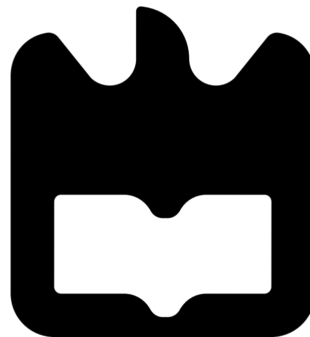
Universidade de Aveiro
2021

Marco António
Gomes Silva

Navegação Multi-Objetivo de um Robô Móvel Usando Aprendizagem por Reforço Hierárquica

Multi-Goal Navigation of a Mobile Robot Using Hierarchical Reinforcement Learning



Universidade de Aveiro
2021

Marco António
Gomes Silva

Navegação Multi-Objetivo de um Robô Móvel Usando Aprendizagem por Reforço Hierárquica

Multi-Goal Navigation of a Mobile Robot Using Hierarchical Reinforcement Learning

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Dr. Filipe Miguel Teixeira Pereira da Silva, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri / the jury

presidente / president

Professor Doutor Joaquim João Estrela Ribeiro Silvestre Madeira
Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Professor Doutor João Paulo Morais Ferreira
Professor Adjunto do Instituto Superior de Engenharia de Coimbra (arguente)

Professor Doutor Filipe Miguel Teixeira Pereira da Silva
Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Queria agradecer ao Professor Filipe Silva pelo excelente acompanhamento e pelo tempo despendido durante a realização desta dissertação.

Desejo também agradecer aos meus pais e irmã, e a todos os meus amigos pelo apoio durante todos estes anos.

Finalmente, quero dar um especial agradecimento à minha namorada por estar sempre presente e por me puxar para cima nos momentos mais difíceis.

I would like to thank Professor Filipe Silva for the excellent support and for the time spent on this dissertation.

I also want to thank my parents and sister, and all my friends for their support over the years.

Finally, a special thanks to my girlfriend for always being there and for pulling me up in the toughest times.

Palavras-Chave

Robótica Móvel; Representação Topológica; Navegação Multi-Objetivo; Aprendizagem Por Reforço; Estrutura Hierárquica; Ambiente Tipo Labirinto

Resumo

Atualmente, há um crescente interesse no desenvolvimento de tecnologias de navegação autónoma para aplicações em ambientes domésticos, urbanos e industriais. Ferramentas de Aprendizagem Automática, como redes neurais, aprendizagem por reforço e aprendizagem profunda têm sido a escolha principal para resolver muitos problemas associados à navegação autónoma de robôs móveis. Esta dissertação tem como foco principal a solução do problema de navegação de robôs móveis em ambientes tipo labirínto com múltiplos objetivos. O ponto central aqui é aplicar uma estrutura hierárquica de algoritmos de aprendizagem por reforço (Q-Learning e R-Learning) a um robô num ambiente contínuo para que ele possa navegar num labirinto. Tanto o espaço de estados quanto o espaço de ações são obtidos através da discretização dos dados recolhidos pelo robô para evitar que estes sejam demasiado extensos. A implementação é feita com uma abordagem hierárquica, que é uma estrutura que permite dividir a complexidade do problema em vários subproblemas mais fáceis, ficando com um conjunto de tarefas de baixo-nível seguido por um de alto-nível. O desempenho do robô é avaliado em dois ambientes tipo labirinto, mostrando que a abordagem hierárquica é uma solução bastante viável para reduzir a complexidade do problema. Além disso, dois cenários diferentes são apresentados: uma situação de multi-objetivo onde o robô navega por múltiplos objetivos usando a representação topológica do ambiente e a experiência memorizada durante a aprendizagem e uma situação de comportamento dinâmico onde o robô deve adaptar suas políticas de acordo com os mudanças que acontecem no ambiente (como caminhos bloqueados). No final, ambos os cenários foram realizados com sucesso e concluiu-se que uma abordagem hierárquica tem muitas vantagens quando comparada a uma abordagem de aprendizagem por reforço clássica.

Keywords

Mobile Robotics; Topological Representation; Multi-goal Navigation; Reinforcement Learning; Hierarchical Structure; Maze-Like Environment

Abstract

Currently, there is a growing interest in the development of autonomous navigation technologies for applications in domestic, urban and industrial environments. Machine Learning tools such as neural networks, reinforcement learning and deep learning have been the main choice to solve many problems associated with autonomous mobile robot navigation. This dissertation mainly focus on solving the problem of mobile robot navigation in maze-like environments with multiple goals. The center point here is to apply a hierarchical structure of reinforcement learning algorithms (Q-Learning and R-Learning) to a robot in a continuous environment so that it can navigate in a maze. Both the state-space and the action-space are obtained by discretizing the data collected by the robot in order to prevent them from being too large. The implementation is done with a hierarchical approach, which is a structure that allows to split the complexity of the problem into many easier sub-problems, ending up with a set of lower-level tasks followed by a higher-level one. The robot performance is evaluated in two maze-like environments, showing that the hierarchical approach is a very feasible solution to reduce the complexity of the problem. Besides that, two more scenarios are presented: a multi-goal situation where the robot navigates across multiple goals relying on the topological representation of the environment and the experience memorized during learning and a dynamic behaviour situation where the robot must adapt its policies according to the changes that happen in the environment (such as blocked paths). In the end, both scenarios were successfully accomplished and it has been concluded that a hierarchical approach has many advantages when compared to a classic reinforcement learning approach.

Contents

Contents i

List of Figures iii

List of Tables v

Acronyms vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2
- 1.3 Document Outline . . . 2

2 Background 5
- 2.1 Reinforcement Learning . . . 5
  - 2.1.1 Elements of Reinforcement Learning . . . 5
  - 2.1.2 Learning Methods . . . 7
  - 2.1.3 Q-Learning Application Example - GridWorld . . . 11
- 2.2 Hierarchical Reinforcement Learning . . . 17
  - 2.2.1 Options . . . 18
  - 2.2.2 Hierarchies of Abstract Machines . . . 19
  - 2.2.3 MAXQ . . . 20
- 2.3 Mobile Robot Navigation . . . 21
  - 2.3.1 Mobile Robots . . . 21
  - 2.3.2 Reinforcement Learning in Mobile Robotics . . . 23
- 2.4 Final Remarks . . . 26

3 Low-Level Exploratory Behaviour 27
- 3.1 Programming Environment and Software Tools . . . 27
  - 3.1.1 Webots Robot Simulator . . . 28
  - 3.1.2 Mobile Robot and Environment . . . 28
- 3.2 State-Action Discretization . . . 30
- 3.3 R-Learning Algorithm . . . 31
- 3.4 Reward Specification . . . 33
  - 3.4.1 Corridor . . . 34
  - 3.4.2 Corner . . . 36
  - 3.4.3 Doors . . . 39

3.5 Performance Evaluation . . . 41
3.6 Final Remarks . . . 48

4 Hierarchical Robot Navigation Approach **49**
4.1 Hierarchical Decomposition . . . 49
4.2 Higher-Level Reinforcement Learning (RL) Problem . . . 51
4.2.1 RL Problem Formulation . . . 51
4.2.2 Learning Evaluation . . . 53
4.2.3 Execution for a Single Goal . . . 54
4.3 Multi-Goal Navigation . . . 56
4.4 Dynamic Behaviour . . . 59
4.5 Final Remarks . . . 61

5 Conclusions and Future Work **63**
5.1 Final Conclusions . . . 63
5.2 Future Work . . . 64

References **65**

Appendices **69**

A Sensor Values Conversion to Distances **70**

B Artificial Neural Network for Robot Localization **74**
B.1 Simulation Environment and Data Extraction . . . 74
B.1.1 Data Pre-processing . . . 76
B.2 Model . . . 76
B.3 Results . . . 78

List of Figures

2.1 Agent-Environment interaction. 7
2.2 The 4 mazes used in the agent training. 14
2.3 Hyperparameters evaluation. 15
2.4 Heat maps in Maze A. 16
2.5 Final result of the agent training. 17
2.6 Unmanned Ground Vehicles (UGVs). 22
2.7 Unmanned Aerial Vehicles (UAVs). 22
2.8 Autonomous Underwater Vehicles (AUVs). 23

3.1 Robots analyzed. 29
3.2 Example of environment. 30
3.3 Corridor reward function representation. 34
3.4 Three possible scenarios. 35
3.5 Corridor maze. 35
3.6 Corridor special situations. 36
3.7 Corner sections and axis transformation. 37
3.8 Corner reward function representation. 38
3.9 Types of doors. 40
3.10 Four-way symmetry in Front-Left-Right door. 40
3.11 Robot convergence to center of corridor. 41
3.12 Robot deviation from center of corridor. 42
3.13 Corner trajectories with training. 42
3.14 Corner trajectories with table transformation. 43
3.15 Maze in T-shape to evaluate performance on doors. 44
3.16 Robot trajectory on maze with all doors combined. 44
3.17 Robot trajectories in each door. 45
3.18 Maze for low-level evaluation. 46
3.19 Actions on doors evaluation. 47

4.1 The hierarchical approach focuses on a sequence of sub-policies that appear both during training and execution. 50
4.2 Mazes used to train and evaluate high-level performance (Maze 1 on the left; Maze 2 on the right). 52
4.3 Most visited states on Maze 1 (left) and Maze 2 (right). 53
4.4 Steps per episode on Maze 1 (left) and Maze 2 (right). 54
4.5 Reward sum per episode on Maze 1 (left) and Maze 2 (right). 54

4.6 Superposition of the robot's trajectories to four different starting positions in Maze 1 (top) and Maze 2 (bottom). 55
4.7 Multi-goal trajectories w/o action to invert direction (top) and with (bottom). . 58
4.8 Top right corner blocked maze 2. 60
4.9 Robot normal trajectories (top) and adaptation to blocked path (bottom). . . . 61

A.1 Webots equations. 70
A.2 Raw voltage relation to theoretical distance. 71
A.3 Curve linearization. 72
A.4 Equation validation. 72
A.5 System of 3 equations vs Webots equations. 73

B.1 1 square meter arena in Webots Simulator where data were collected. 75
B.2 Neural networks using dropout stochastic regularization. 77
B.3 Analysis of two implemented models. 78
B.4 Accuracy and Loss with final model configuration, 1500 epochs and adaptive learning rate. 79

List of Tables

3.1 Webots vs Gazebo main aspects. 28
3.2 Sensors discretization levels. 31
3.3 Actions discretization (wheel rotation speed in rad/s). 31
3.4 Robot low-level efficiency. 46

Acronyms

ACM Adjoining Cell Mapping.

AdaGrad Adaptive Gradient Algorithm.

AI Artificial Intelligence.

ANN Artificial Neural Network.

API Application Programming Interface.

AUV Autonomous Underwater Vehicle.

DP Dynamic Programming.

EPFL École Polytechnique Fédérale de Lausanne.

GPI Generalized Policy Iteration.

HAM Hierarchy of Abstract Machine.

HRL Hierarchical Reinforcement Learning.

HSMQ Hierarchical Semi-Markov Q-Learning.

IDE Integrated Development Environment.

LEDs Light Emitting Diodes.

MCQ-L Modified Connectionist Q-Learning.

MDP Markov Decision Process.

ML Machine Learning.

MSE Mean Squared Error.

ODE Open Dynamics Engine.

OS Operating System.

OSFR Open Source Robotics Foundation.

ReLU Rectified Linear Unit.

RL Reinforcement Learning.

RMSProp Root Mean Square Propagation.

ROV Remotely Operated Underwater Vehicle.

SARSA State-Action-Reward-State-Action.

SLAM Simultaneous Localization And Mapping.

SMDP Semi-Markov Decision Process.

TCP/IP Transmission Control Protocol/Internet Protocol.

TD Temporal-Difference.

UAS Unmanned Aerial System.

UAV Unmanned Aerial Vehicle.

UGV Unmanned Ground Vehicle.

UUV Unmanned Underwater Vehicle.

Chapter 1

Introduction

1.1 Motivation

In recent decades, there has been an exponential growth in technology applications in domestic, urban and industrial environments which allows for automation of multiple tasks that previously required human intelligence and intervention. With this evolution, terms such as Artificial Intelligence (AI) and Machine Learning (ML) started to emerge to improve, even more, the efficiency and effectiveness of agents spread through all human environments, even though many times they go unnoticed. Nowadays there are self-driving vehicles, robots that perform tasks more quickly and effectively than any human ever could and even mobile robots that navigate autonomously across real environments without colliding with any type of obstacles whatsoever.

Most studied tasks in AI relate to areas such as autonomous driving, medicine, voice, image and handwriting recognition and many others as this theme is continuously growing and spreading. AI's first appearance goes as far back as 1943, when Warren McCulloch and Walter Pitts proposed the first mathematical model for building a neural network [1]. A few years later, in 1950, Alan Turing proposed that machines could make decisions and solve problems the same way humans can [2] and came up with the famous Turing Test, a method to determine if a machine is intelligent. More than 70 years later, AI is present in everyday life and continues to expand in a way that its direction cannot be predicted and only time will tell where it is heading to.

In this dissertation, the application of ML to a mobile robot is explored. ML is the branch of AI that studies computer algorithms that improve automatically through experience and by the use of data [3]. These algorithms create models based on acquired data (*training data*) in order to make decisions or predictions without being explicitly programmed to do so. Currently, ML is split into three different categories: *Supervised Learning*, *Unsupervised Learning* and *Reinforcement Learning*.

The context of this dissertation comes up with the interest of using a mobile robot to perform tasks involving navigation in an environment. Having a robot going from point A to point B is interesting, however, the ability to travel between multiple goals, that is, going from A to B and from B to C and so on, can be much more useful to some applications. Taking as an example mobile robots in industrial environments. Large companies, usually, have multiple mobile robots transporting and delivering cargo between storages. These tend to be guided with lines drawn on the floor or multiple beacons for self-localization with almost

no intelligence. This type of guidance tends to be very restrictive and can lead to many failures such as the wear of the lines, obstacles placed on the path or beacon failure, failures that those mobile robots cannot overcome most times. Implementing an AI solves these situations as the robot can safely travel between the desired points without any kind of physical guidance, with the exception of the sensors built to it, being able to adapt when obstacles are encountered. In these situations, multi-goal knowledge is very important since it allows the robot to travel between many predefined positions in the same environment instead of being capable of reaching just one position from every location in the maze, as in single-goal implementations.

1.2 Objectives

In this work, the navigation problem is formulated as a RL problem. The main concept underlying the work is the hierarchical learning structure in which the navigation task is decomposed into simpler subtasks. On the one hand, the low-level tasks allow the robot to be able to explore the environment, being easily reused in similar environments. Reward shaping is used aiming to engineer a reward function able to provide frequent feedback about the desired behaviors. On the other hand, the high-level problem involves making decisions in specific locations in the environment until reaching the final goal. The knowledge about the interconnectivity of locations forms an experience-dependent topological graph of the environment. Such a map would endow the robot with planning capabilities to robustly reach previously visited locations. The proper exploration of the environment allows the robot to memorize previous experiences.

The proposed approach emphasizes this topological view as being the key element to tackle the problem of multi-goal navigation, as well as to address the robot's behaviour in a dynamic environment. Maze-like environments represent a common way of limiting the problem to a manageable form. However, the ideas explored in this dissertation can be adapted to more realistic situations such as navigating in a building or in a more complex environment like a city without an initial map. Therefore, the main objectives of this dissertation are the following:

1. To provide the robot with elementary functionalities so that it is able to safely explore maze-like environments composed of corridors, 90-degree corners and T-junctions.

2. To conceive and develop a hierarchical structure to solve the problem of navigating in a maze-like environment resorting to its topological representation.

3. To evaluate the proposed approach in two scenarios: autonomous navigation among multiple goals and in dynamic environments.

1.3 Document Outline

This remainder of the dissertation is organized as follows:

- Chapter 2 presents a theoretical framework on RL and its key elements, with a demonstration of the application of a RL algorithm (Q-Learning) in a classic GridWorld problem. Alongside this, there is a summary on some previously done work in RL and Hierarchical Reinforcement Learning (HRL) with a focus on mobile robot navigation.

- Chapter 3 starts with a description of the experimental setup (Integrated Development Environments (IDEs), Operating System (OS), Robot Simulators and Robots) followed by a specification of the methodologies on the implementation of the lower-level tasks as well as a discussion and analysis of the results obtained by the robot when performing these tasks.
- Chapter 4 describes how the HRL approach is decomposed in this problem and how the high-level methodologies are implemented, followed by an interpretation of the results obtained when integrating those tasks with the lower-level ones. The multi-goal and dynamic behaviour scenarios are presented in this chapter as well, with a discussion on the robot performance in the end.
- Chapter 5 discusses the main contributions of the work, some of the limitations, and it presents some guidelines on the future work.

Chapter 2

Background

2.1 Reinforcement Learning

RL can be seen as the mimic, on a very smaller scale, of human learning from birth. At birth, a human has no knowledge whatsoever and does not know how to behave and what to do. However, over the years the humans start to understand how to eat by themselves and how to perform some other tasks, all learned with experience. Similarly to humans, a robot can also learn with experience when RL is applied to it, requiring only a way to collect information from the environment, like sensors, and some metrics to reward or punish each taken action.

RL belongs to the class of semi-supervised learning methodologies and is the segment of Machine Learning in which the agent is not presented with any type of previously collected data whatsoever. Instead, the agent should learn with practice, like in the real world, where no one tells it which action is best in a state and so it takes one of the available actions and interprets its value according to the produced reward, either good or bad. Usually, the main purpose of an agent is to achieve an objective while using the minimum number of actions possible, consequently obtaining the highest sum of rewards. The reward can either be just a static number or can be variable depending on the situation, giving RL the amplitude to be applied to various numbers of situations and environments.

During the learning, it is expected that the agent fails many times in the initial phase, however as the number of interactions with the environment increases and the agent starts to learn the best actions for some states, its behaviour starts to improve as well. In the end of the learning process, the agent is able to perform the desired tasks without any errors by following the policy which dictates the best action for each moment.

Most of the information presented below about reinforcement learning and everything related to it is based on both versions of the book by Richard S. Sutton and Andrew G. Barto [4, 5], together with some points from the dissertation written by Diogo Vidal e Silva [6].

2.1.1 Elements of Reinforcement Learning

RL is a methodology composed of many elements, from which the agent and the environment stand out. However, these two alone do not result in a learning process and so they need four more elements beyond them: a policy, a reward, a value function and a model of the environment (only in some cases, which are addressed below).

- **Policy**: Element that defines the way the agent behaves in a given time. It dictates the relationship between the states of the environment and the actions the agent takes. It can be a simple function or a lookup table, but, in more complex situations it may involve complex functions. Is considered the core of what the agent learns.
- **Reward**: This element determines the objective of a RL problem. Every action the agent takes results in a reward that has been previously defined or is computed in real-time according to some environment parameters. The main goal of the agent is to maximize this reward, consequently being able to distinguish the good from the bad actions, that is, the ones that take the agent to the highest reward states from the ones that take it to the lowest reward ones. This element causes the main impact on the policy since if an action selected by the policy results in a low reward, the policy will be updated and changed so that some other action is selected in the same state.
- **Value Function of a state**: It is the total aggregated sum of rewards the agent can expect to get in the future if it starts from that state. Is used to indicate the long-term desirability of a set of states since, even if a state has a low immediate reward, it can still have a high value if it is followed by higher reward states.
- **Environment model**: The last of the elements is a representation of the environment which mimics its behaviour when some action is performed. It allows the agent to infer how the environment will react and so help it to predict the next reward if an action is taken and hence, base the current action on the predicted reaction of the environment. This element is just used in model-based methods, which will be explained below.

Markov Decision Process

The Markov Decision Process (MDP) is one of the most popular mechanisms to solve reinforcement learning problems. It decomposes scenarios as a set of states, connected through actions and associated with specific rewards. The agent travels from state to state by selecting actions that result in the corresponding rewards. Is represented by a 4-tuple (S, A, P_a, R_a), where:

- S: set of states, called state-space.
- A: set of actions, called action-space (A_s when there are different action-spaces for different states).
- $P_a(s, s') = Pr(s_{t+1} = s'|s_t = s, a_t = a)$: probability that action a in state s at time t will lead to state s' at time $t + 1$.
- $R_a(s, s')$: immediate reward received after transitioning from state s to state s', after performing action a.

The goal of a MDP is to help the agent to find the best policy in a target environment: a function π that specifies the action $\pi(s)$ that the agent will choose when in state s. From all policies, the one that maximizes the value of all states at the same time is considered to be the optimal policy.

In the end, the MDP constitutes an interaction between the agent and the environment using only three variables: action performed by the agent, reward resulting from the transition to a new state and, lastly, the possible next states, as is represented in Figure 2.1 [5].

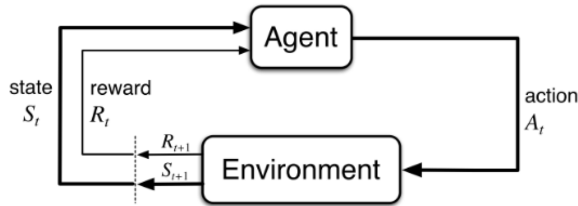


Figure 2.1: Agent-Environment interaction.

Policy and Value Function

A policy dictates the action the agent chooses at the moment. This function gives the probabilities, for each state, of selecting each one of the possible actions and is constantly changing during the learning phase until the agent reaches an optimal policy.

The value function of a state is obtained under a certain policy π, using Equation 2.1 where r_t is the reward in step t and $\gamma \in [0, 1)$ is the discount factor. Designated as $v_\pi(s)$, the state-value function for policy π is the expected sum of rewards starting in the state s and following the policy π until reaching the goal. Roughly, the value function estimates "how good" being in a state is. It should be noted that, here, the notion of "how good" is defined in terms of future rewards that the agent expects. Of course, these rewards are obtained according to the taken actions and might not match what is expected.

$$v_\pi(s) = E_\pi\big[G_t|S_t = s\big] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \text{ for all } s \in S \tag{2.1}$$

Similarly, in Equation 2.2, there is the value function of an action, denoted $q_\pi(s, a)$, which defines the expected reward for taking action a in state s and, thereafter, following policy π. Following the thoughts of the state-value function, this function is called the action-value function for policy π.

$$q_\pi(s, a) = E_\pi\big[G_t|S_t = s, A_t = a\big] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{2.2}$$

2.1.2 Learning Methods

As said in Section 2.1.1, the main objective in RL is to achieve an optimal policy. If all the elements that compose a MDP were known then it would be possible to compute the solution before any actual execution of any action. However, what distinguishes a RL problem from a path planning solution is the fact that the agent does not know all those elements, especially how the environment will react to a transition between states, that is, the probability given by $P_a(s, s') = Pr(s_{t+1} = s'|s_t = s, a_t = a)$ as well as the immediate reward from that transition, given by $R_a(s, s')$.

Two approaches can be used when solving this problem, a model-based one and a model-free one. In the model-based one, the agent acquires information about the probability of each transition and generates a model of the environment, which is used during the agent training to achieve the optimal policy. Contrasting the model-free methods, in the model-based ones the interaction with the actual environment is very small, since that after the creation of the

model the whole learning can be done in a simulated environment. Even though this process speeds up the learning, this method may not be convenient since a simulated environment requires highly computational processing and can introduce some errors in the learning when not modeled correctly.

Opposingly to the model-based approach, model-free methods, as the name suggests, do not require any model of the environment to work as the agent will learn the optimal policy solely by its interaction with the real environment. In these approaches, the agent performs each action and receives the immediate reward from the environment, being it positive or negative. Through the learning, the robot will explore the environment and learn the best action to take according to the state it currently is in, always trying to maximize the cumulative sum of rewards.

Inside the category of model-free methods, there are two main divisions: Monte Carlo Methods and Temporal Difference learning, both of which are explained below, alongside Dynamic Programming (DP), a model-based method, as it is important to understand the other two.

Dynamic Programming Method

DP is a collection of algorithms that can be used to compute and obtain optimal policies as a MDP, when given a perfect model of the environment. Taking into consideration the assumption of a perfect model and the need for great computational expense, DP algorithms have a very limited utility in RL. Beyond that, DP requires finite state and action spaces, however, it can provide solutions for some special cases in continuous spaces. Although all of that, DP provides an essential foundation for the rest of the methods approached below.

Monte Carlo Method

Unlike DP algorithms, Monte Carlo methods only require experience obtained while interacting with the environment during the learning process. This type of learning is striking, as the agent does not need any prior knowledge whatsoever about the environment. The agent learning is split into episodes, being each one the full iteration from a starting state until a goal state, and only then the value function and policy are estimated.

These methods follow the Generalized Policy Iteration (GPI) and, in order to improve policy, calculate the average of the value functions starting in the same state, instead of resorting to a template. To obtain an optimal policy, the agent has to start each episode in a random position, and so this can only be performed in a simulated environment since if it were performed in a real environment, moving the agent to a random position in every iteration would be very difficult. These starts in random positions allow the agent to obtain enough information for all state-action pairs and learn more.

For a learning problem in a real environment where those random starts are not possible, two different solutions were created: On-Policy Monte Carlo and Off-Policy Monte Carlo. With both methods very similar, the real difference between them is that, for the On-Policy one, the agent is constantly exploring the best policy it can get, while for the Off-Policy, the agent explores an optimal policy different from the one it is currently using.

Temporal-Difference Learning

Temporal-Difference (TD) learning is the central idea to reinforcement learning. TD is the combination of Monte Carlo methods with Dynamic Programming ideas. As in Monte Carlo, TD can learn just from raw experience while interacting with the environment without the need for a model. Like DP, TD methods update estimates based in part of other learned estimates, without waiting for a final outcome, that is, the end of an episode.

The agent chooses an action by following a policy interacting with the environment and, when in the next state, updates the value of the current state taking into account the learning rate factor, α, as in Equation 2.3.

$$V(s_t) \leftarrow V(s_t) + \alpha\big[r_{t+1} + \gamma * V(s_{t+1} - V(s)\big] \tag{2.3}$$

This is the methodology of RL behind Q-Learning and State-Action-Reward-State-Action (SARSA), two very simple yet effective algorithms.

Q-Learning

Q-Learning is a very famous and discussed model-free **off**-policy TD algorithm. Its first appearance goes back to 1989 in Chris Watkins [7] Ph.D. Thesis and three years later, Watkins himself, together with Peter Dayan [8], proved the convergence of the Q-Learning algorithm.

This algorithm allows an agent to learn by its experience, that is, without having any idea about the environment behaviour, the agent takes actions and learns from the respective consequences, through Equation 2.4.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha * [R_{t+1} + \gamma * max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.4}$$

The learned action-value function Q, will directly approximate $Q*$, the optimal action-value function, independently of the policy the agent is following. This approximation dramatically simplifies the analysis of the algorithm and enables early convergence proofs. However, the policy still has an effect since it determines which state-action pairs are visited and updated. For that reason, the only requirement to achieve convergence is that all pairs continue to be visited, a requirement that is needed throughout most of the methods in RL. To fulfill this, methods such as ϵ-greedy or *Softmax* can be used to balance exploration and exploitation when choosing actions.

As can be seen in Equation 2.4, this algorithm relies on two hyperparameters: learning rate α and discount factor γ. These two, together with ϵ in the case of using an ϵ-greedy approach, and the reward function are all the parameters that must be defined *a priori* to the learning and are all explained in detail in Section 2.1.3.

A pseudo-code procedure to implement the Q-Learning algorithm is shown in Algorithm 1.

Algorithm 1 Q-Learning algorithm.

1: Algorithm parameters: $\alpha \in (0, 1]$, $\gamma \in (0, 1]$, small $\epsilon > 0$, max_episodes
2: Define R, reward function
3: Initialize Q(s,a) for all states and actions available
4: **while** nr_episodes < max episodes **do**
5: Obtain state s
6: **while** s is not goal **do**
7: Choose an action a from state s using policy derived from Q (e.g. ϵ-greedy)
8: Perform action a and observe R and next state s'
9: $Q(s, a) \leftarrow Q(s, a) + \alpha * [R + \gamma * max_{a'} Q(s', a') - Q(s, a)]$
10: $s \leftarrow s'$

SARSA

SARSA, in opposition to Q-Learning, is a model-free **on**-policy TD method, first introduced in 1994 as Modified Connectionist Q-Learning (MCQ-L) by Rummery and Niranjan in a technical note [9], to which was given the name of SARSA later by Richard Sutton. This name simply reflects how the algorithm works when updating the Q-value function as it depends on the current state of the agent, S_1, the current action, A_1, the reward, R, the next state, S_2, and the next action A_2, resulting in the acronym (S_1, A_1, R, S_2, A_2).

While Q-Learning updates an estimate of the optimal state-action value function $Q*$ based on the maximum reward for the available actions, SARSA learns the Q-values associated with the policy the agent is currently following, dropping the $max_{a'} Q(S_{t+1}, a)$ present in the Q-Learning equation and ending up with Equation 2.5.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha * [R_{t+1} + \gamma * Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.5}$$

In all on-policy methods, Q^{π} is continuously estimated for the behaviour policy π, which changes at the same time so that the best action can be chosen. SARSA convergence properties depend on the nature of the policy's dependence on Q. For example, ϵ-greedy or ϵ-soft policies could be used. The probability of convergence is 1 to an optimal policy and action-value function, if it is guaranteed that all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy.

A pseudo-code procedure to implement the SARSA algorithm is shown in Algorithm 2.

Algorithm 2 SARSA algorithm.

1: Algorithm parameters: $\alpha \in (0, 1]$, $\gamma \in (0, 1]$, small $\epsilon > 0$, max_episodes
2: Define R, reward function
3: Initialize Q(s,a) for all states and actions available
4: **while** nr_episodes < max episodes **do**
5: Obtain state s
6: **while** s is not goal **do**
7: Choose an action a from state s using policy derived from Q (e.g. ϵ-greedy)
8: Perform action a and observe R and next state s'
9: Choose an action a' from state s' using policy derived from Q (e.g. ϵ-greedy)
10: $Q(s, a) \leftarrow Q(s, a) + \alpha * [R + \gamma * Q(s', a') - Q(s, a)]$
11: $s \leftarrow s'$; $a \leftarrow a'$

2.1.3 Q-Learning Application Example - GridWorld

Gridworld is a 2D maze of customizable size, built, as the name suggests, in the form of a grid where the agent aims to achieve a goal cell, from any initial position, in the least number of iterations possible by following the optimal path. On its way, there are multiple obstacles making the arrival to the destination even harder.

Hyperparameters

During the learning phase, for each state, the agent wants the action that offers the best reward. However, to achieve this, the agent needs to take all the actions so that it knows which one is the best, this is the **exploration** phase where it searches all possible scenarios. In the next step, after having all the information about all the actions it can take at each step, the agent can choose which action provides the best reward for each state and so, goes into **exploitation**. In RL a trade-off between both exploration and exploitation is necessary in order for the agent to discover new states that otherwise may not be selected during the exploitation process.

RL, as explained in Subsection 2.1.2, has two different methods: a model-free one where the agent uses only its actions and rewards to choose the next action to take and a model-based one where the agent uses its actions and reward alongside a model of the environment behaviour to choose the next action. To solve this problem a model-free method is sufficient enough and so the Q-Learning algorithm is used. In order to achieve an optimal policy, this algorithm has some hyperparameters that need to be carefully chosen: learning rate, discount factor and action selection probability, that is, exploration probability.

The learning rate, or α, dictates how important the new Q-Value is in comparison with the previous one. It is a positive small value, usually between 0 and 1, and should be set with discretion. When it is equal to 0, the agent discards all the new values and, consequently, does not learn anything since the old values are never updated. Therefore, the higher the learning rate is the faster the learning process will be. Although that, a higher learning rate may cause the agent to not converge correctly as it will give too much importance to newer Q-values compared to the ones already learned. For that reason, using a lower value, closer to zero, is recommended even if the learning is more time-consuming since the agent convergence will be more accurate. With this in mind, it is common to set a learning rate value like 0.1, 0.2 or 0.3.

The discount factor, or γ, is used to define the importance of future rewards. Similarly to the learning rate, it is usually a value between 0 and 1. When it is 0, the agent completely discards the future reward and only takes into consideration the reward obtained for the current state. This is used in problems where the actions only need to have short-term influence and impact the instant in which they are taken. In this problem that is not the case as the agent needs to learn an entire path and so actions should have a long-term impact. For that reason, a value closer to 1 is preferable, in order to give weight to the reward of the next state when the values are updated. Typically this value ranges anywhere from 0.8 to 0.99.

When choosing an action, there must be a trade-off between exploitation and exploration, since that if the chosen action is always random, the agent never acts according to the knowledge it obtains by exploiting the environment and if the action is always the one with the highest value, the agent will always follow the same path and never find new ones. The most common methods for choosing which action to take are **ϵ-greedy** and **Softmax**. In the ϵ-

greedy strategy, there is a balance between exploration and exploitation using epsilon (ϵ) to define how often the agent takes a random action or the highest valued one. The larger the epsilon, the greater is the probability of taking a random action, and consequently, the higher the agent exploration rate. Therefore if the epsilon has a low value, the agent will privilege the exploitation to the exploration. The ϵ is a value between 0 and 1 and having a lower epsilon is equivalent to a higher exploitation rate, whereby the agent chooses the action with the highest value with probability 1-ϵ and a random action with probability ϵ, allowing it to have a good balance between exploration and exploitation.

However, it should be noted that this strategy has a problem. When the agent is exploring, the chosen action may be the worst possible or one of the best. A solution to this is to vary the probability of each action, giving a higher probability to the best actions and lower to the worst ones, using a graded function. This is the **Softmax** [10] strategy and uses the Boltzmann distribution function to assign the probability $\pi(S_t, a)$ to the actions as shown in Equation 2.6.

$$\pi(S_t, a) = \frac{e^{\frac{Q_t(a)}{\tau}}}{\sum_{b=1}^{n} e^{\frac{Q_t(b)}{\tau}}} \tag{2.6}$$

In this equation, there is a positive parameter called temperature represented with τ. This defines how the probabilities of the actions are distributed. When the temperature is high, all the actions have equivalent probabilities, otherwise, the probabilities are different. Even though this is the best of both methods, the temperature value is very difficult to define in order to get the best results and, for that reason, the ϵ-greedy approach is used in this problem since ϵ is very easy to define and gives good results as well [11].

Algorithm Implementation

The Q-Learning algorithm uses TD learning to estimate the value of performing a certain action in a certain state and has the advantage of being a very simple algorithm that converges really fast when its hyperparameters are well defined. In each iteration, a state-action pair is visited and consequently updated using the next equation:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * [r + \gamma * max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)] \tag{2.7}$$

Where:

- α is the learning rate.
- r is the reward according to the action taken at the current state.
- γ is the discount factor.
- $max_{a'} Q(s_{t+1}, a')$ is the maximum reward that can be obtained from all the actions in the state s_{t+1}.

Algorithm 3 is a pseudo-code representation of the Q-Learning implementation in this problem, specifying, as well, how all the data for the metrics shown below are collected.

Algorithm 3 Q-Learning used in this implementation.

1: Define hyperparameters and max episodes (500 was used in this experiment)
2: Initialize rewardSum, paths, statesList and VisitedCells
3: Initialize Q-table with zeros for each state-action
4: **while** episode count < max episodes **do**
5:     **if** state = goal **then**
6:         VisitedCells[state] = VisitedCells[state] + 1
7:         paths[episode count] = statesList
8:         rewardSum[episode count] = rewardSum[episode count] + reward, r
9:         Update Q-table in goal position and all actions with reward, r
10:         Choose random initial position for next episode
11:         episode count = episode count + 1
12:         Clear statesList
13:     **else**
14:         Choose action, a_t, from a state, s_t, using the ϵ-greedy approach
15:         Update statesList with new state-action pair
16:         VisitedCells[state] = VisitedCells[state] + 1
17:         Take choosen action and get the reward, r and next state, s_{t+1}
18:         rewardSum[episode count] = rewardSum[episode count] + reward, r
19:         $Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * [r + \gamma * max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$
20:         Update Q-table in position state-action with $Q^{new}(s_t, a_t)$

Problem Approach

In this problem, 4 different mazes, shown in Figure 2.2, all with the same size (10*10 cells), are used in order to test the agent in different levels of difficulty. Actions that allow diagonal movement are not considered and so there are only 4 possible actions (up, down, left and right). Since each environment has a size of 10*10, the Q(s,a) table has 100 different states, each one with 4 possible actions.

Every time the agent reaches the goal, it is randomly deployed in a different cell of the environment and searches for the goal all over again. To ensure that every cell is visited, the initial position, in each episode, is always different from the previous ones until the agent has started once on all of the available cells. Only after that, it starts repeating them.

The Q-learning equation uses a reward to update the Q-value of each state-action pair. This reward is given to the agent each time it takes an action and, therefore, dictates how the agent converges. In this approach, the only time the agent gets a positive reward is when it reaches the goal, when it is given +100 points. When it decides to take an action that will lead to hitting a wall or obstacle a reward of -10 points is given. For every action, the agent takes a penalty reward of -1 point, assuring that it tries to reach the goal in the minimum possible steps and so, finds the optimal path.

When it comes to choosing the best value for each of the hyperparameters, many executions with different combinations of the learning rate, $\alpha = [0.1, 0.2, 0.3]$, the discount factor, $\gamma = [0.8, 0.9, 0.99]$, and the exploration rate, $\epsilon = [0.1, 0.2, 0.3]$, were performed. In order to make this comparison, some information has been collected from each execution and shown in the graphs from Figure 2.3 and the heat maps from Figure 2.4.

(a)

(b)

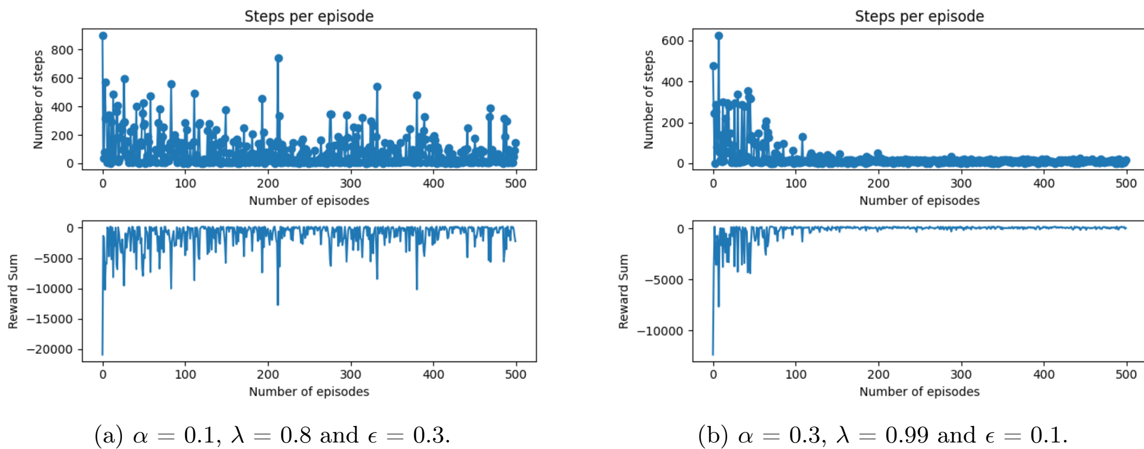
(c)

(d)

Figure 2.2: The 4 mazes used in the agent training.

The top graphs from Figure 2.3 show the number of steps the agent requires, in each episode, to reach from the start position to the goal state. Although it always starts in random positions of the environment, it is noticeable that the required number of iterations decreases as the episodes increase, which means the agent is learning over time. In the bottom ones it is shown the sum of all the rewards the agent took in every iteration for each episode. Initially, the agent has no idea where to go since it has not learned anything yet and so, this sum is extremely negative. With the increase of episodes, it is clear an increase in this sum since the agent is learning which is the best policy to follow.

From the analysis of both images and the comparison with other hyperparameter configurations, it can be concluded that, when the agent has a higher learning rate and discount factor it converges faster than with low ones and, with the increase of the ϵ, as expected, the agent does more exploration and so the convergence takes longer and is more uncertain. This can be seen when, comparing Figure 2.3a, where both the reward sum and the number of steps are very uncertain, to Figure 2.3b, where the convergence curve can clearly be seen.

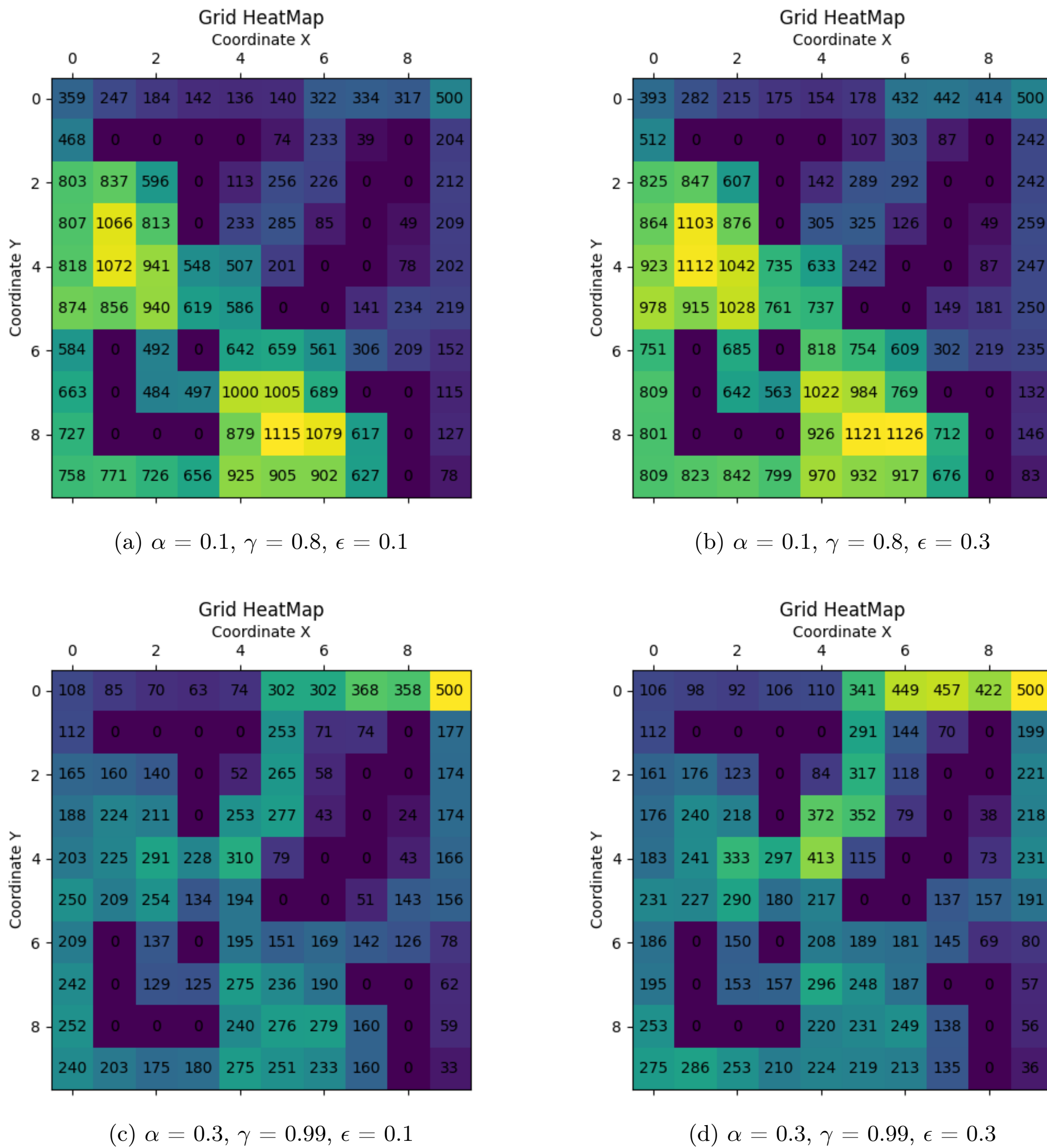


(a) $\alpha = 0.1$, $\lambda = 0.8$ and $\epsilon = 0.3$.

(b) $\alpha = 0.3$, $\lambda = 0.99$ and $\epsilon = 0.1$.

Figure 2.3: Hyperparameters evaluation.

Figure 2.4 presents the agent footprint ranging from purple to yellow, on Maze A (Figure 2.2a), with four different hyperparameter configurations, where the number in each cell is the number of visits by the agent during the whole learning phase. Obviously, the agent never goes through obstacles and so each one of them has a total of zero visits. Looking at each footprint it can be noticed that the agent explores the map and takes actions in very different ways according to the hyperparameter configurations. In the cases where the learning rate, α, and discount factor, γ are both low the agent visits the cells much more often which means that the agent is learning much slower and giving less value to the expected reward for the next state. It should also be noted that increasing the learning rate, α and the discount factor, γ, makes the agent perform a more uniform exploration through the whole map, despite the value of the ϵ being 0.1, 0.2 or 0.3. With all this in mind, the final hyperparameters configuration for this problem is $\alpha = 0.3$, $\gamma = 0.99$ and $\epsilon = 0.1$.



(a) $\alpha = 0.1$, $\gamma = 0.8$, $\epsilon = 0.1$

(b) $\alpha = 0.1$, $\gamma = 0.8$, $\epsilon = 0.3$

(c) $\alpha = 0.3$, $\gamma = 0.99$, $\epsilon = 0.1$

(d) $\alpha = 0.3$, $\gamma = 0.99$, $\epsilon = 0.3$

Figure 2.4: Heat maps in Maze A.

With all the hyperparameters well defined, the agent is trained in the four mazes from Figure 2.2 and the results are presented in Figure 2.5, where the best action for each state is indicated. When looking at each maze result, it can be seen that whatever state the agent starts in, it will always reach the goal and will never get stuck in two cells. However, it should be noted that, for some positions, the final path is not optimal but still very close to optimal. An adaptation of the reward function or extending the learning phase with more episodes could lead to better results.

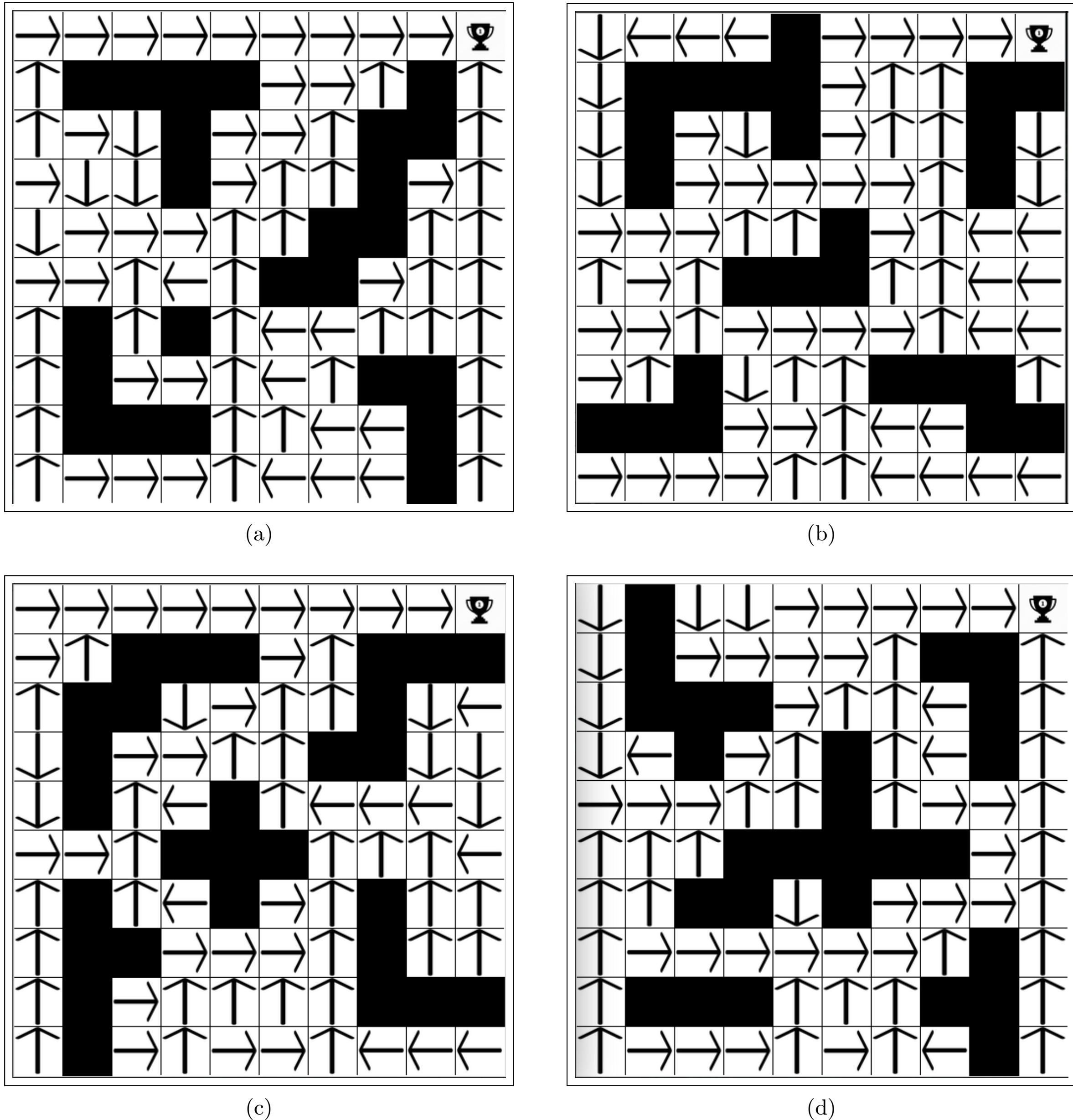


Figure 2.5: Final result of the agent training.

2.2 Hierarchical Reinforcement Learning

HRL, as the name suggests, is an area of RL where a problem is decomposed into a hierarchy of sub-problems or sub-tasks in a way that higher-level sub-tasks invoke lower-level ones as if they were atomic. This decomposition can have multiple levels and the sub-tasks can be RL problems as well.

A classic RL problem introduces a range of problems when applied in a real-world situation as, for instance, the curse of dimensionality, where the agent has to deal with the exponential increase of states and actions, consequently increasing the amount of time needed for learning.

Another problem happens when the agent is presented with a continuous state-space, meaning it will not visit every state in the environment and, consequently, will not find an optimal policy.

Applying a HRL can help to solve some of the problems of RL, creating sub-tasks and decreasing the state and action spaces to just the necessary for each sub-task. Besides this, HRL has some benefits on top of classic RL, such as

- Reduces problem complexity, since one task is divided into multiple smaller ones.
- Reduces computational complexity, since one sub-task can be reused multiple times.
- Value functions from one sub-task can help accelerate the learning process of another one or can be transformed in order to perform other tasks.
- State-space can be trimmed to just the necessary states for a sub-task, ignoring everything else.

From all the higher-level tasks, the ones which are formulated as RL problems are, usually, defined as Semi-Markov Decision Process (SMDP) since their actions will invoke sub-tasks which execute for an extended period of time. In contrast to MDPs, where the state transitions occur at discrete time steps, SMDPs generalize MDPs by allowing the state transitions to occur in continuous irregular times [12]. SMDPs can model either continuous-time events or discrete-time ones, considering always that the system is in a state for a random period of time and transits to another state instantaneously at the end of that time.

Instead of a random period of time, *Options* can be used, a partial policy that is defined with a subset of the state-space and a termination condition. With this framework, the wait time for each option will be the amount of time needed to complete the partial policy and not a random period.

On pair with options, Hierarchy of Abstract Machine (HAM) and MAXQ are two other known approaches when it comes to HRL [13], all explained in more detail below.

2.2.1 Options

Probably the most well-known framework in HRL and is represented in the format of Equation 2.8, with:

- I_o: Initiation set.
- $\pi_o : S \times A \to [0, 1]$: Option policy.
- $\beta_o : S \to [0, 1]$: Termination condition.

$$o =< I_o, \pi_o, \beta_o > \tag{2.8}$$

For an option to be used, the agent must be in a state that belongs to the *Initiation set*, and if so, the policy π_o will be followed until a termination condition is met. When the agent is in a state s that belongs to I_o, the next action a will be taken with probability $\pi(s, a)$ and, in the next state s', the called option could terminate with probability β or continue. Only after ending one option, the agent can select another one.

When using this framework, if the action-space is built by primitive actions and options, an algorithm will converge to an optimal policy [14], in any other case, the algorithm will still converge, however to a hierarchically optimal policy, that is, the achieved policy will be the best according to the given hierarchy, as the tasks will not only depend on the policies of its sub-tasks but on the context as well.

With this in mind, the final result of this framework is a structure composed of two levels:

- Bottom level sub-policy: given the environment states, the sub-policy outputs actions and runs until termination.
- Top level policy of sub-policies: given the environment states, the policy outputs sub-policies and runs until termination.

2.2.2 Hierarchies of Abstract Machines

In 1998, Parr and Russel [15] developed a new approach to structure MDPs hierarchically, called HAMs. As in the *Options* framework, HAMs use the theory of SMDPs, however with an emphasis on simplifying complex MDPs resorting to restricting the range of realizable policies instead of expanding the set of actions.

HAMs are an aggregation of non-deterministic state-machines, that is, machines where the optimal action is yet to be learned or decided, in which, the transition between states, can invoke lower-level machines. As the optimal action is yet to be determined, the learning algorithm still has to discover which lower-level sub-task to take at each time step.

HAMs are defined by a set of states, a transition function that determines the next machine state after an action or call state, and a start function that defines the initial state of the machine. A state can be of four types:

- **Action** state, that executes an action in the environment.
- **Call** state, that executes another machine as its subroutine.
- **Choice** state, that, non-deterministically, select the next machine state.
- **Stop** state, that halts the execution of the machine and returns control to the previous call state.

HAMs, in a RL context, have the advantage that the effort required to obtain a solution typically scales very badly with the size of the problem. Alongside this, HAM constraints can focus the exploration of the state-space to important sub-spaces, resulting in the reduction of the *blind-search* that agents must perform while learning about a new environment. As the state-space where the agent is effectively operating is reduced, the HAM-model learning rate will be faster as well as its policy iteration.

The authors from [15], introduced a new variation of the Q-Learning algorithm called HAMQ-Learning which learns in the reduced state-space. In this variation, the agent keeps track of the current environment state: t, the current machine state: n, the environment state and machine state at the previous choice point: s_c and m_c, respectively, the choice made at the previous choice point: a, and, finally, the total accumulated reward as well as the discount since the previous choice point: r_c and β_c, respectively. Alongside all these variables, there is an extended Q-table: $Q([s, m], a)$, which is indexed with an environment-state/machine-state pair together with an action taken at a choice point.

For each transition from state s to t with reward r and discount β, the algorithm updates $r_c \leftarrow r_c + \beta_c r$ and $\beta_c \leftarrow \beta\beta_c$. In conjunction with this, the Q-table is updated resorting to Equation 2.9 and, after that, $r_c \leftarrow 0$, $\beta_c \leftarrow 1$.

$$Q([s_c, m_c], a) \leftarrow Q([s_c, m_c], a) + \alpha[r_c + \beta_c V([t, n]) - Q([s_c, m_c], a) \tag{2.9}$$

In some experiments where the exploration was done according to a Boltzman distribution with a temperature parameter for each state and an inverse decay applied to $alpha$, the authors of [15] concluded that the HAMQ-Learning learned much faster than regular Q-Learning: Q-Learning required 9,000,000 iterations to reach the level that HAMQ-Learning achieved in just 270,000. Even after 20,000,000 iterations, Q-Learning did not achieve the levels of performance of HAMQ-Learning.

2.2.3 MAXQ

In MAXQ, each main task can be decomposed into a set of sub-tasks and each one of those can be decomposed, until the point where each existing sub-task is composed of a set of primitive tasks. Having this hierarchy well defined, the goal is to obtain, recursively, an optimal policy resorting to Hierarchical Semi-Markov Q-Learning (HSMQ), an algorithm developed by T. Dietterich [16], that is applied simultaneously to each task within the task hierarchy. For each sub-task p, there is its own Q function, $Q(p, s, a)$, which is the total expected reward of performing sub-task p in state s, executing action a and, thereafter, following the optimal policy.

While the HSMQ algorithm treats the HRL problem as a collection of independent Q-Learning problems, it lacks representational decomposition of the value function, which is represented and learned independently. The MAXQ framework allows this decomposition of value functions and so, the Q-value of a state-action pair is decomposed into the sum of two components, as stated in Equation 2.10, where $V(a, s)$ is the total expected reward when executing the action a in state s and $C(p, s, a)$ the total expected reward for completing the parent-task, denoted by p, after taking action a, where a can be either a primitive task or a sequence of tasks.

$$Q(p, s, a) = V(a, s) + C(p, s, a) \tag{2.10}$$

Equation 2.10 shows how the Q-Value of a parent-task is related to the value function of a child-task. When applied recursively, it shows that the Q-value of the **root** task can be decomposed into a sum of Q-values of all its descendant tasks:

$$V(p, s) = max_a[V(a, s) + C(p, s, a)] \tag{2.11}$$

MAXQ combined with Q-Learning introduced the MAXQQ-Learning algorithm, and this, when compared with the traditional Q-Learning, shows much faster convergence to an optimal policy, with and without state abstraction, because of the support for reusing and sharing sub-tasks that MAXQ entails with it [17].

2.3 Mobile Robot Navigation

Robotics has become a very challenging and famous case of study. Initially people thought it would be possible to have a robot doing activities a human does on a daily basis, however, today it is still impossible to have a robot performing many mundane tasks fully autonomously. ML is the main technique used nowadays when it comes to the process of teaching a task to a robot in order to increase its level of autonomy. Currently, it is already possible to buy cars with auto-pilot integrated [18] thanks to ML together with Computer Vision. Robots using ML can learn through data, as in *Supervised* and *Unsupervised Learning*, or can learn through what is called trial-and-error experience, in RL.

In [19], Leonard and Durrant-Whyte stated that the problem of navigation can be summarized into three questions:

- **Where am I?**: addresses the problem of localization. How can a robot find where it is in a given environment, based on the information it can collect and on what it knows it has done before.
- **Where am I going?**: addresses the problem of having an objective. The robot must know that the finish point exists and where it is.
- **How should I get there?**: addresses the problem of path planning. The robot knows where the objective is but needs to know what actions/steps it should take at each moment in order to reach the final destination from its current position.

2.3.1 Mobile Robots

Robots are machines capable of carrying out a complex task or set of tasks. Some are guided through external controllers and others have embedded systems to control them. They can be of many shapes and usually are designed with a specific task in mind rather than aesthetics.

Mobile robots are robots that have locomotion capabilities and so can move around in their environment and are not fixed on a physical location. They can be autonomous mobile robots which are capable of navigating an uncontrolled environment without the need for a physical guidance or might need to rely on guidance devices that allow them to travel predefined routes in controlled environments.

Inside mobile robots there are three main categories:

- **UGVs** - Unmanned Ground Vehicles are vehicles that operate on the ground without an onboard human presence. Can be used for many applications where human presence might be inconvenient, dangerous or impossible. Generally, they have a set of sensors and can be controlled remotely through a human or autonomously based on the information collected about the environment. They can be either wheeled, tracked or legged robots. Figure 2.6 shows two very famous UGVs: Spot by Boston Dynamics [20] on the left and Gladiator Tactical UGV [21] on the right.

(a) Boston Dynamics Spot.

(b) Gladiator TUGV.

Figure 2.6: UGVs.

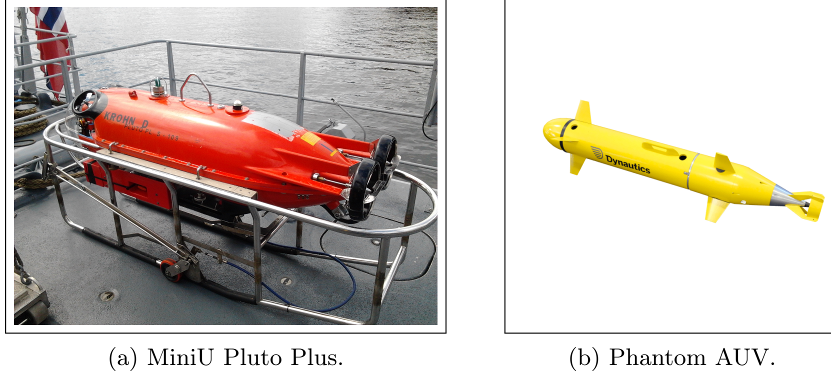
- **UAVs** - Unmanned Aerial Vehicles, commonly known as **drones**, are aircrafts without any human pilot, crew or passengers on board and are part of Unmanned Aerial System (UAS) which includes a ground-based controller and a system of communication with the UAV. They can be operated under human remote control or with multiple degrees of autonomy from autopilot assistance up to fully autonomous piloting without human intervention. Figure 2.7 shows two examples of UAVs: General Atomics MQ-9 Reaper [22] on the left and a consumer drone from DJI [23] on the right.

(a) General Atomics MQ-9 Reaper.

(b) DJI Drone.

Figure 2.7: UAVs.

- **AUVs** - Autonomous Underwater Vehicles are robots that travel underwater without requiring input from an operator. Together with Remotely Operated Underwater Vehicles (ROVs), these robots constitute the undersea system called Unmanned Underwater Vehicles (UUVs).Figure 2.8 shows two examples of AUVs: MiniU Pluto Plus [24] on the left and Phantom by Dynautics Ltd [25] on the right.



(a) MiniU Pluto Plus.

(b) Phantom AUV.

Figure 2.8: AUVs.

2.3.2 Reinforcement Learning in Mobile Robotics

As said in Subsection 2.3.1, some robots operate completely autonomously, however, for them to have this ability they must learn how to perform the tasks they are designed for. When it comes to ML, they can learn from three categories as stated earlier. Since the algorithms developed in this dissertation will rely on RL and HRL techniques, some work that is related to what will be done here, in certain aspects, is summarized below.

The authors of [26] use Q-Learning to help a mobile robot move out of an unknown maze. The robot is placed in the center of a spiral maze and, using sonar sensors, must learn how to reach the exit without hitting any obstacle whatsoever. The world states are defined as a 3-element vector with left, right and front distances as well as their relative amplitudes. Along with this, the robot can only perform three actions: move forward 10 cm and turn 15 degrees to the left or right. In order to prevent hitting walls, every time one or more of the measured distances is less than 17 cm the robot will move backwards 10 cm. So that the world states are distinguished more effectively, they are split into two categories: *health* states and *sub-health* ones, where the first indicate that obstacles are still very far and the last that they are too close. In this problem the reward is defined by the user in a table and will vary according to each state: if it is a healthy one, the result is a positive reward, if it is a sub-healthy one, the result is a negative reward. With everything defined and after training the robot, the authors concluded that the Q-Learning algorithm converges fairly quickly, in just under 300 steps.

As in the previous example, in [27] the authors use Q-Learning as well, however, with a different approach. Here the algorithm will learn the shortest path, avoiding obstacles, from the current position of the robot to a goal state by analyzing captured images of the environment. The vision-based obstacle detection algorithm, developed in OpenCV with edge detection algorithms, allows the creation of a grid map which is then given to the Q-Learning algorithm and so, the problem can now be treated and solved as a Gridworld problem. Two action-spaces were used in this problem: one with 4 actions (up, down, left and right) and another with 8 (the previous 4 plus diagonals). The state-space always depends on the image size and number of obstacles present. The reward developed can be split into three parts: one where the agent reaches out of the image and gets the worst reward possible, another that happens when the agent reaches the goal and gets the best reward and, finally, a living penalty that happens in every move where the agent gets a very small negative reward.

Two approaches to solve RL problems with Q-Learning have been discussed already, how-

ever, when it comes to continuous non-linear environments, this might not be the best option. In [28], Tomás Martínez-Marín developed a new algorithm which incorporates the adjoining property, a mechanism to select the state transitions that will be learned by the robot and allows to overcome some limitations of RL techniques when applied to continuous non-linear systems, such as nonholonomic vehicles. The author used a method based on the Adjoining Cell Mapping (ACM) technique, which creates a cell mapping where only transitions between adjoining cells are allowed [29]. While the Q-Learning algorithm transitions are evaluated at fixed sample times, the transitions of the RL algorithm developed by the author have to satisfy the adjoining distance condition that had been previously defined. In the end, the new RL approach was successfully employed for optimal motion planning of a real robot, in contrast with conventional RL algorithms such as Q-Learning, since it does not need to use function interpolation to find a close to optimal behaviour in continuous state-spaces. Alongside that, this approach is robust to noise and changes in the vehicle parameters, since the robot model is estimated on-line updating the optimal motion law in real time.

Moving to a more complex approach, the authors of [30] propose a HRL architecture in order to solve a robot navigation problem. The proposed HRL consists of two layers: one for movement planning and another for the movement execution itself. In the first one discrete RL is applied in order to generate navigation trajectories based on movement primitives. With those defined, the policy for the movement execution can be learned with the application of continuous RL. For this work, 8 different primitive actions were defined: forward, backward, left and right both with 0.25-meter and 1-meter displacements. For the movement planning layer the defined reward was 1 in the goal and 0 in any other state, however, to avoid the risk of collision, a penalty was defined for moving directly along walls. Alongside this, a bigger reward was given to actions with 1-meter displacement compared to the ones with 0.25 meters only. For the movement execution layer, the reward was set based on the robot X and Y positions, together with its yaw θ. In the end, the authors concluded that, for all primitives actions, sufficiently accurate policies could be learned. The navigation policy could be learned within a few seconds, taking just 2.5 seconds on a map with 1000 cells.

Usually, developers resort to Q-Learning algorithms in problems where the robot must navigate in static environments since dynamic ones make the problem much more complex as it will have an infinite state-space. However, the authors of [31] proposed an application of Q-Learning for solving this by limiting the number of states through a new definition of the state-space. With this limitation, the Q-table size can be reduced and hence, the navigation algorithm can perform faster. So that this approach could work, some assumptions had to be made among which, the robot knowing its position and velocity at each instant as well as the position and velocity of its goal and obstacles, if dynamic. It is assumed as well that the robot's speed is always higher than its target's and obstacles' speeds. In order to overcome the problem of the infinite state-space, the robot is considered the center of the universe and everything around it can be split into four different regions as if it were a 2D cartesian coordinate system. Thus, each state is defined by the region containing the goal, the region containing the closest obstacle and the angle formed between straight lines from the robot to the goal and obstacle. The action-space is defined with only three actions: move forward, turn left and turn right. To define a reward the states are split into four categories: *Safe* states, *Non-Safe* states, *Winning* state, which happens when the robot reaches the goal and, finally, *Failure* states, which happen when the robot collides with obstacles. With this classification, the authors defined the reward as being 1 when going from *Non-Safe* to *Safe* state, -1 for the opposite as well as when transitioning between *Non-Safe* states and getting closer to obstacles,

0 when traveling between *Non-Safe* states and getting away from obstacles, 2 when reaches the *Winning* state and, lastly, -2 when reaches *Failure* states. In the end, the authors concluded that the performance of Q-Learning, when compared to the potential field method in dynamic environments, resulted in much slower learning since approximately half the time was needed by the robot to reach the target using the potential field method.

With the objective of having a mobile robot navigating between multiple goals, the authors of [32] presented a new algorithm, called GM-Sarsa(O), in order to find approximate solutions to multi-goal RL problems modeled as MDPs and coupled by the requirement of sharing actions. In contrast to other algorithms that find optimal policies for each goal in isolation, their approach finds good policies taking into account the composite task. Given that SARSA is an on-policy method, the value updates are based on the actions that are actually taken and not on the best possible actions, and so, unlike the Q-learning algorithm, it does not suffer from positive bias which is a problem when trying to find a good policy for multiple goals in an environment. In the end, the authors concluded that some empirical results showed that their approach performed better than some other algorithms (Negotiated-W, GM-Q and Top-Q), however they did not prove convergence for their algorithm.

The authors of [33] are the ones that have a closer multi-goal implementation to the one of this dissertation. In their paper, it is proposed a two-stage framework for visual navigation in which the experience of the agent during exploration of one goal is shared to learn to navigate to other goals. They developed a deep neural network for estimating the position of the robot in the environment using ground-truth information provided by a classical Simultaneous Localization And Mapping (SLAM) approach. A multi-goal Q-function learns to navigate in the environment using a discretized map previously provided. However, their learning process begins in a 2D simulator and then is deployed in a 3D simulator where the robot resorts to the developed deep neural network to estimate its position and location. The authors focus more in the deep neural network and so compare multiple architectures to select the best one. After that, a comparison between the multi-goal RL method and traditional RL is done, showing significant improvement when the multi-goal method is used. The study done around the deep neural network showed that this type of network can learn and generalize in different environments using camera images with high accuracy in position and orientation.

In order to recognize certain places in the environment the mobile robot must have some sort of vision recognition system. Since vision-based navigation has been researched extensively in multiple articles, the authors of [34] have written a survey where many pieces of work related to this theme are summarized. In this survey two major approaches are dealt with: *map-based navigation* and *map-less navigation*, in which the map-based navigation is subdivided in *metric map-based navigation* and *topological navigation*. Alongside this, the authors explore both indoor and outdoor environments, where the outdoor ones can be structured environments (road exploring) or unstructured ones (random exploration). Different types of vehicles were considered in this research as well. When it comes to ground vehicles in the map building category, multiple strategies with different configurations of sensors (single camera, omnidirectional camera, stereo cameras...) were analyzed, going from a visual SLAM with landmarks and tracking [35] to 3D construction of occupancy grids [36], topological maps [37] and visual sonars [38]. In the map-less category, again with multiple configurations of sensors, strategies like optical flow [39] and appearance-based methods [40] were reviewed. In the end, there are multiple techniques with the most varied configurations of sensory vision that can be applied together with this dissertation's algorithms in order to go from simulation to real-world environments.

2.4 Final Remarks

In Subsection 2.3.2 a few articles involving different areas of mobile robot navigation and RL have been analyzed and presented. As can be understood, they all discuss work in part similar to the one developed in this dissertation, however, none of them have a work that fully matches the one developed here. Some articles discuss the advantages of HRL, others the use of Q-Learning or similar algorithms either in static or dynamic environments with discrete or continuous state-spaces while a couple of other articles approach the multi-goal problem, but none of those brings up a work of a HRL implementation with a multi-goal solution in a continuous environment with dynamic behaviour. Although that, it should be noted that some work discussed in the previous section is not implemented in this dissertation but is very important when transferring the developed algorithms from simulation to real-world environments, as is the case of the visual recognition and map building systems from [34].

Some articles present more in-depth work than the one developed here, as is the case of [33], where the authors developed and implemented a deep neural network for localization together with multi-goal navigation.

Chapter 3

Low-Level Exploratory Behaviour

This chapter describes the developments carried out to provide a simulated robot with an exploratory behaviour inside a maze-like environment made up of corridors, corners and doors. The exploratory behaviour defined in the scope of this work differs from navigation in that there is no specification of a goal location. The objective is to learn a set of low-level functionalities allowing the robot to pass through a corridor, corner or door, regardless of the specific robot's location in the maze. For example, if the robot learns the policy to cross a corridor, then it can be used in any other corridor because they are all identical. In this context, Section 3.1 describes the programming environment and software, followed by a discussion about the two robots used. Section 3.2 provides an insight on how the state and action spaces are discretized. Section 3.3 presents an overview on the algorithm used to train the robot. Section 3.4 specifies how the rewards for each task have been done. Section 3.5 reports the robot behaviour and discusses the main results achieved and the implications for the objectives to be obtained. Finally, Section 3.6 summarizes the work done in this chapter.

3.1 Programming Environment and Software Tools

This dissertation has been developed in *Elementary OS 5.1.7 Hera*, a Linux OS based on *Ubuntu 18.04.4 LTS*. All the algorithms were developed in the Microsoft Visual Studio Code IDE, using Python version 3.6.9, an interpreted high-level general-purpose programming language, together with some libraries such as Matplotlib [41] and Keras [42].

An important part of the development of this dissertation is the simulator used. Using a robot simulator software brings many benefits to the world of robotics, such as allowing to save time and speed up the execution and iteration processes, be able to test code in a safe environment and easily make small adjustments to it, and the ability to simulate complex environments and train machine learning algorithms quicker than in the real world resorting to the process speed up. Even though there are many benefits, there are also some drawbacks as well. Simulator will never, at least for now, fully replicate the complexity of the real world and they still need powerful computers to simulate complex 3D environments fast.

Two very famous robot simulators are compared in Table 3.1: the Webots Robot Simulator from Cyberbotics Ltd [43] and the Gazebo Simulator from Open Source Robotics Foundation (OSFR) [44], both with their advantages and disadvantages. Taking into account these two simulators and the very complete comparison between them in [45], it can be concluded that both simulators have very similar features and any of them would be a good choice. How-

ever, the Webots Robot Simulator is the only one that include a python external Application Programming Interface (API), making it the natural choice for this work.

	Developers	**Platforms Supported**	**Main Programming Language**	**Extensibility**	**External APIs**
Webots	Cyberbotics Ltd	Linux, macOS, Windows	C++	API, PROTOs, Plugins(C/C++)	C, C++, Python, Java, Matlab, ROS
Gazebo	OSFR	Linux, macOS, Windows	C++	Plugins(C++)	C++

Table 3.1: Webots vs Gazebo main aspects.

3.1.1 Webots Robot Simulator

Webots is a cross-platform, user-friendly robot simulator that was released by the Swiss company Cyberbotics and runs in Linux, macOS and Windows. It has support for the most common programming languages in robotics such as C/C++, URBI, MATLAB and Python. Besides this, the simulator can be controlled externally using the provided API or any standard Transmission Control Protocol/Internet Protocol (TCP/IP) network. It uses a fork of Open Dynamics Engine (ODE) to simulate physics in realistic 3D environments and supports accurate modeling of collisions and contact points, allowing for tests in a wide range of scenarios using virtual robots. This simulator also has an interesting library of robot models and more can be imported from most modeling software like Solidworks, AutoCAD or Blender, as well as maps and terrains to create accurate 3D representations of the world in the testing environment. Notwithstanding, with the increasing complexity of the environment, more powerful computers are required to perform the simulation smoothly.

To control each physics step, that is, the speed of the simulation, Webots has the **basicTimeStep** parameter. This represents, in milliseconds, the duration of a simulation step, i.e., the time interval between two computations of the position, speed, collisions, etc. of every simulated object. To obtain a quicker simulation, this parameter can be increased, with the cost of decreasing the precision of the simulation compared to the real world. Webots documentation recommends a value between 8 and 16 for regular use of the simulator, so 16 is used. Besides this, each robot controller has a parameter that defines its own simulation step, usually named **TIME_STEP**, and should be a multiple of the **basicTimeStep** value.

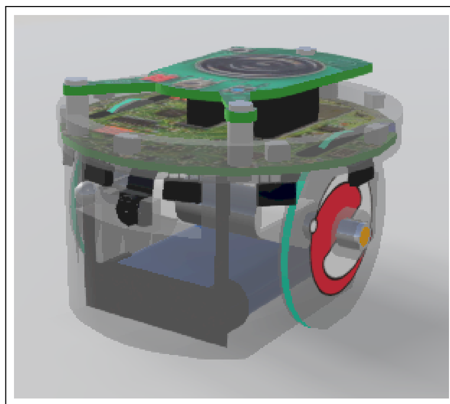
3.1.2 Mobile Robot and Environment

During the initial phase of development, two mobile robots from the Webots library were analysed: the Khepera-IV and the E-puck. Taking advantage of its Linux Core, the KTeam's Khepera-IV robot embeds a standard Linux OS, providing a well-known C/C++ environment for application development, as well as a python one. This allows almost any existing library to be easily ported to its system which enables the development of portable embedded algorithms and applications. It is considered the new standard tool for robotic experiments and demonstrations, such as navigation, AI, control, real-time programming or advanced electronics demonstrations. The robot is equipped with two brushed motors with incremental encoders, 5 built-in ultrasonic sensors plus 12 infrared ones (four of which directed to the ground with short-range), a 752×480 color camera, 3-axis accelerometer and gyroscope, and 3

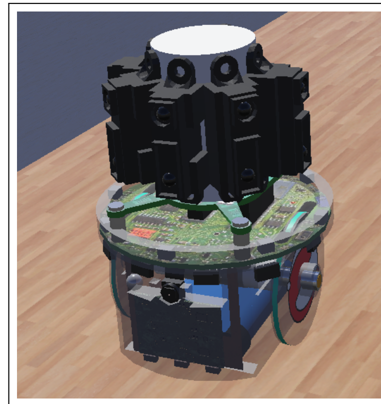
RGB Light Emitting Diodes (LEDs) on top. In terms of dimensions, the robot has a diameter of 140.80 mm, a separation between wheels of 105.40 mm, and a height of 57.70 mm.

The GCTronic E-puck is the one used on the following experiments and is a miniature mobile robot originally developed for teaching purposes at École Polytechnique Fédérale de Lausanne (EPFL) by the designers of the successful Khepera robot. The hardware and software are fully open source, providing low-level access to every electronic device and offering unlimited extension possibilities. It has two wheel motors capable of a maximum speed of 6.14 rad/s, 8 infrared sensors for proximity and light measurements, an accelerometer, a gyroscope, a camera with resolution 640×480, surrounding LEDs, bluetooth communication and support for ground sensors modules. The diameter is 71 mm, the separation between wheels is 52 mm and the height is 50 mm.

Two versions of this robot are illustrated in Figure 3.1. Figure 3.1a is the original robot version, while Figure 3.1b is a modified version which includes a belt of six extra infrared sensors attached onto the top of the robot. The orientation of the sensors follows the structure proposed in [6]: one sensor facing backward and five facing forward and sides 45 degree apart. Each Sharp infrared sensor, model GP2Y0A41SK0F, is able to measure between 4 and 30 cm [46], distances a lot farther than the original infrared sensors built into the E-puck robot which can only measure up to 4 cm.



(a) GCTronic's E-puck.



(b) GCTronic's E-puck with 6 Sharp sensors.

(c) K-Team's Khepera-IV.

Figure 3.1: Robots analyzed.

All maze-like environments to be created consist of corridors, corners, and doors. The

corridors must all be 30 cm wide, the corners are characterized by their orientation at 90-degree angles, and the doors are points that allow access to new corridors by turning left, right or moving forward. There is a transition region between doors called junction, always placed at 90-degree angles as well (i.e., T-junction). Figure 3.2 presents an example of an environment respecting all the above rules, where each junction is labeled for better identification.

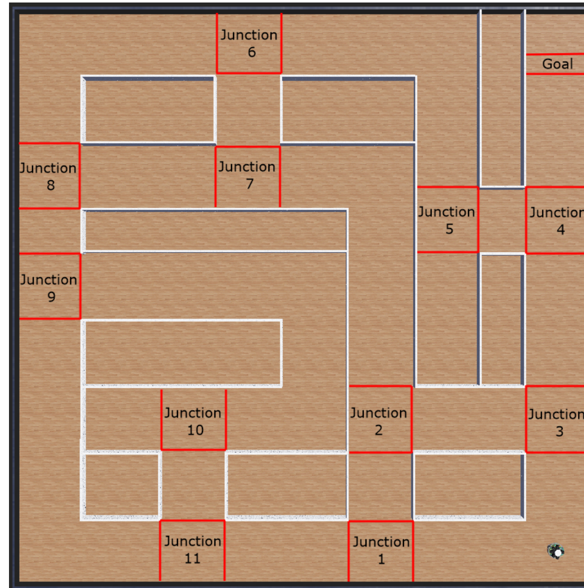


Figure 3.2: Example of environment.

3.2 State-Action Discretization

The task of navigating in a maze can be very complex for a robot to understand, however, to travel through a maze it just needs to know how to behave in three main situations: corridors, corners and doors. These behaviours can be isolated and turned into low-level tasks so that the robot can perform each situation independently, ending up with a task to move forward in a corridor, another two to turn in corners to the right and left and, finally, three more for actions on doors.

The robot's state is represented by the distance measured provided by the 6 IR-sensors installed on-board. The 6-dimensional state defines the robot's local position and orientation relative to the corridor boundaries. The robot's actions will be the speed commands to be sent to the right and left wheels. The first step was to reduce the dimensionality of states and actions through a manual discretization process. The main challenge is to find the right number of regions for each dimension such that the robot achieves a good performance, and, at the same time, learns quickly. This work follows the experience reported in [6], however with different levels of discretization. In that work 4-dimensional states were used instead of 6 and action discretization was done based on linear and angular velocities instead of direct speed application to the wheels.

The exploratory behaviour defined in the scope of this work differs from navigation in that there is no specification of a goal location. Instead, the objective is to train the robot so that it always follows the middle of the corridors, having a safe behaviour in corners and

doors. Learning this lower layer of functionality can be compared to a scenario in which a car driver learns basic skills on a road (intersection, roundabout, among others) and applies them regardless of the actual location on that road and regardless of the specific road the driver is on.

After a conversion using the equation developed in Appendix A, the voltage of the infrared sensors can be used as distance between 4 and 30 cm. If it was considered that each centimeter change was a new state, then each one of the six sensors would have 27 different states which would correspond to a total of $27^6 = 387420489$ states. This would create an excessively big state-space table and so, the distances were discretized into 6 levels, as shown in Table 3.2, shrinking the table size to just $6^6 = 46656$ states. In this experiment, the robot does not have access to its global position, it can only estimate its local position related to the maze walls, which increases the difficulty relatively to the Gridworld experiment where the agent knew its global position at each step.

With the state-space well defined, it is still missing the possible actions the robot can take in each state. Considering that the robot must be able to go forward, turn left or right in corners or doors and compensate for eventual approximations to walls, nine different actions, seen in Table 3.3, are contemplated and used across all low-level tasks.

Measure Distances (cm)	**Level**
$4 \leq d < 6$	1
$6 \leq d < 10$	2
$10 \leq d < 15$	3
$15 \leq d < 20$	4
$20 \leq d < 25$	5
$d \geq 25$	6

Table 3.2: Sensors discretization levels.

Actions	Front (F)	Light Left (LL)	Light Right (LR)	Mid Left (ML)	Mid Right (MR)	Left (L)	Right (R)	Hard Left (HL)	Hard Right (HR)
Speed (Left, Right)	(2,2)	(1.5,2)	(2,1.5)	(1,2)	(2,1)	(0,2)	(2,0)	(-2,2)	(2,-2)

Table 3.3: Actions discretization (wheel rotation speed in rad/s).

With both state and action spaces specified, it is time to move to the learning phase. As the problem happens in a continuous environment, in contrast to what was presented in the GridWorld experiment, the algorithm used is not the Q-Learning, but instead R-Learning, which will be explained in the following section.

3.3 R-Learning Algorithm

Most works in RL focus on finite horizons and the better studied discounted framework such as Q-learning, where long-term rewards are attenuated based on the delay in their occurrence. Similar derivations exist for the average-reward case. For example, the average-reward algorithm proposed by Schwartz [47], so-called R-learning, is more appropriate for undiscounted continuing tasks such as those to be addressed in the simulated environment. This

is an off-policy method where the training is not split into episodes with finite returns. Instead, R-learning is a method for optimizing the average reward which is updated for every non-exploratory action, aiming to obtain the maximum reward in each time-step (near-term and far-future reward are the same). Likely to Q-Learning, R-Learning uses the action-value representation, where $R^{\pi}(x, a)$ represents the average adjusted value of performing an action a in a state x and then, following the policy π. Thus, R-Learning has four major steps that are described in Algorithm 4, where x is the current state, y the next state, a the action and ρ the average reward.

In this version of the R-Learning algorithm, there are two hyperparameters, $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$, where the first is the learning rate controlling how quickly errors in the estimated rewards are corrected and the second is the learning rate for updating ρ. Note that, just as a matter of preference, the hyperparameters are switched when compared to the original article [47], that is, where should be an α is a β and vice-versa. According to the sensitive analysis made in [47], these hyperparameters have four main properties:

- More exploration is better than less: Higher values of exploration generally produce better results than lower values.
- Slow decay of α is better than fast decay, however, in this experiment, no decay is considered.
- Low values of β are better than high values: values such as 0.05 produce better performance than values like 0.5.
- High values of α are better than low values, however, this may depend on the particular experiment being made.

Before starting the training itself, it is necessary to initialize the R-Table for each state-action pair, as if it were a Q-Table, and fill all states with their reward, obtained from the reward functions explained in Section 3.4. As in Q-Learning, every time the robot is in a state and performs an action, the corresponding state-action pair is updated, but now resorting to the equation presented in Algorithm 4. If the action taken is non-exploratory, then the average reward, ρ, is updated as well.

Algorithm 4 R-Learning used in this experiment.

1: Define hyperparameters α and β
2: Initialize R-Table with zeros for each state-action and ρ with zero
3: **while** True **do**
4: reward = RewardTable[y]
5: $R_{t+1}(x, a) \leftarrow R_t(x, a) * (1 - \alpha) + \alpha * [reward - \rho_t + max_{a \epsilon A} R_t(y, a)]$
6: **if** Non-exploratory action **then**
7: $\rho_{t+1} \leftarrow R_t * (1 - \beta) + \beta * [reward + max_{a \epsilon A} R_t(y, a) - max_{a \epsilon A} R_t(x, a)]$
8: Update R-Table in position state-action with $R_{t+1}(s, a)$
9: Set next state as current state, $x = y$

3.4 Reward Specification

In the RL framework, the specification of the desired robot's behaviour is done implicitly through the reward function. During training, the robot must observe variation in the reward signal in order to be able to improve the policy. The main difference to the gridworld problem (Subsection 2.1.3) is the way the reward is defined. As the robot does not have a predefined goal, the reward must adjust to the state it is in, instead of always being the same and only changing when in a goal state or a wall has been hit. For that reason, each low-level task has a reward function which returns different values according to the local position of the robot in the maze where the training is performed. This function is one of the major aspects of RL since it dictates how good or bad an action is in a given state whereby, learning how to complete a task successfully depends majorly on this reward and so, they must be carefully designed so that the robot can achieve an optimal (or close to optimal) policy.

Unlike the experiment from Subsection 2.1.3 and as stated in Section 3.3, the robot does not have a goal and the training is not split into episodes, but instead, its objective is to complete each task successfully and obtain the max reward possible in every time-step. With this in mind, it is not possible to define a sparse reward since the robot will never achieve a goal state, whereby each reward function must take into consideration other features than a final objective, in order to shape the primary reward to appropriately distinguish good from bad actions in every state and fill the gap of the sparse reward. For that reason, instead of relying on a reward function encoding success or failure, this work adopts a process known as reward shaping [48, 49] in which intermediate rewards are used to guide the learning process to a good solution (close to the desired behaviour). Depending on the desired task to be fulfilled, this reward shaping will consider different features such as the robot orientation or its distance to the walls.

During the training the robot does not know its local position, it only understands that it is in a specific state according to the measures obtained from the sensors, however, all the reward functions depend on its local position in the environment. With this in mind, before the training, a table with 46656 states is created and filled with the appropriate reward the robot should get in each state, so that when training it can access a reward based on the sensors' state instead of needing to know its local position in the maze.

The first approach to fill this table was to obtain the robot position and orientation based on geometric calculations with the measured distances obtained from the sensors. This calculations were previously made in a Matlab Simulator where the sensors had no noise and so it worked perfectly. However, when it was adapted to python to work together with Webots Simulator, even after some adjusts and trials it was not possible to obtain accurate positions and orientations based just on the sensors' measurements since most of the calculations relied on sines and cosines and a small variance in the measured distances could change the obtained angle, ending up with an incorrect position of the robot which would change the reward for that state. For that reason, another way to fulfill the reward table has been done.

Resorting to the Webots Simulator *Supervisor* mode, the robot controller can know its current position and use it instead of trying to calculate it. It should be noted that this function is only available in Webots Simulator and it works as a *God Mode* or human intervention, but since this mode is just necessary to fill the reward table and is not related to the actual training of the robot, this is not a problem and after successfully training the robot in the environment, it should work in a real version of the robot as fine as in the simulator. The following subsections explain how the reward functions for each low-level task are defined.

3.4.1 Corridor

The task of going through a corridor is the most basic one. Here the only goal of the robot is to go forward and be as much aligned with the center of the corridor as possible. To accomplish this, only the robot orientation and horizontal displacement, that is, its distance to the walls need to be considered. With this in mind, and after some trials with other rewards, the best reward achieved can be visualized in Figure 3.3.

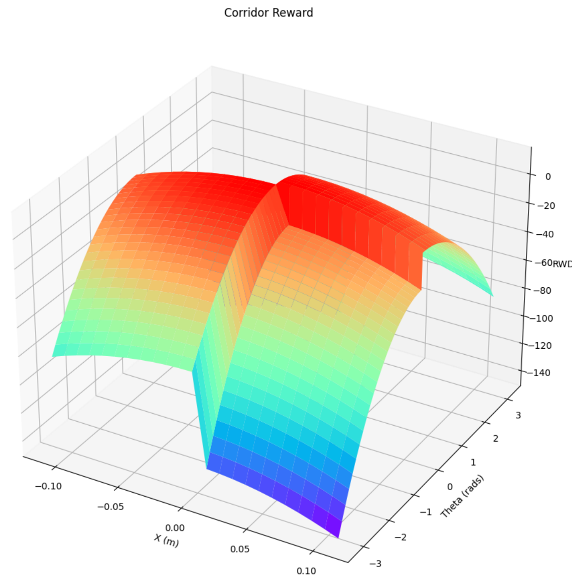
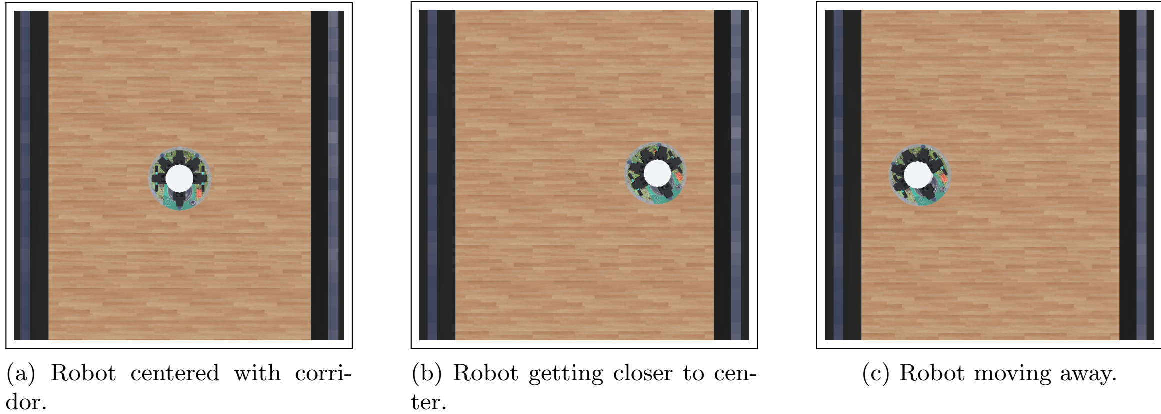


Figure 3.3: Corridor reward function representation.

As can be seen in Equation 3.1, the reward is calculated using two equations, one considered when the robot is centered or turned facing the center of the corridor and the other when it is moving away from the center. To better understand these equations, Figure 3.4 shows the three possible situations where the robot can be. In Figure 3.4a the robot is aligned with the corridor and centered, which means it is in the best scenario possible where it just needs to go forward, and so has the maximum reward of all states since both its x and θ values are zero. However, in Figure 3.4b and Figure 3.4c, the scenarios are a little different. In the first one, the robot orientation allows it to get closer to the center of the corridor just by going forward and when reached the center can correct its orientation to keep itself aligned with the corridor, but, in the second situation, it needs to correct itself first in order to move towards the center of the corridor and, after that, the scenario changes to one similar to the previously mentioned. Alongside the robot orientation, the horizontal displacement is even more important since it defines if the robot is too close to a wall or in a safe position in the corridor and, because of that, it has much more importance in the calculation of the reward.

$$\begin{cases} -400 * (4 * x^2 + 0.02 * \theta^2) + 15, & \text{if } (x \geq 0 \land \theta \geq 0) \lor (x \leq 0 \land \theta \leq 0) \\ -400 * (4 * x^2 + 0.03 * \theta^2) - 10, & \text{else} \end{cases} \tag{3.1}$$



(a) Robot centered with corridor.

(b) Robot getting closer to center.

(c) Robot moving away.

Figure 3.4: Three possible scenarios.

With the reward function defined, the 30 cm wide corridor from Figure 3.5 was created and used to fill the reward table with a resolution of 1 centimeter and 1 degree, that is, the robot is put in 23 positions on the corridor (from -11 to 11 centimeters so that it does not hit walls) and rotated 360 degrees (from -180 to 180 degrees), however, in this case a conversion is made to keep the orientation between -90 and 90 degrees since, even though the orientation is not the same, the state is because of the symmetry of the corridor walls. An example can be seen in Figure 3.6b, where the left robot's position is (-0.07, 160) and the right one is (0.07, 20) and both measure the same state (when placed alone, otherwise one would detect the other).

Besides this, dead-ends are considered as well and so, the robot goes through those same positions but at some distances from a wall in its back, as can be seen in Figure 3.6a. In these cases, since there is an extra wall, the symmetry ceases to exist and so the 360 degrees are considered. This allows the robot to turn around when it reaches a dead-end and to fill more states in the reward table that could be essential later.

Even though the table is constituted by 46656 states, almost 43000 states are left unfilled, which means almost 92% of the table, and so, the table size is not worrisome, as it can be trimmed to just the filled states after training. During training those states are never achieved by the robot and so they are just left with a very low reward.

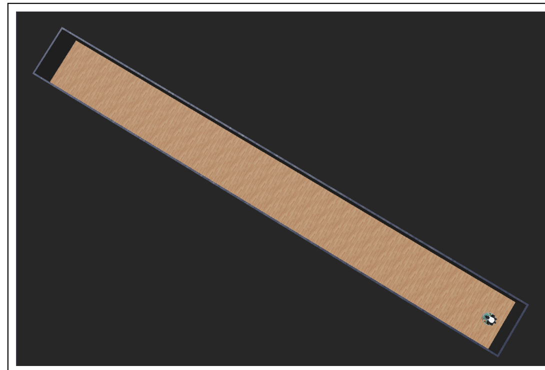
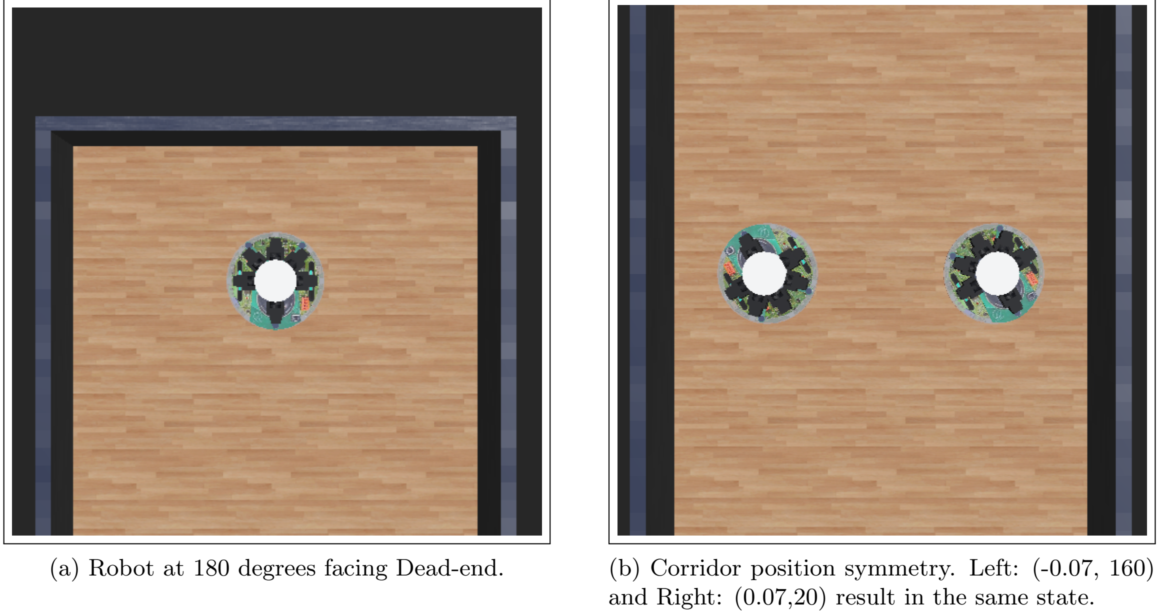


Figure 3.5: Corridor maze.



(a) Robot at 180 degrees facing Dead-end.

(b) Corridor position symmetry. Left: (-0.07, 160) and Right: (0.07,20) result in the same state.

Figure 3.6: Corridor special situations.

3.4.2 Corner

When solving the task of turning in a corner, two different approaches were considered: one where a new reward function is developed and the robot trained just like in the corridor and other where a corner is considered to be the begin of a new corridor and so, a transformation of the already trained table from the corridor is done in order to obtain the table for the robot to turn in a corner.

While in a corridor there are just two variables that need to be considered, however, for the robot to be able to do a corner there are three different variables present: X, Y and θ. Taking as an example the corner to the right from Figure 3.7a, the goal here is to get around in the best way possible. Being the local axis represented in the middle of the corner, in this first approach, a reward function is designed considering the three variables previously mentioned and is split in three main sections the robot can go through when getting across a corner:

- Section 1 - Corner entrance. Equation 3.1, from corridor, is used as this is part of it.
- Section 2 - Most of the area of the corner. The robot should go through this zone while crossing the corner. Uses the Equation 3.2.
- Section 3 - Danger zone. This section, composed of two areas in the corner, is the one to be avoided and uses the Equation 3.3 which results in very low rewards.

In Figure 3.8 can be seen a graphical representation of the composition of those three rewards into one. As it is composed by three variables it results in a four-dimensional plot which cannot be fully represented in the three-dimensional world and so, four angles are considered for demonstration purpose only:

- Figure 3.8a mimics a situation seen in the corridor reward as well. When $y \geq 0$, the robot is moving closer to the center and so gets a higher reward, however, if it passes

to a position with $y < 0$, the reward decreases a lot as it starts moving away from the center.

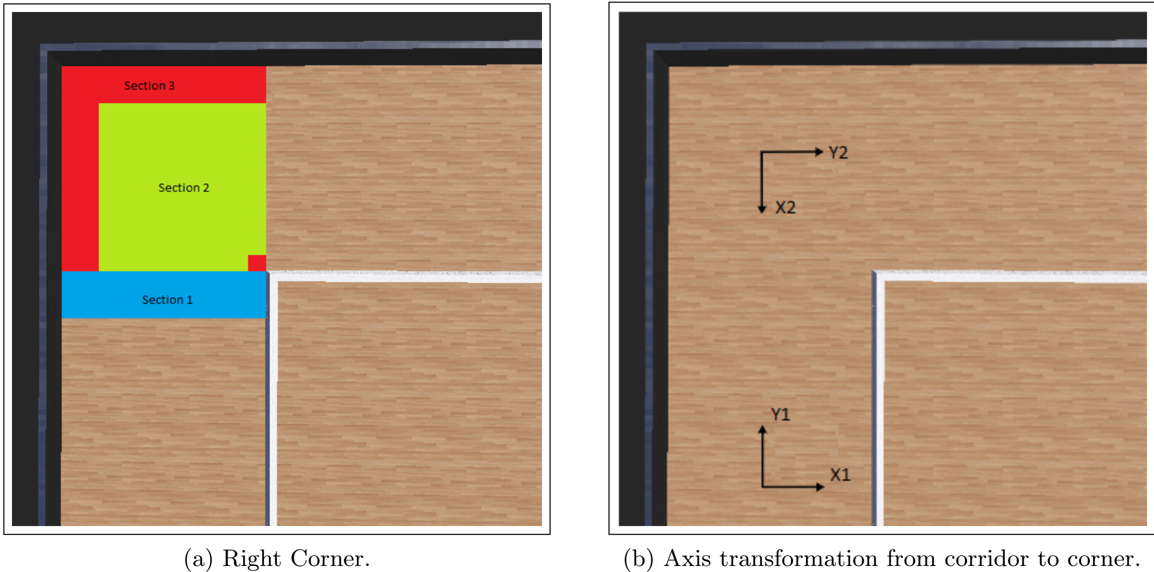
- Figure 3.8b represents one of the worst situations since the robot is facing the opposite direction to the corner. This is a good representation of the separations between Sections 2 and 3. The reward is almost always the same in the whole Section 2 since the best action is to turn around and so the robot position is not very important.

- Figure 3.8c represents the situation where the reward is the best. This is the case where the robot is turned 90 degrees to the right and all it needs to do is go forward and so, it increases as the x value increases.

- Figure 3.8d results in a similar graph as Figure 3.8b since that only when angles are between 0 and -180 degrees the robot is considered to be in a good orientation when in a corner to the right.

$$\begin{cases} (-400 * ((x - 0.11)^2 + 2 * y^2) + 10) + (|\theta| * 180/\pi)/20, & \text{Section 2 moving to center} \\ (-400 * ((x - 0.11)^2 + 2 * y^2) - 8) + (|\theta| * 180/\pi)/10, & \text{Section 2 moving away} \end{cases} \tag{3.2}$$

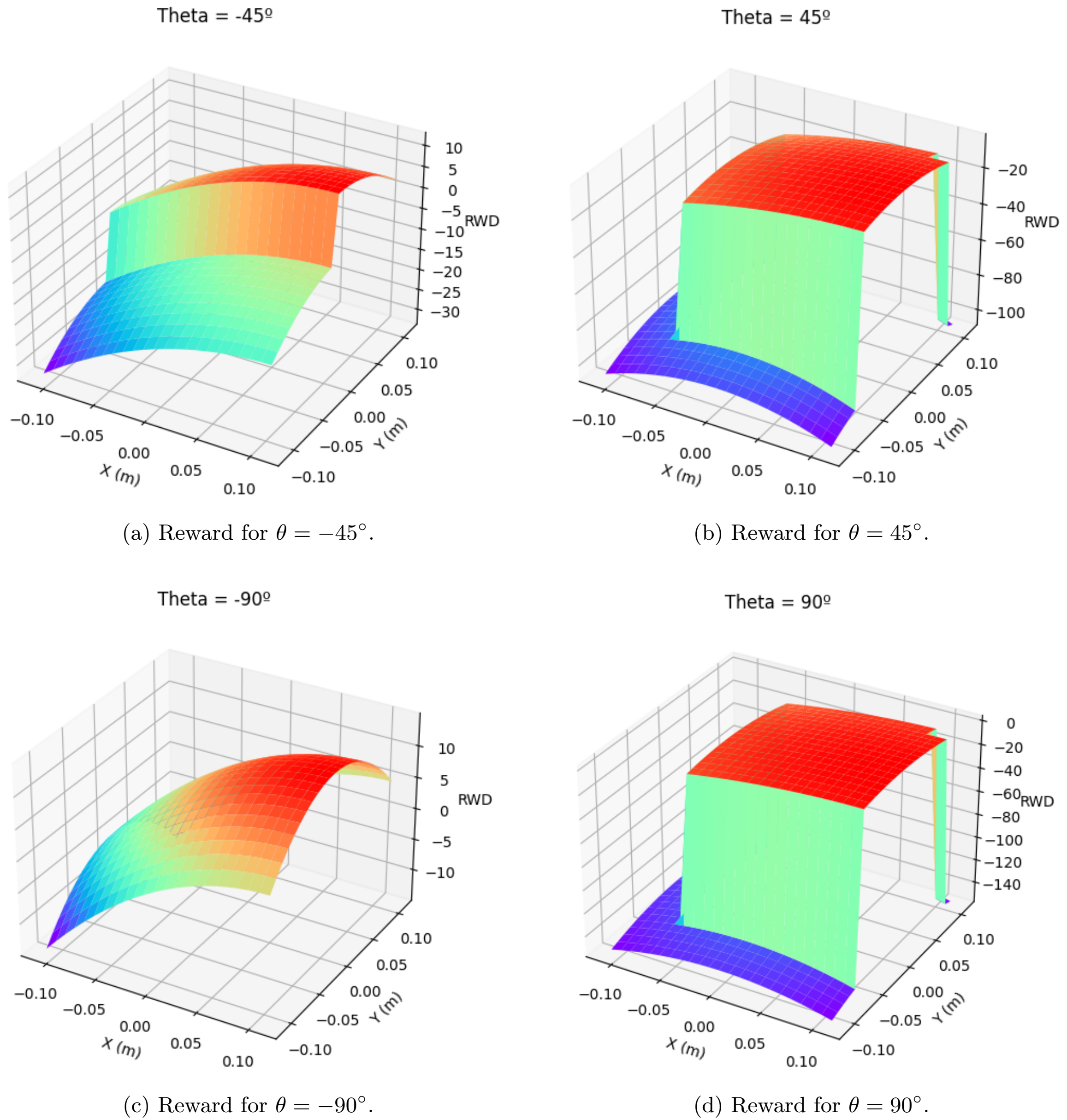
$$(-500*(2*x^2+2*y^2)-40)-(\theta*180/\pi), \text{if } (x \leq -0.07 \vee y \leq -0.07 \vee (x \geq 0.1 \wedge y \geq 0.1)) \wedge \theta > 0 \tag{3.3}$$



(a) Right Corner.

(b) Axis transformation from corridor to corner.

Figure 3.7: Corner sections and axis transformation.



(a) Reward for $\theta = -45°$.

(b) Reward for $\theta = 45°$.

(c) Reward for $\theta = -90°$.

(d) Reward for $\theta = 90°$.

Figure 3.8: Corner reward function representation.

For the second approach it is considered that a corner is the junction of two corridors. With this in mind, the table from the corridor can be transformed to suit the corner, assuming it is just the beginning of a new corridor and so it is just necessary to convert the corridor states to match the corner ones. A visual representation of this transformation is shown in Figure 3.7b, where the local axis x_1 and y_1 are transformed to match x_2 and y_2. The y_1 component makes no difference to the state along the corridor but does make in the dead-ends. So that the robot moves away from the walls, this component is considered and the transformation is made as if the robot was moving away from a dead-end. In order to fill the table for the corner to the right the Algorithm 5 has been developed.

Algorithm 5 Algorithm used to transform corridor table to right corner table.

```
 1: Load corridor R-table
 2: for p in Corner Positions do
 3:     if robot inside corner then
 4:         θ = θ + π/2
 5:         Transform current position to match a 90-degree corridor: P = (y, x, θ)
 6:     else                                        ▷ Robot is in Section 1 - Corner entrance
 7:         Cut θ to −90 ≤ θ ≤ 90        ▷ Symmetric positions. Subsection 3.4.1, Figure 3.6b
 8:         P = (x, y, θ)
 9:     Obtain corridor state in position P
10:     Select the best action for that state
11:     Select the Q-value of that action in that state
12:     Read current state from sensors
13:     Add the corridor Q-value to the corner table in the corresponding state and action
```

3.4.3 Doors

For the tasks of crossing doors, only the approach of transforming the corridor table is done here, following the same idea as the second approach for solving the problem of crossing a corner. At first, four different types of doors were considered:

- Front-Right door: Figure 3.9a, is a door where the robot can either turn right or go forward.
- Front-Left door: Figure 3.9b, is the symmetric door to the Front-Right one.
- Left-Right door: Figure 3.9c, is a door where the robot cannot go forward but can go left or right.
- Front-Left-Right door: Figure 3.9d, is a door where the robot has the three options: go forward, turn left or turn right.

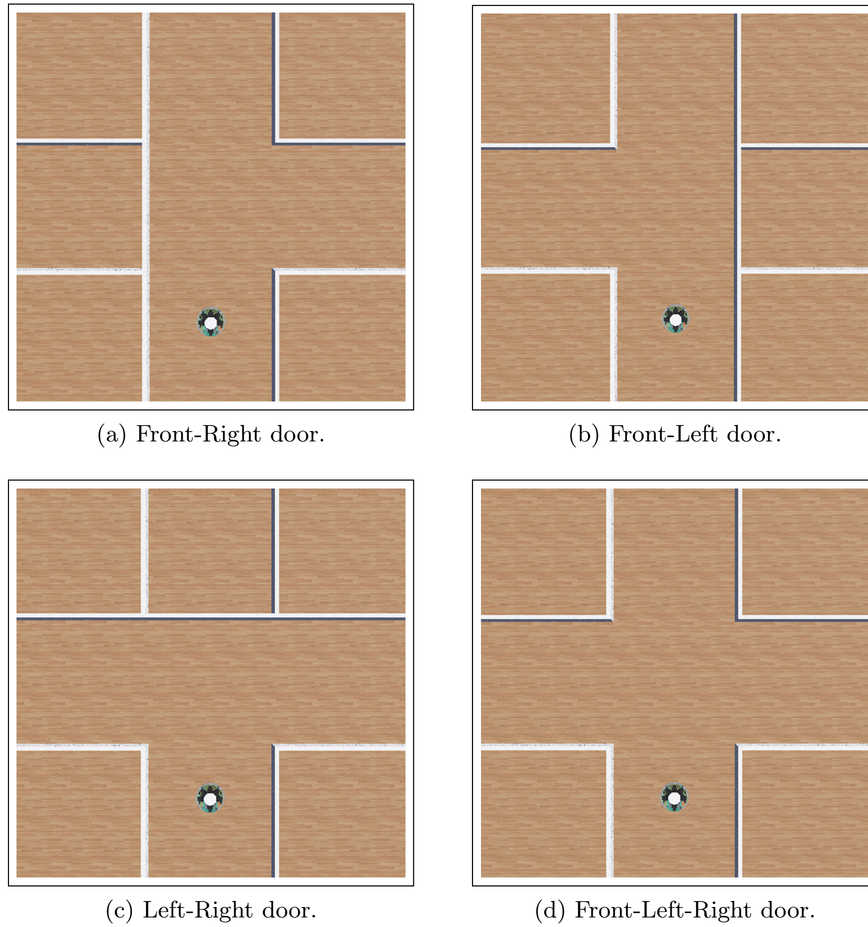
However, analyzing each door, some situations have to be discussed. Firstly, the Front-Left-Right door has a problem which is the 4-way symmetry inside it. Since the robot sensors can just measure up to 30 cm and this door can be split into four equal triangles, as can be seen in Figure 3.10, the robot cannot understand where it is and so cannot perform very well in some situations, so this type of door has been discarded.

The other three types of door are considered and have some interesting aspects as well. When comparing the Front-Left and Front-Right doors, it can be seen that the actions of turning left and turning right are symmetric, and so only one of the actions is considered when creating a new table from the corridor and then symmetry is applied to that table in order to generate the symmetric one to use on the other door.

The same happens to the Left-Right door. Even though there is just one of these doors, it is symmetric in its actions and, for that reason, just the table to turn right is created from the corridor table, being the table to turn left generated with symmetry.

Having already the tables to turn left or right in all the considered doors, there are just missing the ones where the chosen action is to go forward. At first two tables have been created, one for the Front-Right door and another for the Front-Left but, since the doors are

symmetric, all the new states from each door will be symmetric as well, and so, they can be together in the same table without conflict, resulting in just one table to move forward in both doors and corridor.



(a) Front-Right door.

(b) Front-Left door.

(c) Left-Right door.

(d) Front-Left-Right door.

Figure 3.9: Types of doors.

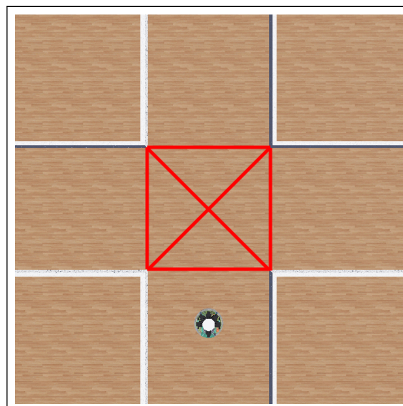


Figure 3.10: Four-way symmetry in Front-Left-Right door.

3.5 Performance Evaluation

After having the corridor reward table filled, the robot is trained in the environment from Figure 3.5 and, using the Webots Simulator *Fast Mode*. Given that after some time the robot is almost always in the center of the corridor, and so that it can converge from multiple positions, using the *Supervisor* mode from Webots the robot is placed in random positions and orientations in the corridor every time it reaches the center.

After approximately 17500 hours of training, in *Fast Mode* which is equivalent to approximately 215 real hours, the robot achieves a very good policy where it can converge from every position to the middle of the corridor easily. It should be noted that this amount of training was not done all at once. After each piece of training in which the results proved to not be satisfactory, the R-Table was loaded and a new piece of training was performed until reaching the desired results, presented below.

To demonstrate its convergence, the robot is placed in 6 different positions (from -9 to 9 cm away from the center) with 8 different angles (from -60 to 60 degrees) and always converges in a space of about 30 cm, as shown in Figure 3.11. In Figure 3.12 is exposed the behaviour of the robot when crossing the corridor. As can be noticed the robot is never fully aligned with the center of the corridor and always 1 cm away. This happens because of the discretization applied to the sensors, since their readings in the center and up to 1 cm away result in exactly the same state. The presence of sensor noise and wheel friction and slipperiness has an impact on the robot behaviour as well, however, this is most noticed when it turns around after reaching the end of the corridor. As witnessed in the same image, the robot does not always take the same path when turning back which happens because it does not reach the corridor end through the same position every time and so chooses different paths when turning. Nonetheless, it is acknowledged that with more training focused on corridor dead-ends, the robot would perform even better when reaching them.

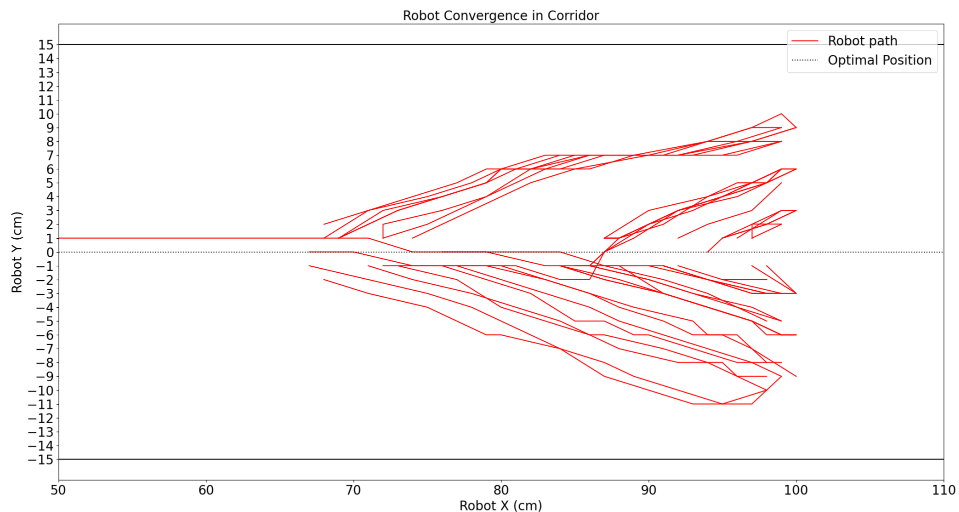
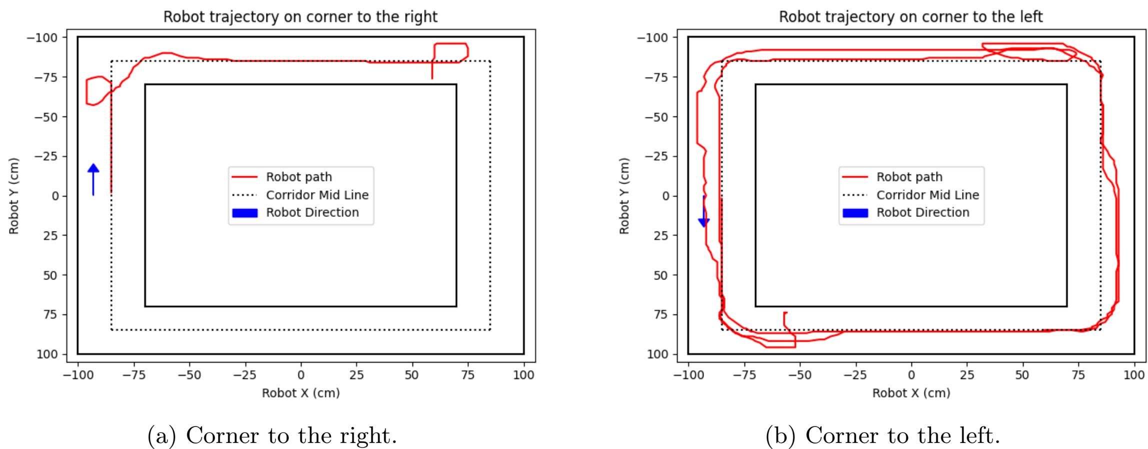


Figure 3.11: Robot convergence to center of corridor.

Figure 3.12: Robot deviation from center of corridor.

As stated in Section 3.4, two approaches have been done when solving the problem of turning in a corner. For the first approach, even though multiple pieces of training were performed over extended periods and various reward functions were specified, the robot was never able to solve the corner situation. In Figure 3.13 is demonstrated the path the robot follows as it tries to turn both left and right corners and, as is observable, none of them is complete with a good trajectory. Figure 3.13b shows a trajectory to the left where the robot made a few corners but, as it is perceptible looking at the top-right corner, it failed and wandered randomly until figuring out it was in a corridor and changing to the corridor task. The same way in the bottom-left corner the robot simply did not stop turning until it hit a wall. From Figure 3.13a it is clear that the robot cannot perform corners to the right with the trained table.



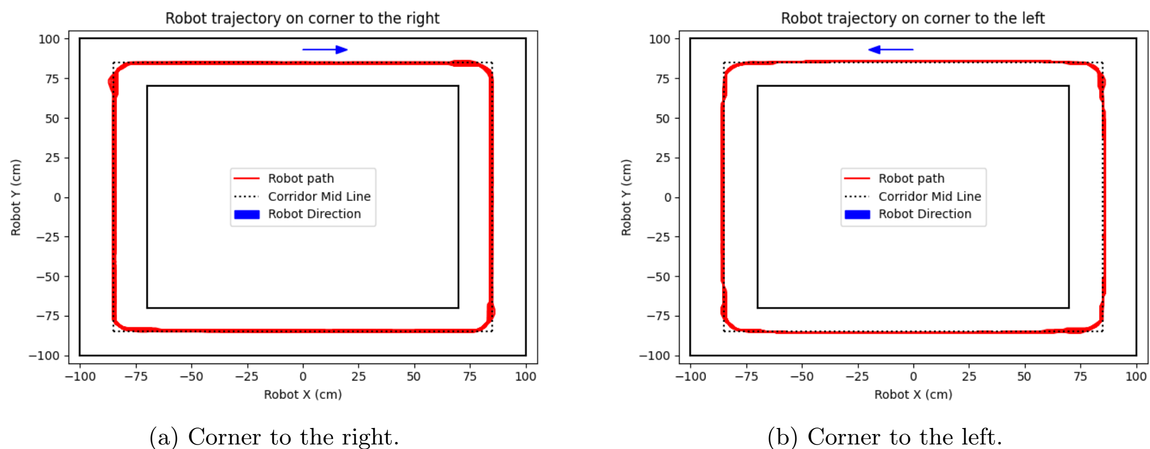
(a) Corner to the right.

(b) Corner to the left.

Figure 3.13: Corner trajectories with training.

Moving to the second approach, after transforming the corridor table into the corresponding one for the right corner, an evaluation of the robot behaviour has been done. Assuming that a corner is just the beginning of a new corridor, together with the convergence capacity of the robot in corridors, can make it perform the task of crossing a corner fairly well. The resulting trajectory of the robot crossing multiple times a corner to the right is shown in Figure 3.14a. As can be noticed, the robot follows a very good trajectory given that no more training was performed besides the already done in the corridor.

It can be perceived that the left corner is symmetric to the right one and, with the good performance obtained in the right corner, the table for the left one is obtained resorting just to this symmetry. In Figure 3.14b is presented the trajectory performed by the robot doing the left corner and, as can be observed, it is identical to the one performed in the right one. Instead of generating a new table to perform a corner to the left, the symmetry could be applied directly to each state the robot is in, executing the symmetric action of the obtained. Although this eliminates an entire table, it would add extra computational weight to the algorithm.



(a) Corner to the right.

(b) Corner to the left.

Figure 3.14: Corner trajectories with table transformation.

Lastly, there are the tables to cross the doors. With all of them fulfilled, the behaviour of the robot is evaluated in the maze from Figure 3.15 by repeatedly performing actions in doors, resulting in the trajectories from Figure 3.17. Each sub-figure represents an action that the robot is able to perform and, as can be verified, it goes through every considered door multiple times with some robustness.

At first, one table per action was used and so, six tables, just for the doors were necessary. These, together with the ones to move in a corridor and cross corners, make up a total of 9 tables for the robot's exploratory behaviour. However, given that the same action in all the doors and corner (turn right for example) is performed similarly, all the tables for that action were joined into one, ending up with just three tables: one to move forward in doors and corridors, one to turn right in doors and right corners and, finally, one to cross doors and corners to the left. With these tree tables, a new evaluation has been made in a maze that could represent all the three types of doors and the resulting trajectories are demonstrated in Figure 3.16. The robot performs a good trajectory across all the maze multiple times, whereby these are the final tables used to perform the low-level tasks.

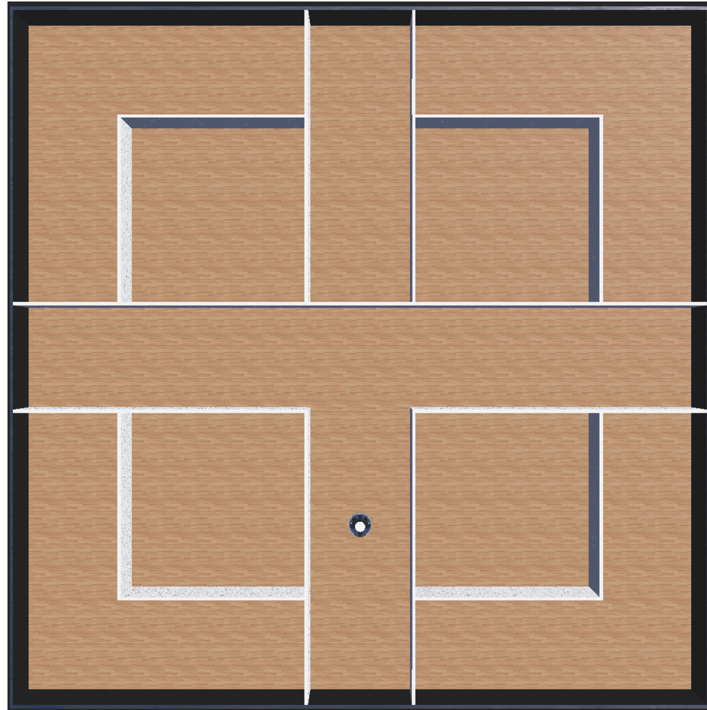


Figure 3.15: Maze in T-shape to evaluate performance on doors.

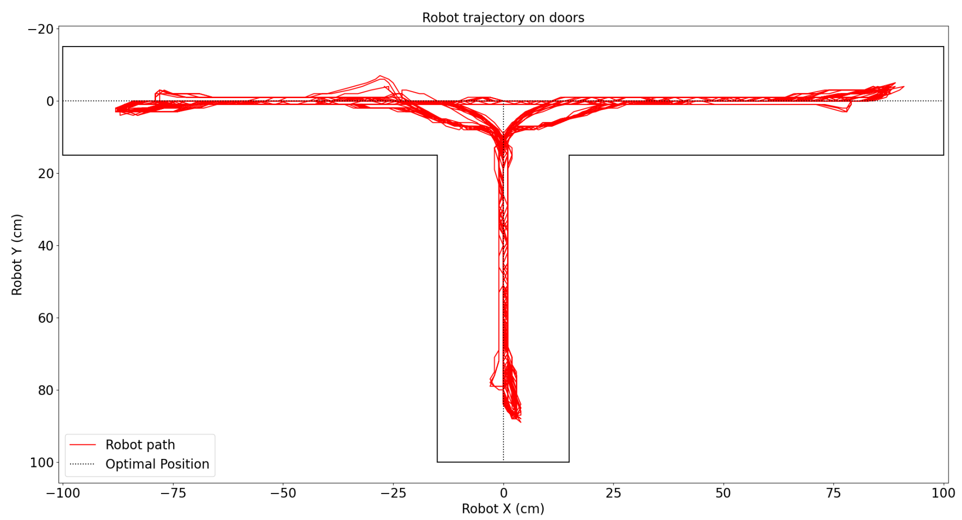
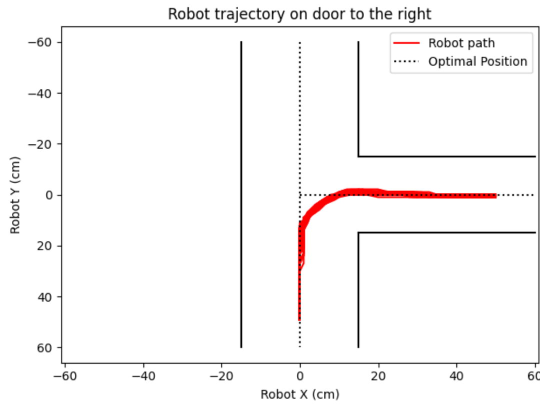
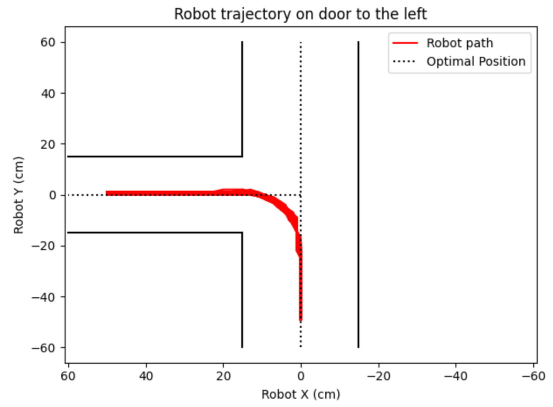


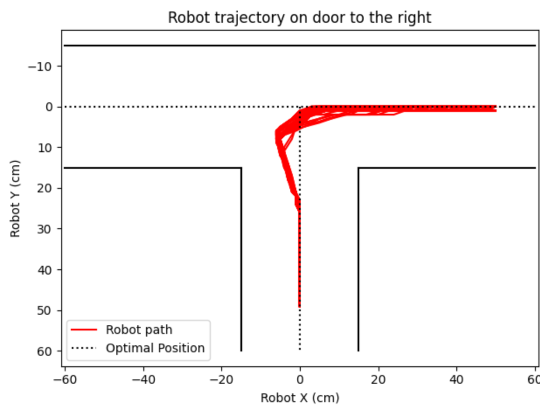
Figure 3.16: Robot trajectory on maze with all doors combined.



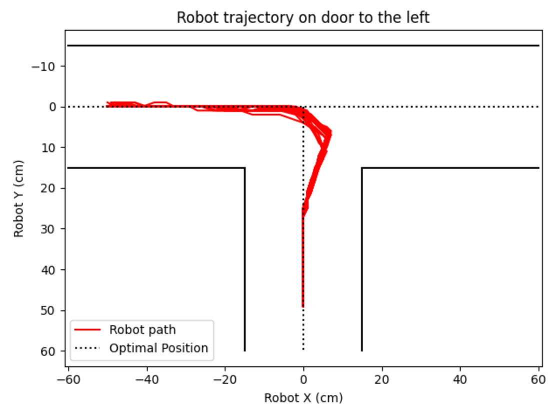
(a) Turn right on Front-Right door.



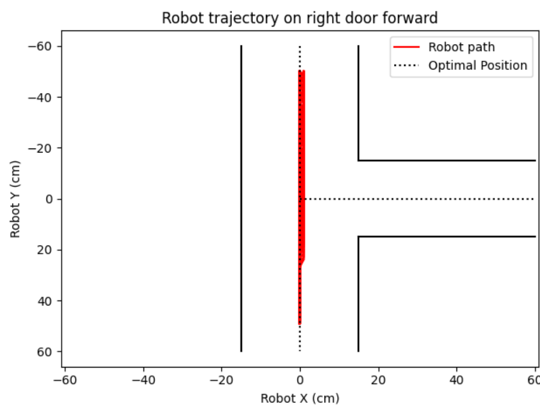
(b) Turn left on Front-Left door.



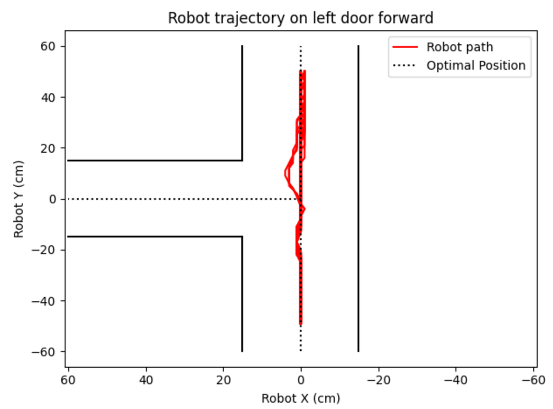
(c) Turn right on Left-Right door.



(d) Turn left on Left-Right door.



(e) Go forward on Front-Right door.



(f) Go forward on Front-Left door.

Figure 3.17: Robot trajectories in each door.

Doing a deeper analysis of the robot's behaviour it is concluded that it explores very efficiently any type of maze that is constituted by 30 cm wide corridors, 90-degree corners and the three types of doors referred above. To evaluate this efficiency, the robot is placed in the

maze from Figure 3.18 and evaluated for each one of the actions on the doors independently, while performing corners to both directions and moving in a corridor. The Figure 3.19 presents the trajectories made by the robot when performing 500 laps. The robot is not 100% efficient and sometimes can fail when turning left or right in a door. Looking at Figure 3.19b it can be seen that the right action has the highest fail rate, with 7% of fails. The average error for each action is presented in Table 3.4. For the 500 laps, the action of going forward never failed and the one of turning left failed just 0.8%. When analyzing the three images from Figure 3.19 it can be seen that the robot does not perform the same trajectory every time, being the ones with the most divergences the trajectories when turning right on doors. The behaviour difference between left and right actions has no other justification than sensor noise and wheel slippery since the table for the left actions is obtained with symmetry from the one for the right actions. Although that, it should be remembered that for all these tasks there was just one learning phase, in a corridor, whereby the final results are very interesting given the amount of training and the low error percentage.

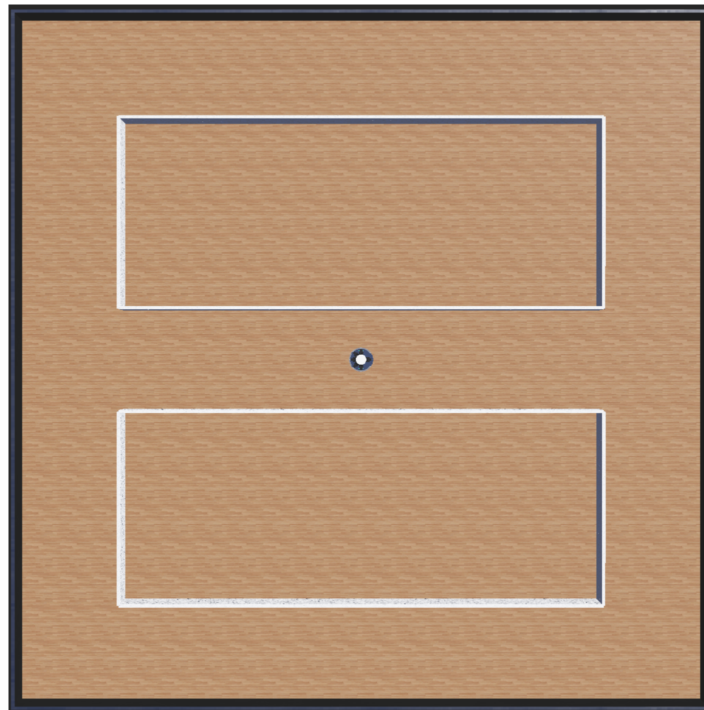
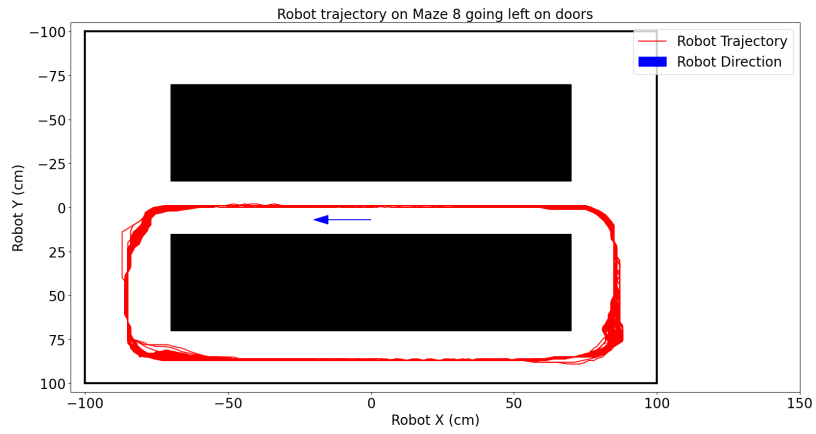


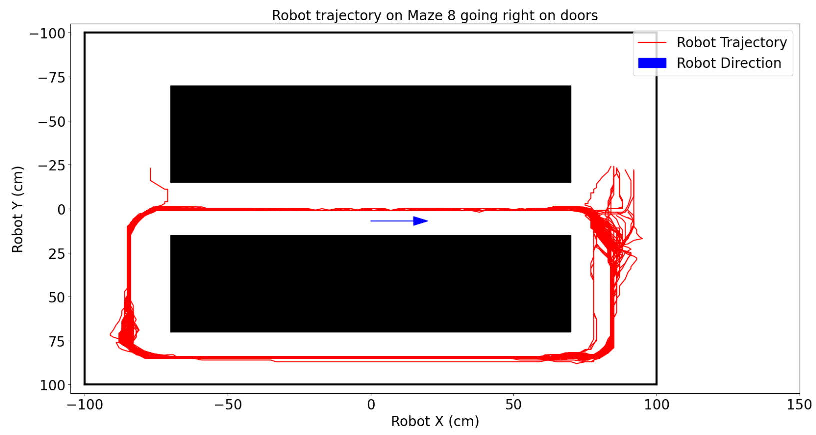
Figure 3.18: Maze for low-level evaluation.

Action on Door	**Fail Rate**
Forward	0.0%
Left	0.8%
Right	7.0%

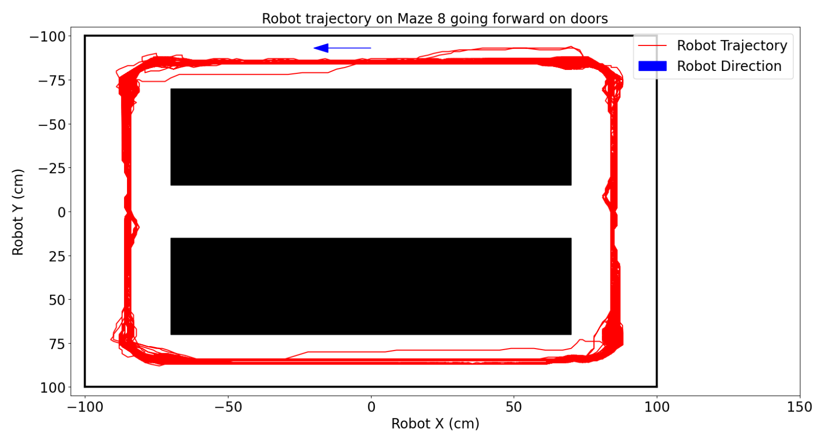
Table 3.4: Robot low-level efficiency.



(a) Left action on door.



(b) Right action on door.



(c) Forward action on door.

Figure 3.19: Actions on doors evaluation.

There is a need to mention that, even though the robot performs good trajectories across all the tasks, particular situations can happen in T-junctions where the robot struggles to get out of a door and stays there stuck in-between two states for some time before finally being able to continue on its path. This situation does not translate into a huge problem or disadvantage but is something to be taken into consideration. In the other two lower-level tasks (corridor and corners) no particular situations were found whatsoever.

3.6 Final Remarks

The reward specification for the task of moving forward in a corridor resorting to reward shaping allowed to take into consideration multiple variables (robot position and orientation) which, together with the action of placing the robot in multiple positions in the corridor during the learning phase allowed it to learn how to converge from every point to the center of the corridor. This ability ended up being very important when solving the remaining tasks as this convergence enabled the robot to be able to cross corners and doors with just the learning phase from the corridor.

The exploratory behaviour of the robot ended-up being very efficient and successful when it is placed in an environment that follows the rules and constraints stated earlier. Despite that, the robot is just endowed of wandering safely in a maze without any type of objective whatsoever. For that reason, the next chapter is introduced, in which a hierarchical approach will give the robot one or multiple objectives in a maze so that it navigates to them.

Chapter 4

Hierarchical Robot Navigation Approach

This chapter proposes a HRL approach to the navigation problem of a mobile robot in a maze where it learns to achieve goal locations based on task decomposition and the concept of topological navigation. Section 4.1 describes the hierarchical structure and the assumptions of the work carried out. Section 4.2 provides an insight on the RL algorithm that allows the robot to navigate to a specific goal location, as well as results obtained in two mazes of different complexity. Section 4.3 proposes a solution to the problem of multi-goal navigation based on the extraction of a topological map of the environment. Section 4.4 outlines an approach to solve the navigation problem in a dynamic environment where obstacles can appear and block paths. Finally, Section 4.5 summarizes the main results achieved and the implications for the objectives to be attained.

4.1 Hierarchical Decomposition

In HRL, the task is decomposed into a hierarchy of sub-tasks where policies at the top of the hierarchy call upon policies from a lower level. The hierarchical decomposition allows to reduce the original problem to a smaller set of related problems, being a promising solution to scale RL techniques to complex domains. In the context of this work, it is assumed the robot system is endowed with visual information provided by a camera that would allow the observation of the environment to detect locations. Visual information enables the representation of the environment through metric, topological and semantic maps, being commonly used in tasks such as path-planning, localization, and obstacle avoidance. The idea behind the work is to develop a navigation system that can automatically extract topological information about the environment and navigate to a goal location using RL. The challenges of visual navigation will not be addressed in this dissertation; however, it is assumed that the robot is able to recognize the target locations (goals), as well as each of the doors defined in the environment.

Having this assumption in mind, this section formalizes the hierarchical approach for solving the navigation task in a maze-like environment. The simulator resources are used to emulate the visual information and, in this way, to extract a graph-based representation of the environment where the nodes correspond to junctions and the edges encode the adjacency relations. In practice, the robot should detect some environmental cue that could distinguish one node from the other, regardless of the arrival point. The design is reduced to T-junctions

of identical geometry, i.e., 3-way intersection points where a decision is required. The edges are associated with intermediate policies to leave the transition region (e.g., turning left, turning right, or going straight ahead), followed by the primitive actions to cross a new corridor. These simpler problems were solved separately in the previous chapter and the results need to be recombined to find a solution to the original one.

Figure 4.1 provides an intuitive description of the HRL approach applied to the simulated navigation problem. Once learnt, the execution of the higher-level policy will determine the action to leave the T-junction at the current node. Control is passed to the sub-task that leads the robot out of the junction through a region of transition to the next corridor. When leaving the intersection, the IR-sensors are used to trigger control for the sub-task *Corridor*. These IR-sensors are used in the same way whenever the robot is in a corridor and reaches a corner, however, these situations do not require intervention of the higher-level and are performed autonomously. The visual detection of a new door terminates the corridor sub-task and passes the control back to the higher-level that chooses the next action to leave the T-junction. This process is repeated until the robot recognizes the goal location. This hierarchical decomposition divided the lower-level sub-tasks in the three categories presented below:

- **Automatic Actions**: actions performed by the robot without the intervention of the high-level layer.
- **Actions on Doors**: actions dependent on high-level intervention for decision making on doors.
- **Invert Direction**: action only invoked by the high-level that allows the robot to invert its direction whenever necessary.

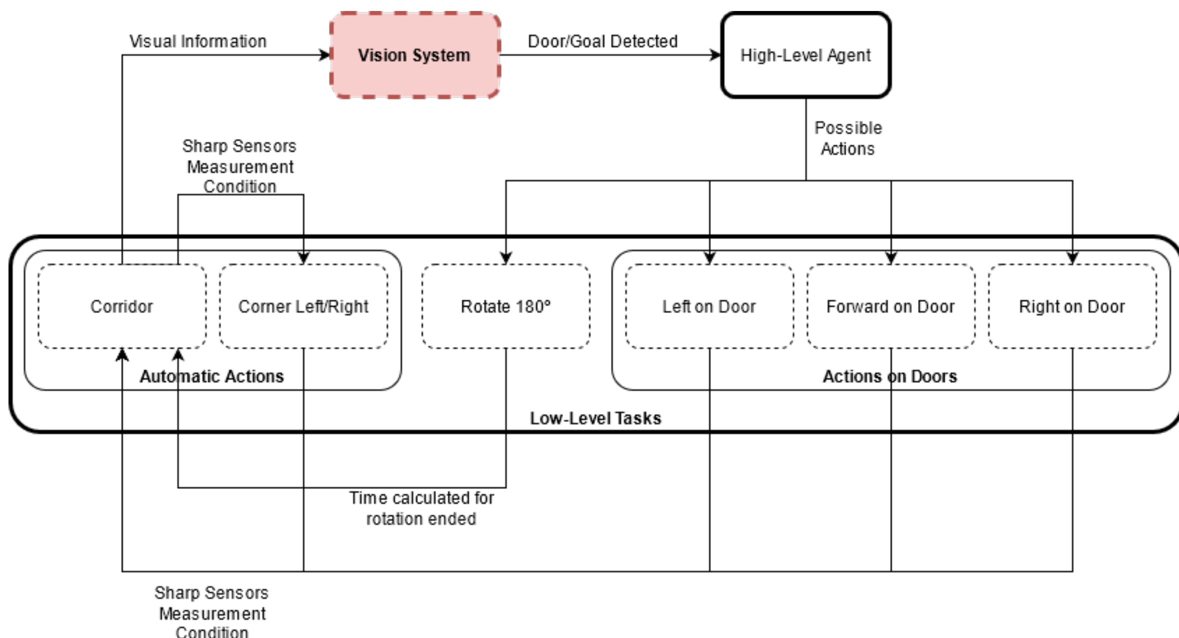


Figure 4.1: The hierarchical approach focuses on a sequence of sub-policies that appear both during training and execution.

The hierarchical structure provides multiple benefits, namely it allows to abstract actions at the top levels like *"move left to the next node"*. These abstract actions skip over large parts of the state-space terminating in a small subset of states, requiring less effort to learn a policy. At the same time, it also provides temporal abstraction at the higher-levels of the hierarchy. For the higher-level, the low-level policies are viewed as temporally extended actions, because once they are invoked tend to persist for multiple time-steps until the robot recognize a new door. This is a core concept of HRL that results in multi-step value bootstrapping when temporal difference algorithms are used. Another potential benefit of HRL is to make exploration easier by reducing the number of steps required to explore the state-space. Moreover, the hierarchical approach can provide effective solutions when transferring knowledge between mazes of varying design and complexity since it just requires training the higher-level policies, while reusing the low-level learned behaviours.

4.2 Higher-Level RL Problem

4.2.1 RL Problem Formulation

Unlike the discretization on the lower-level, the top-level RL problem is represented as a MDP with discrete states and actions. In this case, the states are associated with the doors that allow access to the T-junctions (three doors for each junction). Therefore, the possible actions to leave a junction will depend on the arrival point. This information is not available initially and it should be extracted from the environment through exploration. Given the specified hierarchy, the higher-level task is learned using standard off-policy Q-learning. In the end, the learned policy makes decisions about the actions to take in each door and the optimal sequence of doors/junctions the robot must follow to accomplish a specific goal.

In contrast to the low-level, here the table size will depend on the maze. Taking as an example the mazes from Figure 4.2, with the increasing number of T-junctions, the number of states will increase as well. In Figure 4.2a there are four T-junctions, but since each junction is composed by three doors, each associated to a state representing the sides from where the robot can come from, this first maze has a total of $4*3 = 12$ states. The maze from Figure 4.2b is somewhat more complex and has more junctions, and so the number of states increases. Here, there are a total of eleven junctions, whereby there are $11 * 3 = 33$ states. This is a bigger table compared to the one from maze 1 but nothing compared to the ones used in low-level.

As there is a need to identify each door so that the robot knows where it is on a T-junction and since a visual system is not implemented, the *Supervisor* mode has been used once again to obtain the exact position of the robot in the maze. With these coordinates and a text-file containing a configuration with each door coordinates and orientation, as well as the goal position, it is possible to determine each state and choose the adequate action. This is a simple approach to emulate the information that otherwise would be provided by the vision system.

When creating the new maps some constraints have to be fulfilled. After every corner or T-junction, there must be a segment of a corridor since the robot is not capable of detecting two junctions or corners in a row and will only detect them when coming from a corridor. Given the symmetry presented in the corridor, explained in Subsection 3.4.1, the robot will not know which direction it is going until it reaches a door, that is, a state and, only after that, it will follow the best path to the goal.

So that the robot learns multiple paths to a goal, as in GridWorld, it is placed in different positions in the maze, however, these positions must be carefully selected because they must be in a corridor as the robot is not capable of detecting a T-junction or corner when it has surpassed the respective entrance and so, it would not know how to react in that situation. Finally, the last constraint to fulfill is the fact that the goal should not be very close to a wall since the robot learned in the low-level to move away from walls and dead-ends and so if the goal is too close to one wall, the robot might move away and never be able to reach it.

Figure 4.2: Mazes used to train and evaluate high-level performance (Maze 1 on the left; Maze 2 on the right).

Having already approached the hyperparameters in Section 2.1.3, in this problem the same values are used without further research, with the exception of the ϵ used in the ϵ-greedy. This value is set a bit higher from its original value of 0.1, being used here with a value of 0.3.

When it comes to the number of episodes, the type of maze where the robot is going to learn is the main factor to dictate how many are needed. In a more complex maze, that is, with more doors, the robot will need more episodes to reach an optimal policy. A good example of this is both mazes developed for the experiment. While in maze 1 the robot is trained with just 300 episodes and reaches optimal paths for all the starting positions defined, in maze 2 the robot needs 500 episodes to be able to perform the best path.

Finally, the last hyperparameter to be considered is the reward. Here the main aspect that counts is the amount of time the robot spends between two states since this is what dictates how a path is shorter than another. For example, a path can have many more doors, consequently, many more high-level states, and be shorter than another that consists of just one door and a very long corridor. With that in mind, the reward for each state is calculated using Equation 4.1. When the robot reaches the goal it gets a very high reward, however, in any other state, it will depend on the amount of time spent performing low-level tasks.

$$\begin{cases} 100, & \text{if } state = goal \\ -500 * (\frac{timeBetweenStates}{3600}), & \text{else} \end{cases} \tag{4.1}$$

4.2.2 Learning Evaluation

During the learning, some metrics are collected so that an analysis on the algorithm performance is done after finishing this phase. In Figure 4.3 are presented the number of visits to each state. In both mazes all states are visited multiple times, meaning the robot did explore the entire maze while learning. Although that, the most visited one is the goal which means that the exploration rate was not very high and, consequently, the robot did not wander a lot through the maze. If the rate was higher then probably some other state would have been more visited since the robot would have spent more time exploring. In the end it was concluded that this was not necessary as the robot visited all states in both mazes and proved to have a good behaviour as will be presented below.

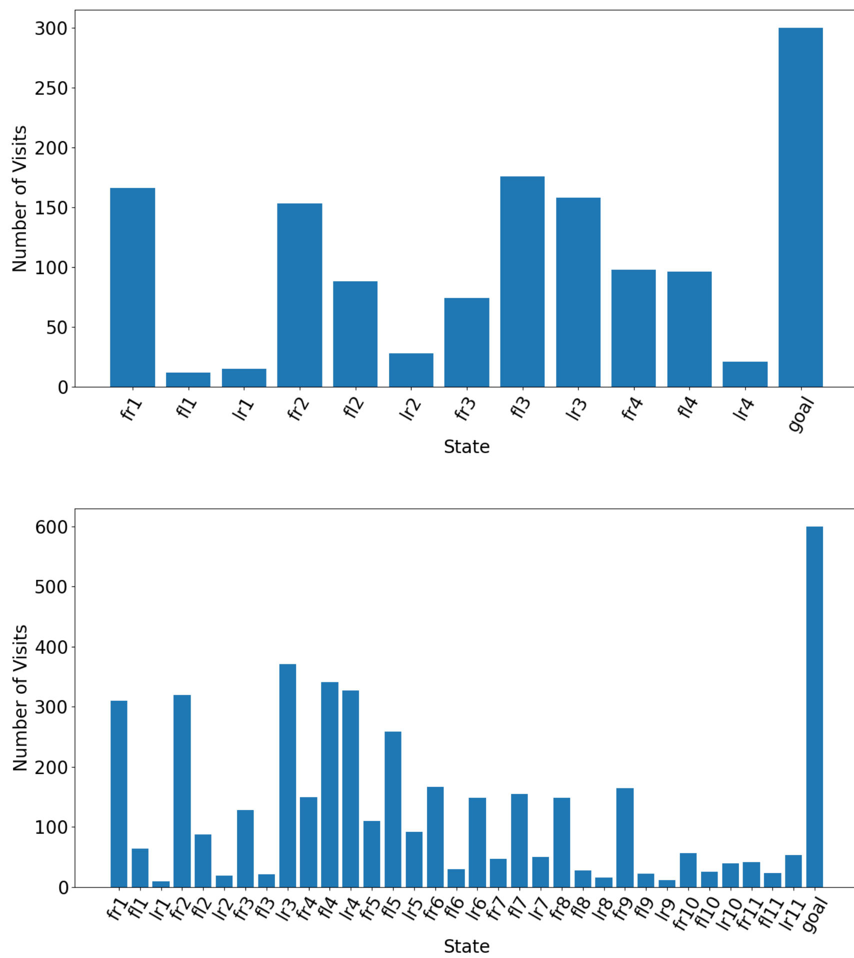


Figure 4.3: Most visited states on Maze 1 (left) and Maze 2 (right).

The plots from Figure 4.4 and Figure 4.5 are correlated with each other. As the reward for each higher-level action is calculated based on the time spent performing lower-level tasks, if the robot does not follow the best policy it will perform more steps than strictly necessary whereby it will spend more time in lower-level tasks which, in the end, translates into a lower reward. This way, the higher the number of steps, the worst the reward will possibly be. It should be noted that this relation might not happen since, as mentioned earlier, a path can

have many higher-level states and be shorter than another that consists of just one door and a very long corridor. Looking at both images from Figure 4.4, it can be concluded that the robot converges to the best path after some iterations. Since the exploratory rate is never zero, the robot will follow some policies different from the optimal in some iterations, which in the plots translate as the spikes in the number of steps. In Figure 4.5 are presented the reward sums for each iteration performed in both mazes. These plots show that this reward tends to stabilize in high values, meaning the robot is achieving the best sum of rewards it can when starting from its current position and navigating to the goal.

Some particular situations can occur, as mentioned in Section 3.5, and the robot might take longer to perform the desired lower-level task, consequently reflecting on the reward of a higher-level state and on the reward sum of the respective iteration, whereby, although its occurrence is very rare it should be taken into consideration. Not all starting positions are at the same distance of the goal and that is shown clearly in Figure 4.4a where it can be seen that the starting position three (in blue) requires the least high-level decisions and the starting position four (in cyan) requires the most.

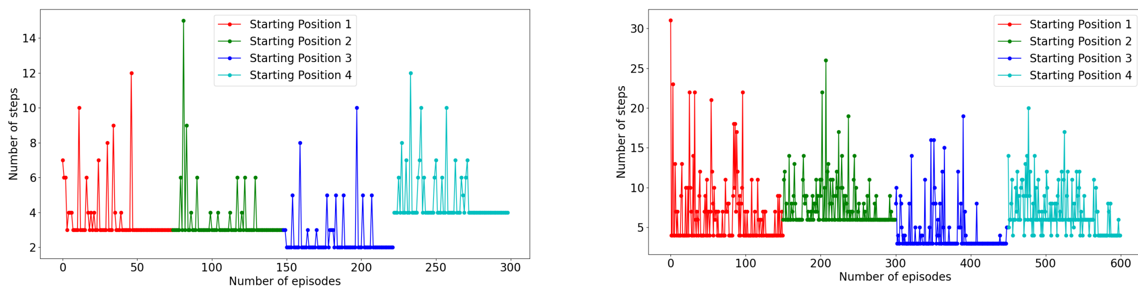


Figure 4.4: Steps per episode on Maze 1 (left) and Maze 2 (right).

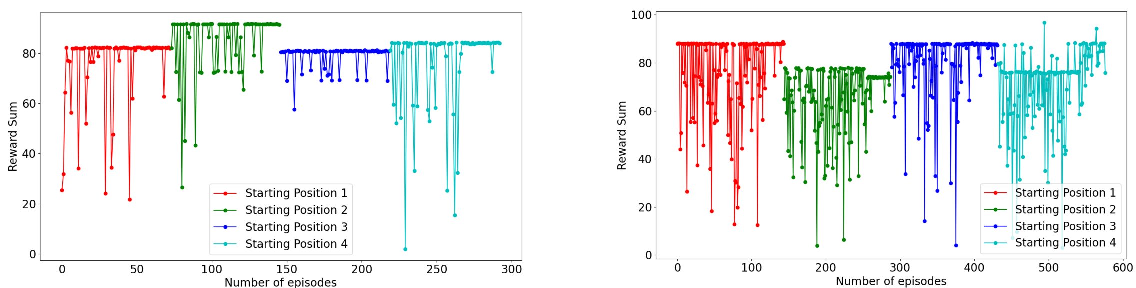


Figure 4.5: Reward sum per episode on Maze 1 (left) and Maze 2 (right).

4.2.3 Execution for a Single Goal

After the learning phase, the robot performance is visually analyzed by collecting its trajectories in the two mazes. Figure 4.6 presents ten runs for each starting position in both mazes, where each position is represented by different colors. The robot performs all the runs very robustly in both mazes. Looking at Maze 2, to solve the situation of starting in a position facing backwards the goal, the robot opts to go to a dead-end and turn back, being this the

shortest path it can perform with the abilities and information it has. In the next section, where a multi-goal implementation is discussed, this situation is resolved as the robot will always start from another goal, that is, a known position, whereby it can invert its direction, using the action specified below, and reach directly the goal. For Maze 1, as there are no dead-ends nearby, the solution found by the robot is to use the middle block as if it were a roundabout. In this maze, there are other situations where inverting the direction would have been the best option for the robot (e.g. starting position three) but, as stated before, the robot starts navigating in the lower-level until reaching a door, from where it will follow the optimal path to the goal.

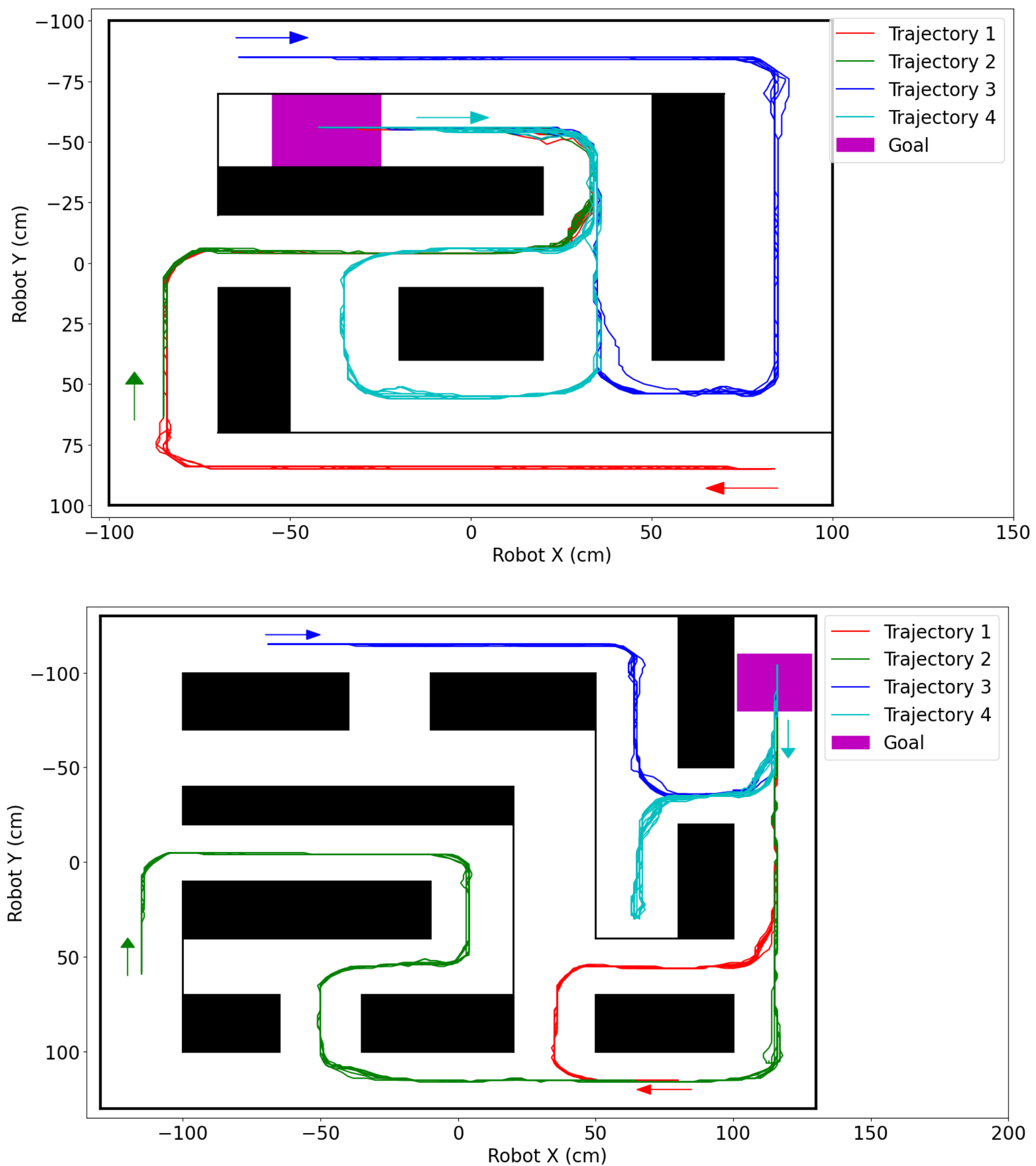


Figure 4.6: Superposition of the robot's trajectories to four different starting positions in Maze 1 (top) and Maze 2 (bottom).

In order to solve the situation that happens in starting position four from Figure 4.6b, an action for inverting the robot direction has been developed and used in Section 4.3. With this, the robot can rotate by itself a desired amplitude. As the objective here is to invert its direction, the robot will always rotate 180 degrees. To perform this task there is no need for training or reward specification as it is only based on mathematical equations. The final equation shown in Equation 4.4 is used to determine how long the robot must apply symmetrical speeds to the wheels in order to rotate the desired amplitude. Equation 4.2 and Equation 4.3 can be related to each other, reaching Equation 4.4, since $wheelSpeedRight$ and $wheelSpeedLeft$ are symmetric and can be used as $2 * wheelSpeedRight$.

Knowing the duration that the robot has to apply the defined speed to the wheels (symmetrically to each wheel), it can now perform the rotation of $\Delta\theta$ at a speed of $wheelSpeed$. It should be noted that this is considered an atomic action (is performed from start to finish without any interruption) and is controlled by the higher-level. This action must only be performed while the robot is in a corridor and never in a corner or T-junction as it might lead to errors since both situations are detected while the robot is approaching them and never when it is already performing them.

$$w = wheelRadius * \frac{wheelSpeedRight - wheelSpeedLeft}{axleLength} \tag{4.2}$$

$$\frac{\Delta\theta}{\Delta t} = w \tag{4.3}$$

$$\begin{aligned}
\frac{\Delta\theta}{\Delta t} &= wheelRadius * \frac{wheelSpeedRight - wheelSpeedLeft}{axleLength} \\
\Delta t &= \frac{\Delta\theta}{wheelRadius * \frac{wheelSpeedRight - wheelSpeedLeft}{axleLength}} \\
\Delta t &= \frac{\Delta\theta * axleLength}{wheelRadius * (wheelSpeedRight - wheelSpeedLeft)} \\
\Delta t &= \frac{\Delta\theta * axleLength}{wheelRadius * (wheelSpeedRight + wheelSpeedRight)} \\
\Delta t &= \frac{\Delta\theta * axleLength}{wheelRadius * 2 * wheelSpeedRight}
\end{aligned} \tag{4.4}$$

4.3 Multi-Goal Navigation

When thinking about a real-world application, the robot's ability to navigate to a single goal will not be of much use. Being able to go to a desired position and from there to another and so on is much more interesting and useful than just being able to go to one position in an entire maze. One approach to implement multiple goals could be having as many learning phases as goals and, consequently, one Q-Table per goal which would allow to create one table with the combination of all the others. Even though this could be achievable in small and simple environments, it would be impracticable in very large and complex ones with many goals and states, since with the increment of states both the Q-Table size and the learning time would exponentially increase. Alongside this, deducing a table from multiple others might bring some problems as there is no guarantee the optimal policies will be kept for any situation.

With this in mind, Algorithm 6 has been created for the robot to be able to save all the necessary information, during the learning phase, so that it can travel between multiple goals, in the same maze, resorting just to one data structure. Here, the robot learns the same way as explained in Section 4.2, but keeps a sense of the states it went through and, when it finds a goal that is not the starting one, the robot saves the path created, being able to navigate between those two goals whenever necessary after the learning is over. As this is all performed in the learning phase, the robot, in the beginning, has very exploratory actions which can lead to paths far from the optimal ones for some cases but, as the robot learns, the size of these paths starts to decrease and get closer to optimal. As the algorithm relies in the exploratory side of the robot during the learning phase, its starting positions must be carefully assigned in order for the robot to be able to explore the entire maze while learning how to achieve one goal. Therefore, if the full exploration of the maze is not guaranteed, the algorithm might not find the optimal paths between goals or might not find some of the goals at all. After the learning phase, some paths can be obtained by reverting others, that is, paths going from the *finishGoal* to the *startGoal* can be used to obtain the desired path from the *startGoal* to the *finishGoal*.

Algorithm 6 Multi-Goal algorithm to save information and train high-level to one goal.

```
1: Define hyperparameters and max episodes (600 was used in this experiment)
2: Define a few starting positions (other goals preferably) as well as the final goal
3: Initialize pathsList, an empty list to save paths between goals
4: Initialize statesList, an empty list to save all states for each path
5: Initialize Q-table with zeros for each state-action
6: while episode count < max episodes do
7:     if state = finalGoal then
8:         if (startGoal, finishGoal) not in pathsList then
9:             pathsList[(startGoal, finishGoal)] = statesList
10:        else
11:            if len(pathsList[(startGoal, finishGoal)]) > len(statesList) then
12:                pathsList[(startGoal, finishGoal)] = statesList
13:        Update Q-table in goal position with reward, r
14:        Choose starting position for next episode from the ones defined before
15:        episode count = episode count + 1
16:        Clear statesList
17:    else if state = otherGoal then
18:        if (startGoal, otherGoal) not in pathsList then
19:            pathsList[(startGoal, otherGoal)] = statesPath
20:        else
21:            if len(pathsList[(startGoal, otherGoal)]) > len(statesList) then
22:                pathsList[(startGoal, otherGoal)] = statesList
23:        Update Q-table in goal position and all actions with reward, r
24:        startGoal = otherGoal
25:        Clear statesList
26:    else
27:        Update Q-table using Q-Learning algorithm
28:        Add state-action pair to statesList
```

It should be noted that all the goals are detected the same way doors are, using visual information (as explained in Section 4.2). However, as each goal is in a corridor and the robot cannot distinguish between the two orientations, explained in more detail in Subsection 3.4.1, each goal is represented by two states, one facing each way of the corridor. For this experiment, four different starting positions that matched four goals were considered: *home*, *library*, *college* and *restaurant*, from which the robot learned how to reach the fifth goal in the environment: *gym*. These strategic positions allowed the robot to explore the entire maze and find connections between all the goals.

After ending the learning phase, the performance is evaluated by making the journey from *home* to the *library*, from there to a *gym*, then to a *restaurant* and, finally, returning *home*. As a pair of goals can have multiple paths between them, only the shorter is considered from both the reverse paths and the ones in the correct direction, retrieving just the best of all. When reversing a path, Algorithm 7 is used since the robot detects T-junctions by each door and so both the state and action must be converted to the corresponding ones. For example: *If the robot reaches Front-Left door in Junction 1 and turns left, the reverse action and state will be turning right in Left-Right door of that same junction.*

Figure 4.7a presents the trajectories between those goals, each represented by the color of the goal the robot starts from. It is clearly perceptible that the robot does not execute the best policy in some paths, such as the path between *restaurant* and *home*, in blue. This situation results from the lack of ability for the robot to turn back, that is, rotate 180 degrees and go in the opposite direction. To solve this, the high-level action for inverting direction is used, while Algorithm 7 is upgraded to Algorithm 8. The robot is trained one more time and evaluated in the same conditions as before (see Figure 4.7b).

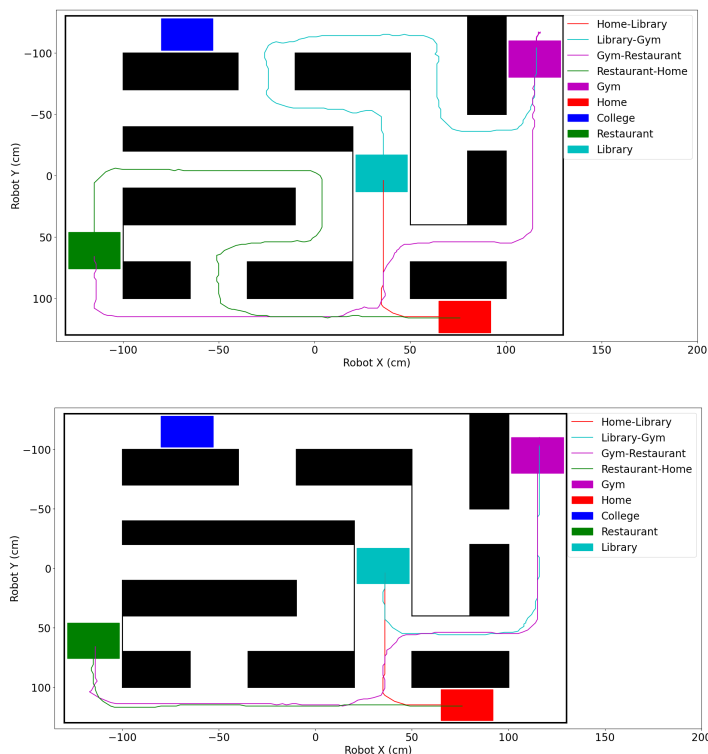


Figure 4.7: Multi-goal trajectories w/o action to invert direction (top) and with (bottom).

Comparing the two figures, without and with the reverse action, it can be perceived that not only the blue path between the *restaurant* and *home*, but also the cyan path between the *library* and the *gym* changed. In short, the robot opted to turn back and followed a shorter path to the desired goal.

Algorithm 7 Reverse path from finishState to startState.

```
1: Create newPath, an empty dictionary
2: newPath[finishState] = go forward                ▷ finishState is the starting goal
3: newPath[startState] = stop wheels                ▷ startState is the final goal
4: for (state, action) in path do
5:     if state != startState and state != finishState then
6:         newState, newAction = invertedStateAction(state, action)    ▷ Invert state and action as stated in the text above
7:         if newState not in newPath then
8:             newPath[newState] = newAction
```

Algorithm 8 Reverse path from finishState to startState, with invert direction action.

```
1: Create newPath, an empty dictionary
2: newPath[finishState] = go forward                ▷ finishState is the starting goal
3: newPath[finishState180] = invert direction    ▷ finishState180 is the starting goal in wrong direction
4: newPath[startState] = stop robot motors          ▷ startState is the final goal
5: newPath[startState180] = stop robot motors       ▷ startState180 is the final goal as well
6: for (state, action) in path do
7:     if state != startState and state != startState180 and state != finishState and state != finishState180 then
8:         newState, newAction = invertedStateAction(state, action)    ▷ Invert state and action as stated in the text above
9:         if newState not in newPath then
10:            newPath[newState] = newAction
```

4.4 Dynamic Behaviour

With the robot capable of completing the objectives and navigate between multiple goals in the maze, it is important if it can adapt its knowledge about the environment and change behaviour according to changes in the environment. Considering that the maze is not fully static and objects can appear and block the optimal path, the robot must adapt and privilege another path while that one is blocked. In order to do this, the more complex maze (Figure 4.2b) is considered, as it is simpler to demonstrate this problem since there are more places where to block the robot.

While going from a goal to another the robot goes through multiple junctions and doors. The robot will detect if a path has been blocked if it ends up in the same T-junction it lastly went through. For that, each junction must be labeled in order for the robot to be able to identify it independently the door it came from. With this in mind, each junction is labeled with a number and each door from that junction has a label composed by two letters (the

first letter of each possible actions that can be chosen in that door) together with the number associated to the junction. Therefore, if the robot reaches a door in T-junction number four where it can go *(f)orward* or *(r)ight*, then it is in the door with label *fr4*. This theory is applied to all T-junctions in the maze, with each door represented by *frX*, *flX* and *lrX*, where X is the number of the junction. Having these labels, the robot is capable of checking if it went back to the same T-junction or not by isolating the numbers at the end of each door label. Once again, the visual system is not implemented whereby these information is kept in a text-file and accessed the same way as each high-level state. In the real robot, the visual system should be capable of providing similar information whenever the robot reached a T-junction.

When the robot identifies a blocked path it needs to learn that that path it was following is blocked and so it should choose any other action available in the state prior to the blocking. To do this, the previously obtained Q-Table is updated, only in the state that leads to the blocked path so that the robot prioritizes another action and follows a different path from that state on. Depending on the discrepancy between the Q-Values of each action, this update could take more than one episode and, since these type of obstacles are considered dynamic, the robot does not discard the actions that lead to blocking states entirely, and so, a percentage of exploration is enabled, for that state only, so that the robot, sometime in the future, explores the blocked path and verifies if it is still blocked or is available again. For the robot to continue this exploration in a different execution, the information about the blocked doors is saved in a file, like a memory of what happened, and so, the robot will always seek for that blocked path and verify if it eventually becomes available.

Figure 4.8 provides a visual representation of maze 2 (Figure 4.2b) with a wall blocking the top corner (highlighted with a red rectangle). Figure 4.9a shows the initial trajectories the robot uses to reach the goal without any blocked paths. In Figure 4.9b it is demonstrated what happens to the trajectories when the top corner is blocked. As can be seen, the robot followed the same trajectories and just adapted the actions in the state that led to the blocked path so that it could reach the goal from another path.

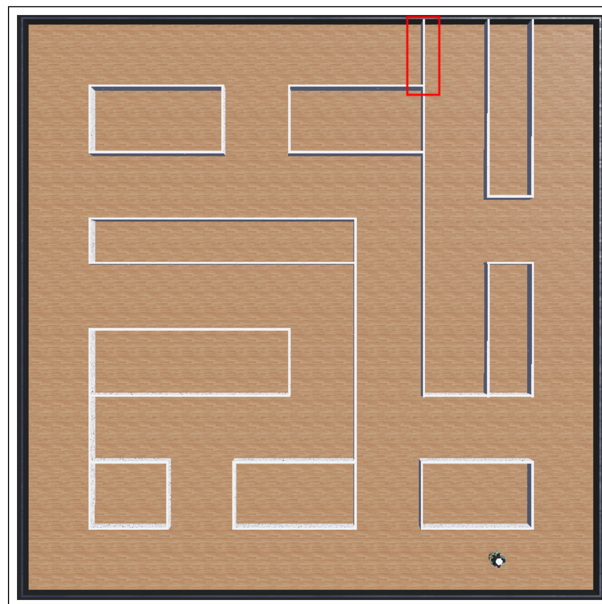


Figure 4.8: Top right corner blocked maze 2.

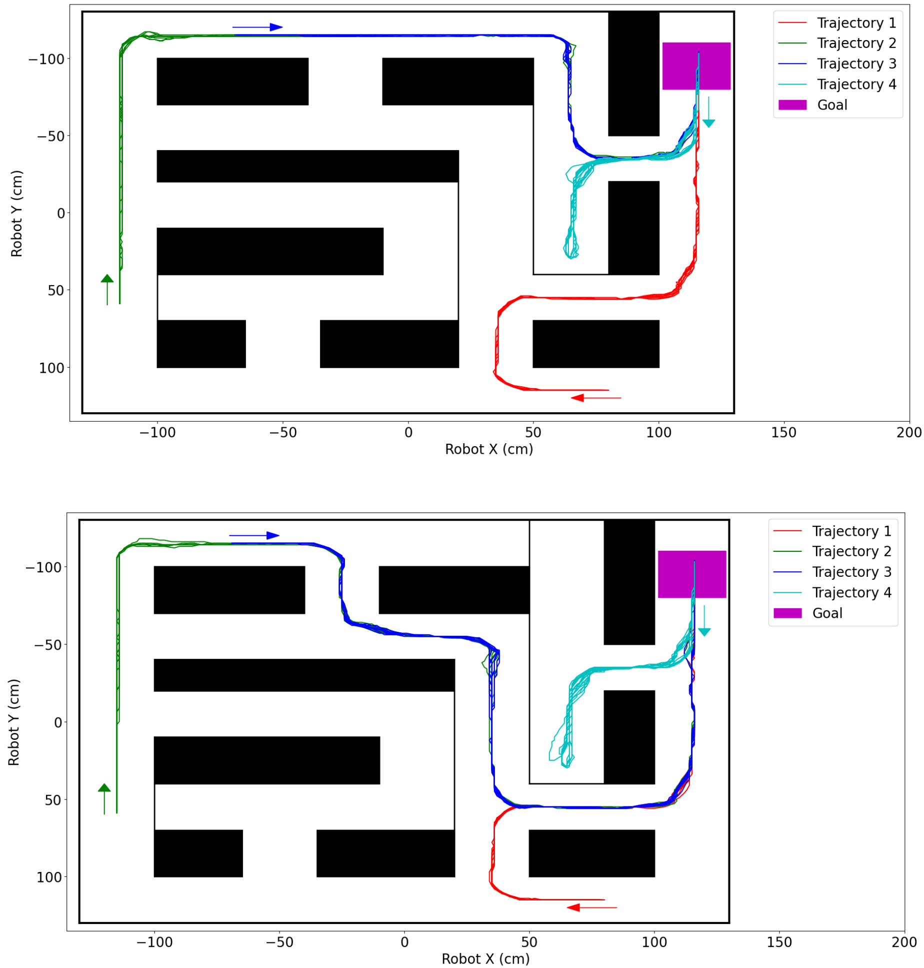


Figure 4.9: Robot normal trajectories (top) and adaptation to blocked path (bottom).

4.5 Final Remarks

The HRL approach discussed in this chapter allowed for a simplification of the whole problem into a set of smaller problems. This has the advantage of creating a more generalized approach to the problem to be solved since only the high-level must be trained whenever a new maze is used, respecting the conditions for good working of the low-level tasks such as the width of the corridor and having only 90-degree corners and doors.

The dynamic behaviour developed, even though not integrated with the multi-goal, worked very well (demonstrated in Section 4.4) and acts as a proof of concept whereby the technique implemented can be used for the robot to adapt to blocked paths in the environment.

The multi-goal implementation can easily be escalated to larger mazes with more goals, however it must be assured that the robot always visits every goal, when exploring the map, in order to create paths between all of them. The developed algorithm relies on a graph-like structure where each pair of goals has a set of decisions on all doors the robot must pass when going from the starting goal to the finish goal. The implementation of the task for the robot to be able to do 180-degree rotations gave it the ability to perform much shorter paths in some

situations when compared to the performance without this task (demonstrated in Figure 4.7).

Even though no vision system is actually implemented, there are many solutions for this, such as image recognition or bar-code reading to detect each door and goal. Another approach could be done using a similar Artificial Neural Network (ANN) to the one from Appendix B. The work described there was supposed to be implemented into the HRL problem in order for the robot to create a metric map of the unknown environment resorting just to its sensors and so be able to know where it was in each moment. However, given some time constraints it was not possible to accomplish what was previously foreseen. Despite that, a proof of concept using the Khepera-IV (Subsection 3.1.2) had already been done and so is explained in the appendix mentioned above (Appendix B).

Chapter 5

Conclusions and Future Work

5.1 Final Conclusions

The main focus of this dissertation was the navigation of a small omnidirectional mobile robot across multiple goals in a maze-like environment resorting to a hierarchical decomposition of the problem into smaller sub-problems. To solve this problem of navigation, a HRL framework has been used within a robot simulator that mimics real-world constraints such as wheel slippery and sensor noise. So that the robot had some self-awareness about its surrounding environment, 6 IR-sensors have been used with adequate voltage-to-distance conversion. Experimental validation within two different mazes was done proving that a hierarchical approach is very robust for these types of problems.

With this in mind, this dissertation provides the following contributions:

- Elementary functionalities were developed for the robot to be able to explore maze-like environments composed of 30 cm wide corridors, 90-degree corners and T-junctions.
- An hierarchical approach has been developed to solve the problem of navigation in a maze-like environment resorting to its topological representation. This approach allowed for faster learning of the higher-level decisions and to overcome some of the RL weaknesses, such as learning inefficiency and data complexity.. The results obtained in the simulation show that this learning approach for mobile robot navigation in maze-like environments is very effective and easier to adapt than classic RL as only the higher-level must be adapted in the new environments.
- The robot can navigate between multiple goals in a maze resorting just to the topological representation of the environment alongside the experience memorized during the learning.
- When navigating a dynamic environment the robot is endowed with the ability to adapt. If the optimal path gets blocked the robot can adapt its behaviour to choose another path, with sporadic verifications to the blocked one to check if it is still unavailable.

While developing this dissertation some limitations were encountered:

- Even though the simulator allows the replication of real-world environments, when testing the developed HRL approach in a real mobile robot, first it is necessary to implement a vision system to detect each door and goal. In simulation this emulated using the global

coordinates of the robot together with a text-file with the topological representation of the environment.

- When training the low-level tasks, mainly the corridor, several hours were needed (200+ hours), whereby, even with an hierarchical approach there is a need for a simulator to speed up the process and then transfer the learning to the real robot.
- Given the distance measurement constraints from the sensors used, everything is prepared to work with corridors exactly 30 cm wide and 90-degree corners and junctions. In order to have different sizes, some modifications are mandatory, mainly to the discretization of the sensors, so that each level has the correct equivalence between a 30 cm wide corridors and the one created. Even though not tested, after this modification to the discretization table everything should work the same way, always taking into account the maximum distance measurement of the sensors (30 cm).

5.2 Future Work

AI is a very large and continuously evolving area whereby there is a lot of work that can be improved here. Future work related to this dissertation can either be the continuation of the work already done and its improvement or take an approach into a new direction of investigation:

- Improvement of the low-level tasks (specially on doors) and respective rewards as well as implementation of new tasks.
- Implementation of reliable methods to track and detect doors and goals in real environments.
- Evaluate the R-Learning algorithm performance when compared to others and the impact of the hyperparameters on the learning.
- Evaluate the performance and advantages of the HRL approach when compared to other approaches.
- Improve the multi-goal algorithm to ensure connections between all goals exist and the shortest path is chosen.
- Join the dynamic behaviour algorithm with the multi-goal implementation.
- Implement the ANN in order to create a metric map of the environment and self-localize the robot.
- Evolve the entire work developed in this dissertation to a more abstract level so that the used algorithms can be applied in more generalized and complex environments as well as to different robots.

References

[1] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), 1943.

[2] Allan M Turing. Computer Machinery and Intelligence. *Mind*, LIX(236), 1950.

[3] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[4] R.S. Sutton and A.G. Barto. Reinforcement Learning: An Introduction. *IEEE Transactions on Neural Networks*, 9(5), 1998.

[5] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning, Second Edition: An Introduction - Complete Draft. *The MIT Press*, 2018.

[6] Diogo Vidal e Silva. Mobile robot navigation using reinforcement learning. Technical report, University of Aveiro, 07 2019.

[7] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford, 1989.

[8] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3–4):279–292, May 1992.

[9] G. Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

[10] John Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In D. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 211–217. Morgan-Kaufmann, 1990.

[11] Arryon D. Tijsma, Madalina M. Drugan, and Marco A. Wiering. Comparing exploration strategies for q-learning in random stochastic mazes. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, 2016.

[12] Arash Khodadadi, Pegah Fakhari, and Jerome Busemeyer. Learning to maximize reward rate: A model based on semi-markov decision processes. *Frontiers in neuroscience*, 8:101, 05 2014.

[13] Yannis Flet-Berliac. The promise of hierarchical reinforcement learning. *The Gradient*, 2019.

[14] Laurence A. Baxter and Martin L. Puterman. Markov Decision Processes: Discrete Stochastic Dynamic Programming. *Technometrics*, 37(3), 1995.

[15] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*, 1998.

[16] Thomas G. Dietterich. An overview of MAXQ hierarchical reinforcement learning. In *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)*, volume 1864, 2000.

[17] Thomas G. Dietterich. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 13, 2000.

[18] Shantanu Ingle and Madhuri Phute. Tesla Autopilot : Semi Autonomous Driving, an Uptick for Future Autonomy. *International Research Journal of Engineering and Technology*, 3(9), 2016.

[19] John J. Leonard and Hugh F. Durrant-Whyte. Mobile Robot Localization by Tracking Geometric Beacons. *IEEE Transactions on Robotics and Automation*, 7(3), 1991.

[20] Spot | Boston Dynamics, 2021.

[21] Wikipedia contributors. Gladiator tactical unmanned ground vehicle — Wikipedia, the free encyclopedia, 2021. [Online; accessed 21-June-2021].

[22] Wikipedia contributors. General atomics mq-9 reaper — Wikipedia, the free encyclopedia, 2021. [Online; accessed 21-June-2021].

[23] Phantom 4 - DJI, 2021.

[24] Wikipedia contributors. Pluto plus — Wikipedia, the free encyclopedia, 2019. [Online; accessed 21-June-2021].

[25] FIZZ Marketing & Communicatie". Phantom AUV, 2021.

[26] Bashan Zuo, Jiaxin Chen, Larry Wang, and Ying Wang. A reinforcement learning based robotic navigation system. In *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, volume 2014-Janua, 2014.

[27] V. Madhu Babu, U. Vamshi Krishna, and S. K. Shahensha. An autonomous path finding robot using Q-learning. In *Proceedings of the 10th International Conference on Intelligent Systems and Control, ISCO 2016*, 2016.

[28] T. Martinez-Marin. On-line optimal motion planning for nonholonomic mobile robots. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 512–517, 2006.

[29] P. J. Zufiria and R. S. Guttalu. The adjoining cell mapping and its recursive unraveling, part i: Description of adaptive and recursive algorithms. *Nonlinear Dynamics*, 4:207–226, 1993.

[30] B. Bischoff, D. Nguyen-Tuong, I. H. Lee, F. Streichert, and A. Knoll. Hierarchical reinforcement learning for robot navigation. In *ESANN 2013 proceedings, 21st European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2013.

[31] Mohammad Abdel Kareem Jaradat, Mohammad Al-Rousan, and Lara Quadan. Reinforcement based mobile robot navigation in dynamic environment. *Robotics and Computer-Integrated Manufacturing*, 27(1), 2011.

[32] Nathan Sprague and Dana Ballard. Multiple-goal reinforcement learning with modular sarsa(O). In *IJCAI International Joint Conference on Artificial Intelligence*, 2003.

[33] Amirhossein Shantia, Rik Timmers, Yiebo Chong, Cornel Kuiper, Francesco Bidoia, Lambert Schomaker, and Marco Wiering. Two-stage visual navigation by deep neural networks and multi-goal reinforcement learning. *Robotics and Autonomous Systems*, 138:103731, 2021.

[34] Francisco Bonin-Font, Alberto Ortiz, and Gabriel Oliver. Visual navigation for mobile robots: A survey. *Journal of Intelligent and Robotic Systems*, 53:263–296, 11 2008.

[35] R. Sim and G. Dudek. Learning generative models of scene features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–I, 2001.

[36] A. Hilton. Scene modelling from sparse 3d data. *Image and Vision Computing*, 23(10):900–920, 2005.

[37] N. Winters, J. Gaspar, G. Lacey, and J. Santos-Victor. Omni-directional vision for robot navigation. In *Proceedings IEEE Workshop on Omnidirectional Vision (Cat. No.PR00704)*, pages 21–28, 2000.

[38] Martin C. Martin. Evolving visual sonar: Depth from monocular images. *Pattern Recognition Letters*, 27(11):1174–1180, 2006. Evolutionary Computer Vision and Image Understanding.

[39] José Santos-Victor and Giulio Sandini. Visual-based obstacle detection a purposive approach using the normal flow. In *INTELLIGENT AUTONOMOUS SYSTEMS. IOS*. Press, 1995.

[40] Hideo Morita, Michael Hild, Jun Miura, and Yoshiaki Shirai. Panoramic view-based navigation in outdoor environments based on support vector learning. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2302–2307, 2006.

[41] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[42] François Chollet et al. Keras. `https://keras.io`, 2015.

[43] Webots. http://www.cyberbotics.com. Open-source Mobile Robot Simulation Software.

[44] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004.

[45] Wikipedia contributors. Robotics simulator — Wikipedia, the free encyclopedia, 2021. [Online; accessed 25-January-2021].

[46] Sharp. *Sharp GP2Y0A41SK0F Datasheet*, 2011.

[47] S Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22(1-3):159–195, 1996.

[48] Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *In Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287. Morgan Kaufmann, 1999.

[49] Adam Daniel Laud. *Theory and Application of Reward Shaping in Reinforcement Learning*. PhD thesis, University of Illinois, USA, 2004. AAI3130966.

[50] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[51] J. Heaton. *Artificial Intelligence for Humans: Deep learning and neural networks*. Artificial Intelligence for Humans. Heaton Research, Incorporated., 2015.

[52] J. Brownlee. Difference Between a Batch and an Epoch in a Neural Network, 10 2019.

[53] D. Gupta. Fundamentals of Deep Learning – Activation Functions and When to Use Them?, 07 2020.

[54] S. Doshi. Various Optimization Algorithms For Training Neural Network, 08 2020.

[55] J. Brownlee. Understand the Impact of Learning Rate on Neural Network Performance, 09 2020.

[56] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[57] J. Brownlee. How to Avoid Overfitting in Deep Learning Neural Networks, 08 2019.

Appendices

Appendix A

Sensor Values Conversion to Distances

As said in Subsection 3.1.2 the Sharp sensor model attached to the E-puck is able to read values between 4 and 30 centimeters, however, these data are given in voltage and have to be converted. The Webots documentation already has an equation (Equation A.1) that represents this conversion but it could introduce some errors given the deviations it has, as can be seen in Figure A.1.

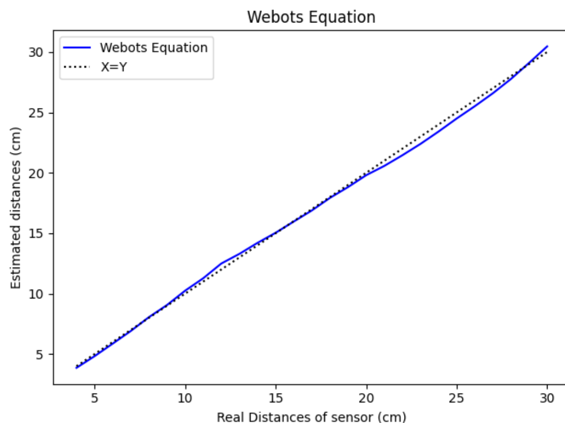


Figure A.1: Webots equations.

In order to compute a better equation, firstly it is necessary to get voltages and the theoretical distance associated with those. To accomplish this, 26 samples have been captured (one per centimeter) to generate the equation and 26 more for validation, giving the relation shown in Figure A.2.

As can be perceived, there is no linear relation between the voltage and the distance and so, Equation A.2 is used to reverse the theoretical distances in order to make the curve look more like a straight line. Here multiple values are considered for the k constant and, in the end, three are used: $k = 0.7$, $k = 1.2$ and $k = 1.4$, resulting in the curves from Figure A.3.

$$distance = \frac{0.1594 * voltage^{(-0.8533)} - 0.02916}{100} \tag{A.1}$$

$$\frac{1}{distance + k} \tag{A.2}$$

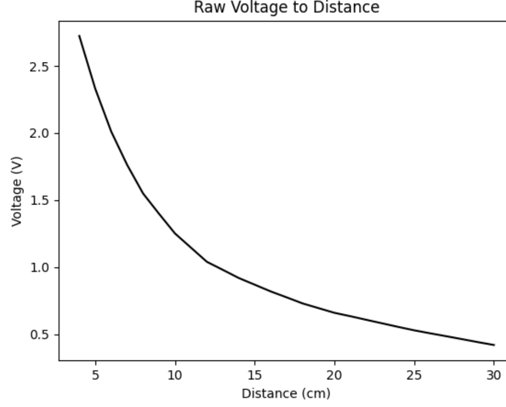


Figure A.2: Raw voltage relation to theoretical distance.

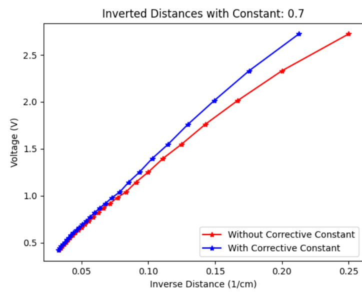
After having a good approximation to a straight line, the least-squares method from polynomial regression is used with each curve and three equations are obtained: Equation A.3 for $k = 0.7$, Equation A.4 for $k = 1.2$ and Equation A.5 for $k = 1.4$. The three are then validated, using the validation data which produces the results from Figure A.4. Looking at these equations, the one in Figure A.4c is the most linear, however, when examined in more carefully, there are some points where the estimation deviates too much from the real value, points in which it does not happen in other equations. Therefore, in order to overcome this problem and minimize the conversion error, the system with three equations from Equation A.6 has been created with some rounding, considering just the range where each equation produces the best estimation. In Figure A.5 the final result, obtained with the system of equations, is compared to the Webots equation, and can be concluded that the system is much less prone to errors.

$$distance = \frac{13.045778715415159}{voltage - 0.0282955300647411125} - 0.7 \tag{A.3}$$

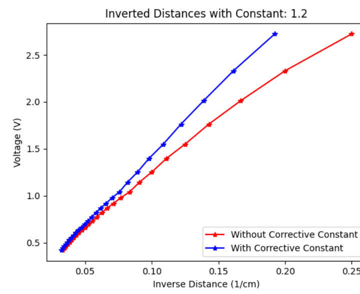
$$distance = \frac{14.483674005107527}{voltage + 0.02681880954262667} - 1.2 \tag{A.4}$$

$$distance = \frac{15.065187821049603}{voltage + 0.04822905725919005} - 1.4 \tag{A.5}$$

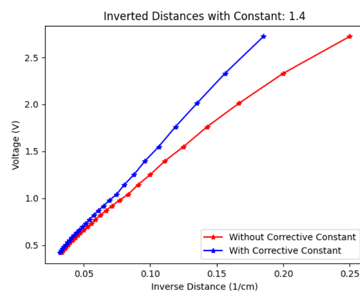
$$\begin{cases} \frac{15.06519}{voltage+0.04823} - 1.4, & \text{if } voltage \leq 0.48 \\ \frac{14.48367}{voltage+0.02682} - 1.2, & \text{if } voltage > 0.48 \land voltage \leq 0.53 \\ \frac{13.04578}{voltage-0.02830} - 0.7, & \text{else} \end{cases} \tag{A.6}$$



(a) k = 0.7.

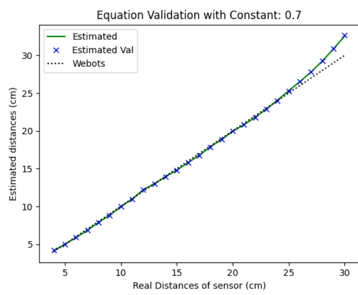


(b) k = 1.2.

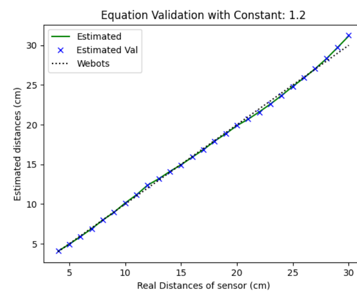


(c) k = 1.4.

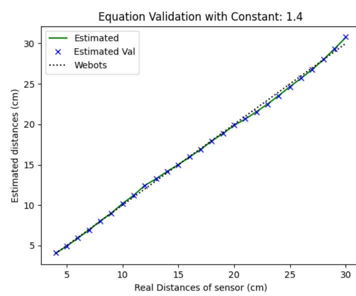
Figure A.3: Curve linearization.



(a) k = 0.7.



(b) k = 1.2.



(c) k = 1.4.

Figure A.4: Equation validation.

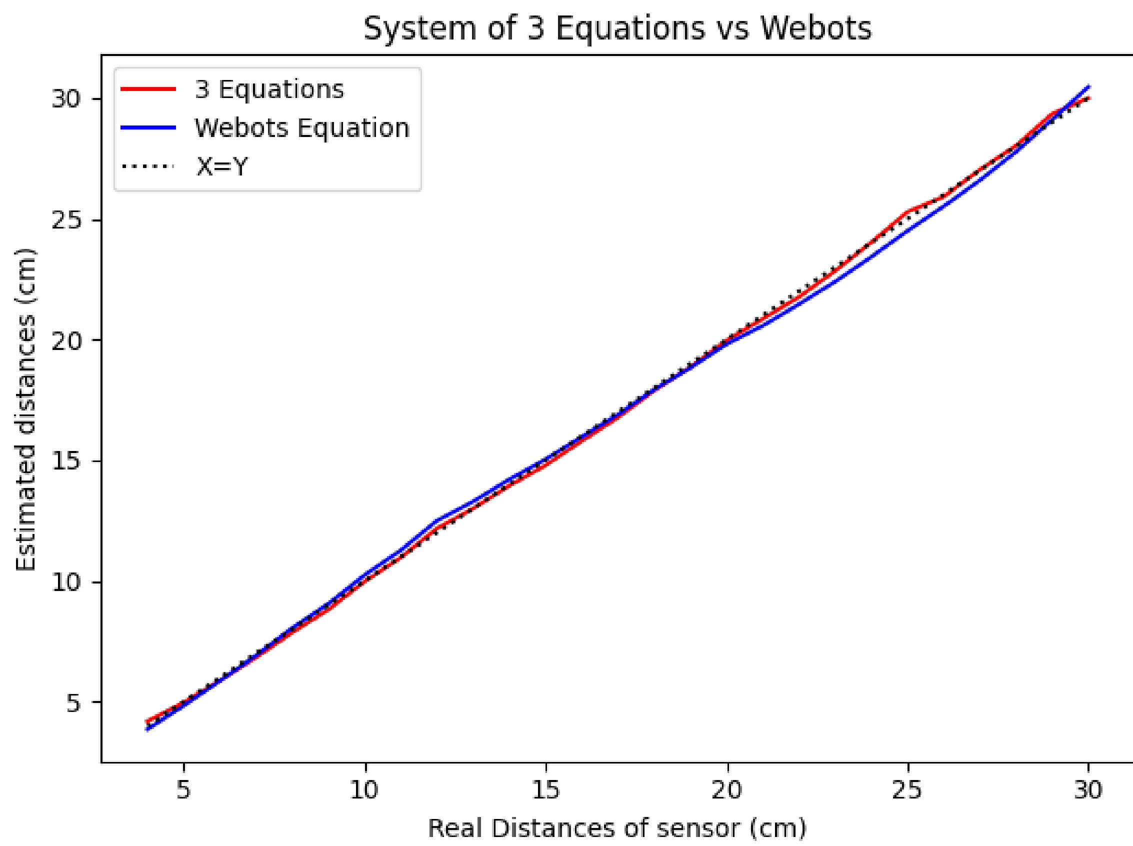


Figure A.5: System of 3 equations vs Webots equations.

Appendix B

Artificial Neural Network for Robot Localization

ANN, or Neural Network, is the component of AI that is meant to simulate the function of a human brain. An ANN is a collection of nodes, called artificial neurons, or only neurons, linked together, where each connection can transmit a signal to other neurons. The one that receives the signal, processes it and can signal more neurons connected to it. Those connections between neurons are the edges, and both neurons and edges typically have weights that are adjusted as the learning proceeds. These weights can increase or decrease the strength of the signals that are sent. Neurons are, usually, aggregated into layers, and each layer can perform different transformations to the signals received in the input. A signal travels from the input layer (first layer) to the output layer (last layer) crossing, sometimes, multiple layers in-between.

In order to determine the current position and orientation of a robot in an arena using only the available sensors, an ANN has been developed. The robot used for this problem has a total of 8 infrared and 5 ultrasonic sensors. Its position and orientation are represented only in a 2d plan, that is, are represented by two coordinates and an angle.

B.1 Simulation Environment and Data Extraction

In order to train an ANN, it is necessary a huge data set with thousands of data to provide all the possible situations to the algorithms and have a good result in the end. To collect these data a simple simulation environment has been developed in the Webots Simulator, being this a 1-meter wide square arena as can be seen in Figure B.1. The data is collected using the Khepera-IV robot, seen in Figure 3.1c and with a controller developed in *Python language*. This controller makes the robot wander randomly across the entire arena in order to make the data set as random as possible and, at the same time, collects data from the 8 infrared sensors and 5 ultrasonic ones and saves it in a file with *CSV* format. The random wander is achieved by using the front and side sensors to keep the robot moving in different directions whenever it is very close to a wall, without ever hitting one.

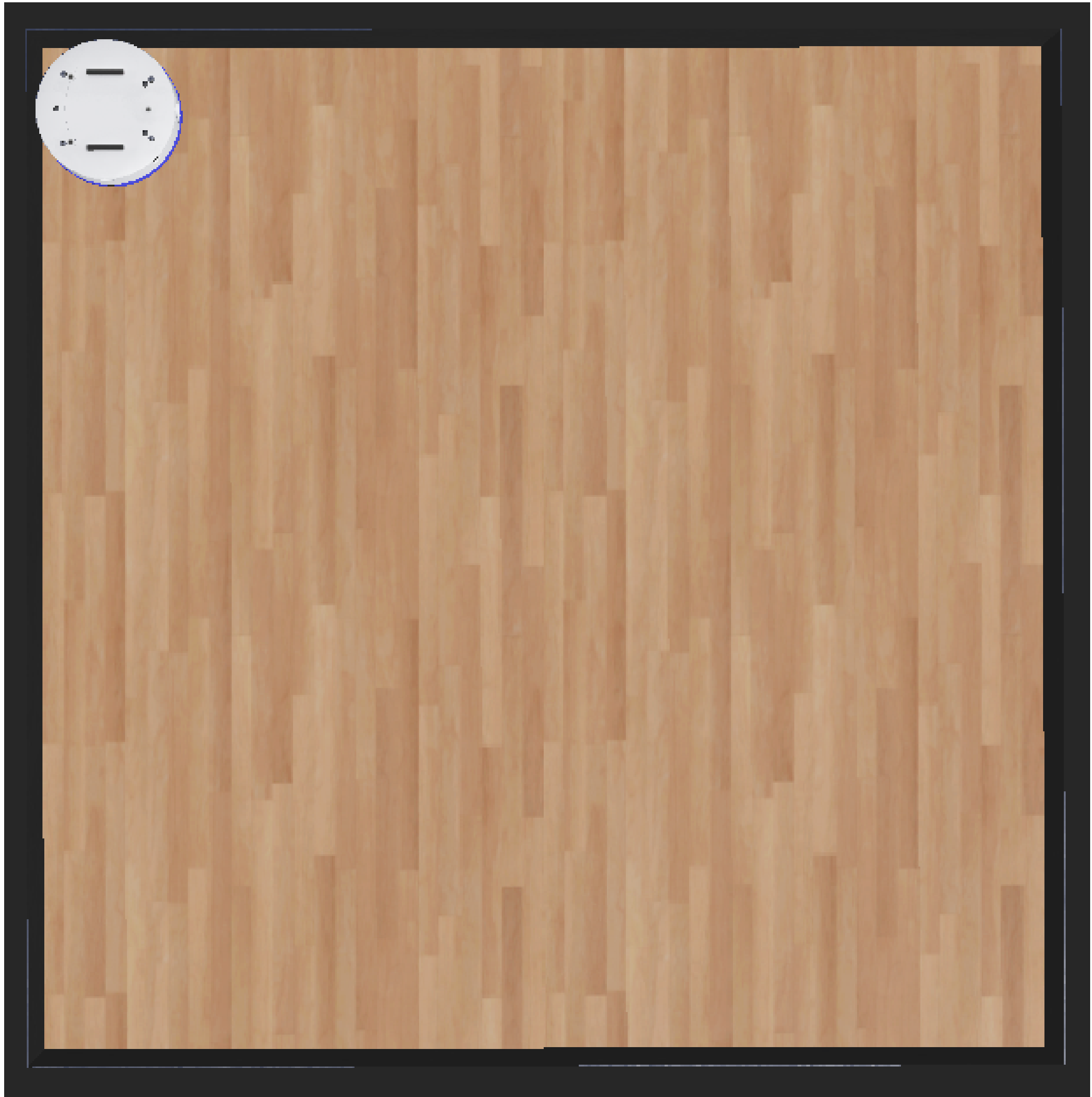


Figure B.1: 1 square meter arena in Webots Simulator where data were collected.

B.1.1 Data Pre-processing

To perform the training of this ANN, firstly were collected about 25 thousand rows of data, but, seeing that it was not enough, more were collected to a total of 59 thousand rows of data. Since the original resolution of each type of sensor used is different, the data are preprocessed using the scikit-learn [50] library, transforming all the data to the same resolution, improving the results in the training and testing.

That same training and testing are done with different subsets of the original data set. To do this, the data are sliced into 3 subsets, for training, validation and testing, again using the scikit-learn library. Initially, the original data set is sliced into two, one with 80% of the data and another one with 20%, being this last one corresponding to the testing data set, with a total close to 11 thousand rows of data. The remaining 80% is then sliced into two more subsets with, again, 80% and 20%, giving rise to the training data set, with more than 37 thousand rows, and the validation one, with more than 9 thousand rows, corresponding this two to 64% and 16% of the original data set, respectively.

B.2 Model

The model for this ANN has been developed using the Keras [42] library. As said before, an ANN is a collection of nodes organized in layers. The input and output layers size must match the number of features and the number of results that those features produce, respectively. In this case, the number of features used changed when constructing the network and so the input layer has a size of both 10 or 13 nodes, being this equal to the number of sensors in the robot. When using 10 input nodes, the rear sensors of the robot are discarded, and with 13 all the sensors, except the ground ones, are taken into account, however, for this particular problem, this specification does not affect the performance, and using either 10 or 13 sensors produces good results whereby, all the features are used, being the final input layer constituted by 13 nodes. On the other hand, the output layer has only 3 nodes, being these the position and orientation of the robot. Since the output layer is not expected to be a classification between classes, but the values themselves of the robot position and orientation, this ANN is called a regression neural network.

The real challenge is to determine the number of hidden layers and how many nodes each one should have, being the hidden layers the ones between the input and output layers. In the beginning, a small configuration with just one layer and with the number of hidden neurons equal to 2/3 the size of the input layer, plus the size of the output layer [51] was used. However when training the results were not satisfactory at all, and so, the number of hidden layers and nodes has been increased reaching a final model configuration of 8 hidden layers with 250, 220, 200, 180, 160, 140, 120 and 100 nodes, which is a considerable size already and considerably complex, resulting in more interesting results.

From the beginning of the model creation to the end, the implementation suffered many changes in the hyperparameters such as the number of epochs and batch_size, as well as the activation function, optimizer and learning rate. The number of epochs defines the number of times that the learning algorithm goes through the entire data set, the batch_size defines how many samples are used before updating the internal parameters, such as the weights, being a sample equal to one row of data [52]. The activation function is a function that is applied to the output of a layer that serves as the input of another layer, and provides the non-linearity that distinguishes an ANN from a simple logistic regression model [53]. The optimizer is the

method that ties together the loss function and model parameters and updates the model in response to the output of that loss function, that is, the optimizer molds the model into its most accurate possible form [54]. Finally, the learning rate defines how quickly the model adapts to the problem, but it should noted that, adapting too quickly, that is, having a higher learning rate, might make the model converge to a suboptimal solution [55].

Initially learning rate was static however, Keras library allows to configure adaptive learning rates, which are functions that change the learning rate as the model evolves. This way, the *ReduceLROnPlateau* function is used, which is a technique to alter this hyperparameter when the model reaches a plateau, that is, if the model loss does not change in a defined number of epochs then it means that it reached its best configuration or a plateau [55]. Decreasing the learning rate, guarantees that the model keeps learning, at a slower pace than before, and surpasses that plateau. If the loss does not change even though the learning rate has decreased then it means that the model reached its optimal solution.

As mentioned earlier, the ANN model has been developed using the Keras library, but this has many different configurations for the hidden layers and respective nodes. In this model, only Core layers are used, more specifically, Dense layers. Those are fully connected layers, which means that each node from one layer connects to all the nodes from the next layer. For the model developed, the kernel_initializer, which is the parameter that defines the initial weights of each connection between layers, is set to *he_uniform*, that is, the weights matrix is initialized with a uniform distribution in-between two limits defined by the number of input units in the weight tensor. The activation function is defined specifically for each layer, being the Rectified Linear Unit (ReLU) function the one chosen, as it is the most used in regression neural networks. After the construction of the network, the optimizer and loss function must defined in order for the model to be compiled. The chosen optimizer is the Adam optimization, which is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments and that realizes the benefits of both Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp), being effective and fast as presented in Figure B.2 [56]. The loss function used in the model is the Mean Squared Error (MSE), which is a regression metric that computes the average of the squares of the errors, that is, the average squared difference between the estimated values (the ones predicted by the ANN) and the actual values (the ones from the data set).

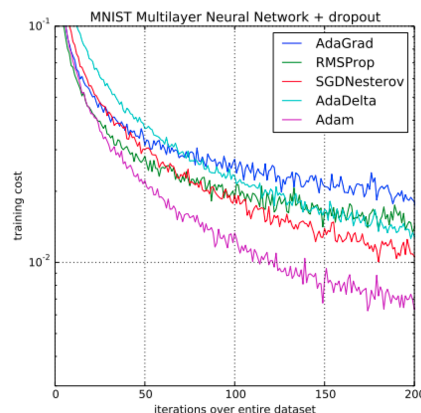
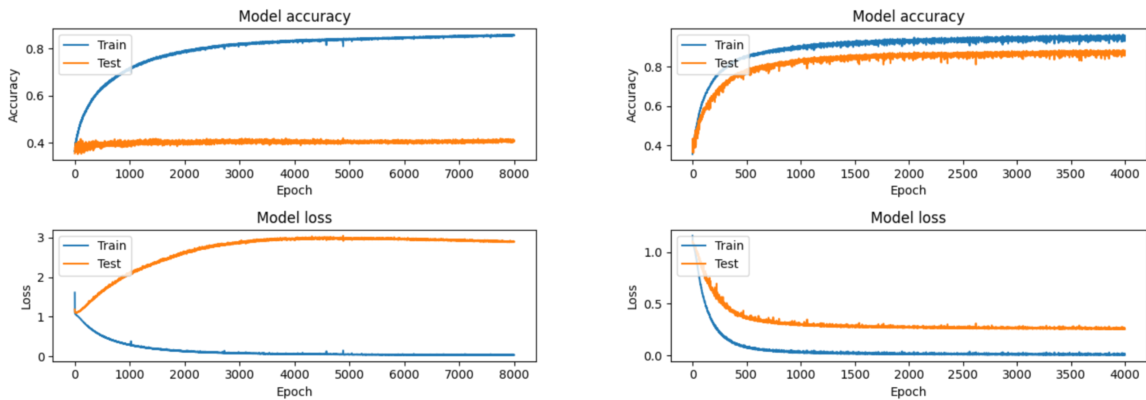


Figure B.2: Neural networks using dropout stochastic regularization.

B.3 Results

As said in Section B.2, with the initial network the results were not satisfactory at all even with a huge number of epochs. In Figure B.3a, it can be seen that, even though training accuracy and loss were reasonable, the test accuracy and loss were very low and high respectively, which is the opposite of what is intended. This could mean that an overfitting error has happened, which occurs when the model is too closely fit to a limited set of data. To resolve this there are many solutions, for example, using fewer features or training more data [57]. In this case more data were collected ending up with more than double the data than previously. With this, the model was trained again and the overfitting error has been corrected, achieving results more interesting, with both training and test accuracy over 80%, as well as, a loss value close to zero, which can be seen in Figure B.3b. In order to increase the learning performance, the same data set is tested in the same model but with an adaptive learning rate, allowing it to adapt if the model reaches a plateau during the training. With this modification there is an increase in learning performance which makes possible to lower the number of epochs and still achieve better results, as demonstrated in Figure B.4, where the training is done with just 1500 epochs, with the same model of the Figure B.3b, but with an adaptive learning rate. Comparing both figures the model accuracy increased faster with the adaptive learning rate and achieved higher results than the model with a static learning. This is justified because, using the static learning rate, the model is learning at a very low rhythm from the beginning to the end of the training, while with the adaptive one, the learning rate starts higher and goes down when a plateau is reached, until it hits a minimum value of 0.00001, allowing the network to learning at a faster pace in the begin and slowly decreasing the pace during the training originating better results.



(a) Model Accuracy and Loss. Model configuration: 6 hidden layers with 100 nodes each, lr 0.0001.

(b) Model Accuracy and Loss. Model configuration: 8 hidden layers with 250, 220, 200, 180, 160, 140, 120 and 100 nodes, lr 0.001.

Figure B.3: Analysis of two implemented models.

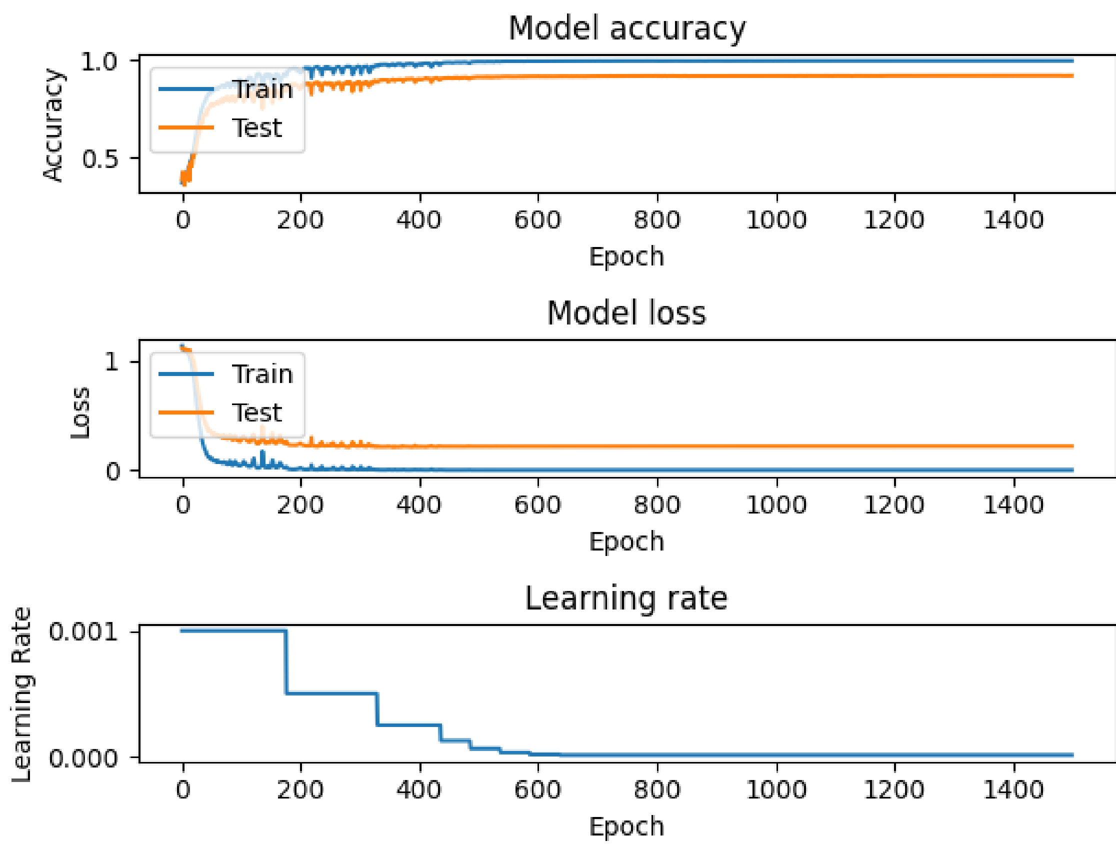


Figure B.4: Accuracy and Loss with final model configuration, 1500 epochs and adaptive learning rate.