UNIVERSIDADE D
COIMBRA

Gabriel de Jesus Simões Gonçalves

DeepRL-Based Dynamic Motion Planning for Robot Navigation in Unknown Indoor Environments

Dissertation supervised by Professor Doctor Urbano Nunes and Doctor Luís Garrote, and submitted to the Electrical and Computer Engineering Department of the Faculty of Science and Technology of the University of Coimbra, in partial fulfillment of the requirements for the Master's Degree in Electrical and Computer Engineering, specialization in Robotics, Control and Artificial Intelligence.

September of 2023

1 2  9 0

UNIVERSIDADE D
COIMBRA

DeepRL-Based Dynamic Motion Planning for Robot Navigation in Unknown Indoor Environments

Gabriel de Jesus Simões Gonçalves

Coimbra, September of 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DeepRL-Based Dynamic Motion Planning for Robot Navigation in Unknown Indoor Environments

Dissertation supervised by Professor Doctor Urbano Nunes and Doctor Luís Garrote, and submitted to the Electrical and Computer Engineering Department of the Faculty of Science and Technology of the University of Coimbra, in partial fulfillment of the requirements for the Master's Degree in Electrical and Computer Engineering, specialization in Robotics, Control and Artificial Intelligence.

Supervisor:
Prof. Dr. Urbano José Carreira Nunes

Co-Supervisor:
Dr. Luís Carlos Artur da Silva Garrote

Jury:
Prof. Dr. Rui Alexandre de Matos Araújo
Prof. Dr. Luís Alberto da Silva Cruz
Prof. Dr. Urbano José Carreira Nunes

Coimbra, September of 2023

Acknowledgments

I wish to express my profound gratitude to all those who have contributed to the completion of this dissertation. This journey would not have been possible without the support, guidance, and encouragement from numerous individuals and institutions.

First and foremost, my deepest appreciation to my thesis supervisor, Prof. Dr. Urbano Nunes, for granting me the opportunity to tackle this challenging dissertation topic and for providing all the necessary resources to achieve the defined objectives. I extend my sincere appreciation to Dr. Luis Garrote for his invaluable contributions of knowledge and unwavering assistance throughout the course of this work. I am truly grateful for their patience, insightful feedback, and continuous guidance.

A heartfelt thanks to my family for their unwavering support and encouragement throughout my academic journey. Their love, understanding, and sacrifices have been a constant source of motivation, and I am forever thankful for their presence in my life.

My sincere thanks to my friends and colleagues who provided invaluable moral support, engaging discussions, and a sense of camaraderie during the course of this dissertation. A special thanks go to Ulisses Reverendo, Gonçalo Arsénio, and Gabriela Simões. Your encouragement and friendship have made this journey more enjoyable.

A particular acknowledgment to my colleagues from the Laboratory for Automatics and Systems at Instituto Pedro Nunes, for their invaluable support. A particular mention to Guilherme Correia and João Quintas, who consistently demonstrated their support, interest, and concern for this project.

I extend my gratitude to the Institute of Systems and Robotics, and to the Department of Electrical and Computer Engineering of the University of Coimbra for providing a conducive academic environment and resources that were indispensable for this research.

Last but not least, a special thanks to Cláudia Lima for the invaluable moral support and encouragement throughout the course of this work.

This work has been supported by "GreenAuto", with reference C644867037, Portugal, and by ISR-UC FCT through grant UIDB/00048/2020.

Abstract

Mobile robots are progressively taking over tasks in various industries, from industrial operations to space exploration, enhancing efficiency, safety, and productivity while also expanding the possibilities for automation and exploration in challenging environments. Difficulties in performing tasks such as navigation, target recognition and obstacle avoidance must be overcome.

Motion planning is a crucial component within mobile robot navigation that establishes a route from an initial to a target point. However, in unknown domains, this task becomes significantly more challenging. In the absence of a global map of the environment, a local navigation strategy must be exploited. In local motion planning, short-term paths are devised based on real-time sensory observations of the surrounding environment.

This dissertation proposes a Deep Reinforcement Learning (DeepRL)-based approach to solve robot local motion planning in environments populated by both static and dynamic obstacles. It leverages the Double Dueling Deep Q-Network (D3QN) and a costmap representation of the robot's surrounding environment paired with distances and orientation measurements to define the Reinforcement Learning (RL) state model. In conventional DeepRL approaches, experiences used to train the RL agent are usually sampled uniformly. In this work, the prioritized experience replay technique is implemented to enhance the learning efficiency by giving priority to training samples with higher impact. Reward propagation was also implemented to address the delayed rewards' problem common in RL, by assigning responsibility for a specific outcome to the various actions that contributed to it.

The introduced motion planning algorithm comprises two separate stages: training and testing. During training, the agent learns via trial-and-error which actions lead to a collision-free movement towards the target. The testing phase assesses the agent's decision-making strategy in an online stage. To enhance the training stage, facilitating convergence and improving long-term generalization, curriculum learning techniques were integrated.

Evaluation and validation took place within Gazebo simulation environments using the turtlebot virtual robot. The presented results showcase the developed framework's effectiveness in both static and dynamic environments, highlighting the benefits of the proposed techniques.

Keywords: Mobile Robot Navigation, Local Motion Planning, Deep Reinforcement Learning, Curriculum Learning, Dynamic Environments

Resumo

Os robôs móveis estão a assumir cada vez mais tarefas em diversas áreas, desde aplicações industriais até exploração espacial, aumentando a eficiência, segurança e produtividade, e expandindo as possibilidades de automação e exploração em ambientes complexos. Dificuldades na execução de tarefas como navegação, reconhecimento de alvos e desvio de obstáculos devem ser superadas. O planeamento de movimento é um componente crucial na navegação de robôs móveis que estabelece uma rota de um ponto inicial para um ponto destino. No entanto, em domínios desconhecidos, esta tarefa torna-se significativamente mais complicada. Na ausência de um mapa global do ambiente, uma estratégia de navegação local deve ser explorada. No planeamento de movimento local, trajetos de curto prazo são elaboradas com base em observações sensoriais do ambiente envolvente em tempo real.

Esta dissertação propõe uma abordagem baseada em DeepRL para resolver o planeamento de movimento local de robôs em ambientes com obstáculos estáticos e dinâmicos. Beneficia da utilização de uma Double Dueling Deep Q-Network (D3QN), e de uma representação de mapa de custos do ambiente circundante do robô, em conjunto com medidas de distância e orientação para definir o modelo de estado de RL. Nas abordagens convencionais de DeepRL, as experiências usadas para treinar o agente de RL são amostradas de forma uniforme. Neste trabalho, a técnica de repetição de experiência priorizada é implementada para melhorar a eficiência da aprendizagem, dando prioridade às amostras de treino com maior impacto. A propagação de recompensas também foi implementada para lidar com o problema de recompensas atrasadas comuns em RL, atribuindo responsabilidade por um resultado específico às várias ações que contribuíram para o mesmo.

O algoritmo de planeamento de movimento introduzido envolve duas etapas distintas: treino e teste. Durante o treino, o agente aprende, por tentativa e erro, as ações que levam a um movimento sem colisões em direção ao alvo. A fase de teste avalia a estratégia de tomada de decisão do agente numa etapa 'online'. Para aprimorar a etapa de treino, facilitando a convergência e melhorando a generalização a longo prazo, foram integradas técnicas de aprendizado por currículo. A avaliação e validação foram efetuadas em ambientes de simulação Gazebo utilizando o robô virtual turtlebot. Os resultados apresentados destacam a eficácia da estrutura desenvolvida em ambientes estáticos e dinâmicos, bem como as vantagens das técnicas propostas.

Palavras-chave: Navegação de Robôs Móveis, Planeamento de Movimento Local, Aprendizagem Profunda por Reforço, Aprendizagem por Currículo, Ambientes Dinâmicos

"Learn as if you will live forever, live like you will die tomorrow."
Mahatma Gandhi

Contents

Acknowledgments **i**

Abstract **iii**

Resumo **v**

List of Acronyms **xiii**

List of Figures **xvi**

List of Tables **xviii**

1 Introduction **1**

1.1 Context and Motivation 1

1.2 Proposed Framework 2

1.3 Objectives and Key Contributions 3

2 Background Material **4**

2.1 Motion Planning 4

2.1.1 Motion vs. Path Planning 4

2.1.2 Global vs. Local Motion Planning 4

2.1.3 Static vs. Dynamic Motion Planning 4

2.2 Reinforcement Learning 5

2.2.1 Policy Evaluation 6

2.2.2 Exploration vs. Exploitation 7

2.2.3 Q-Learning 7

2.3 Deep Learning 8

2.3.1 Artificial Neural Networks 8

2.3.2 Neural Network Training 9

2.3.3 Neural Networks Architectures 10

2.3.3.1 Convolutional Neural Networks 10

2.3.4 Transfer Learning 11

2.4 Deep Reinforcement Learning 12

2.4.1 Value-based vs. Policy-based Learning 12

2.4.2 Deep Q-Learning . . . 12
2.4.3 Dueling Deep Q-Network . . . 14
2.4.4 Double Deep Q-Network . . . 15
2.4.5 Dueling Double Deep Q-Network . . . 15
2.4.6 Curriculum Learning . . . 16

3 State of the Art **18**
3.1 Global Motion Planning . . . 18
3.1.1 Graph-Search algorithms . . . 18
3.1.2 Sampling-based algorithms . . . 19
3.2 Local Motion Planning . . . 19
3.2.1 Reaction-based algorithms . . . 19
3.2.2 Reinforcement Learning algorithms . . . 19
3.2.2.1 Policy Gradient RL algorithms . . . 19
3.2.2.2 Optimal value RL algorithms . . . 20

4 Developed Work **22**
4.1 Proposed DeepRL-based Framework . . . 22
4.2 Deep Neural Network . . . 25
4.3 Replay Buffer . . . 27
4.4 State, Action and Reward Models . . . 27
4.4.1 State Model . . . 27
4.4.2 Action Model . . . 28
4.4.3 Reward Model . . . 29
4.5 Simulation Environments . . . 30
4.5.1 Dynamic environment . . . 30
4.5.2 Curriculum Environments . . . 31

5 Software Tools and Hardware Materials **33**
5.1 Ubuntu Operating System . . . 33
5.2 Robot Operating System . . . 33
5.2.1 RViz . . . 35
5.3 Gazebo . . . 36
5.4 Turtlebot3 . . . 36
5.5 Hardware Components . . . 37
5.5.1 Graphical Processing Unit . . . 38
5.5.2 Central Processing Unit and Random-Access Memory . . . 39
5.6 Python and VSCode . . . 39
5.6.1 Pytorch . . . 39

6 Results and Discussion **42**
6.1 Motion Planning in Environments with Static Obstacles . . . 42
6.1.1 Scenario 1 . . . 42
6.1.1.1 Prioritized Experience Replay . . . 44

6.1.1.2 Reward Propagation 44
6.1.1.3 Deep Q-Learning Variants 45
6.1.2 Scenario 2 47
6.1.3 Curriculum Learning 48
6.2 Motion Planning in Environments with Dynamic Obstacles 51

7 Conclusion **55**
7.1 Future Work 56

Bibliography **57**

List of Acronyms

A3C Asynchronous Advantage Actor Critic

ADAM Adaptative Moment Estimation

ANN Artificial Neural Network

CNN Convolutional Neural Network

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

cuDNN CUDA Deep Neural Network

D2QN Double Deep Q-Network

D3QN Double Dueling Deep Q-Network

DDPG Deep Deterministic Policy Gradient

DDQN Dueling Deep Q-Network

DL Deep Learning

DNN Deep Neural Network

DP Dynamic Programming

DQN Deep Q-Network

DWA Dynamic Window Approach

DeepRL Deep Reinforcement Learning

GPU Graphical Processing Unit

LiDAR Light Detection And Ranging

MC Monte Carlo

ML Machine Learning

MSE Mean Squared Error

NN Neural Network

PER Prioritized Replay Buffer

PFM Potential Field Method

PPO Proximal Policy Optimization

PRM Probabilistic Roadmap

RL Reinforcement Learning

RNN Recurrent Neural Network

ROS Robot Operating System

RRT Rapidly-exploring Random Tree

SGD Stochastic Gradient Descent

TD Temporal Difference

TL Transfer Learning

List of Figures

1.1 Proposed DeepRL-based local navigation framework taken from [1]. 3

2.1 Reinforcement Learning framework. 5
2.2 Q-Learning pipeline. 7
2.3 Artificial Neural Network architecture. 9
2.4 Overview of Convolutional Neural Networks architecture for image classification. 11
2.5 Deep Reinforcement Learning framework. 12
2.6 Deep Q-learning training stage overview. 13
2.7 Dueling Deep Q-Network architecture. 15
2.8 Curriculum learning environments. 16

4.1 Training stage pipeline. 23
4.2 Original Dueling Deep Q-Network architecture [1]. 25
4.3 Proposed Dueling Deep Q-Network architecture. 26
4.4 Costmap representation and C_{stack} structure. 28
4.5 Robot-target-relative data representation. 29
4.6 Gazebo's *Stage 4* environments. 31
4.7 Gazebo's Curriculum environments. 32

5.1 Publisher-subscriber Robot Operating System (ROS) Protocol. 34
5.2 ROS computation graph. 35
5.3 Gazebo environment and respective RViz costmap representation. 36
5.4 Gazebo's *Stage 4* environment. 37
5.5 TurtleBot3 Burger and respective Gazebo's representation. 37

6.1 Gazebo's *Stage 4* environment Scenario 1. The agent's initial position is located at the turtlebot's location, and the target point is represented by the red circle. . 43
6.2 Training results in Gazebo's *Stage 4* Scenario 1. (a) shows the distances between the robot and the target and the end of each episode. In (b) is illustrated the episodic rewards, and in (c) is illustrated the paths taken by the robot, with the reward on each step represented by color. 43
6.3 Training results in Gazebo's *Stage 4* Scenario 1 with Prioritized Replay Buffer (PER). 44
6.4 Training results in Gazebo's *Stage 4* Scenario 1 with PER and reward propagation. 45

6.5 Training results in Gazebo's *Stage 4* Scenario 1 with Deep Q-Network (DQN). . . 45
6.6 Training results in Gazebo's *Stage 4* Scenario 1 with Dueling Deep Q-Network (DDQN). 46
6.7 Training results in Gazebo's *Stage 4* Scenario 1 with Double Deep Q-Network (D2QN). 46
6.8 Gazebo's *Stage 4* environment Scenario 2. The agent's initial position is located at the turtlebot's location, and the target points are represented by the red circles. 47
6.9 Training results in Gazebo's *Stage 4* Scenario 2. 48
6.10 Gazebo's curriculum environments. 49
6.11 Results of training with knowledge transfer in the curriculum environments (see Fig. 6.10). 50
6.12 Results of training from scratch in the curriculum environments (see Fig. 6.10). . 51
6.13 Gazebo's *Stage 4 Dynamic* environment. 52
6.14 Training results in Gazebo's *Stage 4 Dynamic* Scenario 2 with knowledge transfer. 52
6.15 Training results in Gazebo's *Stage 4 Dynamic* Scenario 2 without knowledge transfer (training from scratch). 53

List of Tables

2.1 Reinforcement Learning parameters. 6

3.1 Summary of DeepRL-based optimal value motion planning strategies. 21

4.1 Layer configurations of the proposed Deep Neural Network (DNN). 26
4.2 Action Set. 29

5.1 ROS framework main components. 34
5.2 ROS topics. 35
5.3 TurtleBot3 Burger hardware specifications [2]. 37
5.4 360 Laser Distance Sensor LDS-01 specifications [3]. 38
5.5 Graphical Processing Units (GPUs) specifications. 38
5.6 Central Processing Units (CPUs) specifications. 39
5.7 Python libraries. 39

6.1 Simulation parameters utilized in Scenario 1. 42
6.2 Training and testing details in Scenario 1. 43
6.3 Training and testing details in Scenario 1 with PER. 44
6.4 Training and testing details in Scenario 1 with PER and reward propagation. . . 45
6.5 Training and testing details in Scenario 1 with DQN, DDQN, D2QN, and D3QN. 46
6.6 Simulation parameters utilized in Scenario 2. 47
6.7 Training details in Gazebo's *Stage 4* Scenario 2. 48
6.8 Testing details in Gazebo's *Stage 4* Scenario 2. 48
6.9 Simulation parameters utilized in curriculum environments. 49
6.10 Training and testing details with knowledge transfer. 50
6.11 Training and testing details, without knowledge transfer (training from scratch). 51
6.12 Training details in Gazebo's *Stage 4 Dynamic* Scenario 2 with knowledge transfer. 52
6.13 Testing details in Gazebo's *Stage 4 Dynamic* Scenario 2 with knowledge transfer. 52
6.14 Training details in Gazebo's *Stage 4 Dynamic* Scenario 2 without knowledge transfer. 53
6.15 Testing details in Gazebo's *Stage 4 Dynamic* Scenario 2 without knowledge transfer. 53

1

Introduction

This chapter introduces the concepts explored in the proposed work, including the project's motivation, predefined objectives, and key contributions.

1.1 Context and Motivation

Mobile robots have the potential to revolutionize various industries, including healthcare, logistics, and manufacturing. These robots can perform repetitive and physically demanding tasks, allowing humans to focus on more creative and strategic work. Additionally, they can improve workplace safety by taking on dangerous tasks, such as handling hazardous materials or working in high-risk areas [4]. However, to integrate mobile robots into human-populated domains, they must be able to navigate and interact with their environment in a safe, reliable and efficient manner. This requires advanced perception, planning, and decision-making capabilities, as well as the ability to understand and respond to human behavior [5]. Recent advancements in Machine Learning (ML) and computer vision have enabled robots to better understand and model human behavior, allowing them to operate in complex and dynamic environments [6]. Robots now possess the capability to utilize sensors for detecting human presence and movement, analyzing facial expressions and body language, and even comprehending speech and natural language.

For a robot to operate and navigate effectively in complex environments, it must be able to know how to travel from point A to point B while avoiding obstacles in its way. This is denominated as path or motion planning, and it is an essential component of the robot navigation process, the other being localization and mapping [7]. Path planning algorithms determine the best path for a robot to follow from its current location to its intended destination while avoiding obstacles and minimizing costs such as distance, time, or energy consumption. ML-based approaches have shown great potential in path planning for indoor mobile robots. RL is one such approach that has been used to learn optimal policies for navigating in complex environments [8, 9, 10, 11]. RL agents learn, via trial-and-error, to take actions that maximize a reward signal, in order to attain a predetermined objective such as reaching the destination while avoiding obstacles. However, RL algorithms can face limitations in terms of sample complexity, memory requirements, and computational complexity. One way to overcome these limitations is by using Deep Learning (DL) techniques. DL [12, 13, 14] involves training DNNs, multi-layered configurations inspired by biological neural networks, allowing them to learn more complex and

abstract representations of data. DNNs have been used to build maps from sensory data and to estimate the robot's position and orientation accurately [15]. These perceptual capabilities are vital for effective motion planning in complex environments. The merge of RL and DL has resulted in a contemporary machine learning division, denominated Deep Reinforcement Learning (DeepRL). DeepRL has allowed the learning of complex behaviors and control policies in real-world environments, thus enabling the development of methods capable of facing previously intractable problems [16]. By using DeepRL for motion planning in indoor robot navigation, the agent can assess environmental *stimuli* in real-time and make decisions that are more appropriate to the situation at hand. This can help to improve the overall efficiency and effectiveness of indoor navigation in dynamic unknown environments, in which traditional approaches tend to be less effective since they typically involve using an obstacle map to plan a route through a known environment.

1.2 Proposed Framework

The main purpose of this dissertation was to design and implement a motion planning algorithm for robot navigation in unknown indoor environments populated by static and dynamic obstacles using DeepRL techniques. Inspired by the work presented in [1], the proposed method, based on DQN frameworks [17], is specifically designed to operate solely based on real-time sensory data. To mitigate the influence of the sensors' intrinsic properties, a costmap representation of the surrounding environment is employed. The framework diagram, presented in Fig. 1.1, was initially introduced in [1] for robotic motion planning in environments populated by static obstacles. In this work, specific components of this framework, particularly the reward model and the neural network, were meticulously refined and enhanced to empower the RL-agent to execute efficient motion planning not only within static environments but also within dynamic ones. The framework consists of two stages: training and testing (online).

During the training stage, sensory data in the form of laser scans retrieved from a Light Detection And Ranging (LiDAR) sensor is used to build a costmap at each time step t. The DeepRL agent uses a stack of 4 sequential costmaps C_{Stack} along with a stack of 4 sequential affordance-based measurements P_{Stack} related to the robot's current pose and the target location to formulate an environment state s_t. This state is then fed into a two-stream DNN that maps it into an action a_t which is taken by the robot, triggering an environment change. In the subsequent time step, $t+1$, a new state s_{t+1} is computed, and a reward value is assigned to the state transition $s_t \rightarrow s_{t+1}$ resultant from the robot taking action a_t. The DNN is then fed with transition tuples (s_t, a_t, s_{t+1}, r_t) of past experiences to tune its parameters in order to learn, over time, to accurately determine the viability of executing the action a_t on similar s_t states. The goal of the training phase is to obtain a model that maximizes the total episodic rewards, compelling collision-free motion of the robot towards the target.

In the testing or online stage, the fine-tuned models derived from the training phase are employed to guide the agent's decision-making process. The ultimate goal is for the agent to autonomously navigate from a source to a target destination while avoiding any collision with obstacles.

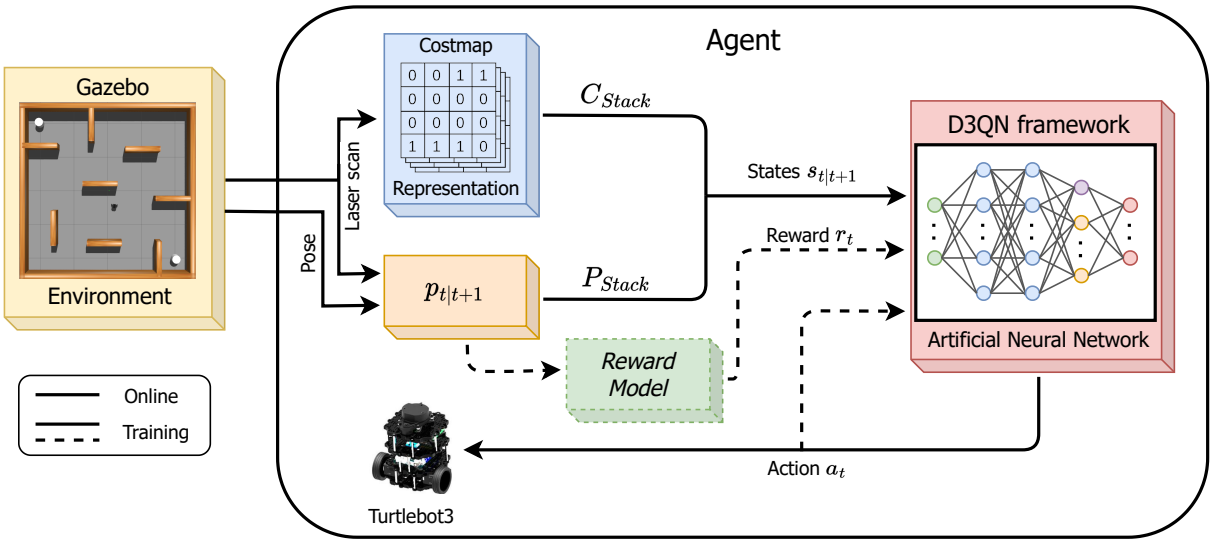


Figure 1.1: Proposed DeepRL-based local navigation framework taken from [1].

1.3 Objectives and Key Contributions

The development of the proposed DeepRL-based local motion planning algorithm involved the accomplishment of several objectives, executed chronologically as follows:

1. Adaptation of the framework 1.1 introduced in [1] to effectively address dynamic environments. This included:
   - Designing an enhanced reward model and action set.
   - Designing a suitable Deep Neural Network architecture.
2. Implementation of the PER and reward propagation techniques.
3. Deployment of curriculum and dynamic environments.
4. Validation of the DeepRL-based local navigation method within static environments.
5. Evaluation of the proposed PER, reward propagation and curriculum learning techniques.
6. Validation of the DeepRL-based local navigation method within dynamic environments.

The main contributions of the project are addressed in the following chapters:

- **Developed Work (Chapter 4)**: Introduces and explains the design of each component within the developed DeepRL framework, along with the simulation environments specifically created to validate the framework.
- **Software and Hardware Materials (Chapter 5)**: Describes the software and hardware resources utilized to accomplish the defined objectives.
- **Results and Discussion (Chapter 6)**: Presents the results obtained from the validation of the proposed framework across both static and dynamic simulation environments, while also providing an evaluation of the proposed techniques: PER, reward propagation and curriculum learning.

2

Background Material

This chapter covers the fundamental concepts of Robot Motion Planning, Reinforcement Learning, Deep Learning, and Deep Reinforcement Learning.

2.1 Motion Planning

Robot motion planning or path planning is, in a simplified point of view, one of the three stages of robot navigation, the other two being localization and mapping.

2.1.1 Motion vs. Path Planning

Path planning aims at finding an optimal path between the origin and destination by strategies like shortest distance or shortest time. Motion planning, on the other hand, aims at handling local short-term decisions, therefore having to consider kinetics features, velocities and poses of the robot and the dynamic objects nearby them when they move towards the goal. Therefore, in addition to achieving long-term optimal or suboptimal planning goals as path planning, motion planning must consider short-term optimal or suboptimal reactive strategies to make instant or reactive response [18].

2.1.2 Global vs. Local Motion Planning

Motion planning can be categorized as global or local, depending on whether all the information about the environment is known or not [19]. Global path planning operates in well known environments, while local path planning operates in unknown environments, generating short-term paths based on local representations built from sensory data in real-time.

2.1.3 Static vs. Dynamic Motion Planning

Motion Planning is also often categorized as static and dynamic. Static motion planning assumes an unchanging environment. Dynamic motion planning, on the other hand, considers the presence of moving obstacles and adapts the robot's motion in real-time to handle any change in the environment. Typically, dynamic planning involves online methods, while static planning can be successfully accomplished using offline techniques [20].

2.2 Reinforcement Learning

Reinforcement Learning (RL) [21, 22] is a subfield of ML that is concerned with the design and development of algorithms and models that allow an agent to learn and make decisions based on feedback received from the environment. Table 2.1 describes the parameters of conventional RL techniques. The agent's objective is to learn, by trial-and-error, a policy that maximizes the expected cumulative reward.

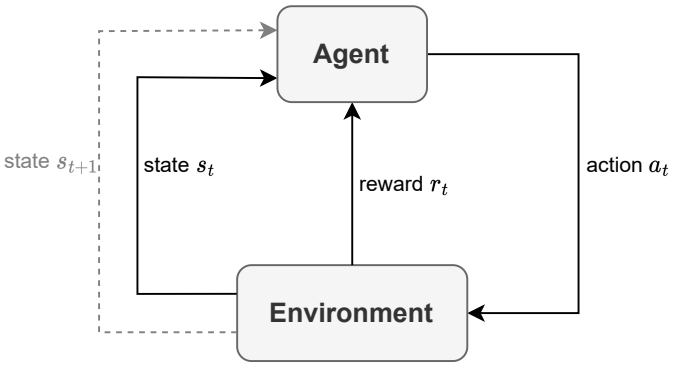


Figure 2.1: Reinforcement Learning framework.

As illustrated in Fig. 2.1, RL is composed of five main components: the agent, the environment, the states, the actions, and the rewards. At each time step t, the agent interacts with the environment and takes an action a_t based on its current state s_t. The environment responds, in the subsequent step $t+1$, with a reward signal r_t, which the agent uses to update its policy $\pi(s_t)$. This process continues until a terminal state condition is met, or a step limit T is reached, at which point, the environment resets, initiating a new episode. The return of each episode e is calculated as the cumulative sum of step rewards, discounted by a factor γ:

$$R_e = \sum_{i=0}^{T} \gamma^i \cdot r_{t+i}, \quad \gamma \in [0,1] \tag{2.1}$$

According to how they handle the knowledge and representation of the environment, RL algorithms can be categorized as follows:

- **Model-based**: the agent builds an explicit model $P(s_{t+1}|s_t, a_t)$ and uses it to estimate which action to take based on each transition probabilities and reward r_t.
- **Model-free**: the agent learns and refines a value function $V(s_t)$ solely through interaction with the environment, without any prior knowledge of its dynamics or transition probabilities.

According to how they utilize the learning data during training, RL algorithms can also be categorized as:

- **On-Policy**: the agent uses the current version of its policy to interact with the environment and acquire data. This data is then used to update that same policy.

Table 2.1: Reinforcement Learning parameters.

Parameter	Designation	Description
s_t	State	Instance of the state space S, a set of all possible states of the environment.
a_t	Action	Command that the agent executes based on its policy or strategy, selected from a set of possible actions A.
r_t	Reward	Scalar feedback signal that the agent receives from the environment after taking an action.
$\pi(s_t)$	Policy	Agent's strategy for selecting actions based on the current state. Maps states to actions.
$V(s_t)$	State-Value Function	Function that estimates the expected cumulative reward starting from a given state and following a particular policy.
$Q(s_t, a_t)$	Action-Value Function	Function that estimates the expected cumulative reward for selecting the action a_t in the state s_t following a particular policy.
γ	Discount Factor	Determines the importance of future rewards relative to immediate rewards, 0 means only immediate rewards matter while 1 means that future rewards are equally important.
α	Learning Rate	Controls how much the agent updates its value estimates based on new information. The higher learning rate, the more reactive the agent will be to new information.

- **Off-Policy**: the agent acquires data by following one policy, denominated as behavior policy, while learning and improving a different policy called target policy, using the acquired data.

2.2.1 Policy Evaluation

The goal of policy evaluation is to assess the performance of a given policy in the environment. In order to maximize the expected cumulative reward achieved by following a policy π, the agent must estimate the value function for that policy, which can be categorized as:

- **State-Value** function $V(s_t)$: Quantifies how good it is for the agent to be in a specific state s_t under the policy $\pi(s_t)$.
- **Action-Value** function $Q(s_t, a_t)$: Represents the value of taking action a_t in state s_t and following the policy $\pi(s_t)$.

Some of the most prominent policy evaluation methodologies include Dynamic Programming, Monte Carlo and Temporal Difference Learning [21]:

- **Dynamic Programming (DP)**: model-based approach that involves solving Bellman equations iteratively through policy or value iteration to converge to the optimal policy or value function:

$$V(s_t) \leftarrow r_t + \gamma \cdot \sum_{s_{t+1} \in S} P(s_{t+1}|s, a_t) \cdot V(s_{t+1}) \tag{2.2}$$

- **Monte Carlo (MC)**: estimates the value function by simulating complete episodes, starting from a given state and following the policy until the end of the episode. The value $V(s_t)$ is updated using the observed return r_t and the learning rate α:

$$V(s_t) \leftarrow V(s_t) + \alpha \cdot [r_t - V(s_t)] \tag{2.3}$$

- **Temporal Difference (TD)**: combining ideas from the previous two methods, TD learn-

ing updates the value function based on the difference between the current estimate and an estimate from a subsequent state, using the observed immediate reward r_t and the estimated value of the next state $V(s_{t+1})$:

$$V(s_t) \leftarrow V(s_t) + \alpha \cdot [r_t + \gamma \cdot V(s_{t+1}) - V(s_t)] \tag{2.4}$$

2.2.2 Exploration vs. Exploitation

The exploration versus exploitation trade-off is a critical aspect of RL [16]. Exploration refers to the process of trying out random actions to gather more information about the environment's dynamics in order to potentially discover better actions that lead to higher long-term rewards. Exploitation, in turn, involves using the agent's acquired knowledge and experience to select actions that are expected to yield high immediate rewards based on the learned policy or value function. Overemphasizing exploration might lead the agent to not take advantage of the learned knowledge and, consequently, make uninformed decisions. On the other hand, an excessive focus on exploitation might result in suboptimal actions and global policy.

Model-free RL algorithms use an epsilon-greedy (ϵ-*greedy*) exploration strategy, where the agent explores by selecting a random action with probability ϵ and exploits by choosing the action with the highest estimated value, with probability of 1-ϵ.

2.2.3 Q-Learning

Q-learning [23] is a model-free off-policy RL algorithm used to optimize, by trial-and-error, an agent's ability to make decisions.

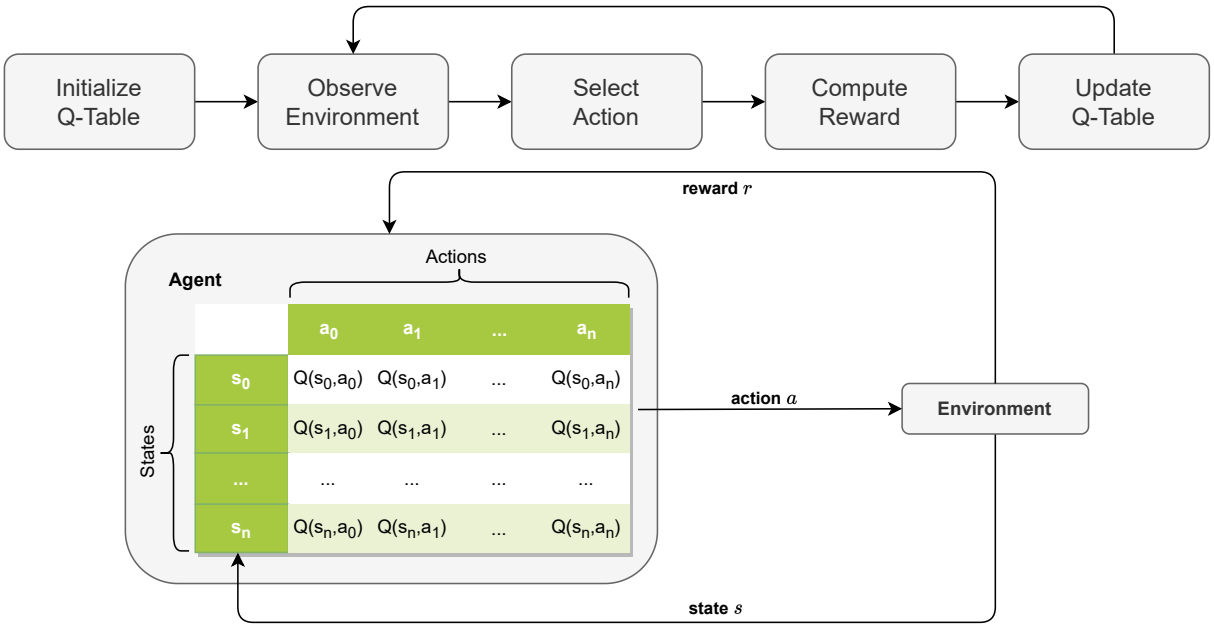


Figure 2.2: Q-Learning pipeline.

As illustrated in Fig. 2.2, the Q-learning algorithm starts by initializing, with arbitrary *Q-values*, a *Q-table* composed by a row for each state, and a column for each action. Through

interaction with the environment, the agent selects an action $a \in \mathbb{A}$ using an ϵ-*greedy* exploration method, previously described in Section 2.2.2. It can either explore, by picking a random action, or exploit, by picking the action with the highest Q-value for the current state $s \in \mathbb{S}$. The agent then moves into a new state and receives a reward from the environment that is used to update the *Q-table* with *Q-values* which represent the predicted usefulness of performing a specific action in a particular state. The *Q-values* are computed using the Bellman equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [r_t + \gamma \cdot max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.5}$$

- $Q(s_t, a_t)$: State-action pair prediction;
- α: Learning rate;
- r_t: Immediate reward received after taking the action a_t;
- γ: Discount factor that balances immediate and future rewards;
- $max_a Q(s_{t+1}, a)$: Highest expected reward for all possible actions a in state s_{t+1}.

2.3 Deep Learning

Deep Learning (DL) [12, 13] is a subfield of ML that emerged from the foundation of Artificial Neural Network (ANN) [24] and focuses on artificial intelligence, making use of Deep Neural Networks (DNNs) and their ability to learn and make predictions or decisions autonomously.

2.3.1 Artificial Neural Networks

ANNs are inspired by the structure and function of the human brain and designed with multiple layers of interconnected nodes, known as artificial neurons or perceptrons. As illustrated in Fig. 2.3, each neuron takes input values, performs a computation, and passes the result to the next neuron (feed-forward) until the final output layer is produced. The intermediate layers between the input and output layers are called hidden layers and allow the network to learn hierarchical representations of the input data, extracting increasingly abstract features at each layer. Each connection between layers has a weight w which determines the strength of the connection and a bias b which allow the network to make predictions even when all input values are zero or near-zero.

The computation is performed through the application of a nonlinear activation function (f_x) to the sum of a bias term (b) and the product of the signals from active neurons in preceding layers (x_j) and the corresponding weights ($w_{i,j}$) associated with the channels that establish connections to the neuron, for each one of the n neurons:

$$\hat{y}_i = f_x(b + \sum_{j=1}^{n} x_j \cdot w_{i,j}) \tag{2.6}$$

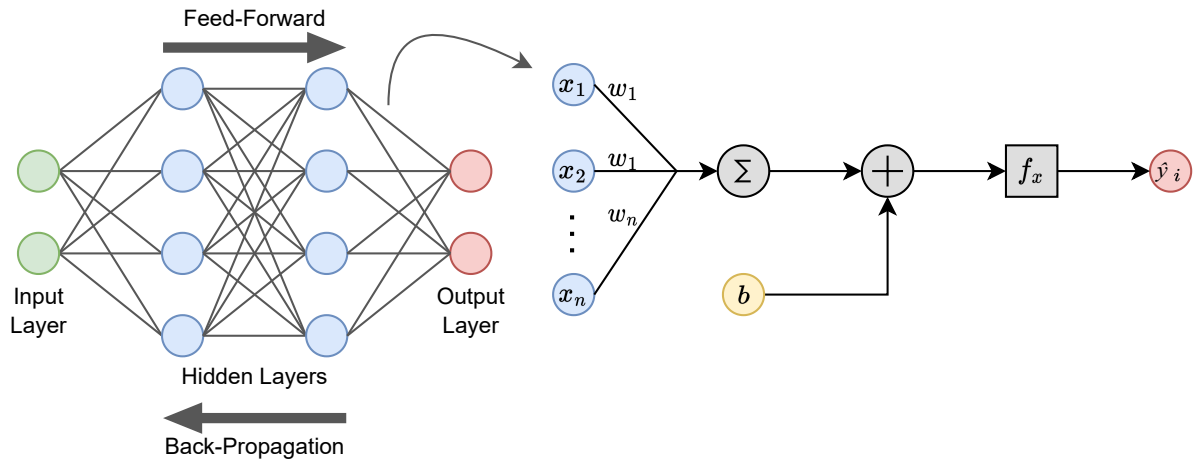


Figure 2.3: Artificial Neural Network architecture.

The activation function f_x is a crucial component in ANNs that introduce nonlinearity to the network's output, enabling it to model complex relationships in data and learn intricate patterns. They determine whether a neuron should be activated or not based on the weighted sum of the inputs. Some popular activation functions include [12]:

- **Sigmoid**: maps the input to a value between 0 and 1.

$$f_x = \frac{1}{1 + e^{-x}} \tag{2.7}$$

- **Hyperbolic Tangent (tanh)**: maps the input to a value between -1 and 1.

$$f_x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.8}$$

- **Rectified Linear Unit (ReLU)**: replaces negative inputs with zero and passes positive inputs unchanged.

$$f_x = \max\{0, x\} \tag{2.9}$$

2.3.2 Neural Network Training

The neural network training is a key process of DL that involves the tuning of the network's parameters θ, denoted as $(w; b)$ and randomly initialized. The goal is to enable the network to make accurate predictions or classifications. The training process is segmented into the following phases:

1. **Forward propagation**: data fed into the input layer travels through the hidden layers via weighted connections until an output layer is produced.

2. **Loss calculation**: the output of the network y_i is compared to the actual target values $\hat{y}_i$ and a loss function $L(\theta)$ is used to quantify the discrepancy between the network's prediction and the actual target. One of the most commonly used loss function is the

Mean Squared Error (MSE) function:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.10}$$

3. **Backpropagation**: using the chain rule, the gradient of the loss associated to θ is calculated, starting from the output layer and moving backward through the network. This gradient indicates how much each parameter contributed to the error.

4. **Gradient descent**: the gradients calculated during the backpropagation process are utilized to update the network's parameters θ. This update involves both magnitude and direction, which are determined by optimization algorithms like Stochastic Gradient Descent (SGD) [25] and Adaptative Moment Estimation (ADAM) [26]. This iterative process aims to adjust the parameters θ as to minimize the loss function $L(\theta)$.

The primary goal of training Neural Networks (NNs) is to ensure that the resulting fine-tuned model demonstrates satisfying performance not only on the training data but also on similar data during testing. Nonetheless, throughout experimentation, the intricate and potentially inaccurate learning process can give rise to certain behavioral challenges. The two most prevalent issues encountered when training DNNs are the following:

- **Overfitting**: the network excels in performance exclusively on the training data.
- **Underfitting**: the model lacks the capacity or complexity to represent the relationships present in the dataset. This is often caused by poor design of the networks' architecture or insufficient training.

Several methods have been proved to be efficient in addressing overfitting [27]. These include techniques such as early stopping, network reduction, data expansion, and regularization. When it comes to tackling underfitting, the optimal strategies involve augmenting the model's complexity and extending the training duration.

2.3.3 Neural Networks Architectures

The deployment of DNNs has shown remarkable success across diverse domains such as computer vision, natural language processing, speed recognition, and reinforcement learning. The variety of applications has prompted the use of different neural network architectures, including:

- **Convolutional Neural Network (CNN)**: used for image and video analysis tasks [28].
- **Recurrent Neural Network (RNN)**: used for sequential data, like text and speech [29].

2.3.3.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [28, 12, 13] are a specialized type of DNN architecture designed for processing grid-like data, such as images or videos. Their distinguishing factor

lies in the integration of convolutional layers, enabling them to automatically learn and identify patterns in visual data. This type of DNN have led to significant advances in image recognition and object detection, among others tasks.

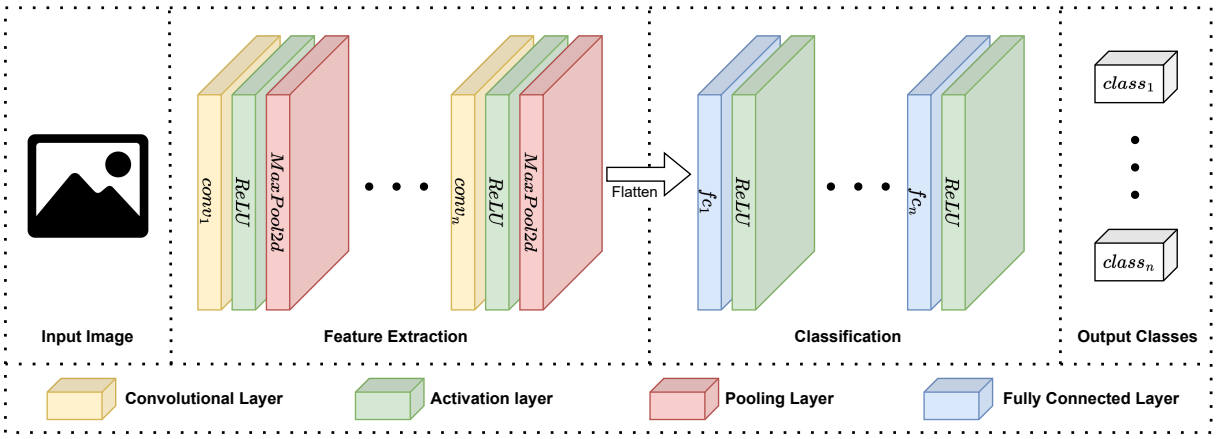


Figure 2.4: Overview of Convolutional Neural Networks architecture for image classification.

As shown in Fig. 2.4, a general architecture of a CNN includes five different components:

- **Convolutional layers**: responsible for feature extraction. They apply filters or kernels to small regions of the input data. These filters learn to detect specific basic features such as edges, corners and textures. The filter is slid over the input data and dot products are computed to produce a feature map. This convolutional operation captures local patterns and allows the network to learn hierarchical representations.
- **Pooling layers**: often used after the convolution layers to reduce the spatial dimensions of the feature map while retaining important information. Max pooling is a popular pooling technique where the maximum value within a region of the feature map is taken, effectively downsampling the data.
- **Activation layers**: used to apply activation functions to the output of convolutional layers in order to introduce non-linearity to the network and help it learn more complex patterns.
- **Batch normalization layers**: often applied after the activation layers to stabilize and accelerate the training process. It normalizes the mean and standard deviation of the activations, reducing internal covariate shift and improving gradient flow.
- **Fully connected layers**: dense layers responsible for classification. They process the flattened features outputted by the last convolutional layer and make final predictions.

2.3.4 Transfer Learning

Transfer Learning (TL) [30] is a ML technique that involves leveraging knowledge acquired from training a model on one task to improve performance on a different, yet related, task. Instead of training from scratch, a pre-trained model's learned features, weights, or architecture are used as a foundation for a new model, making learning faster and more effective. This

approach has proved particularly useful when there is limited data for the task or when training from scratch is resource-intensive.

CNNs frequently take advantage of this method, using networks pretrained on large datasets like ImageNet [31] and fine-tuning them for specific tasks with smaller datasets. This approach leverages the learned features to boost performance on new tasks.

2.4 Deep Reinforcement Learning

Despite achieving satisfactory results in the past, RL algorithms were always limited in terms of scalability and constrained to low-dimensional problems. This is due to their tabular framework, which grows computationally over expensive when dealing with high-dimensional data. To overcome this challenge, researchers introduced DNNs into RL architectures, providing them powerful tools such as function approximation and representation learning [16]. The combination of DL and RL methods defines the revolutionary field of DeepRL, whose framework is illustrated in Fig. 2.5. This technique has enabled robots to interact with complex and dynamic environments and make accurate decisions [32].

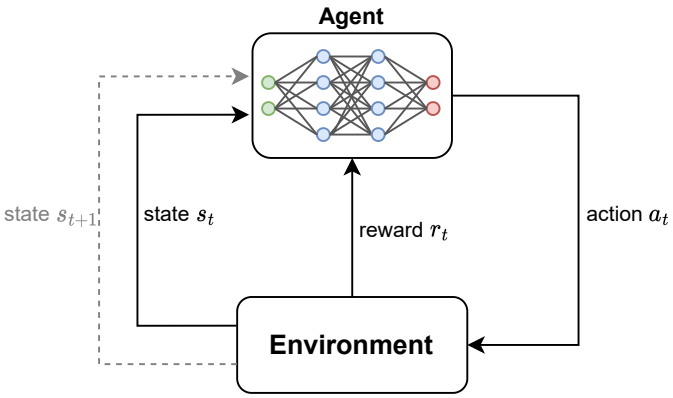


Figure 2.5: Deep Reinforcement Learning framework.

2.4.1 Value-based vs. Policy-based Learning

DeepRL agents employ one of two different learning methodologies to learn optimal decision-making strategies: Value-based and Policy-based. These strategies diverge in how they approach the learning process and represent the agent's knowledge:

- **Value-based**: the agent focuses on learning the values associated with different state-action pairs using DNNs as action-value functions $Q(s_t, a_t; \theta)$.
- **Policy-based**: the agent focuses on directly learning the best decision-making strategy by optimizing the policy $\pi(s_t, a_t; \theta)$ using DNNs.

2.4.2 Deep Q-Learning

Deep Q-Learning, also denominated DQN is a value-based learning method introduced by Mnih *et al.* [17] that integrates DL with the Q-learning algorithm described in Section 2.2.3. By

replacing *Q-tables* with neural networks to estimate the *Q-values*, it becomes able to handle high-dimensional state spaces, therefore improving the performance and stability of the traditional Q-learning method.

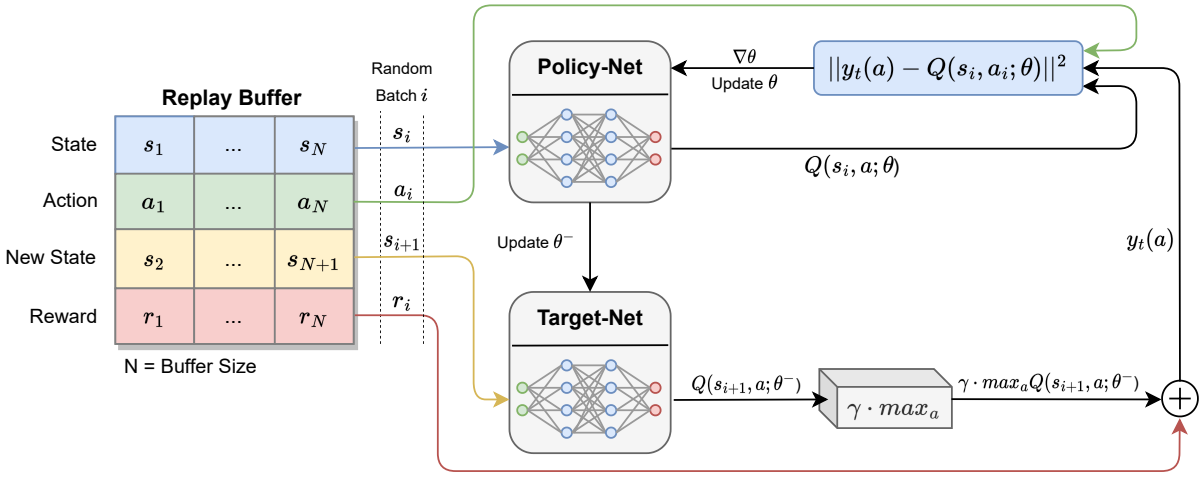


Figure 2.6: Deep Q-learning training stage overview.

As illustrated in Fig. 2.6, the DQN framework comprises three main components: a policy network, a target network, and a replay buffer. The policy network (θ) is fed with the current state of the environment s_t and approximates *Q-values* for all possible actions a_t. The target network (θ^-), in turn, is used to compute the target Q-values for updating the policy network, in order to stabilize the learning process and prevent *Q-value* divergence. The target network mirrors the policy network's architecture but undergoes less frequent parameter updates $\theta^- \leftarrow \theta$.

While in Q-learning, the computation of the *Q-values* relies on the conventional Bellman equation 2.5, DQN operates under the assumption of the Bellman equation's optimal condition (α =1):

$$Q(s_t, a_t) \leftarrow r(s_t, a_t) + \gamma \cdot \max_a Q(s_{t+1}, a) \tag{2.11}$$

The replay buffer [33] is an experience replay mechanism used by DQN to store transitions (state, action, next state, reward), witnessed in each training episode step t, in a buffer with size N. This buffer helps in breaking the temporal correlations between consecutive transitions, leading to more stable learning.

In the training phase, the agent starts by interacting with the environment, observing the current state s_t, selecting an action a_t through an ϵ-greedy exploration strategy, receiving a reward r_t and transitioning to the next state s_{t+1}, filling the replay buffer with transition tuples (s_t, a_t, s_{t+1}, r_t). After each step, a random batch of tuples (s_i, a_i, s_{i+1}, r_i) is sampled from the buffer and fed into the policy and target networks. Subsequently, a loss value is computed based on the networks' outputted *Q-values*, typically using the MSE function:

$$L(\theta) = ||y_t(a) - Q(s_i, a_i; \theta)||^2 \equiv ||r_i + \overbrace{\gamma \cdot \max_a Q(s_{i+1}, a; \theta^-)}^{\text{Target Net}} - \overbrace{Q(s_i, a_i; \theta)}^{\text{Policy Net}}||^2 \tag{2.12}$$

The loss is then backpropagated through the policy network to update its parameters θ:

$$\triangle_\theta = \alpha \cdot L(\theta) \cdot \nabla_\theta Q(s_i, a_i; \theta) \tag{2.13}$$

In the testing or online phase, only the trained policy network is used, and a low exploration rate ϵ is set, to prioritize exploitation. The agent interacts with the environment, using the learned *Q-values* to select the action that yields the highest expected total return $Q^*(s_t, a; \theta^*)$:

$$a_t \leftarrow \underset{a \in Q(s,:;\theta^*)}{\text{argmax}} \; Q(s_t, a; \theta^*) \equiv Q^*(s_t, a; \theta^*) \tag{2.14}$$

Ideally, the agent is able to use the fine-tuned network model θ^* to drive its decision-making process accurately.

2.4.3 Dueling Deep Q-Network

DDQN is an extension of DQN proposed by Wang *et al.* [34] that separates the neural network into two streams (Fig. 2.7):

- **Value stream** fc_V: composed by a single neuron layer, it estimates the state value $V(s_t)$ which represents the baseline level of expected reward from the state s_t regardless of the action.
- **Advantage stream** fc_A: composed by a layer with number of neurons equal to the actions space k. It estimates the advantage value $A(s_t, a_t)$, which indicates the advantage of taking a particular action a_t in that state s_t

This two stream network architecture allows the algorithm to learn which actions are valuable in which states separately, improving its stability and learning efficiency. The advantage value is calculated by subtracting the predicted state value $V(s_t)$ from the optimal state-action $Q^*(s_t, a_t)$:

$$A(s_t, a_t) = Q^*(s_t, a_t) - V(s_t) = r_t + \gamma \cdot V^*(s_{t+1}) - V(s_t) \tag{2.15}$$

The *Q-value* $Q(s_t, a_t)$ is then estimated by adding the state value $V(s_t)$ to the difference between the advantage value $A(s_t, a_t)$ and the mean of the advantage of all k actions. This difference ensures that the advantage values are centered around zero, which helps in stabilizing learning:

$$Q(s_t, a_t) = V(s_t) + A(s_t, a_t) - \frac{1}{k} \sum_{a=1}^{k} A(s_t, a) \tag{2.16}$$

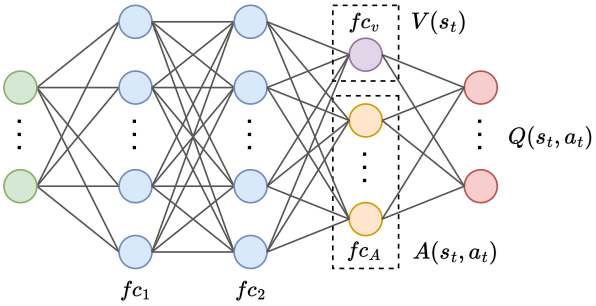


Figure 2.7: Dueling Deep Q-Network architecture.

Thus, in DDQN, the policy network update process is not performed only considering the output neuron with the highest *Q-value*, like in DQN, as it's also given information about every experience from the state value neuron. Therefore, the value stream allows for more efficient learning.

In DDQNs, the policy network update process differs from DQNs. It isn't solely based on the output neuron with the highest *Q-value*, as it also use information from every experience through the state value neuron. This inclusion of the value stream results in more efficient learning.

2.4.4 Double Deep Q-Network

In standard DQN algorithms, the Q-network uses the same set of parameters to both select the best action and estimate its value (see Equation 2.17), which can lead to overestimation of *Q-values*, particularly in cases where the agent hasn't learned the true *Q-values* accurately. Hasselt *et al.* proposed a solution to mitigate this issue by decoupling the action selection and value estimation steps (see Equation 2.18), giving rise to the D2QN architecture [35].

$$y^{DQN}(t) = r_t + \gamma \cdot \max_a Q(s_{t+1}, a; \theta^-) \tag{2.17}$$

$$y^{D2QN}(t) = r_t + \gamma \cdot Q(s_{t+1}, \operatorname*{argmax}_a Q(s_{t+1}, a; \theta); \theta^-) \tag{2.18}$$

Analogously to DQN, D2QN uses the policy network θ to select the action with the highest *Q-value* for the current state. However, while DQN uses the same policy network to estimate the *Q-values* for the selected action, D2QN uses the target network θ^-.

2.4.5 Dueling Double Deep Q-Network

The combination of the dueling architecture proposed by Wang *et al.* and the double variant proposed by Hasselt *et al.* resulted in the D3QN. This method benefits from the use of two separate DNNs to estimate the optimal action values, as well as the separation of the value function into two components. This allows the algorithm to more effectively learn the optimal action-selection policy, leading to a stabler and faster learning of value functions and policies.

2.4.6 Curriculum Learning

Curriculum learning [36, 37] is a ML technique where data is presented to a model in a structured order, starting on simpler examples and gradually progressing towards more complex ones. This method benefits from:

- **Facilitating convergence**: by starting on simpler examples, this method can speed up convergence and help the model avoid local minima during training;
- **Improving long-term generalization**: learning from simpler cases initially can lead to better generalization on more complex cases in later stages.

Curriculum learning has shown promising results in tasks such as image recognition, sequence generation, and reinforcement learning, as it guides the learning towards a more efficient and effective process.

This technique can be applied to RL algorithms, where the agent undergoes training in a series of progressively more complex environments, exemplified in Fig. 2.8. The agent is initially trained on environment (a) until a fine-tuned network model is obtained, and saved. Following the transfer learning methodology addressed in Section 2.3.4, the saved network's parameters θ are then preloaded in the network, and the agent is trained on the subsequent environment (b). This process repeats until the final and more complex environment (d). As knowledge is transferred from one environment to the next, the sequence of tasks induces a curriculum, which has been proven to enhance the performance of RL-agents on challenging tasks and expedite their convergence to an optimal policy [37].

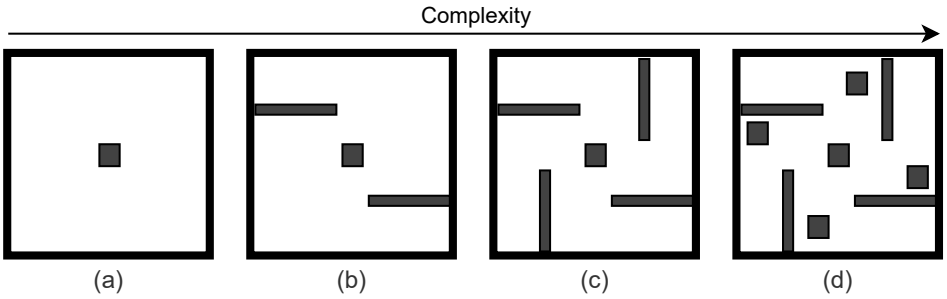


Figure 2.8: Curriculum learning environments.

3

State of the Art

In this chapter, some of the most popular mobile robotic motion planning algorithms are described, with particular focus on optimal value approaches based on DeepRL.

Motion Planning

The robotic motion planning problem has been widely studied and a variety of methods have been implemented to tackle this problem [18].

3.1 Global Motion Planning

Global motion planning algorithms are categorized as traditional algorithms. Under this category are graph-search, sampling-based, and interpolating curve algorithms.

3.1.1 Graph-Search algorithms

Some of the most popular graph-search algorithms include $Dijkstra's$ and A^* algorithms. $Dijkstra's$ [38] works by exploring a graph from a starting node and maintaining a set of visited nodes and unvisited nodes. At each step, it selects the node with the smallest tentative distance from the starting node and updates the tentative distances of its neighboring nodes. This process is repeated until the destination node is reached or until there are no more unvisited nodes. The algorithm then backtracks to find the shortest path from the starting node to the destination node. A^* [39] is an extension of Dijkstra's that uses a heuristic evaluation function to guide the search towards the destination node and improve its efficiency:

$$f(i) = g(i) + h(i) \tag{3.1}$$

- i - robot current localization;
- g(i) - past-cost function from the starting node to i;
- h(i) - Euclidean distance from i to the target node.

3.1.2 Sampling-based algorithms

Sampling-based motion planning is a popular approach that involves sampling random configurations and using them to generate a roadmap of the robot's workspace. Sampling-based methods are widely used in mobile robotic motion planning due to their ability to handle high-dimensional state spaces and complex environments [40]. Some examples of sampling-based methods include Probabilistic Roadmap (PRM) [41], Rapidly-exploring Random Tree (RRT) [42], and their variants. PRM consists of randomly sampling the configuration space of the robot and creating a graph representation of the free space. The algorithm uses this graph to plan collision-free paths for the robot. RRT is based on randomly growing a tree from the start configuration towards the goal configuration. The path planning algorithm then uses this tree to plan collision-free paths for the robot.

3.2 Local Motion Planning

Local motion planning algorithms include reaction-based, classical ML, policy gradient RL and optimal value RL.

3.2.1 Reaction-based algorithms

Reaction-based motion planning algorithms use real-time feedback from sensors to generate and adjust motion plans for the robot. Widely used reaction-based algorithms include Potential Field Method (PFM) [43] and Dynamic Window Approach (DWA) [44]. PFM consists of modelling the robot's environment as a field of attractive and repulsive forces, usually associated with the robot's desired destination and the obstacles in the robot's path, respectively. These forces are then summed up, and the resultant force is used to define the robot's movement course. DWA consists of defining a window of possible velocities and accelerations and dynamically update it based on the robot's current velocity and available space in the environment. The algorithm then selects the next motion command by calculating a cost function for each possible velocity and acceleration pair and choosing the pair with the minimum cost.

3.2.2 Reinforcement Learning algorithms

Reinforcement Learning (RL) and DeepRL have emerged as promising approaches for robotic motion planning, showing great potential in addressing the limitations of traditional methods in handling high-dimensional state spaces and dynamic environments [15, 45].

3.2.2.1 Policy Gradient RL algorithms

Policy gradient RL algorithms directly optimize a parameterized policy to maximize the expected cumulative reward. These algorithms are able to learn stochastic policies that can explore the state space more effectively and can handle high-dimensional state and action spaces efficiently. Some of the most popular policy gradient RL algorithms include A3C [46], PPO [47] and DDPG [48, 49].

Asynchronous Advantage Actor Critic (A3C) [46] is a DeepRL algorithm that combines policy gradient and value-based methods. It allows for efficient parallelization of the learning process, by running multiple copies of the environment and the agent in parallel, with each copy interacting with the environment and updating its own weights asynchronously. This technique uses an actor-critic architecture, where the actor selects actions based on the policy and the critic estimates the value of each state.

Proximal Policy Optimization (PPO) [47] is another DeepRL algorithm that combines policy gradient and value-based methods. This algorithm iteratively optimizes a clipped surrogate objective function, which limits the size of the policy update at each iteration to prevent large changes. PPO uses a value function to estimate the state-value function, which is used to estimate the expected reward of being in a particular state.

Deep Deterministic Policy Gradient (DDPG) [48, 49] is a DeepRL algorithm that combines policy gradient and Q-learning methods to learn a deterministic policy that can be used to directly select actions in continuous action spaces. DDPG uses an actor-critic architecture, where the actor network maps states into actions, and the critic network evaluates the quality of those actions. Similarly to DQN, this method also uses experience replay and target networks to improve stability during training, and is capable of learning in high-dimensional continuous state and action spaces.

3.2.2.2 Optimal value RL algorithms

In optimal value RL algorithms, the robot learns a value function that assigns a value to each possible action in each possible state of the environment. The value function is then used to select the optimal action to take in each state to maximize a reward signal, which is defined by the designer to reflect the robot's objectives. The most popular optimal value RL algorithms are the Q-learning algorithm [8, 50, 51], and the DQN algorithm along with its variants D2QN, DDQN and D3QN.

Mnih *et al.* introduced the DQN [17], a value-based DL method designed upon a Q-Learning architecture [23] that approximates a *Q-function* with a DL architecture, obtaining, at the time, ground-breaking results in learning policies to play Atari games. Over the years many improvements over the DQN were proposed, Wang *et al.* [34] proposed the DDQN, an extension of the DQN method that improves learning efficiency by separating the action-value estimation into state-value and advantage functions, thus enabling more effective exploration and decision-making capability of mobile robots in complex environments. Hasselt *et al.* [35] proposed the D2QN, a DQN variant that addresses the overestimation bias issues present in the original algorithm by using an additional network to estimate a separate Q-value and update the other network, thus disentangling updates from biased estimates.

Due to its remarkable success in learning control policies from raw pixel images in various games [17], DeepRL inspired researchers to apply DQN to robotic motion planning tasks, where agents use raw sensor data to learn collision-free paths in cluttered and dynamic environments.

Ozdemir *et al.* proposed a DDQN-based algorithm [52] that uses LiDAR scans and RGB-D

images to obtain a 3D point cloud and converts it into 2D laser data. This data is then used, along with robot-target distance and angle and a heuristic reward, to obtain motor speed pairs to drive the robot from a source to a target destination.

Lei *et al.* [53] and Wang *et al.* [54] proposed approaches to robot motion planning using the D2QN method. In [53], LiDAR dot matrix information (angle and distance) and local target point coordinates are fed into a CNN and, with a simple and straight forward reward model, fixed-length omnidirectional movements are obtained. In [54], robot-obstacles and robot-target distances (obtained from LiDAR scans), robot location and velocity and target location are used, along with a heuristic reward, to obtain motor speeds. Also, the latter makes use of PER to improve learning efficiency. Leão *et al.* proposed a DQN approach [55] that feeds the robot's distances to obstacles and to the target, obtained through a LiDAR, and a heuristic reward to the DNN which outputs motor speed pairs.

Ramirez *et al.* proposed another approach to robot motion planning [56] that uses D3QN, an extension of DQN that, by combining the D2QN and DDQN variants, benefits from the advantages of both. The network is fed with depth images, polar coordinates (distance and angle) and a simple straight forward reward and outputs motor speed pairs. In addition to this, PER, Multi-step Q-learning, Categorical DQN and NoisyNet methods are implemented, forming the commonly called Rainbow DQN which resulted in a better performance than the standard DQN. A brief summary of the aforementioned works is presented in Table 3.1.

Table 3.1: Summary of DeepRL-based optimal value motion planning strategies.

Article	Simulator	Method	States	Actions	Rewards
Leão *et al.*[55]	Flatland	DQN	Laser measurements	Linear-angular speed pairs	Heuristic reward
Ozdemir *et al.*[52]	Gazebo	DDQN	Laser scan; Robot-target-relative data	Linear-angular speed pairs	Penalty for collision; Reward for reaching the goal; Reward based on velocity and robot-target angle
Lei *et al.*[53]	Pygame	D2QN CNN	2D LiDAR Scan	Fixed-length omni-directional movements of eight directions	Penalties for each step and collision; Reward for reaching the goal
Wang *et al.*[54]	Gazebo	D2QN	Laser measurements; Robot's location and velocity; Target's location	Linear-angular speed pairs	Heuristic reward
Ramirez *et al.*[56]	Gazebo	D3QN	Depth Image and Polar Coordinates	Linear-angular speed pairs	Penalties for each step and collision; Reward for reaching the goal

4

Developed Work

This chapter presents an in-depth description of the implemented DeepRL framework, including the developed ANNs, state, action, and reward models. A description of the simulation environments built to validate the framework is also presented.

4.1 Proposed DeepRL-based Framework

This work proposes a solution to robot local motion planning in unstructured and dynamic environments. To effectively manage the high-dimensional state spaces imposed by such environments, as discussed in Sections 2.4 and 3.2.2, a DeepRL-based approach was adopted, implying the design and implementation of an DNN, and a state, action and reward model.

The developed DeepRL-based framework, depicted in Fig. 1.1 of Section 1.2, features a simulated environment, a costmap representation of the environment, and the Double Dueling Deep Q-Network (D3QN) method addressed in Section 2.4.5. Analogously to any RL methodology, the proposed method is divided into two distinct stages:

- **Training stage**: this phase involves iteratively refining the frameworks' DNN through trial-and-error. The pipeline of the training phase is illustrated in Fig. 4.1 and described in-depth in Algorithm 1. After each training episode where the robot successfully reaches the target, a network model is stored, containing the employed DQN architecture along with its current parameter values θ.
- **Testing or online stage**: a saved network model derived from the training phase is used in this stage to control the robot's movements and verify its decision-making strategy. Ideally, the network model should guide the robot towards the target while avoiding collisions with obstacles.

Throughout the development of this framework, several DNNs, reward models and action sets were explored and refined until achieving satisfactory results. The DNN, reward model and action set that ultimately yielded these results are discussed in the following sections.

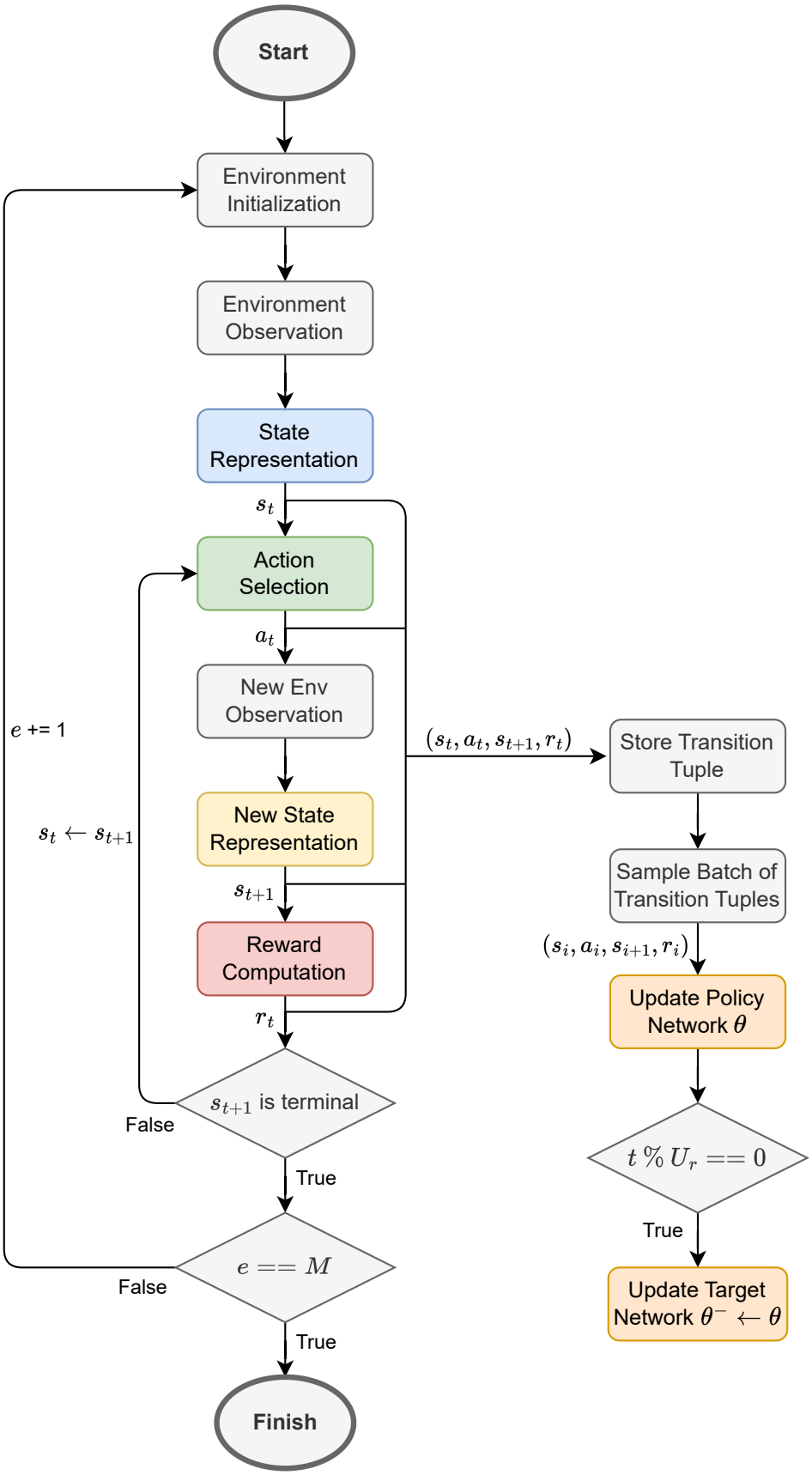


Figure 4.1: Training stage pipeline.

Algorithm 1: DeepRL-based Motion Planning for Indoor Robot Navigation

Define the agent's state, action, and reward models
Define the initial and target points
Define the number of training episodes M and maximum steps T
Define the network's hyperparameters
Define the target network θ update rate U_r
Define the reward propagation window P_w
Initialize the system's ROS nodes and establish its subscriptions and publishers
Initialize the DQN policy and target networks (θ and θ^-)
Initialize the replay buffer D to capacity N and define the batch size N_b
for *episode* $e = 1$ *to* M **do**
    Initialize environment
    for *step* $t = 1$ *to* T **do**
        Take action a_t using an ϵ-greedy exploration method
        Get the robot's odometry data from /odom callback
        Get the sensor readings from /scan callback
        Define s_{t+1}
        if s_{t+1} *is terminal* **then**
            if $d_T \leq d_{min}$ **then**
                Reward the agent, $r_t = R_T$
                Save the DQN model
            else
                Penalize the agent, $r_t = P_O$ **if** *using Reward Propagation* **then**
                    for $i = 1$ *to* P_w **do**
                        Propagate the last P_w rewards in the buffer, $r_i = r_t$
            break
        else
            Compute the reward value r_t following the reward model
        Store transition tuple (s_t, a_t, s_{t+1}, r_t) in D with maximal priority $p_t = \max_{i<1} p_i$
        Sample a batch of N_b tuples (s_i, a_i, s_{i+1}, r_i) from D with probability $P(i)$ (4.1)
        Compute importance-sampling weight w_i (4.3)
        Calculate TD-error δ_i (4.2)
        Update transition priority $p_i \leftarrow |\delta_i|$
        Compute $y_t(a)$ (2.18)
        Compute the policy network Q_{values}, $Q(s_i, a; \theta)$
        Perform a gradient descent step with loss $||y_t(a) - Q(s_i, a_i; \theta)||^2$, updating θ
        if t % U_r = *0* **then**
            Replace the target DQN parameters $\theta^- \leftarrow \theta$
        $s_t \leftarrow s_{t+1}$

4.2 Deep Neural Network

The DeepRL architecture proposed in this work is inspired by the work of Daniel Palaio [1]. In [1], a two-stream D3QN architecture (see Section 2.4.5), shown in Fig. 4.2 is used to enable the RL agent to learn how to perform effective motion planning towards its target, while avoiding static obstacles. In order to efficiently extend this collision-free motion to contemplate not only static but also dynamic obstacles, this work introduced enhancements to the DNN employed in [1] (refer to Fig. 4.2), giving rise to the augmented DNN illustrated in Fig. 4.3.

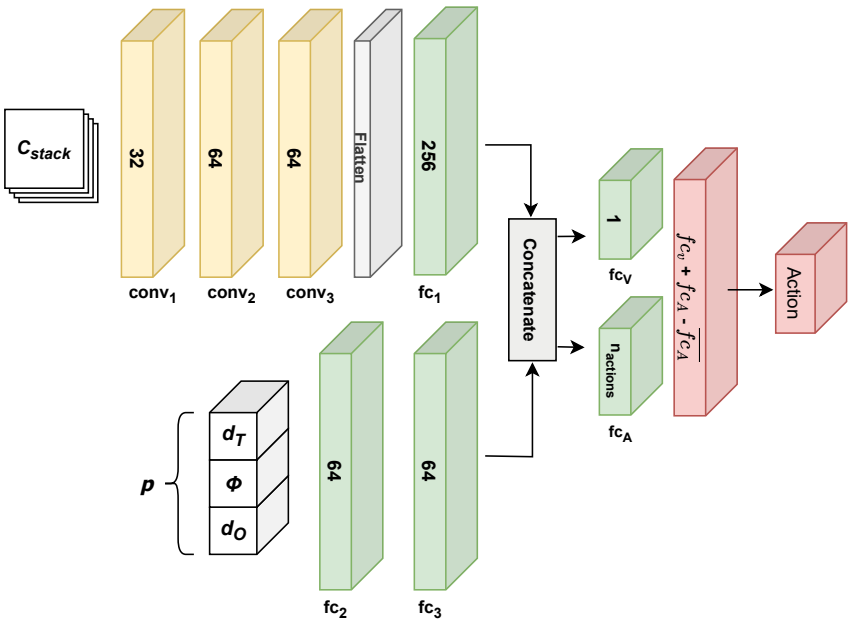


Figure 4.2: Original Dueling Deep Q-Network architecture [1].

The first stream of the proposed DNN (Fig. 4.3) is a feature extraction stream, addressed in Section 2.3.3.1, and is used for dealing with the costmap grid-like representation (C_{stack}). This stream is composed by 6 convolutional layers, detailed in Table 4.1, followed by a flatten layer.

The second stream is composed by 3 fully-connected layers: fc_1, fc_2, and fc_3, described in Table 4.1 and deals with affordance-based inputs $p = \{d_T, \phi, d_O\}$ relative to the robot's Euclidean distances to the target (d_T) and to the nearest obstacle (d_O) and the orientation disparity between the robot and the target (ϕ).

The outputs of these two stream are concatenated and forwarded through an additional set of two fully-connected layers (fc_4 and fc_5). The outputs of these layers are subsequently fed into the advantage and value fully-connected layers (fc_A and fc_V) which are used to compute the *Q-values* considering a dueling architecture (see Section 2.4.3).

Addressing the need for a stable learning process, as discussed in Section 2.4.2, two equal networks with characteristics as aforementioned, are employed as the policy network with parameters θ, and the target network with parameters θ^-. The policy network estimates the *Q-values* for the current state-action pair, $Q(s_t, a_t)$ while the target network estimates the *Q-values* for

the next state-action pair $Q(s_{t+1}, a_{t+1})$. Only the policy network is trained, and its weights are periodically cloned to the target network at a defined interval U_r.

To tackle the problem of state-action value overestimation that leads to impoverished policies, as explained in Section 2.4.4, the D2QN variant [35] is exploited in this work.

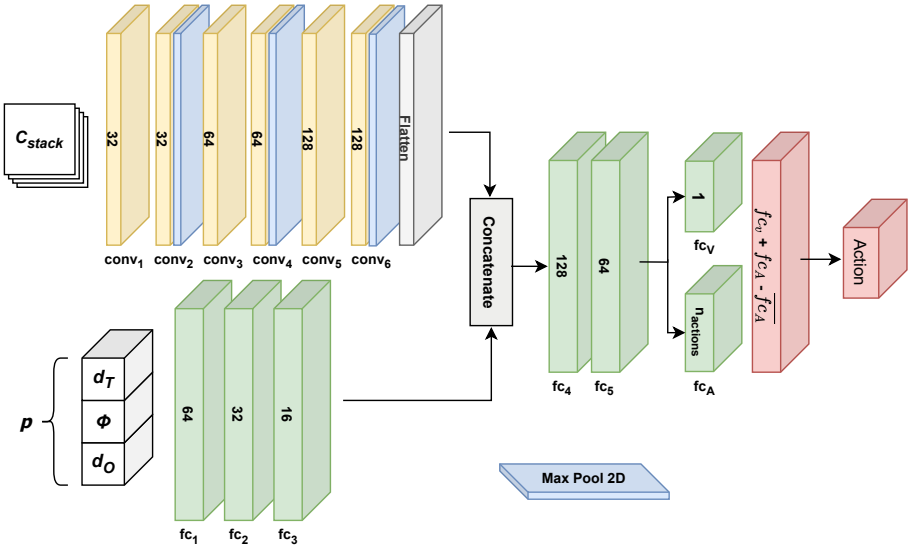


Figure 4.3: Proposed Dueling Deep Q-Network architecture.

Table 4.1: Layer configurations of the proposed DNN.

Layer	Parameters
Input	$\{C_{stack}, P_{Stack}\}$
$conv_1$	Filters = 32, Kernel size = 3, Stride = 1, Activation = ReLU
$conv_2$	Filters = 32, Kernel size = 3, Stride = 1, Activation = ReLU
maxpool	Kernel size = 2, Stride = 2
$conv_3$	Filters = 64, Kernel size = 3, Stride = 1, Activation = ReLU
$conv_4$	Filters = 64, Kernel size = 3, Stride = 1, Activation = ReLU
maxpool	Kernel size = 2, Stride = 2
$conv_5$	Filters = 128, Kernel size = 3, Stride = 1, Activation = ReLU
$conv_6$	Filters = 128, Kernel size = 3, Stride = 1, Activation = ReLU
maxpool	Kernel size = 2, Stride = 2
fc_1	Neurons = 64, Activation = ReLU
fc_2	Neurons = 32, Activation = ReLU
fc_3	Neurons = 16, Activation = ReLU
fc_4	Neurons = 128, Activation = ReLU
fc_5	Neurons = 64, Activation = None
fc_V	Neurons = 1, Activation = None
fc_A	Neurons = $N_{actions}$, Activation = None

4.3 Replay Buffer

In this work, the Prioritized Replay Buffer (PER), an improvement of the experience replay buffer addressed in Section 2.4.2, is exploited.

In the traditional replay buffer, explained in Section 2.4.2, the agent's experiences are stored in the replay buffer and then uniformly sampled from it to be used to update the target network θ^-. However, not every experience is equally valuable or informative for learning and thus, uniformly sampling this experiences makes learning inefficient. The PER, introduced by Schaul *et al.* [57], aims to improve the model's learning efficiency by prioritizing the training samples used.

Whenever a new experience is stored in the PER, a probability of transition P is assigned to it based in the magnitude of the TD-error p_i associated with that experience. During training, the agent samples experiences from the PER with probabilities proportional to their priority values. Each priority value $P(i)$ is formulated as:

$$P(i) = \frac{\|p_i^\alpha\|}{\sum_k \|p_k^\alpha\|} \tag{4.1}$$

Here, α is the impact factor for TD-error, which is defined as δ:

$$\delta_i = r_i + \gamma \cdot Q^-(s_i, \underset{a}{\operatorname{argmax}}\, Q(s_{i+1}, a)) - Q(s_i, a) \tag{4.2}$$

This prioritization of experiences introduces, however, a bias in the sampling process that can lead to overfitting. In order to mitigate this issue, the implemented PER uses a technique called importance sampling for correcting this bias. The sampling weights are calculated based on the priority value $P(i)$, the number of samples in the buffer N, and an impact factor for sampling importance β:

$$w_i = \left(\frac{1}{N \cdot P(i)}\right)^\beta \tag{4.3}$$

The integration of these three components ($P(i)$, δ_i, and w_i) is described in Algorithm 1 (Section 4.1).

4.4 State, Action and Reward Models

4.4.1 State Model

As shown in the proposed DeepRL-based framework illustrated in Fig. 1.1, the agent employs information about its surrounding environment acquired through sensors, specifically a 2D LiDAR sensor. This information, comprising laser scans and distance measurements, is converted into states, specific data structures supported by the framework's DNN.

Traditional DeepRL methods often feed the laser scans directly into the DNN which can result in network updates being influenced by intrinsic and extrinsic features such as sensor positioning, distance range and angular resolution. Drawing inspiration from [1], this work proposes a solution based on costmaps to mitigate these biases in generalization. The proposed costmaps are defined as 40×40 grids with a length of 1-meter, centered in the robot. Each cell within the grid is initialized to 0 at each step and set to 1 if overlapped by any laser scan reading, as illustrated in Fig. 4.4. The convolution layer of the DNN (see Fig. 4.2) is fed with a stack of four sequential costmaps (see Fig. 4.4) in order to represent continuous environment variations rather than short-term paths only.

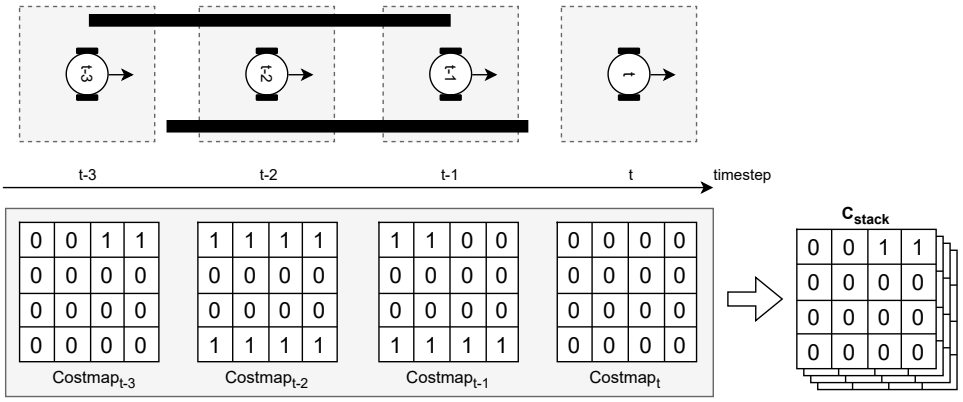


Figure 4.4: Costmap representation and C_{stack} structure.

In addition to the costmap representation, data pertaining to correlations between the robot and the target, as well as the robot and the nearest obstacle $p = \{d_T, \phi, d_O\}$ is integrated into the state model to augment the agent's situational awareness (see Fig. 4.5). Similarly to the C_{stack} structure, affordance-based measurements p from 4 sequential time steps are employed to formulate the structure $P_{stack} = \{p_{t-3}, p_{t-2}, p_{t-1}, p_t\}$. The stop action was exclusively used on environments with dynamic obstacles.

$$p = \begin{cases} d_T : \text{Euclidean distance between the robot and target;} \\ \phi : \text{Orientation disparity between the robot and target;} \\ d_O : \text{Euclidean distance between the robot and the nearest obstacle.} \end{cases}$$

The complete state model is thus formulated as $\mathbb{S} = \{C_{stack}, P_{stack}\}$.

4.4.2 Action Model

As addressed in Section 2.4, DeepRL agents respond to environment observations through actions. Policy gradient algorithms such as PPO and DDPG operate within continuous action spaces, while deep Q-learning algorithms, such as the one proposed in this work, use discrete action spaces. In this work, the robot's actions were employed as linear and angular speed pairs (v, w). The action set defined to control the robot's movement is composed of 29 distinct actions described in Table 4.2.

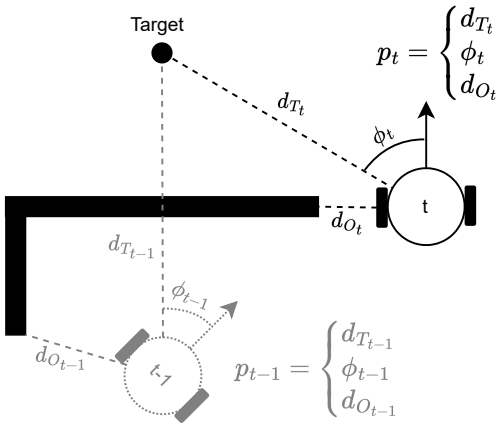


Figure 4.5: Robot-target-relative data representation.

Table 4.2: Action Set.

Linear vel. (m/s); Angular vel. (rad/s)	Description
{0.0; 0.0}	Stop action
{0.15; 0.0}, {0.25; 0.0}	Forward movements
{0.15; 0.25}, {0.15; 0.5}, {0.15; 0.75}, {0.15; 1.0} {0.15; 1.25}, {0.15; 1.5}, {0.15; 2.0} {0.25; 0.5}, {0.25; 0.75}, {0.25; 1.0}, {0.25; 1.25} {0.25; 1.5}, {0.25, 2.0}	Left turns with linear component
{0.15; -0.25}, {0.15; -0.5}, {0.15, -0.75}, {0.15; -1.0} {0.15; -1.25}, {0.15; -1.5}, {0.15; -2.0} {0.25; -0.5}, {0.25; -0.75}, {0.25; -1.0}, {0.25; -1.25} {0.25; -1.5}, {0.25; -2.0}	Right turns with linear component

4.4.3 Reward Model

In DeepRL frameworks, the reward model plays a crucial role in guiding the agent's learning process by providing feedback on its actions. The reward model defines the scalar value r_t assigned to the action a_t based on the originated state transition $s_t \leftarrow s_{t+1}$.

In this work, the reward model is structured to provide high rewards for actions that guide the agent toward a compelling collision-free motion towards the target. This model incorporates three distinct reward components: r_T, r_O and r_S, based on the robot-pose-relative data (see Fig. 4.4) and the number of steps t taken, defined as follows:

$$r_O = 0.05 \cdot (-0.5 + \frac{1}{1 + e^{-50|d_{O_{t+1}} - 0.3|}})$$

$$r_T = 0.05 \cdot \frac{d_{T_t} - d_{T_{t+1}}}{d_{T_{t+1}}} r_O + e^{(-0.9|d_{T_{t+1}}|)} \tag{4.4}$$

$$r_S = 0.05 \cdot \max(-0.01t, -2), \quad t \in [0, T]$$

The r_O component rewards the agent for actions that help maintain a safe distance between the robot and the nearest obstacle. The r_T component rewards the agent for actions that contribute to reducing the robot's Euclidean distance to the target. The r_S component is used to penalize the agent for actions that contribute to an increasing in the time taken to reach the target.

For non-terminal states, the cumulative reward r_t at each step is computed as the sum of these three reward components. For terminal states, the agent receives a substantial reward R_T if the robot's distance to the target is below a specified threshold $d_{T_{min}}$ or a penalty (negative reward) P_O if the robot's distance to the nearest obstacle is below a given threshold $d_{O_{min}}$:

$$r_t = \begin{cases} R_T, & if\ d_T < d_{T_{min}} \\ P_O, & if\ d_O < d_{O_{min}} \\ r_T + r_O + r_S, & otherwise \end{cases} \tag{4.5}$$

In order to address the problem of delayed rewards [8], common in RL, and achieve high levels of performance in complex environments, this work proposes the use of reward propagation. The reward propagation technique consists of assigning credit or blame for a specific outcome to the different actions that contributed to that outcome, improving the agent's learning efficiency over time. The reward is propagated whenever the robot collides with an obstacle by applying the penalty P_O not only to the current step's reward r_t but also to the previous w steps' rewards stored in the replay buffer:

$$r_{n-i} = r_t, \quad i \in [1, w] \tag{4.6}$$

Here, n denotes the number of experiences stored in the replay buffer at step t.

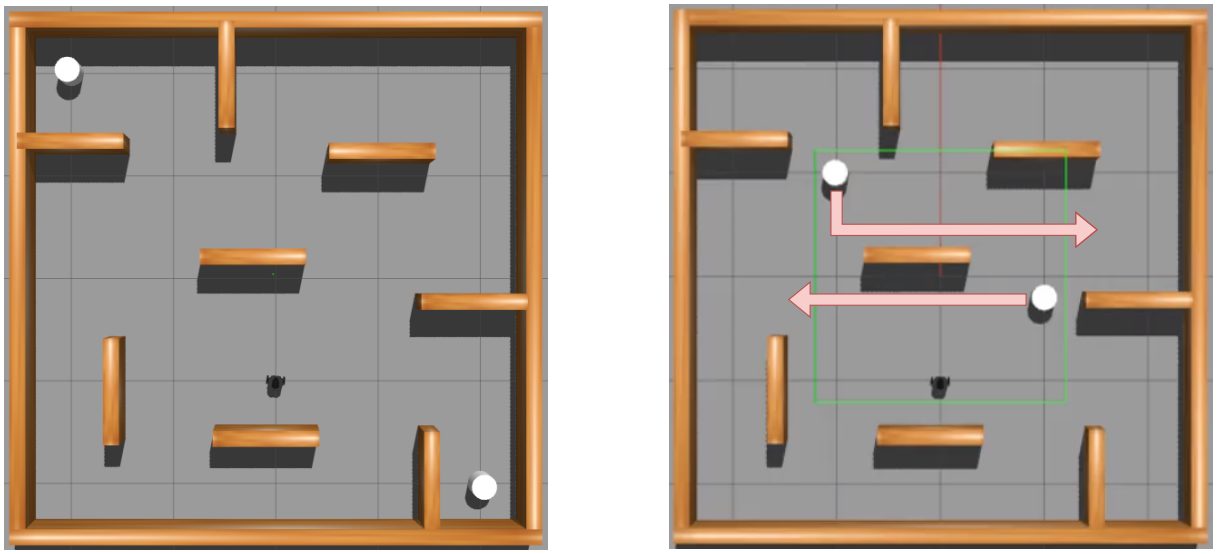
4.5 Simulation Environments

In order to validate the developed framework and evaluate the proposed methods, apart from employing the pre-existing *Stage 4* environment, additional simulation environments were built using the Gazebo software described in Section 5.3.

4.5.1 Dynamic environment

To validate the developed DeepRL-based framework for dynamic motion planning, a simulation environment featuring dynamic obstacles was required. Therefore, this work involved

the creation of the *Stage 4 Dynamic* environment depicted in Fig. 4.6b. This environment was derived from the Gazebo's *Stage 4* environment depicted in Fig. 4.6a.



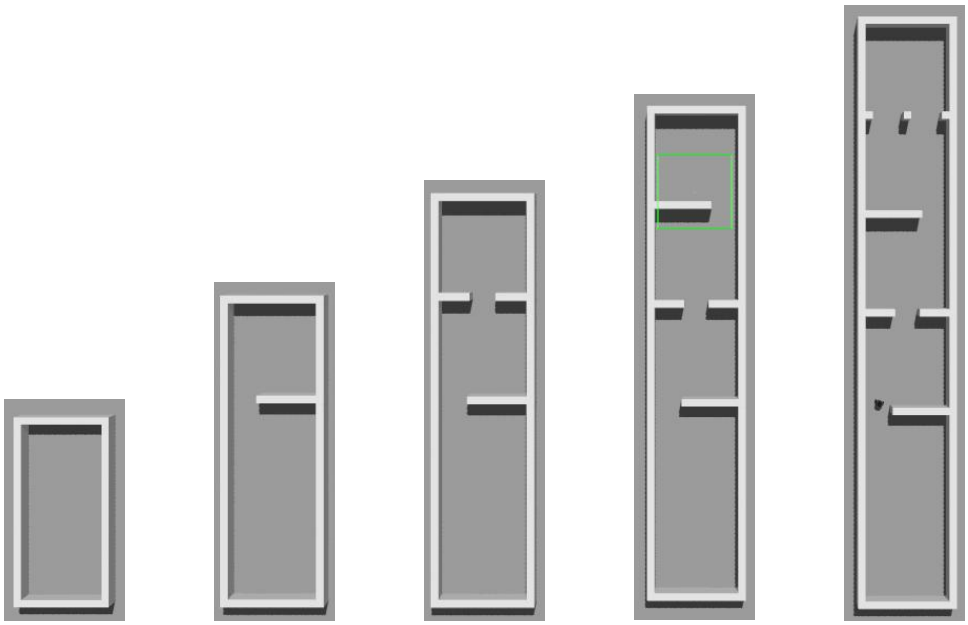
(a) Gazebo's *Stage 4*. **(b)** Gazebo's *Stage 4 Dynamic*.

Figure 4.6: Gazebo's *Stage 4* environments.

Similarly to the original Gazebo's *Stage 4* environment, the *Stage 4 Dynamic* environment, shown in Fig. 4.6b, has four walls with five meter length defining the workspace area. Furthermore, eight additional walls with a length of one meter are positioned within the workspace to simulate static obstacles. However, the *Stage 4 Dynamic* environment introduces motion to the two white cylindrical objects present in the *Stage 4* environment. These cylinders follow a designated trajectory indicated by the red arrows with a velocity of 0.2 meters per second.

4.5.2 Curriculum Environments

To evaluate the application of curriculum learning techniques (discussed in Section 2.4.6) to the developed DeepRL framework, a set of five environments, represented in Fig. 4.7, was created within Gazebo. The base configuration of all five environments consists of a rectangular layout with four walls delineating the operational space for the robot. In addition to this base setup, supplementary walls are gradually added into each environment to simulate additional static obstacles, progressively increasing each environment's complexity in comparison to the preceding one.



(a) (b) (c) (d) (e)

Figure 4.7: Gazebo's Curriculum environments.

5

Software Tools and Hardware Materials

This chapter introduces and briefly describes the software and hardware resources employed to achieve the defined objectives.

5.1 Ubuntu Operating System

Ubuntu [58] is a free and open-source Linux distribution based on Debian, and is widely used for desktop computing. It is developed and maintained by Canonical, and is a popular choice for developing robot frameworks due to its support for a wide range of both hardware and software components, as well as its user-friendly nature and adaptability. Ubuntu includes a variety of software tools and libraries that are commonly used in the development of robotic applications, highlighting ROS [59] (addressed in Section 5.2), and Gazebo (Section 5.3). It also offers a range of programming languages and development tools, including Python, C++, and Git, which are commonly used in robot development.

The version chosen for this work was Ubuntu 20.04 LTS, a long-term support release of the Ubuntu operating system, released on April 23, 2020.

5.2 Robot Operating System

Robot Operating System (ROS) [59] is an open-source framework designed for building and programming robotic applications. It provides a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across diverse robotic platforms. ROS is characterized by its modular and scalable architecture, which promotes code reusability and the integration of new components into existing systems. It supports multiple programming languages including Python, C++ and Java, and provides a distributed computing infrastructure that enables efficient communication between different processes and devices. The main components of the ROS framework are outlined in Table 5.1. Some key features of ROS include:

- **Node-based architecture**: ROS employs a node-based architecture in which each node is a separate process, facilitating internode communication through messages;

- **Message passing**: ROS provides a flexible and efficient mechanism for communication between different components of a robot system;
- **Package management**: ROS provides a range of both 2D and 3D visualization tools useful for understanding and debugging robot systems.

Table 5.1: ROS framework main components.

Component	Description
Core	Provides the foundational components
Master	Manages the communication between nodes
Parameter Server	Stores and manages parameters used by nodes
Nodes	Individual processes that make up a ROS system
Topics	Named channels that allow nodes to send and receive messages
Messages	Data structures used to communicate information between nodes
Services	Allow nodes to synchronously request and receive data from other nodes

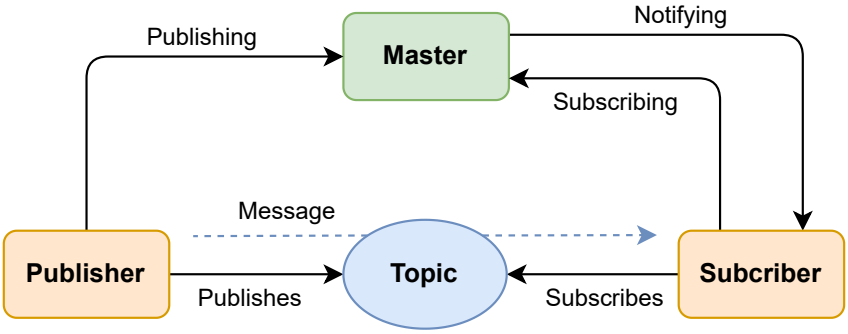


Figure 5.1: Publisher-subscriber ROS Protocol.

The publisher-subscriber protocol provides a way for different components (nodes) within a robotic system to communicate without tight coupling, enhancing the modularity and flexibility of the system. This protocol operates through the following steps:

1. A publisher node informs the master node that it is publishing messages on topic A;
2. A subscriber node informs the master that it wishes to subscribe to topic A;
3. The master notifies the subscriber whenever a message in published in topic A;
4. Upon receiving notification from the master, the subscriber establishes a connection with the publisher to receive the message.

To ensure compatibility between the ROS and the various hardware and software components utilized in this project, the thirteenth distribution release of ROS was employed: ROS Noetic Ninjemys. The publisher-subscriber protocol depicted in Fig. 5.1 was exploited, and specific ROS topics were created and utilized, as enumerated and detailed in Table 5.2.

As previously discussed, a DeepRL framework operates through interactions of two key

Table 5.2: ROS topics.

Topic	Message Type	Description
/odom	nav_msgs/Odometry	Robot position and orientation
/scan	sensor_msgs/LaserScan	Cluster of scanned points
/tf	tf2_msgs/TFMessage	Translation and rotation between frames
/cmd_vel	geometry_msgs/Twist	Linear and angular speeds
/costmap	nav_msgs/OccupancyGrid	Binary grid with occupation information
/map_metadata	nav_msgs/MapMetaData	Grid width and height

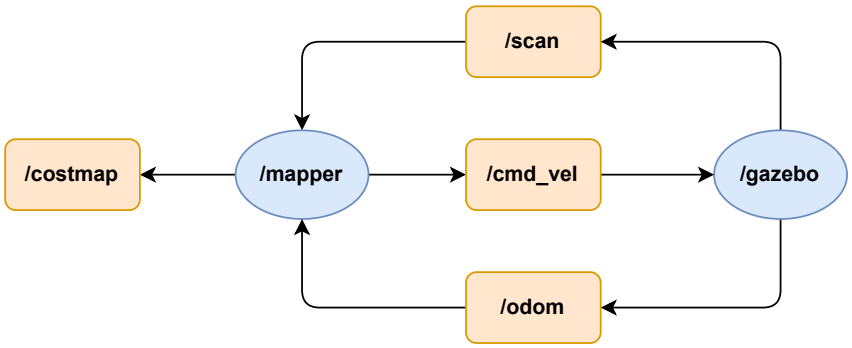


Figure 5.2: ROS computation graph.

components: the agent and the environment. The communication between them is established via the /mapper and the /gazebo ROS nodes.

As illustrated in Fig. 5.2, the /mapper node subscribes to the /scan and /odom topics to acquire LiDAR sensor readings and the robot's odometry data, respectively. The agent uses this data to build the C_{Stack} and the P_{Stack}, as described in Section 4.4.1, and computes the actions (detailed in Section 4.4.2). These discrete actions are subsequently converted into geometry_msgs/Twist to match ROS message type and published in the /cmd_vel topic to control the virtual robot's motors.

For diagnostic purposes, an additional topic, /costmap, was employed to visualize the costmaps generated by the agent at each step. This topic subscribes to the /mapper node, which supplies nav_msgs/OccupancyGrid messages containing an occupancy grid computed from data received from the /scan topic.

5.2.1 RViz

As mentioned in Section 5.2, the ROS framework offers various visualization tools, and one prominent tool is RViz [60]. RViz is a 3D visualization tool that allows users to visualize and interact with sensor data and robot models. It is often used for robot simulation, testing and debugging, providing a user-friendly interface for real-time display of sensor data such as point clouds, laser scans and camera images. Moreover, RViz facilitates the visualization of 3D models of robots and their surrounding environment and interact with them using tools like selection,

translation, and rotation.

In the context of this work, RViz played an important role in visualizing the costmaps created at each time step, as detailed in the previous Section 5.2. Figure 5.3 presents an example of RViz's visualization of a costmap produced from the obstacles present in the Gazebo environment.

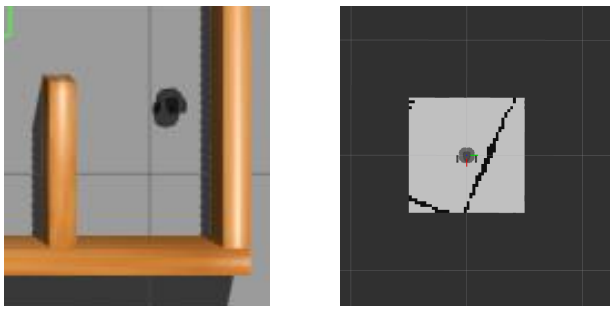


Figure 5.3: Gazebo environment and respective RViz costmap representation.

5.3 Gazebo

The robot development is often a costly, time-consuming and potentially unsafe process. Robotic simulation involves creating virtual environments in which robotic systems can be tested, validated, and optimized. This helps avoid potential injuries and damages, prevents the need for design changes after part production has started, reduces cycle times in manufacturing processes, and minimizes the need for excessive paperwork [61]. Therefore, simulation plays a vital role in modern robotics by reducing costs, enabling rapid experimentation, and enhancing the quality of robotic systems.

Gazebo is an open-source 3D multi-robot simulation environment [62]. Is allows users to simulate the behavior and interactions of multiple robots in a virtual environment before deploying them in the real world [63]. Gazebo can also simulate a wide range of sensors, including cameras, LiDAR, and sonar, while incorporating physics features such as gravity and friction to make the simulation more realistic.

To integrate the implemented ROS framework with Gazebo simulator, the ROS meta package *gazebo_ros_pkgs* [64] was utilized. This meta package includes multiple packages that provide the necessary interfaces for simulating a robot in Gazebo using ROS messages, services, and dynamic reconfigure [62].

5.4 Turtlebot3

For the deployment of the proposed DeepRL framework within the simulation environment addressed in the previous Section 5.3, a mobile robot is essential to acquire and transmit all the data necessary to the software control modules. To fulfill this requirement, the TurtleBot3 Burger, depicted in Fig. 5.5, was selected.

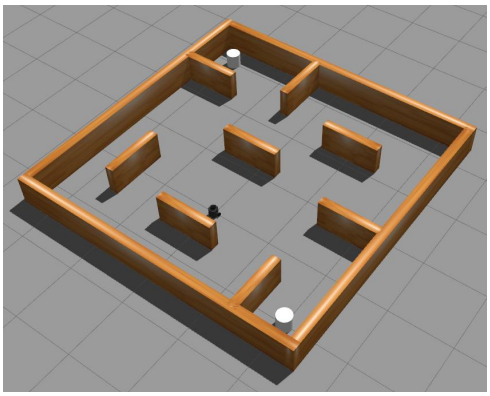


Figure 5.4: Gazebo's *Stage 4* environment.

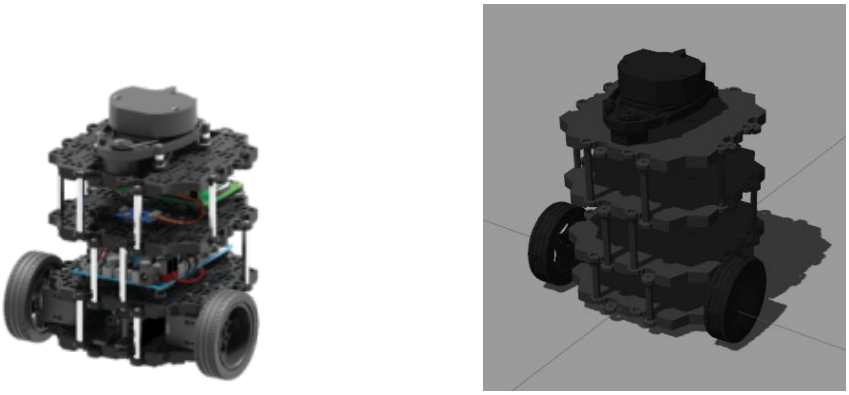


Figure 5.5: TurtleBot3 Burger and respective Gazebo's representation.

TurtleBot3 [65] is an affordable and versatile open-source robot platform designed for education, research, and hobbyist purposes [66]. It is developed by the Open Robotics and part of the ROS ecosystem [67]. The hardware specifications of the robot are described in Table 5.3 which includes a LiDAR sensor detailed in Table 5.4, enabling it to perceive its surrounding environment.

Table 5.3: TurtleBot3 Burger hardware specifications [2].

Size (L x W x H)	138mm x 178mm x 192mm
Maximum translational velocity	0.22 m/s
Maximum rotational velocity	2.84 rad/s
Weight	1kg
Maximum payload	15kg
Laser Distance Sensor	360 Laser Distance Sensor LDS-01

5.5 Hardware Components

This section provides an overview of the key technical specifications of the two computers employed during the development of this work, with particular emphasis on the GPU, due to its

Table 5.4: 360 Laser Distance Sensor LDS-01 specifications [3].

Distance Range	120 ∼ 3,500mm
Distance Accuracy (120mm ∼ 499mm)	±15mm
Distance Accuracy(500mm ∼ 3,500mm)	±5.0%
Distance Precision(120mm ∼ 499mm)	±10mm
Distance Precision(500mm ∼ 3,500mm)	±3.5%
Scan Rate	300±10 rpm
Angular Range	360°
Angular Resolution	1°

impact and importance on DNNs training.

5.5.1 Graphical Processing Unit

Deep Learning (DL) has revolutionized the field of artificial intelligence by enabling the development of complex models capable of learning from large datasets. However, training these models can be computationally demanding and time-intensive. Thanks to their parallel processing capabilities and high performance, GPUs have played a crucial role in accelerating DL tasks significantly.

NVIDIA has significantly contributed to this acceleration with its CUDA framework. Compute Unified Device Architecture (CUDA) [68] is a parallel computing platform and API that allows developers to leverage the computational power of GPUs. It enables tasks to be separated into smaller parallel operations, significantly speeding up computations. To further enhance DL on GPUs, NVIDIA provides the CUDA Deep Neural Network (cuDNN) library [69]. This GPU-accelerated library offers optimized implementations of DNN operations. It is designed to increase the efficiency of training and inference processes in DL models.

Throughout the development of this work, two separate GPUs were utilized: the NVIDIA GeForce RTX 3060 and the NVIDIA GeForce RTX 3080ti. Their specifications are listed in Table 5.5.

Table 5.5: GPUs specifications.

NVIDIA GeForce	**RTX 3060**	**RTX 3080ti**
CUDA Cores	3584	7424
Base Engine Clock	1.320 GHz	1.125 GHz
Boost Engine Clock	1.807 GHz	1.590 GHz
Video Memory	12GB GDDR6	12GB GDDR6
Memory Clock	7.5 GHz	1.875 GHz
Memory Bus	192-bit	256-bit
Memory Bus Bandwidth	360 GB/s	512 GB/s

5.5.2 Central Processing Unit and Random-Access Memory

The CPUs employed in this work were the AMD Ryzen 7 3800X and the Intel Core i7-12700H, specified in Table 5.6. Additionally, both computer systems were configured with 32GB of memory.

Table 5.6: CPUs specifications.

	AMD Ryzen 7 3800X	**Intel Core i7-12700H**
Cores	8	14
Threads	16	20
Boost Clock	4.5 GHz	4.7 GHz

5.6 Python and VSCode

For the development of the software algorithm used in this project, Python was selected as the programming language due to its simplicity, and robust ecosystem of libraries and frameworks tailored for ML and DL. The libraries utilized to support the implementation of the proposed framework are outlined in Table 5.7.

Table 5.7: Python libraries.

Library	**Version**	**Description**
PyTorch [70]	2.0.0	Deep Learning library
Rospy [71]	1.16.0	Client library for ROS that enables Python to interface with ROS components
NumPy [72]	1.24.1	Numerical computation library
Matplotlib [73]	3.1.2	Graphical data visualization library
Pandas [74]	1.5.2	Data analysis, time series, and statistics library
Seaborn [75]	0.12.2	Statistical data visualization library

The integrated development environment (IDE) in which the algorithm was developed was Visual Studio Code. VSCode [76] is a popular and widely-used code editor developed by Microsoft. It is known for its lightweight yet powerful features, extensibility, and support for a wide range of programming languages, including Python.

5.6.1 Pytorch

PyTorch [70, 77] is an open-source DL framework developed by Facebook's artificial intelligence research group. It provides a Python-based scientific computing package that allows users to perform machine learning tasks such as DNN development, training, and deployment. It is known for its dynamic computational graph system, which allows for easier and faster model development and training. It supports both CPU and GPU computation and can run on a wide range of hardware, from laptops to clusters of GPUs. It also provides an active community of developers who contribute to its development and provide support.

PyTorch served as a powerful tool for implementing and training deep learning models in the proposed framework. Its dynamic nature, user-friendly design, and strong community support were instrumental in achieving the project's goals.

6

Results and Discussion

This chapter presents the results of testing and validating the developed DeepRL-based framework for motion planning in static and dynamic simulation environments.

6.1 Motion Planning in Environments with Static Obstacles

This section provides an overview of the validation process for the developed motion planning strategy in environments populated by static obstacles. It also discusses the evaluation of various techniques, including prioritized experience replay, reward propagation, and curriculum learning, and compares different Deep Q-Learning variants. The evaluation within static environments was performed in Gazebo's *Stage 4* environment, described in Section 5.3 of Chapter 5.

6.1.1 Scenario 1

With the proposed DeepRL framework implemented and employing the D3QN and the hyperparameters detailed in Table 6.1, the agent was trained in Gazebo's *Stage 4* environment, with a simple scenario - Scenario 1 - illustrated in Fig. 6.1. In this Scenario, the agent's goal was to navigate from its initial position (-1.0, 0.0) to a single target point (1.0, 0.0), while avoiding collision with any obstacle.

Table 6.1: Simulation parameters utilized in Scenario 1.

Parameter	**Value**	**Parameter**	**Value**
Number of episodes (M)	2000	Discount Factor (γ)	0.99
Buffer Size (N)	200000	Immediate penalty (P_O)	-1.5
Batch Size (N_b)	256	Immediate reward (R_D)	2.0
Exploration threshold (ϵ)	0.01∼1.0	Distance to target threshold ($d_{T_{min}}$)	0.1
Exploration decay (ϵ_{decay})	1.25/M	Distance to obstacle threshold ($d_{O_{min}}$)	0.12
Number of Steps (T)	500	Reward propagation window (P_w)	N/A
Learning Rate (lr)	0.0001	Target Net Update rate (U_r)	10

As depicted in Fig. 6.2a, the agent reached the target with consistency in the last 200 training episodes. Figure 6.2c further illustrates that the paths more often taken by the robot

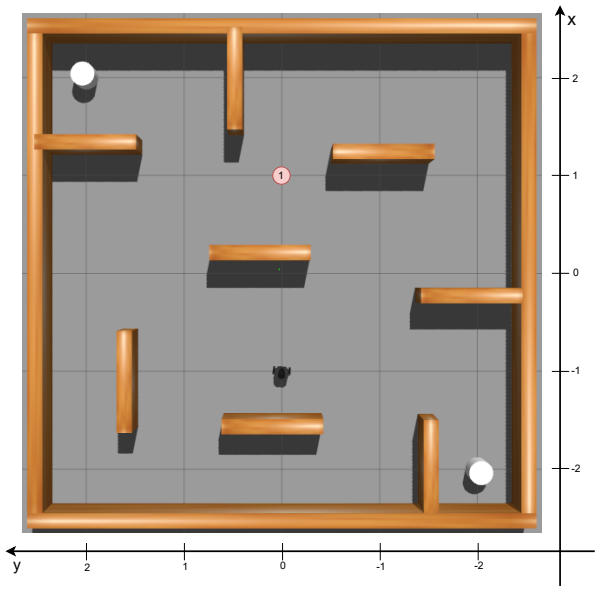
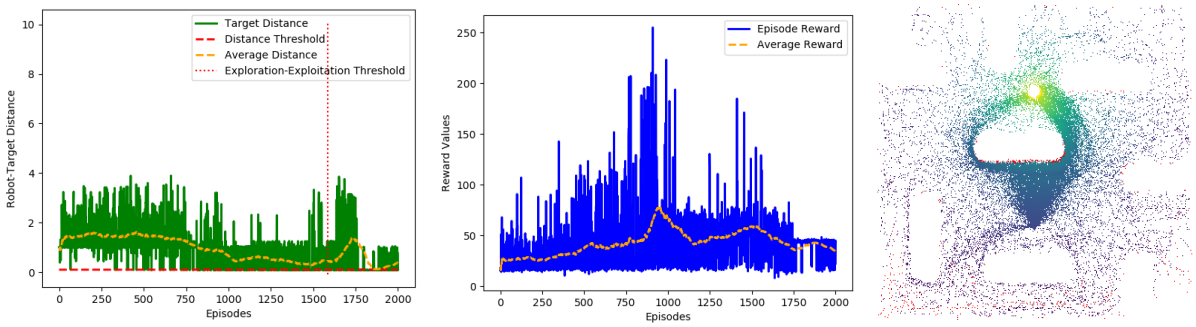


Figure 6.1: Gazebo's *Stage 4* environment Scenario 1. The agent's initial position is located at the turtlebot's location, and the target point is represented by the red circle.



(a) Distances. **(b)** Rewards. **(c)** Paths.

Figure 6.2: Training results in Gazebo's *Stage 4* Scenario 1. (a) shows the distances between the robot and the target and the end of each episode. In (b) is illustrated the episodic rewards, and in (c) is illustrated the paths taken by the robot, with the reward on each step represented by color.

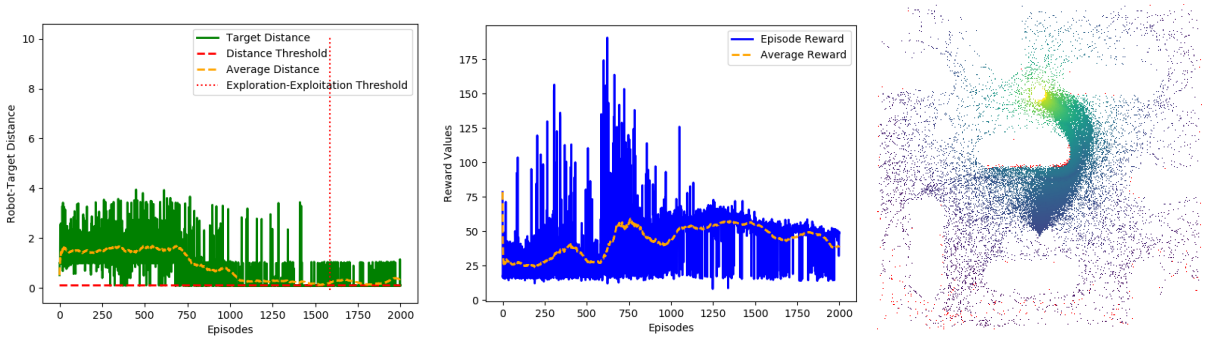
were near-optimal collision-free paths towards the target. As detailed in Table 6.2, the training phase took about 8 hours and resulted in a total of 773 saved network models. The accuracy of the test model was assessed based on the number of times that the agent successfully reached the target, in 25 tests. The outcome indicated a success rate of 40%, meaning that the agent managed to reach the target in 10 out of 25 attempts.

Table 6.2: Training and testing details in Scenario 1.

Training Duration	Training Results	Saved Models	Test Model	Accuracy
7h44min	Fig. 6.2	773	2000/2000	0.4

6.1.1.1 Prioritized Experience Replay

With the base DeepRL framework validated on Scenario 1, the first iteration was to evaluate the employment of the PER, explained in Section 4.3 (Chapter 4). With the PER implemented, the agent was once again trained in Gazebo's *Stage 4*, Scenario 1. The training results are shown in Fig. 6.3.



(a) Distances. **(b)** Rewards. **(c)** Paths.

Figure 6.3: Training results in Gazebo's *Stage 4* Scenario 1 with PER.

The integration of the proposed PER method resulted in significantly better results. As demonstrated in Fig. 6.3a, the agent exhibited significantly better consistency in reaching its target. Figure 6.3c indicates that the agent effectively learned the shortest collision-free path towards the target point. As detailed in Table 6.3, the integration of PER led to a more than 30% increase in the number of saved models and improved the agent's success rate in reaching the target by 120%. Given the substantial advantages of PER, all subsequent tests incorporated this technique.

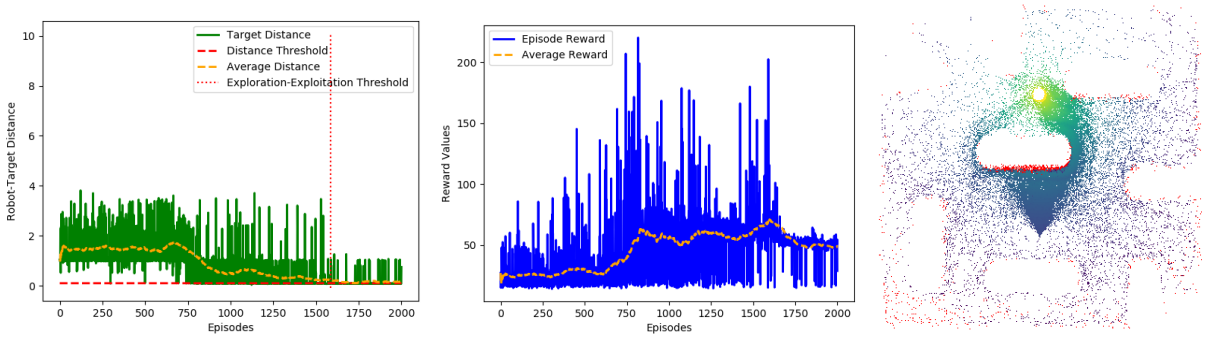
Table 6.3: Training and testing details in Scenario 1 with PER.

Training Duration	Training Results	Saved Models	Test Model	Accuracy
7h25min	Fig. 6.3	1010	1988/2000	0.88

6.1.1.2 Reward Propagation

The second iteration was made to evaluate the reward propagation technique, addressed in Section 4.4.3 of Chapter 4. Thus, upon integrating equation 4.6 in the framework's reward model, the agent was trained using Scenario 1 (see Fig. 6.1 and Table 6.1), with a reward propagation window of 5 steps ($P_w = 5$). The training results are presented in Fig. 6.4.

The effects of the reward propagation technique are evident in the increased density of red dots in Fig. 6.4c. This is due to the fact that whenever the agent collides, the penalty P_O received in that step was propagated to the preceding 5 steps. Figure 6.4a shows further enhancement in the agent's consistency in reaching the target point over the last 500 episodes. This improvement is apparent when comparing the average distance (yellow) line in Fig. 6.4a to that in 6.3a. As detailed in Table 6.4, the integration of reward propagation resulted in a nearly 14% increase in the agent's accuracy in reaching its goal destination, achieving a remarkable



(a) Distances. **(b)** Rewards. **(c)** Paths.

Figure 6.4: Training results in Gazebo's *Stage 4* Scenario 1 with PER and reward propagation.

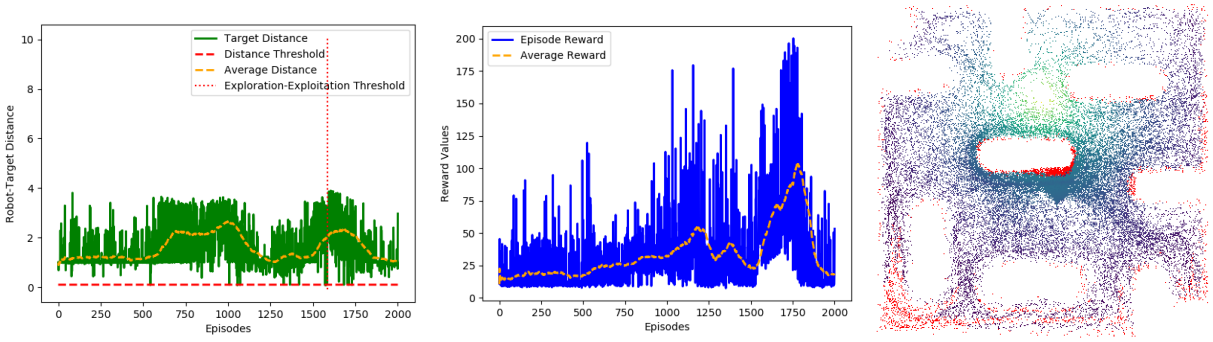
100% accuracy in 25 attempts. Due to the significant benefits of this approach, all subsequent tests included the reward propagation technique in addition to the PER technique.

Table 6.4: Training and testing details in Scenario 1 with PER and reward propagation.

Training Duration	Training Results	Saved Models	Test Model	Accuracy
7h47min	Fig. 6.4	966	1999/2000	1.0

6.1.1.3 Deep Q-Learning Variants

In this section, the DQN variant employed in the proposed DeepRL-based framework is evaluated through a comparison with other variants discussed in Section 2.4 of Chapter 2 in order to assess its advantages. The agent underwent three distinct training phases, differing solely on the DQN method employed: DQN, DDQN, and D2QN. The training results for these three variants are illustrated in Figures 6.5, 6.6, and 6.7, respectively.

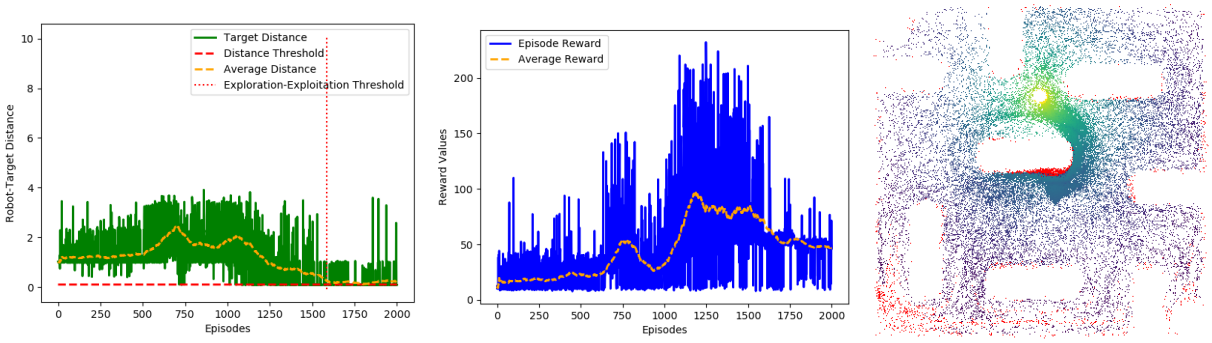


(a) Distances. **(b)** Rewards. **(c)** Paths.

Figure 6.5: Training results in Gazebo's *Stage 4* Scenario 1 with DQN.

DQN, the conventional Deep Q-Learning algorithm, lacks the advantages found in DDQN and D2QN. Consequently, as expected, it exhibited less satisfactory performance during the training process, as illustrated in Fig. 6.5a.

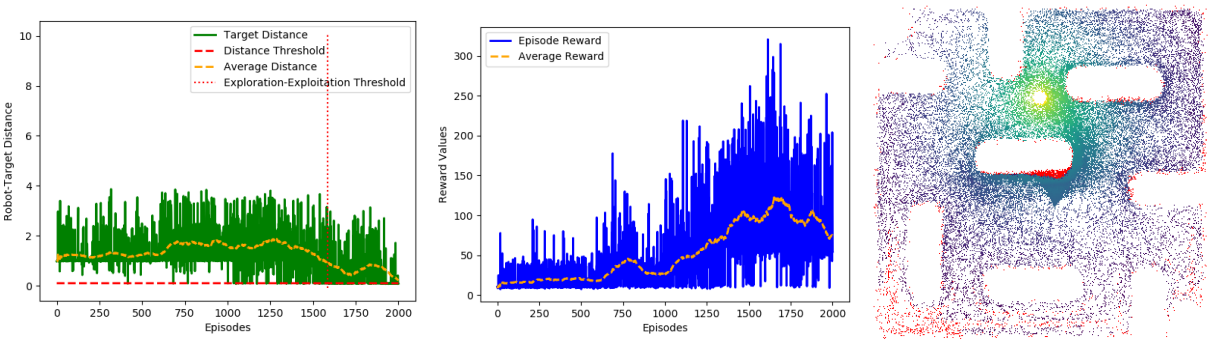
As previously explained in Section 2.4.3 (Chapter 2), the DDQN variant features a two



(a) Distances. **(b)** Rewards. **(c)** Paths.

Figure 6.6: Training results in Gazebo's *Stage 4* Scenario 1 with DDQN.

stream network architecture that allows the agent to independently assess the value of actions in different states. This technique enhances both the stability and learning efficiency of the learning process, as evident in the results displayed in Fig. 6.6.



(a) Distances. **(b)** Rewards. **(c)** Paths.

Figure 6.7: Training results in Gazebo's *Stage 4* Scenario 1 with D2QN.

The D2QN variant introduces the decoupling of the action selection and value estimation steps, as explained in Section 2.4.4 (Chapter 2). This separation contributes to more stable and faster learning by mitigating the issue of *Q-values* overestimation, which is reflected in the results shown in Fig. 6.7.

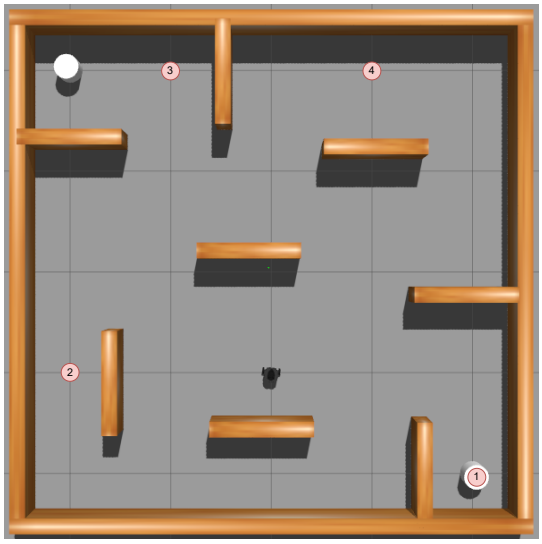
Table 6.5 provides an overview of the training details and testing results for all four Deep Q-Learning approaches. Notably, the D3QN variant emerged as the top performer among the others. It achieved the shortest training duration of 7 hours and 47 minutes, and retained the highest number of saved network models. In the testing phase, it achieved the highest accuracy of 100%, meaning that the agent flawlessly reached the target in all 25 attempts.

Table 6.5: Training and testing details in Scenario 1 with DQN, DDQN, D2QN, and D3QN.

Variant	Training Duration	Training Results	Saved Models	Test Model	Accuracy
DQN	8h06min	Fig. 6.5	14	1710/2000	0.08
DDQN	8h16min	Fig. 6.6	640	1795/2000	0.96
D2QN	9h53min	Fig. 6.7	316	2000/2000	0.92
D3QN	7h47min	Fig. 6.4	966	1999/2000	1.0

6.1.2 Scenario 2

With the developed DeepRL-based framework successfully validated in the relatively simple Scenario 1, the next step was to assess its performance in a more challenging Scenario. For that purpose, Scenario 2 was defined within Gazebo's *Stage 4* environment, featuring four distinct target points scattered throughout the environment, as illustrated in Fig. 6.8. At the beginning of each training episode, one of the four targets is randomly chosen, and the agent initiates its motion at (-1.0, 0.0). With its target destinations being further away, the agent faces more obstacle-avoidance and orientation-amendment situations and, consequently, more state representations to learn from, which ultimately increases the training complexity and duration. Furthermore, this multi-target configuration allows the agent to explore diverse regions of the environment, improving its knowledge of the entire workspace and consequently enhancing its decision-making capabilities. Employing the D3QN along with the PER and reward propagation techniques, and the hyperparameters detailed in Table 6.6, the agent was trained in Gazebo's *Stage 4* environment within Scenario 2. The training results are presented in Fig. 6.9 and detailed in Table 6.7.

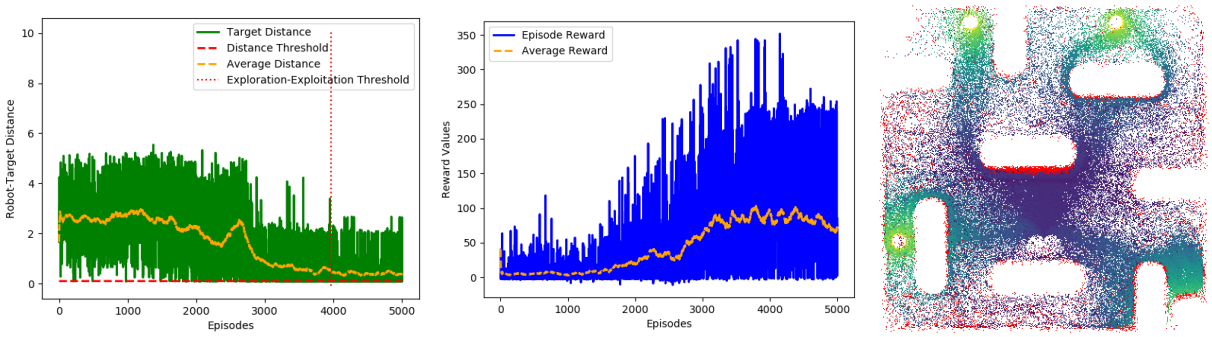


Target	Coordinates
1	(-2.0, -2.0)
2	(-1.0, 2.0)
3	(2.0, 1.0)
4	(2.0, -1.0)

Figure 6.8: Gazebo's *Stage 4* environment Scenario 2. The agent's initial position is located at the turtlebot's location, and the target points are represented by the red circles.

Table 6.6: Simulation parameters utilized in Scenario 2.

Parameter	Value	Parameter	Value
Number of episodes (M)	5000	Discount Factor (γ)	0.99
Buffer Size (N)	200000	Immediate penalty (P_O)	-1.5
Batch Size (N_b)	256	Immediate reward (R_D)	2.0
Exploration threshold (ϵ)	0.01∼1.0	Distance to target threshold ($d_{T_{min}}$)	0.1
Exploration decay (ϵ_{decay})	1.25/M	Distance to obstacle threshold ($d_{O_{min}}$)	0.12
Number of Steps (T)	500	Reward propagation window (P_w)	5
Learning Rate (lr)	0.0001	Target Net Update rate (U_r)	10



(a) Distances. **(b)** Rewards. **(c)** Paths.

Figure 6.9: Training results in Gazebo's *Stage 4* Scenario 2.

Figure 6.9a indicates a promising learning curve and shows that the agent consistently reached the target in the last 1000 training episodes. Figure 6.9c showcases the agent's ability to learn optimal paths towards each of the four targets. As expected, the training duration increased significantly, primarily due to the larger number of the training episodes and the increased complexity of Scenario 2. Nonetheless, the number of saved network models was similar to Scenario 1 (see Table 6.4). In the testing stage, detailed in Table 6.8, the agent demonstrated its proficiency by reaching the targets with an overall accuracy of 77%.

Table 6.7: Training details in Gazebo's *Stage 4* Scenario 2.

Training Duration	Training Results	Saved Models	Test Model
27h18min	Fig. 6.9	1163	4993/5000

Table 6.8: Testing details in Gazebo's *Stage 4* Scenario 2.

Target	Tests	Accuracy
1	32	0.66
2	29	0.90
3	17	0.65
4	23	0.87
Overall	100	**0.77**

6.1.3 Curriculum Learning

The curriculum learning technique discussed in Section 2.4.6 of Chapter 2 was evaluated using the 5 distinct environments described in Section 4.5.2 of Chapter 4. Figure 6.10 illustrates the initial and target points, represented by green and red dots, respectively, for each environment.

Employing the parameters specified in Table 6.9, the agent was consecutively trained within all five environments, starting on the simplest (a) and finishing on the most complex (e). In each transition between environments, the last saved network's parameters from the preceding training phase are employed to initialize the target and policy networks for training in the

subsequent environment. Furthermore, after training in the initial environment (a), the agent's exploration threshold is halved ($\epsilon = 0.01 \sim 0.5$) and its learning rate is decreased by 10 times ($lr = 0.00001$). These adjustments are implemented to encourage exploitation, leveraging the knowledge acquired in the previous environments. The training results are shown in Fig. 6.11.

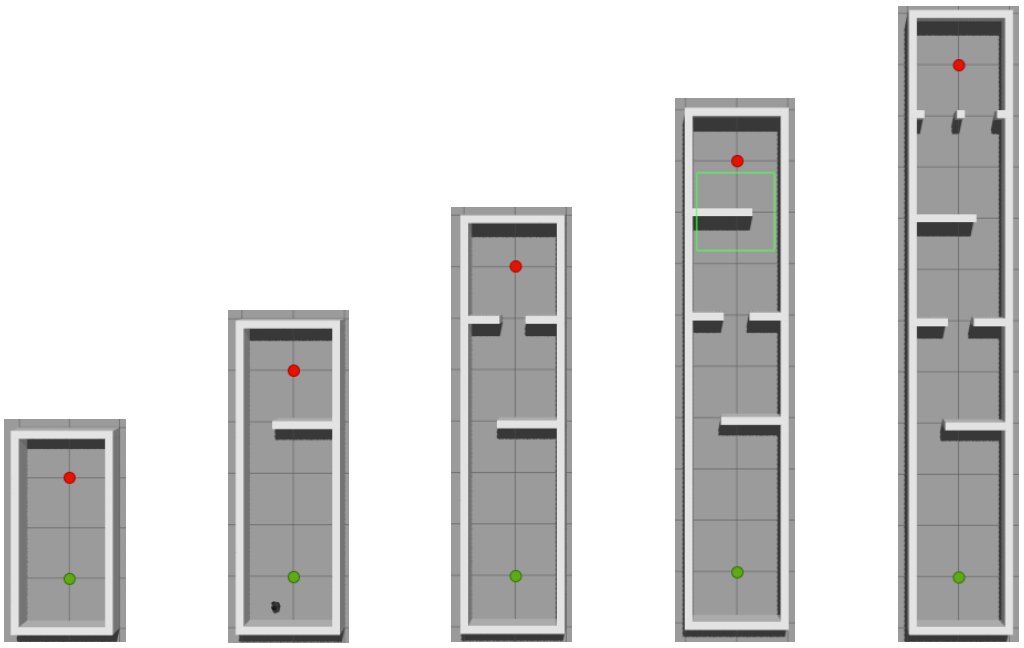


Figure 6.10: Gazebo's curriculum environments.

Table 6.9: Simulation parameters utilized in curriculum environments.

Parameter	Value	Parameter	Value
Number of episodes (M)	2000 (1)	Discount Factor (γ)	0.99
Buffer Size (N)	200000	Immediate penalty (P_O)	-1.5
Batch Size (N_b)	256	Immediate reward (R_D)	2.0
Exploration threshold (ϵ)	0.01∼0.5 (2)	Distance to target threshold ($d_{T_{min}}$)	0.1
Exploration decay (ϵ_{decay})	1.25/M	Distance to obstacle threshold ($d_{O_{min}}$)	0.12
Number of Steps (T)	500 (3)	Reward propagation window (P_w)	5
Learning Rate (lr)	0.00001 (4)	Target Net Update rate (U_r)	10

(1) The number of episodes was set to 2000 for all environments except (a), where it was set to 1000.
(2) The exploration threshold was set to 0.5 for all environments except (a), where it was set to 1.0.
(3) The number of max steps was set to 500 for all environments except (e), where it was set to 600.
(4) The learning rate was set to 0.00001 for all environments except (a), where it was set to 0.0001.

The results depicted in Fig. 6.11 show, as expected, that the agent's consistency in reaching the target point decreased as the environment's complexity increased. Nonetheless, the results were satisfactory, and the trend observed in the average robot-to-target distance suggests that by increasing the number of training episodes, the agent's performance is likely to improve. As detailed in Table 6.10, the agent exhibited proficiency in reaching the target in all environments, attaining a success rate of 100% in the simplest one (a) and 44% on the most intricate one (e).

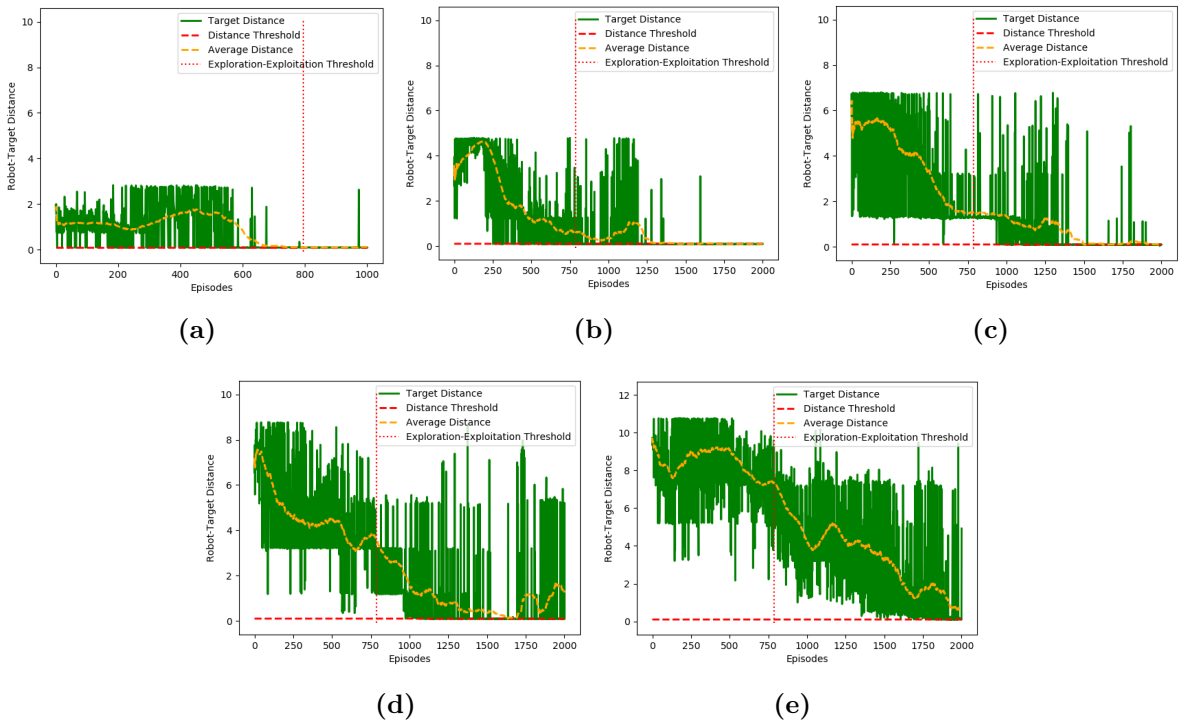


Figure 6.11: Results of training with knowledge transfer in the curriculum environments (see Fig. 6.10).

To further assess the benefits of this technique, the agent was trained from scratch on all environments, using the same parameters (Table 6.9). To ensure a fair comparison, the number of episodes for each training was increased to 3000, 5000, 7000, and 9000, for environments (b), (c), (d), and (e), respectively. The training results are depicted in Fig. 6.12 and detailed in Table 6.11.

The comparison between the results presented in Fig. 6.11 and those in Fig. 6.12 clearly proves the advantage of the curriculum learning technique employed. When trained from scratch in each environment, the agent's learning curves exhibited significantly greater instability, and its consistency in reaching the target was notably inferior. In the testing phase, the accuracy of the agent trained from scratch (see Table 6.11) was considerably lower than the accuracy of the agent trained with knowledge transfer (see Table 6.10), mainly in the two most complex environments ((d) and (e)).

Table 6.10: Training and testing details with knowledge transfer.

Environment	Training Duration	Training Results	Saved Models	Test Model	Accuracy
(a)	4h11min	Fig. 6.11a	570	989/1000	1.0
(b)	13h48min	Fig. 6.11b	1330	2000/2000	1.0
(c)	15h01min	Fig. 6.11c	829	1700/2000	0.84
(d)	19h40min	Fig. 6.11d	783	1650/2000	0.96
(e)	26h30min	Fig. 6.11e	66	1972/2000	0.44

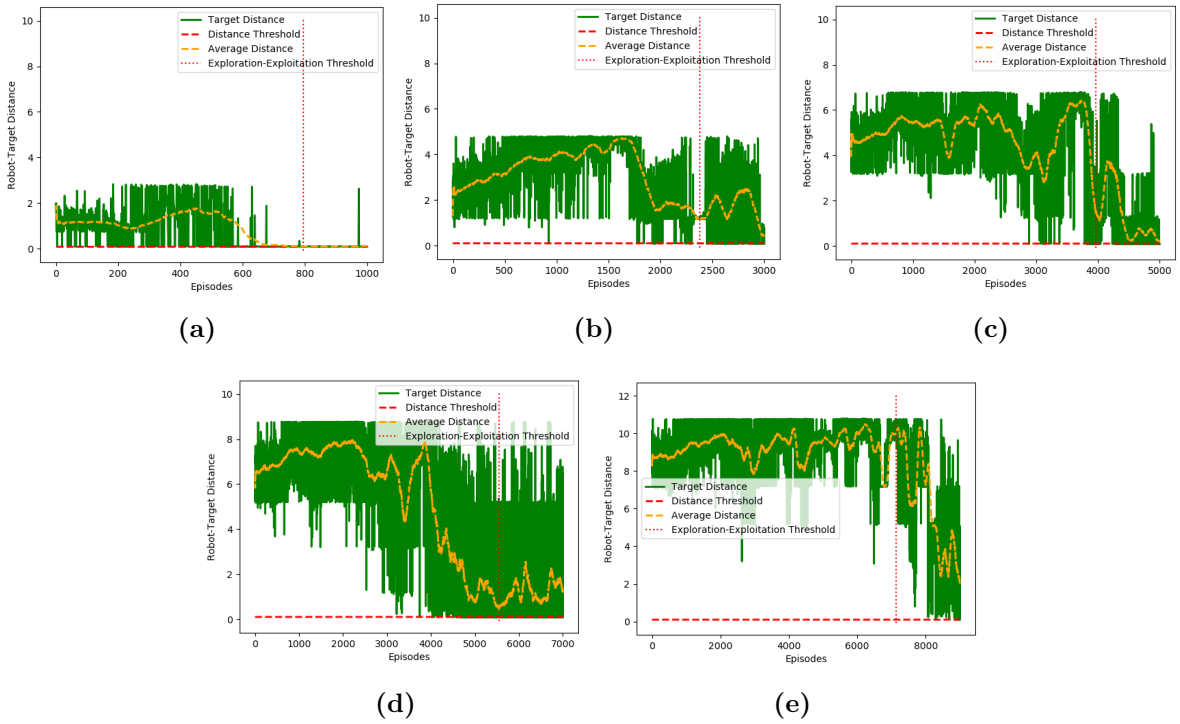


Figure 6.12: Results of training from scratch in the curriculum environments (see Fig. 6.10).

Table 6.11: Training and testing details, without knowledge transfer (training from scratch).

Environment	Training Duration	Training Results	Saved Models	Test Model	Accuracy
(a)	4h11min	Fig. 6.12a	570	989/1000	1.0
(b)	12h39min	Fig. 6.12b	303	3000/3000	0.88
(c)	28h39min	Fig. 6.12c	630	5000/5000	0.84
(d)	47h39min	Fig. 6.12d	1607	5577/7000	0.24
(e)	38h14min	Fig. 6.12e	5	9000/9000	0.12

6.2 Motion Planning in Environments with Dynamic Obstacles

The ultimate goal of this work was to evaluate the performance of the developed framework in dynamic environments. To achieve this, the agent was trained in the Gazebo's *Stage 4 Dynamic* environment depicted in Fig. 6.13 and addressed in Section 4.5.1 of Chapter 4 within the Scenario 2 described in Section 6.1.2. Due to the advantages of transfer learning verified in Section 6.1.3, this technique was employed in the training process. The networks were initialized with the parameters from the network model previously obtained in the static Gazebo's *Stage 4* environment (see Table 6.7). The training results are presented in Fig. 6.14 and further detailed in Table 6.12.

Figure 6.14a shows that the agent learned how to efficiently navigate towards the target points. The paths taken by the robot (see Fig. 6.14c) reveal that the agent avoided the non-optimal paths and exploited the near-optimal ones previously learned in the static environment, while learning how to behave towards the new dynamic obstacles. This demonstrates the transfer of knowledge acquired in the static environment (see Fig. 6.8) to the dynamic environment.

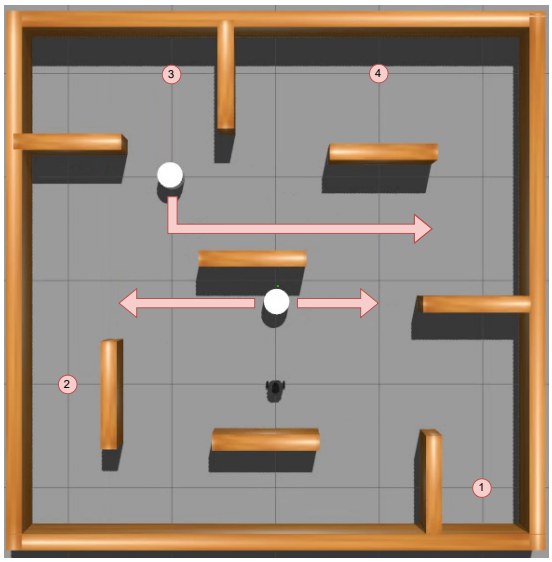
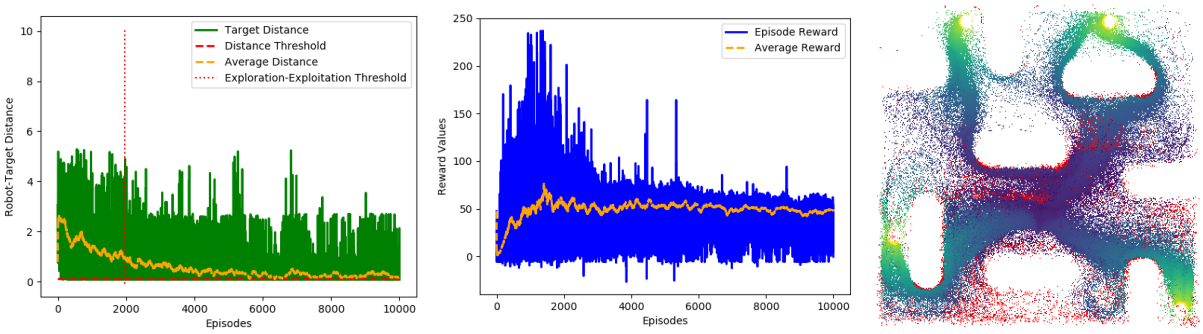


Figure 6.13: Gazebo's *Stage 4 Dynamic* environment.



(a) Distances. **(b)** Rewards. **(c)** Paths.

Figure 6.14: Training results in Gazebo's *Stage 4 Dynamic* Scenario 2 with knowledge transfer.

During the testing stage, detailed in Table 6.13, the agent showcased proficient motion planning abilities by reaching the targets with an overall accuracy of 84% in 100 attempts.

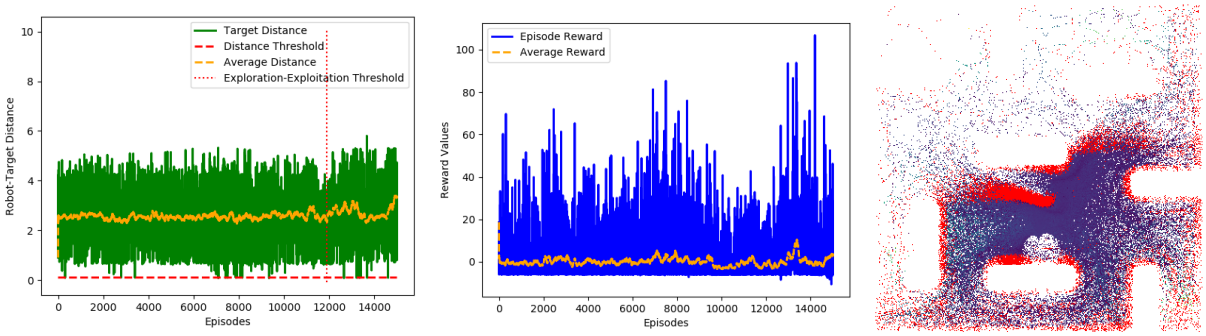
Table 6.12: Training details in Gazebo's *Stage 4 Dynamic* Scenario 2 with knowledge transfer.

Training Duration	Training Results	Saved Models	Test Model
45h30min	Fig. 6.14	7293	10000/10000

Table 6.13: Testing details in Gazebo's *Stage 4 Dynamic* Scenario 2 with knowledge transfer.

Target	Tests	Accuracy
1	25	1.0
2	28	0.82
3	23	0.78
4	24	0.75
Overall	100	**0.84**

To verify the advantages of knowledge transfer from the static environment to this dynamic one, the agent was trained under the same conditions but from scratch. Figure 6.15 and Tables 6.14 and 6.15 present the results obtained.



(a) Distances. **(b)** Rewards. **(c)** Paths.

Figure 6.15: Training results in Gazebo's *Stage 4 Dynamic* Scenario 2 without knowledge transfer (training from scratch).

Table 6.14: Training details in Gazebo's *Stage 4 Dynamic* Scenario 2 without knowledge transfer.

Training Duration	Training Results	Saved Models	Test Model
34h30min	Fig. 6.15	23	13407/15000

Table 6.15: Testing details in Gazebo's *Stage 4 Dynamic* Scenario 2 without knowledge transfer.

Target	Tests	Accuracy
1	25	0.0
2	28	0.04
3	23	0.0
4	24	0.07
Overall	100	**0.03**

Upon comparing the results depicted in Fig. 6.14 and those in Fig. 6.15, the benefits of transfer learning become evident. When trained from scratch in the *Stage 4 Dynamic* environment, the agent struggled to learn how to execute a collision-free motion towards any of the targets, as it only reached a target in 23 out of 15000 training episodes (see Table 6.14). Consequently, the poor performance in the training stage led to unfavorable results in the testing stage, as detailed in Table 6.15. During testing, the agent showcased an accuracy of only 3% in stark contrast to the 84% accuracy achieved with knowledge transfer (see Table 6.13).

7

Conclusion

This dissertation proposes an approach for robotic local motion planning within unknown indoor environments populated by both static and dynamic obstacles. The approach is based on DeepRL techniques, particularly leveraging the Deep Q-Learning algorithm. The system was developed to address the need for adaptable strategies in DeepRL-based dynamic indoor robot navigation for sensor-equipped mobile platforms. The ultimate aim was to establish a sensor-agnostic local motion planning method capable of seamless transferability across different robotic systems. A comprehensive survey of robot motion planning, RL, DL, and DeepRL, was meticulously conducted to provide the foundational conceptual support for the developed framework.

The development process began with the design of the D3QN and state, action, and reward models, drawing inspiration from the prior work [1] and meticulously adapted to align with the specific objectives of this work. The defined state model incorporated a costmap representation of the robot's surroundings, along with a goal-based affordance that includes the robot's distance and orientation towards the target, as well as its proximity to the nearest obstacle. To comply with the defined state model and properly convert its representations into *Q-values*, the D3QN was devised to employ a dual-stream architecture. Over the course of development, several reward models and action sets were iteratively conceived and assessed until the desired level of performance was achieved. Additionally, the proposed PER and reward propagation techniques were implemented, and dedicated environments were created to facilitate the evaluation of the curriculum learning methodology in later stages.

Upon completing the design of each component, the motion planning algorithm proposed in this work was ready for deployment on a virtual robot - the Turtlebot - to execute its action-selection policy in a virtual environment. Initial experiments took place in a Gazebo environment populated by static obstacles to validate the developed DeepRL-based local motion planning approach. The PER and reward propagation methods revealed advantageous, leading to their incorporation into the system onwards. Additionally, a comparative assessment between the proposed D3QN variant and other variants was conducted, highlighting the advantages of the D3QN. Furthermore, experiments performed within the curriculum environments, employing transfer learning techniques, demonstrated the effectiveness of the proposed curriculum learning technique. The RL agent effectively leveraged knowledge acquired in previous simpler tasks to efficiently tackle more complex ones. Lastly, having successfully validated the framework

the in static environments and considering the observed benefits of the appliance of curriculum learning techniques in the networks' training, the framework was validated in a dynamic environment. The evaluation yielded outstanding results, as the agent was able to learn how to efficiently behave towards dynamic obstacles, demonstrating the successful accomplishment of the dissertation's main goal.

7.1 Future Work

Although the mobile robot navigation framework proposed and developed in this work has yielded promising results, there is still room for improvement. This section briefly presents some potential areas for enhancement.

Enhanced Training

Incorporate an augmentation to the training regimen by substantially increasing the number of episodes per session and simultaneously engaging multiple robots in contributing to the network model's training. Further evaluation of the RL agent's capacity to generalize its learned skills.

3D Representation

Develop and implement a 3D representation of the robot's surrounding environment to empower it to execute more complex behaviors, such as docking.

Global Context

Exploit existing data from global metric or semantic representations to enhance the RL agent's ability to effectively navigate dynamic environments.

Geometry Awareness

Take into consideration the robot's platform geometry to enable learning orientation adjustments, allowing it to navigate through narrow passages like as door frames.

Real-World Validation

Validate the framework in real mobile robots and environments using simulation-to-reality transfer techniques.

Bibliography

[1] Daniel Palaio. DeepRL-based motion planning for indoor robot navigation. Master's thesis, University of Coimbra, 2021.

[2] Turtlebot3 hardware specifications. `https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#specifications`.

[3] Turtlebot3 hardware specifications. `https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/`.

[4] James P. Trevelyan, Sung-Chul Kang, and William R. Hamel. *Robotics in Hazardous Applications*, pages 1101–1126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[5] Francesco Semeraro, Alexander Griffiths, and Angelo Cangelosi. Human–robot collaboration and machine learning: A systematic review of recent research. *Robotics and Computer-Integrated Manufacturing*, 79:102432, 2023.

[6] Mohsen Soori, Behrooz Arezoo, and Roza Dastres. Artificial intelligence, machine learning and deep learning in advanced robotics, a review. *Cognitive Robotics*, 3:54–70, 2023.

[7] Carlos A. García-Pintos, Noé G. Aldana-Murillo, Emmanuel Ovalle-Magallanes, and Edgar Martínez. A deep learning-based visual map generation for mobile robot navigation. *Eng*, 4(2):1616–1634, 2023.

[8] Luís Garrote, Diogo Temporão, Samuel Temporão, Ricardo Pereira, Tiago Barros, and Urbano J. Nunes. Improving local motion planning with a reinforcement learning approach. In *2020 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2020.

[9] Luís Garrote, João Paulo, João Perdiz, Paulo Peixoto, and Urbano J. Nunes. Robot-assisted navigation for a robotic walker with aided user intent. In *2018 27th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, 2018.

[10] Luís Garrote, João Paulo, and Urbano J. Nunes. Reinforcement learning aided robot-assisted navigation: A utility and RRT two-stage approach. *International Journal of Social Robotics*, 12, 07 2020.

[11] Luís Garrote, João Perdiz, Gabriel Pires, and Urbano J. Nunes. Reinforcement learning motion planning for an EOG-centered robot assisted navigation in a virtual environment. In

2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN), 2019.

[12] Laith Alzubaidi, Jinglan Zhang, Amjad J. Humaidi, Ayad Q. Al-Dujaili, Ye Duan, Omran Al-Shamma, Jesus Santamaría, Mohammed Abdulraheem Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: concepts, cnn architectures, challenges, applications, future directions. *Journal of Big Data*, 8, 2021.

[13] Iqbal Sarker. Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions. *SN Computer Science*, 2, 08 2021.

[14] Luís Garrote, João Perdiz, and Urbano J. Nunes. Costmap-Based Local Motion Planning Using Deep Reinforcement Learning. In *2023 32nd IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*. IEEE, 2023.

[15] Binyu Wang, Zhe Liu, Qingbiao Li, and Amanda Prorok. Mobile robot path planning in dynamic environments through globally guided reinforcement learning. *IEEE Robotics and Automation Letters*, 5(4):6932–6939, 2020.

[16] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[18] Chengmin Zhou, Bingding Huang, and Pasi Fränti. A review of motion planning algorithms for intelligent robots. *Journal of Intelligent Manufacturing*, 33, 11 2021.

[19] Yuncheng Lu, Xue Zhucun, Gui-Song Xia, and Liangpei Zhang. A survey on vision-based uav navigation. *Geo-spatial Information Science*, pages 1–12, 01 2018.

[20] Zvi Shiller. *Off-Line and On-Line Trajectory Planning*, volume 29, pages 29–62. 02 2015.

[21] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[22] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 1996.

[23] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, PP:1–1, 09 2019.

[24] Su-Hyun Han, Ko Kim, SangYun Kim, and Young Chul Youn. Artificial neural network: Understanding the basic concepts without mathematics. *Dementia and Neurocognitive Disorders*, 17:83, 09 2018.

[25] Yingjie Tian, Yuqi Zhang, and Haibin Zhang. Recent advances in stochastic gradient descent in deep learning. *Mathematics*, 11(3), 2023.

[26] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations (ICLR)*, 12 2014.

[27] Xue Ying. An overview of overfitting and its solutions. *Journal of Physics: Conference Series*, 1168(2):022022, feb 2019.

[28] Sakshi Indolia, Anil Kumar Goswami, S.P. Mishra, and Pooja Asopa. Conceptual understanding of convolutional neural network- a deep learning approach. *Procedia Computer Science*, 132:679–688, 2018. International Conference on Computational Intelligence and Data Science.

[29] Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent advances in recurrent neural networks. 12 2017.

[30] Mohammadreza Iman, Hamid Reza Arabnia, and Khaled Rasheed. A review of deep transfer learning and recent advancements. *Technologies*, 11(2), 2023.

[31] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[32] Sajad Mousavi, Michael Schukat, and Enda Howley. Deep reinforcement learning: An overview. In *SAI Intelligent Systems Conference*, 06 2018.

[33] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 3061–3071. PMLR, 13–18 Jul 2020.

[34] Ziyu Wang, Tom Schaul, Matteo Hessel, H. V. Hasselt, Marc Lanctot, and N. D. Freitas. Dueling network architectures for deep reinforcement learning. *ArXiv*, abs/1511.06581, 2016.

[35] H. V. Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with Double Q-Learning. In *Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, 2016.

[36] Petru Soviany, Radu Tudor Ionescu, Paolo Rota, and Nicu Sebe. Curriculum learning: A survey. *International Journal of Computer Vision*, abs/2101.10382, 2021.

[37] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *CoRR*, abs/2003.04960, 2020.

[38] Syed Abdullah Fadzli, Sani Iyal Abdulkadir, Mokhairi Makhtar, and Azrul Amri Jamal. Robotic indoor path planning using dijkstra's algorithm with multi-layer dictionaries. In *2015 2nd International Conference on Information Science and Security (ICISS)*, pages 1–4, 2015.

[39] Akshay Kumar Guruji, Himansh Agarwal, and D.K. Parsediya. Time-efficient a* algorithm for robot path planning. *Procedia Technology*, 23:144–149, 2016. 3rd International Conference on Innovations in Automation and Mechatronics Engineering 2016, ICIAME 2016 05-06 February, 2016.

[40] Mohamed Elbanhawi and Milan Simic. Sampling-based robot motion planning: A review. *IEEE Access*, 2:56–77, 2014.

[41] Lijun Qiao, Xiao Luo, and Qingsheng Luo. An optimized probabilistic roadmap algorithm for path planning of mobile robots in complex environments with narrow channels. *Sensors*, 22(22), 2022.

[42] Luís Garrote, Cristiano Premebida, Marco Silva, and Urbano Nunes. An RRT-based navigation approach for mobile robots and automated vehicles. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 2014.

[43] Sabudin elia nadira, Rosli Omar, and Che Ku Nor Hailma. Potential field methods and their inherent approaches for path planning. *ARPN Journal of Engineering and Applied Sciences*, 11:10801–10805, 01 2016.

[44] Marija Seder and Ivan Petrovic. Dynamic window based approach to mobile robot motion control in the presence of moving obstacles. In *Robotics and Automation, 2007 IEEE International*, 05 2007.

[45] Laiyi Yang, Jing Bi, and Haitao Yuan. Dynamic path planning for mobile robots with deep reinforcement learning. *IFAC-PapersOnLine*, 55(11):19–24, 2022. IFAC Workshop on Control for Smart Cities CSC 2022.

[46] Hartmut Surmann, Christian Jestel, Robin Marchel, Franziska Musberg, Houssem Elhadj, and Mahbube Ardani. Deep reinforcement learning for real autonomous mobile robot navigation in indoor environments. *ArXiv*, abs/2005.13857, 2020.

[47] Woohyeon Moon, Bumgeun Park, Sarvar Hussain Nengroo, Taeyoung Kim, and Dongsoo Har. Path planning of cleaning robot with reinforcement learning, 2022.

[48] Yuansheng Dong and Xingjie Zou. Mobile robot path planning based on improved ddpg reinforcement learning algorithm. In *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, 2020.

[49] Hanying Sang and Shuquan Wang. Motion planning of space robot obstacle avoidance based on ddpg algorithm. In *2022 International Conference on Service Robotics (ICoSR)*, 2022.

[50] Richard Meyes, Hasan Tercan, Simon Roggendorf, Thomas Thiele, Christian Büscher, Markus Obdenbusch, Christian Brecher, Sabina Jeschke, and Tobias Meisen. Motion planning for industrial robots using reinforcement learning. *Procedia CIRP*, 63:107–112, 2017. Manufacturing Systems 4.0 – Proceedings of the 50th CIRP Conference on Manufacturing Systems.

[51] Yuepeng Hu, Lehan Yang, and Yizhu Lou. Path planning with q-learning. *Journal of Physics: Conference Series*, 1948(1):012038, jun 2021.

[52] Koray Ozdemir and Adem Tuncer. Deep reinforcement learning based mobile robot navigation in unknown indoor environments. In *International Conference on Interdisciplinary Applications of Artificial Intelligence (ICIDAAI)*, 05 2021.

[53] Xiaoyun Lei, Zhian Zhang, and Peifang Dong. Dynamic path planning of unknown environment based on deep reinforcement learning. *Journal of Robotics*, 2018:1–10, 09 2018.

[54] Yang Wang, Yilin Fang, Ping Lou, Junwei Yan, and Nianyun Liu. Deep reinforcement learning based path planning for mobile robot in unknown environment. *Journal of Physics: Conference Series*, 1576(1):012009, jun 2020.

[55] Gonçalo Leão, Filipe Almeida, Emanuel Trigo, Henrique Ferreira, Armando Sousa, and Luís Reis. Using deep reinforcement learning for navigation in simulated hallways. In *2023 IEEE International Conference on Autonomous Robot Systems and Competitions*, 04 2023.

[56] Miguel Quinones-Ramirez, Jorge Ríos-Martínez, and Víctor Uc Cetina. Robot path planning using deep reinforcement learning. *ArXiv*, abs/2302.09120, 2023.

[57] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *International Conference on Learning Representations (ICLR)*, abs/1511.05952, 2016.

[58] Mark G Sobell. *A practical guide to Ubuntu Linux*. Pearson Education, 2015.

[59] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, volume 3, 01 2009.

[60] HyeongRyeol Kam, Sung-Ho Lee, Taejung Park, and Chang-Hun Kim. Rviz: a toolkit for real domain data visualization. *Telecommunication Systems*, 60:1–9, 10 2015.

[61] Shivraj Narayan Yeole. Simulation in robotics. In *National Conference on Recent Advances in Manufacturing Engineering Technology (RAMET 2011)*, 01 2011.

[62] Gazebo. `https://gazebosim.org/`.

[63] 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, 2004.

[64] Interface for using ros with the gazebo simulator. `http://wiki.ros.org/gazebo_ros_pkgs/`.

[65] Turtlebot3 platform. `https://www.turtlebot.com/turtlebot3/`.

[66] Robin Amsters and Peter Slaets. *Turtlebot 3 as a Robotics Education Platform*, pages 170–181. 01 2020.

[67] Turtlebot3 ros package. `http://wiki.ros.org/turtlebot3`.

[68] Cuda. `https://developer.nvidia.com/cuda-toolkit`.

[69] cudnn. `https://developer.nvidia.com/cudnn`.

[70] Pytorch. `https://pytorch.org/`.

[71] Rospy. `http://wiki.ros.org/rospy`.

[72] Numpy. `https://numpy.org/`.

[73] Matplotlib. `https://matplotlib.org/`.

[74] Pandas. `https://pandas.pydata.org/`.

[75] Seaborn. `https://seaborn.pydata.org/`.

[76] Vscode. `https://code.visualstudio.com/`.

[77] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.*, 2019.