FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Deep Reinforcement Learning Methods for Cooperative Robotic Navigation

José Pedro Ferreira Pinheiro de Carvalho

MASTER IN ELECTRICAL AND COMPUTER ENGINEERING

Supervisor: Professor A. Pedro Aguiar

October 11, 2023

Abstract

The field of intelligent autonomous systems has experienced exponential growth, with robots being developed for a wide range of applications. Research in this area is moving towards fully autonomous robots, and cooperation among multiple robots emerges as a natural next step in their evolution. Collaborative robots offer inherent advantages, such as the ability to handle more complex tasks, increased redundancy, and improved overall efficiency. Artificial Intelligence has witnessed significant advancements, with Reinforcement Learning standing out as a solution that exhibits super-human performance in various decision-making problems, some of which directly apply to enhancing autonomous robot systems.

Robotic navigation plays a crucial role in any autonomous mobile robotic system, with path planning being a key component. However, traditional path-planning solutions for autonomous robots often rely on extensive, specialized software stacks tailored to specific robots, configurations, and tasks.

Furthermore, in a future where robots become part of everyday life, they will seldom exist in isolation from the world. Thus, awareness of other agents and the ability to cooperate become crucial. Tasks like coverage path planning and active exploration inherently benefit from the collaboration of multiple robots, enabling faster completion times or tackling more complex environments. However, combining robotic navigation with multiple agents introduces greater complexity, especially in real-life scenarios with constraints imposed by limited and distributed communication networks, uncertainty, dynamic environments, and onboard computation and sensing limitations.

In this context, this dissertation introduces a novel approach to robotic navigation by harnessing the potential of Deep Reinforcement Learning. In this document, multiple algorithms based on this framework will be introduced with the objective of handling diverse tasks, including point-to-point path planning, coverage path planning, and active exploration, either in single or multi-agent domains. Taking this into consideration, this dissertation focuses on developing algorithms for both single-agent and multi-agent robotic navigation, utilizing a Reinforcement Learning framework. Through this approach, the achievement of a one-size-fits-all solution to robotic navigation is demonstrated, yielding near-optimal results across various tasks and even surpassing existing approaches in the literature.

Resumo

Nos últimos anos, tem-se verificado um enorme crescimento na área dos sistemas autónomos inteligentes, com robôs a serem desenvolvidos para várias aplicações. A investigação nesta área orienta-se para o desenvolvimento de robôs totalmente autónomos, sendo que a cooperação entre múltiplos robôs surge como uma etapa natural na sua evolução. Os robôs colaborativos apresentam vantagens inerentes, nomeadamente a capacidade de efetuar tarefas mais complexas, uma maior redundância e melhor eficiência. A área de Inteligência Artificial tem experienciado progressos significantivos, com o *Reinforcement Learning* a destacar-se como uma solução eficaz com melhor desempenho que o ser humano em vários problemas de tomada de decisão, alguns dos quais diretamente aplicáveis a sistemas robóticos autónomos.

A navegação robótica apresenta um papel fulcral em qualquer sistema robótico móvel, tendo o planeamento de trajetórias como uma componente-chave. Contudo, as soluções tradicionais de planeamento de trajetórias para robôs autónomos frequentemente dependem de bibliotecas de *software* extensas e especializadas, concebidas para robôs específicos, com configurações e tarefas pouco flexíveis.

Adicionalmente, num futuro em que os robôs se tornam parte integrante do quotidiano, dificilmente um robô irá operar de uma forma completamente isolada. Consequentemente, a consciência acerca da existência de outros agentes e a capacidade de cooperação tornam-se fundamentais para qualquer robô a vir a ser desenvolvido. Tarefas como o planeamento de trajetórias e a exploração ativa beneficiam da colaboração entre múltiplos robôs, permitindo reduzir os tempos de conclusão ou aumentar a complexidade das tarefas exigidas. No entanto, a navegação robótica com múltiplos agentes acarreta maior complexidade, sobretudo em cenários reais, onde se verificam restrições impostas por redes de comunicação, incertezas, ambientes dinâmicos, e limitações de computação e perceção.

Neste contexto, esta dissertação apresenta uma abordagem inovadora para a navegação robótica, capitalizando o potencial do *Deep Reinforcement Learning*. Serão introduzidos uma série de algoritmos baseados neste paradigma, com o objetivo de lidar com diversas tarefas, incluindo o planeamento de trajetórias ponto-a-ponto, o planeamento de trajetórias de cobertura de áreas e a exploração ativa, quer em cenário cooperativos ou a solo. Deste modo, o foco da dissertação é o desenvolvimento de algoritmos para navegação robótica, tanto para agentes individuais como para múltiplos agentes, recorrendo a uma base de *Reinforcement Learning*. Através desta abordagem, demonstra-se a possibilidade de alcançar uma solução genérica para a navegação robótica, obtendo resultados excelentes em diversas tarefas, ultrapassando abordagens existentes na literatura.

Acknowledgements

I would like to express my deepest gratitude to all those who have played a significant role in my academic journey. Your unwavering support and guidance have not only shaped this work but have also been instrumental in my personal growth, molding me into the person I have become today.

First and foremost, I am immensely thankful to my supervisor, Professor António Pedro Aguiar. Throughout this journey, he not only believed in my potential but also provided me with numerous opportunities to flourish both academically and personally. Professor Pedro's expertise, invaluable mentorship, and unwavering support played a pivotal role in shaping this dissertation, and I am profoundly thankful for his guidance.

I extend my sincere appreciation to Professor Paulo Tabuada, who provided me with the opportunity to work within the Cyber-Physical Systems Laboratory at the University of California, Los Angeles. His insightful suggestions and enduring support were essential in realizing this dissertation. Additionally, I want to express my gratitude to the Ph.D. candidates Jonathan Bunton, Matteo Marchi, Marcus Lucas, and Master Yskandar Gas for going out of their way to help me with my research and making me feel at home on the other side of the planet.

Additionally, I would like to thank C2SR Lab and SYSTEC for providing me with excellent working conditions and the necessary hardware for this dissertation. I am also grateful to the lab members for their support and integration. In particular, I would like to thank Gustavo Andrade for always being available to help me with the computer simulations.

I would also like to express my gratitude to all my friends who accompanied me throughout this journey, especially João Martins, Maria Lopes, and João Costa. I will never forget all the memories we have created these last few years, and I hope we will keep accomplishing great things together. My gratitude is also extended to all members of the 5dpo robotics team, to my colleagues Lourenço Pinho, Marco Costa, Maria Fonseca, and my childhood friend Helena Vaz. You were all the greatest support net I could ever hope for.

Lastly, I want to express my heartfelt gratefulness to my incredible family, particularly my parents, sister, and grandmother, who have been my pillars of support throughout my entire academic journey. They have supported and encouraged me to reach my full potential, and without such amazing role models and their support, this dissertation would never have been possible. I would also like to extend a special thank you to my girlfriend, Maria João. I can not thank you enough for all you have done for me. From the time I was just a freshman to the current day, you have helped me become the best version of myself. From the darkest times to the brightest days, you were always there looking after me. Whether it was the shared coffee breaks, study sessions, or trips downtown, your companionship and love permeated into every aspect of my life.

Thank you all,

José Pedro Carvalho

"Machines take me by surprise with great frequency."

Alan M. Turing

Contents

List of Figures

List of Tables

Abbreviations

A3C	Asynchronous Advantage Actor-Critic
ACER	Actor-Critic with Experience Replay
AI	Artificial Intelligence
ALE	Atari Learning Environment
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CPP	Coverage Path Planning
CTDE	Centralized Learning Distributed Execution
DAG	Directed Acyclic Graph
DDPG	Deep Deterministic Policy Gradient
Dec-POMDP	Decentralized Partially Observed Markov Decision Process
DFS	Depth-First Search
DNN	Deep Neural Networks
DQL	Double Q-Learning
DQN	Deep Q Network
DQN-PER	Deep Q Network with Prioritized Experience Replay
E-BF	Energy-Aware Back-and-Forth
E-Spiral	Energy-Aware Spiral Algorithm
GPI	Generalized Policy Iteration
GUI	Graphical User Interface
IQR	Inter-Quartile Range
LiDAR	Light Detection and Ranging
MADDPG	Multi-Agent Deep Deterministic Policy Gradient
MARL	Multi-Agent Reinforcement Learning
MAS	Multi-Agent Systems
MDP	Markov Decision Process
MSE	Mean Square Error
MRP	Markov Reward Process
Nash Equilibrium	NE
Neural Network	Neural Network
NSI	Non Size Invariant
PBRS	Potential-Based Reward Shaping
POMDP	Partially Observed Markov Decision Process
POMG	Partially Observed Markov Game
POI	Point of Interest
PPO	Proximal Policy Approximation
ReLU	Rectified Linear Unit
RL	Reinforcement Learning

SARSA	State-Action-Reward-State-Action
SGD	Stochastic Gradient Descent
SI	Size Invariant
SLAM	Simultaneous Location and Mapping
UAV	Unmanned Aerial Vehicle
UML	Universal Modeling Language

Chapter 1

Introduction

1.1 Context

The number of intelligent autonomous systems has been increasing in recent years as advances in electronics and Artificial Intelligence (AI) has allowed the systems to take on more challenging and complex tasks. The next natural step of evolution for these kinds of systems is to communicate and cooperate with each other, forming Multi-Agent Systems (MAS). Having systems composed of more than one entity (agent) has inherent advantages as it unlocks more complex tasks, coexistence with other agents while having more redundancy and overall efficiency.

Despite these advantages, MAS have a lot of complex problems to overcome before their use can become widespread. As the number of agents increases, so does the complexity of the system, having constraints imposed by limitations of the communication network, sensor uncertainty, and the inherent difficulty of decision-making over a large number of variables, some of which are not observable by the agent.

Multi-agent systems find applications in a variety of domains ranging from swarm robotics, distributed control, resource management, collaborative decision support systems, and data mining [2]. For the purpose of this dissertation, the application considered will be swarm robotics, mainly developing algorithms for robotic navigation, while giving special attention to the Coverage Path Planning (CPP) problem, where the main goal is to compute collision-free paths that pass through all points of an area or volume [6].

Coverage Path Planning is a hard problem to solve, being classified as NP-Hard in terms of complexity [7]. Naturally, the problem in a multi-agent setting is also NP-Hard, but also significantly more complex and harder to solve. In literature [6, 7, 8] there are many approaches to solving this problem in a single-agent setting, however, it is quite lacking in multi-agent solutions and most of them are heuristic or graph-based, requiring full information of the environment and having poor performance in dynamic environments.

One of the most interesting solutions that are emerging for decision-making and control in robotics is Reinforcement Learning (RL) [9], and recent advancements in this research domain

show potential to solve complex and dynamic problems [2] such as Multi-Agent Coverage Path Planning (mCPP).

The work developed in this dissertation is a novel approach to Robotic Navigation and Path Planning. It will introduce an unifying approach to robotic navigation tasks, where instead of using a specific algorithm for a specific situation, an intelligent agent with competencies in navigation is able to extract semantic information from a map representation and complete multiple navigation tasks. These tasks include Coverage Path Planning, Active-Exploration or Sensor-Based Coverage Path Planning, and Point-To-Point Path Planning either in single-agent or multi-agent scenarios.

The intelligent agents will be developed by leveraging Machine Learning techniques such as Deep Reinforcement Learning (DRL) and Multi-Agent Reinforcement Learning (MARL). As far as the author's knowledge, it will be the first work in the literature to develop a generic reinforcement learning-based algorithm for single and multi-agent robotic navigation on 2D Spaces and provide results on its capabilities to generalize to different maps, starting positions, number of obstacles, sensor type and range, tasks, and number of agents.

1.2 Motivation

In recent years, there have been incredible breakthroughs in Reinforcement Learning, mainly due to the increase in the availability of computational power and the development of Deep Learning methods [10]. These methods rely on Deep Neural Networks (DNNs), which are function approximators that enable algorithms to work on high-dimensional data, bringing significant developments to various research topics in Machine Learning.

Although there has been a recent rise in the prominence of Reinforcement Learning in the last decade, the field has already achieved impressive results decades ago. In 1995, the algorithm TD-Gammon achieved super-human performance, being capable of beating world-class players to the point where the best players now play positions inspired by TD-Gammon [11]. More recently, the AlphaGo Series marks a huge milestone in single-agent decision-making problems, as an RL agent trained through self-play achieved super-human performance in the game of Go [12], which has a search space of 10^{761}[2]. Later, in 2018, the AlphaGo algorithm was generalized to a single general algorithm, Alpha Zero, that achieved world-class levels of performance in chess, shogi, and naturally Go [13].

These impressive results in single-agent Reinforcement Learning were soon followed by significant achievements in Multi-Agent Reinforcement Learning, first by the algorithm AlphaStar, achieving a performance level higher than 99.8% of players [14] in Starcraft II. This multiplayer game has its own professional league, in which the agent has to control thousands of units and make high-level economic decisions in a planning horizon that is thousands of actions long while having only imperfect information [14]. Another impressive feat of MARL is in the game Dota2, in which the RL agent has not only to cooperate with four agents that are part of his team but also to compete with five agents in the enemy team. In this context, the OpenAI Five achieved

super-human performance, defeating the 2019 reigning world champions, once again showing the potential of Reinforcement Learning in difficult decision-making problems [15].

Motivated by the successes of Reinforcement Learning in these problems, this dissertation aims to extend the RL methodologies to Robotic Navigation Tasks. This class of problems has many applications that are useful for society and can be greatly improved by the combination of introducing multiple agents and reinforcement learning. Despite this document being focused on navigation as a whole, most of the focus, especially on the Multi-Agent setting, will be towards Coverage Path Planning, as it is the most complete and complex task, and an intelligent agent capable of performing it a near-optimal level should be able to complete similar, less demanding navigation tasks. The next subsection will present applications of Multi-Agent Coverage Path Planning to contextualize the importance of this work further.

1.2.1 Applications

Coverage path planning is a common task in robotics that consists of computing collision-free paths that cover all points of interest (POI) of a determined area or volume. It has varied applications in cleaning robots [16], agriculture [17], surveillance and monitoring [18], photogrammetry [19], search and rescue [20], exploration and mapping [21], structural inspection [22], and wireless sensor networks [23] to name a few. Some more specific applications that were the motivation for the work in this dissertation will be presented below in more detail.

1.2.1.1 Wildfire Monitoring and Prevention

In the year 2022, Europe experienced the highest number since 2006, with a burnt area that is over 8600 km^2 [24], corresponding to more than four times the size of the Porto Metropolitan Area. Besides the economic damage caused by wildfires, in 2021 alone, more than 1000 km^2 were burnt in protected areas of Europe's Natura 2000 network, causing long-lasting effects in Europe's biodiversity reservoirs [24].

In the JRC Technical Report - Forest Fires in Europe, Middle East, and North Africa 2021 [24], it is mentioned that 96% of wildfires are man-made. Therefore, most of the measures to mitigate the wildfire problem are preventive. One of the mentioned measures is "the development of early warning and information systems for wildfires" [24], in which a solution based on UAV long-term coverage of critical areas would help towards this cause.

This research problem was already approached in literature with classical methods such as Kalman Estimation and swarm consensus protocols [25] and leader-follower swarm topology [1]. Another approach based on the Deep Reinforcement Learning techniques was also explored in [26].

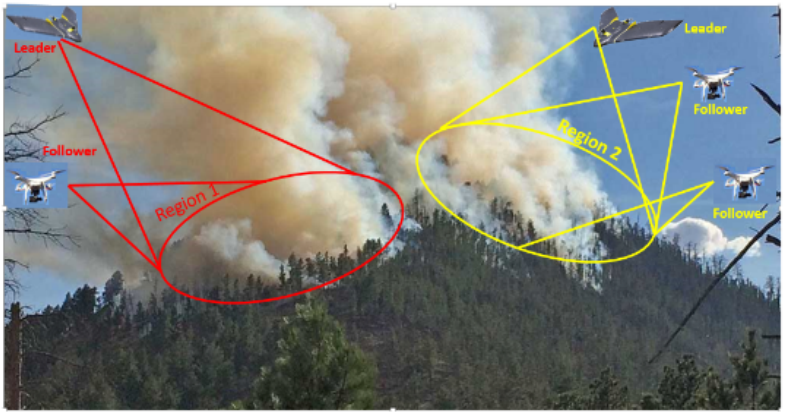


Figure 1.1: Example of Swarm of UAVs for Wildfire Monitoring [1]

1.2.1.2 Long-Term Monitoring and Patrolling

One advantage of using multiple robots for coverage path planning is the long-term potential of operations and the cooperation between agents with different attributes and capabilities. The usage of reinforcement learning techniques can be of great value for these types of problems, as it can merge the cooperation between agents with the extraction of features from sensor data.

In [27], the authors develop a DRL-based algorithm for patrolling areas with different coverage requirements. On the other hand, in [18], a learning-based algorithm for persistence surveillance was developed, with an emphasis on a more military application, where the agent tries to find targets in an area while being risk-aware of the surrounding environment.

Another interesting application for long-term monitoring is data harvesting. In this task, the goal of the robots is to collect data from several Internet of Things devices distributed in the environment. It is interesting in scenarios where the devices are in an area without connectivity, and the robot can operate as a data mule. In [28], the author developed a DRL algorithm that can perform the data harvesting task and also perform a generic coverage path planning task, demonstrating that the reinforcement learning framework has the potential for adapting to dynamical scenarios and is capable of generalization.

1.3 Objectives

In light of the context and motivation outlined in the previous sections, the primary objective of this dissertation is to develop algorithms for robotic navigation tasks based on reinforcement learning methodologies. From these tasks, cases of point-to-point path planning, coverage path planning, and active exploration will be considered, with special attention to the coverage path planning task. The problems will be formulated in a comprehensive and unifying view, where the algorithm should be able to complete any of the mentioned tasks with proficiency. The robotic

navigation will also be extended to cooperative multi-agent scenarios to augment the algorithm's potential. Overall, the goals of this dissertation can be summarized in the following items:

1. Conduct a literature review on Reinforcement Learning Methodologies, Coverage Path Planning approaches, and the intersection of both.

2. Develop a sandbox Gym-like [1] environment for robotic navigation tasks on 2D Grid Worlds.

3. Study and formulate a solution based on classical reinforcement learning methods for sensor-based coverage path planning that can cope with the large state space.

4. Design and develop a Deep Reinforcement Learning framework for generic robotic navigation, test its generalization capabilities and proficiency on different scenarios and compare it to other state-of-the-art approaches on the Coverage Path Planning task.

5. Extend the single-agent framework to a Multi-Agent Reinforcement Learning method focused on solving Multi-Agent Coverage Path Planning.

1.4 Contributions

From the work developed in this dissertation, the following contributions to the field of Artificial Intelligent Robotics were made:

1. Conducted a review of the existing literature on Reinforcement Learning, Multi-Agent Reinforcement Learning, Coverage Path Planning, and their intersection.

2. Developed an open-source customizable Gym-like Reinforcement Learning environment that can be used for any generic navigation task on 2D Grid Worlds [2].

3. Developed an algorithm based on Classical Reinforcement Learning methods and Heuristics to solve the sensor-based coverage path planning task without using an explicit map representation.[3]

4. Formulated the Robotic Navigation Problem as a unified Partial Observed Markov Decision Process with a generalizable and map-size invariant value function.

5. Implemented the state-of-the-art Deep Reinforcement Learning Method Rainbow DQN and customized the architecture and training algorithm to fit the robotic navigation paradigm.

6. Introduced a dataset for benchmarking of Coverage Path Planning algorithms [4].

[1] OpenAI Gym Documentation: https://gymnasium.farama.org/
[2] https://gitlab.com/jpfpc/drl_cpp/-/tree/main/src/Environment?ref_type=heads
[3] https://github.com/Jose-PCarvalho/GridWorld-RL
[4] https://gitlab.com/jpfpc/drl_cpp/-/tree/main/maps/datasets?ref_type=heads

7. Developed an algorithm that can perform single-agent point-to-point path planning, coverage path planning, and active exploration at a near-optimal level from any scenario. This algorithm outperforms other state-of-the-art approaches.

8. Extended the single-agent formulation to a Decentralized Partial Observed Markov Decision Process for the Multi-Agent Approach.

9. Introduced various reward structures for the cooperative coverage path planning problem and compared their performance.

10. Adapted the Rainbow DQN for multi-agent settings using the Parameter Sharing Technique.

11. Developed a Multi-Agent Reinforcement Learning based algorithm that can perform cooperative coverage path planning without explicit limitation on the number of simultaneous agents. The algorithm works with minimal information sharing and synchronization and displays significant cooperation skills between agents.

Additionally, a conference paper has been derived from this research and accepted for publication at the 23rd IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC2023) with the title "A Reinforcement Learning Based Online Coverage Path Planning Algorithm" [29]. Furthermore, two additional journal publications on the single-agent and multi-agent approaches detailed in this dissertation are currently in progress and will be published in due course.

1.5 Dissertation Outline

Besides the Introduction, the remainder of this document is structured as follows:

- Chapter 2 presents the necessary theoretical background on Machine Learning, Reinforcement Learning, and Multi-Agent Reinforcement Learning.

- Chapter 3 analyzes the existing literature on classical methods for coverage path planning and the current state-of-the-art Reinforcement Learning algorithms for coverage path planning. It also identifies gaps and opportunities for further research.

- Chapter 4 delineates the system architecture and the software developed for the Reinforcement Learning Environment.

- In Chapter 5, a case study on Tabular Temporal Differences Reinforcement Learning Algorithms for sensor-based Coverage Path Planning without Explicit Map Representation is presented and analyzed.

- Chapter 6 delves into the development of a Deep Reinforcement Learning-based solution for generic robotic navigation, presenting results and comparisons with other state-of-the-art approaches.

- Chapter 7 presents the culmination of all the research conducted in this dissertation, extending the single-agent approach to a versatile multi-agent reinforcement learning algorithm for Coverage Path Planning.
- Chapter 8 provides the main conclusions drawm from this dissertation and outlines future work.

Chapter 2

Background

This chapter aims to give the reader a comprehensive explanation and description of the theoretical background necessary through this document. With this objective in mind, this chapter is divided into four sections. First, Section 2.1 gives a brief introduction to the fundamental concepts of Machine Learning algorithms. Afterward, Section 2.2 introduces serves as an introduction to Reinforcement Learning and its theoretical background. That is followed by Section 2.3, where the previous concepts are expanded into the main learning framework that is used in this dissertation, Value-Based Reinforced Learning Methods. This chapter ends by exploring the Multi-Agent Reinforcement Learning paradigm in Section 2.4.

2.1 Machine Learning

The concept of Machine Learning (ML) can be seen as algorithms that transform raw data into knowledge [2]. A simple example of such an algorithm is a computer program that receives as input an image of a cat or a dog and gives as output the animal that was identified (Figure 2.1). In recent years, this domain of research has seen multiple breakthroughs, mainly due to the increasing computational power available and the introduction of Deep Neural Networks (DNNs) as a function approximator [10].

In Machine Learning, there are three types of learning algorithms: Supervised Learning, Unsupervised Learning, and Reinforcement Learning (RL). The example given earlier is a classic usage of supervised learning, being that these kinds of algorithms are the ones that are most commonly associated with ML. In terms of differences between the first two classes of learning, the main difference is the existence of labeled data, where supervised learning algorithms need labeling of the data to extract hidden knowledge. In contrast, unsupervised learning algorithms try to find hidden patterns in the data in order to analyze and cluster the data into groups. Finally, Reinforcement Learning operates within an interaction loop with an environment. This loop provides raw data to the algorithm, which it subsequently analyses to make informed decisions. This process creates a feedback loop where the decisions made based on previously acquired knowledge shape the algorithm's interpretation of new data.

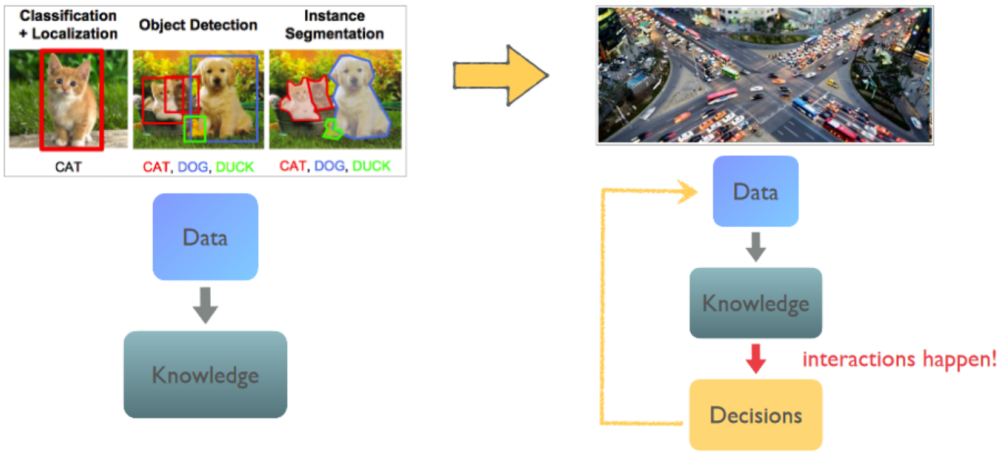


Figure 2.1: On the left is an example of a Supervised Learning algorithm, where the task is to exclusively get knowledge from data, and on the right, an example of a Reinforcement Learning algorithm where a decision-making process takes place and interacts with the data. [2]

This section will provide the essential background for understanding various Machine Learning algorithms. The emphasis will be on the foundational concepts of a Deep Learning framework, particularly concerning Artificial Neural Networks (ANNs), and their utilization in the creation of learning algorithms.

2.1.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are data processing systems consisting of nodes and connections in an architecture inspired by the cerebral cortex portion of the brain [30]. In the context of machine learning, ANNs are formally described as weighted directed-acyclic graphs, where each edge of the graph carries a specific weight. From a mathematical perspective, a Neural Network (NN) can be considered a parameterized non-linear function $f(x, \theta)$, where the parameter θ can be tuned to approximate a specific target function.

2.1.1.1 The Perceptron

The fundamental building block of an Artificial Neural Network is the Perceptron or neuron, a simplified mathematical model of a biological neuron, first introduced by Frank Rosenblatt [31]. A perceptron functions as a linear map from $\mathbb{R}^N \to \mathbb{R}$, transforming an input vector $x \in \mathbb{R}^N$ into an output scalar value $o \in \mathbb{R}$, resulting in:

$$o = \sum_i w_i x_i + b, \tag{2.1}$$

where w and b are the weights and biases which can be adjusted to fit the target function. In this document, the set of weights and biases, or in other words, the parameters of the network, will be referred to as θ for simplicity of notation.

The linear transformation depicted in (2.1) can be followed by a non-linear activation function to enhance the approximation capabilities of the neural network. An illustration of the perceptron can be seen in Figure 2.2.

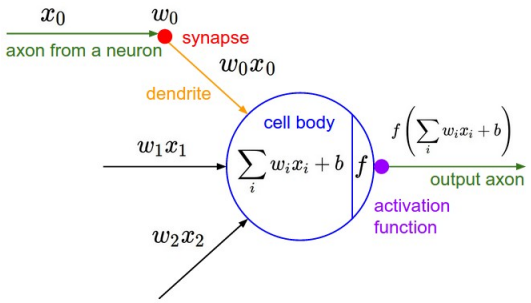


Figure 2.2: Visualization of a Perceptron[1]

.

2.1.1.2 Activation Functions

While neurons serve as linear maps converting an input vector to an output scalar, the effectiveness of a neural network can be enhanced by incorporating non-linearities. This nonlinearity is introduced by using an activation function f on the neuron's output $f(o)$. Although any mathematical operation can serve as an activation function, it should be differentiable, or at least piece-wise differentiable, to facilitate gradient computation. Some of the most common activation functions are:

Sigmoid: The sigmoid function, often used as an activation function in neural networks, has characteristics resembling the behavior of biological neurons due to the saturating nature of the function. This function is a special case of the logistic function and can be defined as:

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

However, this function has certain limitations. It's not zero-centered, and the gradient approaches zero at both ends of the function (saturating regions), which can lead to slower convergence during the training process. These issues are often referred to as the vanishing gradients problem.

ReLU: The Rectified Linear Unit (ReLU) has emerged as one of the most essential and popular activation functions in recent years. Not only is it less computationally intensive than many other functions, but it also significantly accelerates the convergence speed of the learning process [32].

$$f(x) = \max(0,x) \tag{2.3}$$

[1] Image taken from http://cs231n.github.io/neural-networks-1/

However, ReLU has a limitation. Because the function outputs zero for all negative inputs, in some situations, a parameter update can cause a neuron to output zero regardless of the input – a state often referred to as a "dying ReLU". This can significantly slow down the learning process and can cause non-convergence. To mitigate this issue, variants of the ReLU function, such as the **Leaky ReLU**, have been proposed. Leaky ReLU introduces a small slope α on the left half plane, which can prevent this problem. However, the effectiveness of this technique is not consistent.

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha x & \text{otherwise,} \end{cases} \tag{2.4}$$

Tanh: The tanh function has very similar properties to the sigmoid but is, in general, an improvement over it. It maps the inputs into an output bounded in $[-1,1]$, making it zero-centered. However, the saturation makes it so that the gradients will still be small in those regions.

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

The selection of an activation function is often not trivial and remains an active area of research.

2.1.1.3 Architecture of a Neural Network

A neural network can be thought of as a collection of neurons organized into multiple layers. Each neuron takes in inputs, processes them, and then forwards the output to units in the subsequent layer. Crucially, these connections are directed (meaning information flows in one direction), and no loops or cycles are allowed. This arrangement, which prevents the creation of infinite loops during the input feeding process, makes the network a type of structure known as a directed acyclic graph (DAG). A typical neural network has the following organization of layers:

- **Input Layer**: This layer serves as the interface between the neural network and external input. Typically, the number of neurons in this layer corresponds to the number of input scalars.
- **Hidden Layer**: It is considered a hidden layer, any layer that is in between the input and output layers. Its size can be arbitrarily big or small, depending on the needs of the problem.
- **Output Layer**: The final layer of the network, its size reflects the dimension of the function the network is approximating. Its activation function needs to be selected accordingly. For instance, in a binary classification problem, a single neuron with a sigmoid function might be used.

Whenever all the neurons of a previous layer are connected to every neuron of the next layer, the architecture is known as a fully connected network (or layer if it is not consistent throughout the whole network). A fully connected network example can be visualized in 2.3.

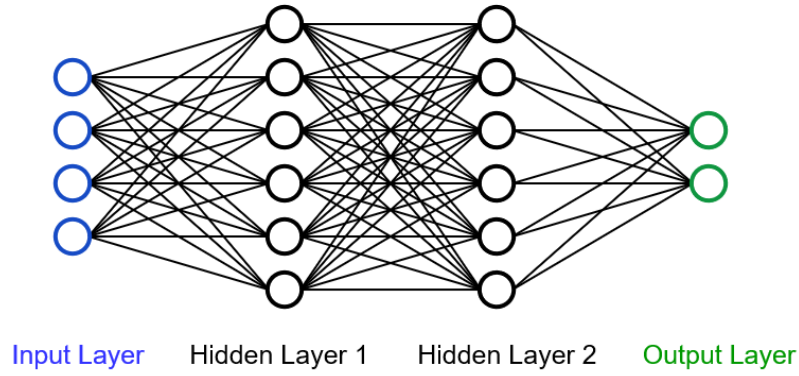


Figure 2.3: A typical fully connected neural network. It is a 3-layer network with four inputs, two hidden layers of seven neurons each, and two neurons in the output layer.

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specific class of neural networks optimized for processing structured grid data, notably images. They utilize a set of convolutional filters or kernels with learnable weights, making them especially effective in feature extraction from structured data. This approach is superior to traditional methods that rely on manually tuned convolution filters and hand-picked features, as CNNs can also learn the most beneficial features directly from the data. Similar to Artificial Neural Networks, CNNs are organized in layers, each hosting its unique set of filters. The filters have key hyperparameters that the user needs to define. These parameters include the size of the kernel, the stride (which determines the step size that the filter moves after each computation), the type of padding (if any), and the number of filters in the layer.

It is important to note that while convolution filters can theoretically be n-dimensional, the ones most commonly available in popular Machine Learning libraries are 1D, 2D, and 3D. Among these, 2D convolution filters are the most widely used due to their applicability in image processing and computer vision tasks. Figure 2.4 provides a visualization of the effect of a 2D kernel filter on an image.

After the convolution operation, it is common practice to apply an operation to the resulting values. This process, known as pooling, can be understood as a form of downsampling that helps to make the output invariant to small translations and rotations and reduces the amount of data to be processed by following layers. The most common types of pooling are Average Pooling and Max Pooling, which respectively compute the average and maximum value of a particular patch of the output.

In Average Pooling, an averaging kernel is used in the output, calculating the average of various patches, which provides a smoothing effect. Conversely, Max Pooling selects the maximum

[2]Image Taken from https://openlearninglibrary.mit.edu/courses/course-v1:MITx+6.036+1T2019/courseware/Week8/ .

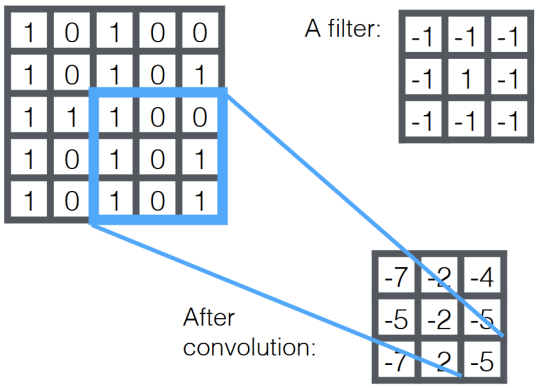


Figure 2.4: A convolution kernel of size 3×3 being used on a 5×5 image.[2]

value from each patch, preserving the most significant features while reducing the spatial size of the output.

In addition to pooling operations, activation functions can also be applied to introduce non-linearity into the model. Due to its efficacy and computational efficiency, the ReLU function is one of the most commonly used activation functions in convolutional networks. These additional steps contribute to the ability of CNNs to learn complex patterns and features from input data effectively.

2.1.3 Learning Process

The remaining question is how to adjust the learnable parameters θ to achieve the desired purpose of the network. Then, the learning process of any machine learning algorithm can be seen as an optimization problem, where the objective is to find the optimal set of parameters θ for the specific approximation task. This is usually done by minimizing a Loss Function.

2.1.3.1 Loss Functions

Considering that the neural network represented by $\hat{y} = f(x, \theta)$ approximates the function y, one can define a loss or cost function to quantify how well the neural network fits the target function. Many candidate loss functions exist, such as the Mean Square Error (MSE), Cross-Entropy Loss, Kullback-Leibler Divergence, etc. Now focusing on the most simple and popular one, the Mean Square Error, consider a vector $y \in \mathbb{R}^N$ consisting of N data points sampled on the target function, and the vector $\hat{y} \in \mathbb{R}^N$ with the N corresponding data points approximated by the neural network, one can define the Loss Function $\mathrm{L}(\theta)$ as:

$$\begin{aligned} L(\theta) &= \frac{1}{N}\sum_{i=1}^{N}[y_i - \hat{y}_i]^2 \\ &= \frac{1}{N}\sum_{i=1}^{N}[y_i - f(x,\theta)]^2 \end{aligned} \tag{2.6}$$

Some small variations to the MSE loss can be made, leading to other loss functions, such as the Smooth L1 Loss. This function can make the loss less sensitive to outliers and in some cases prevents exploding gradients [33]

$$L(\theta) = \begin{cases} 0.5 \cdot (y - \hat{y})^2 & \text{if } |y - \hat{y}| < \beta, \\ |y - \hat{y}| - 0.5\beta & \text{otherwise.} \end{cases} \tag{2.7}$$

In this function, a squared loss term (analogous to MSE) is used when the absolute difference between the target and predicted values (L1 norm) falls below a predefined threshold, denoted by the hyperparameter β. Conversely, an L1 term is applied when this difference exceeds β. The utility of this function lies in its ability to combine the benefits of both MSE and L1 loss types, thereby accommodating both small and large errors effectively in different scenarios.

2.1.3.2 Backpropagation

After defining a loss function, the problem can be transformed into an optimization task. Here, the optimization objective can be represented as follows:

$$\min_{\theta} L(\theta), \tag{2.8}$$

where the goal is to identify the optimal set of parameters, θ, that would minimize the loss function. A common method to achieve this is by using gradient-based techniques such as Stochastic Gradient Descent (SGD) or ADAM. Assuming the Mean Squared Error (MSE) as the loss function as defined in equation 2.6, the gradient with respect to θ can be expressed as:

$$\nabla_\theta L(\theta) = -\frac{2}{N}\sum_{i=1}^{N}(y_i - f(x_i,\theta)) \cdot \nabla_\theta f(x_i,\theta). \tag{2.9}$$

In the above equation, $\nabla_\theta f(x_i,\theta)$ denotes the derivative of the network output with respect to its parameters. Considering the gradient defined in equation (2.9), it is possible to do a gradient descent step by following:

$$\theta_{i+1} = \theta_i - \alpha \nabla_\theta L(\theta), \tag{2.10}$$

in which the hyperparameter $\alpha > 0$ is the gradient-step size or learning rate. This value must be small enough to ensure smooth parameter updates yet large enough to facilitate a rapid convergence. Please note that a negative sign is added to the gradient term to follow the direction that leads to a decrease in the loss function, which is the aim of the optimization process.

The derivative of the network with respect to its parameters $\nabla_\theta f(x_i, \theta)$ is calculated using a technique called backpropagation. The name "backpropagation" comes from how the method operates: it calculates the gradient by moving backward through the network, starting from the output layer to the input layer, applying the chain rule to compute the derivative of each layer's output with respect to its input. This process is automated in automatic differentiation libraries such as PyTorch and TensorFlow[3] [4].

2.1.3.3 Strategies for Learning

A spectrum of learning strategies exists in most Machine Learning algorithms, particularly in training neural networks. At one end of this spectrum is Batch Learning, with Online Learning standing at the opposite end.

Batch Learning refers to a training approach in which the entire dataset is utilized to compute the gradient of the loss function for each iteration during the model's training process. In other words, model parameters are updated only once the entire dataset has been processed. This method offers better computational efficiency as large operations can be parallelized or distributed. However, batch learning may not be an adequate choice when dealing with rapidly changing datasets or scenarios that require a real-time response.

On the other hand, Online Learning involves a process where model parameters are updated after each individual data point in the dataset is processed. Generally, these methods are more computationally demanding, hence slower. They also typically exhibit slower rates of convergence.

Despite the polarities of these two strategies, a middle ground does exist - Mini-batch Learning. In this strategy, the hyperparameter M for mini-batch size replaces the need to process the whole dataset or a single data point at a time. Instead, a batch of size M is used to update the model parameters. This approach enables a compromise between the advantages and disadvantages associated with batch and online learning methods.

2.1.3.4 Transfer Learning

Transfer Learning is a method used in Machine Learning where a model initially developed for a specific task is repurposed to perform a related yet distinct task [34]. The objective of this technique is to leverage the knowledge gained during the initial task to enhance performance and potentially accelerate the parameter optimization process during the subsequent task.

The level of similarity between the initial and subsequent tasks largely determines the degree of modification required for the model and the amount of time needed to re-adapt it. It is also important to note that the model architecture must be suited for both situations. For example, if the input type or shape changes, the model might have to suffer more substantial changes that might undermine the utility of Transfer Learning.

[3] https://pytorch.org/ , https://www.tensorflow.org/ .

[4] These libraries are more than just automatic differentiation libraries. They are comprehensive machine learning frameworks that provide end-to-end tools for building and deploying machine learning models.

2.2 Reinforcement Learning

Reinforcement Learning (RL), unlike other Machine Learning algorithms, is intrinsically goal-oriented. It is essentially a decision-making framework wherein an agent seeks to understand the system's behavior and determine how it should act optimally to achieve a specified goal, usually through a trial-and-error approach. This learning process can be succinctly represented through the following sequence of steps:

- Observe the environment through raw data (observation).
- Deciding the optimal action based on the most recent observation, the existing model, and the overarching goal.
- Update the model based on the taken action, the previous and new state, and the received reward.

A visualization of the interaction loop between an RL agent and the environment can be seen in Figure 2.5. It is essential to note that the "model" is inherent to the agent and serves as its decision-making tool, and is not necessarily a model of the environment. Furthermore, this model is continuously updated throughout the agent's interactions with the environment.

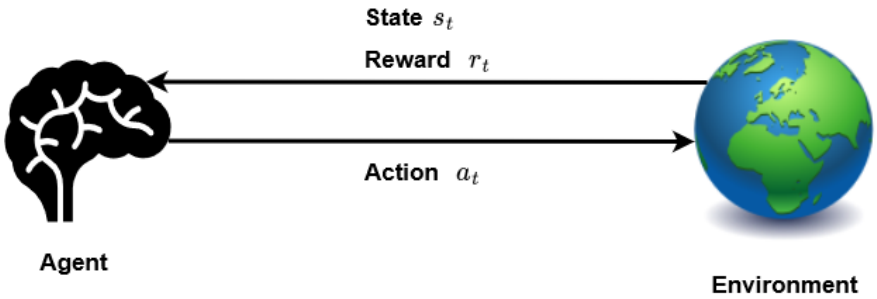


Figure 2.5: Interaction model between an RL agent and a generic environment.

2.2.1 Markov Decision Processes

As mentioned, a reinforcement learning agent tries to find the optimal set of actions that maximize its long-term reward, which can be seen as achieving the goal in an optimal way. This problem is formalized as a Markov Decision Process (MDP), which can be defined by the tuple $\langle \mathbb{S}, \mathbb{A}, P, R, \gamma \rangle$ [2]:

- $\mathbb{S}$ - Set that contains all states s of the environment.
- $\mathbb{A}$ - Set that contains all possible actions a of the environment.
- P: $\mathbb{S} \times \mathbb{A} \to \mathbb{S}$ - State Transition Probability Function; Given a state $s \in \mathbb{S}$ and an action $a \in \mathbb{A}$, the probability of the state being $s' \in \mathbb{S}$ in the next time step is given by P.

- R: $\mathbb{S} \times \mathbb{A} \times \mathbb{S} \to \mathbb{R}$ - Reward function; Given the action-state pair $(s,a) \in (\mathbb{S},\mathbb{A})$ and the next state $s' \in \mathbb{S}$, the function returns a scalar value corresponding to the reward of the state-transition.

- γ - Discount Factor; a scalar value in $[0,1]$, representing the discount factor for future rewards. Note that if $\gamma = 1$, the value of any future reward is equivalent to the next immediate reward. Conversely, if $\gamma = 0$, all future rewards are disregarded, considering only the immediate reward.

Markov Decision Processes, therefore, provide a robust framework for modeling decision-making problems in stochastic domains. They characterize an agent that synchronously interacts with an environment, receiving the environmental state as input and returning an action as output, which in turn influences the state. It should be highlighted, however, that while the effect of the action on the future state embodies inherent uncertainty, the current state is entirely deterministic, as the agent maintains full observability.

2.2.1.1 Partially Observed Markov Decision Processes

In certain scenarios, the agent lacks full observability of the state, resulting in Partially Observed stochastic domains. Drawing parallels between stochastic optimization and modern control theory, there are control challenges wherein obtaining measurements over the entire state space is unfeasible. As such, observers and state-estimation techniques are used to retrieve the missing information. Analogously, in situations of state uncertainty within a Markov Decision Process, the problem is formulated as Partially Observed Markov Decision Processes (POMDP) [35], defined by the tuple $\langle \mathbb{S}, \mathbb{A}, P, R, \Omega, \mathscr{O}, \gamma \rangle$:

- $\mathbb{S}$, $\mathbb{A}$, P, R, γ - Describe the Markov Decision Process.

- Ω Set that contains all the observations the agent can experience.

- $\mathscr{O} : \mathbb{S} \to \Omega$: Is the observation function, which maps the current state s to the observation of the agent o.

It is worth noting, however, that the observation function $\mathscr{O}$ is modeled as in [28], which is a simplified approach to POMDPs, where the observation solely depends on the current state and the agent's observation mechanisms. In a more comprehensive approach [35], the observation function can be modeled as $\mathscr{O} : \mathbb{S} \times \mathbb{A} \to \Pi(\Omega)$, which yields a probability distribution over possible observations for any action a and subsequent state s'. Hence, the probability of observing o given the subsequent state s' and the executed action a is denoted as $O(s',a,o)$. Throughout this document, the simplified approach will be employed, and for the sake of simplifying notation, the state s will represent all types of MDPs, unless explicitly stated otherwise.

2.2.1.2 Markovian Propriety

A Markov Decision Process is a specific formulation of a class of stochastic processes known as Markov Chains [36]. Markov Chains provide an interesting framework to study and analyze discrete event systems and stochastic processes in general.

The main characteristic of a Markov Chain is that it is memoryless, which in this situation is also called Markovian Propriety. In practical terms, it implies that the probability P of the following state being $s_{t+1} \in \mathbb{S}$ only depends on the current state s.

$$P[S_{t+1} = s_{t+1} | S_t = s_t, S_{t-1} = s_{t-1}, ..., S_0 = s_0] = P[S_{t+1} = s_{t+1} | S_t = s_t] \tag{2.11}$$

2.2.1.3 Interaction between Agent and Environment

Unlike a regular Markov Chain, an MDP is conditioned by the interaction between an agent and the environment. For the sake of simplicity, it will be assumed that the agent can observe the whole state space. As for the interaction, at each time step t, the agent observes the environment state s_t, and executes an action a_t. This action is chosen based on the policy π of the agent. Without loss of generality, a policy can be defined as a stochastic function that maps states into actions $\pi : \mathbb{S} \rightarrow \Delta(\mathbb{A})$, where $\Delta(.)$ denotes the probability simplex [2]. Meaning that one can specify the probability distribution over actions given states as:

$$\pi(a|s) = \mathbf{P}[\mathbf{A_t} = \mathbf{a_t} | \mathbf{S_t} = \mathbf{s_t}] \tag{2.12}$$

After every action, the environment transitions into the next state s_{t+1} with probability $P(s_{t+1}|s_t, a_t)$, and a scalar reward is given to the agent based on the reward function $R(s_t, a_t, s_{t+1})$. A general finite interaction between an agent and the environment can be seen in Figure 2.6.

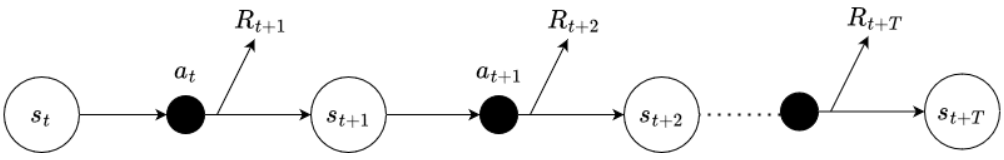


Figure 2.6: Sequence of state, action and rewards based on the interaction between the agent and the environment.

2.2.1.4 Value Functions

As seen in the previous subsection, at each time step t, the agent receives a scalar reward R_t. When considering the evolution of the process over a larger time horizon, one can look into the cumulative reward obtained by the agent. As such, the cumulative reward G_t at the time step t is

defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.13}$$

The discount factor γ, gives the possibility of changing the value of future rewards. If $\gamma = 1$, then future rewards are valued just as much as the next, and if $\gamma = 0$ then only the next reward is considered. This factor is natural for these kinds of learning algorithms, as it is a natural behavior in decision-making and cognitive models found in animals and humans to give more value to proximal rewards [3].

Without loss of generality, a state value-function $v(s) : \mathbb{S} \to \mathbb{R}$ can be defined as:

$$v(s) = \mathbb{E}[G_t | S_t = s] \tag{2.14}$$

The state value function is the value of cumulative rewards that the agent is expected to receive if it is in state s at a time t. As the analysis is based on MDPs [5], the value function is dependent on the policy that the agent is following, and therefore the value function is usually defined as $V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$. In an intuitive way, this function tells the agent the value, as in the future cumulative rewards, of being in the state s and following the policy π.

On the other hand, one might also define the state-action value function that gives information regarding how good is taking action a when in state s, and then continue following policy π as $Q_\pi(s,a)$:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{2.15}$$

These equations can be expressed recursively by using the Bellman Expectation function [37]. It decomposes the value functions into immediate reward plus the discounted value function of the successor state. For the state-value function, it can be written as:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \sum_{n=1}^{\infty} \gamma^n R_{t+n+1} | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(s_{t+1}) | S_t = s] \end{aligned} \tag{2.16}$$

As for the state-action value function, yields:

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi[R_{t+1} + \sum_{n=1}^{\infty} \gamma^n R_{t+n+1} | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) | S_t = s, A_t = a] \end{aligned} \tag{2.17}$$

2.2.1.5 Solving Markov Decision Processes

By analyzing the Markov Decision Process formulation, one can infer that the overall goal of using this setup is to find a policy that maximizes the overall expected cumulative reward. This

[5] The value function presented in equation 2.14 is derived from Markov Reward Processes (MRPs), which would be the equivalent of an MDP where the agent can not take any actions, making the state transitions purely stochastic. However, since any MDP can be reduced to an MRP, it is the most generic way to represent a value function.

framework intuitively gives rationality to the agents, as based on the expected utility theory [38], rationality can be modeled as the maximization of an expected value, in other words, a rational agent will always choose the action that maximizes the expected utility.

With this goal in mind, and using the Bellman Optimality Equation, the state-value function can be re-written as:

$$\begin{aligned} v_*(s) &= \max_a R_s^a + \gamma \sum_{s' \in \mathbb{S}} P_{ss'}^a v_*(s') \\ &= \max_\pi v_\pi(s) \\ &= \max_a Q_*(s,a) \end{aligned} \tag{2.18}$$

For the case of the state-action value function, we obtain:

$$\begin{aligned} Q_*(s,a) &= R_s^a + \gamma \sum_{s' \in \mathbb{S}} P_{ss'}^a \max_{a'} Q_*(s',a') \\ &= \max_\pi Q_\pi(s,a) \end{aligned} \tag{2.19}$$

Reinforcement Learning is one of the methods used to solve Markov Decision Processes, where the agent learns the optimal policy through trial and error. The process is based on turning the information gained through interaction with the environment into knowledge about the dynamics of the environment. The following subsection will present a brief overview of the existing reinforcement learning methods.

2.2.2 Reinforcement Learning Methods

Reinforcement Learning algorithms can be seen as a subset of Machine Learning concerned with learning to make a finite set of decisions over an environment. In this setting, there are two main classes: 1) One that can learn to make decisions without any explicit knowledge (a model) of the environment, which is model-free; and 2) those that use a model, learned or not, to plan a solution for the problem, being this approach called model-based.

A model-based approach to Reinforcement Learning can naturally seem the better choice, as having a model may allow one to transfer knowledge to different tasks and to plan a task better than a model-free approach could. However, having an accurate model of the environment can be a challenge in itself, and learning one that is imperfect can be worse than having none to start with, as one can argue that directly sampling from the environment will provide more reliable information and in general be less computationally intensive [39]. Nonetheless, model-based can be significantly more sample-efficient, especially in high-dimensional environments.

From a psychology perspective, these two approaches can be linked to different aspects of intelligence [40]. Model-Free Reinforcement Learning relies on trial and error to learn learning to get a grasp on the environment dynamics, putting more emphasis on a reactive decision-making type of thinking. On the other hand, Model-Based Reinforcement Learning relies on understanding the environment on a deeper level and making decisions based on planning with the knowledge that the agent has of the environment.

The Reinforcement Learning taxonomy is illustrated in Figure 2.7. The following subsections will provide an overview of existing model-based and model-free methods. Note, however, that this document will mainly focus on model-free methods. This choice is due mainly to model-based methods being underdeveloped, especially in multi-agent domains [2], while keeping in mind that a model-based approach could be interesting for a Coverage Path Planning algorithm with complete information on the environment because of the planning component.

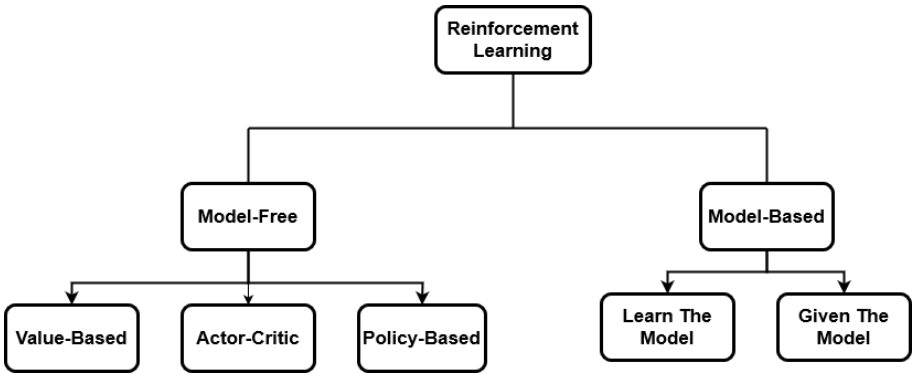


Figure 2.7: Reinforcement Learning Taxonomy.

2.2.3 Model-Based Reinforcement Learning

When solving decision-making problems, one recurring theme is to plan ahead. In reinforcement learning literature, a class of methods explicitly has this capacity: model-based. In this case, the expression to plan will refer to any algorithm that can use computational means to improve its decision-making without requiring sampling the environment for data, which on the other side of the spectrum, "to learn" is to depend on new sampled information to update the decision-making process [39].

The capacity to plan is due to the fact that the agent has a model. This model can be seen as a function that takes the state and action as inputs and outputs what would be sampled in the next timestep: the next state and reward. This can be seen as an advantage when sampling the environment is costly or risky, for example, in robotic applications. By repeatedly sampling a model, an agent can experiment with multiple trajectories over the state space and choose the best one.

The model-based methods are divided into two classes regarding how the model is obtained: Given The Model or Learn The Model. The first, as the name suggests, the agent is given a model at the beginning of training and sticks to it during the process. These models are usually handcrafted or learned by other methods, such as supervised learning. The latter are methods where the agent constructs a model of the environment from the data that is sampled from the environment. This approach can be the most interesting as it's the most versatile and the most similar to a concept of general artificial intelligence.

In algorithm 1 [39], a generic model-based reinforcement learning is provided.

Algorithm 1 Model-based reinforcement learning

Input: state sample procedure d, policy π, predictions v, environment E
Input: model m (if given, otherwise initialize arbitrarily)
Get initial state $s \leftarrow E$
for iteration $\in \{1,...,K\}$ **do**
  for interaction $\in \{1,..,M\}$ **do**
    Generate action: $a \leftarrow \pi(s)$
    Generate reward, next state: $r, s' \leftarrow E(a)$
    $m, d \leftarrow$ UPDATE MODEL(s,a,r,s')
    $\pi, v \leftarrow$ UPDATE AGENT(s,a,r,s')
    Update current state: $s \leftarrow s'$
  end for
  for planning step $\in \{1,...,P\}$ **do**
    Generate state, action $\tilde{s}, \tilde{a} \leftarrow d$
    Generate reward, next state: $\tilde{r}, \tilde{s'} \leftarrow m(\tilde{s}, \tilde{a})$
    $\pi, v \leftarrow$ UPDATE AGENT$(\tilde{s}, \tilde{a}, \tilde{r}, \tilde{s'})$
  end for
end for

By doing slight variations on this generic algorithm, it is possible to implement various model-based RL methods from the literature [39]. It is also important to note that for Given The Model methods, the update of the model step might not be necessary. Furthermore, algorithms might not treat the update of the agent step the same way depending on if the information is generated from the model or sampled. In the case they are both treated the same, the prediction is the state-action values, and the update is the same as in Q-Learning, then the obtained algorithm is Dyna-Q [41, 42].

2.2.4 Model-Free Reinforcement Learning

Model-Free Reinforcement Learning methods appear as a valuable tool for solving many real-world problems. Their usefulness stems from the fact that the agent does not need to have any information about the environment, discarding the model entirely. This approach is beneficial in cases where the model is too complex, the dynamics are unknown, or when the tasks performed by the agent are challenging to replicate with model-based algorithms.

An RL agent learns how to achieve the optimal policy by exploiting the information that it receives by interacting with the environment. RL algorithms can be categorized into value-based and policy-based methods based on how this process is done. Either the agent can directly search the policy space to find the best course of action or learn the value of each action and then act greedily with respect to the value. The first approach is called policy-based, and the latter is value-based. There is also a mixture of both concepts, the actor-critic methods, where an actor searches the policy space, and a critic evaluates the state-action pairs for credit assignment. This classification is illustrated in Figure 2.8.

The next sections will present the concepts behind the different types of model-free RL methods, with an emphasis on value-based, as they are the ones more commonly used in coverage path planning literature.

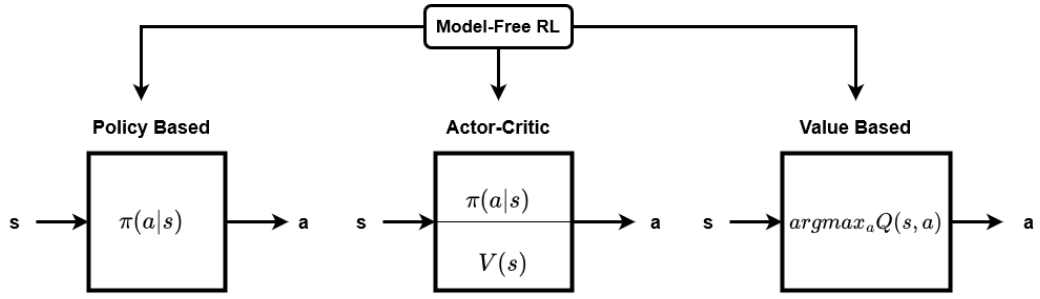


Figure 2.8: The three main classes of model-free Reinforcement-Learning methods.

2.2.4.1 Policy-Based and Actor-Critic Methods

Policy-based is a class of methods that learn a policy π_θ by directly searching over the policy space [43]. This policy π_θ is a parametrization of the policy as $\pi \approx \pi_\theta(a|s)$, where $\pi_\theta(a|s)$ is a function that is differentiable with respect to the parameters. The θ parameter is updated based on the gradient ascent of some scalar performance measure $J(\theta)$ with respect to the parameter θ. More intuitively, the update is done in the direction that maximizes the performance:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{2.20}$$

To delve a bit deeper into the mechanisms of policy-based methods, the classic REINFORCE algorithm [44] will be used as an explanation, as most of the concepts are similar to other policy gradient methods. In this algorithm, the scalar performance measure is the value function v_{π_θ} from the initial episode:

$$\begin{aligned} J(\theta) &= v_{\pi_\theta}(s_0), \\ &= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} R_t\right] \end{aligned} \tag{2.21}$$

Then, by using the policy gradient theorem, the gradient of the score function can be derived, arriving at (see [40] for more details):

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_\theta(a_t|s_t)} q_\pi(s_t,a_t)\right] \\ &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) G_t\right] \end{aligned} \tag{2.22}$$

where G_t is the return of the episode. Then, the REINFORCE algorithm simply samples the Monte Carlo return at the end of every episode, with equation (2.13) and updates the policy with the expression:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) G_t \tag{2.23}$$

This makes REINFORCE a purely policy-based method, as no explicit value-function estimation exists. However, this fact makes the algorithm converge slower and suffer from a higher variance in the return G_t, which are also common downsides of Monte-Carlo methods. Furthermore, not having a value function estimation and relying only on episodic returns makes the credit assignment problem more challenging, as there is no clear way for the agent to learn the good and bad actions in the episode.

In the most recent literature, new policy-based methods are scarce as actor-critic methods have risen in popularity. Since the value function is usually the score function, and there are a lot of methods that are good at estimating this function, then having a "critic" that supports the "actor" with estimates of the value function and its gradient (Figure 2.9) with respect to the parameters is the natural evolution of policy-gradient methods. This fact makes actor-critic appear in the same class as policy-based in some recent surveys [2], and being some of the newest and most promising Reinforcement Learning algorithms, such as Proximal Policy Approximation (PPO) [45], Deep Deterministic Policy Gradient (DDPG) [43], Asynchronous Advantage Actor-Critic (A3C) [46], Actor-Critic with Experience Replay (ACER) [47].

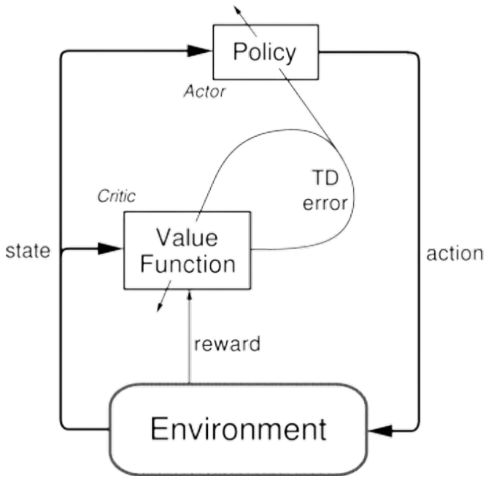


Figure 2.9: Actor-Critic Typical Architecture.

2.2.4.2 Value-Based Methods

As the name suggests, value-based methods do not search directly for the optimal policy. Instead, they estimate the value functions. After the estimation, there is an implicit optimal policy in reference to the state-action value function, which corresponds to acting greedily, or in other words, choosing the action that has the highest expected cumulative returns:

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \text{argmax}_a \, Q_*(s,a) \\ 0 & \textit{Otherwise} \end{cases} \tag{2.24}$$

For MDPs with a finite number of states and actions, it is proven that at least one deterministic optimal policy exists [40].

The classical Q-Learning [42] algorithm is an example of a value-based method, that estimates the state-action value function $Q(s,a)$ by Off-Policy Temporal Difference Learning. The following sections will look in more detail into the inner workings of classic Temporal Differences Algorithms, as they are one of the simplest and most effective RL methods, and most of the ideas are used in other Model-Free algorithms.

2.2.5 Temporal Differences Learning

Temporal Differences (TD) methods have emerged as a game-changing concept in reinforcement learning, deriving their fundamental ideas from Dynamic Programming and Monte Carlo (MC) approaches, both classical solutions for solving Markov Decision Processes.

One of the most important concepts of TD Learning is Bootstrapping, which allows for the update of value function estimations without awaiting the final outcome [40]. This attribute enables TD methods to converge faster, a particularly notable advantage in scenarios involving long-time horizons. To illustrate, consider an agent that is learning to drive. While training, the agent is in a collision course and it does not need to wait for the actual crash to understand that its current course is not correct.

However, the process of bootstrapping, which estimates the value of a state based on the value estimate of the subsequent state, can cause increased instability in the learning process and, in some cases, even divergence. Moreover, when compared to Monte Carlo estimation approaches, a trade-off between bias and variance is apparent. TD methods tend to have a higher bias, while MC methods tend to have a higher variance. Nevertheless, TD methods' ability to work without having to sample a complete episode makes them a more practical solution for most situations.

2.2.5.1 Generalized Policy Iteration

The majority of model-free Reinforcement Learning (RL) methods employ a universal framework inspired by Dynamic Programming to compute the value functions. This framework, known as Generalized Policy Iteration (GPI), encompasses two stages: Policy Evaluation and Policy Improvement.

The initial phase, Policy Evaluation, poses a prediction challenge, where the algorithm focuses on estimating the state-action value function. Conversely, the subsequent phase, Policy Improvement, represents a control problem concerned with identifying the optimal action.

By iterating this process, the algorithm is guided towards convergence to the optimal value function and the corresponding greedy policy, as illustrated in Figure 2.10.

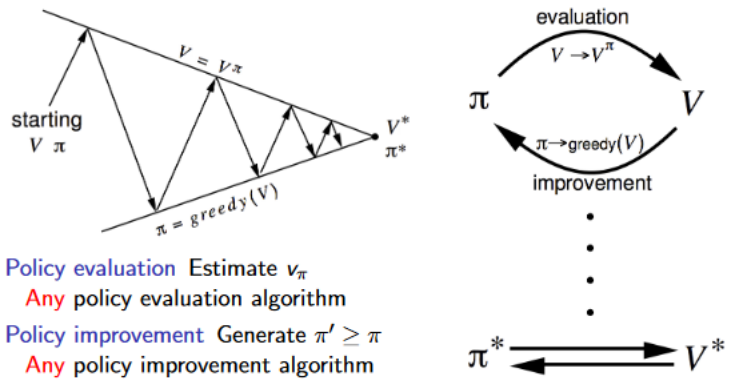


Figure 2.10: Representation of Generalized Policy Iteration; Up Arrows correspond to Policy Evaluation, whereas Downwards arrows are Policy Improvement [3].

2.2.5.2 TD Prediction - TD(0)

The most novel idea of TD Learning is the way it predicts the value function. In the most simple form, the prediction algorithm is called TD(0) and corresponds to doing a one-step look ahead and updating the value function.

In this way, at time step t in the state s, after taking action a, receiving the reward R_t and transitioning to state s', the estimation of the state-action value function $\hat{Q}$ can be updated with :

$$\begin{aligned} \hat{Q}(s,a) &\leftarrow \hat{Q}(s,a) + \alpha[R_t + \gamma\hat{Q}(s',a') - \hat{Q}(s,a)] \\ \hat{Q}(s,a) &\leftarrow \hat{Q}(s,a) + \alpha\delta_t \end{aligned} \tag{2.25}$$

where α is a hyperparameter called the learning rate, that should be tuned to help with convergence and δ_t is the TD error, which is equal to $R_t + \gamma\hat{Q}(s',a') - \hat{Q}(s,a)$.

A general description of the algorithm is seen in algorithm 2:

Algorithm 2 TD(0) Policy Evaluation

Require: policy π, learning rate α

Initialize $Q(s,a)$, for all $s \in \mathbb{S}$ and $a \in \mathbb{A}$, arbitrarily except $Q(\text{terminal}) = 0$
for each episode **do**
    Initialize s
    for each step of episode **do**
        Select action a based on policy π
        Take action a, and observe R, s'
        Choose action a', with policy π
        $Q(s,a) \leftarrow Q(s,a) + \alpha[R + Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$
    end for
    Continue until s is terminal
end for

2.2.5.3 Eligibility Traces - TD(λ)

Despite the differences that were mentioned between MC and TD methods, there is a unifying and generalizing view between both methods that is possible by using Eligibility Traces. This mechanism enables the augmentation of TD methods that produces a family of methods that range from being equal to TD(0) when the hyperparameter $\lambda = 0$ and equivalent to Monte Carlo methods when $\lambda = 1$. This often makes it possible to create methods between extremes that can be better suited for a specific problem. The name of this family of algorithms is TD(λ).

Eligibility traces provide a mechanism for temporal credit assignment, where the trace is a temporary record for an event during the training process, making it possible to assign credit or blame to a specific event in the training process instead of being limited to just one-step lookaheads like in TD(0). There are multiple types of eligibility traces such as accumulating, dutch, and replacing traces.

The backward view TD(λ) is the one of most interest, as it enables an equivalent implementation of MC without having to wait for the termination of the episode, and in general, is the most suited for implementation. This implementation can be seen in algorithm 3 [40].

Algorithm 3 TD(λ) Policy Evaluation

Initialize $V(s)$, for all $s \in S$, arbitrarily except $V(\text{terminal}) = 0$
for each episode **do**
    Initialize s
    for each step of episode **do**
        Select action a based on policy π
        Take action a, and observe R, s'
        $\delta \leftarrow R + V(s') - V(s)$
        $E(s) \leftarrow E(s) + 1$ (Accumulating traces)
        or $E(s) \leftarrow (1-\alpha)E(s) + 1$ (Dutch Traces)
        or $E(s) \leftarrow E(s) + 1$ (replacing traces)
        for all $\hat{s} \in \mathbb{S}$ **do**
            $V(\hat{s}) \leftarrow V(\hat{s}) + \alpha\delta E(\hat{s})$
            $E(\hat{s}) \leftarrow \gamma\lambda E(\hat{s})$
        end for
        $s \leftarrow s'$
    end for
    Continue until s is terminal
end for

2.2.5.4 On-Policy vs Off-Policy

In the TD(0) update equation 2.25, the next action a' was left open as a generic action. However, there are two distinct ways to go about choosing the action, which are On-Policy and Off-Policy prediction.

In On-Policy prediction, the agent uses the policy π to decide the next action and to update the state-action value function using the TD(0) equation, as shown in algorithm 2. A classic example of an On-Policy Temporal-Difference Learning algorithm is SARSA.

In Off-Policy prediction, the agent uses two different policies π and μ. The agent follows policy π to choose the next action but samples from policy μ to get the value for action a' in the update equation. Q-Learning [42], one of the most popular RL algorithms uses this approach.

There is no definitive method to know the optimal approach for a particular problem. On-Policy prediction, typically, offers the advantage of simultaneously improving the policy employed for exploration and exploitation and can converge faster in scenarios with larger time horizons. However, this approach tends to yield more conservative policies [40], since its formulation leads to more risk-aware agents.

Off-Policy algorithms, despite these considerations, remain the most popular choice. They allow the agent to utilize both a greedy policy for exploitation and an exploratory policy for understanding the environment. Conceptually, these algorithms can be visualized as enabling the agent to "look over its own shoulder" and learn from a different policy than the one it uses for control. Consequently, Off-Policy methods are widely regarded as having more potential for future advancements in reinforcement learning [40]. This is confirmed by the fact that the most recent developments in value-based deep RL methods such as DQN[48] and the further improvements that culminated in the Rainbow DQN [49] all use off-policy learning.

2.2.5.5 Exploitation vs Exploration

One of the main questions in TD Learning is the trade-off between Exploitation and Exploration. As seen previously, TD Learning methods use the information obtained from the interaction with the environment to update the value function. However, a question remains: How should we get the information?

It might be intuitive to always choose the greedy action for controlling the agent, however, this policy will run into trouble and eventually not converge. This is because the agent has zero information about the environment, therefore, making the greedy choice is the same as taking a random action. After sampling this random action, it will be biased into taking the same action, when it is possible (and likely) that there are better actions.

The problem described above is the lack of exploration. To solve this problem, a possible solution is using the ε-greedy policy instead of the greedy policy, which can be defined by either of the two equations:

$$\pi(a|s) = \begin{cases} \frac{\varepsilon}{m} + 1 - \varepsilon & \text{if a*=argmax}_a\, Q(s,a) \\ \frac{\varepsilon}{m} & \text{Otherwise} \end{cases} \tag{2.26}$$

$$a_t = \begin{cases} \text{argmax}_a\, Q(s,a) & \text{w.p } 1-\varepsilon \\ \text{Random action in } \mathbb{A} & \text{w.p } \varepsilon \end{cases}, \tag{2.27}$$

where ε is a hyperparameter that determines the probability of the action being random, and m is the number of actions in the action space. This way, the agent can both explore and exploit continuously. The ε-greedy policy can be improved using a decaying ε that converges to a small value over time.

Now, it is possible to thoroughly analyze the difference between Q-Learning and SARSA, being that SARSA uses the ε-greedy policy both for predicting and controlling, whereas Q-Learning differs by using the greedy policy for the evaluation phase.

2.2.6 Function Representation

Until this point, the representation of the value functions has not been addressed. This subsection will look into the two main ways to represent the value functions, the first and the most used in classical reinforcement learning literature is Tabular representations, and the latter is the usage of function approximation techniques to parameterize the value functions.

2.2.6.1 Tabular Methods

The classical algorithms store the values of the functions in look-up tables, usually called Tabular Methods. In the most common case where the state-action value function $Q(s,a)$ is stored, the look-up table will have to have an entry for all possible combinations of actions $a \in \mathbb{A}$ and states $s \in \mathbb{S}$, as illustrated in Figure 2.11.

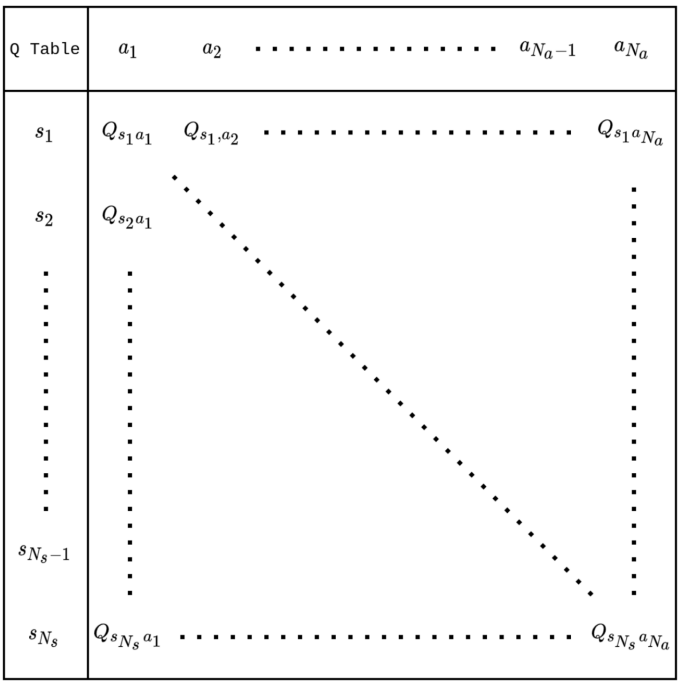


Figure 2.11: A visualization of a Q-Table where N_s is the dimension of the state space and N_a is the dimension of the action space [4].

The Curse of Dimensionality naturally limits the potential of this type of method [40], as memory usage will grow exponentially with the number of states and actions. Furthermore, any non-visited state-action pair during training will not have an estimated value, and even if visited, the convergence guarantees are lost if the state-action pairs are not visited multiple times. These limitations make these approaches unsuitable for problems with high-dimensional state spaces or continuous control problems, as discretization will always be necessary.

2.2.6.2 Function Approximation

A first approach to this problem was using Linear Functions for approximating the value function [50], obtaining a parameterized function $Q(\theta, s, a)$, where the value function is given by a linear combination with θ as controllable parameters. This approach can work well, especially with policy-gradient methods, as good convergence and stability properties exist.

However, this approach is somewhat outdated, as Deep Neural Networks (DNN) started to be seen as a feasible tool for non-linear function approximation [10], making it possible to approximate the value functions with a much more powerful tool when compared to linear approximation. However, the convergence guarantees are lost compared to the latter, and the training is considerably unstable, necessitating various tricks to make it work.

Despite that, the introduction of DNNs made the field explode in popularity when DeepMind unveiled the Deep Q Network [51] and showed that reinforcement learning agents could display super-human potential in very complex and highly-dimensional tasks such as Atari Games. Later, even more impressive developments were achieved, such as the strongest chess, shogi, and Go engines with the AlphaZero/Go series [13] and being as strong as the best players in modern online games like Dota2 [15], Starcraft II [14] and Gran Turismo [52].

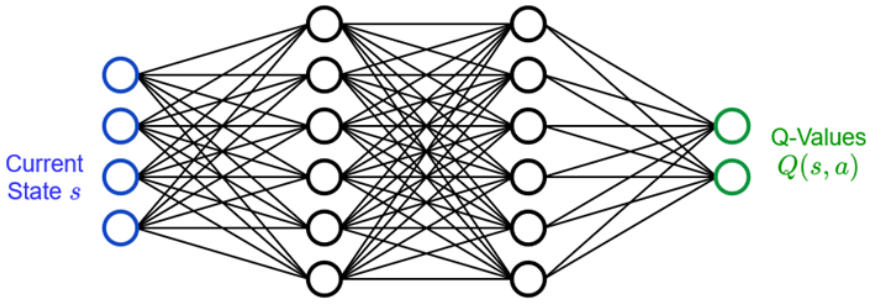


Figure 2.12: A visualization of a Deep Neural Network used for approximating the state-action value function $Q(s,a)$.

2.3 Value-Based Reinforcement Learning Methods

This section provides a comprehensive discussion of the reinforcement learning methods utilized in this dissertation. The discussion will be kept at a high level, focusing on the conceptual un-

derstanding of the methods, while specific details regarding the methods' implementation and their adaptations to the particular problem at hand will be reserved for subsequent sections of the document.

2.3.1 Tabular Temporal Differences Learning - Q-Learning and SARSA

Q-Learning and SARSA are classical Reinforcement Learning algorithms that use a tabular function representation and TD(0) as the policy evaluation mechanism. They differ in the On-Policy vs. Off-Policy dichotomy, where SARSA is an On-Policy method, usually using the ε-greedy policy both for control and evaluation. In contrast, Q-Learning is Off-Policy, meaning it will use a greedy policy for evaluation while keeping the control policy untouched. As mentioned previously, this will mean that, in general, SARSA can be more risk-aware since a bad action can be selected as the following action in the evaluation scheme. A pseudo-code scheme for both algorithms can be seen in the following algorithm 4, where the text written in blue is exclusive to SARSA and the one in red refers to Q-Learning.

Algorithm 4 SARSA and Q-Learning Pseudocode

Require: Initialize values for Q, typically $Q(s,a) = 0$, for all $s \in \mathbb{S}$, $a \in \mathbb{A}$

for each episode **do**
  Initialize s
  Choose a using the ε-greedy policy.
  for each step of episode **do**
    Take action a, observe R and s'
    Choose a' from s' using the ε-greedy policy.
    $Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma Q(s',a') - Q(s,a)]$ ▷ SARSA Update
    $Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \max_{a^*} Q(s',a^*) - Q(s,a)]$ ▷ Q-Learning Update
    $s \leftarrow s'$, $a \leftarrow a'$
  end for
end for

2.3.2 The Overestimation Problem - Double Q-Learning

Tabular Reinforcement Learning methods such as Q-Learning are proven to converge to the optimal action values as long as all actions are continuously sampled in all states, and the state-action values are represented in a discrete manner [42].

However, it is known property that Q-Learning tends to overestimate the state-action values, and it is proven in [53] that the single estimator of Q-Learning is biased, introducing this overestimation. While the values could be overestimated and the policy still be optimal, usually, this estimation error is not uniform and leads to poor policies.

Van Hasselt introduced the concept of Double-Q Learning in [53]. The main difference is the introduction of two different Q-Tables, A and B. With two different estimates for state-action values $Q^A(s,a)$ and $Q^B(s,a)$, the novel idea of a double estimator is introduced and proven to be unbiased, removing the overestimation problem. The double estimator is implemented by

selecting an action using either Q^A or Q^B. When updating the Q-Table using equation (2.25), the Q-Table not used for action selection will be used to estimate the next state-action value $Q(s', a')$.

Algorithm 5 Tabular Double Q-Learning

1: Initialize Q^A, Q^B
2: **for** each episode **do**
3: Initialize s
4: **for** each step of episode **do**
5: Choose a, based on Q^A and Q^B, observe r, s'
6: Choose either UPDATE(A) or UPDATE(B)
7: **if** UPDATE(A) **then**
8: $a^* = \arg\max_a Q^A(s', a)$
9: $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha \left[r + \gamma Q^B(s', a^*) - Q^A(s,a) \right]$
10: **else if** UPDATE(B) **then**
11: $b^* = \arg\max_a Q^B(s', a)$
12: $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha \left[r + \gamma Q^A(s', b^*) - Q^B(s,a) \right]$
13: **end if**
14: $s \leftarrow s'$
15: **end for**
16: **end for**

Algorithm 5 outlines the basic pseudocode for tabular Double Q-Learning. An important point to note, specifically for line 5 of the algorithm, is that the usual implementation involves averaging both Q-tables, treating it as a singular table. This strategy is also typically applied when utilizing the Q-tables post-training.

2.3.3 TD(λ) methods - Q(λ) and SARSA(λ)

The previously discussed algorithms utilize the TD(0) scheme for policy evaluation. However, a broader class of algorithms, known as TD(λ), extends this framework by not being limited to a one-step lookahead when estimating value functions. This ability is tuned with a hyperparameter λ bounded in $[0,1]$. At the extremes of this range, TD(λ) can function equivalently as TD(0) or Monte Carlo methods. Pseudocode for the SARSA(λ) and Q(λ) algorithms, which embody this principle, can be found in Algorithm 6 [42]. Once again, lines colored in blue are exclusive of SARSA(λ) and lines in red are only for Q(λ).

2.3.4 Deep Q Networks

Deep Reinforcement Learning began to garner widespread attention with the introduction of Deep Q Networks (DQN) [51]. Developed by DeepMind, these networks marked a substantial advance in reinforcement learning techniques by incorporating deep learning to approximate the state-action value function. This breakthrough approach alleviated the scalability issues of tabular methods and surpassed the performance of linear function approximation, solving problems that were not feasible with the available techniques.

Algorithm 6 Q(λ) and SARSA(λ)

Initialize $Q(s,a)$ arbitrarily for all $s \in S$, $a \in A(s)$
repeat ▷ for each episode
  Set $E(s,a) = 0$ for all $s \in S$, $a \in A(s)$
  Initialize state S and action A
  repeat ▷ for each step of episode
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    Compute $\delta = R + \gamma Q(S',A') - Q(S,A)$ ▷ SARSA(λ)
    $A^* = \arg\max_a Q(S',a)$ ▷ If A' ties for the max, then $A^* = A'$
    Compute $\delta = R + \gamma Q(S',A^*) - Q(S,A)$ ▷ Q(λ)
    Increment trace: $E(S,A) = E(S,A) + 1$
    for each $s \in S$, $a \in A(s)$ **do**
      Update $Q(s,a) = Q(s,a) + \alpha\delta E(s,a)$
      Update $E(s,a) = \gamma\lambda E(s,a)$
      if $A' = A^*$ **then**
        $E(s,a) = \gamma\lambda E(s,a)$
      else
        $E(s,a) = 0$
      end if
    end for
    Update $S = S'$, $A = A'$
  until S is terminal
until termination

The results of the DQNs were remarkable. The network exhibited super-human performance across 49 Atari 2600 games [54, 48], showcasing the potential and efficacy of deep reinforcement learning techniques. Notably, the DQN model maintained the same architecture and hyperparameters across all games, demonstrating its impressive versatility and adaptability. This achievement set the stage for a new era in the development and application of reinforcement learning methods.

Before going more in-depth on the in-workings of this algorithm, firstly, it is essential to know why it was such a significant breakthrough and what were and are the challenges of using neural networks for reinforcement learning problems.

2.3.4.1 The Deadly Triad

The significance of DQNs and the challenges of integrating neural networks with reinforcement learning cannot be overstated. This intersection presents a unique set of issues, famously known as the "Deadly Triad."

The Deadly Triad, as described by Sutton and Barto [40], consists of three aspects of reinforcement learning methods that individually can be beneficial but, when combined, can lead to instability and divergence of the learning process. These aspects are:

- **Function Approximation:** The representation of the value function, or the policy, as a parameterized function. This is necessary to handle large state spaces or continuous state-action spaces.

- **Bootstrapping:** The use of existing estimates to update estimates, as is done in TD methods. This can significantly speed up learning compared to Monte Carlo methods, which wait until the return is known.

- **Off-policy Learning:** Learning using a distribution of experiences other than the one produced by the target policy (the ones experienced by the agent).

It is crucial to consider that these three aspects are not only part of the DQN, but they are desirable for it to achieve good results. The function approximation can not be given up at all, as it is the most significant novelty and strength of the DQN. As for bootstrapping, while it introduces some instability, it also allows for more sample-efficient learning than Monte Carlo methods, which only update after a complete episode. Bootstrapping is a more practical choice in complex environments with long time horizons, such as Atari games. Finally, although the off-policy learning strategy could technically be given up, it is not in the best interest as it would lose the possibility of learning from a replay buffer or even imitating another agent.

Therefore, while these factors pose challenges, they constitute the essence of the DQN approach. Rather than trying to eliminate these elements, the key lies in mitigating their negative effects.

2.3.4.2 Mitigating The Issues

Although the Deadly Triad was addressed, the problems that it caused were not expanded beyond the fact that the combination leads to instability in training. Before going more in-depth on the issues caused by this combination of factors, first is important to present how the learning is done. Without going into much detail, the DQN generally uses a loss function equal to the squared TD error and updates its parameters using a gradient-descent scheme in this loss function.

$$L(\theta) = (R_{t+1} + \gamma \max_{a} Q_\theta(S_{t+1}, a') - Q_\theta(S_t, A_t))^2 \tag{2.28}$$

From looking at this loss function, several problems can arise.

- **Correlation in Transitions.** The sequence of transitions (also called experience) $(s_t, s_{t+1}, a_t, R_{T+1})$ acquired by an agent is often highly correlated. This is because the state at time $t+1$ is often very similar to the state at time t, and the actions taken are based on the agent's current policy, which usually changes slowly over time. Suppose the updates are done in an online fashion. In that case, the strong correlation between subsequent states and actions can severely affect learning stability, as this issue breaks the "independent and identically distributed" (i.i.d) assumption of many popular stochastic gradient-based algorithms.

- **Correlation Between Target Values and Network Parameters.** The process of updating the parameters of the approximated Q-function involves adjusting them based on the differences between current estimates and target values. In traditional Q-learning, the target

values are also produced by the same Q-function that is being updated. This creates a situation where the target values and the estimates are interdependent. As a result, an update to the estimate can immediately influence future target values, leading to a bootstrapping problem. In this case, the "moving target" is a consequence of the correlation between the current estimate and the target value. It leads to a form of circular reasoning where the algorithm can end up chasing its tail, resulting in unstable or divergent behavior.

- **Non-Stationary Targets.** Reinforcement Learning algorithms often operate in a dynamic environment where the Q-values (or targets for learning) are non-stationary. This refers to the phenomenon where the Q-values, serving as targets for future learning updates, are continuously revised and updated due to the environment's changing conditions or the agent's evolving policy. This aspect of non-stationarity poses significant challenges to the learning process. When the agent explores different sections of the environment, or the state of the environment itself alters, it influences the Q-values.

These issues are addressed with two mechanisms that are part of the DQN algorithm: The target network and an experience replay buffer [55].

The target network is a mechanism introduced to address the issue of correlation between targets and parameters, as well as non-stationary targets. In the DQN algorithm, two networks are used. One, the value network, is updated continually to learn and adapt to the agent's experiences. The other, the target network, is 'frozen' and updated less frequently, typically every few thousand timesteps, by copying the parameters from the value network. This approach creates a set of more stable target values, reducing oscillations and fluctuations during the learning process, thus promoting stability and improving the algorithm's effectiveness.

On the other hand, the experience replay buffer is utilized to break the correlation between consecutive transitions. In this process, all agent's transitions $(s_t, s_{t+1}, a_t, R_{t+1})$ are stored in a buffer. These transitions are then randomly sampled during the learning process. This means that the strong temporal correlations between them are broken. This approach helps prevent the value estimates from swinging back and forth in response to consecutive, highly correlated updates while also making it possible that some rare transitions are not instantly forgotten after the first update, once again improving the algorithm's stability.

2.3.4.3 The Vanilla DQN

The Vanilla DQN, or Nature DQN from the DeepMind article in the Nature journal [48], serves as a baseline for almost every algorithm for Atari Learning Environment (ALE). A more thorough investigation to the DQN will be provided in this subsection.

Consider the state-action value function estimated by the value network Q_θ and the one estimated by the target network as Q_{θ^-}. Furthermore, consider that at every timestep t the agent stores an experience $e_i = (s_t, a_t, R_{t+1}, s_{t+1})$ on the replay buffer $D = \{e_1, ...e_t\}$. During learning, the algorithm randomly samples a minibatch m of size M from the replay buffer, using it to calculate the loss function:

$$L(\theta) = \mathbb{E}_{m\sim D}\left[\left(R_t + \gamma \max_a Q_{\theta^-}(s_{t+1}, a) - Q_\theta(s_t, a_t)\right)^2\right]. \tag{2.29}$$

Here, the term $R_t + \gamma \max_a Q_{\theta^-}(s_{t+1}, a)$ serves as an estimate of the expected return from the state-action pair (s_t, a_t), taking into account both the immediate reward R_t and the discounted future return from state s_{t+1} as estimated by the target network. The optimization process aims to minimize the discrepancy between this estimate and the value given by the current estimate given by the value network $Q_\theta(s_t, a_t)$.

This optimization is done through a stochastic gradient descent scheme, so the expectation operation is removed. The value network parameters are updated based on the optimization minimization of the loss function. The target network is updated every C steps (10000 in the Nature DQN) by copying the parameters of the value network. The whole pseudocode algorithm can be seen in Algorithm 7.

Algorithm 7 Deep Q-Learning

Initialize replay memory D to capacity N
Initialize action-value function Q_θ with random weights θ
Initialize target action-value function Q_{θ^-} with weights $\theta^- = \theta$
for episode = $1, M$ **do**
    Initialize the State s_t
    for $t = 1, T$ **do**
        Select action a_t using ε-greedy policy on Q_θ
        Take a step with action a_t and observe next state s_{t+1} and next reward R_{t+1}
        Store transition $(s_t, a_t, R_{t+1}, s_{t+1})$ in D
        Sample random minibatch m of transitions (s_j, a_j, R_j, s_{j+1}) from D
        Perform a gradient descent step on $[R_j + \gamma \max_{a'} Q_{\theta^-}(s_{j+1}, a') - Q_\theta(s_j, a_j)]^2$ w.r.t θ
        Every C steps reset $Q_{\theta^-} = Q_\theta$
    end for
end for

The DQN architecture, illustrated in Figure 2.13, is built on the foundation of a convolutional neural network (CNN), which makes it well-suited to handling image-based inputs. In the context of Atari games, the agent's state or observation at a given moment is represented by a stack of the four most recent frames from the game.

Upon receiving this stacked input, the DQN's convolutional extracts and learn useful features from the raw image data. Following this process, the CNN's output is flattened and fed to a dense fully connected network with seven neurons with no activation function on the output layer. Each of these neurons outputs one of the values $Q(s,a), \forall a \in \mathbb{A}$, being that the greedy policy would choose the neuron with the maximum output value.

It is important to note that this architecture can be changed and adapted to a myriad of other problems, even outside of ALE. However, the DQN is still not an adequate solution for continuous control problems or big action spaces, as the number of neurons on the output layer discretizes the Q function.

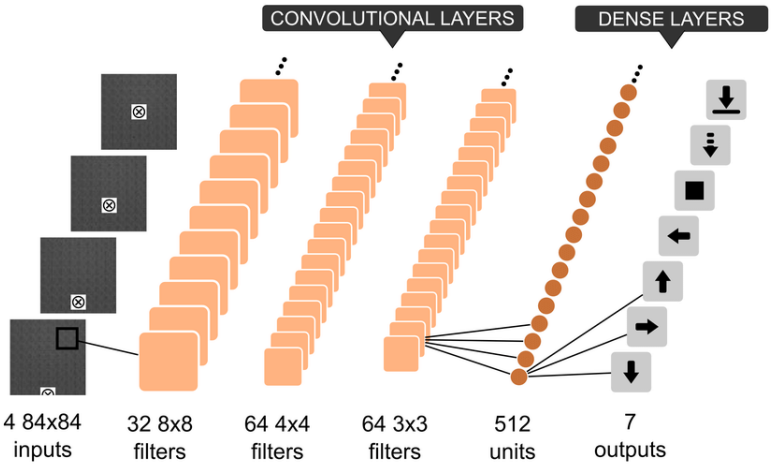


Figure 2.13: The Nature DQN architecture[6]

2.3.5 Extensions to the DQN - The Rainbow DQN

After the proposed vanilla DQN algorithm, multiple extensions, improvements, and new concepts were proposed on top of this framework. These improvements culminated in the Rainbow DQN, proposed by Hessel *et al.* [49], a version of the DQN algorithm with six extensions that were added over the years, proving to be a significant improvement over the other approaches in the Arcade Learning Environment. The following subsections will present a concise overview of these key extensions.

2.3.5.1 Double Deep Q-Learning

While the principle of "optimism in the face of uncertainty" is one of the main tenants of Reinforcement Learning, as analyzed in Section 2.3.2, the tabular Q-Learning is known to overestimate the state-action values. This issue persists in the Deep Learning version of Q-Learning, where it becomes even more critical due to the resultant instabilities during training.

A solution to mitigate this problem was the Double Deep Q Network (DDQN), proposed by Hasselt *et al.* [56]. The DDQN implements the double estimator by decoupling the selection from the evaluation by using the online network to select the action and the target network to evaluate that action.

$$L(\theta) = \left[R_{t+1} + \gamma Q_{\theta^-}(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q_\theta(s_{t+1,a'})) - Q_\theta(s_t, a_t)\right]^2 \tag{2.30}$$

2.3.5.2 Prioritized Experience Replay

In the standard DQN framework, all samples in the experience replay buffer are treated equally during the sampling process. The Prioritized Experience Replay (PER) [57] introduces the concept

[6]Taken from https://paperswithcode.com/method/dqn.

that not all transitions should be treated the same. It prioritizes the most "important" transitions, with importance, in this case, being defined by the magnitude of the Temporal Difference (TD) error δ.

Given $p_i = |\delta_i| + \varepsilon$, where ε is a small positive constant to ensure all experiences have a non-zero probability of being sampled, the probability of transition i being sampled is defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_i^\alpha}, \tag{2.31}$$

where α is a hyperparameter that determines the degree of prioritization. This sampling method, however, introduces bias as it modifies the distribution of the expected value. To correct the bias, an importance-sampling weight is used:

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta \tag{2.32}$$

This equation presents the importance-sampling weights w_i, a function of the probability of a transition being sampled $P(i)$, and the total number of stored transitions N. The parameter β controls the degree to which these weights influence learning. Initially, β is set to a low value to facilitate learning from a broader range of experiences. It is then gradually annealed to 1 to compensate for the bias introduced by prioritized sampling. These weights are normalized by the term $1/max_i w_i$ for stability purposes and then multiplied by the TD error on the gradient descent step.

2.3.5.3 Dueling Networks

Another improvement of the DQN is the dueling network architecture[58]. In this architecture, instead of having a single stream with an output layer equal to the size of the action space that estimates the state-action value function, a second stream is introduced. This results in the value stream that estimates the state value function $V(s)$ and the advantage stream, estimating respectively the advantage function $A(s,a)$. An intuition for this function is that it gives the advantage, as in value, of choosing an action a when compared to the expected value of choosing the other possible actions. Then, the Q-Function is possible to be estimated via:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|A|}\sum_{a'} A(s,a') \tag{2.33}$$

This scheme for estimating the state-action values proves beneficial when actions hold similar values or are considered redundant. A visualization of the architecture can be seen in Figure 2.14.

2.3.5.4 n-Step Learning

Traditional TD-Learning methods use a one-step lookahead to estimate the value functions. One alternative to this method is to use forward view multi-step targets, utilizing the n-step return

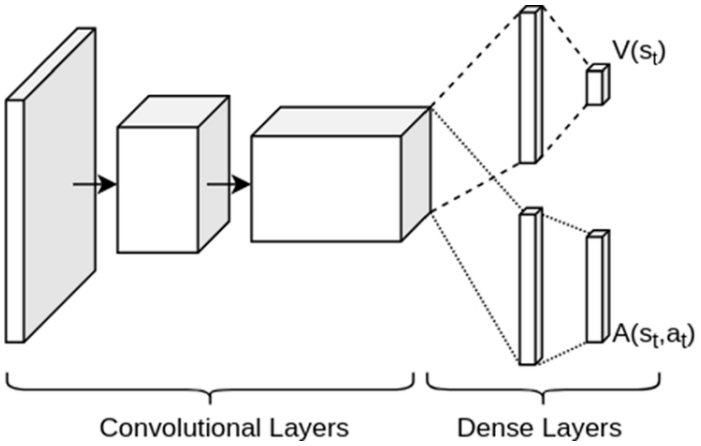


Figure 2.14: The Proposed Dueling Network Architecture. On top is the Value Stream, with a single output neuron, and on the bottom is the Advantage Stream, with a number of neurons in the output layer equal to the action-space size. Image adapted from [5]

.

instead of looking only one step into the future.

$$L(\theta) = \sum_{t}^{t+n} R_t + \gamma^n \max_a Q_{\theta^-}(s'_{t+n}, a') - Q_\theta(s_t, a_t) \tag{2.34}$$

This learning scheme can also help deal with the bootstrapping on the deadly triad dilemma. Furthermore, it reduces the bias in learning, with the downside that the variance will be higher.

2.3.5.5 Noisy Nets

The exploitation-exploration problem is one of the core problems of every TD Learning method. While ε-greedy has proven to be a reliable solution in most situations, this policy can fail in specific environments. Noisy Nets [59] propose a novel approach to exploration by introducing parameter space noise. In Noisy Nets, noise is injected into the linear layers of a neural network, and this noise is updated through gradient descent. This creates a state-dependent and automatically tuned exploration strategy. The noisy layer outputs are computed as:

$$y = (b + Wx) + (b_{noisy} \odot \varepsilon^b + (W_{noisy} \odot \varepsilon^w)x) \tag{2.35}$$

where b and W are the weights of the original network, $\odot$ denotes the element-wise product, b_{noisy} and W_{noisy} are the learned noise parameters, and ε^b and ε^w are independent random variables sampled from a factorized normal distribution. By making the random variables equal to zero, it is possible to implement a deterministic policy that is only used for exploiting, for example, when deploying the model in the real world.

2.3.5.6 C51-Distributional RL

Bellemare *et al.* [60] introduced Distributional Reinforcement Learning as a novel concept in Deep Reinforcement Learning. Instead of approximating the expected return, this approach learns to approximate the distribution of returns.

The distributions are modeled with probability masses on a support vector z with $N_{atoms} \in \mathbb{N}^+$, defined by $z^i = v_{min} + (i-1)\frac{v_{max}-v_{min}}{N_{atoms}-1}$ for $i \in \{1,...N_{atoms}\}$. Considering the probability mass of each atom i $p^i_\theta(S_t,A_t)$, the approximating distribution can be defined as $d_t = (z, p_\theta(S_t,A_t))$, and the goal becomes to update θ to approximate this distribution to the actual distribution of returns.

The target distribution can be formulated as

$$d'_t = (R_{t+1} + \gamma z, p^-_\theta(S_{t+a}, \bar{a}^*_{t+1})), \tag{2.36}$$

and the network weights are optimized by minimizing the Kullbeck-Leibler divergence between distributions $D_{KL}(\phi_Z d'_t || d_t)$, where ϕ_z is a L2-projection of the target distribution onto the fixed support z and , $\bar{a}^*_{t+1} = \arg\max_a q_\theta(S_{t+1},a)$ is the greedy action, where $q_\theta(S_{t+1},a) = z^T p_{\theta^-}(S_{t+1},a)$.

The parameterized distribution is represented by a neural network, similar to the DQN, but with $N_{\text{atoms}} \times N_{\text{actions}}$ outputs. To ensure that the distribution for each action is appropriately normalized, a softmax function is applied independently for each action dimension of the output. The softmax function takes the values of each dimension and transforms them into a valid probability distribution, where the probabilities of all actions sum up to 1.

2.4 Multi-Agent Reinforcement Learning

Multi-Agent Reinforcement Learning (MARL) expands upon the traditional reinforcement learning (RL) framework, accommodating scenarios involving multiple agents. Conventional RL typically involves a singular agent interacting with an environment to optimize its behavior. However, real-world situations can involve the presence of multiple agents, each with distinct objectives and policies. These scenarios complicate decision-making as the environment's evolution is now dictated by the collective actions of all agents. Consequently, each agent must contemplate the behaviors of others when making decisions and negotiate optimally to achieve their respective goals.

The study of multi-agent systems (MAS), is of particular interest in robotics. Most applications and use cases inevitably involve robots directly or indirectly interacting with other agents in the environment. Consequently, a framework that equips these agents with the awareness of others is critical to fully harnessing a robot's capabilities and potential. This section will present a brief overview of the main concepts of MARL.

2.4.0.1 Markov Games

In Multi-Agent Reinforcement Learning, a decision-making problem can be formulated as a Markov or Sthocastic Game. A Markov Game (MG) can be interpreted as a multi-agent extension to the MDP. Therefore, analogously to the single case, an MG can be defined by the tuple $\langle N, \mathbb{S}, \{\mathbb{A}^i\}_{i\in\{1,\dots,N\}}, P, R_{i\in\{1,\dots,N\}}, \gamma\rangle$ [2]:

- N - The number of agents in the environment.
- $\mathbb{S}$ - Set encompassing all states s of the environment, shared by all agents.
- $\mathbb{A}^i$ - Set of actions of agent i. The joint action space if denoted as $\mathbf{\mathbb{A}} := \mathbb{A}^i \times \dots \times \mathbb{A}^N$.
- P: $\mathbb{S} \times \mathbf{\mathbb{A}} \to \mathbb{S}$ - State Transition Probability Function; Given a state $s \in \mathbb{S}$ and a combination of actions $a^i \in \mathbb{A}^i$, where i varies from 1 to N, the function defines the probability of the next state being $s' \in \mathbb{S}$.
- R^i: $\mathbb{S} \times \mathbf{\mathbb{A}} \times \mathbb{S} \to \mathbb{R}$ - A reward function for each agent i; Given the current state and action tuple $(s, a^i) \in (\mathbb{S}, \mathbb{A}^i)$ and the next state $s' \in \mathbb{S}$, the function yields a scalar value equivalent to the reward of the state-transition for agent i.
- $\gamma \in [0, 1]$ Is the Discount Factor, the same as in the MDP.

Ultimately, the approach is very similar to the single-agent scenario. A visualization of the interaction loop between the agents and the environment can be seen in Figure 2.15.

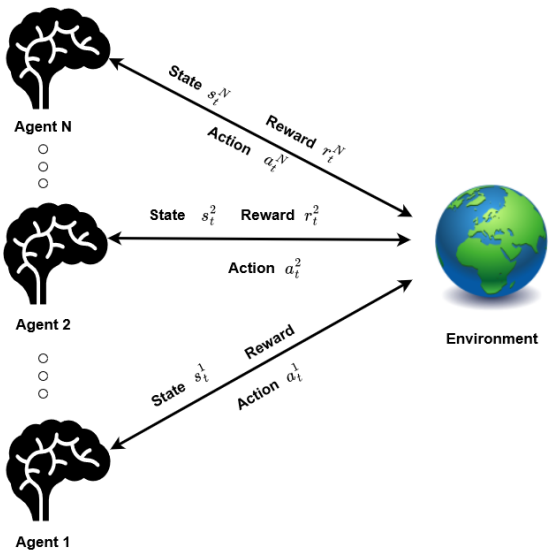


Figure 2.15: Interaction model in a Markov Game, N agents and a generic environment.

2.4.0.2 Partially Observed Markov Games

Similar to the Markov Decision process framework, a Markov Game can also be formulated in a partially observed setting, being a Partially Observed Markov Game (POMG). To the definition in the previous setting, a POMG adds to the existing tuple the following terms:

- Ω^i Set that contains all observations of agent i. The joint observation space if denoted as $\mathbf{\Omega} := \Omega^i \times ... \times \Omega^N$.
- $\mathscr{O} : \mathbb{S} \rightarrow \mathbf{\Omega}$: Is the observation function, which maps the current state s to the observation of the agent o.

A particularly significant instance of a POMG is the Decentralized Partially Observed Markov Decision Process (Dec-POMDP). In this scenario, the Reward function for all agents is the same: $R^1 = R^2 = \ldots = R^N$.

2.4.1 MARL Taxonomy

The categorization of Multi-Agent Reinforcement Learning (MARL) strategies can be quite diverse, primarily due to the multiple aspects that differentiate multi-agent environments. In the following subsections, some of the main aspects of MARL will be covered.

2.4.1.1 Task Nature: Cooperative vs. Competitive

One of the main classifications can be related to the objective of each agent and if there is a common goal. This can create three types of settings: cooperative, competitive, and mixed [61].

- **Cooperative Setting:** In most cooperative settings, the agents usually share a common reward function $R^1 = R^2 = \ldots = R^N$, or at the very least share a similar reward function, with the same goal in mind. In the latter case, the objective of the agents is to maximize the Average Reward $\bar{R}(s,a,s') := \frac{1}{N} \cdot \sum_i^N R^i(s,a,s'), \quad \forall (s,a,s') \in \mathbb{S} \times \mathbb{A} \times \mathbb{S}$. It is noteworthy that, in the average reward model, the agents consistently act in the interest of the collective rather than prioritizing individual gains.
- **Competitive Setting:** This type of MARL problem features self-interested agents who often compete with others for rewards. The problem is typically a zero-sum game, where the sum of all rewards equals zero, implying that one agent's gain necessitates an equal loss for others.
- **Mixed Setting:** This type of setting is the least restrictive and known as general-sum games. In these problems, no specific regulations regarding the goals or relationships among agents apply. This flexibility makes the problem as general as possible, where, within the same game, an agent can act in the best interest of a team, compete with other agents, or a combination of both, in which there is a common team objective, but individual agent goals may conflict with those of other team members.

2.4.1.2 Types of Agent

Multi-Agent Reinforcement Learning systems can also be classified based on the homogeneity of the agents. In the most straightforward approach, homogeneous agents are those that share observation and action spaces, reward functions, and typically also the end goal. On the contrary, heterogeneous agents can have a wide range of differences. They might share the same action space but have individual goals or be identical in all aspects except their objectives. For ease of classification, any group of agents that do not strictly conform to the definition of homogeneous agents is considered heterogeneous.

2.4.1.3 Information Availability

The classification of Multi-Agent Reinforcement Learning (MARL) problems can also be based on the level of information available to the agents. As outlined in the survey by Yang[2], there are seven distinct levels of information, ranging from the most basic - where an agent can only observe its own reward - to the most comprehensive, where the agent has the so-called perfect information and can therefore solve the problem in its entirety. These levels of information are detailed in Table 2.1 [2].

Table 2.1: Levels of Information in Multi-Agent Reinforcement Learning.

Level	Assumption
0	Each agent observes the reward of his selected action.
1	Each agent observes the rewards of all possible actions.
2	Each agent observes others' selected actions.
3	Each agent observes others' reward values.
4	Each agent knows others' exact policies.
5	Each agent knows others' exact reward functions.
6	Each agent has perfect information.

2.4.2 Challenges of Multi-Agent Reinforcement Learning

Multi-Agent Reinforcement Learning is a much less mature field when compared to the single-agent approaches. There are still a lot of challenges, both from a theoretical and technical point of view, having a lot of opportunities for innovation and new ideas. Some of these challenges will be summarized below. The following sections will outline some of these prevalent challenges.

2.4.2.1 The Scalablity Problems

In a multi-agent learning problem, the agent policy is no longer limited to itself, considering all other agents' policies when determining the best action. This naturally leads to the joint action space $|\mathbb{A}|^N$ growing exponentially, and the same applies to the state space, as every agent will inherently influence the state. As the number of agents increases, the complexity of the problem

grows, resulting in more computationally demanding algorithms and potentially infeasible real-time solutions. Furthermore, issues related scalability of communication networks, information sharing, coordination, and negotiation also become significantly more complex as the number of agents increases.

From a theoretical standpoint, multi-agent system analyses often limit to scenarios involving only two agents. Thus, algorithms that guarantee convergence and an equilibrium solution are typically confined to these scenarios [2]. These issues indicate a considerable scope and potential for further in-depth research on large multi-agent systems in MARL.

2.4.2.2 Non-Stationary

One of the most common and most addressed problems from MARL is the non-stationary nature of the environment [62]. This issue arises because an agent's actions can influence another agent's reward, disrupting the stationary assumption typically maintained in reinforcement learning algorithms. Since all agents are learning policies, and part of learning its policy is to learn how the other behaves, from the agent's point-of-view, it is learning behavior that is consistently changing. This creates a chain of reactions where one's policy changes because of the others, and the other's policy changes because of the change in the agent's behavior.

Despite not having a concrete solution to this issue, numerous strategies have been proposed to mitigate its effects [62]. These include promoting communication between agents, deploying decentralized learning techniques, utilizing meta-learning, and modeling agent behaviors. Interestingly, in scenarios where agents share similar interests, one might act selfishly without considering the impact on others, knowing that others will do the same, thereby preserving the environment's stationarity [2].

2.4.2.3 Learning Goals and Credit Assignment

Another issue is that the overall goal of a MARL algorithm is not as clearly defined as in single-agent RL, making it difficult to align it with a single metric. In a heterogeneous team, for example, in robotic soccer [63], while the overall goal can be described as scoring more goals than the opponent team, a goalkeeper robot will have a very different role than a striker. This creates a problem where the problem is significantly harder to model, especially when it comes to learning cooperation between very different agents of the same team.

Furthermore, there are also credit assignment problems. If a robotic soccer team concedes multiple goals, should the goalkeeper bear all the blame, or should in-field players also shoulder some responsibility? Even in homogeneous teams, is it fair to rely excessively on an overperforming agent or to single out an underperforming one for blame? These questions remain unresolved, but several methods have been proposed to mitigate these issues, involving complex models and comprehensive frameworks, including game theory approaches.

2.4.3 Multi-Agent Reinforcement Learning Methods

Similarly to the single-agent paradigm, solution methods can also be divided into model-free and model-based. However, compared to model-free methods, model-based techniques are considerably less developed in the MARL context [2].

In terms of model-free approaches, there are value-based algorithms such as QMIX [64], and policy-gradient methods like the Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [65], which draw parallels to the single-agent RL strategies.

Nevertheless, this document will mainly concentrate on the usage of a method termed "parameter sharing" [66]. This technique allows the application of single-agent RL methods in multi-agent contexts by employing the same model for all agents. Typically, these methods utilize a centralized learning and decentralized deployment framework, where agents are trained collectively using the shared model and complete information but are subsequently deployed individually.

This technique was considered restricted to homogenous cooperative agents due to the inherent nature of using a shared model. However, a more recent work [67] shows that by using "agent indicators," padding the inputs and outputs layers and using adequate masks for both, parameter sharing can effectively be extended to heterogeneous agents as well, achieving state-of-the-art performance.

Chapter 3

State of the Art

The purpose of this chapter is to give an overview of the current state-of-the-art in Coverage Path Planning algorithms. The chapter is divided into six sections. The first Section 3.1 provides a brief introduction to the CPP Task. It is followed by Section 3.2, which explores Exact Cellular Decomposition algorithms. Section 3.3 presents algorithms with No Celullar Decomposition, whereas Section 3.4 reviews Approximate Celular Decomposition methods. Afterward, attention is shifted to Reinforcement Learning approaches to Coverage Path Planning in Section 3.5. Finally, some brief conclusions on the current state-of-the-art and research opportunities are given in Section 3.6.

3.1 Coverage Path Planning

Coverage path planning is a common task in robotics, where the main goal is to compute collision-free paths that pass through all points of an area or volume [6]. Some of the most common applications were already presented in Chapter 1. However, in this section, the discussion will focus on generic CPP tasks in a 2D plane, with different approaches regarding online or offline planning, algorithms, optimality, and performance metrics.

In one of the first works on the topic [68], a CPP task is defined with the following goals and constraints:

(1) The agent must move through all the defined area of interest.

(2) The agent path must not overlap.

(3) Continuous and sequential operation without any repetition of paths is required.

(4) The paths must be collision-free.

(5) The agent should follow simple motion (straight-line, circle).

(6) An "optimal" path is desired under available conditions.

It is clear from analyzing these criteria that they cannot always be met, especially in complex environments with local maximums and non-convex settings. Moreover, there are certain tasks where there is no information about the environment, and it is necessary to construct local information based on sensor data. In these cases, it is rather obvious that criteria such as "optimality", full coverage, and non-overlapping path may just not be feasible. It is therefore necessary to take into consideration priorities when developing CPP algorithms.

In one of the first surveys on the topic, Choset [69] classifies CPP algorithms as either complete or heuristic. This classification is based on whether the algorithm has a guarantee of coverage of the full area of interest, being a complete algorithm in the case of meeting this requirement.

Nonetheless, the algorithms can also be defined as offline or online (also called sensor-based [69]). In the first case, the agent has a priori knowledge of the environment, and the path planning is done before the start of the mission. This situation is, however, sometimes unrealistic on real robotic applications and also not adaptable to very dynamic environments [6]. On the other hand, online algorithms have to extract data about the environment through the sensors and build a model of the surroundings during the mission. In this approach to CPP, being complete or heuristic, there are no guarantees of optimality of the algorithm, as one can always create an antagonistic example [69].

The following subsections will give an overview of coverage path planning underlying concepts.

3.1.1 Performance Metrics

Coverage Path Planning is a task that can be used in varied applications, therefore, there is no performance metric that can truly fit all cases. One can look into the six goals defined in the previous section to evaluate how good an algorithm is. However, even in these concrete goals, some might not be desirable depending on the application. For example, in patrolling, it might be desirable to have paths overlap in points that have a higher priority in coverage.

For the case of continuous coverage, such as patrolling, the survey [7] defines as possible performance metrics the number of detected objects/events, interval, and frequency of visit in a determined cell as well as the quadratic mean of the first and the standard deviation of the latter.

As for the simple coverage case, some of the most common metrics for evaluation are: energy usage [19, 7, 70], number of turns [71], total traveled distance [72], covered area [73], path overlap [21] or repetition rate [74], and time-to-complete [75].

For offline methods, the metrics such as time-to-complete and total traveled distance might seem the same, but often not, the shortest path is not the fastest, needing to take into consideration kinodynamic limitations of the robot. These optimization concerns are more relevant in offline methods with full knowledge of the environment, where the question moves from "can the area be covered" to "how the area can be covered". As such, for online methods, path overlap and repetition rate are the most relevant metrics.

However, the most important metric for CPP in UAVs is energy usage [75], as battery autonomy is the main setback of using these robots in coverage path planning missions. This criterion

is sometimes formulated as minimizing the number of turns, as whenever a robot performs a turning maneuver, it will most likely reduce the velocity, rotate, and then accelerate again, and all of these actions directly lead to energy consumption. Formulating the problem this way also eases the optimization problem by reducing the number of variables involved, since if one considers the total energy, it is necessary to consider the dynamics of the vehicle and motion constraints, and the optimal values for variables such as velocity are dependent on the trajectory, as mentioned in [19].

3.1.2 Area of Interest

Following the notation in [7], one can define an area of interest as a sequence of vertex $\{v_1, ..., v_p\}$, and the respective connections between vertices, edges $\{e_1, ..., e_p\}$. Each vertex v_i can be described by the pair of cartesian coordinates in the global reference frame $(v_x(i), v_y(i))$, and the internal angle can be represented as γ_i. The set of vertices and edges can be represented in a graph. Furthermore, one can also define obstacles inside the area of interest as a series of points $\{u_1, ..., u_p\}$. A visualization of two examples of areas of interest can be seen in Figure 3.1.

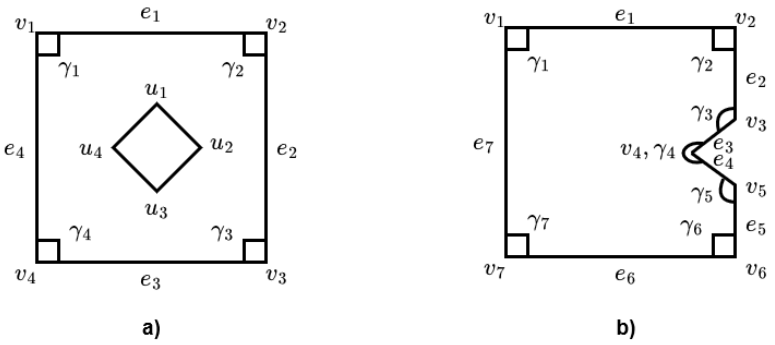


Figure 3.1: Different areas of interest in CPP Tasks: a) Convex Polygon with Obstacles; b) Non-Convex Polygon

One important aspect to consider when implementing a CPP algorithm is the shape of the area of interest. Some algorithms only have optimality and coverage guarantees when the area is rectangular or convex or do not consider obstacles [7].

The majority of CPP algorithms can only be used on simple and convex areas. Therefore, one of the main components of most algorithms is decomposing the area of interest in smaller and simpler subareas.

3.2 Exact Cellular Decomposition

Exact cellular decomposition methods decompose the obstacle-free space of the area of interest down into smaller, non-overlapping subareas called cells [6].

Just as in the case for the area of interest, cells can be represented by an adjacency graph, where a cell corresponds to a node, and boundaries between cells are represented by edges, as can be seen in Figure 3.2.

This type of algorithm can usually be divided into three phases: Decomposition, Planning, and Execution [7]. The decomposition is usually generated by sweeping a line through the area of interest, in a top-down or left-right approach. Then, cell boundaries can be formed when an event occurs. The events that are considered for this decomposition, define the type of algorithm used, which will be analyzed later in the following subsections.

After being done with decomposition, the algorithm will go to the planning phase, where it usually computes a sequence that visits each node in the graph exactly once (if possible). Finally, in the execution phase, the agent will use simple motions to cover each cell. Simple movements that are usually used are back-and-forth [76] or spiral motions [75].

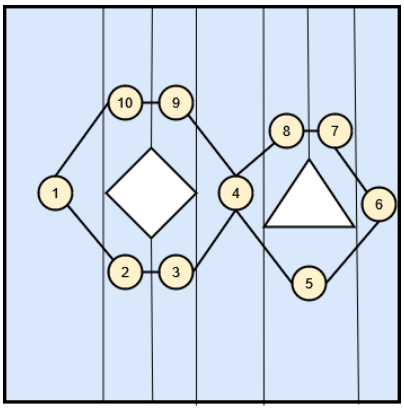


Figure 3.2: Graph representation of an area of interest with trapezoidal decomposition

3.2.1 Trapezoidal Decomposition

Trapezoidal Decomposition is one of the simplest offline coverage path planning algorithms that can be implemented, with the limitation that only polygonal obstacles and areas of interest can be considered [6]. The decomposition phase of the algorithm consists in sweeping the environment until finding the vertices of the obstacles. Whenever a vertex is found, a cell boundary is defined. This decomposition can be seen in Figure 3.2.

3.2.2 Boustrophedon Decomposition

Boustrophedon Decomposition is another offline CPP algorithm based on exact cell decomposition. It builds upon the ideas of trapezoidal decomposition, minimizing the length of the coverage path.

This optimization is achieved by changing the decomposition method. Instead of considering every vertex as a cell boundary, boustrophedon decomposition only considers vertices where a

segment can be extended both above and below the vertex. These vertices are called critical points [76].

In comparison with regular trapezoidal composition, this method produces fewer cells. By inspection of Figure 3.2, one can see that could easily be merged, which is what boustrophedon decomposition does. A comparison between both decomposition techniques can be seen in Figure 3.3.

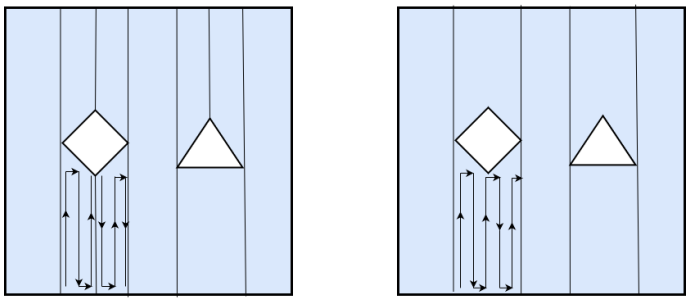


Figure 3.3: Comparison between Trapezoidal and Boustrophedon Decomposition. The trapezoidal decomposition a), has more cells, needing an extra strip to cover when compared to b).

3.2.3 Morse-Based Exact Cellular Decomposition

In an attempt to introduce a general framework for exact cellular decomposition, Acar [77] introduced decomposition based on Morse Functions.

By definition, all critical points of a Morse Function are non-degenerate. A critical point $p \in \mathbb{R}$ of a real valued function $f : \mathbb{R}^n \to \mathbb{R}$ can be defined as a non-differentiable point of the f or a point where all the partial-derivatives of the gradient $\nabla f(p) = [\frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_n}]^T$ are equal to zero, and the Hessian matrix $\frac{\partial^2 f}{\partial x_i \partial x_j}$ is non-singular.

Such as in the case of the previous techniques, the critical points are used for determining cell boundaries, however, the usage of different Morse functions can change the shape of the cells.

For a demonstration of how the algorithm works, the Morse function $f : \mathbf{R}^2 \to \mathbb{R}$, will be $f(x,y) = x$. In fact, this will make the slices a vertical line in the 2D plane, corresponding to the boustrophedon decomposition seen previously. The algorithm will define a sweeping direction parallel to the y-axis and will define a cell boundary whenever it finds a critical point. A critical point can be seen when the surface normal of the obstacle is parallel to the sweeping direction vector, or when the surface normal is perpendicular to the cell boundary defined by the Morse function. This method can be seen in Figure 3.4.

3.2.4 Online Morse-Based Decomposition

A great advantage of Morse-Based decomposition is that, unlike previous techniques, it can be expanded to online algorithms, as demonstrated initially by Acar and Choset in [78] and later a robust algorithm that can handle false readings from the sensors in [79]. To achieve this online

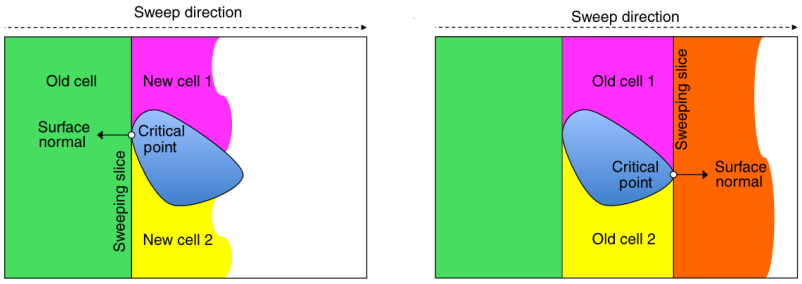


Figure 3.4: Cell decomposition using Morse Function $f(x,y) = x$ [6].

coverage, the authors used an omnidirectional ranger sensor, such as an ultrasonic sensor or a LiDAR, to find critical points in the environment. As such, considering the current agent position as x, let c be a point on the surface obstacle C_i, one can define c_0 as the closest point to the robot as:

$$c_0 = \underset{c \in C_i}{\operatorname{argmin}} \|x - c\| \tag{3.1}$$

Now one can define the distance function between the agent and the obstacle C_i as:

$$d_i(x) = x - c_0 \tag{3.2}$$

And therefore, the gradient of the distance function is:

$$\nabla d_i(x) = \frac{x - c_0}{\|x - c_0\|} \tag{3.3}$$

The gradient of the distance function $\nabla d_i(x)$ corresponds to a unit vector normal to the surface that points from c_0 towards x. By following the same logic as in the Morse function decomposition, one can detect critical points by finding points of the obstacle where the gradient is parallel to the current sweeping direction.

As such, the online algorithm combines the coverage mission with the cellular decomposition by adding cells to the graph while covering. By doing an exhaustive walk through the graph, after covering every cell, the environment should be fully covered.

However, with this algorithm, by performing only back-and-forth motions, some critical points might be missed as seen in Figure 3.5. One solution for this drawback is to combine back-and-forth motion with wall following [6].

3.3 No Decomposition

For CPP missions where the environment is well known, non-complex, regular shape, obstacle-free decomposition might now be necessary. This is the case in a lot of UAV applications, where other than no-fly zones, obstacles are not common or can easily be avoided by a change in altitude.

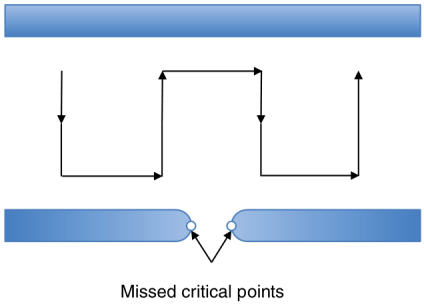


Figure 3.5: Critical Points can be missed in online Morse decomposition if the agent only performs back-and-forth motions [6].

In these situations, simple geometric patterns such as back-and-forth or spiral [7]. As such, one can focus their attention on an optimization problem regarding energy usage, in the following subsections two energy-aware approaches will be considered.

3.3.1 Energy-Aware Back-and-Forth

Di Franco and Buttazzo [19] proposed an energy-aware back-and-forth (E-BF) CPP algorithm, for UAVs in a photogrammetry application.

The algorithm is based on finding one of the vertexes of the longest edge. The sweeping direction is always parallel to this edge, after this, it calculates the optimal number of turns, the length of the segments and turns, and the UAV optimal speed. An example of the algorithm can be seen in Figure 3.6.

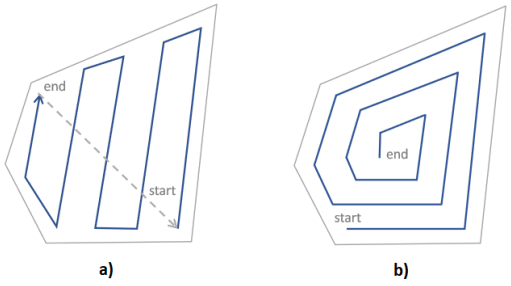


Figure 3.6: a) Energy-Aware Back-and-Forth Algorithm; b) Energy-Aware Spiral Algorithm [7].

3.3.2 Energy-Aware Spiral

Inspired by the E-BF algorithm, Cabreira proposed the energy-aware Spiral algorithm (E-Spiral) [75].

This algorithm starts by building a path that contains all the vertex of the area of interest and afterward starts reducing the radius, converging to the center of the polygon. In this approach, the UAV performs wider turns in comparison with the back-and-forth, enabling it to maintain more

speed and therefore waste less energy in accelerations. An example of this algorithm can be found in Figure 3.6

According to the author, it is currently the most efficient CPP algorithm for convex polygons, considering energy usage.

3.4 Approximate Cellular Decomposition

One of the most common approaches is approximate cellular decomposition[69]. It is usually based on grid maps, where the environment is discretized in square cells, usually the size of the robot or the size of the sensors. It is an approximate technique, as a there is loss of information on the discretization process as seen in Figure 3.7.

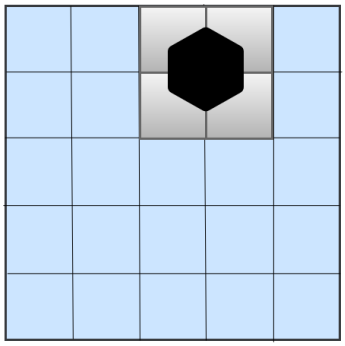


Figure 3.7: Discretization of a polygonal environment with an obstacle.

Despite these drawbacks, grid maps are very versatile, as a cell can have associated information, such as if it is an obstacle, if it was already seen, or the coverage priority, and it can easily be represented as an array. However, grid maps suffer from the curse of dimensionality, with an exponential growth of memory usage with its growth in size and resolution, moreover, they typically require accurate localization systems [6].

3.4.1 Wavefront Propagation Algorithm

One of the first online approaches to CPP is the wavefront (distance transform) propagation algorithm proposed by Zelinsky et al. [80]. In this algorithm, it is necessary to specify a starting and final position, which is already a feature that the previously presented algorithms do not have.

The algorithm is based on the distance transform, as it propagates a wavefront from the goal towards the start cell, assigning a number to each member of the grid in the process. It starts by assigning the value 0 to the goal cell and a 1 to all adjacent cells. This process is repeated iteratively, where all unmarked cells that are adjacent to cells with value n, are assigned with value n_{+1}. It finishes when the wavefront reaches the start cell.

After computing the distance transform, the path is defined by starting on the start cell and choosing the unvisited cell with the highest value. In case of having more than one possible cell, it chooses a legal cell by random choice. An example of the algorithm can be seen in Figure 3.8.

This process is equivalent to doing a pseudo-gradient descent from the start point on a potential function that considers the value of each cell, following the equipotential curves from the top to bottom [6].

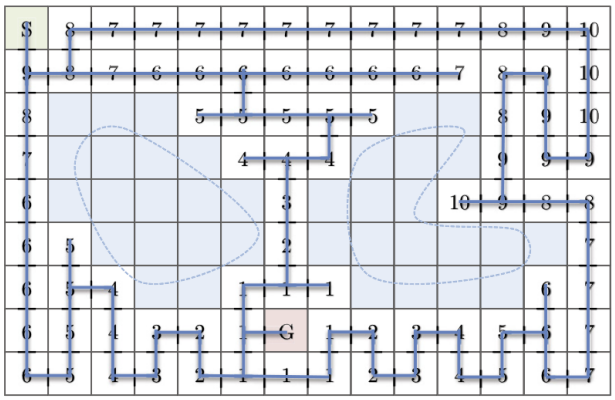


Figure 3.8: Wavefront distance transform for start position (S) and goal (G) and the corresponding coverage path [6].

3.5 Reinforcement Learning Approaches

Most of the surveys in the literature are somewhat outdated and do not consider Reinforcement Learning solutions [6, 7, 69], and the work in [8] does not dwell much time on the topic either.

In this section, some RL approaches to CPP will be presented and discussed.

3.5.1 Offline Q-Learning

The work presented in [72] develops a simple Tabular Q-Learning algorithm for offline coverage path planning in an environment with known obstacles. The performance of the algorithm is non-optimal, although shows better results than the genetic algorithm it was paired against in an 8×8 grid map. Although it shows that RL can be applied to CPP tasks, the choice of tabular algorithms does not enable it to scale to bigger maps, and the utility of the algorithm is somewhat reduced by the fact that the performance is non-optimal and there are a lot of existing algorithms that can solve the proposed problem.

3.5.2 Distributed Multi-Agent Online Q-Learning

In [73], the authors present a Distributed Multi-Agent Online Q-Learning area coverage algorithm. It is once again based on Tabular Q-Learning with an adaptation of the learning process for multi-agent (Q-Transversal). It introduces some relevant ideas such as having an information map γ that is shared between all agents, its environment state-space contains the next action of each agent, enabling recursive thinking and the use of a heuristic in the training process when it is priorly known that there are no good moves.

It clearly shows the potential of RL for this specific problem, especially for multi-agent settings, which are inherently more difficult to tackle with conventional algorithms. However, the usage of a Tabular algorithm when the state space has the whole map, makes the algorithm scale poorly with map size. Despite that, the work shows interesting results when compared to state-of-the-art multi-agent algorithms and has built-in fault tolerance and coverage guarantees.

3.5.3 Work Developed by LG Electronics Advanced AI Team

The LG Electronics Advanced AI team has carried out a series of studies [81, 82, 21] exploring the use of Deep Reinforcement Learning (Deep RL) algorithms for Coverage Path Planning (CPP). Notably, these works represent some of the few Deep RL studies aimed at developing a general, application-agnostic algorithm, with a broader focus on evaluating the potential of DRL for this type of problem.

In [21] a study on Deep RL for the CPP is conducted, comparing various state-of-the-art algorithms. The algorithms used were Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C), Deep Q Network (DQN), and Deep Q Network with Prioritized Experience Replay (DQN-PER). They showed that from all of them, DQN-PER performed the best. After training the RL algorithm, the authors compared it to five different state-of-the-art classical CPP algorithms and achieved significantly better coverage and less overlap. Another important result of this work is a Hybrid RL approach that combines the BA* algorithm with the DQN-PER algorithm, substituting the A* with RL. This proved a significant improvement when compared to the simple RL approach by reducing the policy space of the RL algorithm.

Another work focused on an algorithm that could generalize across various environments [82]. Using DQN-PER again, the authors demonstrated that a model trained in one environment could generalize to others with minimal transfer learning. However, without retraining the model on the new environment, the results exhibited around 40% overlap, and even after training, overlap remained high at about 20%. One could argue that because transfer learning is necessary, the agent may not genuinely be learning to generalize, and the application could always be limited by the need to include a learning process whenever it encounters a new environment. Another interesting finding was developing an area-agnostic agent, which could achieve 90% coverage in a 19x19 environment using a 15x15 representation. Still, there were no guarantees for complete coverage or a clear understanding of how generalizable the results were. Nevertheless, this approach is among the most promising for developing a general CPP algorithm.

A final work [81], shows that the algorithm can handle dynamic obstacles. However, it was unclear how the agent could identify dynamic obstacles, as the state-space representation was based on a single frame with all obstacles in the same channel. This suggested that the agent had no way of discerning which obstacles were moving and in which direction. As the algorithm was always trained on the same scenario, it is plausible that the agent "memorized" the environment's patterns. The work only documents the initial location of the moving obstacles, and the number and location are fixed and deterministic, but their movement pattern and ranges are not elaborated.

3.5.4 DQN Methods for CPP and Data Harvesting

The works of Mirco Theile and Harald Bayerlein [28, 83, 84, 85] offer a significant contribution to the field of deep reinforcement learning approaches for coverage path planning and data harvesting for UAV applications. As these works build incrementally upon each other, the focus of this document will be on the latest single-agent paper [28] and the multi-agent work [84].

One of the most promising approaches to the CPP problem is [28]. The authors developed a Deep RL algorithm based on Deep Q Networks that receive as input images corresponding to grid maps. One of the grid maps is global, containing all the information about the problem with little detail. There is another map with local information observed by the agent, with greater detail and resolution. This approach combines both the offline and online methods of CPP, and gives an intuition where the local map is used for more immediate decision-planning and the global map is used for long-term planning. The authors also show another innovation. By centering the image on the agent's position, the results are improved significantly. This can be seen as the agent learning in its own inertial referential, being that all positions in the map are relative to it.

The results of the work are promising, showing not only good performance but also the ability of the algorithm to generalize. In the paper, the algorithm is benchmarked in a generic CPP scenario and a Data Harvesting scenario, showing that the same agent could perform well in both situations. However, the main limitation is that the framework is limited to a single map, not being flexible to size and different configurations. This limitation is evident in the paper, as two different network configurations are used for the two distinct scenarios.

All the innovations presented in the single-agent framework are generalized to a Multi-Agent Reinforcement Learning Algorithm that uses the Double Deep Q Network in a parameter-sharing scheme [84]. While the algorithm is specifically tailored to a data-harvesting scenario, it is one of the most important works in the literature, and some techniques and ideas presented in this work will be expanded in this dissertation.

3.5.5 Patrolling the Lake Ypacarai

One example of Deep Reinforcement Learning being used in real-world scenarios is in the proposed Automated Surface Vessels to patrol Lake Ypacarai [86, 87].

In the first work [86], algorithms based on the Deep-Q Network are used to perform patrolling tasks in a known environment. In this specific task, the agent can have two roles: homogenous patrolling or heterogenous patrolling. The state space is represented in an RGB image representing the importance matrices, or the priority of visiting a specific region. The authors show that the Double DQN achieves better results than the vanilla DQN and that the agent can perform well in both scenarios. However, it is mentioned that if any non-modeled obstacles appear, retraining would be necessary, and the algorithm is also restricted to this naturally specific to this scenario.

In the latter work, the authors expand the work to a multi-agent framework once again by parameter sharing. They also introduce the Dueling architecture to the RL scheme, showing that it improves the results, even in the single-agent scenario. The results for multi-agent are better than

classical algorithms and show that this type of task can be adequately solved with Deep RL and parameter sharing.

3.5.6 PPO For Cleaning Robots

An example of work developed in this area getting to consumer electronics can be seen in [88, 74], where PPO offline algorithms are developed and tested for the LG cleaning robot R9, improving the results obtained with classical algorithms and showing the interest of companies to develop Reinforcement Learning solutions for their products. LG Korea developed this work, and the main objective was to minimize energy consumption during cleaning tasks on the R9 robot. Compared to the other works in the literature, it is one of the only works focusing more on Policy Gradient methods.

3.6 Final Considerations

The CPP problem has a very large literature on classical and heuristic methods. Although these methods can provide satisfactory solutions, they are often tailored for specific scenarios and lack flexibility for scenarios that require real-time decision-making or involve complex problem constraints, especially in multi-agent scenarios.

Reinforcement Learning is an interesting and promising solution for Coverage Path Planning [8]. It gives the adaptability that classical algorithms can not, and if used correctly, can give the potential to achieve better solutions than the classical methods. Despite that, it is clear that the literature on this topic is still evolving, with new approaches being presented every year.

One thing to take away from the literature analysis is the lack of general algorithms for CPP. Most of the works are focused on only specific maps [85], or designed thinking about only a use case [86]. The closest to a general algorithm is the work done by the LG Electronics Advanced AI Team [21], as in some of their works, the starting position is randomly generated, but the tested map pool is still very shallow and limited. The author of this document finds that the current literature lacks algorithms capable of generalizing to any scenario, known or unknown, and capable of handling different map sizes, especially when the size exceeds the state representation. Another interesting, unexplored approach is to have an algorithm that can deal with different sensor payloads and work in both online and offline scenarios without requiring retraining.

Keeping these considerations in mind, the remainder of this document will describe the developed work, aimed at addressing the gaps identified in the existing literature.

Chapter 4

System Architecture

This chapter will focus on the software development side of the dissertation. It is divided into three sections. Section 4.1 will give a brief overview of the overall software architecture. This will be followed by a more in-depth overview of the Environment in Section 4.2 and the Agents in Section 4.3.

4.1 Architecture

A Reinforcement Learning algorithm has two main components to take into consideration: the Environment and the Agent. With this in mind, the chapter will begin by analyzing this overall architecture.

Every reinforcement learning method assumes an episodic format, where one or multiple agents interact with an environment, which will evolve until episode termination. As seen in the theory of Markov Decision Processes, at every timestep, the agent takes an action and receives a reward and an observation of the environment. On the software side of development, the work must ensure this fundamental interaction is possible above all else. A visualization of the interaction can be seen in Figure 4.1.

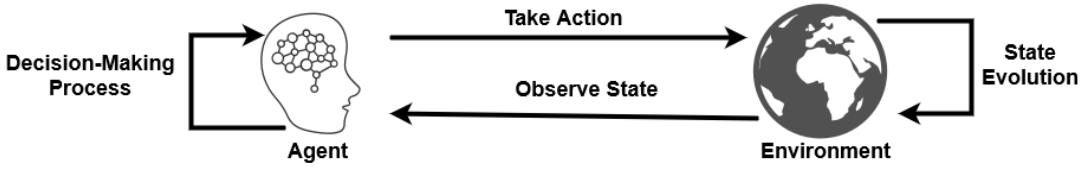


Figure 4.1: UML Class Diagram depicting the high-level architecture.

This process results in two clear software components - the Agent and the Environment. It is important to note that both components must be completely decoupled in implementation. The agent and environment are seen as black-boxes by each other, and therefore, the only assumption and guarantee is that the actions selected by the agent are valid for the environment and that the agent's decision-making process can be based on the state it observes.

A typical interaction can be described in the Algorithm 8.

Algorithm 8 Simplified Agent-Environment Interaction

```
1: s ← env.reset()                                  ▷ Sample first environment state
2: while True do
3:     a ← agent.select_action(s)                   ▷ Agent chooses next action
4:     s′, r ← env.step(a)          ▷ Environment returns observation and reward
5:     s ← s′                       ▷ Update the state for the next iteration
6:     if s is Terminal then                        ▷ Episode Termination
7:         s ← env.reset()                          ▷ Sample next episode first state
8:     end if
9: end while
```

Making a comparison between Figure 4.1 and Algorithm 8, the decision-making process is depicted by the agent method "**select_action**" and the process of taking an action, the evolution of the state, and its observation is encapsulated by the environment method "**step**." The "**reset**" method serves only as a way to deal with episode termination.

This interaction sets the tone for the remainder of the chapter, where the software structures for both the Environment and Agent will be described and explained. The most important thing to keep in mind is that the framework will be developed to be as generic and reusable as possible. This enables the use of the same architecture for all problems described in the dissertation and other future works.

4.2 Environment

The Environment is a fundamental piece of every Reinforcement Learning algorithm. For learning to occur, relevant data must be generated through interactions. Most of the time, using real-world data is not feasible, either due to the huge amounts of necessary data or due to the cost of operation. This makes simulated environments a common feature of most works, and this one is not an exception.

The developed environment will follow the design patterns of the OpenAI Gym environments [89], which focus on creating the environment as a purely black-box interface for the agent, only requiring two functions for any learning algorithm: ***step*** and ***reset***. The first is used to receive the agent's action and update its state, and the latter serves to deal with episode termination.

4.2.1 Architecture

The environment was designed using Object-Oriented Programming (OOP) principles. This approach was chosen to streamline the development of new features and to separate the environment's components distinctly. A UML Class Diagram illustrating the architecture is provided in Figure 4.2.

The UML class diagram displays a high-level description of the developed software and does not have every detail. Nonetheless, the document will go into further detail in each component.

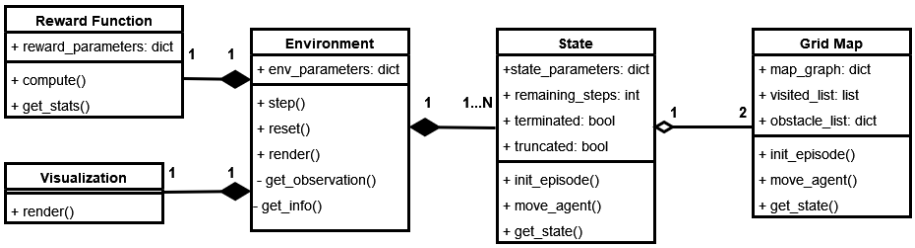


Figure 4.2: UML Class Diagram depicting the high-level architecture.

Considering the proposed architecture, the **Environment** class serves as an interface for the agent to interact with, and most of the functions and features are in the other classes that the agent does not have direct access to. A typical sequence of interactions between an agent and the environment is depicted in Figure 4.3 through a UML Sequence diagram.

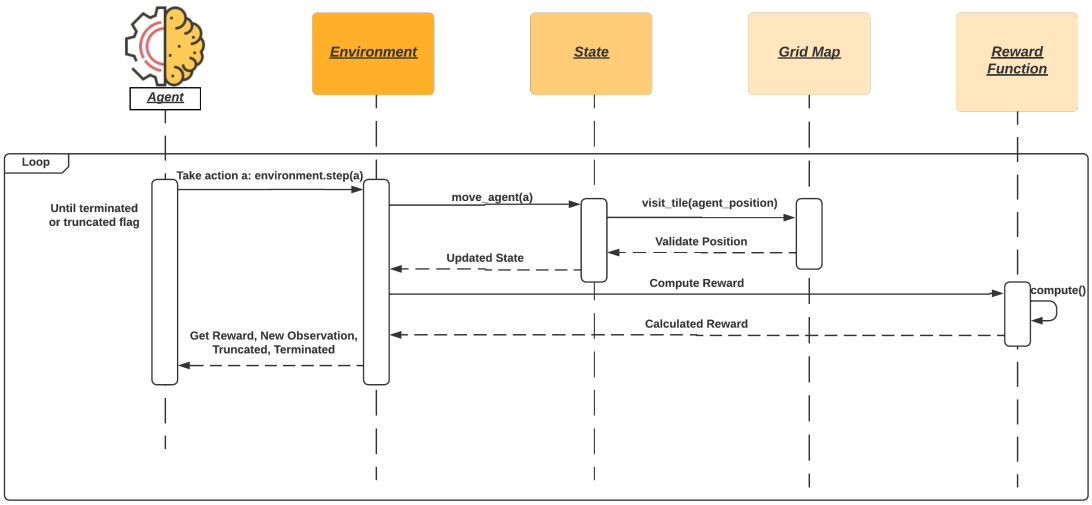


Figure 4.3: UML Sequence Diagram of the proposed architecture.

4.2.1.1 Configuring The Environment

The primary motivation behind developing the environment from the ground up was to maintain complete control over its features, essentially transforming it into an adaptable sandbox. Compared to many well-established Gym environments, this custom environment offers a higher level of customization and can be dynamically adjusted during runtime.

Configuration for the environment can be loaded from a YAML file. The values stored in the file are then assigned to variables within the **Environment** and **State Parameters** variables. The structure of the configuration file is detailed in Table 4.1.

It is possible to have multiple environments in the same configuration file by repeating the same structure. This allows the implementation of techniques such as curriculum learning and transfer learning without interrupting the learning algorithm.

Table 4.1: Environment Configuration.

Variable Name	Possible Values	Description
name	Any	Environment Name
base_steps	Any $\mathbb{N}^+$	Number of Steps for training.
number_agents	$[1, \text{size}^2[$	Number of Agents.
number_agent_random	True/False	If true, then the number of agents in each episode will be a random number in [1,number_agents].
sensor	Sensor Type	Type of sensor of the agent.
sensor_range	Any $\mathbb{N}^+$	Range in map cells of the sensor.
size	$[2, +\infty[$	Size of an edge of the map.
min_size	$[2, \text{size}[$	Minimum size of an edge of the map.
random_size	True/False	If True, the map size will be a random number in [min_size, size], else will be *size*.
number_obstacles	$[0, \text{size}^2 - 1[$	The number of obstacles cells in the map.
obstacles_random	True/False	If True, the number of obstacles will be a random number in [0,number_obstacles], else will be *number_obstacles*.
starting_position	Any Valid Position	Starting Position of the agent. Only functional in single-agent.
starting_position_random	True/False	If True, the agent starts in a random position. Overwrites *starting_position*.
random_coverage	True/False	If True, up to 70% of the map will be already covered in the first step of the episode.
map_configuration	Empty or YAML file	Load a Single Map.
dataset_path	Empty or path to a file	Load a dataset of maps.
load_state	True/False	If True, the whole ***State*** class is loaded, else only the ***Grid Map*** is loaded.
load_random	True/False	If True, the maps in the dataset will be loaded in random order.

These configurations also ensure that the generated environments can vary in every episode, for example, by setting any random variable flag to *True*. Furthermore, by altering the values in the parameter objects, the environment's configuration can be dynamically modified during runtime since these parameters are accessed each time the ***reset*** method is used.

4.2.2 State

While the ***Environment*** class is an interface class that holds all the components, the ***State*** class is responsible for the actual simulation and holding all the relevant state information. It has two main methods, the ***init_episode*** and ***move_agent***. These two methods are fundamental for the inner workings of the environment and will be explained in the following subsections.

4.2.2.1 Initializing An Episode From Scratch

The ***init_episode***, as the name suggests, is the one responsible for initializing every episode. It is called whenever the agent uses the environment ***reset*** method. In this method, the parameters of the environment are read, and a suitable scenario is created. A pseudo-code of this method will be provided in Algorithm 9.

Algorithm 9 Initialization of Episode

```
1:  procedure INIT_EPISODE
2:      width, height ← The defined map size          ▷ Randomly sample size if necessary
3:      Randomly assign each agent a valid unique position
4:      if parameters.map_data is provided then
5:          global_map ← GridMap initialized with parameters.map_data
6:      else
7:          Determine the number of obstacles
8:          Randomly assign each obstacle a unique position
9:          global_map ← GridMap initialized with the data the map size and obstacles
10:         fix global_map if necessary
11:     end if
12:     if params.sensor = "full information" then
13:         local_map ← global_map
14:     else
15:         local_map ← GridMap initialized with agent positions
16:     end if
17:     Mark each agent's position as visited on the local map
18:     Cover up to 70% random tiles if params.random_coverage is True
19:     Initialize all state variables                ▷ Truncated, Terminated etc.
20: end procedure
```

4.2.2.2 Fixing Randomly Generated Maps

As the initialization algorithm relies upon randomly sampling obstacles, there are situations where the generated map can not be fully covered by the agent, creating impossible scenarios. An example of such a scenario can be seen in Figure 4.4. Consider that the black cells are obstacles, and the blue circle is the agent position.

A custom algorithm is deployed to fix scenarios like this, ensuring that every scenario can be solved. As the Grid Map is represented as a bidirectional graph, the algorithm is based on deploying depth-first search (DFS) from the agent position and verifying that every non-obstacle cell in the graph is reachable. If a node in the graph can not be visited, remove it and its edges. This algorithm pseudo-code can be visualized in Algorithm 10.

4.2.2.3 Loading Maps From Dataset

One potential drawback of using the described algorithm to generate the map pertains to the time complexity of the map correction algorithm. This algorithm operates at $\mathcal{O}(V+E)$ time complexity,

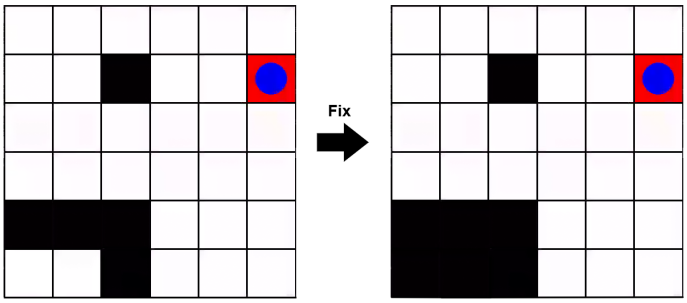


Figure 4.4: On the left is a scenario that is impossible to finish, and on the right is the same scenario after applying the fixing algorithm.

where V represents the number of vertices and E denotes the number of edges. Consequently, it scales poorly with map size.

In practical terms, in maps bigger than 20×20, the learning algorithm spent one-third of the runtime generating and fixing maps, being a significant slowdown to the learning process.

To address this issue, a functionality was added to load a file containing multiple instances of the ***State*** class. Such a file can easily be generated by repeatedly calling the ***init_episode*** method and appending the resulting object to a list. After reaching the desired number of iterations, the list can be saved as a file.

There are two ways to utilize a dataset in this context. The first method simply involves randomly sampling a state from the loaded list and using it as is. However, this method necessitates that all aspects of the state be reused, including the number of agents and sensor type. For greater flexibility, an alternate method was implemented that only uses the map from the loaded state and then initializes the episode as done in the ***init_episode*** method. This alternate method, ***init_from_map***, skips all parts related to map generation. Given that the most time-consuming parts of the method involve generating and correcting the map, reusing the map offers almost all the benefits of a newly generated state without the associated time expenditure.

This approach has enabled a speed increase of approximately 33% in training on larger maps.

Algorithm 10 Map Correction

```
1:  procedure FIX_MAP(start)
2:      visited ← Apply depth-first search (DFS) starting from start
3:      non_obs ← Tiles that are not considered obstacles
4:      should_be_obs ← Tiles that are in the set (non_obs \ visited)
5:      for each tile t in should_be_obs do
6:          for each neighbor of t do
7:              Remove t from the neighbor's adjacency list
8:          end for
9:          Remove all elements from t's adjacency list
10:         Add t to the obstacle list
11:     end for
12: end procedure
```

Furthermore, this method enhances the evaluation and validation of algorithms, providing a robust and replicable setting for their performance testing.

One significant advantage of this feature is that users can share datasets, enabling them to test their own approaches. This is particularly useful given the CPP literature's lack of standardized evaluation datasets.

4.2.2.4 Implementing Partial Observability

One of the objectives of the algorithms that will be tested in the environment is to have agents that can do active exploration or online coverage path planning, completely relying on sensor information. To implement this, the ***State*** class has two ***Grid Map*** objects. One is the ***global_map***, which has all information, and the other is the ***local_map***, which is initialized and built using the agent information.

Two types of sensors are implemented: a camera and a laser scanner. A third situation where the agent has full information and does not rely on sensors is also considered. The main difference between the laser and the camera is that it is considered that the camera has a field-of-view that enables it to view beyond obstacles, for example, mounted on a UAV, whereas the laser does not see beyond any obstacle.

When the agent has access to full information, the local map can be a duplicate of the global map. When an agent calls the ***step*** method from the ***Environment*** object to perform an action, this information must be transmitted to the ***State*** object for updating the state, as illustrated in Figure 4.3.

4.2.2.5 Updating The State

Whenever the agent takes an action, this action must be used to update the environment state. While agent calls the ***step*** method from the ***Environment*** object to take the action, then the environment object must send this information to the ***State*** object in order to update the state, as seen in Figure 4.3. The ***State*** object has a method named ***move_agent***, that receives the action, and its logic can be visualized in Figure 4.5.

This method has a built-in safety controller that does not let the agent take dangerous actions. Every time the action results in a crash with the environment or other agents, the safety controller does not let the agent move.

The method operates synchronously. Thus, when multiple agents are present, priority is granted to whichever action was first added to the action list. In a synchronous operation, if two agents move to the same cell, the agent given priority occupies the tile while the other agent remains in place.

If an agent does not select an action, either due to communication failure or to simulate an asynchronous system where each agent acts individually, the controller ensures that the agent remains stationary.

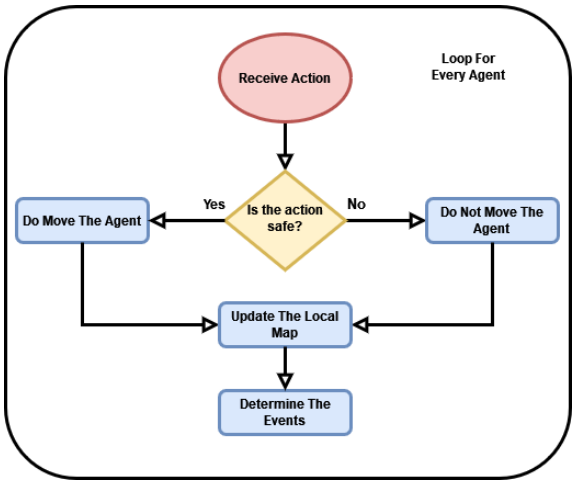


Figure 4.5: Flowchart of the ***move_agent*** method.

The ***move_agent*** method also returns a list of events that occurred during the transition. These events, such as visiting a new tile or a collision, are used to implement the reward function. However, as these events are problem-dependent and this section aims to provide only a high-level overview, they will not be explored in depth.

4.2.3 Reward Function

The Reward Function is the environment component responsible for computing the reward for the agents. In this environment, the ***compute*** implements the function $R(s,a,s')$, by receiving the current state s, action a, and next state s' and the list of events that occurred in the transition.

The utilization of a list of events in the reward function is motivated by the need for adaptability and generalization. Despite the function being problem-specific, a set of events is commonly encountered across various navigation tasks. Incorporating these events as the basis for the reward function makes generalization and adjustments easier. The basic implemented events can be seen in Table 4.2.

Table 4.2: Available Events

Event	Explanation
Blocked	The agent had a collision.
New	The agent visited a new tile.
Repeated	The agent repeated the same action as in the last timestep.
Finished	The agent has visited all cells.
Timeout	The episode was truncated due to time constraints.
Waited	The agent did not move.
Time Step	Constant Reward in every transition.

Each event occurrence is linked with a specific reward value, which can be adjusted to adapt the function to different problems. These values are stored in a class named ***Reward Parameters***.

Given that these values are often specific to the algorithm and the problem at hand, a more in-depth discussion of the reward function be provided in the next chapters.

4.2.4 Visualization

To enable the visual representation of the environment, a Graphical User Interface (GUI) was developed leveraging the PyGame framework[1].

The ***Visualization*** class contains a ***render*** method that takes the current state as an input and visually renders the map in a new window. The rendering process involves transforming the constructed graph into a 2D RGB array, where each element of the array corresponds to a cell on the map and its color denotes the type of cell it represents.

The color scheme used is as follows: black cells represent obstacles, white cells indicate points of interest, red cells are those that have already been visited, and blue cells are the ones that the agent has not yet visualized in scenarios of partial observability. The agents are represented by blue circles. An example of this color-coded visualization can be seen in Figure 4.6.

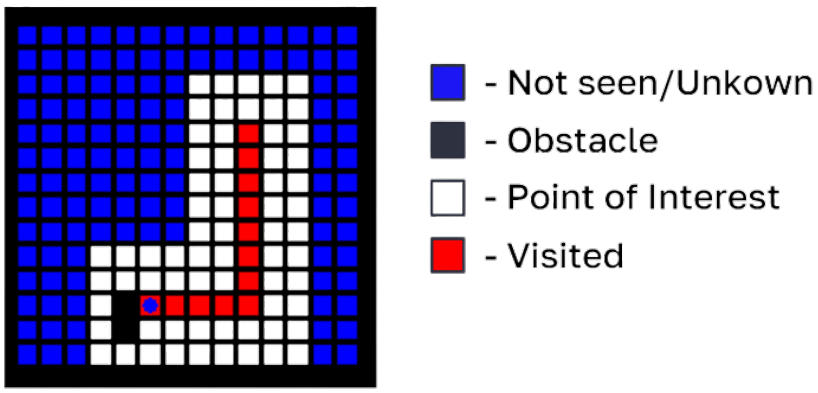


Figure 4.6: An example of environment rendering with the associated color code.

4.3 Agent

In the context of Reinforcement Learning, the definition and boundaries of an agent are not as distinct as those of the environment. The agent is essentially an entity that is capable of interacting with the environment. From a theoretical standpoint, this could be something as simple as a random number generator. This section provides an overview of two types of agents that have been developed for this project: Tabular Agents and Deep Reinforcement Learning Agents.

[1] https://www.pygame.org/

4.3.1 Tabular Reinforcement Learning Agent

In Tabular algorithms, due to the simplicity of the techniques, the agent can be reduced to a look-up table. The implementation is based on the Python collections *defaultdict* [2], a dictionary that can be configured with a default number of entries for any key. By defining the entries as the number of actions in the action space and the key as the state, one can implement a tabular state-action value function $Q(s,a)$.

For example, if the agent follows the greedy policy, it merely needs to query the dictionary with the current state as the key and then apply the *argmax* function to the entries. This allows the agent to select the action corresponding to the highest value.

4.3.2 Deep Reinforcement Learning Agent

The implementation of Deep Reinforcement Learning Agents is inherently more complex compared to Tabular Agents. To manage this complexity, an OOP approach is utilized in implementing the agent, mirroring the structure used for the environment. This is a widely adopted strategy because it facilitates components' modularization, isolation, and reusability across different algorithms. However, some authors argue for a single-file functional approach, suggesting it provides a clearer overall picture and leads to faster development times [3].

Regardless, the architecture for the agent used in this context can be visualized through a UML Class Diagram, as shown in Figure 4.7.

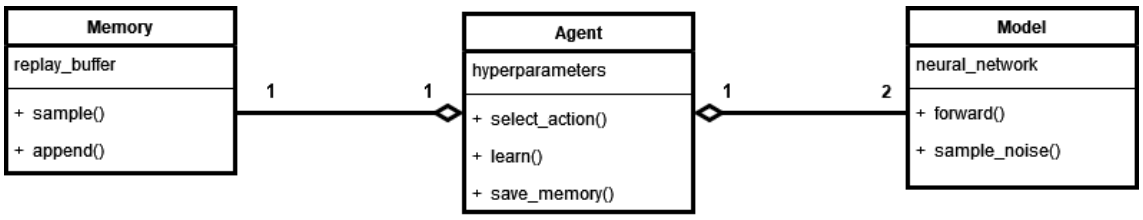


Figure 4.7: UML Class Diagram of the Deep Reinforcement Learning Agent.

As with the environment, the ***Agent*** class encapsulates all components and serves as the primary interface.

4.3.2.1 Implementation

The selected Deep Learning framework for implementing the model and learning algorithm is PyTorch[4]. This comprehensive framework facilitates the construction of various Neural Network configurations with its extensive feature set and well-documented resources, alongside implementing the training algorithm. In this study, ADAM was the optimizer of choice.

[2] https://docs.python.org/3/library/collections.html#defaultdict-objects
[3] https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/
[4] https://pytorch.org/

The development of the algorithm was substantially inspired by, and based on, the OpenAI baselines repository[5]. Notably, the implementation of the ***Memory*** class, which employs Prioritized Experience Replay, was heavily informed by this repository. Consistent with the recommendations in the original paper [57], the ***Memory*** class utilizes the min sum-tree and max sum-tree for effective priority management.

The developed code can be accessed in the authors GitLab repository [6].

4.3.2.2 Configuration Hyperparameters

The agent and the learning algorithm can be configured using a range of hyperparameters. These hyperparameters are passed through an argument parser in the main launching script. Table 4.3 provides a summary of these hyperparameters.

Table 4.3: Hyperparameters for Configuring an Agent

Hyperparameter	**Description**
history-length	Number of consecutive frames stacked
noisy-std	Initial standard deviation of noisy layers
memory-capacity	Experience replay memory size
replay-frequency	Frequency of sampling from memory
priority-exponent	Prioritized experience replay exponent
priority-weight	Initial prioritized experience replay importance sampling weight
multi-step	Number of steps for multi-step return
discount	Discount factor
learning-rate	Adam Optimizer Learning rate
adam-eps	Adam Optimizer epsilon
batch-size	Mini-Batch size for learning
norm-clip	Max L2 norm for gradient clipping
learn-start	Number of steps before starting training
tau	Factor for soft update of target parameters

The used hyperparameters and choices will be expanded in the following chapters, as these choices are very much problem-dependent.

[5] https://github.com/openai/baselines
[6] https://gitlab.com/jpfpc/drl_cpp/-/tree/main/src/Rainbow

Chapter 5

Online Coverage Path Planning with no Explicit Map Representation

This chapter will delve into a case study focusing on the application of classical Tabular Reinforcement Learning Methods for Online Coverage Path Planning. It will begin in Section 5.1 with a presentation of challenges and objectives for the work in this Chapter. This is followed by a formulation of the problem in Section 5.2. After defining the problem, the solution will be developed and discussed in Section 5.3. In Section 5.4 the developed algorithm is tested, and the results are analyzed and discussed. Finally, some brief conclusions on this work are outlined in Section 5.5.

5.1 Objectives and Challenges

The work developed in this chapter is a stepping-stone for more complex methods and algorithms that will be analyzed in the next chapters. It serves as an introduction to the usage of Reinforcement Learning, grounding the complexity of the topic by using only classical Tabular methods and analyzing how adequate these methods can be for a coverage path planning problem. By developing simpler solutions, this chapter sets the tone for how the problems will be tackled in the rest of the document.

The main point of contention is how to model this problem to be solved by Reinforcement Learning methods. The state space in a coverage path planning setting can be very large. When considering the state as the direct map representation, with the presence of obstacles, varying starting positions, and differing map sizes. This can render the problem non-trivial to solve within a tabular setting. As an illustration, for a 10×10 map with no obstacles, there are approximately $(10^2)^3 = 1 \times 10^6$ potential states[1]. Considering four possible actions, the Q-Table would contain 4×10^6 entries. Assuming each entry is a 64-bit floating point number, the Q-Table size is approximately 32 Megabytes, and that is for environments of this specific size alone and with no obstacles. By adding just one obstacle, there are 100 more possible maps, and each would require

[1] There are 100 cells, and they can either be a point of interest or not, leading to 100^2 combinations. Considering that the agent can take any position, there are a total of 100^3 possible states.

approximately the same memory size. Considering that this analysis is just done for this map size and the objective is to be able to deal with any map size and number of obstacles, it is evident that the state cannot merely be the map representation and that the problem must be modeled appropriately.

With these constraints in mind, the final contribution of this chapter will be a Tabular Temporal Reinforcement Learning Method that can be used by a generic agent in a general Coverage Path Planning setting. The novelty comes from the techniques that are used by the algorithm to adapt to the challenges of the problem and deliver a general and dependable solution.

The objectives of this chapter are as follows:

- Model the Online Coverage Path Planning problem as a Partially Observed Markov Decision Process.
- Develop Tabular Temporal Reinforcement Learning Methods to solve the POMDP.
- Study the performance of different classical RL Methods on this setting.
- Address the challenges of using unaltered RL methods for the CPP problem.
- Deliver an algorithm that is capable of performing the task in various different configurations at a near-optimal level of performance.

5.2 Problem Statement

The goal is to perform a general online coverage path planning task that minimizes coverage path overlap, or in simpler terms, minimizes the number of unnecessary steps. As depicted in Figure 5.1, the environment undergoes a process of approximate cell decomposition, leading to a $N \times N$ square grid map where the cell size equates to the robot's dimensions - a common practice in most CPP approaches. The grid's referential is based on cardinal directions, with the map's northernmost points located at the top of the grid.

The agent is conceptualized as a generic robot capable of moving in four directions (North, East, South, West) at any given timestep, irrespective of the previously taken direction. The robot is equipped with a range sensor that can detect any obstacle in the eight intercardinal directions, meaning it can also detect obstacles that are diagonal to the agent. The range of the sensor is limited with the value of r grid cells in the directions of possible movement and a value of $\frac{r}{\sqrt{2}}$ in the diagonals.

For visualization purposes, the environment is depicted as a $N \times N$ grid map where each cell encodes specific information (as seen in Figure 5.1 and Figure 5.2). Red cells correspond to covered areas, white cells represent non-covered, black cells designate obstacles, blue cells are unseen areas, and the blue circle symbolizes the agent.

Despite the environment being discretized into a 2D grid, the algorithm should solely rely on the sensor data to accomplish the task. This implies that even though this form of representation is

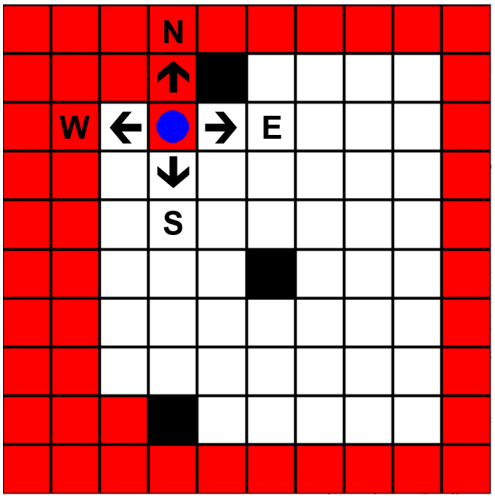


Figure 5.1: Example of a 10×10 environment featuring three obstacles.

used in the environment, the Reinforcement Learning algorithm itself remains agnostic to the map, requiring only sensor readings to function. Because of this reason, this method will be considered as a Coverage Path Planning approach with no explicit map representation.

This sensor-dependent design means that the state-space size is limited only by the sensor's range and resolution, effectively addressing the problem of map representation scalability with size. Moreover, this approach synergizes well with novel localization methods such as PASTA [90], which rely solely on LiDAR information for environment localization.

5.3 Methodology

This section will expand on the concepts and ideas from the problem statement and provide a solution to the Online CPP Problem using Reinforcement Learning. With this in mind, the problem will be formulated as a POMDP and the design choices for the Observation Space, Action Space, Reward Function, and Policy will be described and explained.

5.3.1 Observation Space

An intuitive approach to represent the environmental state might be to utilize an $N \times M$ array that entirely represents the discretized map, encoding information about coverage, obstacles, and the agent's position. However, this representation's scalability is inadequate [6], especially for a Tabular RL approach. For a basic 6×6 grid map with no obstacles, there are $36^3 = 46656$ potential states, corresponding to each cell's state (visited or not) and the agent's position. Furthermore, this representation is solely applicable to this specific environment, implying that if a single obstacle appears, or if there is a size change or a dynamic environment, a completely new algorithm would need to be trained.

With this in mind, for an online coverage task, the observation space Ω is represented as the last K processed readings of a range sensor in every direction and the last K actions. This sensor

will also detect the distance to walls or obstacles, and a map will be built and stored in memory as in Simultaneous Localization and Mapping (SLAM) techniques.

For this purpose, at each time step t, the ranging sensor will give the agent an array of distances $r_t = [d_N, d_S, d_E, d_W, d_{NE}, d_{Nw}, d_{SE}, d_{SW}]$, where the d_x values are between 0 and the maximum range r. The observation function $\mathcal{O}$, will receive this array and the constructed map as inputs, and modify the values of r_t resulting in the array r'_t. This way, a value will be changed to -1 when the adjacent cell is a wall/obstacle, kept at 0 when the adjacent cell was already covered, and any value bigger than 0 is the distance to the closest covered cell or obstacle in the direction. A visualization of this transformation is shown in Figure 5.2

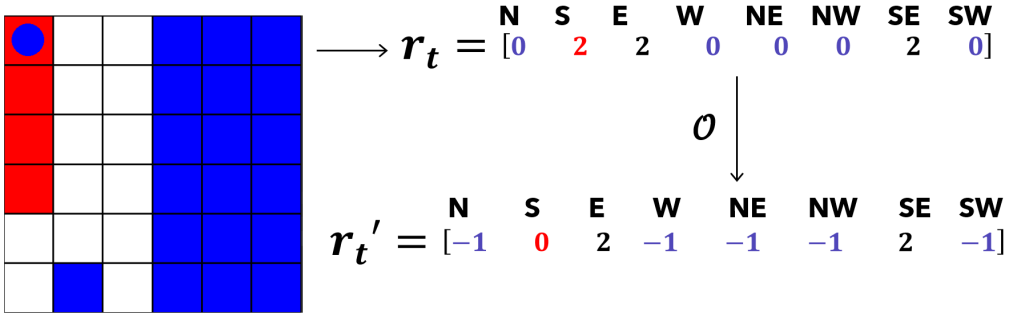


Figure 5.2: Example of the transformation form $r_t \rightarrow r'_t$. Sensor range $r = 2$.

After modifying the values, the observation function will create a tuple with (r'_t, a_t), add it to the newest observation, and remove the oldest tuple in o.

As such, an observation $o \in \Omega$ is a tuple composed as follows: $o = [(r'_t, a_t), (r'_{t-1}, a_{t-1}), ..., (r'_{t-K}, a_{t-K})]$. The number of sensor readings K can be tuned depending on the environment size and the available memory. It is important to note, however, that the nature of the algorithm inherently limits it, as the number of possible observations will increase exponentially, making it unfeasible to train.

5.3.2 Action Space

The formulated agent has five actions. Four of these are directly related to the motion of the agent, corresponding to the directions it can take at each time step: North, East, South, and West as seen in Figure 5.1. The agent can only move one cell at each timestep.

The fifth action is not exactly an action in the sense of classical RL concepts, but in this proposed framework can be viewed as one. More precisely, it is a heuristic that moves the agent in the direction of the closest non-visited cell, where the closest cell is determined by euclidean distance and the path towards it is calculated by the Dijkstra Algorithm, considering that the map is represented by an adjacency graph. This heuristic is not always available and is a way to mix concepts of classical algorithms with RL. The usage of this heuristic speeds the training process by minimizing the time in non-interesting parts of the state space, i.e., surrounded by already

visited cells. Furthermore, it gives complete coverage guarantees in any non-dynamic environment considering there are no sensor faults.

Therefore, the action space can be defined as $\mathbb{A}$: [North, South, West, East, Heuristic].

5.3.3 Policy

Temporal Differences Reinforcement Learning Algorithms require a policy that provides continuous exploration for the learning process. For the proposed problem, there are a lot of states $s \in \mathbb{S}$ that are inherently uninteresting, such as going against a wall or revisiting an already-covered cell. A solution is proposed based on a heuristic, however, this should only be used when deemed necessary.

For this purpose, the chosen policy was a modified decaying ε-greedy. The policy can be described by the expression:

$$a_t = \begin{cases} \text{argmax}_a\, Q(s,a) & \text{w.p } 1-\varepsilon_t(s) \\ \text{Random action in } \mathbb{A}' & \text{w.p } \varepsilon_t(s), \end{cases} \tag{5.1}$$

where choosing the optimal action is referred to as exploiting and choosing a random action as exploring. The trade-off between exploration-exploitation is controlled by the hyperparameter ε_t. This parameter is time-varying and its value decays following the expression $\varepsilon_t(s) = \frac{1}{\sqrt{n_t(s)}}$, where $n_t(s)$ is the number of times that the state $s \in \mathbb{S}$ was visited at time-step t. This function was chosen based on [53].

Regarding the available actions, we introduce the following modifications at each time step:

1. The only available actions of the action space $\mathbb{A}$ are the ones that move towards an adjacent non-covered cell, forming the action space $\mathbb{A}'$.

2. The chosen action will be the heuristic if and only if all adjacent cells are already visited or obstacles.

3. In the case that the state was never explored, the chosen action will be the heuristic.

A visualization of an example of rule number two can be seen in Figure 5.3.

5.3.4 Reward Function

The reward function is one of the most important components of any RL algorithm, being absolutely necessary for good convergence and performance. For this problem, it was designed by rules based on the intended behavior of the agent, and on a set of events. More precisely, the

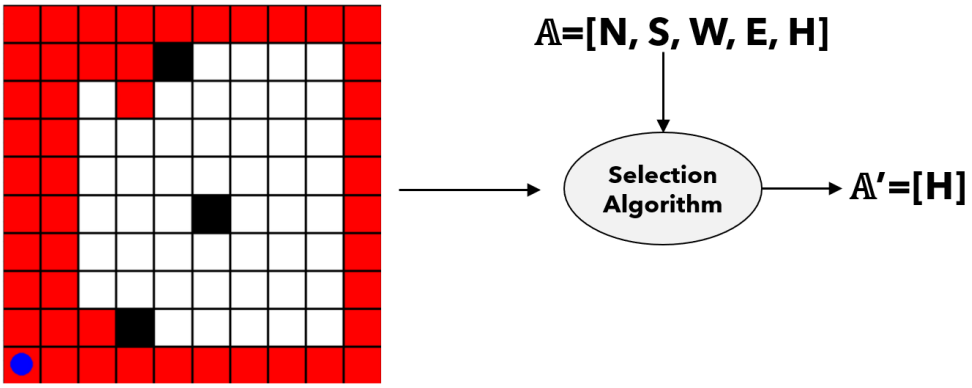


Figure 5.3: An example of the heuristic-based modified Policy.

reward function is formulated as follows:

$$R(s,a,s') = \begin{cases} \text{Size}^{-1} & \text{if a new cell is covered} \\ \text{Size}^{-1} & \text{if the } a_t \text{ is equal to } a_{t-1} \\ \text{Size}^{-1} & \text{if the agent finishes a row/column} \\ \text{Size}^{-1} & \text{if the agent is adjacent to a wall} \\ -10 \cdot \text{Size}^{-1} & \text{if there is a collision} \\ -5 \cdot \text{Size}^{-1} & \text{if an already covered cell is visited.} \end{cases}$$

All rewards have a scaling factor of $\text{Size}^{-1} = \frac{1}{N \times N}$, making it suitable for training in environments with different sizes. It is important to note that multiple events can happen in one time step, and all possible combinations should be considered. Furthermore, a positive reward can only be given if a new cell is covered. The rationale of the proposed reward function is to capture some characteristics of classical algorithms, such as Back-and-Forth and Wall-Following, by rewarding consecutive moves in the same direction, prioritizing finishing rows or columns, and staying close to walls. It discourages collisions and unnecessary moves.

5.4 Results

With the methodologies set and the environment implemented, the algorithm was trained and tested in a simulation environment. To make the simulation and training as general and realistic as possible, the conditions of the environment change every episode. The user can specify the maximum and minimum size of the environment and the same for the number of obstacles. These parameters are randomly generated at each episode respecting the user specifications. Furthermore, the agent starting position is also randomly generated.

5.4.1 Comparison Between Algorithms

To find the best TD Learning algorithm for this problem, four different algorithms were benchmarked in a simpler version of the task. These algorithms are: Q-Learning, SARSA, Double Q-Learning (DQL) [53], Watkins Q [42].

In the first phase, the task was to do CPP in a 6×6 environment, where the agent starting position would be limited to only the corners of the map and with zero obstacles. In the second phase, the environment had a single random obstacle in it. For both these tasks, the range of the agent sensor is $r = 3$ and the number of samples in memory is $K = 3$. The algorithms were trained for 100000 episodes in both cases, and the main metric of performance is the ratio between the number of used steps n_u and the number of free cells of the environment n_f ($\rho = \frac{n_u}{n_f}$), where the lower the value of the ratio, the better performance. This ratio has a lower bound equal to 1, which corresponds to an optimal solution in a convex environment. Notice, however, that this analysis does not apply in non-convex settings, which is the case when one or more obstacles are present. In this case, the optimal solution depends on the configuration of obstacles and starting position, and therefore may not be necessarily $\rho = 1$.

The results from both situations can be seen in Figure 5.4. In a simpler environment with no obstacles, near-optimal performance is achieved by all the algorithms besides SARSA, and the best-performing algorithm is Q-Learning, followed by Double-Q Learning. On the other hand, in the more complex environment, after 100000 episodes, the results are very similar to the previous situation, however, the performance is slightly worse, suggesting that more training is required. These results make it clear that the off-policy methods perform better in this specific task.

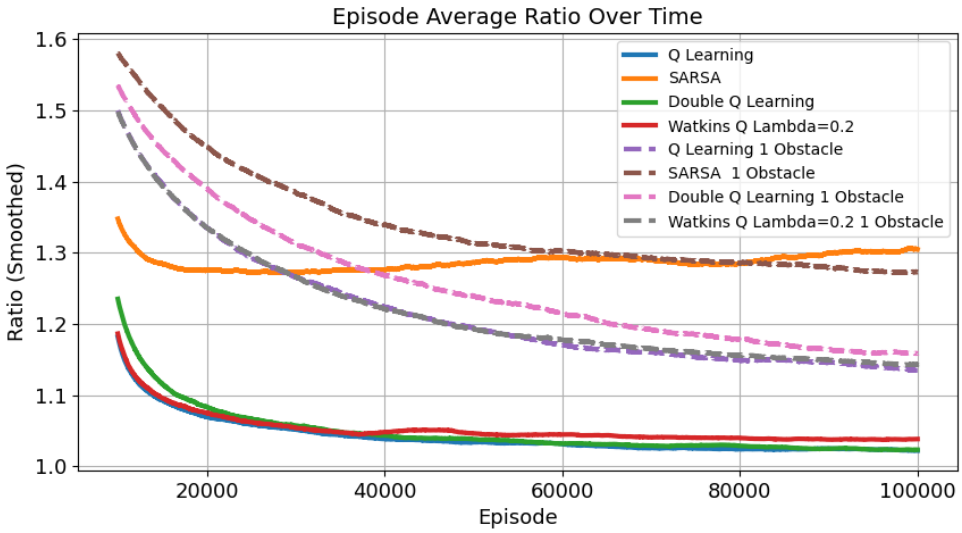


Figure 5.4: Average episode ratio in the first 100000 episodes of training in a 6×6 with the heuristic action

Table 5.1 displays the time that each algorithm took to complete the training phase. The simulations were done in a computer equipped with an AMD Ryzen™ 7 5800H processor, NVIDIA

Table 5.1: Training Times (s) for all used methods in both situations, with and without the usage of the heuristic action.

Method	*Heuristic, No Obstacles*	*Heuristic, One Obstacle*	*No Heuristic, No Obstacles*	*No Heuristic, One Obstacle*
Q-Learning	491.1	492.2	571.5	698.9
SARSA	615.8	529.7	688.9	791.7
DQL	477.8	498.4	1521.5	1993.6
Watkins Q	985.1	1448.0	2285.9	6934.1

GeForce RTX 3060 (Laptop, 120W), and DDR4 16GB RAM. By analyzing the results, it is noticeable the positive influence of using the heuristic action during training, which speeds up the training process, and in turn, will yield better results when considering the same number of training episodes. Focusing on the results of algorithms that use the heuristic action, it is possible to see that the training time is affected by two factors: the performance of the algorithm and the computational complexity.

The first becomes evident in the case of SARSA with no obstacles. Since it has by far the worst results, it took more time to complete each episode, as it took more steps to reach the terminal state. Furthermore, this issue will also make slower training in bigger maps, as even in optimal conditions, a bigger number of steps is necessary. As most algorithms use TD(0) for the value function estimation, the complexity is practically the same, leading to a similar time to execute each iteration. However, that is not the case with Watkins-Q, which uses eligibility traces. This mechanism is slower the more state-action exists in the environment and the number of different state-action pairs seen in the episode, making it scale poorly with obstacles and bigger map sizes. These issues would make prioritize both Q-Learning and Double Q-Learning since they have the best performance and time efficiency. However, in our experience with this problem, Double-Q Learning was more sensitive to changes in hyperparameters and would frequently not converge, making Q-Learning the method of choice for the following section.

5.4.2 Q-Learning for Coverage Path Planning

For evaluating the efficacy of the proposed algorithm in a broader range of scenarios, the Q-Learning-based algorithm was trained for 17.5 million episodes in randomly generated environments, with different starting positions, sizes between 4×4 and 10×10, and a number of obstacles in $[0,3]$. The agent hyperparameters are the sensor range of $r = 3$ and $K = 3$ samples in memory.

The trained algorithm was used to perform the task in environments with different settings. Each setting consists on the number of obstacles and the size of the environment, being that for each specification, there will be multiple environments. The algorithm was used to complete a set of 1000 coverage tasks in each setting, and it was evaluated by the average ratio of each set. The results can be observed in Figure 5.5.

These results clearly show the ability of the agent to generalize even in environments that have more than twice the size and obstacles of the ones in which it was trained. It obtained near-optimal

performance in environments with fewer obstacles, and achieved low ratios even in bigger map sizes, validating the design choices of the algorithm.

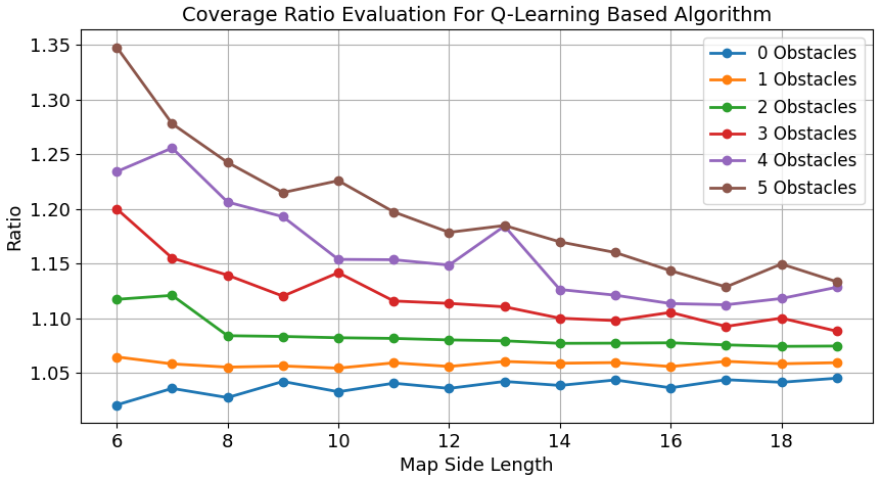


Figure 5.5: Obtained results for the evaluation of the Q-Learning CPP algorithm.

5.5 Final Considerations

In this chapter, a RL algorithm for Online Coverage Path Planning was formulated, designed, and evaluated. To address the large state space of this problem, an efficient and compact formulation of the problem as a Partially Observed Markov Decision Process was made, together with the development of a modified policy with a heuristic method and a limited action space, which accelerated the training process by optimizing and reducing the policy-space. The results validate the effectiveness of this approach, as the proposed algorithm demonstrates near-optimal performance in certain scenarios and exhibits generalizability to various map configurations, relying solely on sensor measurements. Notably, it stands out as one of the few methods that do not require an explicit map representation.

These encouraging results showcase the potential of value-based techniques and heuristics in solving navigation tasks like Coverage Path Planning. However, it is important to acknowledge that the tabular methods employed here have reached their performance limits, and further improvements would be unfeasible with this approach.

This work provides a solid foundation for the subsequent chapters, which will delve into Deep Reinforcement Learning approaches. A consistent theme will be the careful consideration of problem modeling and the utilization of heuristics and simplifications to manage the inherent complexity of robotic navigation. By building upon the insights gained here, the forthcoming chapters will explore more sophisticated techniques to tackle CPP and related challenges.

Chapter 6

Deep Reinforcement Learning for Single Agent Navigation

This chapter serves as the final stepping-stone before delving into the study of multi-agent methods. It presents a comprehensive case study on the utilization of Deep Reinforcement Learning (DRL) methods for various robotic navigation tasks. These tasks encompass point-to-point path planning, partial and complete coverage path planning, as well as active exploration.

The chapter will commence by introducing the problem and presenting the objectives 6.1. Subsequently, the problem is formulated in Section 6.2 and translated into a generic Partial Observed Markov Decision Process in Section 6.3, particularly emphasizing the problem's modeling. In Section 6.4 all the techniques employed in the learning algorithm's design will be thoroughly explained. Section 6.5 will present the obtained results, highlighting the algorithm's versatility, adaptability, and performance compared to other state-of-the-art algorithms. Finally, conclusions are drawn in Section 6.6.

6.1 Objectives and Challenges

This chapter will focus on the design of Deep Reinforcement Learning methods for robotic navigation. Contextualizing this work on the overall document, it is perhaps the most fundamental piece for the multi-agent approach in the next chapter and is, therefore, the foundation for everything related to Deep Reinforcement Learning.

Unlike the previous work and most of the literature, this chapter will take a different approach to coverage path planning and robotic navigation in general. Traditional path-planning solutions for autonomous robotic systems are often burdened by extensive, specialized software stacks designed for specific robots, configurations, and tasks. When a robot wants to perform different tasks, such as point-to-point path planning or coverage path planning, it will have to use different algorithms and develop complex interfaces between tasks. Moreover, the algorithm might only work on a specific robot and must be rewritten in a different use case.

Instead, this chapter will approach the robotic navigation tasks as a spectrum, illustrated in Figure 6.1. The intuition is that an intelligent robotic agent capable of doing a coverage path planning task should also be able to complete point-to-point path planning tasks. In a way, complete coverage path planning is a n-point coverage path planning task where n is the number of cells in the map. By analyzing the problem from this perspective, creating an agent that can extract semantic information from a map and perform any of the tasks should be possible. Furthermore, if it can extract this information from the map, it would also be trivial to do sensor-based coverage path planning as long as the information is consistently encoded.

Figure 6.1: The Navigation Spectrum Framework.

The contribution of this chapter will be a generic single-agent Reinforcement Learning algorithm that can be used in various navigation tasks, adapt to any 2D Grid Map, and be able to deal with different agent payload configurations. The main objectives of this chapter are as follows:

- Model Robotic Navigation as a generic Partial Observed Markov Decision Process with a similar structure for every task.
- Develop a method that can cope with different map sizes.
- Employ the state-of-the-art Deep Reinforcement Learning method, Rainbow DQN, to tackle Robotic Navigation.
- Train the algorithm in a manner that allows it to perform various tasks without the need for retraining.
- Investigate the versatility and generalizability of the developed algorithm, assessing its capacity to handle diverse scenarios and challenges in robotic navigation.
- Compare the results with other state-of-the-art algorithms.

6.2 Problem Statement

For a generic robotic navigation task, without loss of generality, consider a generic 2D square-shaped map $\mathscr{M}$ of maximum size $M \times M$. This map is obtained via approximate cellular decomposition where the cell size is equal to the robot size, as in most CPP approaches. The obstacles in the map are also cell-sized, and the number and position can be randomly generated. All the elements that are out-of-bounds are considered obstacles, and therefore, obstacles might be used to change the shape of the map without changing the general square shape.

The map has four types of cells: Non-Covered Cells (Point of Interest), Covered Cells, Obstacles, and Cells that were not yet observed. The never-seen cells are only relevant when the agent performs a sensor-based coverage path planning task or active exploration, and the agent must use its sensor to extract the cell information. This configuration enables the modeling of most navigation tasks. For instance, if all Non-Covered Cells, except for one, are changed to Covered Cells, a point-to-point problem can be produced where the only non-covered cell is the destination. This enables the agent to obtain the necessary information from the map representation. This basic configuration does not directly handle cases where the agent is expected to return to a home cell when the mission ends. Despite not being addressed in this document, a way to model this behavior can be to define the home cell as one that is simultaneously encoded as a Non-Covered Cell and a Covered Cell.

The agent is omnidirectional and can choose at each discretized time-step t. This action can move in any cardinal direction $\mathbb{A} = [North, South, East, West]$. The agent does not need to be adapted for a robot with other kinematics as long as a lower-level algorithm deals with the motion-planning problem, and the rewards can be tailored for this case, i.e. by penalizing turns. The agent has energy constraints represented by a battery level equal to the number of remaining actions an agent can take. The agent can have varied sensor payloads ranging from Camera, with a FOV of $r \times r$ cells, and can view over obstacles; 360 degrees LiDAR that functions similarly to the camera but can not view over obstacles.

The agent can start at any non-obstacle cell and does not have to finish the mission in a specific position. Therefore, the navigation problem can be seen as an optimization problem, where the objective is to minimize the number of discrete timesteps to finish covering all points of interest.

6.3 Partially Observed Markov Decision Process

The problem will be modeled as a Partially Observed Markov Decision Process (POMDP) for two main reasons. Firstly, the agent does not always have full information, for example, in scenarios where it performs sensor-based coverage path planning. Secondly, information compression or loss needs to occur when the map representation exceeds the maximum size acceptable by the network architecture.

The primary motivation behind this modeling choice is to ensure that the same POMDP framework can represent every navigation task. By adopting a unified POMDP formulation, a flexible and consistent representation is created, capable of effectively handling various scenarios and tasks, despite potential variations in the availability of information, map sizes, and objectives.

6.3.1 Action Space

The chosen action space for the agent can be described as:

$$\mathbb{A} = [North, South, West, East], \tag{6.1}$$

where at each timestep, the agent can choose a direction to move. It is assumed that the agent is omnidirectional, meaning that it can choose to move in any direction independently of the last action. While this assumption does not hold for every robot, more complex kinematics can be dealt with lower-level algorithms and will not be approached in this work, as it mainly focuses on the path-planning aspect of robotic navigation. The action space and the output layer of the neural network can be visualized in Figure 6.2.

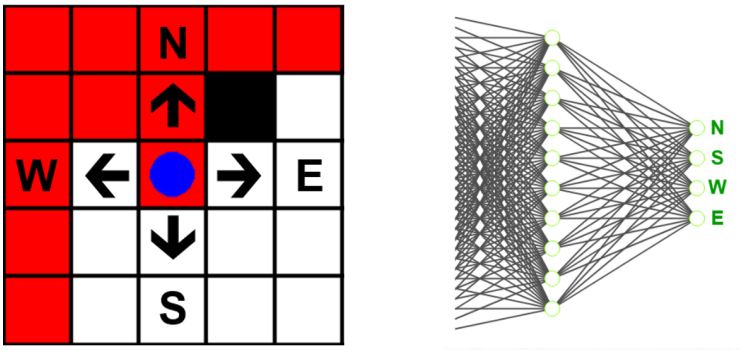


Figure 6.2: Example of the Action Space and Neural Network Output Layer.

On top of this action space, the agent is also equipped with a heuristic action. This heuristic action is the equivalent of moving toward the closest point of interest. If two points are equidistant from the agent, a cell is randomly chosen, effectively making this heuristic action a form of reinforced random walk.

Unlike the approach discussed in Chapter 5, the heuristic action is not explicitly integrated into the action space. The design choice in that work was rooted in the observation space lacking sufficient information to determine a non-heuristic action effectively, meaning that the agent could not learn a policy if the heuristic action was not part of the action space. In this chapter, the heuristic action selects one of the possible actions, and the agent and learning algorithm deal with it in the same manner as if it were chosen through the Neural Network output.

This design choice enables the agent to learn by imitating the reinforced random walk, which despite not being optimal, can be a better exploration policy than a purely random walk if used correctly. Moreover, including the heuristic ensures complete coverage guarantees for the agent, leading to a more understandable and reliable policy. Additionally, the heuristic action is used to address certain issues that arise from employing function approximation through Neural Networks in POMDPs. The nature of these issues and the incorporation of the heuristic in training and deployment policies will be discussed in subsequent sections

6.3.2 State and Observation Space

The State and Observation Space representation is one of the most fundamental pieces of the work. This section will describe the design choices from the ground up.

Considering most RL CPP approaches, the state representation is usually an N-channel $M \times M$ matrix where M is the map size. The number of channels is problem dependent. However, the

most common is three channels corresponding to the points of interest obstacles and agent position. This makes it so that the state representation can be seen as any vector $\mathbb{S} = \mathbb{B}^{3\times M\times M}$, where $\mathbb{B}$ is the Boolean domain $\{0,1\}$. In practice, the representation will be implemented as a tensor, which will be floating-point numbers instead of booleans.

Nonetheless, this approach is very simplistic and can be improved. The subsequent sections delve into a comprehensive analysis and description of the enhancements introduced within this work.

6.3.2.1 Channel Enconding

Since the objective of this work is to have the most generic algorithm possible, the basic state representation of the map was changed. This created a 4-channel 2D Matrix $\mathbb{S} = \mathbb{B}^{4\times M\times M}$ with the following channels encoding:

- **Channel 1:** Points of Interest.
- **Channel 2:** Points of Non-Interest.
- **Channel 3:** Obstacles.
- **Channel 4:** Agent Position.

Values within these channels are binary, either 0 or 1, corresponding to the information in a particular cell.

Considering the two first channels, while it is true that they could be compressed into just one channel, this approach is more flexible as it enables to implement more features by combining the values of the channels, for example, the cell where the agent needs to return in the end can be valued as a point of interest and point of non-interest at the same time. Furthermore, separating the two channels showed better performance when compared to the compressed version.

As for the agent position, it could also be removed. As seen in the next section, the map will be centered on the agent position, meaning there is no need for explicit position representation on the global frame. However, this channel was kept for two reasons: When this algorithm is expanded to a multi-agent setting, this channel must exist; The algorithm still showed better performance in a single-agent setting with the explicit position.

Finally, one can notice that there is no channel for non-seen cells in sensor-based CPP. Since this channel would only be used for this specific problem and obsolete for every other, it would be overhead in most situations. Consequently, non-seen cells are depicted by assigning zero values across all channels, signifying the absence of information within the corresponding cell. This design choice also synergizes with the separation of the first two channels. In the case of a singular channel where the value 1 signified points of interest and 0 if the cell was already visited, the representation where all channels are zero-valued would not represent non-seen cells.

Using the color scheme presented in Section 4.2.4, a visualization of the encoding of the channels can be seen in Figure 6.3. Once again, points of interest are marked as white cells, points

of non-interest as red, obstacles as black, the cells with no information are blue, and the agent is denoted with a blue circle.

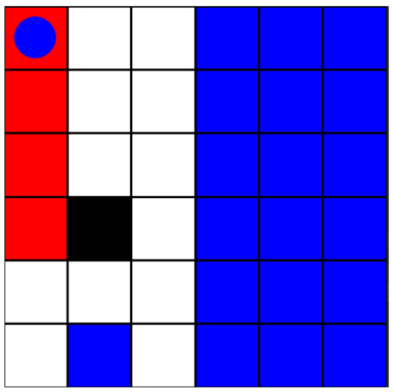


Figure 6.3: Visualization of all types of cells.

6.3.2.2 Map Centering

In work by Theile and Bayerlein [28, 83, 84], the authors introduce the concept of centering the map representation on the agent position. Intuitively, this representation can be seen as visualizing the map in the agent reference frame instead of the global or map reference frame. The authors showed that using this scheme significantly improves the performance of reinforcement learning algorithms because the neural network does not have to learn spatial relations in the global reference frame.

In practice, if a centered map is fed into a fully connected linear neural network, the neuron responsible for the cell where the agent is will always be the same, the same for the cell on its right or left, and so on. This enables a better function approximation generalization and is shown empirically to achieve better results.

Considering the map tensor $A \in \mathbb{R}^{4\times M\times M}$, one can apply a centering function to obtain the centered map tensor $B \in \mathbb{R}^{4\times M_c\times M_c}$, where $M_c = 2M-1$. Meaning that the tensor B can be defined by:

$$B = f_{center}(A), \tag{6.2}$$

where the function f_{center} can be seen as mapping from:

$$f_{center} : \mathbb{R}^{4\times M\times M} \rightarrow \mathbb{R}^{4\times M_c\times M_c} \tag{6.3}$$

Considering the position of the agent $p = [p_x, p_y]^T$, which is obtained from the fourth channel of the tensor A and the padding vector $x = [0,0,1,0]^T$, wherein all padded cells are encoded as obstacles, the elements of B can be determined through:

$$b_{i,j} = \begin{cases} a_{i+p_x-M+1,j+p_y-M+1}, & \text{if } M \leq i+p_x+1 < 2M \\ & \wedge\, M \leq j+p_y+1 < 2M \\ x, & \text{otherwise.} \end{cases} \tag{6.4}$$

This formulation implies that the representation size is limited by M_c. In this work, the edge size M_c of the centered map equals 41, meaning that the max map size would theoretically be $M = 21$. Nonetheless, to maintain a consistent representation across all maps, the edges are consistently treated as obstacles, thus establishing an effective usable size of $M_c = 39$ and a corresponding maximum usable map size of $M = 20$. Figure 6.4 visually depicts the process of centering a 16×16 map.

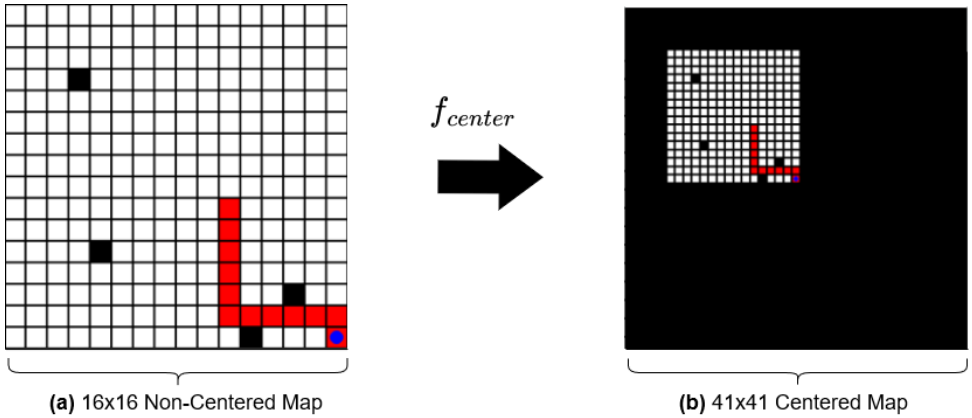


Figure 6.4: Example of the map centering function on a 16×16 Map.

6.3.2.3 Compressing Maps

One objective of this algorithm is to be able to generalize to any map. While 20×20 grid maps are larger than most cases studied in the existing literature, the algorithm's adaptability remains constrained by this limit. To address this limitation, the observation space must be expanded to enable the agent to recognize maps exceeding its visual scope. This enhancement has been realized through two different approaches.

The initial approach involves compressing all edges that would extend beyond bounds in the original 39×39 centered map, consolidating them into the last visible row or column. This compression process is executed by computing the average of all edge values. A pseudocode representation of this algorithmic enhancement can be found in Algorithm 11.

This approach ensures the algorithm can effectively adapt to varying map sizes beyond the original constraints. By dynamically compressing and extending edges in the tensor $M_p \in \mathbb{R}^{4\times41\times41}$, the agent gains the ability to comprehend maps larger than its immediate visual frame.

However, this method does have its limitations. Notably, when two connecting edges extend beyond bounds, the corresponding connecting cell becomes the average of both edges, resulting in an uneven information distribution compared to other cells. Moreover, the agent is not aware

Algorithm 11 Expansion of Outer Edges

Require: Map C of size $M_c \times M_c$, where $M_c > 39$
1: **function** COMPRESSDGES(C)
2: $D \leftarrow$ Extract the inner 39×39 array from C. ▷ Centered on the Same Cell
3: NorthEdges, SouthEdges, WestEdges, EastEdges $\leftarrow$ Sets of edges extending beyond bounds.
4: Average each set of edges to compute the compressed boundary values.
5: Replace outside edges in D with the averaged boundary values.
6: **return** D
7: **end function**

of how many edges were compressed, and the averaging means that the boolean representation is no longer viable, causing outside edges cells to have different encodings than inner cells. Furthermore, there is no comparable method in the literature, so further tuning might be necessary, for example, using more than one outside edge per direction.

To further enhance this representation, the observation space incorporates an additional tensor denoted as $oob \in \mathbb{R}^{4\times 2}$. In this tensor, each of the four cardinal directions is associated with two values: the count of compressed edges ne and the number of points of interest np in the corresponding direction, resulting in the tensor

$$oob = \begin{bmatrix} ne_N & np_N \\ ne_S & np_S \\ ne_W & np_W \\ ne_E & np_E \end{bmatrix} \tag{6.5}$$

The idea behind using this tensor and the compressed map together is that the agent can try to "reconstruct" the map using the additional information to grasp the current state better.

6.3.2.4 Deterministic Loops

One common problem in POMDPs and this formulation of the CPP problem is the existence of policies that lead to deterministic loops in the observation space [35]. This is usually caused by uncertainties and errors in the function approximation, and due to its nature, it is impossible to guarantee that there is no single scenario where a loop might happen.

An example of a deterministic loop can be seen in Figure 6.5. In this scenario, the agent first decides that the best action is to move South, and in the next time step, the best action is to move North. While this back-and-forth, most of the time, is undesirable at best, in this situation is catastrophic. Since the agent uses a deterministic policy during deployment, it will get stuck in this state forever. In this formulation, these observations will not change, as the map representations are the same, and therefore from the agent's perspective will be impossible to choose other actions.

While it might seem very unlikely to get stuck in scenarios like this, it is essential to note that these tasks can have time horizons of hundred or even thousand-time steps, and the difference in value between two actions can be very small, especially when the reward is far-away in the time horizon, like the situation in Figure 6.5. Furthermore, the neural network model used in this work

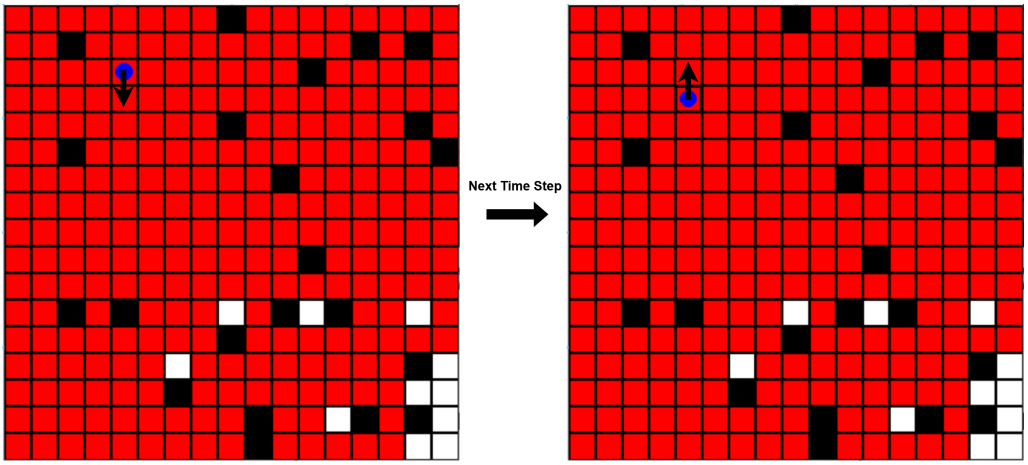


Figure 6.5: Example of a deterministic loop. The agent is stuck moving North and South with its policy.

has noisy layers. Consequently, the noise used for exploration can be harmful by creating more opportunities for this type of loop.

In the following Sections, additions to the observation space will be made specifically to reduce the likelihood of the existence of deterministic loops. Nonetheless, it is impossible to guarantee that the situation will ever occur, as it would be unfeasible to check every possible observation, and the neural network does not have any formal guarantee.

6.3.2.5 Episode Truncation and Battery Level

Another problem that goes hand-in-hand with the deterministic loop problem is the episode truncation problem. This POMDP formulation is a finite process, as there is always a terminal state. Nonetheless, during training, especially in the beginning and with deterministic loops, the agent might take too long to reach the final step. Especially in the case of deterministic looping, the episode may be infinite.

A natural solution to lengthy episodes is to truncate episodes, but if the agent does not have information to know when the episode will be truncated, then instabilities in training will appear due to this early truncation [91].

One way to mitigate the episode truncation problem is to make the agent time aware [91]. In this problem, the agent will have this capability by introducing a battery level $b \in \mathbb{Z}$ to the observation space Ω. This tensor represents the number of remaining time steps until episode truncation.

This variable has three different purposes for the overall problem. Foremost, it mitigates the effects of episode truncation, contributing to training stability. Moreover, by introducing a variable that changes with each time step, it partially mitigates the deterministic looping issue, where the observations are exactly the same. While it does not necessarily solve the problem in every situation, the presence of varying variable values does help prevent certain scenarios. Finally, while the problem is not formulated in the way that the agents should be concerned about their battery

status, this variable facilitates the formulation of energy-aware navigation problems without necessitating alterations to the observation and action space structure. A simple modification to the reward function is enough to formulate diverse energy-aware navigation scenarios using the same proposed framework. Hence, despite not directly contributing to the problem, this tensor b is a valuable tool for the generalization and robustness of the developed framework.

6.3.2.6 Incorporating the Last Action

One fundamental characteristic of POMDPs is the Markovian property. This property is revealed to be very useful for solving these processes, as it states that all the necessary information to solve the process is contained within the last state, i.e., all an algorithm needs to keep track of is the most recent state.

While this property is usually a blessing, it can be a curse in the context of robotic navigation. A trajectory encompasses the state's evolution throughout the state space and, in the context of path planning, signifies the route the agent will undertake. For effective path planning, maintaining a smooth trajectory is paramount, as abrupt changes in direction are expensive energy-wise. Moreover, from a rational perspective, excessive alterations yield an erratic and unpredictable course of action, which an intelligent agent should avoid.

For path planning, the Markovian property means that the agent can ignore all the trajectories that have been done before and focus on the current position and objective. This, once again, can be linked to the deterministic looping problem. When the agent repeatedly oscillates between moving upwards and downwards, it merely decides its next position based on its current location. Nevertheless, an intelligent agent that takes past movements into account can discern that its actions are contributing to loops within the observation space and, as a result, break free from such patterns. Naturally, if the agent is aware that its last action was a downward movement, it would typically refrain from moving upwards in the subsequent time step. This awareness of the preceding action can be seen as a form of odometry, a prevalent component in many robotic navigation frameworks.

This is the rationale behind the addition of the last action $la \in \mathbb{B}^5$ to the observation space Ω. This boolean vector is a one-hot encoding [1] of the action space $\mathbb{A}$, with an additional action included: "Wait." While not explicitly present in the action space, the "Wait" action is utilized when the environment does not receive a valid action in time or when initializing the la vector. This additional action also paves the way for future algorithm expansions that include this action in the action space, which can be useful in multi-agent scenarios.

This new addition to the observation space allows the agent to better understand its trajectory until that point, empowering the agent with more relevant information. This feature helps to mitigate the deterministic looping issue, and it was also observed that the agent naturally produced smoother paths by keeping its momentum. Furthermore, changing the reward function makes it

[1] One-hot encoding: https://pytorch.org/docs/stable/generated/torch.nn.functional.one_hot.html

possible to penalize direction changes, enabling more complex formulations of robotic navigation problems.

6.3.2.7 Frame Stacking

Two common approaches exist to deal with the increased complexity of POMDPs in Deep Reinforcement: Recurrent Neural Networks and Frame Stacking [92, 93]. The first is less explored and more computationally expensive, requiring a more complex network architecture, and therefore will not be used, despite being a possible future improvement. The latter is a common method, even part of the original DQN algorithm [51] to deal with the dynamics of the Atari Games environment.

As the name suggests, frame stacking consists of using the last K frames or states to constitute the state or observation fed to the neural network. In practice, considering the observation o_t generated at time step t, the full observation that is used in the algorithm is $O_t = [o_t, o_{t-1}, ..., o_{t-K}]$.

In this work, the frame stacking is extended to more than the frames, which in this case would be the map representation M_p is also done into the last action tensor l_a and to the out-of-bounds information tensor oob. This extension is rooted in the necessity of incorporating out-of-bounds details for full comprehension of the map, and including more than one of the last actions also enables the analysis of larger trajectories. However, the battery level b is not stacked, as sequential battery levels do not offer additional informative value.

The chosen K value for the number of stacked observations equals 3. While Deep RL can deal with bigger observation spaces when compared to tabular algorithms, by increasing K, the algorithm has to process more information, making for slower training with diminishing returns regarding the amount of available information. An example of the frame stacking on the map representation M_p is illustrated in Figure 6.6.

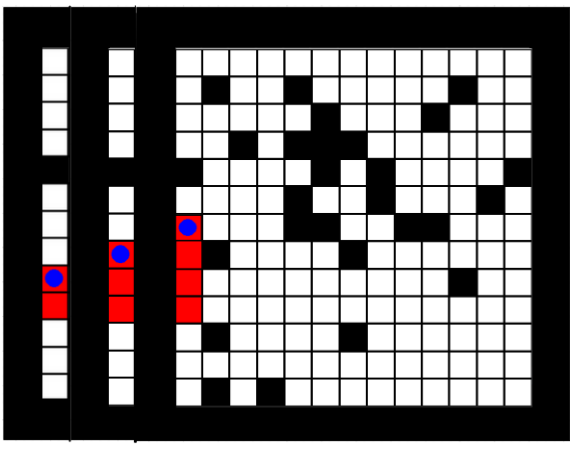


Figure 6.6: An example of Frame Stacking applied to the Map Representation M_p with K=3 Frames.

With all these design choices, the observation space can be defined as:

$$\Omega = \underbrace{\mathbb{R}^{3\times4\times41\times41}}_{\text{Map Representation } M_p} \times \underbrace{\mathbb{Z}^{3\times4\times2}}_{\text{Out of Bounds Information } oob} \times \underbrace{\mathbb{B}^{3\times5}}_{\text{Last Actions } l_a} \times \underbrace{\mathbb{Z}}_{\text{Battery Level } b} \tag{6.6}$$

In the implementation, all the elements of the observation space are treated as floating-number points, as it is needed to interface with the neural network. Furthermore, the dimensions are adjusted when interfacing with the network. The map representation's first two dimensions are flattened into a $\mathbb{R}^{12\times41\times41}$ tensor, so it is possible to use a 2D Convolutional layer, and the 2D tensors are flattened into a 1D tensor so it can be used in linear layers. With this, the observation space is fully defined and described.

6.3.3 Reward Function

The Reward Function is the last piece missing in the formulation of the POMDP. This function dictates the optimization problem that the agent will try to solve. This section will describe and build the reward function from the ground up, analyzing the problem and the objectives, formulating it as an optimization problem, constructing the most simple reward version possible, and then building the final solution.

6.3.3.1 Optimization Problem

Considering the problem formulation, the objective is to have the most general possible robotic navigation algorithm, which in this case will be modeled as a minimum time problem. Independently of the task at hand, the agent's goal will be to visit all the points of interest in the minimal time possible. Formulating the optimization problem, the objective function $J(T)$ can be defined as:

$$J(T) = \min T, \tag{6.7}$$

where T is the terminal time step. Given that Reinforcement Learning typically involves maximizing the expected reward, it becomes necessary to reformulate the optimization problem as follows:

$$J(T) = \max(-T). \tag{6.8}$$

A logical strategy for translating this optimization problem into the RL framework involves assigning a negative scalar reward at each time step, thereby defining the Reward Function as:

$$R(s,a,s') = r_{ts} = -1. \tag{6.9}$$

In this definition, r_{t_s} denotes the reward assigned at each time step. The selection of -1 as the reward value aligns with the widely recognized practice within Reinforcement Learning (RL) to maintain rewards within the $[-1,1]$ range.

6.3.3.2 Reward Shaping

While the presented reward function is appropriate and given sufficient training time, it is likely that the algorithm would eventually converge to a feasible solution, it still has a significant shortcoming. Due to the uniform reward across the entire episode, the agent encounters challenges in appropriately assigning credit. As all actions yield identical short-term expected rewards, the distinguishing factor becomes the moment of episode termination, marked by a non-explicit substantial reward. One thing that is important to note is that the episodes can be very lengthy, especially in the case of coverage path planning, potentially leading to unfinished tasks due to episode truncation. Consequently, the agent might struggle to grasp how to conclude episodes, and in the worst scenario, the algorithm might converge to a random walk. Even in cases where convergence to a reasonable solution occurs, it is likely to be relatively slow.

This issue is common in the Reinforcement Learning domain and is associated with problems in credit assignments and sparse rewards. One common way to mitigate the issue is to use Reward Shaping [94]. Considering an MDP (or POMDP) $\mathbf{M}$ with the reward function R, then there exists an identical MDP $\mathbf{M}'$ with a modified reward function $R' = R + F$, where despite the reward function change, both MDPs share the optimal policy. This property is called policy invariance, and it can be used to solve an identical MDP $\mathbf{M}'$ that will converge faster and use its policy in the original formulated MDP $\mathbb{M}$.

In this work, the family of potential-based reward shaping (PBRS) functions will be used. To further define this family, refer to Theorem 6.3.1 [94].

Theorem 6.3.1 *Consider a general MDP* $\mathbf{M} = \langle \mathbb{S}, \mathbb{A}, T, \gamma, R \rangle$ *and a shaping reward function* $F : \mathbb{S} \times \mathbb{A} \times \mathbb{S} \to \mathbb{R}$. *The function* F *is said to be a potential-based shaping function if there exists a real-value function* $\phi : \mathbb{S} \to \mathbb{R}$ *such that for all* $s \in \mathbb{S}, A \in \mathbb{A}, s' \in \mathbb{S}$

$$F(s,a,s') = \gamma\phi(s') - \phi(s) \tag{6.10}$$

In the context of potential-based shaping function F, *a necessary and sufficient condition is identified for ensuring consistency between the optimal policy learned in the modified MDP* $\mathbf{M}' = \langle \mathbb{S}, \mathbb{A}, T, \gamma, R + F \rangle$ *and the original MDP* $\mathbf{M}$, *as follows:*

- ***Sufficiency****: If* F *is a potential-based shaping function, then every optimal policy in* $\mathbf{M}'$ *will also be an optimal policy in* $\mathbf{M}$, *and vice-versa,*
- ***Necessity****: If* F *is not a potential-based shaping function, meaning that there exists no* ϕ *that satisfies equation* (6.10), *then there exists a transition function* T *and a reward function* R *such that no optimal policy in* $\mathbf{M}'$ *is optimal in* $\mathbf{M}$.

A very natural candidate for potential reward shaping is to use the number of points of interest $n_{poi}(s)$ of the current state as a potential function. In this context, one can define:

$$\phi(s) = -Kn_{poi}(s). \tag{6.11}$$

Here, $K \in \mathbb{R}^+$ symbolizes a generic positive scalar gain. Within this formulation, the shaping function $F(s,a,s')$ provides a positive reward each time the count of points of interest decreases. Considering that the agent's visits to points of interest lead to a decrease in this count, the function qualifies as potential-based, effectively guiding the agent toward the overall goal.

Given that the problem is framed around a single agent, and an agent can at most visit one cell per time-step, with $\gamma = 1$ one can rewrite the potential-based shaping function as:

$$\begin{aligned} F(s,a,s') &= \phi(s') - \phi(s) \\ &= -Kn_{poi}(s') + Kn_{poi}(s) \\ &= K[n_{poi}(s) - n_{poi}(s')] \\ &= \begin{cases} K, & \text{if a new cell was visited} \\ 0, & \text{otherwise.} \end{cases} \end{aligned} \tag{6.12}$$

In other words, the potential function equates to giving a scalar reward $r_{new} = K$ whenever a point of interest is visited, or zero if not.

One can also shape the reward by giving a penalty $r_{crash} = -1$ every time the agent crashes into an obstacle. Despite it not being a potential-based reward function, it does not change the optimal policy, as hitting an obstacle is doubly penalized: It does not move the agent to a better cell, effectively wasting a move and giving a negative reward. Consequently, it is evident that the optimal policy would inherently avoid collisions, even if the penalty to discourage collisions were absent.

Therefore, one can finally define the final Reward function $R'(s,a,s')$ as:

$$R'(s,a,s') = R + F + r_{crash} = r_{ts} + r_{new} + r_{crash} \tag{6.13}$$

where r_{ts} is the penalty that is given after every timestep, r_{new} is the scalar reward given every time the agent visits a new point of interest, and r_{crash} is the penalty given when a collision happens. It is important to note that since the reward is implemented based on events, whenever the event does not happen, the respective portion of the reward is zero-valued.

6.3.3.3 Size Invariant Value Function

After setting the core structure of the reward function in equation (6.13), it is important to analyze how its values impact the overall value function. Consider again the value function $v_\pi(s)$:

$$v_{\pi^*}(s) = R_{t+1} + \gamma v_{\pi^*}(s') = \mathbb{E}_{\pi^*} \sum_{t=0}^{T} [R_{t+1}] \tag{6.14}$$

where π^* is an optimal policy. Going back to the reward function, the only value left without explicit value was r_{new}. Considering $r_{ts} = -1$, assuming that the policy is optimal, the discount factor is $\gamma = 1$, and that at every new step, the agent visits a new point of interest [2], it is possible to deduce the following simplified expression for the value function:

$$v_{\pi^*}(s) = \mathbb{E}_{\pi^*} \sum_{t=0}^{T} [R_{t+1}] \approx \sum_{k=0}^{n_{poi(s)}} (r_{new} + r_{ts}) = n_{poi}(s)(-1 + r_{new}) \tag{6.15}$$

After analyzing this expression, one might be inclined to assign the value $r_{new} = 2$. This choice effectively constrains all non-collision transition rewards within the range of $[-1, 1]$, presenting a binary assessment of the immediate action's outcome. Furthermore, it makes for an easy way to evaluate every scenario, assuming that the mentioned assumptions hold, the value would be $v_\pi(s) = n_{poi}(s)$.

However, this formulation has two shortcomings: It is not size invariant, and evaluations can suffer in long time horizons. While the value function not being size invariant might seem irrelevant for most scenarios, it is critical in the case of sensor-based coverage path planning. If the agent does not know the size of the map, how can it accurately estimate the value of the given state? Whenever the agent explores new cells, the evaluation can oscillate significantly, and instead of being, in general, a negative monotonic function converging to zero, it can now increase in value. Usually, RL methods can deal with this uncertainty of the value function estimates. After all, it estimates the expected return. However, since this algorithm is aimed to be as general as possible, having different behavior for different scenarios is not desirable as it hinders generalization.

As for the second issue, consider that instead of a discount value $\gamma = 1$, it now equals $\gamma = 0.99$. Consider a situation where there are more than 100 points of interest. By incorporating the discount factor in equation (6.15) and solving:

$$v_\pi(s) \approx \sum_{k=0}^{\infty} \gamma^k (r_{new} + r_{ts}) = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1 - 0.99} = 100 \tag{6.16}$$

This deduction reveals that, independently of the number of points of interest, the value function would be capped at 100. This issue ties back to the deterministic looping problem. In very large time horizons, the evaluation will tend to be similar, creating scenarios where small uncertainties will lead to loops. While this problem can be solved by using a discount factor $\gamma = 1$, using this value usually causes instability in training, being desirable to use a smaller one.

However, these issues can be solved by setting r_{new} to a specific value. Consider $r_{new} = 1$, then by plugging it in equation (6.15), the value of every state would be 0. Furthermore, by analyzing equation (6.14) one can infer that if the reward is always 0 and the value of the next state is also 0, then the discount factor is ignored. This new reward function makes the value function size invariant, independent of the map size and number of points of interest. As long as an optimal

[2] When the environment has obstacles, there are scenarios where this assumption does not hold, even with the optimal policy. Therefore, the presented expression is just an approximation that holds when these assumptions are true. Nonetheless, if the algorithm is working, the agent will be functioning in these scenarios most of the time for coverage-type tasks, the most complex tasks approached in this document.

solution is possible, the value will be zero. Even more importantly, this also applies in scenarios where the agent does not have full information.

While the assumption that the agent will be operating in situations where an optimal solution with no overlap is possible might seem too strong, it is essential to note that the algorithm will converge in such a way that at some point in the trajectory, every move forward can be optimal with no overlap.

Another important thing to consider is that this value function formulation transforms the evaluation challenge into a task bearing a strong resemblance to classification, an area where neural networks excel. In Figure 6.7, it is possible to see three different states. The first two are the first and penultimate steps of two different 5×5 maps, being that in both scenarios, the evaluation given by the value function is zero. In the third example, it is also possible to see a zero-valued state, but in this situation, it is in a much larger 13×13 map with a shape that is more complex.

This is an interesting perspective on using neural networks for value function estimation. With this formulation, the neural network essentially operates like a classifier, generating a zero output when an action keeps the agent in a possible optimal trajectory. Furthermore, this formulation presents an enhanced generalization capacity. It not only adapts more effectively to scenarios involving partial information but also maintains consistent values across a multitude of scenarios. As a result, the network can learn to identify the common features in optimal scenarios and how to achieve such circumstances through the action space.

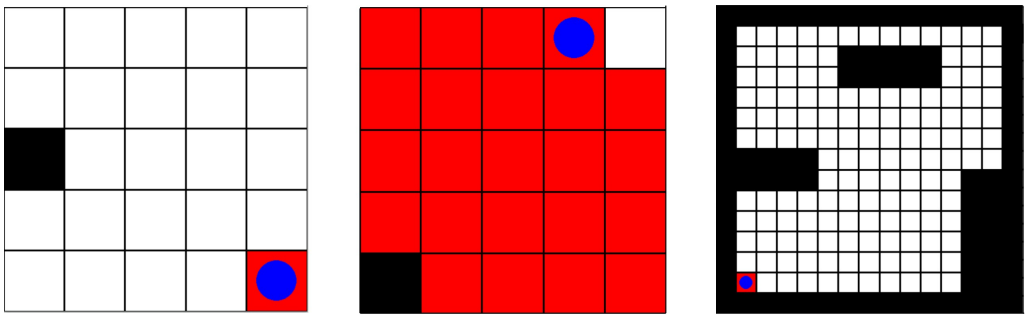


Figure 6.7: An example of three different states with the same evaluation. Since it is possible to complete all the scenarios with a path that does not overlap, then the value of the state will be zero.

6.4 Learning Algorithm

With the problem defined, the task now revolves around designing a solver for the Partial Observed Markov Decision Process. Due to the objectives and constraints caused by the POMDP, the chosen learning framework is Value-Based Deep Reinforcement Learning, and the method will be based on the Rainbow Deep Q Network [49]. This state-of-the-art algorithm is an improved version of the DQN, which is used in most RL approaches in the literature [28, 21]. It also gives all the necessary tools to deal with the large observation space and the discrete action space.

This section will begin by analyzing the Neural Network Architecture and the differences to the original Rainbow DQN and then will look into specific training algorithm adaptations tailored specifically for the robotic navigation problem.

6.4.1 Neural Network Architecture

To solve the formulated POMDP, the chosen RL method was based on the Rainbow DQN [49]. The neural network that was designed is similar to the one used in the canonical Rainbow DQN from the work of Hessel *et al*. For more details on the Rainbow DQN, see Section 2.3.5.

The neural network has two main components. The first consists in three 2D convolutional layers that are used to process the map representation $M_p \in \mathbb{R}^{3\times4\times41\times41}$, which must be reshaped to a tensor $M'_p \in \mathbb{R}^{12\times41\times41}$, and extract the relevant features. The second part receives the features from the convolutional layers and the hand-crafted features l_a, oob, b as scalars. This part consists in two streams of noisy linear layers [59], which outputs the value function $v(s)$ and the advantage function $A(s,a)$.

Regarding the configuration of the convolutional layers, the first convolution has 32 filters, a kernel size of 3, a stride of 1, and zero padding of 1. The choice of these parameters for the first layer is to transform the original input to a higher dimensional space without losing as much detail as possible. This reasoning justifies the choice of a small kernel, stride, and zero padding to keep the information on the edges. The next two convolutional layers are equal to each other, having a kernel size of 3, a stride of 2, and no zero padding. In these layers, the kernel size is still small, as there is a lot of information at the pixel level, unlike an Atari Game, for example. The stride of 2 in these layers facilitates downsampling, which, in turn, is essential for reducing the number of inputs downstream.

On the linear layer side, the output from the convolution is flattened, yielding a tensor $a \in \mathbb{R}^{5184}$. This tensor is concatenated with the hand-crafted feature tensor $c = [l_a, oob, b]$ comprising 40 elements. Among these, 1 corresponds to the battery $b \in \mathbb{R}$, 15 are from the last actions

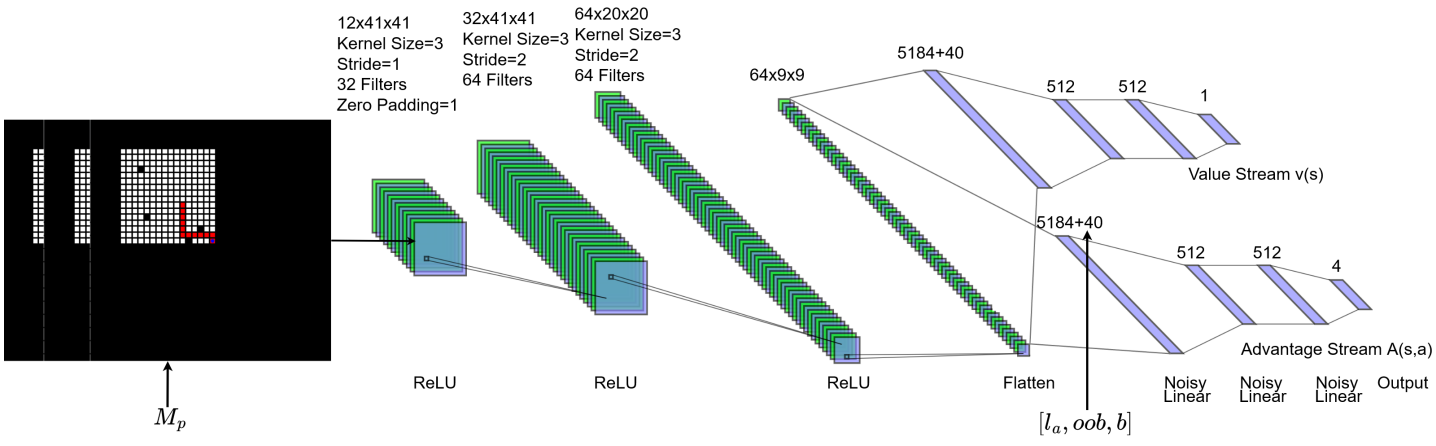


Figure 6.8: The designed Neural Network Architecture.

$l_a \in \mathbb{R}^{3\times5}$ and finally 24 steam from the out of bounds information $oob \in \mathbb{R}^{3\times4\times2}$. It is important to note that these tensors must also be flattened into 1D tensors before being concatenated. Unlike the canonical Rainbow, this architecture has two hidden layers of size 512, which was shown to perform better than a single hidden layer. To implement the dueling architecture, all the linear layers described are duplicated into two streams, one with a single neuron in the output layer corresponding to the value stream $v(s)$ and the other with the number of output neurons equal to the action space size $|\mathbb{A}| = 4$ which outputs the advantage function $A(s,a)$.

The full architecture can be visualized in Figure 6.8.

6.4.1.1 Extracting a Policy

When utilizing the dueling architecture for value function approximation, it is important to remember that one can derive a policy π that always chooses the optimal action $a^\star$ through the expression:

$$a^\star = \arg\max_a Q(s,a) = \arg\max_a \left[V(s) + A(s,a) - \frac{1}{|A|}\sum_{a'} A(s,a') \right]. \tag{6.17}$$

However, due to the introduction of noisy layers, the exploration mechanism is fully integrated into the parameter space, therefore not requiring a modified policy such as ε-greedy. In practice, the expression in equation (6.17) represents the policy both in training and deployment. However, this policy would be stochastic, which might be undesirable when deploying the algorithm. Therefore, consider the expression that defines the noisy layer:

$$y = (b + Wx) + (b_{noisy} \odot \varepsilon^b + (W_{noisy} \odot \varepsilon^w)x), \tag{6.18}$$

where b and W are the weights of the original network, $\odot$ denotes the element-wise product, b_{noisy} and W_{noisy} are the learned noise parameters, and ε^b and ε^w are independent random variables sampled from a factorized normal distribution.

Then, one can ignore the stochastic part of the equation by sampling zero-valued noise [3] ε^b and ε^W, effectively creating a deterministic policy for deployment. However, it is important to note that there might be a trade-off in using this technique. Since the policy is deterministic, the agent will be more susceptible to the deterministic looping phenomenon, and therefore, the prospect of better performance in some scenarios comes with this risk. Nonetheless, this technique will be used whenever the algorithm is deployed.

6.4.1.2 Soft Updating

The original Rainbow DQN has two networks: the value and the target network. In the original formulation, the value network parameters θ are updated regularly via gradient descent steps, and then at larger fixed intervals, the target network parameters θ^- are updated by entirely copying

[3] Or by sampling the noise once at the beginning of the episode. This would make the policy consistent throughout it but make it so that the results are not easily replicable.

all the parameters of the value network $\theta^- = \theta$. This hard update was developed to deal with the challenges of the deadly triad, especially the question pertaining to the value network trying to chase a moving target. Nevertheless, this update method also comes with the issue that the changes in the value function can be drastic, and the target network also has problems following the sudden change.

Another option named *soft updating* was proposed in other algorithms such as DDPG [43]. However, despite being more recent, the original Rainbow DQN [49] does not use this update scheme. Despite that fact, this work will use soft updating as it was shown to improve training stability.

The soft update is done by copying only part of the target network parameters. This process can be parameterized by the hyperparameter $\tau \in [0,1]$ resulting in the expression:

$$\theta^- = (1-\tau)\theta^- + \tau\theta, \tag{6.19}$$

where a value of 1 for τ would mean that the update would be the same as doing a hard update, and a value of 0 implies no update at all. This scheme is usually used with values of τ lower than 0.05, and the update frequency can be synchronized with each gradient step or alternately every few gradient steps.

6.4.1.3 Simplifying the Rainbow

This study introduces additional modifications to the conventional Rainbow DQN (see Section 2.3.5) to adapt it more effectively to the demands of robotic navigation tasks.

First, one might notice that the value and action streams in Figure 6.8 do not have adequate dimensions for the categorical reinforcement learning framework. In the original work [60], the authors present a version for a categorical distribution with 51 atoms, being that the algorithm is normally referenced as C51 for this reason. In this work, the categorical framework will not be used, and this design choice is sustained with the following reasoning:

1. **Increased Computational Complexity:** The categorical distribution is significantly more computationally intensive, leading to a training speed that is approximately 75% of the original DQN [60].

2. **Lack of Non-Atari examples:** The Categorical DQN has three fundamental hyperparameters v_{min}, v_{max} and N_{atoms}. The first two are related to the possible range of values the evaluation will take, usually between -10 and 10 for the Atari Learning Environment (ALE). However, this would not translate well as the maximum value in the developed environment would be 0 at best, and the minimum value is not defined and is problem and map-specific. Furthermore, if the values in this problem are mostly around 0 then the distribution probability space is poorly distributed and utilized. As for the atoms, as far as the author knows, most of the studies are done for the ALE, and therefore, 51 atoms might just not be suited for the robotic navigation task.

3. **Ablation Test Results** The results from the original study of the Rainbow DQN [49] show that there are no significant improvements from using the categorical distribution until around 30 Million training steps. This is once again only applicable to the ALE, and in this study, the used training steps will be less than in the Rainbow DQN paper. Furthermore, it does not present results on how the categorical distribution deals with transfer and curriculum learning since the estimated distribution can change drastically depending on the task.

4. **Empirical Results on Robotic Navigation Task:** In this work, the starting point was using the full rainbow architecture. Using the categorical distribution without significant changes did not prove to be of any benefit, yielding, at best, similar results, while increasing training times and complexity in fine-tuning the algorithm.

This choice of not using the categorical distribution is not the only one in the literature. In [95], the authors study an optimized and data-efficient rainbow DQN, where similar arguments also support this design choice.

The last modification is not to use the $n-$step returns. This augmentation to the DQN has the advantage of faster convergence and possible mitigation of the deadly triad. However, using $n > 1$ increases the variance in returns at the cost of reducing the bias.

In the context of robotic navigation, it is undesirable to have a higher variance. Trajectories can be very sensitive to the choice of movements, and a single "bad" move can disrupt a trajectory. Incorporating more than one move in the return complicates credit assignment and makes learning a generalizable value function harder. Moreover, introducing bias into the process is not inherently undesirable. In fact, it can reflect a rational choice when strategizing a path. When testing, using $n = 1$ yielded better results and stability during training. Therefore, the design choice of not considering multi-step returns came naturally. Nonetheless, $n-$step returns are implemented and can be used and experimented with by simply choosing a different value for the hyperparameter n.

6.4.1.4 Hyperparameters

To finish the analysis and description of the Neural Network and general learning framework, it is necessary to present the algorithm's hyperparameters. These hyperparameters and their values can be seen in Table 6.1

The hyperparameters were manually tuned starting from the canonical Rainbow parameters. It is important to note that the replay period K has multiple roles in the algorithm. In the most generic form, it is the number of time steps between every time the experience replay buffer is sampled and gradient descent steps are performed. This algorithm synchronizes the replay period with two other operations: the soft update and re-sampling noise in the noisy layers. While this

[5]Pytorch's ADAM implementation: https://pytorch.org/docs/stable/generated/torch.optim.Adam.html

[5]Pytorch's SmoothL1Loss implementation:https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html#torch.nn.SmoothL1Loss

Table 6.1: Deep Reinforcement Learning Single-Agent Algorithm Hyperparameters.

Parameter	**Value**
Minimum Steps Before Learning T_{start}	5000 Frames
Soft Update Factor τ	0.04
Discount Factor γ	0.999
Batch size B	32
Optimizer	ADAM [4]
Loss Function	Smooth L1 Loss[5]
ADAM Learning Rate	0.00005
ADAM ε	0.000156
Maximum Gradient L2 Norm	2.5
Priority exponent α	0.5
Priority correction β	$0.4 \rightarrow 1$
Noisy Net Initial std deviation σ	0.5
Memory size	1 Million Transitions
Learning Period K	4
Number of Stacked Frames	3
Multi-step return length n	1

could be done at every step, it does not necessarily provide better outcomes and comes with a significant training slowdown.

Furthermore, the priority correction β is annealed from $0.4 \rightarrow 1$. This is done linearly with an increase inversely proportional to the number of training steps in such a way that the last step has a β value equal to 1, following the original PER implementation [57]. Most other parameters were manually tuned at the expectation of the ADAM ε, which follows the recommendation of $\varepsilon = 0.005/B$ presented in [96]. Another thing to note is that a memory size of 1 million transitions requires at least 16GB of RAM. While, in general, bigger buffers showed better performance, any size between 500 thousand and 2 million transitions had identical results. For more details on the hyperparameters of the Rainbow DQN, see Section 2.3.4.

6.4.2 Training Algorithm

With the problem formulated and the network architecture set, the last step is to define the training algorithm and some modifications to the "vanilla" approach. This subsection will analyze all the modifications to the general model-free experience replay-based learning approach. Its main objectives are to achieve faster training and convergence, more stability, and overall better results.

6.4.2.1 Partial Episode Bootstrapping

One problem that was already analyzed in this chapter was episode truncation. On top of the modification to the observation space to make the agent time-aware, there are also refinements that can be done to the training algorithm itself, with the objective of having tools for both dealing

with and preventing episode truncation. Consider the expression for the TD error:

$$\delta(\theta) = R_t + (1-\eta)\gamma \max_{a'} Q_{\theta^-}(s',a') - Q_\theta(s,a), \tag{6.20}$$

where $\eta \in \mathbb{B}$ is a boolean variable that takes the value 1 if the next state s' is terminal and 0 if not. The usage of this flag is essential to implement Temporal Difference Learning algorithms, as in situations where the next state is terminal, its value should not be considered (is zero).

One thing that can get lost is that truncated episodes get treated the same way as terminated episodes when they are considerably different. If an episode is terminated, it means that the agent can no longer receive more reward, either positive or negative, and therefore will adapt to that fact. While in truncated episodes, what happens in practice is the same, the agent can create "bad habits," as the episode should not actually end in that timestep, and therefore, future rewards must be considered.

The simplest way to deal with this problem is by using Partial Episode Bootstrapping (PEB) [91]. Basically, the η will be zero-valued if the episode is truncated instead of terminated. This enables the agent to know that the episode would still continue to that point, and remove a source of instability from the training algorithm.

6.4.2.2 Partial Resetting

Staying on the topic of truncated episodes, one might question why these episodes happen, and what their importance in the scheme of the learning algorithm is.

One rather easy thing to infer from a truncated episode is that the agent was not able to complete the task successfully, therefore, the difficulty level of the episode was too high for the agent. Even if the task is one that the agent should be able to complete, the episode might contain small details that the agent has not learned or is struggling to grasp.

If not all episodes are equal, then it is fair to interpret that the agent will not gain the same amount of "knowledge" from every scenario, and in fact, might even learn different aspects from different scenarios. This is somewhat already implied in mechanisms such as Prioritized Experience Replay, which feed the learning algorithm samples that are deemed to have more information.

Consider Figure 6.9 in which it is possible to see the beginning of two very different episodes. It is important to remember that the agent should be able to perform both, and therefore some skills and competencies can be learned from both scenarios. The one on the left, while clearly easier, can be used to learn optimal coverage patterns such as back-and-forth or spiral and is without doubt useful to begin training with. In the other example, there can be a lot more emphasis on obstacle avoidance and making compromises in the chosen path, as one free of overlap is impossible. While undoubtedly more difficult to perform on due to the size and number of obstacles, one agent that trains exclusively on this map would most likely struggle to complete the left scenario optimally.

Therefore, if an agent did not finish a task, it could be of interest to give it a second opportunity to do it, as there is some detail in the map that the agent still is yet to learn. With this rationale, whenever an episode is truncated, the subsequent episode commences from where the previous

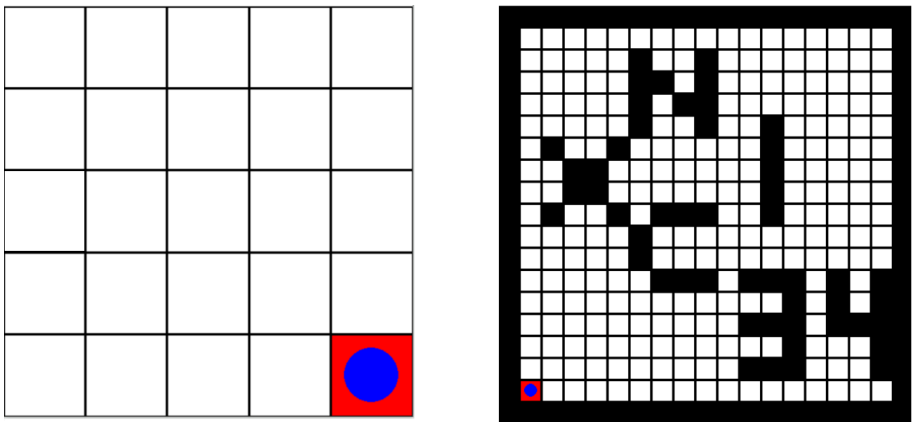


Figure 6.9: Example of two scenarios with different difficulty levels.

one left off. The episode is completed by always selecting the action that moves to the closest point of interest by using Dijkstra's algorithm heuristic. This enables the agent to see a possible solution to the problem it is facing and learn through imitation. It also guarantees that the episode will be finished.

6.4.2.3 Saving Challenging Tasks

Expanding upon the notion of episodes presenting distinct challenges, the partial resetting mechanism can be augmented and made powerful. The reason why this is desirable is due to the stochastic nature of randomly generated scenarios, where intricate obstacle configurations, such as narrow alleyways, rarely appear naturally. Consequently, it becomes advantageous to conserve scenarios for reuse beyond their initial occurrence.

Given the scenario's intricacy and the agent's inability to resolve it, the training algorithm should guarantee that there is at least a point during training in which the agent can completely solve it without external help. Failure to achieve this outcome would compromise the fundamental premise of a generalizable algorithm, even within the training phase.

To address this, an adaptive episode generation algorithm was developed. Every time a new episode commences, it is archived in a buffer as a potential "interesting" episode. Should the agent successfully complete the episode, the algorithm deduces the task can be solvable and subsequently discards the episode. If the episode is not discarded, then it stays in the buffer. Whenever the environment receives a request to generate a new episode, the buffer will have a 50% chance of being randomly sampled. This algorithmic scheme is graphically represented in Figure 6.10.

The buffer is deliberately only sampled 50% of the time to strike a balance between exposing the agent to newly generated episodes and the episodes that were deemed challenging to the agent. This design choice was made to prevent situations where overfitting and bad solutions keep the agent stuck in a limited set of episodes, being especially critical at the beginning of training.

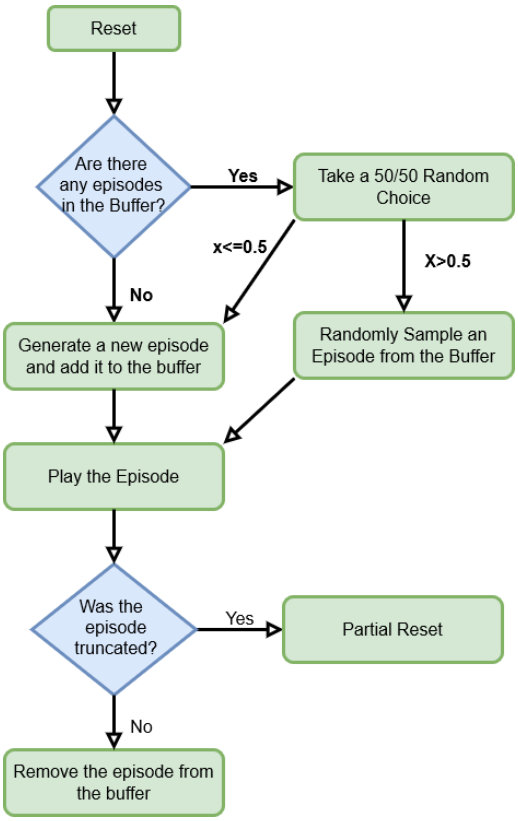


Figure 6.10: Flowchart of the adaptive episode generation algorithm.

6.4.2.4 Imitation Learning

The DQN family of algorithms uses the experience replay mechanism to store transitions, which are subsequently utilized to update parameters via gradient descent. The usage of experience replay buffers enables an interesting perspective on off-policy learning that online learning algorithms such as PPO do not enable. By sampling a replay buffer, the agent selects transitions taken by very different policies. For example, a transition made in time step $t = 0$ will have a widely different policy compared to one taken in time step $t = 1000000$. Furthermore, the agent can not distinguish between a transition generated by itself or another agent or pseudo-agent and, therefore, can learn by "imitation."

With this in mind, if the experience replay is filled by transitions made by another agent or algorithm with expert information, the convergence can be much faster, even if the policy it imitates is not optimal. In this work, a reinforced random walk policy was introduced by using Dijkstra's algorithm to select an action that moves toward the closest point of interest. While not optimal, it will significantly outperform a purely random walk policy when "teaching" the agent what a point of interest is and its shortest path. Of course, the agent can not solely learn by using this policy. Otherwise, it would completely copy it without improving. Nonetheless, augmenting the learning algorithm with this policy will significantly improve its performance.

So, considering this imitation learning paradigm, the heuristic can be used in the following scenarios:

1. Whenever an episode is generated in the first 5000 steps, where the agent is not sampling the buffer.

2. To complete an episode that was partially reset. This enables showing the agent a possible solution.

3. Whenever the agent has not visited a point of interest in more than M timesteps, where M is the map edge size.

4. Whenever the agent has not visited a point of interest and not moved more than two cells in the last five timesteps.

Whenever a condition to take a heuristic action is met, there is a 90% chance that it will be taken. Once again, this stochastic approach is to prevent overfitting to the heuristic policy. Whenever conditions 1 or 2 are met, the whole episode can be completed using the heuristic policy. The other conditions are reset whenever the agent visits a point of interest. Conditions 3 and 4 are mostly done to prevent the deterministic looping problem and mitigate its effects.

6.4.2.5 Generating Uncommon Patters

Overfitting is a very common problem in Machine Learning. When applied to RL can be seen as an agent that overspecializes on a specific task, losing the ability to complete other similar tasks. Considering the robotic navigation problem, more precisely in Coverage Path Planning the agent is notably prone to overfitting.

Consider a hypothetical scenario where an agent consistently produces optimal solutions for complete coverage path planning. This agent is the desirable result of the training process. However, it might have some concealed issues. If an agent is consistently doing optimal solutions, then usually, the map representations create consistent identical patterns where there are solid areas of connected points of interest and solid areas of already visited cells. It is natural, as optimal solutions look similar, even if the map configuration is not necessarily the same. However, consider a scenario where the agent fails to achieve an optimal solution and arrives in a situation where the map is in a non-common pattern. In this case, as the agent was trained consistently in regular patterns, it might fail to achieve the task, even if it is trivial, resulting in poor solutions or deterministic loops.

Furthermore, from the perspective of having the most general robotic navigation algorithm, complete coverage path planning can be seen as the most complex task. Nonetheless, the agent should also be able to complete N-point coverage path planning or point-to-point. So, the agent should not overfit any specific task, and the training scenarios should be as general as possible. After all, the competencies learned in different tasks are generally transferable.

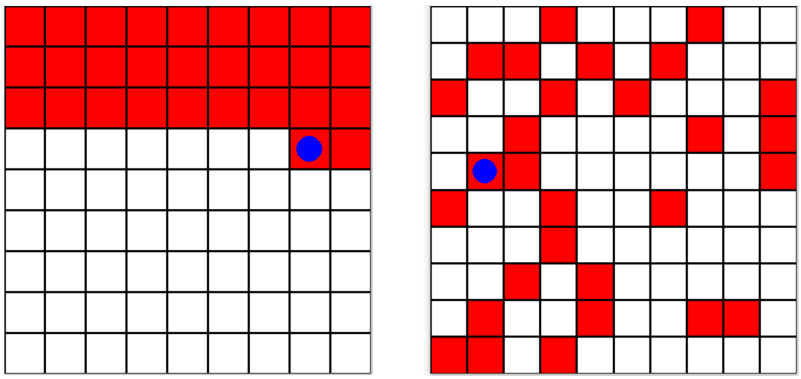


Figure 6.11: On the left, an example of a commonly agent-generated pattern, and on the right, an example of a randomly generated pattern.

For a more comprehensive understanding, refer to Figure 6.11, which presents two scenarios that exemplify the concept of patterns in this context. The scenario on the left corresponds to the algorithm solving a complete coverage path planning task on a straightforward map. It is clear that the agent performed some back-and-forth movement to create a solid area of visited points that are situated in the northern part of the map. This pattern is rather obvious and straightforward, and if the agent keeps it, then the yield solution will be optimal. The issue arises when, due to the proficiency of the algorithm, most of the encountered patterns are similar to this case, and the agent overfits.

To solve this issue, the implemented solution is that every time a training episode is generated, there is a 50% chance that the map will be generated with up to two-thirds of already visited cells, effectively creating a n-point coverage path planning problem. This also creates patterns that are impossible to be generated by the agent motion, as depicted in the scenario on the right in Figure 6.11.

In practice, empirical results showed that agents trained on maps with uncommon patterns were more capable of generalizing and less susceptible to deterministic loops and poor solutions in cluttered environments without losing general performance in most situations.

6.4.2.6 Curriculum and Transfer Learning

To finish the description of the training algorithm, the final set of tools will be presented. This algorithm will leverage the power of curriculum and transfer learning to improve the efficiency and speed of training.

As discussed previously, scenarios have different difficulties depending on the map's parameters. Due to the environment's sandbox nature, it is possible to tailor the environment to the needs of the training algorithm. Therefore, a curriculum training approach to robotic navigation should start with the simplest tasks and build up to the most challenging ones. In this process, it is important not to discard simpler tasks to avoid overfitting and aim for a generic algorithm.

With these considerations, training the initial model involves navigating six distinct environment configurations within the coverage path planning framework, utilizing full information. Detailed environment configurations are provided in Table 6.2.

Table 6.2: Environment Configurations for Curriculum Training on the Single-Agent Full Information Setting.

Parameter	env1	env2	env3	env4	env5	env6
Training Steps	1×10^6	2×10^6	3×10^6	5×10^6	1×10^7	1×10^7
Max Size	5	10	15	20	25	30
Min Size	4	5	10	10	15	5
Number Obstacles	5	10	20	30	40	50
Starting Position	Random	Random	Random	Random	Random	Random
Random Coverage	True	True	True	True	True	True
Sensor	FI [6]	FI	FI	FI	FI	FI
Sensor Range	N/A	N/A	N/A	N/A	N/A	N/A

And a generic training algorithm can be analyzed in algorithm 12.

Algorithm 12 Curriculum Learning CPP Training

Require: Environment configurations E, replay period K, memory size N, PER exponents β, Learning Start Step T_{start}

Initialize replay memory $\mathscr{D}$, PER exponent β and its increment Δ_β,online network θ and target network θ^-
for environment e in E **do**
    Initialize environment e, training budget T
    Observe O_0 and choose A_0
    for $t = 1$ to T **do**
        Use online network θ to choose next action A_t
        Take Action A_t, observe O_{t+1}, R_{t+1} and store the transition
        if $t > T_{\text{start}}$ and $t\%K == 0$ **then**
            Reset Noisy Linear Weights
            Sample k transitions from the replay buffer
            Compute loss
            Perform Gradient Descent to update online network weights θ
            Soft Update the target network weights θ^-
        end if
        $\beta \leftarrow \beta + \Delta_\beta$
        if S_t is Terminal **then**
            Reset the environment
        end if
    end for
end for

After using algorithm 12 to train on the set of environments presented in Table 6.2, the final result will be an agent model for a generic robotic navigation agent with full information. In order

[6] FI stands for Full Information.

to improve the versatility of the agent, it is still necessary to train the agent in scenarios with partial information. For that purpose, the configurations of the original training are adapted and can be seen in Table 6.3.

Table 6.3: Environment Configurations for Curriculum Training on the Single-Agent Camera Settings.

Parameter	env1	env2	env3	env4	env5	env6
Training Steps	1×10^6	2×10^6	3×10^6	5×10^6	1×10^7	1×10^7
Max Size	5	10	15	20	25	30
Min Size	4	5	10	10	15	5
Number Obstacles	5	10	20	30	40	50
Starting Position	Random	Random	Random	Random	Random	Random
Random Coverage	False	False	False	False	False	False
Sensor	Camera	Camera	Camera	Camera	Camera	Camera
Sensor Range	3	3	3	4	4	3

The initially trained model is used as a starting point when training in the environment scenarios from Table 6.3, in a technique usually referred to as transfer learning. When this second stage of training is finished, the agent is fully trained and ready to be deployed in any of the proposed robotic navigation tasks.

6.5 Results

Following the delineation of the problem, the modeling of the Partially Observable Markov Decision Process, and the specification of the learning algorithm, a comprehensive assessment of the solution's performance is now possible. The performance will be assessed by testing the algorithm on the different sets of tasks with different variations on the algorithmic approach. The following subsection outlines the methodology employed for analyzing the results. Subsequently, the subsequent subsection explores the analysis of Coverage Path Planning under full information. This is followed by an examination of sensor-based coverage and point-to-point tasks. Lastly, a comparison is drawn between the algorithm's results and those achieved by state-of-the-art approaches.

6.5.1 Methodology

Before delving into the results, it is crucial to establish concrete evaluation guidelines for the algorithm in the contexts of Coverage Path Planning, sensor-based or not, as well as point-to-point path planning.

6.5.1.1 Coverage Path Planning

In the evaluations of Coverage Path Planning, the algorithm's performance will be examined within scenarios of full coverage path planning across a range of maps. This set consists of 25 subsets organized by map size. Each subset contains 100 randomly generated scenarios featuring

map sizes ranging from 5 to 30. In each map, the agent's starting position is randomized, and the number of obstacles ranges between 0% and 10% of the total cell count. The main metric for measuring the performance of the algorithm in these scenarios is the path overlap O_v, which can be defined as:

$$O_v = \frac{n_r}{n_t - n_r}, \tag{6.21}$$

where the value n_r corresponds to the number of repeated steps, or in other words, steps resulting in visiting cells that were already covered, and n_t is the total step count. This metric is useful for complete coverage path planning, where for an optimal path in a convex environment, the overlap value would be $O_v = 0$, which means that every single step resulted in visiting a point of interest. However, when dealing with non-convex environments due to the presence of obstacles, the optimal O_v value can be different from zero. Nevertheless, a useful rule of thumb is that values close to zero can be considered optimal or, at the very least, near-optimal.

If there were a perfect metric that could guarantee that a solution was optimal, then it would render this algorithm pointless, given that a means to determine the optimal path for every scenario would be available. Therefore, despite flawed, this metric is one of the best available and will be specifically because it is also used in other relevant works in the literature, more specifically the ones done by LG Electronics Advanced AI Team [21, 81, 82].

6.5.1.2 Point-to-Point Path Planning

In the context of point-to-point path planning, the algorithm is subjected to scenarios involving a solitary point of interest. The map size will be randomly determined within the range of [5,30], the number of obstacles will again range up to 10% of the total map size, and both the agent and point of interest positions will be randomized. An example of a point-to-point planning scenario can be visualized in Figure 6.12.

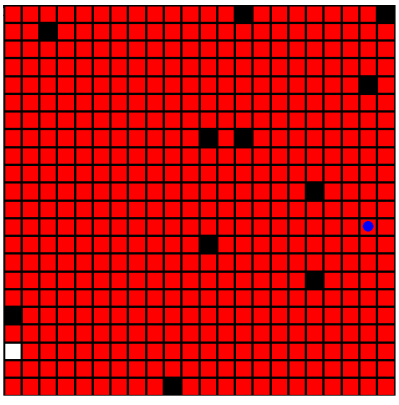


Figure 6.12: Example of a point-to-point path planning scenario.

With these considerations in mind, the performance will be gauged using the metric:

$$\Delta_s = n_D - n_{RL} \tag{6.22}$$

where n_D signifies the number of steps in the optimal solution provided by Dijkstra's algorithm, and n_{RL} signifies the steps in the solution generated by the RL algorithm. While the solution given by Dijkstra's algorithm will always be optimal, therefore making the RL algorithm not inherently necessary, the purpose of the analysis is to show the ability of the RL algorithm in general robotic navigation situations and its capacity to extract the necessary information from the observation representation in any generic situation.

6.5.2 Full Information Coverage Path Planning

This subsection will present the results obtained by the algorithm in a generic coverage path planning setting. The agent model was trained as described in Section 6.4.2.6, and the evaluation will be done in the rules defined by the previous Section 6.5.1.1. Firstly, the objective is to analyze how the algorithm generalizes to specific different map sizes, followed by a general evaluation over the whole size spectrum. The final results will be a comprehensive analysis of the outliers.

6.5.2.1 Distribution of Results and Heuristic Usage

In this section, the algorithm performance will be analyzed within specific subsets. These subsets correspond to maps with sizes $\{5, 10, 15, 20, 25, 30\}$. The objective of this study is to isolate these cases, enabling a more detailed analysis of how the solution's behavior evolves with varying map sizes. Particularly, the distribution of results in each scenario will receive special attention, which will be visualized using Box Plots.

Another thing that will be analyzed and introduced in this section is the reasoning behind utilizing Dijkstra's heuristic with the rules that were defined for training in Section 6.4.2.4.

Refer to Figure 6.13, where the results for the RL algorithm with the heuristic can be visualized. In all the sets of maps, the mean results are near-optimal, always achieving overlap values inferior to 0.05 or 5%. These results are consistent with the evolution of size and, in general, have a slight downward trend as the map gets larger. Moreover, the Inter-Quartile Range (IQR) generally maintains an upper bound below 0.1. Within 600 episodes, there are only 14 maps that yield results exceeding 1.5 times the IQR, warranting their classification as outliers. This implies that roughly 98.7% of the results do not fall into the outlier category. Even in instances where outliers do arise, the results remain satisfactory. For maps larger than 10×10, the outliers are rarer and are located barely outside the upper limit, being not larger than 0.125 overlap, which is still a reasonable result, especially considering that there are difficult maps in the dataset. Smaller maps have larger outliers. However, it is important to consider that smaller maps are susceptible to having higher overlap values. In a 5×5 map, a solution with only five extra steps already scores an overlap of 0.25, which in this case was the maximum value of overlap observed.

Now consider Figure 6.14 that displays the results for the case that the agent never uses the heuristic. It is important to note in Figure 6.14a that there are clearly more outliers, and their values are much larger when compared to the previous case. It is also worth highlighting that 6 maps were not completed before episode truncation and, therefore, are not included in the graph.

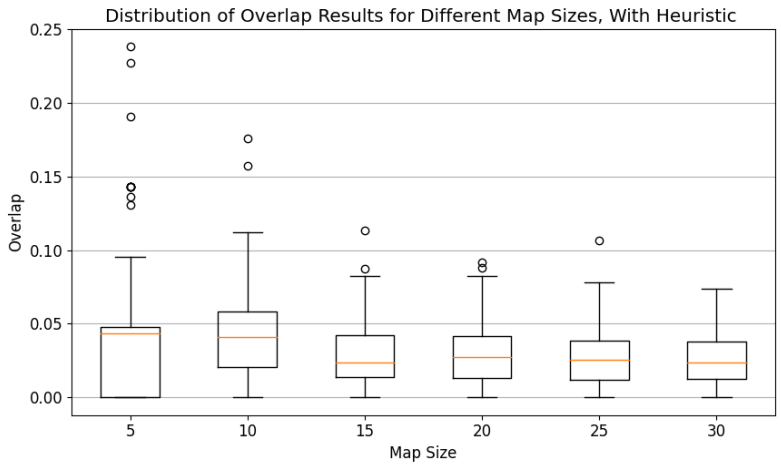
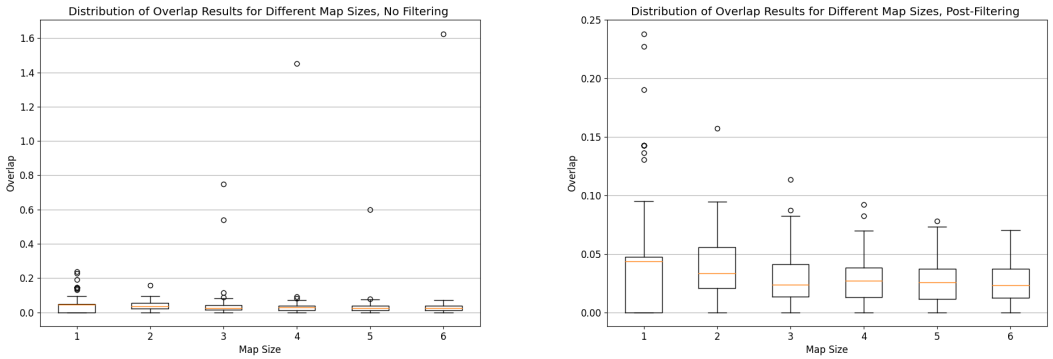


Figure 6.13: Box plot with the overlap distribution for six sets of maps.

For reference, truncation occurs when the episode overlap is larger than 5. However, upon excluding the more extreme outliers, the plot depicted in Figure 6.14b bears a resemblance to the results associated with the heuristic, showcased in Figure 6.13. Intriguingly, both the mean and Inter-Quartile Range (IQR) maintain identical values across these two scenarios. This observation suggests that, in the worst case, employing the heuristic yields results equivalent to not using it, while in the best case, it mitigates the influence of the most severe outliers. For this reason, there is no justification not to use the heuristic, as it guarantees that a map is always finished and makes the algorithm more reliable and explainable.



(a) Non-Filtered Outliers for Overlap Distribution (b) Post-Filtering Outliers for Overlap Distribution

Figure 6.14: Box plot with the overlap distribution for six sets of maps, in the case of not using the heuristic.

6.5.2.2 Overlap Evolution on the Size Spectrum

After analyzing the performance of specific scenarios, it is now time to analyze the whole spectrum. For this purpose, consider Figure 6.15, where the results for the whole set of maps are

presented.

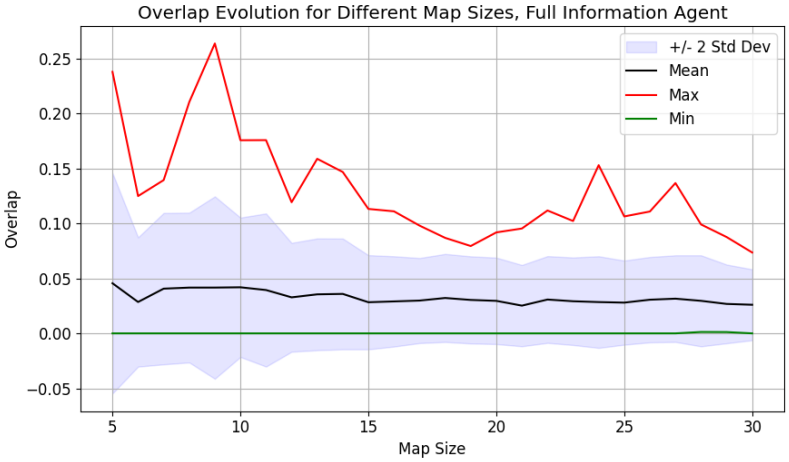


Figure 6.15: Overlap Statistics Across the Size Spectrum for Full Information Agent.

The results are very promising, validating the algorithm's design choices. The performance remains consistent across the entire range of map sizes, with a mean overlap value consistently below 0.05 in all scenarios. Furthermore, a downward trend in overlap values can be observed, reaching around 0.02 mean overlap in the best-case scenario. The maximum overlap values are, in general, inferior to 0.15, which, although not optimal, remains satisfactory. This is especially true when considering the presence of challenging maps within the dataset.

Importantly, it is worth noting that an overlap value of 0 was achieved in almost all map sizes. Additionally, the observation that the 0 overlap solution lies within the $\pm 2\sigma$ range across all cases implies a certain degree of consistency in achieving such results.

6.5.2.3 Outlier Analysis

Directing attention to the outliers, Figure 6.16 depicts a bar graph with the number of outliers when the algorithm does not use the heuristic. An episode is classified as an outlier if the overlap surpasses 0.5 or if episode truncation occurs. It is essential to clarify that this classification is intended to identify subpar outcomes rather than ensure statistical precision.

When tested with the heuristic, there was no single occurrence of an outlier. More importantly, it means that the agent had a completion rate of 100% across 2500 different episodes, and the rate of favorable outcomes is also at 100%. However, when not using the heuristic, the results get significantly worse. Consider Figure 6.16, the completion rate decreases to 99.3%, and the desirable outcome rate is around 97.4%. While these values are generally acceptable, especially in the context of Machine Learning algorithms, it is important to consider that robotics is a safety-critical oriented field, and therefore, the heuristic proves very desirable in this regard.

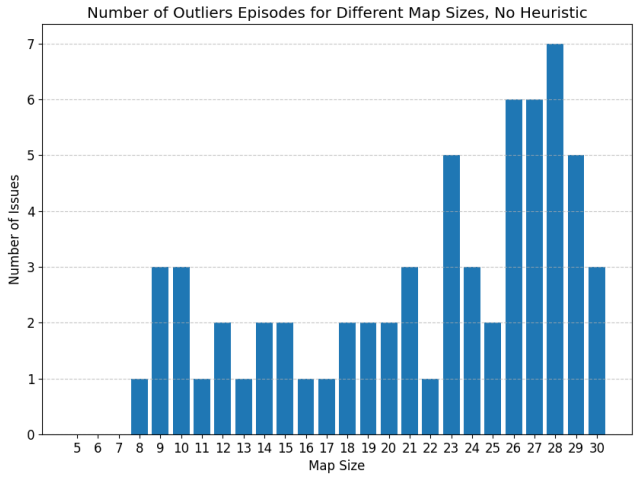


Figure 6.16: Occurrences of Outliers Episodes for Different Map Sizes, Not Using the Heuristic.

6.5.3 Sensor-Based Coverage Path Planning

One of the objectives of the algorithm is to have a model that can be used in different tasks without any change. For this, it is now turn to analyze the model and algorithm performance on the sensor-based coverage path planning problem or active exploration. Similarly to the previous case, the study will begin by analyzing the distribution of results on a generic sensor, and parallels will be drawn to the full information scenario. Subsequently, attention is directed toward comprehending the impact of sensor range and type on performance.

6.5.3.1 Results For Generic Sensor

The attention is now directed toward evaluating an agent equipped with a generic sensor to perform sensor-based coverage path planning. The choice of sensor is a camera with a range of $K = 3$. First, consider Figure 6.17.

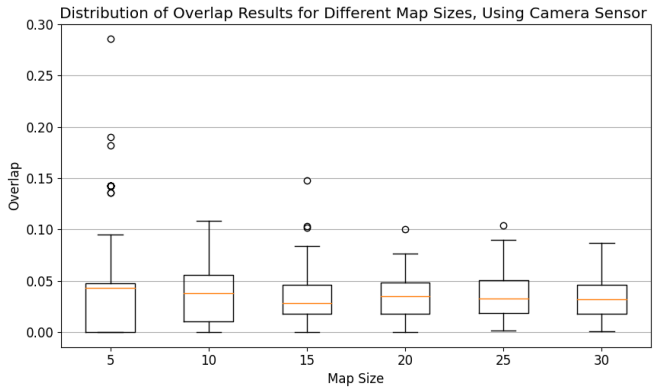


Figure 6.17: Box Plot with the Overlap Distribution for Six Sets of Maps in the Partial Observed Scenario.

The results exhibit remarkable similarity to the full information scenario. It is also worth highlighting that there is not a single result with overlap larger than 0.50, indicating a 100% rate for both completion and successful outcomes. In comparison to the full information agent, it is apparent that there are more data points outside the Inter-Quartile Range (IQR), and overall, the performance is slightly inferior. There is also more variance, which is reflected in larger IQRs, and the downward trend not being as noticeable for the mean overlap value, which is higher and more consistent. However, the fact that the agent fully adapted to a different task while still using the same model is remarkable.

Let us now shift the focus to the whole dataset. Figure 6.18 depicts the results for overlap on the whole set of test maps.

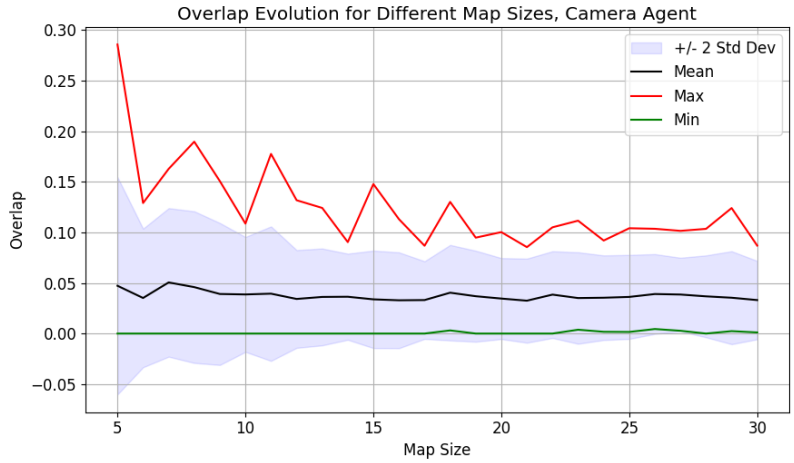


Figure 6.18: Overlap Statistics Across the Size Spectrum for Camera-Based Agent.

Once again, the outcomes are truly promising. Most notably, they closely resemble those attained by the full information agent. This alignment implies that the agent effectively fulfilled the requirement of successfully completing both robotic navigation tasks. While the maximum overlap value is slightly higher compared to the previous case, it still resides within a reasonable range, generally remaining below 0.2. The consistency of the mean overlap value across the dataset suggests that the agent genuinely acquired the skill of active exploration, avoiding the pitfall of overfitting specific scenarios. Once again, even with partial information, the agent achieved optimal solutions. However, such solutions became rarer when the map size surpassed the representation received by the agent in its observations. This phenomenon could stem from the loss of information from multiple sources, resulting in the absence of critical data required for achieving optimal solutions.

Nonetheless, the results are truly remarkable and show that the objectives of the algorithm were met. The one model agent can successfully complete distinct tasks and achieve generalization and near-optimal solutions in both. For an in-depth comparison, refer to Figure 6.19.

Directly comparing the full information agent and the camera-equipped agent, it is evident that their performance is notably similar, as highlighted by the identical shape of the curves in Figure

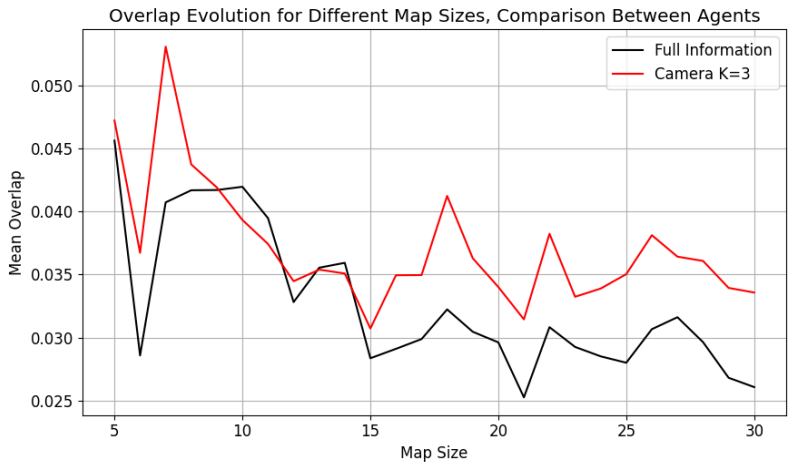


Figure 6.19: Comparison of Mean Overlap Between Camera and Full Information Agent.

6.19. The outcomes are also very similar until the map gets bigger than 15×15, in which a slight degradation of the camera agent performance is visible. It is important to note that this direct comparison is only possible because the agents are tested in the exact same dataset with the same seed. Nonetheless, the model's ability to proficiently handle both tasks remains an impressive feat. The fact that the curves are identical suggests that sets that are difficult for the full information agent are also more difficult for the camera agent. This is visible by the fact that most local maxima and minima occur for the same set in both situations. All in all, these results highlight the generalization ability of the agent.

6.5.3.2 Different Sensor Payload Analysis

The results on the sensor-based agent show that for a camera with a range of $K = 3$ the agent is capable of generalizing and performing the proposed tasks with proficiency. Nonetheless, it is important to remember that the agent spent most of the time training with that specific configuration, and a truly intelligent agent should be able to adapt to different sensors and ranges.

First, we start by comprehending how dependent the performance is of the sensor range parameter. For this evaluation, the parameter was set to values in the range of $K \in [1,5]$, and the agent was submitted to test in the same dataset. The results can be visualized in Figure 6.20.

As seen in the results, the agent can cope with different values for the sensor range. It is noticeable that the performance improves with increasing the sensor range, where $K = 1$ yielded the worst results and $K = 5$ the best. One can also notice that the agent's performance with $K = 1$ had a big spike in overlap values on the set of scenarios with the largest maps, suggesting that there is a breaking point where the sensor range starts not being adequate to the map size. Nonetheless, these results show that the agent mastered using the camera sensor to perform coverage path planning, as it can not only deal with differences in parameter values but as the value is increased, the more similar the solution gets to the full information agent.

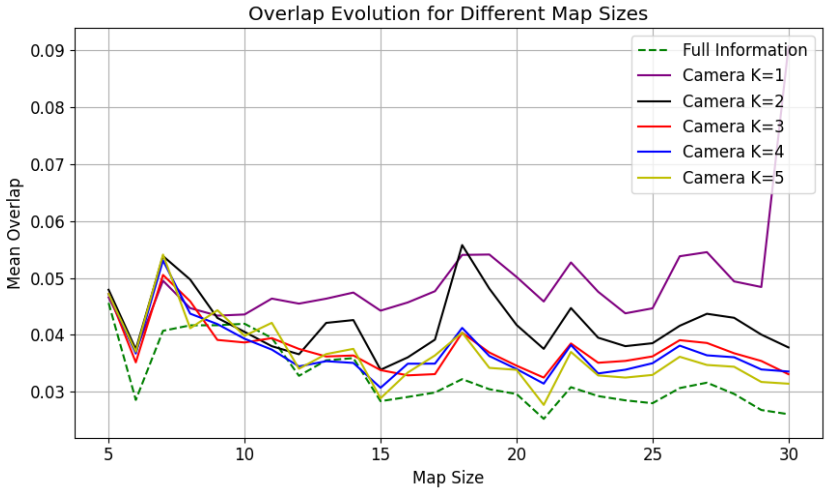


Figure 6.20: Comparison Between Different Sensor Range K for the Camera-Based Agent.

However, the camera sensor is not the only sensor that can be equipped in the agent. The LiDAR sensor is also part of the possible payload, functioning in a similar capacity to the camera, but can not see cells beyond obstacles. For studying the agent's capabilities with the LiDAR sensor, the same study was repeated, with the results depicted in Figure 6.21.

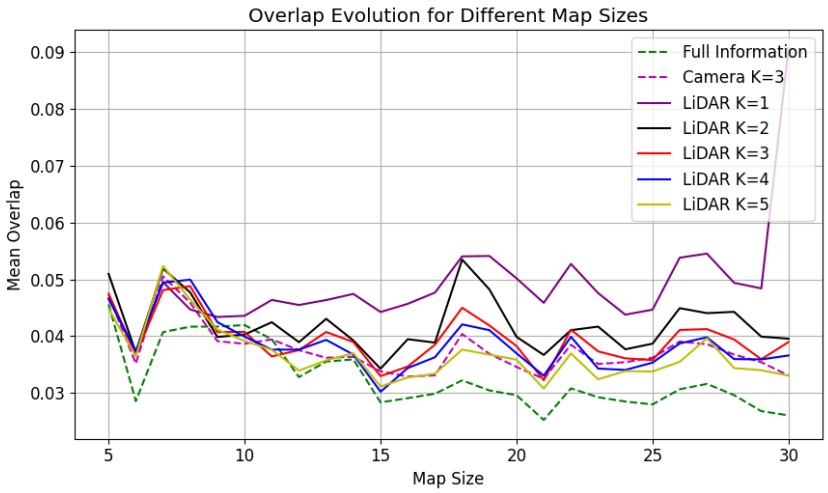


Figure 6.21: Comparison Between Different Sensor Range K for the LiDAR-Based Agent.

In this case, the results are as expected, similar to the camera-based agent. The relation between the camera range and performance remains practically similar. One thing to note, however, is that the LiDAR sensor performs worse than the Camera counterpart. This is evident by the fact that the Camera with $K = 3$ performs at an identical level to the best LiDAR $K = 5$. This result is somewhat expected, as the LiDAR extracts less information from the map when compared with the Camera, and therefore the agent can plan its path better. One curious fact is that by analyzing

the variants of the sensors with $K = 1$ the curve is the exact same. This happens because with this specific range value, the sensors are equivalent, and the algorithm is deterministic.

Nonetheless, it is remarkable that the agent can adapt and perform with a type of sensor it was never trained with, suggesting the generalization capabilities of the learning algorithm approach.

6.5.3.3 Sensor Noise and Failure

In all the given simulations, it was assumed that the sensors were not noisy and would never give false information. To better study the robustness of the model, a study was conducted on how injecting sensor failures affect the agent performance. For this purpose, two scenarios were tested where the values given by sensor readings were only usable 50% and 25% of time steps. The sensor range was set to $K = 3$, and the results can be visualized in Figure 6.22.

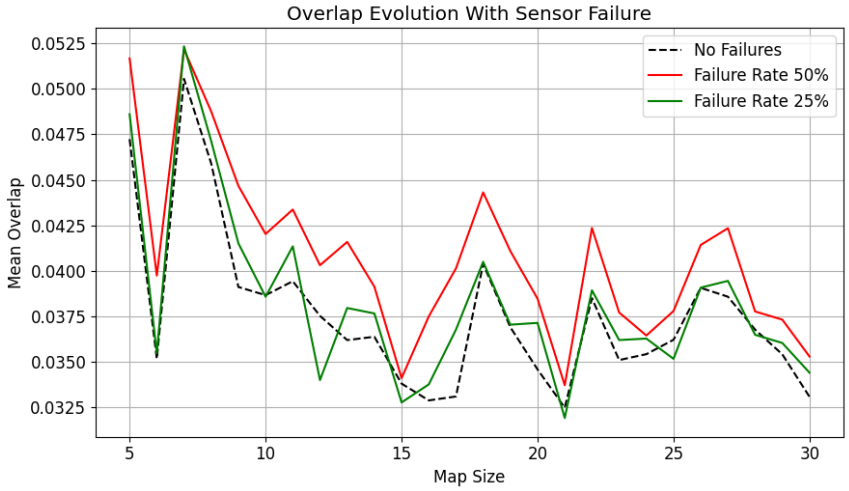


Figure 6.22: Sensor Failure Analysis.

The results show that the model is robust to non-ideal sensor behavior, being able to cope with a non-functioning sensor 50% of time. When the sensor failure rate is below 25%, there is not a noticeable degradation of performance. Even with higher values, although it is clearly visible that the agent struggles more with the task, the performance is still near-optimal.

This analysis suggests the model is robust and does not overfit to specific scenarios. These results give confidence that the model can be adapted to real-world applications where there is a lot of uncertainty and non-ideal behaviors that are not captured through simulation.

6.5.4 Point-To-Point Coverage

The robotic navigation spectrum was introduced as a concept at the beginning of this chapter. This subsection of results will serve to demonstrate that despite the agent being trained for coverage path planning, complete or not, the agent was truly able to learn how to navigate relying only on semantic map information.

For this, following the metric Δ_s defined in Section 6.5.1.2 will be used to measure how well the agent can move from point A to point B. The constructed dataset has 10000 different scenarios in which an agent has to perform point-to-point path planning. The performance metric signifies how lengthier the RL algorithm path is when compared to the Dijkstra's algorithm path.

The results are summarized in a bar plot depicted in Figure 6.23.

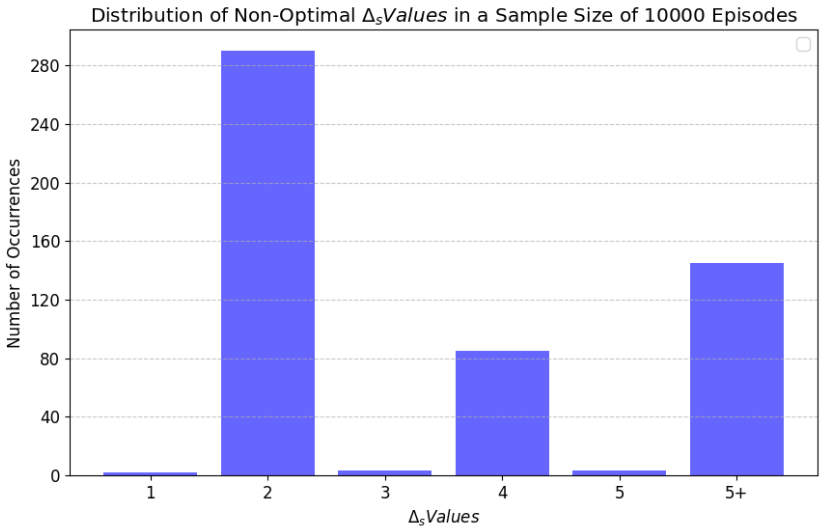


Figure 6.23: Overlap Statistics Across the Size Spectrum for Camera-Based Agent.

Out of the total of 10000 scenarios, 528 did not yield optimal results, leading to the RL algorithm being on par with Djikstra's algorithm in around 95% of instances. If all the solutions that yielded a path that is at maximum two cells longer are considered sub-optimal, then it is expected that the RL algorithm yields near-optimal solutions in around 98% of cases. Furthermore, the agent was able to complete the task in every scenario.

While the RL algorithm might not match the performance of Djikstra's algorithm, it is essential to acknowledge that the agent was never trained in this specific situation, and therefore, the obtained results still show proficiency on the task, even if not at an optimal level. Furthermore, in practical applications, the agent need not excel in this specific scenario since it has access to Djikstra's algorithm as a fallback.

Nonetheless, it is important not to overlook the importance of these achievements. With this algorithm, the agent was able to perform any of the tasks that were proposed, paving the way for more complex algorithms. For instance, Djikstra's algorithm alone cannot effectively minimize energy consumption when accounting for the robot's kinetic and dynamic model. In contrast, this solution can be adapted to consider such factors, contributing to a more adaptable framework. This adaptability stands as one of the primary objectives of this dissertation.

6.5.5 Comparison With State-Of-The-Art Algorithms

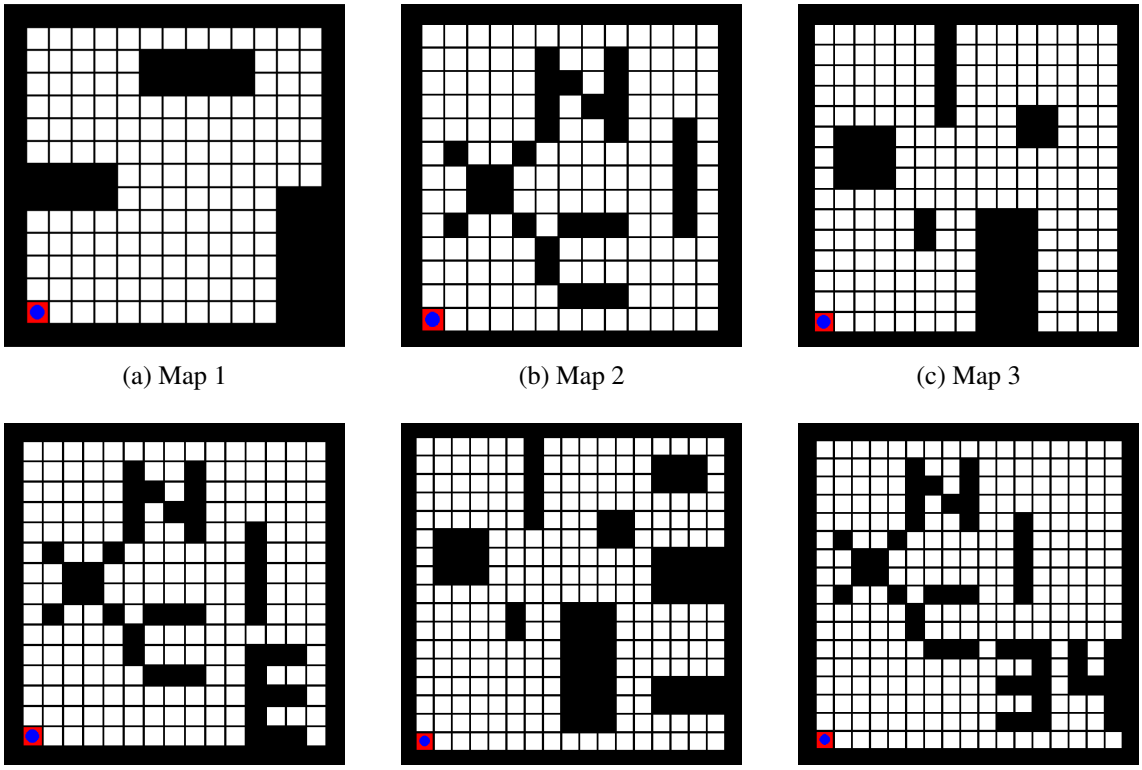
Until this point in the section, there was no basis for comparison between the algorithm and other approaches. In fact, this is a very common issue in this type of work, as there does not exist much standardization when it comes to performance assessment. Some approaches [97] even compare the algorithm's performance with a human controlling the agent due to this problem.

To correctly assess the performance of the algorithm, the agent will be directly compared with the benchmark done by the LG Electronics Advanced AI Team [21].

6.5.5.1 The Benchmark

In the work developed by the LG Team [21], the authors compare their RL and Hybrid RL algorithms with state-of-the-art classical algorithms. The used algorithms include: Full Spiral-STC [98], Smooth Spiral-STC [99], BA* [100], BSA [101], Epsilon* [102], and AD Path [103].

The algorithms are compared by doing 90% coverage in six different maps. These maps were implemented in this dissertation environment and can be visualized in Figure 6.24.



(a) Map 1 (b) Map 2 (c) Map 3

(d) Map 4 (e) Map 5 (f) Map 6

Figure 6.24: The Six Maps That Are Used In The Benchmark.

All the algorithms in the benchmark are set to stop coverage at 90%. In practice, this means that agents can optimize their routes not to cover cells that would inevitably lead to overlapping paths.

On the other hand, the agents developed in this work will always fully cover the whole map. Therefore, the results are not completely directly comparable. While it would be possible to terminate the episode whenever an agent has covered more than the threshold value, this approach might bring more harm than benefits. The agents in this work were trained to minimize the overlap in the complete coverage scenario and, therefore, will make compromises in the short term, which can result in worse outcomes in 90% coverage.

Nonetheless, it is the best comparison available, and since the developed agents are performing a more difficult task, the comparison is by nature unfavorable to them, and the results can still be interpreted.

6.5.5.2 Results

With the benchmark conditions established, the developed agents are ready for testing within this framework. Given the deterministic nature of the agent, a single episode is sufficient to determine its overlap. The results are presented in Table 6.4.

Table 6.4: Comparing overlap (%) of state-of-the-art CPP algorithms.

Method	**Map 1**	**Map 2**	**Map 3**	**Map 4**	**Map 5**	**Map 6**
Full Information Agent	**0.0**	8.0	**0.54**	**4.9**	**1.4**	8.8
Camera Agent $K = 3$	**0.0**	13.2	**0.54**	6.1	3.2	**8.0**
LG RL	9.1	9.6	9.7	10.1	10.0	10.8
LG Hybrid RL	7.1	**7.5**	7.6	8.0	8.1	8.8
BA*	18.3	22.1	21.8	28.9	26.3	34.8
ADP	17.4	20.0	23.0	25.5	26.3	36.8
BSA	16.4	23.3	22.2	27.1	27.4	37.1
Epsilon*	23.3	29.3	32.1	35.2	34.6	41.6
Spiral STC	19.7	24.2	22.8	29.4	29.2	37.4

Before analyzing the results, it is important to remember that all the algorithms that were not developed in this document were set only to complete 90% coverage of the map.

From Table 6.4, it can be inferred that any RL-based solution clearly outperforms all the classical CPP algorithms over these six maps. On the RL side, it is clear that the algorithm developed in this work outperforms the LG agents, both Hybrid and Regular RL.

There is a clear bias in both the Full Information Agent and the Camera Agent toward less cluttered maps, as seen by the optimal performances in Map 1 and near-optimal performances in Map 3 and 5. All these odd-numbered maps share a more structured environment with free space and more regularly shaped obstacles. In all these maps, the developed agents significantly outperform the LG agents, showcasing the power of the algorithm to achieve near-optimal solutions in what can be considered normal room layouts.

On the other hand, this might show a weakness of the algorithm. In the even-numbered maps, while still showing near-optimal performance, the results are slightly worse. This is due to their cluttered nature and unusual patterns, which require more robust decision-making. Nonetheless,

the developed algorithm still outperforms the LG approach in all maps besides Map 2, in which the Hybrid RL approach performs slightly better, resulting in less 0.5% overlap. This map, in particular, is the most cluttered one due to being the smallest of the three.

One curious aspect of the results is that, in general, the camera agent seems to perform at a similar and sometimes better level than the full information agent, especially in cluttered maps. This might be due to the fact that the limited information that is at the agent disposal leads to better overall solutions. While this might seem counter-intuitive, considering that the agents are not perfect, the more information available, the more decisions they can take, leading to more possible paths that can end up being worse. Nonetheless, the performance in both situations shows that any agent is capable of completing the task at a near-optimal level.

These issues show that further adaptations to the training algorithm can be made, focusing on making maps more cluttered and with more difficult decision-making. It is clear that most randomly generated patterns lead to maps more similar to the odd-numbered ones, and therefore, an extra effort to avoid overfitting should be considered. Another thing to consider is that the agent finished the training process with the camera sensor, which might also introduce bias toward the sensor. By finishing the training session with a set of episodes with varying types of sensors, this discrepancy might be mitigated.

Overall, in this specific dataset, it is fair to deduce that the algorithms developed in this work are the best performers, especially considering that the agents are actually doing full coverage. Furthermore, the algorithms developed by LG do not present results in other grid maps, and as far as the author's knowledge, there is no significant study on their generalization and adaptability to other scenarios, unlike in this dissertation.

A video of the algorithm performing on the benchmark's Map 1 can be visualized on the following link.

6.6 Final Considerations

In this chapter, a Deep Reinforcement Learning algorithm for robotic navigation was proposed and developed. The proposal is based on a transformative view of robotic navigation, which tries to view the task of navigation as a spectrum, where on one end, a robot must only go from A to B, and on the other, the robot must visit every point of interest. The idea is that by leveraging the generalization power of Reinforcement Learning, one robot should be able to "look" at a map and extract enough semantic information to perform any navigation task. A video of the algorithm in action can be found in the following link.

First, the problem was formulated as a Partial Observed Markov Decision Process, focusing on having the most generic and adaptable agent possible. The objective was not only to have a coverage path planning agent but also to make an agent capable of sensor-based coverage, active exploration, point-to-point planning, and other generic navigation tasks. The observation and action space was adequately modeled to enable this agent concept, and a heuristic component was added to guarantee coverage and reliability to the algorithm. Finally, an innovative and clever

reward function was designed to guarantee size invariance and to better adapt to the problems imposed by the generalization effort.

The algorithm is based on Deep Reinforcement Learning methodologies, and a value-based framework was chosen. Within this framework, the state-of-the-art method Rainbow Deep Q Network was selected as the base for the algorithm, and necessary adjustments were made. The training algorithm was also adapted on the base of transfer learning, curriculum learning, and the concepts of varying difficulties, competencies, and skills that an agent must have to perform navigation tasks.

The results validate the design choices for this Deep RL solution for robotic navigation. The proposed agents achieved near-optimal coverage path planning for scenarios of a dataset with over 2500 maps, even in the sensor-based case. The agent was also proven to be resilient to sensor failures and varying sensor payload setups, suggesting that the algorithm is robust enough to overcome the Sim-To-Real gap when deploying it on a physical robot. The agent also kept up with optimal solutions in the case of point-to-point planning, even though it was never trained for that specific purpose. Finally, the algorithm is compared in a benchmark with other approaches in the literature, where its capabilities shine by clearly outperforming both classical and RL approaches while doing an even more complex task.

This work provides a solid framework for any robotic navigation task through its developed concepts. It paves the way for robots that can understand what to do just by interpreting a map representation, and the results show that this approach can be viable despite the difficulties in scaling with larger map sizes. The work can still be improved by adding support to multiple agents and enhancing some training procedures, like the agent curriculum or the scenario generation algorithm.

Chapter 7

Multi-Agent Reinforcement Learning for Coverage Path Planning

Having established a robust foundation in the application of Deep Reinforcement Learning for single-agent robotic navigation in the previous chapter, this chapter now directs its attention toward the domain of cooperative multi-agent systems. The focus here revolves around delving into the landscape of cooperative strategies within robotic navigation, thereby exploring methods that can cope with multiple agents, harmonizing their efforts to achieve shared objectives.

The chapter will begin by introducing the problem and presenting the objectives in Section 7.1. Section 7.2, then delves into formulating the problem, highlighting challenges specific to the multi-agent domain. After formulating the problem, the proposed solution is translated to a Dec-POMDP framework in Section 7.3, and the implementation is detailed in Section 7.4. In Section 7.5 the algorithm will be tested and the obtained results will be showcased with a comprehensive evaluation of both the accomplishments and limitations of the proposed solution. Lastly, some final considerations are given in Section 7.6.

7.1 Challenges and Objectives

The main objective of this chapter is to develop Multi-Agent Reinforcement Learning methods for robotic navigation. Unlike the previous chapter, this one will focus more on the complete coverage path planning framework, as it is the one task that benefits the most from multi-agent cooperation. Nonetheless, the algorithm must still be able to generalize and adapt to different situations and tasks.

Within the broader context of this document, this chapter represents the culmination of the groundwork laid thus far. Serving as an incremental advancement, especially in comparison to the Deep Reinforcement Learning presented in Chapter 6, certain concepts and features will not be re-explained. Consequently, unlike the previous chapter, this section cannot stand alone as an independent reading.

The multi-agent setting is significantly more demanding than single-agent. While a lone agent only had to understand and interact with the environment, in the multi-agent context, each agent must possess awareness of its counterparts. In addition to this awareness, agents must comprehend one another's objectives and policies to execute tasks effectively. This requirement places strain on the learning process, particularly as agents attempt to grasp the dynamic policies of other agents, which are subject to constant change. Besides the learning component, the multi-agent setting also necessitates a robust framework for synchronization and information sharing, which are not trivial, especially in real-world scenarios.

The contribution of this chapter will be a generic multi-agent Reinforcement Learning algorithm that can be used in cooperative coverage path planning on any 2D Grid without explicit constraints on the number of agents. The main objectives of this chapter are as follows:

- Model the Cooperative Coverage Path planning problem as a generic Decentralized Partial Observed Markov Decision Process.
- Develop a learning framework that can cope with any number of agents.
- Employ the state-of-the-art Multi-Agent Reinforcement Learning technique Parameter Sharing to extend the Rainbow DQN to the Multi-Agent Setting.
- Use Transfer Learning to adapt the Single-Agent framework to Multi-Agent.
- Study the differences between cooperative and competitive reward structure paradigms for the Cooperative Coverage Path Planning Task.
- Investigate the algorithm's performance and robustness through experimentation with a dataset of maps.

7.2 Problem Formulation

Once again, consider a generic 2D square-shaped map $\mathscr{M}$ with maximum dimensions $M \times M$. The map is obtained via approximate cellular decomposition, and the cell size equals the robot size. The obstacle number and positions are randomly generated, and all out-of-bound cells are considered obstacles.

In this problem, the information is encoded at the cell level, where there exist Non-Covered Cells (Points of Interest), Covered Cells, Obstacles, and Cells with no current information. The last type of cell will not be used in this chapter, as they are used only in sensor-based coverage path planning. However, they are kept consistent with previous models and to ease future expansion. At the beginning of every episode, the map will be composed of Non-Covered Cells (Points of Interest) and Obstacles, and the episode terminates when all non-obstacle cells are visited. The cells only need to be visited once to be considered covered, and can be done by any agent. A typical multi-agent coverage scenario can be observed in Figure 7.1.

With respect to the agents, consider a group of N homogenous agents $A_g = [1,...,N]$ with a minimum of one element and a maximum that is only theoretically limited by the number of available free cells. The agents are synchronized and select an action at every discretized time step t. The action can move them in any cardinal direction $\mathbb{A} = [North, South, East, West]$. If an agent is out of sync, an individual action can be skipped. This can be used to implement an asynchronous framework or to simulate failures in communication. Nonetheless, the only framework that will be considered is the synchronous one.

The agents have an implicit priority given by their id, their position in the vector A_g. An agent is not aware of its own position in the array, so it should not assume priority over any other. The priority is only relevant to prevent situations where agents move to the same cell, and in this implementation, the safety controller that implements the priority acts as a central omniscient controller that blocks the action of the agent with less priority.

The agents will use the centralized training decentralized execution framework. During training, all the agents use the same network and have access to each other observations, being that all observations are stored in the same replay buffer, and there is no explicit information regarding which agent generated it. During execution, each agent has its own network and cannot access the other agent's observations, policies, or actions.

Independently of how the agent's reward structure is modeled, the goal will always be to completely cover the map in the minimum of time steps t possible. This implicitly means that the agents will have to cooperate to reach this goal, as self-focused agents can be harmful to the group. Nonetheless, flexibility regarding the collaborative or competitive reward structures will be given to the agents.

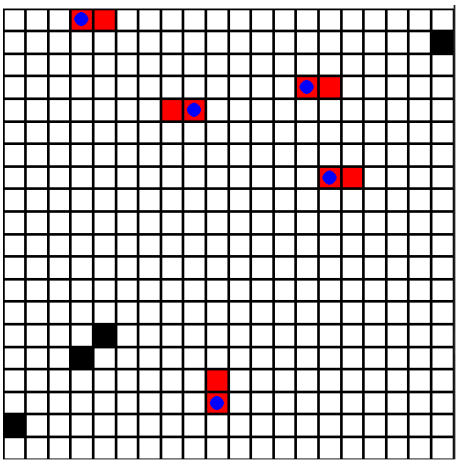


Figure 7.1: A typical multi-agent coverage path planning scenario on a 20×20 map with a group of 5 agents.

7.3 Decentralized Partial Observable Markov Decision Process

To address the cooperative coverage path planning problem, it will be translated to a Decentralized Partially Observable Markov Decision Process (Dec-POMDP), which is defined by the tuple: $\langle N, \mathbb{S}, \mathbf{\mathbb{A}}, P, \boldsymbol{R}, \mathbf{\Omega}, \mathcal{O}, \gamma \rangle$.

This choice is an extension of the POMDP designed in the previous chapter. Since the design choice was proven to be successful, this framework for the decision-making problem is the most adequate choice, as this new problem is an increment to the single-agent problem where the agents will be homogenous, have similar reward structures, and in general, cooperate towards a common goal.

The formulation of the Dec-POMDP will take into account the global objective of having a general framework for robotic navigation tasks and, ideally, being able to complete the same tasks in the single-agent setting. The resulting agents must be able to do coverage path planning cooperative or not, in any given map, with any number of agents and starting positions.

7.3.1 Action Space

The joint action space $\mathbf{\mathbb{A}} := \mathbb{A}^1 \times ... \times \mathbb{A}^N$ is defined by the possible combination of actions taken by the group of agents. In this problem, the agents are homogeneous, meaning that $\mathbb{A}^1 = \mathbb{A}^2 = ... = \mathbb{A}^N$ and therefore, at the singular level the action space can be defined as:

$$\mathbb{A} = [North, South, East, West] \tag{7.1}$$

In the multi-agent setting, an extra action where the agent chooses not to move could be interesting. The wait action would be useful to prevent collisions and save energy whenever the agent deems it cannot contribute more to the overall goal. Nonetheless, it was a deliberate choice not to include it in the action space.

Including this action would remove the compatibility with the single-agent algorithm developed in Chapter 6, meaning it would not be possible to use transfer learning in this new setting. Furthermore, the environment already has a built-in mechanism that prevents collisions, and therefore, it is acceptable that the agent take more risks when choosing their actions regarding inter-agent collisions. The question of saving energy is not relevant to the problem formulation, which is of minimizing the time to cover a map fully. With the waiting action, at best, the time taken would be the same as if it were not present, and therefore it is not included. Furthermore, if the agent does not select an action, it will perform the equivalent to the wait action, which is an implicit method to implement this action without being directly in the action space. The waiting action can easily run into deterministic loops and was also shown to degrade performance in conducted tests on single-agent and multi-agent settings.

The heuristic action of moving towards the closest non-covered cell is still present and not directly part of the action space. The policy chooses the heuristic action in the same way as presented in Section 6.4.2.4. It serves as a way to prevent deterministic loops in the observation

space and to speed up the training process. As the objective is to minimize time, choosing this action when an agent is stuck and not visiting new tiles will not result in worse outcomes but in a more active and aggressive policy.

7.3.2 Observation Space

In this problem, the agents will be considered homogenous, sharing the same action space and having a similar structure in the observation space. Consider the joint observation space $\mathbf{\Omega} := \Omega^1 \times ... \times \Omega^N$ being all the possible combinations of observations between the N agents. In these observation spaces, the observation function $\mathcal{O}$ is shared between all agents. However, it is impossible to have different agents with equal observations $o_1 \neq o_2 \neq \ldots \neq o_N$.

The observation space Ω^i can be defined as:

$$\Omega^i = \underbrace{\mathbb{R}^{3\times4\times41\times41}}_{\text{Map Representation } M_p^i} \times \underbrace{\mathbb{Z}^{3\times4\times2}}_{\text{Out of Bounds Information } oob^i} \times \underbrace{\mathbb{B}^{3\times5}}_{\text{Last Actions } l_a^i} \times \underbrace{\mathbb{Z}}_{\text{Battery Level } b^i} \tag{7.2}$$

The components observation space is the same as in the previous chapter in order to enable reutilizing the trained model for the multi-agent task. A brief description of the observation space components is as follows:

- **Map Representation M_p^i:** A $\mathbb{R}^{3\times4\times41\times41}$ tensor containing all the last three stacked frames of the map. The map is centered on the agent i and has four channels: Points of Interest, Points of Non Interest, Obstacles, and Agents Position. Once again, the nearest 20×20 cells are unmodified, and all cells beyond that limit will be average pooled as described in Section 6.3.2.3.

- **Out of bounds information oob^i:** A $\mathbb{Z}^{3\times4\times2}$ tensor with the additional information that was lost in compression from the point-of-view of agent i in the last 3 frames. Contains the number of points of interest and compressed layers in all cardinal directions.

- **Last Actions l_a^i:** A $\mathbb{B}^{3\times5}$ tensor that contains all the last 3 actions of agent i in one-hot enconding.

- **Battery Level b^i:** A scalar tensor that represents the battery level of agent i.

It is important to note that every element is specific to the agent and will most likely not be shared between agents. Besides the Map Representation, all the other components are only relative to a singular agent. The reason behind this limitation is that in order to consider other agents, it would be necessary to add a dimension to the tensor. This dimension would have to have a fixed size, implicitly capping the number of possible agents. The Map Representation M_p contains the most relevant information regarding the other agents, so there should not be much difference in omitting the additional information. Furthermore, sharing the extra information would require more strain on the communication and synchronization protocol.

An example of how the observation changes depending on the respective agent can be seen in Figure 7.2. It is clear how using a map on the agent referential frame is advantageous. In most MARL solutions, all agents must have some kind of implicit identifier in the observation space in order to identify themselves. In this situation, since the map is centered on the respective agent, this problem does not exist.

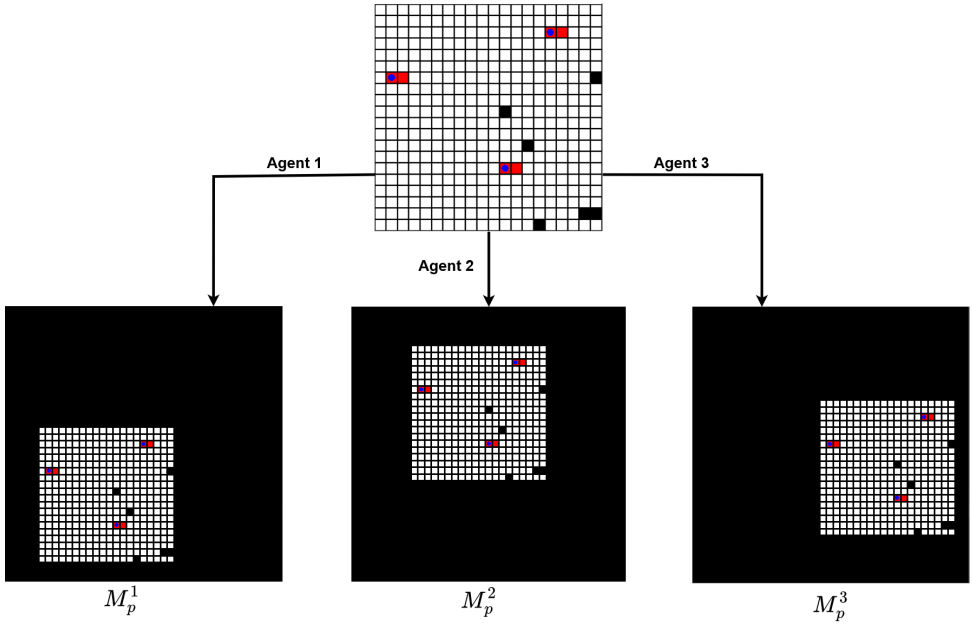


Figure 7.2: An illustrative example of how the Map Representation M_p changes depending on the agent that is visualizing. To ease visualization, Agent 1 is the one on the top right of the map, and the black background is part of the centered map representation.

7.3.3 Reward Function

The missing piece of the Dec-POMDP is the Reward Function $R : \mathbb{S} \times \mathbb{A} \times \mathbb{S}$. Unlike previous approaches to describing the formulation of problems as Markov Decision Processes, in this section, the problem will be left open to various options. The reasoning behind this choice is that the Reward Function structure dictates how the agents will tackle the optimization problem, and in Multi-Agent scenarios, there can exist competitive, cooperative, and Mixed formulations of this function. In further sections, studying and understanding how these changes affect behavior and performance will be of great interest, and therefore, multiple Reward Function structures will be presented.

Before delving into new Reward Functions, consider the starting point as the function presented in Section 6.3.3:

$$R^i = r^i_{ts} + r^i_{crash} + r^i_{new}, \tag{7.3}$$

where $r^i_{ts} = -1$ is a penalty that is given every timestep, $r^i_{crash} = -1$ is a penalty given whenever the agent has a collision, and $r^i_{new} = 1$ is given every time an agent visits a point of interest. It is important to remember that these values were chosen in such a way that the value of any state would most likely be zero-valued if following an optimal policy. This gave considerable benefits in performance due to this power of generalization and size-invariant value function.

For the multi-agent scenario, three different possible prototype functions will be considered and later empirically analyzed.

7.3.3.1 Competitive Setting - Maintaining the Reward Function

One tempting approach might be to keep the reward function untouched since the performance in the previous chapter was very impressive. While very simplistic at first glance, this perspective might have more merit than it seems.

Using the reward function in equation (7.3) implies that the agents will tend to be competitive by nature. There is no common shared objective imbued in the function, as the reward is only dependent on the agent's action and outcomes. Nonetheless, the interesting part of this reward function is that there is no finite amount of achievable reward, and therefore, the agents are not inherently incentivized to outperform the other, simply to minimize their losses.

To better understand the intricacies, consider now a case where $r^i_{new} = 2$. In this situation, there would be a finite reward available equal to the number of points of interest n_p. In the situation where there are two agents, the solution where both agents cover exactly half of the map is a Nash Equilibrium (NE) point. Nonetheless, the algorithm is not guaranteed to converge to this equilibrium point, especially because the average reward is not considered, and therefore, the agents will look into maximizing their own reward. In this situation, having a reward larger than the one obtained in the NE situation will inevitably harm the other agent, as this game can be modeled as a two-player zero-sum game. Therefore, in this competitive setting, the most likely outcome would be the agents trying to maximize their reward by harming others, as effectively, every time an adversary visits a point of interest, it reduces the possible cumulative reward of the agent.

However, in the initial case, the maximum cumulative reward equals zero for all agents, creating a more interesting dynamic. The Nash Equilibrium is still the exact same, where both agents cover exactly half of the map. However, in this case, during the optimization process, an agent does not get more cumulative reward by "stealing" points of interest from the other. In fact, one benefits from the other doing as best as possible due to the episode ending faster, and therefore, there are fewer opportunities to receive penalties. This leads to encouraging behaviors that are more likely to converge into the Nash Equilibria than the other completely competitive scenario.

While the reward function structure is still competitive by the fact that there is no implicit shared goal or component, the fact that agents benefit from cooperating with each other in order to minimize their losses makes for an interesting mixed reward structure that is worth considering.

7.3.3.2 Cooperative Setting - Introducing a Shared Reward

One of the most common ways to change the reward structure is to introduce a shared or common reward into the reward function. By doing this technique, the setting is changed to a cooperative setting. In the literature, to be considered a fully cooperative setting, all agents must share the same Reward Function. However, that can be impractical, and the structure that will be presented almost qualifies in that classification and, therefore, will be referred to as the cooperative setting.

With this in mind, consider the following Reward Function:

$$R^i = r^i_{new} + r^i_{ts} + r^i_{crash} + \frac{K}{N-1} \sum_{j=0, j\neq i}^{N} (r^j_{new}). \tag{7.4}$$

In this function, the scalar gain K serves as a tunable parameter for the importance of the other agents' reward importance. If K=N-1, every visited tile is valued the same, independently if it was the agent i or any other. However, the higher this gain, the more difficult the credit assignment problem gets, as other agents can contribute disproportionally to the overall success. Another thing to note is that the previous function in equation (7.3) is equivalent to this one when the gain is set to $K = 0$, where there is no contribution from the other agents to the reward function.

This prototype function will be tested with multiple values of K in order to understand better how the algorithm performs and how the agents interact with each other.

7.3.3.3 Cooperative Setting - Keeping Size and Value Invariance

The newer cooperative reward function presented in equation (7.4) loses some of the proprieties of the original reward function represented in equation (7.3). The fact that there is an additional positive term to the initial reward function means that the maximum cumulative reward is no longer zero and is now a map size-dependent positive scalar. In practice, this removes most of the useful properties that were studied and presented in the previous chapter and might have undesirable results.

Due to the success of this function type in the single-agent setting, it is desirable to investigate the applicability of those techniques in the multi-agent setting. Therefore, one can change the reward function to:

$$R^i = (1-K) r^i_{new} + r^i_{ts} + r^i_{crash} + \frac{K}{N-1} \sum_{j=0, j\neq i}^{N} (r^j_{new}). \tag{7.5}$$

This yields a function that maintains the size invariance condition as long as K is bounded in $[0,1]$. In the case where $K = 0$, the function is once again equivalent to the original equation (7.3), being that only the singular agent i is considered. On the other hand, if $K = 1$, then only the contribution of the other agents is taken into account. Finally, a middle ground where $K = 1/N$ gives the same contribution for every agent, resulting in what would be closest to a fully shared reward. It is important to note that a fully shared reward is not achieved because the crashes are

penalized individually. Nonetheless, due to the rarity of crashes during most parts of training, it would still be fair to say that the reward structure is fully cooperative.

7.4 Learning Algorithm

With the problem formulated as Dec-POMDP, it is now time to shift focus on how to solve it. This work will use a Multi-Agent Reinforcement Learning framework that builds upon the DRL framework built in Chapter 6. The modified Rainbow DQN is reutilized, and the model is trained via the state-of-the-art technique Parameter Sharing [66]. This paradigm adapts well to the proposed task, as there already exists a model that is very competent in coverage path planning and, therefore, could be used as a starting point to train multiple agents with similar competencies. This section will start by briefly revisiting the network architecture and introducing some details of parameter sharing, and finish by giving an overview of the learning algorithm and used parameters.

7.4.1 Network Architecture and Parameter Sharing

In Chapter 6, the chosen framework for learning was the State-of-The-Art learning algorithm Rainbow DQN. Due to its success and the use of transfer learning to reduce computation, the same framework will be kept.

For these reasons, the network must be kept unaltered, and its architecture can be revisited in Figure 7.3.

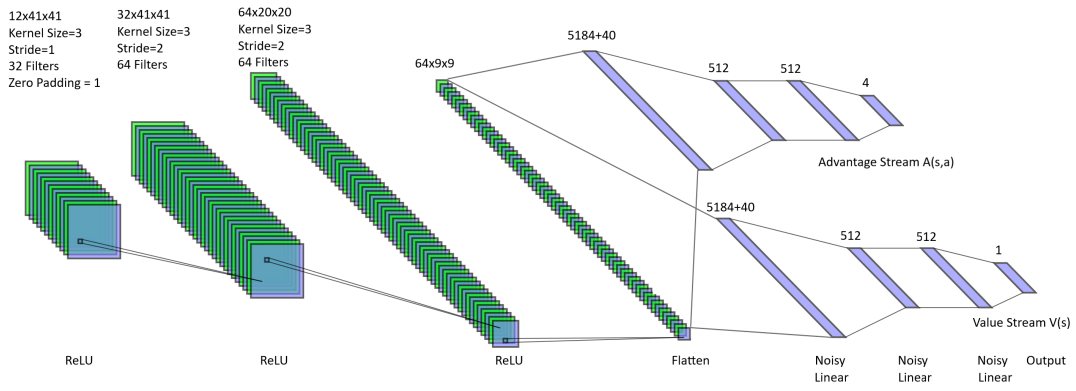


Figure 7.3: The Neural Network Architecture.

It could be argued that the architecture of the network could get larger to accommodate the increasing complexity of the multi-agent setting. Nonetheless, the same architecture proved enough to handle the task and, therefore, did not suffer any further alterations. Furthermore, there is an incentive to maintain the same architecture in both settings to ensure compatibility between the different paradigms.

One important detail of managing parameter sharing is that, as the name suggests, all agents will share the same network during training. The formulated Dec-POMDP excels at abstracting

the agent identity by centering the map on the respective agent, and therefore, from the perspective of the model, there is no difference between any of the agents, and there is no risk of overfitting to a specific agent as can occur when explicit agent id's or marks are used.

The parameter-sharing scheme also introduces benefits regarding computational efficiency, as if observations are fed in batches, the action for every agent can be selected in a single network feedforward operation. To add to this advantage, it only requires one experience replay buffer where all transitions are treated the same, and updating the parameters will also update all agent's policies.

On the learning side, having a singular centralized network can also help with credit assignment and coordination of agents. Technically, all policies and transitions are derived from the same model, resulting in the learning algorithm having less difficulty adapting to dynamic policies. Furthermore, having multiple agents leads to a richer and varied replay buffer, where the same state is observed from different perspectives, once again helping with convergence and recursive thinking. If all agents are controlled by the same model, then in the limit, the model can learn to predict itself on the other instances.

The parameter sharing utilizes a Centralized Learning Distributed Execution (CTDE) framework, where all the agents use the same model for training but can have independent models during execution. This brings flexibility to the solution, as completely unrelated models can be used during execution, and the agents can be more decoupled from each other. This paradigm can be visualized in Figure 7.4, where on the left, a centralized scheme is presented, and on the right, the decentralized operation is demonstrated. In this work, the execution is indeed decentralized, but the model will be the same for every agent.

7.4.1.1 Hyperparameters

To finalize the analysis of the network architecture, Table 7.1 contains the network's and learning algorithm's most relevant hyperparameters.

Comparing these hyperparameters to those of the single-agent algorithm, only two changes are noted in the Learning Rate α and the Soft Update Factor τ. The former has been reduced by a factor of five, and the latter is four times smaller in the multi-agent algorithm. These modifications are driven by two primary considerations:

Firstly, the model is pre-trained, so initial updates should not be so large that the competencies of the agent are lost.

Secondly, Multi-Agent Reinforcement Learning often contends with noisier reward signals and generally exhibits lower robustness, making it more sensitive to larger parameter adjustments. The reduction in these two parameters controls the magnitude of parameter updates, and their reduction is aimed at stabilizing the learning process.

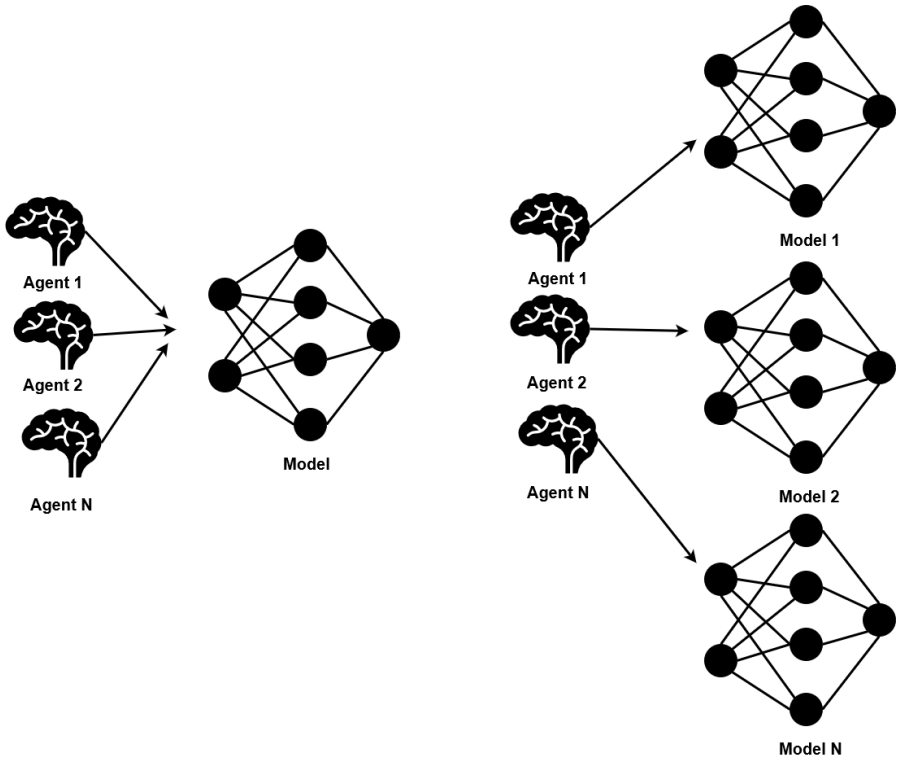


Figure 7.4: An illustration of centralized and decentralized schemes for learning and execution. On the left is the centralized paradigm, and on the right, is the decentralized version.

7.4.2 Training Algorithm

With the network architecture laid out, the missing piece for the solution is the learning algorithm. Once again, the typical Rainbow DQN training scheme is changed to better suit the task at hand. Building from the single-agent algorithm, the multi-agent version utilizes the following features:

- **Partial Episode Bootstrapping:** Changing the algorithm to not deal with truncated episodes as if they were terminated ones.
- **Partial Resetting:** If an episode is truncated, the next episode will start where the last ended.
- **Saving Challenging Tasks:** If an episode is truncated, it should be replayed from the beginning in future iterations
- **Imitation Learning:** Use non-RL policies with expert information to speed up the learning process.
- **Curriculum and Transfer Learning:** Train the agents in multiple scenarios of increasing difficulties. Also, use previously trained agents to bootstrap the process.

From all the features used in the single-agent Section 6.4.2, the only one that is not present is the generation of uncommon patterns. The reasoning behind this choice is that using multiple

Table 7.1: Multi-Agent Algorithm Hyperparameters

Parameter	**Value**
Minimum Steps Before Learning T_{start}	5000 Frames
Soft Update Factor τ	**0.01**
Discount Factor γ	0.999
Batch size B	32
Optimizer	ADAM
Loss Function	Smooth L1 Loss
ADAM Learning Rate	**0.00001**
ADAM ε	0.000156
Maximum Gradient L2 Norm	2.5
Priority exponent α	0.5
Priority correction β	$0.4 \rightarrow 1$
Noisy Net Initial std deviation σ	0.5
Memory size	1 Million Transitions
Learning Period K	4
Number of Stacked Frames	3
Multi-step return length n	1

agents already has this effect. Furthermore, the primary competency to learn is to divide areas and cooperate with each other, and starting with already covered random patterns can be harmful to this goal.

7.4.2.1 Random Number of Agents

On top of the mentioned features, there is a new addition to the algorithm. Whenever a new training episode is initialized, the number of agents is randomly sampled in the interval $[1, N]$.

This is particularly relevant when the learning algorithm is making the jump from one agent to two. It was noted that if the number of agents was not randomly sampled, the agents would learn to cooperate and separate their coverage areas, but the generated paths were far from optimal with lots of unnecessary overlap. The reason behind this phenomenon is that by training only in a multi-agent setting, the learning prioritized learning to cooperate instead of generating patterns with low overlap.

An easy solution to this issue is to have some episodes where there is a singular agent, this way the algorithm can focus solely on learning how to path optimally. While still learning the competencies for the multi-agent scenario, significantly improving performance.

In the results section, the results of two models trained in the exact same conditions except for the random number of agents will compared against each other.

7.4.2.2 Curriculum Learning

In order to finish the description of the learning algorithm, the focus will shift to the curriculum portion of learning. The agent's training will be divided into two stages. The first consists of

adapting the single-agent model to a multi-agent model in scenarios with only two agents. This stage serves as a smaller step into the multi-agent setting where the main objective is for the agents to learn to be aware that they should cooperate in the task. The curriculum of the training is presented in Table 7.2.

Table 7.2: Environment Configurations for Curriculum Training

Parameter	**env1**	**env2**	**env3**	**env4**	**env5**	**env6**
Training Steps	1×10^6	2×10^6	3×10^6	5×10^6	1×10^7	2×10^7
Number of Agents	2	2	2	2	2	2
Max Size	5	10	15	20	25	30
Min Size	4	5	10	10	10	5
Number Obstacles	5	10	20	30	40	50
Random Coverage	False	False	False	False	False	False

After completing this training set, the model enters the second stage where the objective is to increase the number of agents and perfect the multi-agent cooperation. With this in mind, the training set is repeated twice with 5 and 10 agents. Each training set can be completed in less than 24 hours on a regular desktop with a GeForce RTX 3060.

The training algorithm pseudo-code can be visualized in algorithm 13

Algorithm 13 Curriculum Learning multi-agent CPP Training

Require: Environment configurations E, replay period K, memory size M, PER exponents β, Learning Start Step T_{start}

Initialize replay memory $\mathscr{D}$, PER exponent β and its increment Δ_β, online network θ and target network θ^-
for environment e in E **do**
    Initialize environment e, training budget T
    Observe $\boldsymbol{O_0}$ and choose $\boldsymbol{A_0}$
    for $t = 1$ to T **do**
        Use online network θ to choose the batch of actions $\boldsymbol{A_t}$
        Take Action $\boldsymbol{A_t}$, observe $\boldsymbol{O_{t+1}}, \boldsymbol{R_{t+1}}$ and store all transitions
        if $t > T_{\text{start}}$ and $t\%K == 0$ **then**
            Reset Noisy Linear Weights
            Sample k transitions from the replay buffer
            Compute loss
            Perform Gradient Descent to update online network weights θ
            Soft Update the target network weights θ^-
        end if
        $\beta \leftarrow \beta + \Delta_\beta$
        if S_t is Terminal **then**
            Reset the environment
            Randomly sample the number of agents
        end if
    end for
end for

The multi-agent version is very similar to the original algorithm. The main difference is in lines 4, 6, and 7, where instead of having a single scalar for observations O, actions A, and rewards R, all these variables are vectors of size N, distinguished by the usage of the bold font. The advantage of doing it in batches instead of doing a for-loop that goes through every agent is that it is much less computationally expensive, as it only requires one feedforward operation on the neural network instead of N. It also simplifies the synchronization process, as all agents take their action at the same instant. On the other hand, taking action sequentially would ease agent cooperation, as there should not be any collisions because of miscommunication. Nonetheless, in this work, it was opted to use the worst-case scenario. This approach can be more robust in real-world applications where communications are limited.

7.5 Results

With the solution fully developed, this section will focus on analyzing the algorithm's performance and robustness. It will begin by describing the methodology used in the study and the objectives of the experiments. After, the experiments will be conducted, and the results will be presented. Along with the results, an analysis will always be provided, aiming to compare differences between hyperparameters, reward structures, and the overall algorithm performance and advantages.

7.5.1 Methodology

In the evaluation of the multi-agent scenario, a similar approach to the single-agent will be taken. The models will be tested in scenarios ranging from sizes $[5, 30]$. Each set has 100 maps of the same size, with up to 15% obstacles. The dataset will be the same as in Chapter 6 with added agents. To make comparisons as fair as possible, the version of the map with $N+1$ will be the map with N agents where the only change is the new agent. This process can be visualized in Figure 7.5.

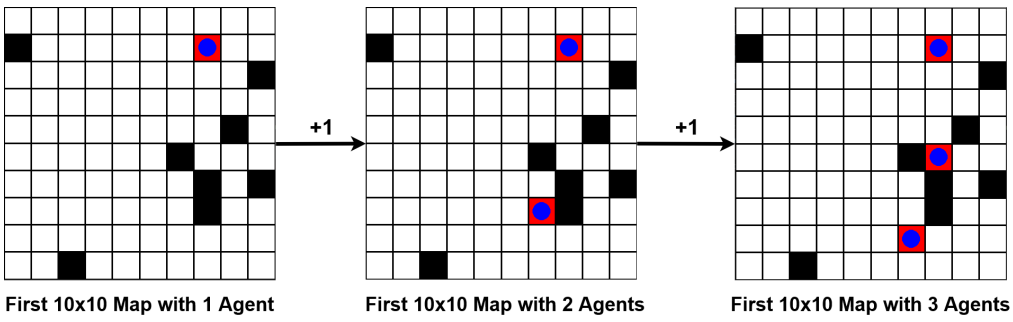


Figure 7.5: The process of adapting maps to different numbers of agents.

One thing important to note in the evaluation is that performance can be very dependent on the initial agent positions, as cooperation is easier when agents are far away from each other. This is the reason behind keeping the scenarios as similar as possible between datasets.

The performance metric Overlap, described in Section 6.5.1 does not translate well to multi-agent settings. While it can definitely still be used, as the lower the overlap, the better the agents performed, its analysis is not as trivial as for the single agent setting. A performance metric that better suits the problem is the Time Saved Factor t_{sf}, which can be described as:

$$t_{sf} = \frac{T}{n_{poi}(s_0)} \tag{7.6}$$

where T is the terminal timestep and $n_{poi}(s_0)$ is the number of points of interest at the initial state s_0. Considering that in a single-agent scenario, the best case is when $T = n_{poi}(s_0)$, implying that the episode was optimal and that at every timestep, the agent visited a new point of interest. So, this metric introduces an easy reference point for good solutions where the optimal value of $t^{\star}_{sf} = 1/N$, where N is the number of agents. Like in the Overlap case, the optimal value might not be possible to achieve due to map configuration. Nonetheless, it serves as a valuable metric to assert algorithm performance.

7.5.2 Comparison Between Different Reward Structures

In this section, the main goal will be to analyze the differences in performance and robustness of the different prototype reward functions presented in Section 7.3.3. First, each prototype will be analyzed individually for different gains K, and in the end, the best-performing prototypes will be compared with each other. The analysis will be based on a smaller yet still representative set of maps with two agents.

7.5.2.1 Randomly Sampling Number of Agents During Training

One of the features of the training algorithm presented in Section 7.4.2.1 is to sample the number of agents in every new episode randomly. Figure 7.6, shows a comparison between the performance of two models with the exact same hyperparameters, one trained with a random number of agents and the other with that value fixed at two. Both agents were trained with a maximum of two agents, and the evaluation was done in the same scenario.

These results are very interesting, as the model that was trained in a scenario that was not evaluated performed around 5% better than the model that was trained exclusively in that scenario. It shows that the competencies between tasks can be transferred, and in this case, learning optimal pathing in single-agent scenarios also translates to better results in multi-agent scenarios. These results validate the philosophy of the algorithm, training in scenarios as generic as possible and using different tasks to learn specific skills. From this point onwards, all algorithms will be trained on this framework.

7.5.2.2 Competitive and Cooperative Non-Size Invariant Structure

Now that all agents will be trained with a randomly sampled number of agents in each episode, the next analysis will be on the cooperative Non-Size Invariant (NSI) reward structure, presented in

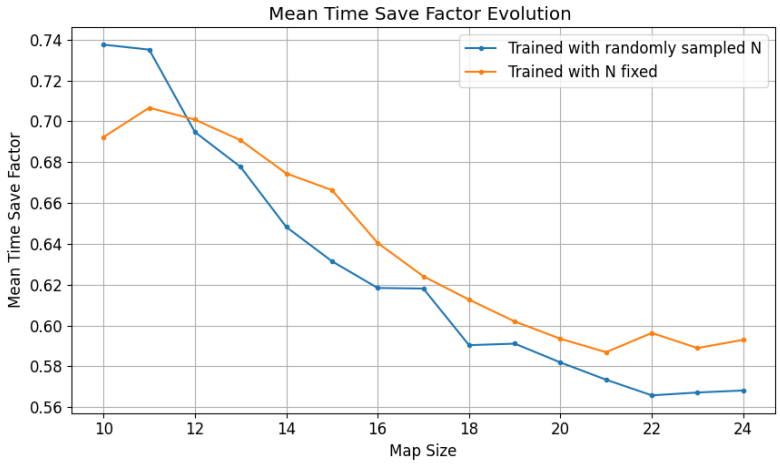


Figure 7.6: Mean Time Save Factor comparison Between Models Trained With Different Strategies.

Section 7.3.3.2. Additionally, the competitive reward structure, presented in Section 7.3.3.1, will also be studied as it is the case where $K = 0$. The agents are all trained following the environments presented in Section 7.4.2.2 except the last environment of Table 7.2. The dataset will consist of maps from size $[10, 24]$ and will all be completed with two agents. The gain of the function will take the values $K = \{0, 0.25, 0.5, 0.75, 1\}$, and the results are presented in Figure 7.8.

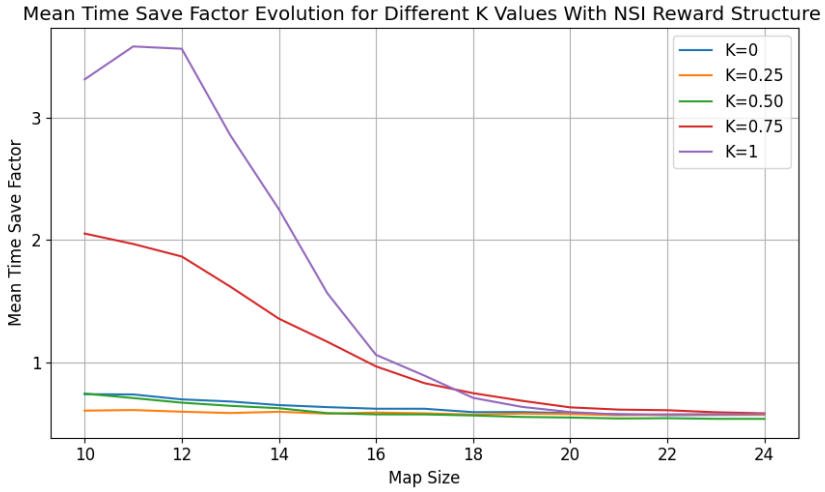


Figure 7.7: Comparison Between Different Gain Values K for the Non-Size Invariant Reward Structure.

One aspect that is immediately clear is that the algorithm is very sensible to changes in values of K. Interestingly, this sensibility is only noticeable in smaller map sizes, suggesting that losing the size invariant property makes the agent lose generalization capabilities. At K values smaller than 0.5, the performance is near-optimal and similar across values, achieving Time-Save Values as close to 0.54, meaning that the map was completed almost twice as fast as the single-agent

scenario. This can be associated with the fact that it is closer to the original function, which is better at generalizing and not overfitting. One interesting result is that the initial reward function $K = 0$ also holds up in the multi-agent scenario.

7.5.2.3 Cooperative Size Invariant Structure

After analyzing the most generic non-size invariant reward structure, the attention is now directed to the cooperative Size Invariant (SI) structure. With this objective, the hyperparameter values $K = \{0.25, 0.5, 0.75\}$ were tested in the same conditions. The $K = 0.5$ serves as the $K = 1/N$ solution as these values are for two agents. Furthermore, $K = 0$ results are the same as in the previous section, being kept only for comparison. The value $K = 1$ would mean that the agent's action would not be considered, and therefore, the results were very poor. It also does not fit in any reward structure, therefore not being analyzed. The obtained results can be visualized in Figure 7.8.

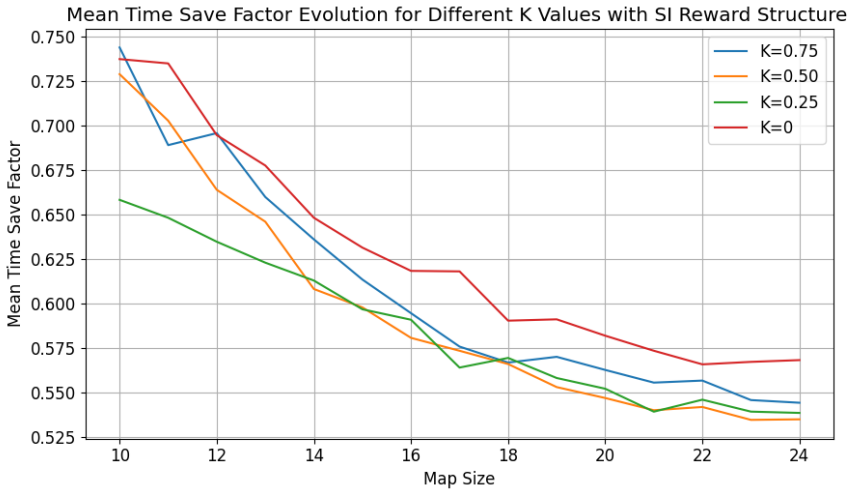


Figure 7.8: Comparison Between Different Gain Values K for the Cooperative Size Invariant Reward Structure.

The size-invariant property of this cooperative reward structure is clearly noticeable. The performance is more uniform across the spectrum, and the overfitting phenomenon is less evident. Furthermore, unlike the previous structure, the algorithm is not nearly as sensible to the gain value K, being that all values generate very similar outcomes. The best results are achieved by the model trained with $K = 0.25$, achieving Time Save Factors as low as 0.53, which is better than any of the Non-Size Invariant scenarios. This choice for the value gives the most importance to the agent itself without completely discarding the other agent. This makes the size invariant property hold better and eases the credit assignment process. The $K = 1/N$ also yields very interesting results, so for better comprehension, it is important to study a case with more agents.

7.5.2.4 Results with $N > 2$

While the analysis gave a good insight into how the models behave when there are two agents, the objective of the algorithm is to deal with any number of agents. This section will analyze how the best reward functions from the previous analysis behave in scenarios with five agents. For this purpose, the model was trained in scenarios with up to five agents, and the results can be observed in Figure 7.9.

From the results, the best models were with the cooperative size invariant reward function with either $K = 0.5$ or $K = 0.25$, although the latter is slightly better. These results show that the model can adapt to scenarios with more agents, yielding mean Time Save factors consistently inferior to 0.30. While the optimal value for these scenarios would still be around 0.20 Time Save Factor, the results still show a speed up of 233% compared to a single agent solution, which is a considerable improvement and is a fair assessment that the architecture can scale up to a larger number of agents.

Figure 7.9: Comparison Between Reward Functions For Scenarios With 5 Agents.

7.5.3 Comprehensive Analysis of The Best Model

After choosing the cooperative SI reward structure with $K = 0.25$, the model was again submitted to training with a maximum number of agents $N = 10$. This subsection will be dedicated to fully analyzing the capabilities of this new model.

7.5.3.1 Direct Comparison With Previous Models

One question that needs to be answered with regard to the model is how well it performs with different values of N. For example, by having a training session with a high number of agents, i.e. N=10, the results with fewer agents might suffer decreases in performance. For this purpose,

Figure 7.10 compares the newly trained model with the previous models trained with $N = 2$ and $N = 5$.

Figure 7.10: Comparision Between Old Models and Newly Trained Model.

From the results, there is a clear degradation in the performance of the newly trained model in the scenario with two agents when compared to the model that was trained specifically for that scenario. On the other hand, in the scenario with five agents, the models are very similar and almost equivalent. This might suggest that by training with higher values of N, the algorithm is more optimized for those types of situations. Nonetheless, the new model is still capable in the scenario with $N = 2$ and is much more versatile since it can deal with other situations. However, if the objective is to have a very optimized and tailored algorithm, it might be worth training it in a less generic manner.

7.5.3.2 Results With Different Number of Agents

Now that a comparison with more tailored models has been made, the focus will shift to a more generic assessment of the performance of the model. To evaluate this performance, the model will be tested on the full dataset with map sizes from [10,30] and for all possible numbers of agents from [2,15]. The results can be visualized in Figure 7.11.

The presented results are very impressive. First of all, the model exhibits a consistent time save factor that is, in general, independent of the size of the map, showcasing the generalization capabilities of the algorithm and its consistency. Another consistent result is the evolution of the performance with respect to the increase in the number of agents. In all tested scenarios, the performance does not degrade with the increase in the number of agents, even in situations that were never trained, like $N = 15$. This shows that the model learned not only to generalize to different map sizes but also to the number of agents, implying that this architecture is scalable to any feasible configuration. Furthermore, the performance is very respectable. While not at an

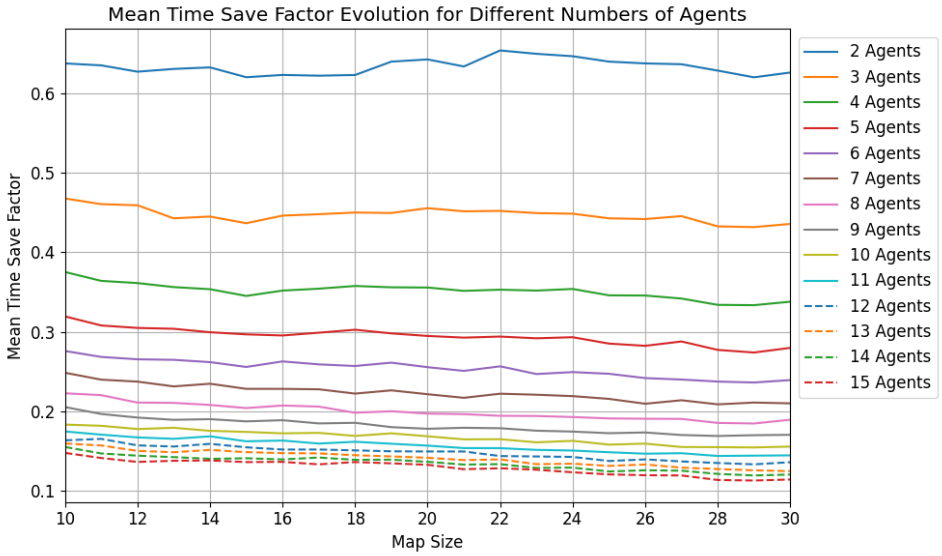


Figure 7.11: Comparision Between Different Number of Agents.

optimal level, the algorithm achieved a mean time-save factor as low as $t_{sf} = 0.15$ in the scenario with ten agents, which would have an optimal solution of only $t_{sf} = 0.10$.

A video showing the algorithm's performance in a 23×23 Map with 5 to 15 agents can be seen in this link.

7.5.3.3 Algorithm Robustness

To evaluate how the algorithm can deal with non-ideal behavior, an additional test was conducted. This test focuses on how the algorithm reacts to a faulty agent. In this scenario, halfway through completing the coverage, one of the agents stops functioning, remaining immobilized until the episode is finished.

The results of the experiment can be visualized in Figure 7.12.

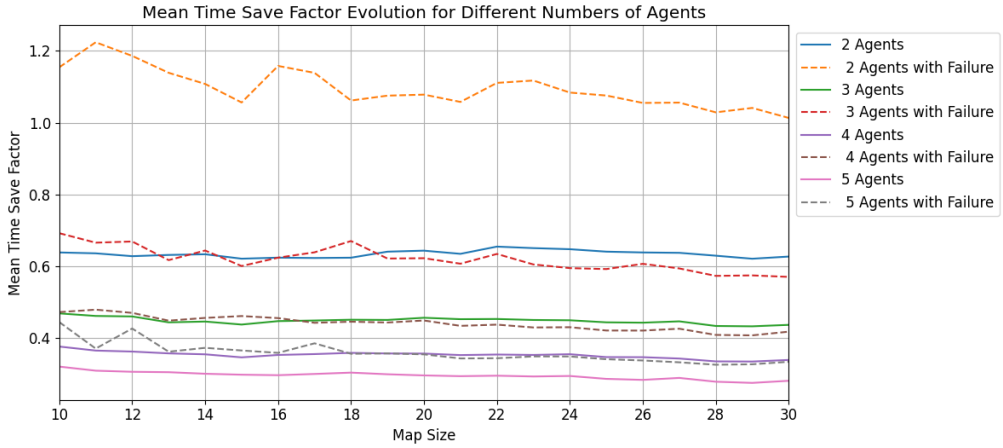


Figure 7.12: Analysis of the robustness of the algorithm when subjected to agent failure.

The figure shows some very interesting results. Firstly, it is notable how the algorithm can deal with one faulty agent, which, despite having a clear downgrade in performance, the results are still satisfactory. One interesting detail is that experiments with $N-1$ fully functioning agents achieve very similar results to the N agents with a faulty one. This shows that the model can adapt and deal with the situation successfully and that the degradation in performance is not much bigger than having one less agent.

7.6 Final Considerations

In this chapter, the Deep Reinforcement Learning algorithm presented in the previous chapter was augmented for Multi-Agent scenarios. The proposal is based on using a state-of-the-art Multi-Agent Reinforcement Learning technique named Parameter Sharing, which enables utilizing the developed modified Rainbow DQN in a multi-agent scenario via a Centralized Training Distributed Execution framework. Once again, the focus of the algorithm was to be as generic and adaptable as possible to multiple scenarios.

First, the problem was formulated as a Decentralized, Partially Observable Markov Decision Process, focusing on adapting the successful POMDP from the previous chapter. The problem was formulated in a generic fashion, giving room for enhancements in the future. Multiple Reward function structures were proposed for the problem, borrowing from cooperative, competitive, and mixed-setting scenarios, and an explanation of the rationale behind each structure is given. One of the objectives of the reward structures is to keep the size invariant propriety, as this gave the single-agent algorithm good generalization capabilities.

The custom-made training algorithm is also used with the same elements as before, with a new addition of randomly sampling the number of agents at the beginning of each episode. This stochastic component proved to be very useful, as it improved performance over multiple scenarios by preventing overfitting and gave the model opportunities to learn optimal pathing in single-agent scenarios, where cooperation is not a concern. This enhancement was made based on the philosophy of curriculum and transfer learning, where the algorithm can learn different competencies from different scenarios and tasks.

The results validate the design choices for this Multi-Agent Reinforcement Learning for Coverage Path Planning. To assert the performance of the agents, the dataset for single-agent evaluation was augmented by only adding $N-1$ agents and keeping the rest of the scenario untouched. The results show that the model can achieve very good performance for any number of agents, even if the algorithm did not train in that specific range. Over the 2500 evaluation maps, the best model obtained consistent results for all numbers of agents and map sizes, reinforcing the idea that the model was able to truly learn how to generalize both covering the map and cooperating with other agents independently of configuration variation. The algorithm was also shown to be robust to adverse situations where agents stopped working.

This chapter is the culmination of all the work developed in this dissertation and delivers all the proposed objectives. It is a contribution to the current state-of-the-art on robotic navigation,

multi-agent reinforcement learning, and the intersection of both domains. Through the results, this chapter showed the viability of using parameter sharing and value-based deep reinforcement learning to deliver solutions to cooperative coverage path planning.

Chapter 8

Conclusions

The primary objective of this dissertation was to develop Reinforcement Learning algorithms for cooperative robotic navigation. The document is structured into incremental steps that lead to the final goal. The initial focus was on presenting the theoretical background necessary to understand Reinforcement Learning solutions for decision-making problems. After exposing the theoretical foundations of Reinforcement Learning, all the algorithms that were used during the dissertation were described

Subsequently, the coverage path planning problem, which is the main robotic navigation task addressed in this work, was studied in detail. Classical solutions to the problem were presented, followed by an analysis of existing reinforcement learning-based approaches. Through this in-depth examination, key features of existing solutions were identified and served as inspiration for the subsequent sections of the document.

One of the most important parts of the dissertation was to develop the software that enabled the development of the algorithms. The first part is developing the environment using the OpenAI Gym guidelines. This environment was made to be as sandbox-like as possible to facilitate the implementation of different features and to be as generic and reusable as possible. Furthermore, generic Tabular and Deep Reinforcement Learning agents were developed with an environment-agnostic philosophy, being able to adapt to any environment as long as the state space and action space are consistent. The developed environment also has default datasets and results to facilitate comparisons and benchmarks of other approaches and to encourage further research on this topic and environment.

After defining the software architecture, the focus is shifted to algorithm development. This effort starts with a Tabular Temporal Differences Reinforcement Learning case study conducted on the online coverage path planning task. In this study, all classical tabular algorithms were benchmarked for this task, and conclusions were drawn regarding time efficiency and performance. Moreover, the final algorithm, which employs both Q-Learning and a crafted heuristic, demonstrated the ability to perform the task solely based on sensor information. This characteristic distinguishes this approach from many other coverage path planning algorithms that heavily rely on explicit map representations and allow it to function as long as the sensor readings are

given. The results also indicated that the agent could generalize to previously unseen maps, even when they are larger and contain more obstacles.

The next step taken in this dissertation was to scale the solution to more recent Deep Reinforcement Learning methodologies. These techniques enable algorithms to work with larger state and action spaces and give an overall much larger generalization capability. With this framework in mind, a Deep Reinforcement Learning-based algorithm was developed to tackle generic robotic navigation tasks.

The robotic navigation problem was formulated as a spectrum instead of singular and independent tasks. The rationale behind most of the developed work is that an intelligent agent that can do a complex task such as coverage path planning should also be able to complete a simpler task like point-to-point path planning. With this formulation, the algorithm was designed to tackle complete coverage path planning, n-point coverage path planning, point-to-point path planning, and active exploration. This is achieved by an adequate formulation of the problem as a Partial Observable Markov Decision Process, with a tailored Observation Space and a hand-crafted reward function that results in a value function that is map size invariant. On top of the attention to formulation, the state-of-the-art value-based reinforcement learning method Rainbow DQN was implemented and adapted to fit the problem. Furthermore, besides this sophisticated approach, a custom-made training algorithm was designed based on curriculum and transfer learning, coupled with other innovative features such as Partial Episode Bootstrapping, environment partial resetting, and saving challenging episodes.

The results from the single-agent algorithm were very exciting. The developed models showed capabilities to perform at a near-optimal level in any map configuration. Furthermore, the agent was successful in learning all the proposed tasks, showing robustness when faced with non-ideal sensor behavior and situations that were never seen during training. When compared with other state-of-the-art approaches, this algorithm clearly outperformed all the other options that were compared, giving confidence that this approach is a viable increment to the current state-of-the-art in coverage path planning. A video of the single-agent algorithm in action can be found in the following link.

After fully studying the single-agent setting, this work made the jump to multi-agent scenarios. The single-agent framework was augmented to deal with the increased complexity. The Partially Observed Markov Decision Process is adapted to its Decentralized version, and the Rainbow DQN algorithm is extended to the multi-agent setting by using the state-of-the-art technique of Parameter Sharing, which introduces a Centralized Training Decentralized Execution framework to the problem.

With the algorithm designed, it was tested on the datasets provided by this dissertation. In this analysis, multiple reward structures and functions were considered and compared until a satisfying configuration for the model was achieved. With the defined model, a series of tests were conducted to assess its performance. The results show that the model was able to learn to cooperate with multiple agents, even in scenarios with up to fifteen independent robots. The performance was promising, showing significant improvements over the single-agent approach, paving the way for

further work in this domain. A video of the multi-agent algorithm can visualized in the following link.

In conclusion, this dissertation makes several contributions to learning-based robotic navigation by developing various state-of-the-art algorithms and presenting a novel perspective on robotic navigation as a whole. Additionally, it contributes to the scientific community through an open-source reinforcement learning environment and multiple benchmark datasets for diverse robotic navigation tasks.

8.1 Future Work

Reflecting on the developed work and the conclusions that were drawn, the author identifies the following areas that can be developed or improved in future work:

- **Expand the Reinforcement Learning Environment:** While the developed environment is generic and can be used to implement any robotic navigation task in a 2D Grid World, it still lacks variety. The implemented tasks are limited to the ones on the proposed coverage path planning spectrum. This leaves opportunities to implement other tasks, such as long-term coverage planning or taking into account different coverage priorities. Another thing that can be improved is adding more sensors and agent kinematics.

- **More Realistic Sensor Model:** The sensor model should be improved to include a better noise model. For example, instead of being a binary of the sensor working or not, the sensor could give false or noisy readings.

- **Dynamic Obstacles:** The implementation of the Rainbow DQN with frame stacking is a great opportunity to add dynamic obstacles to the environment. The dynamics will not be much different from other agents, and the algorithm has already proven to be capable of avoiding collisions.

- **Improve Observation Space:** While the designed observation spaces were proven to be successful, all the mentioned improvements would require a better observation space. In this case, it is important to keep the philosophy of maintaining the most generic formulation possible.

- **Policy-Based Reinforcement Learning Methods:** In general, the literature lacks policy-based approaches to coverage path planning. Implementing such a method could be interesting to compare different RL approaches to the same problem.

- **Use Specific Multi-Agent-Reinforcement Learning Methods:** This work uses a Rainbow DQN with Parameter Sharing for its multi-agent solution. While it proved successful, a more tailored and specific multi-agent method might achieve better results.

- **Improve Agent Communication and Synchronization :** The implemented synchronization mechanism on the multi-agent problem is simple. All agents take actions simultaneously and independently of each other. In order to make a more realistic approach, a communication and synchronization system could be developed.

References

[1] Fatemeh Afghah, Abolfazl Razi, Jacob Chakareski, and Jonathan Ashdown. Wildfire monitoring in remote areas using autonomous unmanned aerial vehicles. *INFOCOM 2019 - IEEE Conference on Computer Communications Workshops, INFOCOM WKSHPS 2019*, pages 835–840, 4 2019. doi:10.1109/INFCOMW.2019.8845309.

[2] Yaodong Yang and Jun Wang. An overview of multi-agent reinforcement learning from game theoretical perspective. *ArXiv*, abs/2011.00583, 2020.

[3] David Silver. Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/, 2015.

[4] Francisco Soares Pinto da Silva Neves. Reinforcement learning for robotic navigation in obstacle scattered environments, 2021.

[5] Ngan Le, Vidhiwar Rathour, Kashu Yamazaki, Khoa Luu, and Marios Savvides. Deep reinforcement learning in computer vision: a comprehensive survey. *Artificial Intelligence Review*, 55, 09 2021. doi:10.1007/s10462-021-10061-9.

[6] Enric Galceran and Marc Carreras. A survey on coverage path planning for robotics. *Robotics and Autonomous Systems*, 61:1258–1276, 12 2013. doi:10.1016/J.ROBOT.2013.09.004.

[7] Tauã M. Cabreira, Lisane B. Brisolara, and R. Ferreira Paulo. Survey on coverage path planning with unmanned aerial vehicles. *Drones*, 3:1–38, 3 2019. doi:10.3390/DRONES3010004.

[8] Chee Sheng Tan, Rosmiwati Mohd-Mokhtar, and Mohd Rizal Arshad. A comprehensive review of coverage path planning in robotics using classical and heuristic algorithms. *IEEE Access*, 9:119310–119342, 2021. doi:10.1109/ACCESS.2021.3108177.

[9] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *International Journal of Robotics Research*, 32:1238–1274, 9 2013. doi:10.1177/0278364913495721.

[10] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature 2015 521:7553*, 521:436–444, 5 2015. URL: https://www.nature.com/articles/nature14539, doi:10.1038/nature14539.

[11] Gerald Tesauro et al. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[12] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature 2016 529:7587*, 529:484–489, 1 2016. URL: https://www.nature.com/articles/nature16961https://www.nature.com/articles/nature16961%7D, doi:10.1038/nature16961.

[13] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362:1140–1144, 12 2018. URL: https://www.science.org/doi/10.1126/science.aar6404, doi:10.1126/SCIENCE.AAR6404/SUPPL_FILE/AAR6404_DATAS1.ZIP.

[14] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature 2019 575:7782*, 575:350–354, 10 2019. URL: https://www.nature.com/articles/s41586-019-1724-z, doi:10.1038/s41586-019-1724-z.

[15] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. 12 2019. URL: https://arxiv.org/abs/1912.06680v1, doi:10.48550/arxiv.1912.06680.

[16] Anirudh Krishna Lakshmanan, Rajesh Elara Mohan, Balakrishnan Ramalingam, Anh Vu Le, Prabahar Veerajagadeshwar, Kamlesh Tiwari, and Muhammad Ilyas. Complete coverage path planning using reinforcement learning for tetromino based cleaning and maintenance robot. *Automation in Construction*, 112, 4 2020. doi:10.1016/j.autcon.2020.103078.

[17] J. Jin and Lie Tang. Optimal coverage path planning for arable farming on 2d surfaces. *Transactions of the ASABE*, 53:283–295, 2010.

[18] Manickam Ramasamy and Debasish Ghose. Learning-based preferential surveillance algorithm for persistent surveillance by unmanned aerial vehicles. *2016 International Conference on Unmanned Aircraft Systems, ICUAS 2016*, pages 1032–1040, 6 2016. doi:10.1109/ICUAS.2016.7502678.

[19] Carmelo Di Franco and Giorgio Buttazzo. Coverage path planning for uavs photogrammetry with energy and resolution constraints. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 83:445–462, 9 2016. doi:10.1007/S10846-016-0348-X.

[20] Y. Gao, G. Jin, Y. Guo, G. Zhu, Q. Yang, and K. Yang. Weighted area coverage of maritime joint search and rescue based on multi-agent reinforcement learning. pages 593–597, 2019. doi:10.1109/IMCEC46724.2019.8984116.

[21] Javad Heydari, Olimpiya Saha, and Viswanath Ganapathy. Reinforcement learning-based coverage path planning with implicit cellular decomposition. 10 2021. URL: https://arxiv.org/abs/2110.09018v1, doi:10.48550/arxiv.2110.09018.

[22] Sina Sharif Mansouri, Christoforos Kanellakis, Emil Fresk, Dariusz Kominiak, and George Nikolakopoulos. Cooperative coverage path planning for visual inspection. *Control Engineering Practice*, 74:118–131, 5 2018. doi:10.1016/J.CONENGPRAC.2018.03.002.

[23] Tua A. Tamba. Optimizing the area coverage of networked uavs using multi-agent reinforcement learning. pages 197–201. Institute of Electrical and Electronics Engineers Inc., 2021. doi:10.1109/ICA52848.2021.9625676.

[24] San-Miguel-Ayanz J, Durrant T, Boca R, Maianti P, Liberta' G, Artes Vivancos T, Jacome Felix Oom D, Branco A, De Rigo D, Ferrari D, Pfeiffer H, Grecchi R, Onida M, and Loffler P. Forest fires in europe, middle east and north africa 2021. (KJ-NA-31-269-EN-N (online),KJ-NA-31-269-EN-C (print)), 2022. doi:10.2760/34094(online),10.2760/058256(print).

[25] Esmaeil Seraj and Matthew Gombolay. Coordinated control of uavs for human-centered active sensing of wildfires. *Proceedings of the American Control Conference*, 2020-July:1845–1852, 6 2020. URL: https://arxiv.org/abs/2006.07969v1, doi:10.48550/arxiv.2006.07969.

[26] Kyle D. Julian and Mykel J. Kochenderfer. Distributed wildfire surveillance with autonomous aircraft using deep reinforcement learning. *Journal of Guidance, Control, and Dynamics*, 42:1768–1778, 10 2018. URL: https://arxiv.org/abs/1810.04244v1, doi:10.48550/arxiv.1810.04244.

[27] Claudio Piciarelli and Gian Luca Foresti. Drone patrolling with reinforcement learning. *ACM International Conference Proceeding Series*, 9 2019. doi:10.1145/3349801.3349805.

[28] Mirco Theile, Harald Bayerlein, Richard Nai, David Gesbert, and Marco Caccamo. Uav path planning using global and local map information with deep reinforcement learning. *2021 20th International Conference on Advanced Robotics, ICAR 2021*, pages 539–546, 2021. doi:10.1109/ICAR53236.2021.9659413.

[29] José Pedro Carvalho and A. Pedro Aguiar. A reinforcement learning based online coverage path planning algorithm. In *2023 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 81–86, 2023. doi:10.1109/ICARSC58346.2023.10129591.

[30] R.E. Uhrig. Introduction to artificial neural networks. In *Proceedings of IECON '95 - 21st Annual Conference on IEEE Industrial Electronics*, volume 1, pages 33–37 vol.1, 1995. doi:10.1109/IECON.1995.483329.

[31] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958. doi:10.1037/h0042519.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[33] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015. URL: http://arxiv.org/abs/1504.08083, arXiv:1504.08083.

[34] Sergios Theodoridis. *Machine Learning: A Bayesian and Optimization Perspective*. Academic Press, Inc., USA, 1st edition, 2015.

[35] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998. doi:10.1016/S0004-3702(98)00023-X.

[36] Christos G Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer, 2 edition, 2008. doi:10.1007/978-0-387-68612-7.

[37] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60:503–515, 1954. doi:10.1090/S0002-9904-1954-09848-8.

[38] P J H Schoemaker. *Experiments on Decisions under Risk: The Expected Utility Hypothesis*. Springer Netherlands, 2013. URL: https://books.google.com.ni/books?id=fuPuCAAAQBAJ.

[39] Hado van Hasselt, Matteo Hessel, and John Aslanides. When to use parametric models in reinforcement learning? *CoRR*, abs/1906.05243, 2019. URL: http://arxiv.org/abs/1906.05243, arXiv:1906.05243.

[40] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[41] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning 1988 3:1*, 3:9–44, 8 1988. URL: https://link.springer.com/article/10.1007/BF00115009, doi:10.1007/BF00115009.

[42] Christopher J.C.H. Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 1992. doi:10.1023/A:1022676722315.

[43] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 9 2015. URL: https://arxiv.org/abs/1509.02971v6, doi:10.48550/arxiv.1509.02971.

[44] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning 1992 8:3*, 8:229–256, 5 1992. URL: https://link.springer.com/article/10.1007/BF00992696, doi:10.1007/BF00992696.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov Openai. Proximal policy optimization algorithms. 7 2017. URL: https://arxiv.org/abs/1707.06347v2, doi:10.48550/arxiv.1707.06347.

[46] Volodymyr Mnih, Adria Puigdomenech Badia, Lehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *33rd International Conference on Machine Learning, ICML 2016*, 4:2850–2869, 2 2016. URL: https://arxiv.org/abs/1602.01783v2, doi:10.48550/arxiv.1602.01783.

[47] Ziyu Wang, Volodymyr Mnih, Victor Bapst, Remi Munos, Nicolas Heess, Koray Kavukcuoglu, and Nando De Freitas. Sample efficient actor-critic with experience replay. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 11 2016. URL: https://arxiv.org/abs/1611.01224v2, doi:10.48550/arxiv.1611.01224.

[48] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature 2015 518:7540*, 518:529–533, 2 2015. URL: https://www.nature.com/articles/nature14236, doi:10.1038/nature14236.

[49] Hessel et al. Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018. doi:10.1609/aaai.v32i1.11796.

[50] John Tsitsiklis and Benjamin Van Roy. Analysis of temporal-diffference learning with function approximation. *Advances in neural information processing systems*, 9, 1996.

[51] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 12 2013. URL: https://arxiv.org/abs/1312.5602v1, doi:10.48550/arxiv.1312.5602.

[52] Peter R. Wurman et al. Outracing champion gran turismo drivers with deep reinforcement learning. *Nature 2022 602:7896*, 602:223–228, 2 2022. doi:10.1038/s41586-021-04357-7.

[53] Hado Hasselt. Double q-learning. volume 23. Curran Associates, Inc., 2010. URL: https://proceedings.neurips.cc/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf.

[54] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013. URL: https://doi.org/10.1613%2Fjair.3912, doi:10.1613/jair.3912.

[55] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Mach. Learn.*, 8(3–4):293–321, may 1992. doi:10.1007/BF00992699.

[56] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *30th AAAI Conference on Artificial Intelligence, AAAI 2016*, pages 2094–2100, 2016. doi:10.1609/aaai.v30i1.10295.

[57] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. 11 2015. URL: https://arxiv.org/abs/1511.05952, doi:10.48550/arxiv.1511.05952.

[58] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. pages 1995–2003. JMLR.org, 2016.

[59] Meire Fortunato et al. Noisy networks for exploration. 2017. URL: https://arxiv.org/abs/1706.10295.

[60] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *CoRR*, abs/1707.06887, 2017. URL: http://arxiv.org/abs/1707.06887, arXiv:1707.06887.

[61] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms, 2021. arXiv:1911.10635.

[62] Georgios Papoudakis, Filippos Christianos, Arrasy Rahman, and Stefano V. Albrecht. Dealing with non-stationarity in multi-agent deep reinforcement learning. *ArXiv*, abs/1906.04737, 2019.

[63] Robin Soetens, René Van De Molengraft, and Bernardo Cunha. Robocup msl - history, accomplishments, current status and challenges ahead. *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)*, 8992:624–635, 2015. doi:10.1007/978-3-319-18615-3_51/TABLES/1.

[64] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder de Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning, 2018. arXiv:1803.11485.

[65] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments, 2020. arXiv:1706.02275.

[66] Jayesh K. Gupta, Maxim Egorov, and Mykel J. Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *AAMAS Workshops*, 2017.

[67] J. K. Terry, Nathaniel Grammel, Sanghyun Son, and Benjamin Black. Parameter sharing for heterogeneous agents in multi-agent reinforcement learning, 2022. arXiv:2005.13625.

[68] Zuo Llang Cao, Yuyu Huang, and Ernest L. Hall. Region filling operations with random obstacle avoidance for mobile robots. *Journal of Robotic Systems*, 5:87–102, 1988. URL: https://www.researchgate.net/publication/229613862_Region_filling_operations_with_random_obstacle_avoidance_for_mobile_robots, doi:10.1002/ROB.4620050202.

[69] Howie Choset. Coverage for robotics - a survey of recent results. *Annals of Mathematics and Artificial Intelligence*, 31:113–126, 2001. URL: https://link.springer.com/article/10.1023/A:1016639210559, doi:10.1023/A:1016639210559/METRICS.

[70] Sedat Dogru and Lino Marques. Eco-cpp: Energy constrained online coverage path planning. *Robotics and Autonomous Systems*, 157:104242, 2022. doi:https://doi.org/10.1016/j.robot.2022.104242.

[71] B. Nasirian, M. Mehrandezh, and F. Janabi-Sharifi. Efficient coverage path planning for mobile disinfecting robots using graph-based representation of environment. *Frontiers in Robotics and AI*, 8, 3 2021. doi:10.3389/FROBT.2021.624333.

[72] Luis Piardi, José Lima, Ana I. Pereira, and Paulo Costa. Coverage path planning optimization based on q-learning algorithm. volume 2116. American Institute of Physics Inc., 7 2019. doi:10.1063/1.5114220.

[73] Jian Xiao, Gang Wang, Ying Zhang, and Lei Cheng. A distributed multi-agent dynamic area coverage algorithm based on reinforcement learning. *IEEE Access*, 8:33511–33521, 2020. doi:10.1109/ACCESS.2020.2967225.

[74] Dong Ki Noh, Woo Ju Lee, Hyoung Rock Kim, Il Soo Cho, In Bo Shim, and Seung Min Baek. Adaptive coverage path planning policy for a cleaning robot with deep reinforcement learning. *Digest of Technical Papers - IEEE International Conference on Consumer Electronics*, 2022-January, 2022. doi:10.1109/ICCE53296.2022.9730307.

[75] Taua M. Cabreira, Carmelo Di Franco, Paulo R. Ferreira, and Giorgio C. Buttazzo. Energy-aware spiral coverage path planning for uav photogrammetric applications. *IEEE Robotics and Automation Letters*, 3:3662–3668, 10 2018. doi:10.1109/LRA.2018.2854967.

[76] Howie Choset and Philippe Pignon. Coverage path planning: The boustrophedon cellular decomposition. *Field and Service Robotics*, pages 203–209, 1998. URL: https://link.springer.com/chapter/10.1007/978-1-4471-1273-0_32, doi:10.1007/978-1-4471-1273-0_32.

[77] Ercan U. Acar, Howie Choset, Alfred A. Rizzi, Prasad N. Atkar, and Douglas Hull. Morse decompositions for coverage tasks. *The International Journal of Robotics Research*, 21:331–344, 4 2002. doi:10.1177/027836402320556359.

[78] Ercan U. Acar and Howie Choset. Sensor-based coverage of unknown environments: Incremental construction of morse decompositions. *The International Journal of Robotics Research*, 21:345–366, 4 2002. doi:10.1177/027836402320556368.

[79] Ercan U. Acar and Howie Choset. Robust sensor-based coverage of unstructured environments. *IEEE International Conference on Intelligent Robots and Systems*, 1:61–68, 2001. doi:10.1109/IROS.2001.973337.

[80] Alexander Zelinsky, Ray A Jarvis, JC Byrne, Shinichi Yuta, et al. Planning paths of complete coverage of an unstructured environment by a mobile robot. In *Proceedings of international conference on advanced robotics*, volume 13, pages 533–538, 1993.

[81] Olimpiya Saha, Guohua Ren, Javad Heydari, Viswanath Ganapathy, and Mohak Shah. Online area covering robot in unknown dynamic environments. In *2021 7th International Conference on Automation, Robotics and Applications (ICARA)*, pages 38–42, 2021. doi:10.1109/ICARA51699.2021.9376498.

[82] Olimpiya Saha, Guohua Ren, Javad Heydari, Viswanath Ganapathy, and Mohak Shah. Deep reinforcement learning based online area covering autonomous robot. In *2021 7th International Conference on Automation, Robotics and Applications (ICARA)*, pages 21–25, 2021. doi:10.1109/ICARA51699.2021.9376477.

[83] Harald Bayerlein, Mirco Theile, Marco Caccamo, and David Gesbert. Uav path planning for wireless data harvesting: A deep reinforcement learning approach. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, pages 1–6, 2020. doi:10.1109/GLOBECOM42002.2020.9322234.

[84] Harald Bayerlein, Mirco Theile, Marco Caccamo, and David Gesbert. Multi-uav path planning for wireless data harvesting with deep reinforcement learning. *IEEE Open Journal of the Communications Society*, 2:1171–1187, 2021. doi:10.1109/OJCOMS.2021.3081996.

[85] Mirco Theile, Harald Bayerlein, Richard Nai, David Gesbert, and Marco Caccamo. Uav coverage path planning under varying power constraints using deep reinforcement learning. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1444–1449, 2020. doi:10.1109/IROS45743.2020.9340934.

[86] Samuel Yanes Luis, Daniel Gutiérrez Reina, and Sergio L. Toral Marín. A deep reinforcement learning approach for the patrolling problem of water resources through autonomous surface vehicles: The ypacarai lake case. *IEEE Access*, 8:204076–204093, 2020. doi:10.1109/ACCESS.2020.3036938.

[87] Samuel Yanes Luis, Daniel Gutiérrez Reina, and Sergio L. Toral Marín. A multiagent deep reinforcement learning approach for path planning in autonomous surface vehicles: The ypacaraí lake patrolling case. *IEEE Access*, 9:17084–17099, 2021. doi:10.1109/ACCESS.2021.3053348.

[88] Aleksandr Ianenko, Alexander Artamonov, Georgii Sarapulov, Alexey Safaraleev, Sergey Bogomolov, and Dong Ki Noh. Coverage path planning with proximal policy optimization in a grid-based environment. *Proceedings of the IEEE Conference on Decision and Control*, 2020-December:4099–4104, 12 2020. doi:10.1109/CDC42340.2020.9304030.

[89] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. arXiv:1606.01540.

[90] Matteo Marchi, Jonathan Bunton, Yskandar Gas, Bahman Gharesifard, and Paulo Tabuada. Sharp performance bounds for pasta. *IEEE Control Systems Letters*, 7:2401–2406, 2023. doi:10.1109/LCSYS.2023.3285514.

[91] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time limits in reinforcement learning. *CoRR*, abs/1712.00378, 2017. URL: http://arxiv.org/abs/1712.00378, arXiv:1712.00378.

[92] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps, 2017. arXiv:1507.06527.

[93] Maxim Egorov. Deep reinforcement learning with pomdps. *Tech. Rep.(Technical Report, Stanford University, 2015), Tech. Rep.*, 2015.

[94] A. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *International Conference on Machine Learning*, 1999. URL: https://api.semanticscholar.org/CorpusID:5730166.

[95] Dominik Schmidt and Thomas Schmied. Fast and data-efficient training of rainbow: an experimental study on atari, 2021. arXiv:2111.10247.

[96] Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning, 2019. arXiv:1803.02811.

[97] Omar Boufous. Deep reinforcement learning for complete coverage path planning in unknown environments. Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2020.

[98] Young-Ho Choi, Tae-Kyeong Lee, Sanghoon Baek, and Se-Young Oh. Online complete coverage path planning for mobile robots based on linked spiral paths using constrained inverse distance transform. *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5788–5793, 2009. URL: https://api.semanticscholar.org/CorpusID:10681677.

[99] Tae-Kyeong Lee, Sanghoon Baek, Se-Young Oh, and Young-Ho Choi. Complete coverage algorithm based on linked smooth spiral paths for mobile robots. *2010 11th International Conference on Control Automation Robotics & Vision*, pages 609–614, 2010. URL: https://api.semanticscholar.org/CorpusID:17670933.

[100] Hoang Huu Viet, Viet-Hung Dang, Md Nasir Uddin Laskar, and TaeChoong Chung. Ba*: an online complete coverage algorithm for cleaning robots. *Applied Intelligence*, 39:217–235, 2013. URL: https://api.semanticscholar.org/CorpusID:14928030.

[101] Enrique González, Maricio Alarcon, Paula Aristizabal, and Carlos Parra. Bsa: a coverage algorithm. *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, 2:1679–1684 vol.2, 2003. URL: https://api.semanticscholar.org/CorpusID:45665154.

[102] Junnan Song and Shalabh Gupta. Epsilon*: An online coverage path planning algorithm. *IEEE Transactions on Robotics*, 34:526–533, 2018. URL: https://api.semanticscholar.org/CorpusID:4953256.

[103] Xin Chen, Thomas M. Tucker, Thomas R. Kurfess, and Richard W. Vuduc. Adaptive deep path: Efficient coverage of a known environment under various configurations. *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3549–3556, 2019. URL: https://api.semanticscholar.org/CorpusID:210972306.