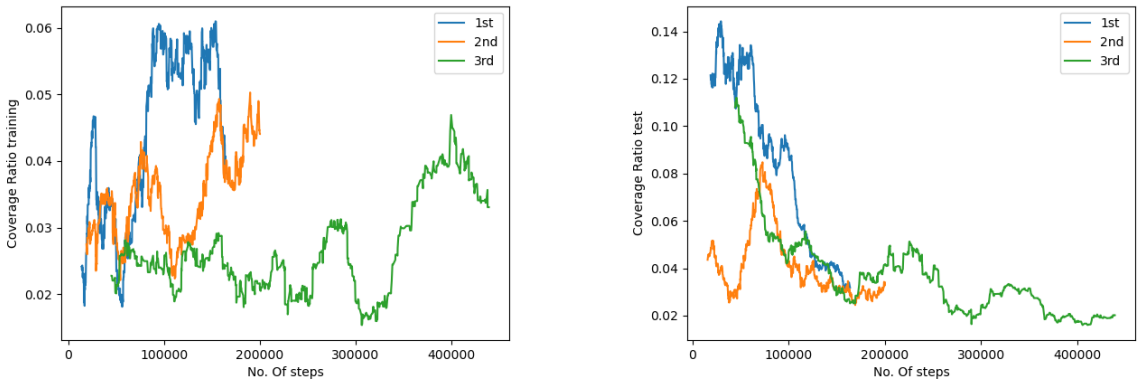

Figure 5.36: CR for MuZero algorithm for three different hyperparameter configurations.

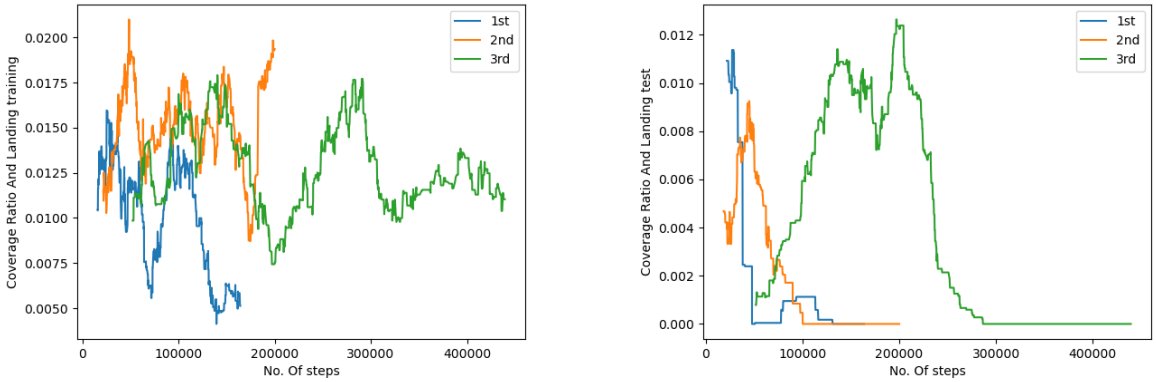

Figure 5.37: CRAL for MuZero algorithm for three different hyperparameter configurations.

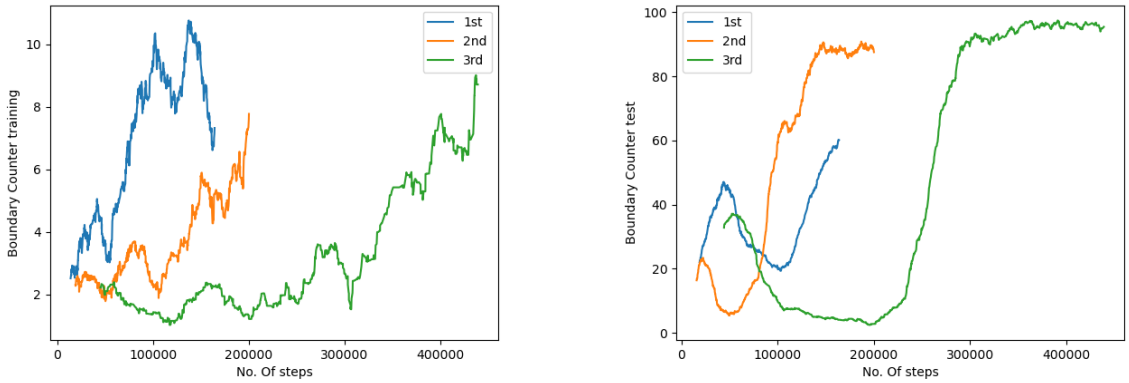

Figure 5.38: Boundary counters for MuZero algorithm for three different hyperparameter configurations.

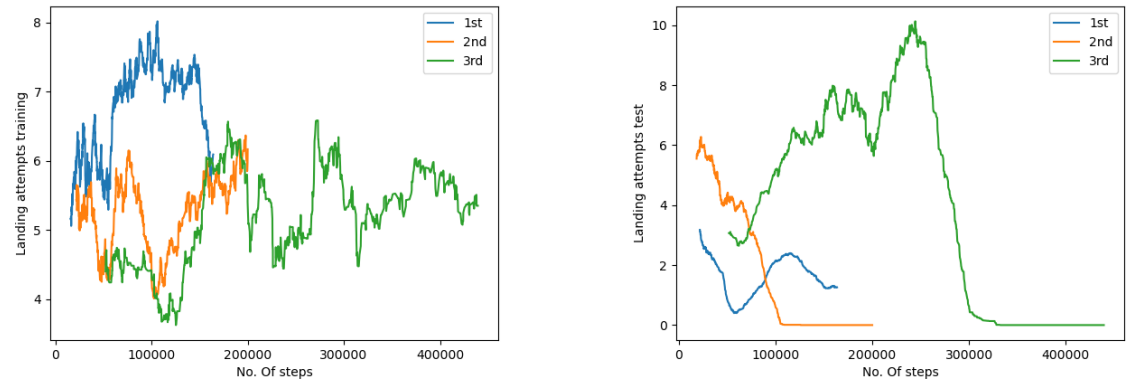

Figure 5.39: Landing attempts for MuZero algorithm for three different hyperparameter configurations.

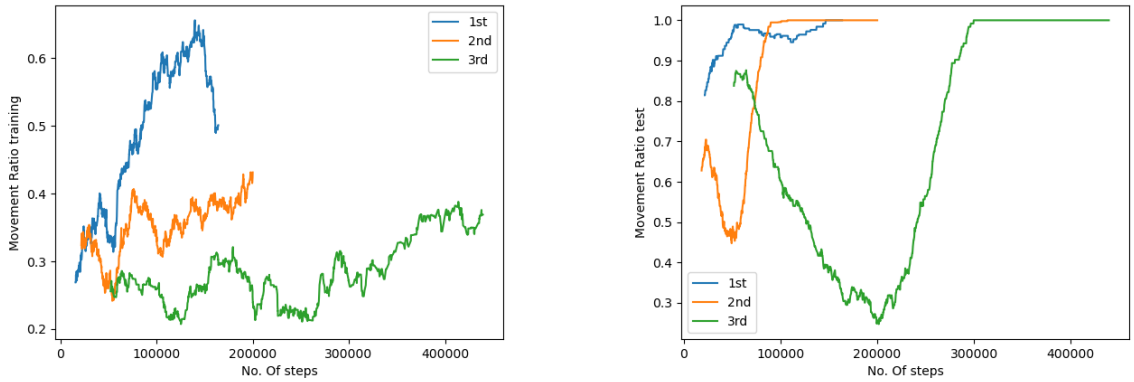

Figure 5.40: Movement ratio for MuZero algorithm for three different hyperparameter configurations.

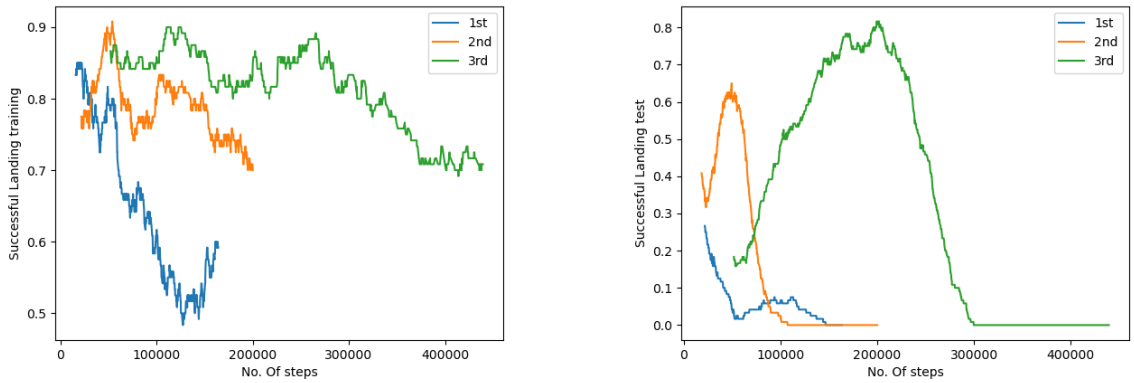

Figure 5.41: Successful landing for MuZero algorithm for three different hyperparameter configurations.

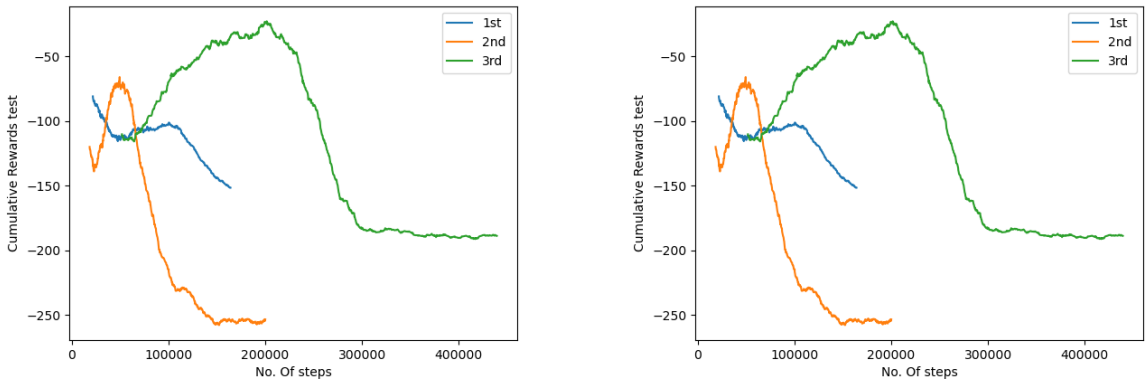

Figure 5.42: Cumulative rewards for MuZero algorithm for three different hyperparameter configurations.

The results are plotted on the graph for different metrics against the different number of steps, i.e., 165k, 200k and 500k. Due to the limited time and very long time consumption to train the MuZero algorithm, the results are shown with concise steps and show the algorithm's performance for the coverage path planning for the "Manhattan32" environment. This training took an average of 70 hours for the available computing resources. However, the hyperparameters set for this experiment are not optimal and can be better adjusted if the computing resources are available. Despite the lack of computing resources, few results are derived for the adjusted parameters and can be analyzed from their results. The above results show that CR hardly reached 10% of the target area. The number of simulations is set to 20, 25 and 25, respectively, and the number of future bootstraps in MCTS is set to 15, 5 and 5, which are not considered enough to get them through the idea of true value inside MCTS. In this simulation, the drone still fails to land safely after taking off to explore the environment. This also indicates that the movement budget is exhausted before landing in the landing zone. However, compared to all three different setup experiments, 3rd one still performs better as it gets to train with more steps, and learning gets better over time. It also shows that it's not overfitting till 500k steps.

In summary, MuZero, the general-purpose algorithm, is suited to train various environments. This is why we try to apply the MuZero algorithm in our coverage Path planning to cover the target area with a limited budget. However, MuZero is computationally intensive, especially in complex environments. This limited our knowledge of learning about MuZero in the given environment, and only some conclusions could be drawn. We needed a more computationally powerful computer to train and analyse the MuZero algorithm. Some of the findings that can be pointed out are that MuZero works better where we need to plan using MCTS, like in "Manhattan32". It must look ahead for a better path for the target area coverage. It is also sample-efficient, learning policies and values simultaneously, enabling effective decision-making and state evaluation.

6 Discussion

In this section, we will discuss our findings afterwards, point out our experiment limitations and finally provide some undiscovered areas for future improvements.

6.1 Exchange of Views

Our experiments covered three algorithms trained and tested for a single environment called "Manhattan32" for coverage path planning. We used all seven metrics, showing how our algorithm performed in general. However, due to a lack of computing resources, we could not explore the effect of each algorithm perfectly, but we still had a chance to learn about the algorithm to some extent. Here, we discussed experimental thoughts on topic comparison.

1. **Time complexity**
   We measured the run time complexity of each algorithm for the training and test for the given environment. The fastest run time for an actor to learn the environment was PPO, and after that, DDQN. The most complex and time-consuming way to train the actor was in the MuZero. MuZero gets faster by reducing the number of rollouts, number of simulations, and other parameters like neural network size, but it further diminishes its learning capacity. DDQN's average run time to train and test was around 90-95 hours, whereas PPO's average run time can be calculated as 60-70 hours for the given computing power mentioned in table 5.1. However, to train MuZero, we need more computing power to analyze the results properly for the given environment.

2. **Overall results**
   Here, we saw the DDQN performing best compared to PPO and MuZero. However, it can be said that the PPO model-free algorithm, which uses a trust region approach, showed more balance between exploration and exploitation during training. Training everything online makes PPO faster than the other two algorithms. On the other hand, DDQN had fast convergence but still took longer to prepare. The coverage ratio and landing were perfect for DDQN, which makes it very suitable for this environment. MuZero, being a very complex algorithm, was comparatively slow and computationally intensive due to MCTS during planning. MuZero was also very hyperparameter sensitive, and choosing hyperparameters along with extended run time was troublesome. It can also be noted from the observed experiments that balancing the exploration and exploitation is challenging to achieve and does not work as smoothly as the other two algorithms.

3. **Hyperparameters findings**
   In finding hyperparameters, we know the learning rate 3e-05 performs best for all the algorithms we used, where the batch sizes depend on each algorithm. Batch size 64 works best for the PPO algorithm, and batch size 128 works best for the DDQN algorithm. On the other hand, there were several hyperparameters to choose carefully to get better results for the MuZero algorithm. For the MuZero algorithm, we tried to configure the number of simulations, where a higher number showed more coverage area. Still, it converged fast and stopped exploring more, decreasing coverage area and landing. Many different temperature parameters were tried in the MuZero to balance between exploration and exploitation. Still, it failed to learn appropriately to explore and get a higher coverage ratio and successful landing. Due to having three different neural networks, we tried to decrease the size of the neural network in MuZero, but it only worsened the results. MuZero, with 3 hidden layers and 256 nodes in each layer, causes the algorithm to become extremely slow for our available computing resources but has some better results comparatively. In addition, all three algorithms ideally support the local map scaling as 17 and global map scaling as 3, except for some hyperparameters combination for PPO, where local map scaling as 9 outperforms the local map scaling as 17 in terms of execution time. It can be noted that it became evident that determining optimal hyperparameters for various algorithms lacked a clear-cut methodology. The approach involved a somewhat guessing process, where we relied on random predictions and approximations informed by insights from relevant research papers. The evaluation of the hyperparameter choices unfolded through multiple algorithm runs. While the prospect of discovering even more potent combinations was on the horizon, the practical limitations of lack of resources and time posed obstacles, rendering it challenging to pursue additional refinement and optimization.

6.2 Limitation

The main drawbacks of our algorithms based on our experiments are listed below.

1. **Double deep Q-network (DDQN):**
   - Overestimation issues: DDQN sometimes overestimates the Q-values due to the max operation involved in the target value calculation. This overestimation hindered the selection of the best policies.
2. **Proximal policy optimization (PPO):**
   - Hyperparameter sensitivity: PPO's performance was sensitive to the choice of hyperparameters. Tuning these parameters was challenging, and suboptimal choices led to algorithm instability and slow learning or, in some cases, not learning.
3. **MuZero:**
   - Computational intensity: MuZero was computationally intensive, especially during the planning phase, where MCTS simulates possible future trajectories. Hence, there is a considerable execution time for the available resources.

7 Conclusion and Future Work

In this section, we will discuss the conclusion and the possible future work that can be done on this thesis work.

7.1 Conclusion

The main objective of our thesis is to find the best algorithm suitable between DDQN, PPO and MuZero for finding the coverage path planning for the map "Manhattan32" presented in Chapter 1. A thorough explanation of methodologies and results is carried out in chapters 4 and 5. Here, we briefly summarise the outcome of this thesis work.

In this thesis, three different algorithms were trained in the same environment. Among these algorithms, we found DDQN was the best-performing algorithm compared to PPO and MuZero. Each of them had some limitations. However, DDQN had the best performance ratio for the available resources. Due to a lack of proper resources to train all these algorithms for the same number of steps for the believed optimal hyperparameter combinations, the results obtained cannot be directly compared, and the conclusion derived from these results cannot be conclusive. Due to this reason, we have not reached the best results for each algorithm directly in a graph-plotted manner. However, observing the results of the PPO algorithm, we can still guess that PPO can perform well in this environment. Running for 2 million steps can score good results, helping the actor in the environment learn a better coverage ratio and have a successful landing. The MuZero algorithm cannot be the preferred algorithm to train this environment due to its computationally extensive structure, hyperparameter sensitivity, and large execution time. Tuning the hyperparameters of MuZero increases the execution time, and conducting those model experiments was impossible. Finally, we decided to train a combination of suboptimal hyperparameters, and the results were shown. In conclusion, for available resources, DDQN was the best-performing algorithm for the "Manhattan32" map in the given environment compared to PPO and MuZero algorithms.

7.2 Future Work

Future work for this thesis involves experimenting with these three algorithms with enough resources to help find the optimal combinations of hyperparameters to understand the capacity of these algorithms truly and if they can be used to train the "Manhattan32" map for coverage path planning. In this thesis work, due to a lack of resources, we only ran our experiments on "Manhattan32", but in future work, we would like to train all three algorithms in the map named "Urban50" and compare the results.

Bibliography

[1] Fadi AlMahamid *et al.* "Reinforcement Learning Algorithms: An Overview and Classification." In: *2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)* (2021), pp. 1–7. URL: https://api.semanticscholar.org/CorpusID:240003074.

[2] Diego A. Aponte-Roa *et al.* "Development and Evaluation of a Remote Controlled Electric Lawn Mower." In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. 2019, pp. 1–5. DOI: 10.1109/CCWC.2019.8666455.

[3] Blackburn. *Reinforcement Learning: Markov Decision Process — Part 2*. Aug. 2019. URL: https://towardsdatascience.com/reinforcement-learning-markov-decision-process-part-2-96837c936ec3.

[4] Leo Breiman. "Statistical Modeling: The Two Cultures (with comments and a rejoinder by the author)." In: *Statistical Science* 16.3 (2001), pp. 199–231. DOI: 10.1214/ss/1009213726. URL: https://doi.org/10.1214/ss/1009213726.

[5] Andrew Connolly. *Deep Learning: Classifying Astronomical Images*. https://www.astroml.org/astroML-notebooks/chapter9/astroml_chapter9_Deep_Learning_Classifying_Astronomical_Images.html. Accessed: July 14, 2023. Year.

[6] Antonio Coronato *et al.* "Reinforcement learning for intelligent healthcare applications: A survey." In: *Artificial Intelligence in Medicine* 109 (2020), p. 101964. ISSN: 0933-3657. DOI: https://doi.org/10.1016/j.artmed.2020.101964. URL: https://www.sciencedirect.com/science/article/pii/S093336572031229X.

[7] Robert Culkin *et al.* "Machine learning in finance: the case of deep learning for option pricing." In: *Journal of Investment Management* 15.4 (2017), pp. 92–100.

[8] DeepMind. *MuZero: Mastering Go, Chess, Shogi, and Atari without Rules*. year. URL: https://www.deepmind.com/blog/muzero-mastering-go-chess-shogi-and-atari-without-rules (visited on 09/05/2023).

[9] Di Deng *et al.* "Constrained Heterogeneous Vehicle Path Planning for Large-area Coverage." In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 4113–4120. DOI: 10.1109/IROS40897.2019.8968299.

[10] Hugging Face. *What is Reinforcement Learning?* https://huggingface.co/learn/deep-rl-course/unit1/what-is-rl?fw=pt. accessed 2023.

[11] Jiajia Fan *et al.* "Second Path Planning for Unmanned Surface Vehicle Considering the Constraint of Motion Performance." In: *Journal of Marine Science and Engineering* 7.4 (2019). ISSN: 2077-1312. DOI: 10.3390/jmse7040104. URL: https://www.mdpi.com/2077-1312/7/4/104.

[12] David Foster. *MuZero: The WalkThrough*. Accessed on September 8, 2023. Medium. 2023. URL: https://medium.com/applied-data-science/how-to-build-your-own-muzero-in-python-f77d5718061a.

[13] Shuai-Feng Guo *et al. Faster Crossing over Quantum Phase Transition Assisted by Reinforcement Learning*. Nov. 2020.

[14] David Isele *et al.* "Safe Reinforcement Learning on Autonomous Vehicles." In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018, pp. 1–6. DOI: 10.1109/IROS.2018.8593420.

[15] Jonas Kiemel *et al.* "PaintRL: Coverage Path Planning for Industrial Spray Painting with Reinforcement Learning." In: June 2019.

[16] Petar Kormushev *et al.* "Reinforcement Learning in Robotics: Applications and Real-World Challenges." In: *Robotics* 2.3 (2013), pp. 122–148. ISSN: 2218-6581. DOI: 10.3390/robotics2030122. URL: https://www.mdpi.com/2218-6581/2/3/122.

[17] Phone Thiha Kyaw *et al.* "Coverage Path Planning for Decomposition Reconfigurable Grid-Maps Using Deep Reinforcement Learning Based Travelling Salesman Problem." In: *IEEE Access* 8 (2020), pp. 225945–225956. DOI: 10.1109/ACCESS.2020.3045027.

[18] Anh Vu Le *et al.* "Coverage Path Planning Using Reinforcement Learning-Based TSP for hTetran—A Polyabolo-Inspired Self-Reconfigurable Tiling Robot." In: *Sensors* 21.8 (2021). ISSN: 1424-8220. DOI: 10.3390/s21082577. URL: https://www.mdpi.com/1424-8220/21/8/2577.

[19] Anh Vu Le *et al.* "Reinforcement Learning-Based Energy-Aware Area Coverage for Reconfigurable hRombo Tiling Robot." In: *IEEE Access* 8 (2020), pp. 209750–209761. DOI: 10.1109/ACCESS.2020.3038905.

[20] LearnDataSci. *Venn diagram for machine learning subfields*. 2017. URL: %5Curl%7Bhttps://i.pinimg.com/564x/4a/e7/de/4ae7de2f2157e8e36f2b563060694a06.jpg%7D.

[21] Bruna G. Maciel-Pearson *et al. Online Deep Reinforcement Learning for Autonomous UAV Navigation and Exploration of Outdoor Environments*. 2019. arXiv: 1912.05684 [cs.CV].

[22] Xu Miao *et al.* "Scalable Coverage Path Planning for Cleaning Robots Using Rectangular Map Decomposition on Large Environments." In: *IEEE Access* 6 (2018), pp. 38200–38215. DOI: 10.1109/ACCESS.2018.2853146.

[23] Volodymyr Mnih *et al. Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[24] DongKi Noh *et al.* "Adaptive Coverage Path Planning Policy for a Cleaning Robot with Deep Reinforcement Learning." In: *2022 IEEE International Conference on Consumer Electronics (ICCE)*. 2022, pp. 1–6. DOI: 10.1109/ICCE53296.2022.9730307.

[25] OpenAI *et al. Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. arXiv: 1912.06680 [cs.LG].

[26] Artem Oppermann. *Deep Double Q-Learning*. Accessed: August 12, 2023. 2023. URL: https://artemoppermann.com/de/deep-double-q-learning/.

[27] Martijn van Otterlo *et al.* "Reinforcement Learning and Markov Decision Processes." In: *Reinforcement Learning: State-of-the-Art*. Ed. by Marco Wiering *et al.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–42. ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_1. URL: https://doi.org/10.1007/978-3-642-27645-3_1.

[28] Sihem Ouahouah *et al.* "Deep-Reinforcement-Learning-Based Collision Avoidance in UAV Environment." In: *IEEE Internet of Things Journal* 9.6 (2022), pp. 4015–4030. DOI: 10.1109/JIOT.2021.3118949.

[29] Harikumar Pallathadka *et al.* "IMPACT OF MACHINE learning ON Management, healthcare AND AGRICULTURE." In: *Materials Today: Proceedings* 80 (2023). SI:5 NANO 2021, pp. 2803–2806. ISSN: 2214-7853. DOI: https://doi.org/10.1016/j.matpr.2021.07.042. URL: https://www.sciencedirect.com/science/article/pii/S221478532104894X.

[30] Chenyang Qi *et al.* "UAV path planning based on the improved PPO algorithm." In: *2022 Asia Conference on Advanced Robotics, Automation, and Control Engineering (ARACE)*. 2022, pp. 193–199. DOI: 10.1109/ARACE56528.2022.00040.

[31] Thomas Renault. "Sentiment analysis and machine learning in finance: a comparison of methods and models on one million messages." In: *Digital Finance* 2.1 (Sept. 2020), pp. 1–13. ISSN: 2524-6186. DOI: 10.1007/s42521-019-00014-x. URL: https://doi.org/10.1007/s42521-019-00014-x.

[32] Julius Rückin *et al.* "Adaptive Informative Path Planning Using Deep Reinforcement Learning for UAV-based Active Sensing." In: *2022 International Conference on Robotics and Automation (ICRA)*. 2022, pp. 4473–4479. DOI: 10.1109/ICRA46639.2022.9812025.

[33] Julian Schrittwieser. *MuZero Intuition*. Accessed on September 8, 2023. furidamu. 2023. URL: https://www.furidamu.org/blog/2020/12/22/muzero-intuition/.

[34] John Schulman *et al. Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[35] SCS. *Venn diagram to distinguish artificial intelligence, machine learning and deep learning concepts*. 2020. URL: %5Curl%7Bhttps://www.scs.org.sg/articles/machine-learning-vs-deep-learning%7D.

[36] Denis Sheynikhovich. "Spatial navigation in geometric mazes: a computational model of rodent behavior." In: (Jan. 2007). DOI: 10.5075/epfl-thesis-3922.

[37] Jenni A. M. Sidey-Gibbons *et al.* "Machine learning in medicine: a practical introduction." In: *BMC Medical Research Methodology* 19.1 (Mar. 2019), p. 64. ISSN: 1471-2288. DOI: 10.1186/s12874-019-0681-4. URL: https://doi.org/10.1186/s12874-019-0681-4.

[38] David Silver. *MDP Lecture Notes*. https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf. 2020.

[39] David Silver *et al.* "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm." In: *CoRR* abs/1712.01815 (2017). arXiv preprint. arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815.

[40] David Silver *et al.* "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

[41] David Silver *et al.* "Mastering the game of Go without human knowledge." In: *Nature* 550.7676 (Oct. 2017), pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270. URL: https://doi.org/10.1038/nature24270.

[42] Thomas Simonini. *Diving Deeper into Reinforcement Learning with Q-Learning.* Accessed: April 10, 2018. 2018. URL: https://www.freecodecamp.org/news/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe/.

[43] Taishi Takeda *et al.* "Path generation algorithm for search and rescue robots based on insect behavior — Parameter optimization for a real robot." In: *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR).* 2016, pp. 270–271. DOI: 10.1109/SSRR.2016.7784310.

[44] Mirco Theile *et al.* "UAV path planning using global and local map information with deep reinforcement learning." In: *2021 20th International Conference on Advanced Robotics (ICAR).* IEEE. 2021, pp. 539–546.

[45] Oriol Vinyals *et al.* "Grandmaster level in StarCraft II using multi-agent reinforcement learning." In: *Nature* 575.7782 (Nov. 2019), pp. 350–354. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. URL: https://doi.org/10.1038/s41586-019-1724-z.

[46] Roland Vogl. *Research Handbook on Big Data Law.* Cheltenham, UK: Edward Elgar Publishing, 2021. ISBN: 9781788972819. DOI: 10.4337/9781788972826. URL: https://www.elgaronline.com/view/edcoll/9781788972819/9781788972819.xml.

[47] Wei Wang *et al.* "Investigation on Works and Military Applications of Artificial Intelligence." In: *IEEE Access* PP (July 2020), pp. 1–1. DOI: 10.1109/ACCESS.2020.3009840.

[48] William Yang Wang *et al.* "Deep Reinforcement Learning for NLP." In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts.* Melbourne, Australia: Association for Computational Linguistics, July 2018, pp. 19–21. DOI: 10.18653/v1/P18-5007. URL: https://aclanthology.org/P18-5007.

[49] Xianlin Wang *et al.* "Bond strength prediction of concrete-encased steel structures using hybrid machine learning method." In: *Structures* 32 (Sept. 2021), pp. 2279–2292. DOI: 10.1016/j.istruc.2021.04.018.

[50] Yinchu Wang *et al.* "Coverage path planning for kiwifruit picking robots based on deep reinforcement learning." In: *Computers and Electronics in Agriculture* 205 (2023), p. 107593. ISSN: 0168-1699. DOI: https://doi.org/10.1016/j.compag.2022.107593. URL: https://www.sciencedirect.com/science/article/pii/S0168169922009012.

[51] Thierry Warin *et al.* "Machine Learning in Finance: A Metadata-Based Systematic Review of the Literature." In: *Journal of Risk and Financial Management* 14.7 (2021). ISSN: 1911-8074. DOI: 10.3390/jrfm14070302. URL: https://www.mdpi.com/1911-8074/14/7/302.

[52] Wikipedia contributors. *Reinforcement learning — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1161565586. [Online; accessed 27-June-2023]. 2023.

[53] Bowen Xing *et al.* "An Algorithm of Complete Coverage Path Planning for Unmanned Surface Vehicle Based on Reinforcement Learning." In: *Journal of Marine Science and Engineering* 11.3 (2023). ISSN: 2077-1312. DOI: 10.3390/jmse11030645. URL: https://www.mdpi.com/2077-1312/11/3/645.

[54] Chao Yan *et al.* "Towards Real-Time Path Planning through Deep Reinforcement Learning for a UAV in Dynamic Environments." In: *Journal of Intelligent & Robotic Systems* 98.2 (May 2020), pp. 297–309. DOI: 10.1007/s10846-019-01073-3. URL: https://doi.org/10.1007/s10846-019-01073-3.

[55] Peng Yao *et al.* "UAV/USV Cooperative Trajectory Optimization Based on Reinforcement Learning." In: *2022 China Automation Congress (CAC)*. 2022, pp. 4711–4715. DOI: 10.1109/CAC57257.2022.10055417.

[56] Hongmei Zhang *et al.* "A Path Planning Strategy for Intelligent Sweeping Robots." In: *2019 IEEE International Conference on Mechatronics and Automation (ICMA)*. 2019, pp. 11–15. DOI: 10.1109/ICMA.2019.8816519.

[57] Maoqing Zhang *et al.* "Modeling and optimization of watering robot optimal path for ornamental plant care." In: *Computers Industrial Engineering* 157 (2021), p. 107263. ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2021.107263. URL: https://www.sciencedirect.com/science/article/pii/S0360835221001674.

[58] Xiaolin Zhou *et al.* "Multi-Robot Coverage Path Planning based on Deep Reinforcement Learning." In: *2021 IEEE 24th International Conference on Computational Science and Engineering (CSE)*. 2021, pp. 35–42. DOI: 10.1109/CSE53436.2021.00015.

[59] Xinyuan Zhou *et al.* "Learn to Navigate: Cooperative Path Planning for Unmanned Surface Vehicles Using Deep Reinforcement Learning." English (US). In: *IEEE Access* 7 (2019). Publisher Copyright: © 2013 IEEE., pp. 165262–165278. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2953326.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, November 13, 2023