REPUBLIC OF TÜRKİYE
YILDIZ TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

DYNAMIC OBSTACLE AVOIDANCE USING REINFORCEMENT LEARNING

Muharrem KÜÇÜKYILMAZ

MASTER OF SCIENCE THESIS
Department of Computer Engineering
Program of Computer Engineering

Supervisor
Asst. Prof. Dr. Erkan USLU

March, 2024

REPUBLIC OF TÜRKİYE
YILDIZ TECHNICAL UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

DYNAMIC OBSTACLE AVOIDANCE USING REINFORCEMENT LEARNING

A thesis submitted by Muharrem KÜÇÜKYILMAZ in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE** is approved by the committee on 28.03.2024 in Department of Computer Engineering, Program of Computer Engineering.

Asst. Prof. Dr. Erkan USLU
Yildiz Technical University
Supervisor

Approved By the Examining Committee

Asst. Prof. Dr. Erkan USLU, Supervisor
Yildiz Technical University _____

Prof. Dr. Sırma ÇEKİRDEK YAVUZ, Member
Yildiz Technical University _____

Asst. Prof. Dr. Nihal ALTUNTAŞ, Member
Istanbul Gelisim University _____

I hereby declare that I have obtained the required legal permissions during data collection and exploitation procedures, that I have made the in-text citations and cited the references properly, that I haven't falsified and/or fabricated research data and results of the study and that I have abided by the principles of the scientific research and ethics during my Thesis Study under the title of Dynamic Obstacle Avoidance Using Reinforcement Learning supervised by my supervisor, Asst. Prof. Dr. Erkan USLU. In the case of a discovery of false statement, I am to acknowledge any legal consequence.

Muharrem KÜÇÜKYILMAZ

Signature

Dedicated to my family

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to several individuals who have contributed to the completion of this thesis.

First and foremost, I extend my deepest appreciation to my family for their unwavering support, encouragement, and understanding throughout this journey. Their love and encouragement have been my source of strength, motivating me to persevere through challenges.

I am profoundly thankful to my supervisor, Asst. Prof. Dr. Erkan USLU, for his invaluable guidance, expertise, and mentorship. His insightful feedback, encouragement, and unwavering support have been instrumental in shaping this thesis and enhancing my academic growth.

Finally, I would like to extend my appreciation to my colleagues and friends for their encouragement, insightful discussions, and moral support throughout this research endeavor.

Thank you to everyone who has contributed to this thesis in any way. Your support and encouragement have been invaluable to me, and I am deeply grateful for your contributions.

Muharrem KÜÇÜKYILMAZ

TABLE OF CONTENTS

LIST OF SYMBOLS vii

LIST OF ABBREVIATIONS viii

LIST OF FIGURES x

LIST OF TABLES xiii

ABSTRACT xiv

ÖZET xvi

1 INTRODUCTION 1

1.1 Literature Review . . . 2
1.2 Objective of the Thesis . . . 6
1.3 Hypothesis . . . 6

2 TOOLS AND METHODOLOGY 8

2.1 Tools . . . 8
2.1.1 Robot Operating System - ROS . . . 8
2.1.2 GMapping . . . 11
2.1.3 Adaptive Monte Carlo Localization - AMCL . . . 11
2.1.4 move_base . . . 14
2.1.5 Gazebo . . . 16
2.1.6 Gym . . . 16
2.2 Methodology . . . 18
2.2.1 Pure Pursuit Algorithm . . . 18
2.2.2 Timed Elastic Band - TEB . . . 20
2.2.3 Dynamic Window Approach - DWA . . . 23
2.2.4 Reinforcement Learning: Overview . . . 25
2.2.5 Q-learning . . . 28
2.2.6 Deep Q-Network - DQN . . . 29
2.2.7 Deep Deterministic Policy Gradient - DDPG . . . 32
2.2.8 Twin Delayed Deep Deterministic Policy Gradient - TD3 . . . 35

2.2.9 Curriculum Learning - CL 36

3 PROPOSED METHOD **38**

3.1 Model Evolution Process 41

4 EXPERIMENTAL SETUP **50**

4.1 Simulation Environment 50

4.2 Real Platform and Environment 54

4.3 Training 56

4.4 Metrics 63

5 RESULTS AND DISCUSSION **64**

5.1 Simulation Environment 68

5.2 Real Platform 77

6 CONCLUSION **87**

REFERENCES **89**

PUBLICATIONS FROM THE THESIS **93**

LIST OF SYMBOLS

A_t	Action
$Q_\pi(s, a)$	Action-value Function
$\mu(s\|\theta^\mu)$	Actor Network
γ	Discount Rate
$E_\pi(.)$	Expected Value Under policy π
$L(.)$	Loss Function of neural network
$f_{ob}(.)$	Objective Function for Obstacle
$f_{path}(.)$	Objective Function of Way Point
$\pi(a\|s)$	Probability of Policy
(s_t, a_t, r_t, s_{t+1})	Replay Buffer
G_t	Return
R_t	Reward
S_t	State
$v_\pi(s)$	State-value Function
$Q(s, a; \theta)$	Q-network

LIST OF ABBREVIATIONS

AC	Actor-Critic
AMCL	Adaptive Monte Carlo Localization
AMRs	Autonomous Mobile Robots
API	Application Programming Interface
AUVs	Autonomous Underwater Vehicles
CL	Curriculum Learning
DDPG	Deep Deterministic Policy Grandient
DL	Deep Learning
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
DWA	Dynamic Window Approach
GPS	Global Positioning System
I2A	Imagination-Augmented Agents
LiDAR	Light Detection and Ranging
MC	Monte Carlo
MCTS	Monte Carlo Tree Search
MPC	Model Predictive Control
ODE	Open Dynamics Engine
OGRE	Object-Oriented Graphics Rendering Engine
PID	Proportional Integral Derivative
PPO	Proximal Policy Optimization
ReLU	Rectified Linear Units
RL	Reinforcement Learning

ROS	Robot Operating System
RRT	Rapidly exploring Random Tree
SLAM	Simultaneous Localization and Mapping
SARSA	State-Action-Reward-State-Action
TD	Temporal Difference Methods
TEB	Time Elastic Band
UAVs	Unmanned Aerial Vehicle

LIST OF FIGURES

Figure 2.1 Simulation environment at Gazebo 12
Figure 2.2 Mapping result using GMapping with LiDAR and wheel odometry 12
Figure 2.3 Example of AMCL particle process 14
Figure 2.4 Navigation Stack Setup [28] 15
Figure 2.5 Simulation environment at Gazebo 16
Figure 2.6 Various Gazebo worlds [30] 17
Figure 2.7 Geometry of the Pure Pursuit Algorithm [33] 19
Figure 2.8 TEB: sequences of configurations and time differences [9] . . . 20
Figure 2.9 Minimal distance between TEB and way point or obstacle [9] . . 22
Figure 2.10 Trajectory roll-out based on v, w pair selection [34] 24
Figure 2.11 Velocity Space [8] 24
Figure 2.12 The agent–environment interaction in a Markov decision process. [35] 26
Figure 3.1 39
Figure 3.2 Two of the state space parameters a) angle difference between look ahead distance and robot heading, b) the distance (L) of the robot to its closest point on the global path 41
Figure 3.3 a) sector tiling around the robot, b) smallest distances of LiDAR data on each sector 42
Figure 3.4 Network architecture of the DQNS14A3 model 43
Figure 3.5 Network architecture of the DQNS363A15 model 44
Figure 3.6 Rewad graph of the Distance(L) 45
Figure 3.7 Rewad graph of the Theta(θ) 46
Figure 3.8 Rewad graph of the new parameter of state space(θ) 47
Figure 3.9 Actor Network architecture of the DDPG and TD3 model 48
Figure 3.10 Critic Network architecture of the DDPG and TD3 model 49
Figure 4.1 Waffle model of Turtlebot3 51
Figure 4.2 ROS nodes and messages summarized for the simulation environment 52
Figure 4.3 An example output of A* as viewed in RViz 53
Figure 4.4 Models of robot used in real platform a) Perspective, b) Front side 55
Figure 4.5 Visualization of scan sensor's FOV(Field of View) 55

Figure 4.6 Office environment (representative) 56
Figure 4.7 2D representation of scenarios a) Scenario1, b) Scenario2, c) Scenario3, d) Scenario4 58
Figure 4.8 The 10x10 m^2 Gazebo world 60
Figure 4.9 The 20x20 m^2 Gazebo world 61
Figure 4.10 The 20x20 m^2 Gazebo world showing the possible target locations. 62
Figure 4.11 The Gazebo world containing obstacle 62
Figure 5.1 The cumulative reward graph during the training phase of the DQNS14A3 model 64
Figure 5.2 Cummulative reward of the DQNS363A15 model 66
Figure 5.3 Cumulative reward of the DQNS363A15V2 model trained under the same conditions as DDPG and TD3 models 66
Figure 5.4 Cumulative reward of the DDPG model 67
Figure 5.5 Cumulative reward of TD3 the model 67
Figure 5.6 Scenario 1b results for a) teb_local_planner, b) dwa_local_planner, c) dqn_local_planner 69
Figure 5.7 dqn_local_planner behaviours for a) scenario 1a, b) scenario 1c 70
Figure 5.8 dqn_local_planner behaviours from different runs for scenario 2 when obstacle is encountered at, a) $y > 0$, b) $y \cong 0$, c) $y < 0$. 71
Figure 5.9 Scenario 2 behaviours for, a) teb_local_planner, b) dwa_local_planner 71
Figure 5.10 Scenario 3 behaviour of dqn_local_planner 72
Figure 5.11 The simulation test in complex environment, a) gazebo environment, b) DQN result, c) DDPG result, d) TD3 result . . . 75
Figure 5.12 Simulation test example of the DQNS363A15V2 model 75
Figure 5.13 The simulation test for slalom movement, a) gazebo environment, b) DQN result, c) DDPG result, d) TD3 result . . . 76
Figure 5.14 Real platform path following test Rviz output for DQNS14A3 model 77
Figure 5.15 Real platform path following test for DQNS14A3 model (in left to right, top to bottom order) 78
Figure 5.16 Real platform obstacle avoidance Rviz output for DQNS14A3 model 79
Figure 5.17 Real platform obstacle avoidance test for DQNS14A3 model (in left to right, top to bottom order) 80
Figure 5.18 Real platform obstacle avoidance Rviz output for DQNS363A15 Model 81

Figure 5.19 Real platform obstacle avoidance test for DQNS363A15 model(in left to right, top to bottom order) 82
Figure 5.20 Real platform obstacle avoidance Rviz output for DDPG Model 83
Figure 5.21 Real platform obstacle avoidance test for DDPG model(in left to right, top to bottom order) 84
Figure 5.22 Real platform obstacle avoidance Rviz output for TD3 Model . . 85
Figure 5.23 Real platform obstacle avoidance test for TD3 model (in left to right, top to bottom order) 86

LIST OF TABLES

Table 3.1 The resolution table by distance 42
Table 4.1 Test scenarios 57
Table 5.1 Done reasons of advanced DQN, DDPG and TD3 models 65
Table 5.2 Comparison of all simulation environment scenarios for DQN . . 73
Table 5.3 The results of the DQN, DDPG, and TD3 models in the simulation environment. 74

ABSTRACT

Dynamic Obstacle Avoidance Using Reinforcement Learning

Muharrem KÜÇÜKYILMAZ

Department of Computer Engineering
Master of Science Thesis

Supervisor: Asst. Prof. Dr. Erkan USLU

The main objective of the study is to develop a deep reinforcement learning (DRL) based local planner compatible with the Robot Operating System (ROS) for Autonomous Mobile Robots. Deep Q-Network (DQN), DDPG, and TD3, among other DRL methods, exhibit superior performance in dynamic environments (such as robotic navigation, etc.) and complex situations compared to classical methods. These methods have the capability to learn and represent complex scenarios, including both linear and nonlinear features. The learning process of DQN is model-independent and operates in a discrete action space. In this study, a model developed using the DQN method is compared with well-established classical methods such as Timed Elastic Band (TEB) and Dynamic Window Approach (DWA). Additionally, models with continuous action spaces, DDPG and TD3, are developed and compared with the DQN model. The models have the ability to understand their location in the environment and partially or fully observed maps of the environment. The behavior of the models is tested for their ability to follow paths and avoid both static and dynamic obstacles. The models are trained using a Curriculum Learning approach to learn the required speed control to avoid obstacles and reach the goal as quickly as possible. Promising results from the study reflect the possibility of a more general local planner that can succeed even in dynamic environments.

Keywords: Deep reinforcement learning (DRL), Deep q-network (DQN), Robot operating system (ROS), Deep deterministic policy gradient (DDPG), Twin-delayed deep deterministic policy gradient (TD3).

ÖZET

Pekiştirmeli Öğrenme Kullanarak Dinamik Engel Kaçınımı

Muharrem KÜÇÜKYILMAZ

Bilgisayar Mühendisliği Anabilim Dalı
Yüksek Lisans Tezi

Danışman: Asst. Prof. Dr. Erkan USLU

Çalışmanın temel amacı, Robot İşletim Sistemi (ROS) ile uyumlu Otonom Mobil Robotlar için derin takviyeli öğrenme (DRL) tabanlı bir lokal planlayıcı geliştirmektir. Deep Q-Network (DQN), DDPG ve TD3 gibi DRL yöntemleri, dinamik ortamlarda (örneğin, robotik navigasyon vb.) ve karmaşık durumlarda klasik yöntemlere kıyasla daha iyi performans göstermektedir. Bu yöntemler, doğrusal ve doğrusal olmayan özellikleri içeren karmaşık durumları öğrenme ve temsil etme yeteneğine sahiptir. DQN'in öğrenme süreci model bağımsızdır ve ayrık aksiyon uzayına sahiptir. Bu çalışmada, DQN yöntemini kullanarak geliştirilen model, klasik yöntemler arasında kabul görmüş TEB ve DWA gibi yerel planlayıcılarla karşılaştırılmıştır. Ayrıca, sürekli aksiyon uzayına sahip DDPG ve TD3 modelleri de geliştirilmiş ve bu modeller, DQN modeli ile karşılaştırılmıştır. Modeller, ortamda nerede olduğunu ve kısmen veya tamamen gözlemlenen ortamın haritasını anlama yeteneğine sahiptir. Modelin davranışı yol takip edebilme ve hem statik hem de dinamik engellerden kaçınımı açısından test edilmiştir. Modeller, engellerden kaçınabilmesi ve en kısa yoldan hedefe gidebilmesi için gereken hız kontrolünü öğrenmek için Müfredat Öğrenme yaklaşımı kullanılarak eğitilmiştir. Çalışmadan elde edilen umut verici sonuçlar dinamik ortamlarda bile başarılı daha genel bir lokal planlayıcı olasılığını yansıtmaktadır.

Anahtar Kelimeler: Derin pekiştirmeli öğrenme, Deep q-network (DQN), Robot işletim sistemi (ROS), Deep deterministic policy gradient (DDPG), Twin-delayed deep deterministic policy gradient (TD3).

1
INTRODUCTION

The Introduction chapter serves as the gateway to this thesis, providing an overarching view of the research, its context, and the rationale behind its execution. The first section of this chapter is dedicated to laying the groundwork for an in-depth exploration of Autonomous Mobile Robots (AMRs). Initially, we will delve into the definition and essence of AMRs, aiming to provide a comprehensive understanding of this technological concept. Furthermore, we will focus on well-established mapping, localization, and navigation methods within the realm of AMRs. These methodologies play a pivotal role in shaping the functionality of AMRs, enabling them to navigate and operate effectively in various environments. To comprehend the functionality of AMRs, it is crucial to explore the mapping and localization techniques employed in the field. This chapter will shed light on widely accepted mapping methods and localization strategies, offering insights into how AMRs perceive and understand their surroundings. Subsequently, our attention will shift towards a detailed examination of navigation methods that facilitate AMRs in reaching their destinations efficiently. We will delve into the historical evolution of navigation, analyzing methods employed over time and establishing connections with contemporary reinforcement learning techniques. By tracing the development of navigation methods, we aim to provide valuable insights into their evolution and highlight the significance of current advancements. Specifically, we intend to leverage modern reinforcement learning methods such as DQN, DDPG, and TD3 to enhance the navigation capabilities of autonomous mobile robots. This approach not only builds upon historical knowledge but also opens new avenues for improving navigation in dynamic environments. Upon laying the groundwork for understanding AMRs technology, the sections of this chapter will define the specific objectives and hypotheses guiding our research. By articulating clear objectives, we aim to provide a roadmap for our exploration, while hypotheses serve as the driving force behind our investigative inquiries. This chapter is meticulously designed to facilitate a thorough understanding of the foundational elements of AMR technology, providing a solid foundation for the subsequent sections of the

thesis.

1.1 Literature Review

In recent years, Autonomous Mobile Robots (AMRs) have become increasingly preferred for improving operational efficiency, increasing speed, providing precision, and enhancing safety[1]. They are utilized in various industrial and commercial environments for automatic material handling and logistic operations, such as material transportation, picking, and placement in large warehouses and distribution centers. Ultimately, this optimizes material management processes and increases warehouse efficiency. Furthermore, AMRs can automate tasks like part transportation, pallet handling, and material replenishment in manufacturing facilities, thereby increasing the efficiency of manufacturing processes and freeing up human labor to focus on more strategic tasks. They are also employed for inventory management, restocking shelves, and fulfilling orders in large retail stores, leading to optimized in-store operations, enhanced customer satisfaction, and reduced inventory holding costs. Moreover, they play significant role in transporting cleaning supplies, delivering room service items, and carrying guest luggage in large hotels and hospitality establishments, allowing hotel staff to utilize their time more efficiently while improving the overall guest experience.

AMRs are a type of robot that has the ability to perceive their environment and move autonomously. For this capability, they must know their location and how to navigate. This topic has attracted the attention of many researchers and various methods have been proposed. One of the most widely accepted and used methods today is to first map the environment and use this map for localization and navigation. Localization and mapping play a key role in AMRs.

Simultaneous Localization and Mapping (SLAM) [2] is a method that allows a mobile robot to simultaneously create a map of its unknown environment using sensors and determine its own position based on this map. Mapping and localization tasks, requires the use of sensors such as cameras, range finders using sonar, laser and global positioning system (GPS).

Localization can be defined as the information of where the robot is on the map. Localization information can be obtained from the wheel odometer. But the error in the wheel odometer is incremental over time [3]. Therefore, robot position can be found with triangulation formula by using beacon [3] or reflector [4]. In addition, it is possible to improve the position with object detection using a depth camera [5] or by matching the measurements taken from the real environment with the map using

Light Detection and Ranging (LiDAR) sensors.

Navigation can be defined as the process of generating the speed commands necessary for a robot to reach a goal successfully. The navigation methods for mobile robots can roughly be divided into two categories: global planner and local planner. Operating on a global scale and considering the entire environment, global planners aim to find an optimal or near-optimal path through free space from the robot's current position to the goal, taking into account factors such as obstacles, varying terrain, and the robot's kinematics. The generated path is designed to optimize criteria such as distance, time, or energy consumption. These planners use various algorithms such as A* algorithm [6], which employs a heuristic approach to efficiently search through the possible paths, or the Rapidly Exploring Random Tree (RRT) which is a probabilistic approach that works well in complex and dynamic environments. RRT [7] incrementally builds a tree of potential paths by randomly exploring the configuration space, making it adaptable to changing surroundings. However, these planners often require the support of local planners, which are designed to handle short-term obstacles and adapt to dynamic changes in the environment. Local planners such as such as Dynamic Window Approach (DWA) [8], Timed Elastic Band (TEB) [9] are specifically tailored to enable the robot to swiftly respond to immediate environmental conditions, navigate through sudden obstacles, and move safely towards the goal. They typically possess the capacity to manage situations that demand a high level of precision and speed. The role of local planning is crucial in effectively implementing global plans, addressing the dynamic variations in environmental conditions, and coping with unexpected obstacles.

Having defined the local planner, machine learning, and deep learning are modern techniques commonly employed to enhance the efficiency of local planning processes. An example in this context could be the study by Xiao et al. [10], which attempts to generate speed commands by imitating the Model Predictive Control (MPC) method using a self-supervised approach. Additionally Pokle et al. [11] aimed to create both global routes and local speed commands using deep learning techniques in their study.

Recently, Reinforcement Learning (RL) has emerged as a prominent approach in this field. Numerous articles in the realm of RL offer valuable insights into various applications and methodologies. In a specific study by Rubi et al. [12], an agent has been trained using the Deep Deterministic Policy Grandient (DDPG) RL algorithm to generate orientation and velocity commands for quadrotor vehicles. Another notable contribution comes from Zhou et al. [13], where the position and

velocity controllers in the PX4 autopilot firmware for Unmanned Aerial Vehicles (UAVs) were replaced with Deep Reinforcement Learning (DRL) techniques. Lastly, Zhang et al. [14] employed their proposed Deep Interactive Reinforcement Learning method to teach an agent how to perform straight-line and sinusoidal curve following for Autonomous Underwater Vehicles (AUVs). So far, various instances of research conducted on diverse robotic platforms, including UAVs, quadrotors, and AUVs, have been presented.

In addition to this, substantial research has been dedicated to AMRs. In their study, Ejaz et al. [15] trained a DRL model for tracked robots to learn collision-free autonomous navigation using depth images. Similarly, Ruan et al. [16] and Marchesini et al. [17] focused on collision-free navigation systems using DRL in their studies.

RL is an area of machine learning that focuses on how agents or systems interact with their environment, aiming to learn optimal behaviours or strategies though trial-and-error experiences to maximize cumulative rewards [18]. RL algorithms enable agents to autonomously learn and adapt their behaviors based on feedback received from the environment. They do this without requiring explicit supervision or labeled datasets. By iteratively exploring different actions and observing the resulting rewards, RL agents can gradually improve their decision-making abilities and achieve higher performance levels in various tasks.

Within RL, there are various approaches to determining which actions to take. As classified in the article [19], reinforcement learning algorithms are categorized from different perspectives, including model-based and model-free methods, value-based and policy-based methods, Monte Carlo methods and temporal-difference methods, on-policy and off-policy methods. Firstly, "model-based" approaches rely on learning how the environment operates through a model. These models attempt to predict the dynamics of the environment and the relationships between events, helping the agent plan its actions accordingly. Notable model-based algorithms include Imagination-Augmented Agents (I2A) and Monte Carlo Tree Search (MCTS) such as AlphaGo and AlphaZero. Conversely, "model-free" approaches do not utilize such models and instead learn directly from experiences. While these approaches may be simpler and more computationally efficient, they may struggle to fully understand the structure of the environment. Notable model-free algorithms include Temporal Difference (TD) methods and Monte Carlo (MC) methods. Subsequently, RL methods can be divided into "value-function-based" and "policy-based" approaches. "Value-function-based" methods select the best actions by calculating a value function for each state. These functions measure

the value of each state and guide the agent's actions accordingly. Examples of value function-based algorithms include dynamic programming-based optimal control approaches such as policy iteration, rollout-based MC methods, TD methods such as SARSA,State-Action-Reward-State-Action, (also off-policy), Q-learning (also on-policy), and DQN. On the other hand, "policy-search" methods find the best actions by directly determining a policy. A policy defines the action to be taken in a particular state and enables the agent to act according to this policy directly. Noteworthy policy-search algorithms include Proximal Policy Optimization (PPO), Actor-Critic (AC) method and Deep Deterministic Policy Gradient (DDPG) method.

These methods are all significant contributions to the field of RL. Value Function methods provide a foundational framework for assessing the quality of different states and actions within an environment. By estimating the value of being in a particular state under a given policy, value functions guided agents towards actions that maximized long-term rewards. Monte Carlo methods, inspired by the principles of statistical sampling, estimated value functions by averaging returns obtained from complete episodes of interaction with the environment. Monte Carlo methods were particularly effective in episodic environments, where agents could interact with the environment until reaching a terminal state. Temporal Difference (TD) methods emerged as a significant advancement in RL, combining elements of both Value Function and Monte Carlo methods. They update value estimates based on the difference between successive estimates at each time step. This allowed for more efficient learning, enabling agents to update their estimates online without waiting for the completion of entire episodes. Within TD methods, SARSA and Q-learning are notable algorithms. SARSA is an on-policy TD control algorithm that updates action values based on the agent's current policy. In contrast, Q-learning [20] is an off-policy TD control algorithm that learns the optimal action-value function without requiring knowledge of the policy being followed. Following the success of Temporal Difference (TD) methods, further advancements in RL were achieved with the introduction of Deep Q-Networks (DQN). DQN represents a significant milestone in the integration of deep learning techniques with reinforcement learning principles. Introduced by Mnih et al. in 2015 [[21], DQN combines deep neural networks with Q-learning. Unlike traditional Q-learning, which operates on tabular representations of state-action values, DQN learns a parameterized action-value function represented by a deep neural network. This capability allows DQN to effectively handle high-dimensional input spaces, such as those encountered in tasks like robot navigation.

In this study, initially, a new local planner method named `dqn_local_planner`

is proposed using the DQN algorithm. For the DQN algorithm, a fully connected deep neural network is employed to generate velocity data for AMR after a series of preprocessing steps involving LiDAR data, robot position, and global path. The training and testing of the model are conducted using the ROS framework, Gazebo simulation, and Gym environment. Static and dynamic environments are created in Gazebo for comparing the proposed model with `teb_local_planner` and `dwa_local_planner`. Subsequently, local planners are developed using the DDPG and TD3 models, and they are compared with the DQN model. This comparison is carried out between the discrete action space of DQN and the continuous action space of DDPG and TD3.

The paper is structured as follows: The literature review, the objective of the thesis, and the hypothesis are explained in Chapter 1. In Chapter 2 tools and algorithms utilized for the proposed method and comparisons are mentioned, the explanation of proposed method is given in Chapter 3, the experimental set up for training and testing the model is explained in Chapter 4, the results obtained and the evaluation of these results are presented and discussed in Chapter 5 and an overall evaluation of the study together with future work is given in Chapter 6.

1.2 Objective of the Thesis

The main objective of this thesis is to develop an effective and adaptive local planner with a high capability for obstacle avoidance for autonomous mobile robots using the DQN, DDPG, and TD3 algorithms. In this context, the aim is to enhance the robot's abilities to understand and learn from environmental conditions, and use this information to safely and effectively follow the global path. Throughout the thesis, the impact of the DQN, DDPG, and TD3 algorithms on local planning will be thoroughly investigated, and the proposed system is intended to contribute to the more efficient navigation of autonomous mobile robots in complex environments. The obtained results will assess the potential of DQN, DDPG, and TD3-based local planning in enhancing the performance of autonomous mobile robots and will serve as a foundation for future similar studies.

1.3 Hypothesis

It is hypothesized that a local planner developed using the DQN algorithm enables autonomous mobile robots to make more successful decisions with respect to navigating obstacles and reacting to dynamic obstacles more effectively in challenging conditions (such as large obstacles along the route) and situations with

limited mobility compared to classical methods like TEB and DWA.

It is hypothesized that training the model with DQN, which has a discrete action space, is easier compared to methods like DDPG and TD3, which have continuous action spaces.

It is hypothesized that the DQN method will achieve more successful results in terms of obstacle avoidance and reaching the goal for robots when compared to the DDPG and TD3 methods.

2
TOOLS AND METHODOLOGY

In this chapter, the tools necessary for training and testing the model (such as ROS, Gazebo, Gym) will be explained. Additionally, a detailed exploration of the methods employed within the scope of the thesis (Pure Pursuit Algorithm, DWA, TEB, Q-learning, DQN, DDPG, TD3) will be provided. Additionally, the use of Curriculum Learning during the training process will be discussed.

2.1 Tools

This section will discuss the tools required for using the simulation environment and the real robot. These tools include ROS, GMapping, AMCL, move_base, Gazebo, and gym.

ROS is an open-source environment used for robot software development. GMapping is a mapping package that enables the robot to create maps of its environment. AMCL is a localization package that determines the robot's position. move_base is a package in ROS used for navigation stack. Gazebo is a simulation environment utilized during the training and testing phases of the robot. Gym, on the other hand, is a framework developed for implementing and testing RL algorithms.

2.1.1 Robot Operating System - ROS

ROS is an open source software development kit that is widely used in robotics applications. Even though it is named as an operating system ROS is mainly a framework. ROS provides solutions as native and as third party tools for hardware drivers to state-of-the-art algorithms. ROS also provides integration with open source development tools and simulation environments [22].

Although ROS has moved recently to a more industry friendly standard and version, namely ROS2 [23], which offers native data security and addresses problems that

are encountered on previous versions, our work still relies on ROS Noetic version for consistency, stability and legacy code-base reasons. Three main algorithms that are also implemented as ROS nodes and used in our study namely; GMapping, Adaptive Monte Carlo Localization (AMCL) and move_base are given in the following subsections. Rviz the visualization tool for ROS also needs a special mention as it is widely used during our study.

The fundamental terminologies in ROS include nodes, topics, messages, services, actions, the parameter server, and launch files [24]. A ROS node is a fundamental component in the Robot Operating System (ROS), typically representing a program or process that performs a specific function within a ROS application and communicates with other nodes via the ROS communication system. Nodes can communicate with each other through messages and register with ROS components such as topics, services, and actions via a central communication point called the ROS Master.

A ROS topic is a named communication channel that enables data exchange between different nodes in the Robot Operating System (ROS). Nodes can publish messages to topics or subscribe to topics to receive messages. Topics facilitate asynchronous communication among nodes, allowing them to communicate independently of each other's execution timelines. This decoupling of communication enables modular and flexible designs in ROS applications, where nodes can interact with each other in a loosely coupled manner.

A ROS service serves as a communication mechanism facilitating specific tasks or functionalities between nodes in a request-response fashion. Unlike topics, which enable one-to-many communication, services establish point-to-point communication between a client node, initiating the service request, and a server node, responsible for executing the service. Services operate on a client-server model, where the client node sends a request to the server node, prompting the server to process the request and provide a response. This request-response protocol allows clients to specify the desired service type along with any necessary input parameters, with servers responding with the result of the operation. Communication between nodes in ROS services is synchronous, meaning the client typically awaits the server's response before proceeding with other tasks. This synchronous nature ensures that the client obtains the result of the service call promptly. Each service in ROS features a well-defined interface, comprising a service name, request message type, and response message type. This interface defines the structure of the messages exchanged between client and server nodes, ensuring compatibility and interoperability. Services find extensive use in ROS

applications for tasks requiring direct interaction or coordination between nodes, such as querying sensor data, initiating robot actions, or executing complex computations.

A ROS action is a specialized communication mechanism designed for managing long-running, asynchronous tasks or behaviors within distributed robotic systems. Unlike services, which follow a request-response pattern, actions operate based on a goal-result protocol. In this protocol, a client node sends a goal request to a server node, which then performs the requested action asynchronously. Throughout the execution of the action, the server node can provide periodic feedback to the client, allowing the client to monitor the progress of the task. Once the action is completed or terminated, the server sends a result back to the client. Actions also support cancellation, enabling clients to request the termination of an action if necessary. This flexible and scalable communication mechanism is commonly used in ROS for tasks that require long-running behavior, such as navigation, motion planning, manipulation, or task execution in robotic systems. With well-defined interfaces and built-in support for feedback and cancellation, actions enable the development of robust and scalable robotic applications within the ROS ecosystem.

The parameter server in ROS serves as a centralized storage system for managing runtime parameters and configuration settings used by ROS nodes. It stores parameters as key-value pairs, allowing nodes to dynamically access and update parameters during execution. This enables nodes to configure their behavior or adjust their operation at runtime without the need for source code modifications or node restarts. Parameters can represent various settings such as sensor calibration values, robot configuration parameters, control gains, or navigation parameters. The parameter server organizes parameters into a hierarchical namespace, providing a structured and organized way to manage parameters within large and complex ROS applications. ROS nodes can access parameters stored on the parameter server to retrieve their current values or update them as needed. Additionally, ROS launch files can specify parameter values to be set on the parameter server when nodes are launched, simplifying the configuration process and ensuring consistency across multiple nodes.

roslaunch is a command-line utility used to launch and manage ROS nodes, streamlining the deployment process through the use of XML-based launch files. With roslaunch, users can start one or multiple ROS nodes specified within a launch file, along with setting parameters, managing namespaces, and remapping topics and services as needed. This tool facilitates efficient node configuration and management by allowing parameters to be set dynamically, enabling nodes to be

organized into namespaces to prevent naming conflicts, and providing options for monitoring and controlling the lifecycle of launched nodes.

After discussing all of these ROS terminologies, let's delve into how we utilized ROS in our thesis. Our model has been integrated into ROS as follows. It subscribes to the `scan` topic to listen to laser data, `odom` topic to obtain information about the robot's position, `rl_global_path` topic to receive route information, and `move_base_simple/goal` topic to receive goal information. Additionally, it publishes velocity data on the `rl_cmd_vel` topic and planner status on the `rl_robot_status` topic. This integration allows our model to effectively communicate within the ROS ecosystem, enabling the exchange of various information about the robot's status and planning process.

2.1.2 GMapping

GMapping ROS package is the implementation of OpenSlam's GMapping algorithm [25]. GMapping is Rao-Blackwellized particle filter based SLAM approach, as described in Algorithm 1. Each particle holds an individual belief of the constructed map and updates this belief according to robot movement and sensor readings. GMapping also incorporates occupancy grid mapping to represent the environment, where occupancy log likelihood of each grid is calculated and accumulated based on occupied and free observations of the grid on sensor readings. GMapping imposes a highly representative particle distribution thanks to particle update based on both robot movement and the actual sensor reading. The output of GMapping, which is an occupancy grid type map, can be saved and later served for localization needs using the `map_server` ROS package.

The mapping steps of environment as shown in Figure 2.1 are shown in figure 2.2.

2.1.3 Adaptive Monte Carlo Localization - AMCL

AMCL is a particle filter based 2D probabilistic localization method [26]. AMCL localization accepts a map and LiDAR readings as input and estimates the robot pose based on up to scale of hundreds or thousands of particle representations. The method is composed of four main stages namely initialization, motion model, weighting and resampling. Initialization takes place only once at the beginning of the localization process or whenever the robot is assumed to be lost. Samples drawn from an any arbitrary probability distribution can be used for each particle's initial pose and orientation. If there is even a rough prior information related to whereabouts of the robot the distribution variance can be kept small and the

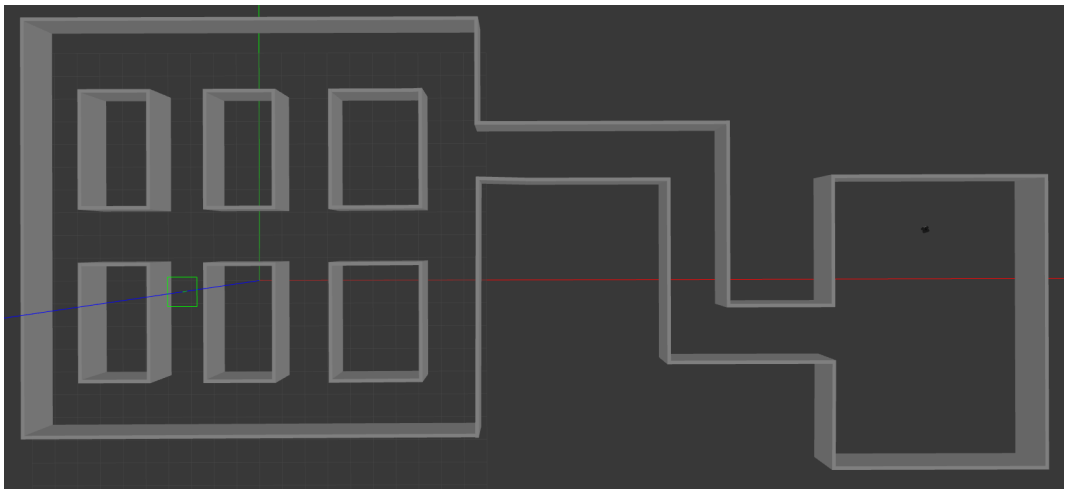


Figure 2.1 Simulation environment at Gazebo

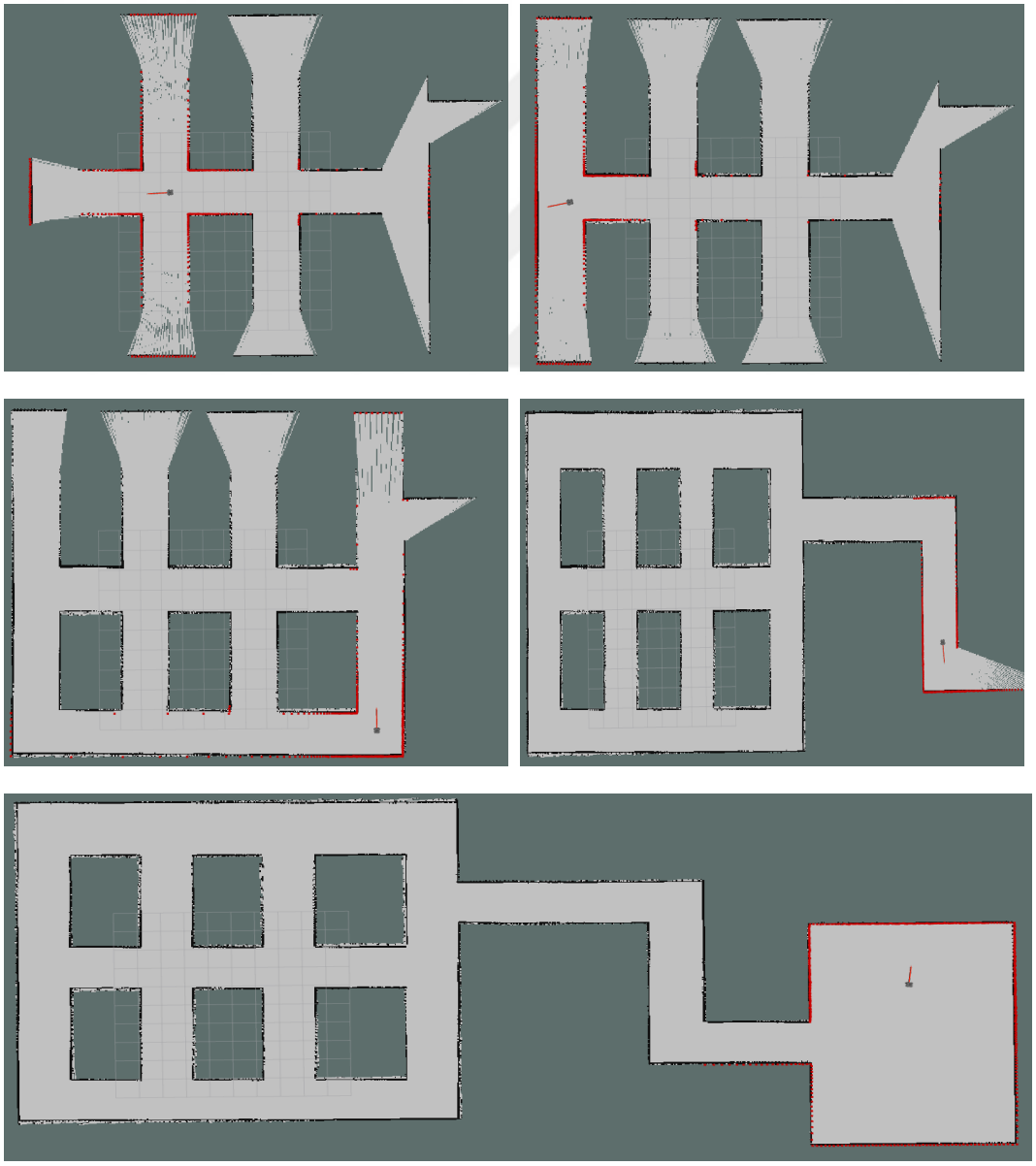


Figure 2.2 Mapping result using GMapping with LiDAR and wheel odometry

Algorithm 1 Improved RBPF for Map Learning

Require:
S_{t-1}, the sample set of the previous time step
z_t, the most recent laser scan
u_{t-1}, the most recent odometry measurement
Ensure:
S_t, the new sample set
$S_t = \{\}$
for $s_{t-1}^{(i)} \in S_{t-1}$ **do**
    $< x_{t-1}^{(i)}, w_{t-1}^{(i)}, m_{t-1}^{(i)} >= s_{t-1}^{(i)}$
    // scan-matching
    $x_t^{'(i)} = x_{t-1}^{(i)} \oplus u_{t-1}$
    $\hat{x}_t^{(i)} = argmax_x p(x|m_{t-1}^{(i)}, z_t, x_t^{'(i)})$
    if $\hat{x}_t^{(i)}$ = failure **then**
        $x_t^{(i)} \approx p(x_t|x_{t-1}^{(i)}, u_{t-1})$
        $w_t^{(i)} = w_{t-1}^{(i)} \cdot p(z_t|m_{t-1}^{(i)}, x_t^{(i)})$
    else
        // sample around the mode
        for $k = 1, ..., K$ **do**
            $x_k \approx \{x_j \,|\, |\, x_j - \hat{x}^{(i)} < \triangle\}$
        end for
        // compute Gaussian proposal
        $\mu_t^{(i)} = (0, 0, 0)^T$
        $\eta^{(i)} = 0$
        for $x_j \in x_1, ..., x_K$ **do**
            $\mu_t^{(i)} = \mu_t^{(i)} + x_j \cdot p(z_t|m_{t-1}^{(i)}, x_j) \cdot p(x_t|x_{t-1}^{(i)}, u_{t-1})$
            $\eta^{(i)} = \eta^{(i)} + p(z_t|m_{t-1}^{(i)}, x_j) \cdot p(x_t|x_{t-1}^{(i)}, u_{t-1})$
        end for
        $\mu_t^{(i)} = \frac{\mu_t^{(i)}}{\eta^{(i)}}$
        $\sum_t^{(i)} = 0$
        for $x_j \in x_1, ..., x_K$ **do**
            $\sum_t^{(i)} = \sum_t^{(i)} + (x_j - \mu^{(i)}) \cdot (x_j - \mu^{(i)})^T \cdot p(z_t|m_{t-1}^{(i)}, x_j);$
                $\cdot p(x_j|x_{t-1}^{(i)}, u_{t-1})$
        end for
        $\sum_t^{(i)} = \frac{\sum_t^{(i)}}{\eta^{(i)}}$
        // sample new pose
        $x_t^{(i)} \approx N(\mu_t^{(i)}, \sum_t^{(i)})$
        //update importance weights
        $w_t^{(i)} = w_{t-1}^{(i)} \cdot \eta^{(i)}$
    end if
    // update map
    $m_t^{(i)} = integrateScan(m_{t-1}^{(i)}, x_t^{(i)}, z_t)$
    // update sample set
    $S_t = S_t \cup \{< x_t^{(i)}, w_t^{(i)}, m_t^{(i)} >\}$
end for
$N_{eff} = \frac{1}{\sum_{i=1}^{N} (\tilde{w}^{(i)})^2}$
if $N_{eff} < T$ **then**
    $S_t = resample(S_t)$
end if

distribution mean can be determined close to this prior information. Yet if there is no prior information about initial robot pose all samples can be distributed randomly on the known map. As a second step each particle is moved accordingly with the local position change read from an internal sensor. Third step requires both the particle pose and the external sensor reading. Each particle is evaluated according their estimated pose on the known map and closeness of each particles expected sensor reading and the actual sensor reading. Cumulative weight sum is normalized to 1 and each particle is assigned an importance directly related to its normalized weight. As the forth step particle space is resampled according to particle importance. That way mostly highly important particles and by chance some other particles are carried out for the next iteration. The algorithm continues with the next internal sensor reading and runs through the iterative steps. The entire process is also demonstrated in Algorithm 2. It can be observed how the blue particles, representing the initial state with AMCL, converge towards the center as the robot moves, indicating improvement in position, much like in Figure 2.3.

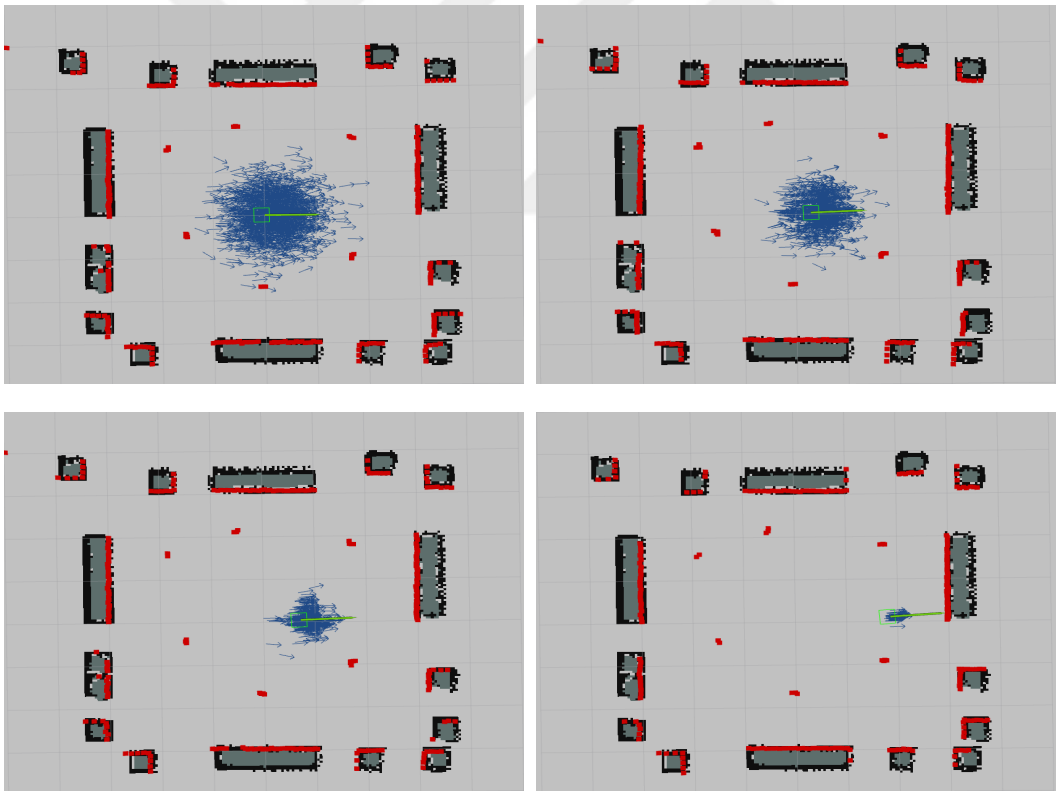


Figure 2.3 Example of AMCL particle process

2.1.4 move_base

`move_base` ROS package is a collection of tools (path planners, navigation controllers, recovery behaviour controllers) and a state machine as shown in

Algorithm 2 Augmented MCL

Argument (x_{t-1}, u_t, z_t, m)
static w_{slow}, w_{fast}
$\overline{X}_t = X_t = \emptyset$
for $m = 1$ to M **do**
    $x_t^{[m]}$ = sample motion model(u_t, x_{t-1})
    $w_t^{[m]}$ = measurement model$(z_t, x_t^{[m]}, m)$
    $\overline{X}_t = \overline{X}_i + < x_t^{[m]}, w_t^{[m]} >$
    $w_{avg} = w_{avg} + \frac{1}{M \cdot w_t^{[m]}}$
end for
$w_{slow} = w_{slow} + \alpha_{slow} \cdot (w_{avg} - w_{slow})$
$w_{fast} = w_{fast} + \alpha_{fast} \cdot (w_{avg} - w_{fast})$
for m=1 to M **do**
    with probability **max**$(0.0, 1 - \frac{w_{fast}}{w_{slow}})$**do**
        add random pose to x_t
    else
        draw $i \in \{1, ..., N\}$ *with probability* $\propto w_t^{[i]}$
        add $x_t^{[i]}$ to x_t
    endwith
end for
return X_t

Figure 2.4 that controls state transitions and message passing [27]. Main purpose of `move_base` package is to control the path planning, navigation control and if added control of recovery behaviors overall which can be defined as the navigation itself. `move_base` package enables navigation tools to be integrated through plugins. Path planners use the global planner plugin interface and navigation controllers use the local planner plugin interface. Our implementation relies on A* path planner which is implemented under name `navfn`. The proposed approach directly targets a replacement for classical navigation controllers such as `teb_local_planner` and `dwa_local_planner`.

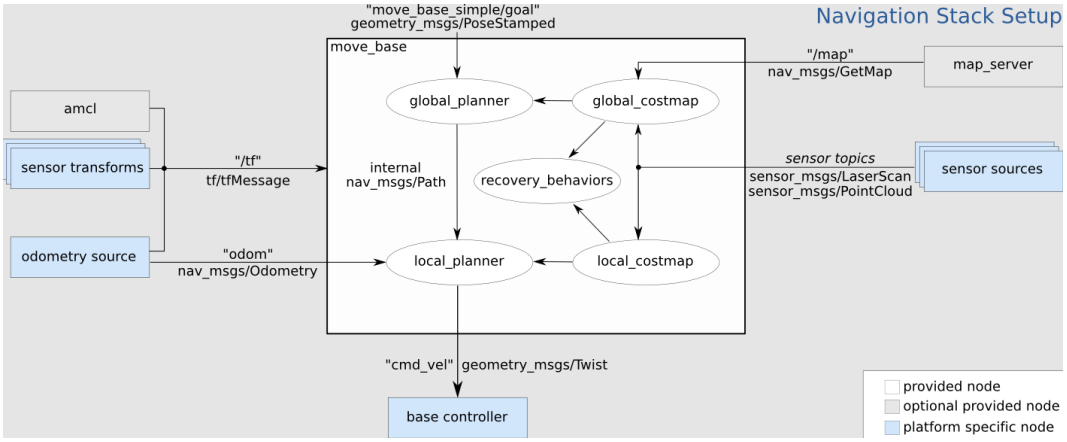


Figure 2.4 Navigation Stack Setup [28]

2.1.5 Gazebo

Gazebo, maintained by Open Robotics, is an open source 3D robotics simulator. Gazebo integrates environment simulation, robot simulation, sensor simulation, message passing transport, physics engine and rendering all together [29]. Gazebo can utilize several physics engine where Open Dynamics Engine (ODE) is the default one. On 3D rendering Gazebo uses Object-Oriented Graphics Rendering Engine (ORGE). Gazebo as a standalone simulator can also be integrated with ROS through plugins. Whereas the newest Gazebo version is Garden, our work relies on Gazebo Classic v11. Our model's training environment at Gazebo is shown in Figure 2.5.

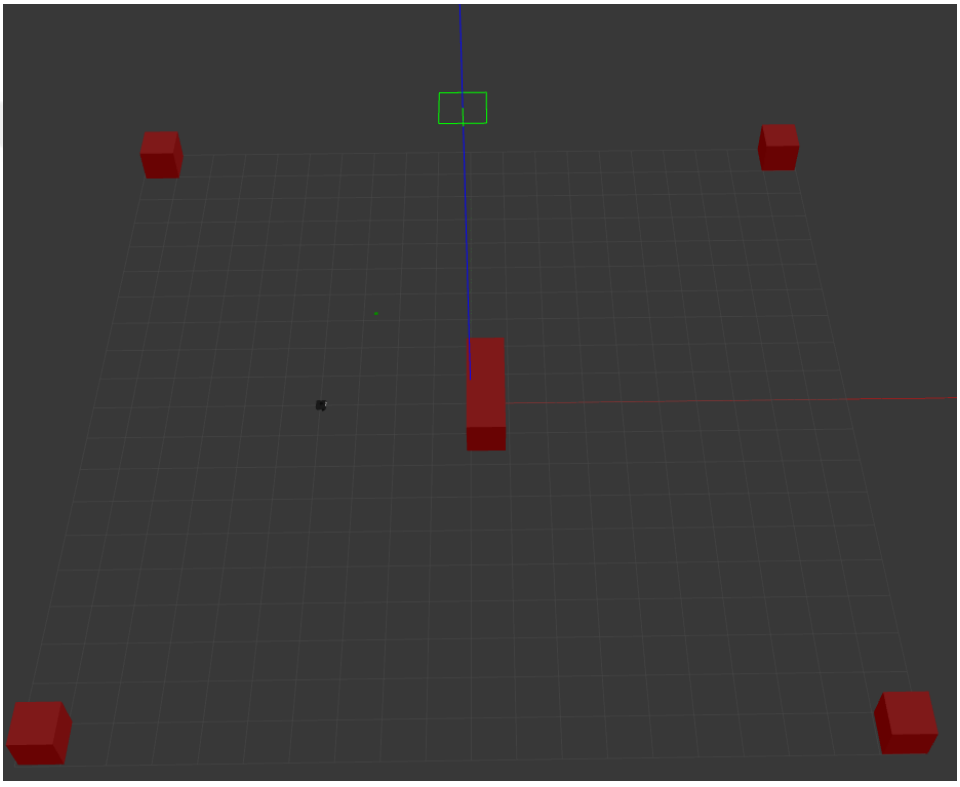


Figure 2.5 Simulation environment at Gazebo

Complex environments can indeed be created in Gazebo worlds, as shown in Figure 2.6. These environments, which encompass natural landscapes, urban settings, indoor spaces, diverse weather conditions, and challenging obstacles, serve as comprehensive testbeds for the development and evaluation of robotic and autonomous systems.

2.1.6 Gym

Gym, developed by OpenAI, is a platform that offers a standardized Application Programming Interface (API) for RL methods along with a vast collection of

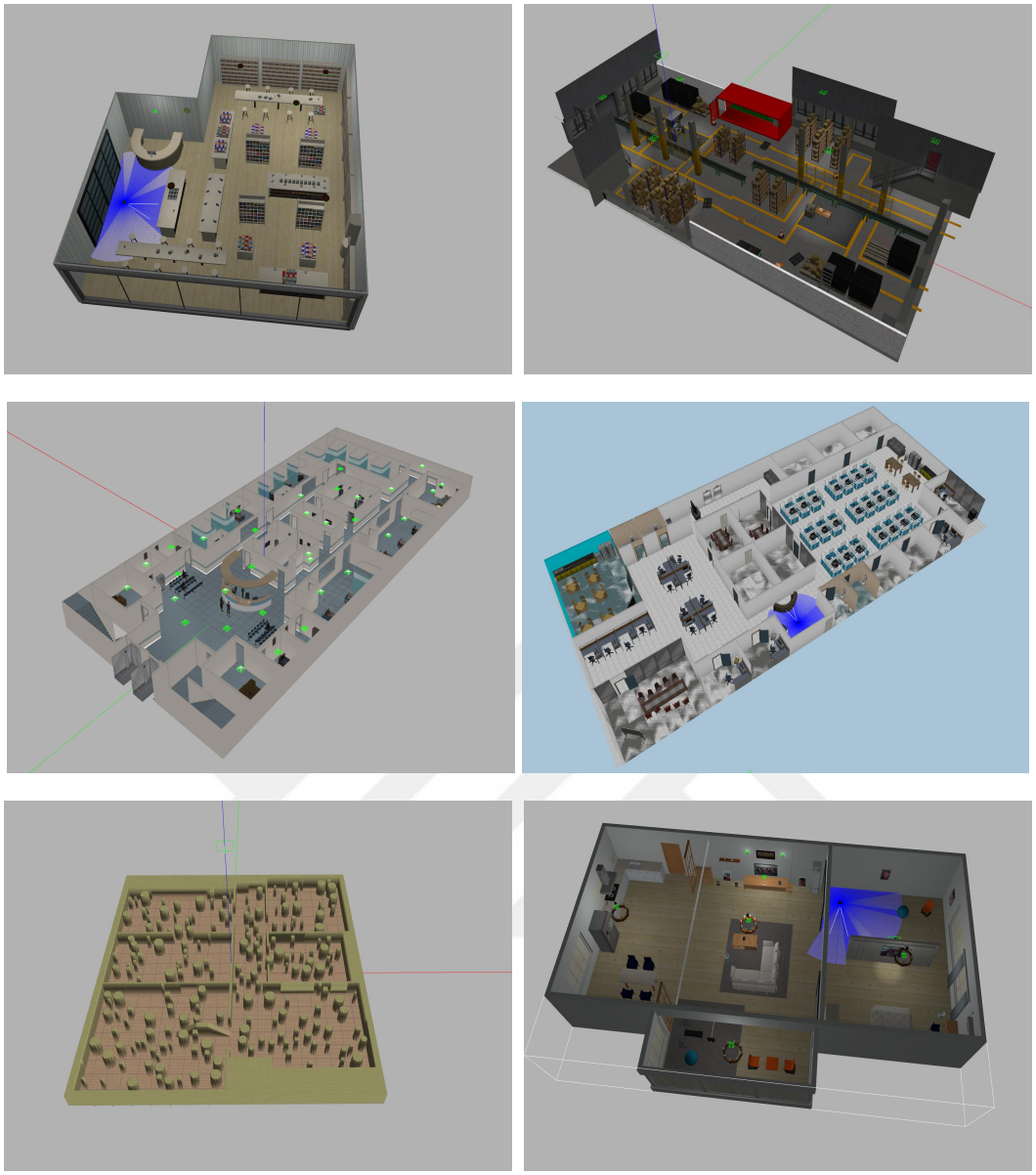


Figure 2.6 Various Gazebo worlds [30]

reference environments [31]. These environments serve as simulated settings for conducting RL research and experimentation. Within Gym, there is a diverse range of reference environments tailored to different tasks and domains. For instance, there are classic control tasks like CartPole and MountainCar, where agents must control simple systems to achieve specific goals. Additionally, Gym provides access to Atari games, offering a variety of classic video games like Space Invaders and Pong, which serve as benchmark environments for testing RL algorithms. Furthermore, there are environments based on the MuJoCo physics engine, featuring realistic physics simulations for tasks such as robot control and manipulation. Gym also includes simulated robotics tasks like Fetch and Hand manipulation, as well as classic board games such as Chess and Go. Apart from native environments that Gym provides, Gym is also highly flexibly on integration with Gazebo simulation environment and consequently with ROS. ROS and Gazebo integration is carried out through the `openai_ros` wrapper package.

Gym is recently moved to a new name, which is Gymnasium, that is created by Farama Foundation starting with version v.26 [32]. Our work relies on OpenAI's Gym version.

2.2 Methodology

This section will encompass a detailed explanation of the RL algorithms like DQN, DDPG and TD3 utilized in our model, the Pure Pursuit algorithm that inspired the creation of our state space, the TEB and DWA algorithms employed for comparing model performance, and the Curriculum Learning (CL) topic, which offers a unique perspective on the learning process of our model.

2.2.1 Pure Pursuit Algorithm

The Pure Pursuit algorithm [33] is a commonly used tracking algorithm in robotics and autonomous systems. This algorithm utilizes the geometric relationships between a robot's current position, motion direction, and a point on its path at a specified look-ahead distance, often referred to as the target point. The algorithm continually evaluates the robot's current position and the target point, adjusting the robot's movement accordingly. The primary objective is to calculate the curvature of the arc connecting the robot's current position, with a chord length of l as shown in Figure 2.7. Subsequently, this calculated curvature is translated into the steering angle of the robot.

The derivation of the curvature of the arc is described in Equation 2.1.

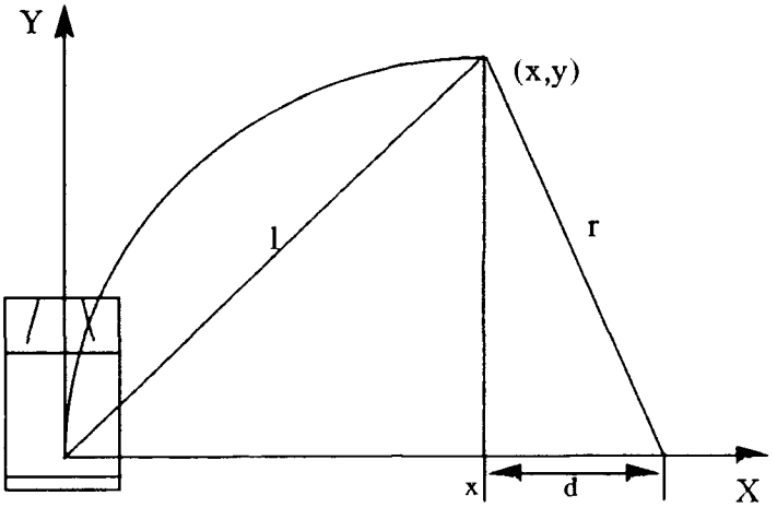


Figure 2.7 Geometry of the Pure Pursuit Algorithm [33]

$$
\begin{aligned}
x^2 + y^2 &= l^2 \\
x + d &= r \\
d &= r - x \\
(r - x)^2 + y^2 &= r^2 \\
r^2 - 2rx + x^2 + y^2 &= r^2 \\
2rx &= l^2 \\
r &= \frac{l^2}{2x} \\
\gamma &= \frac{2x}{l^2}
\end{aligned}
\tag{2.1}
$$

Thus, the robot follows an optimized path as it tracks towards the target.

The Pure Pursuit algorithm can be outlined in Algorithm 3.

Algorithm 3 Pure Pursuit Algorithm

1: Initialize current position and motion direction
2: Choose a target point on the path (look ahead distance away)
3: **while** Not reached the target **do**
4: Evaluate current position and the target point
5: Calculate new motion direction
6: Adjust the robot's movement
7: **end while**

2.2.2 Timed Elastic Band - TEB

TEB is a motion planning method used in robotics and autonomous navigation systems. It is designed to efficiently navigate through dynamic environments while considering both static obstacles and moving objects. The TEB algorithm operates by generating a time-parameterized trajectory for the robot to follow while avoiding collisions with obstacles.

The classic 'elastic band' is described as a sequence of n intermediate configuration Q, which consist of robot positions x_i, y_i and orientations β_i. Mathematically, it can be represented as:

$$Q = \{\mathbf{X}_i\}_{i=0...n} \quad n \in \mathbf{N} \tag{2.2}$$

Here, **X** denotes $(x, y, \beta)^T$

TEB represents an augmented configuration $B := (Q, t)$ that includes the time t required for the robot to move from one configuration to the next in sequence, as shown in Figure 2.8. t can be expressed as:

$$t = \{\triangle T_i\}_{i=0...n-1} \tag{2.3}$$

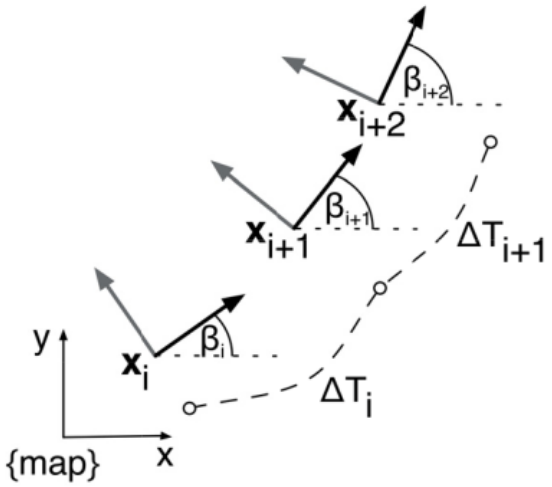


Figure 2.8 TEB: sequences of configurations and time differences [9]

The purpose of TEB is to produce the most ideal velocity command for mobile robots by optimizing both the configuration and the time interval with weighted multi-objective optimization in real-time, considering the kino-dynamic constraints such as velocity and acceleration limits [9]. The objective function is described as

in Equation 2.4:

$$\begin{aligned} f(B) &= \sum_{k} \gamma_k \cdot f_k(B) \\ B^* &= \underset{B}{argmin} f(B) \end{aligned} \tag{2.4}$$

where γ denotes weights, f_k denotes the constraints and objectives with respect to trajectory, and B^* denotes the optimized TEB.

The objective functions of the TEB belong to two types: constraints such as velocity and acceleration limits formulated in terms of penalty functions and objectives with respect to trajectory such as shortest or fastest path or clearance from obstacles. Sparse constrained optimization algorithms are not readily available in robotic frameworks (e.g. ROS) in a freely usable implementation. Therefore, in the context of "timed elastic band" these constraints are formulated as objectives in terms of a piecewise continuous, differentiable cost function that penalize the violation of a constraint as described in Equation 2.5.

$$\epsilon_r\left(x, x_r, \epsilon, S, n\right) \cong \begin{cases} \left(\frac{x-(x_r-\epsilon)}{S}\right)^n, & \text{if } x > x_r - \epsilon \\ 0, & \text{otherwise} \end{cases} \tag{2.5}$$

where x_r denotes the bound, S denotes the scaling and n denotes the polynomial order and ϵ denotes the small translation of the approximation.

The TEB algorithm takes into account the positions of both way points and obstacles in the environment when generating the trajectory for the robot. It aims to produce a path that allows the robot to efficiently navigate from its current location to the designated way points while avoiding collisions with obstacles along the way. The objective function is based on the minimal separation $d_{min,j}$ between the TEB and the way points or obstacles as shown in Figure 2.9. The objective function of way points can be expressed as:

$$f_{path} = e_\Gamma(d_{min,j}, r_{p_{max}}, \epsilon, S, n) \tag{2.6}$$

where the distance is bounded above by a maximum target radius $r_{p_{max}}$

Similarly, the objective function for obstacles is given by:

$$f_{ob} = e_\Gamma(-d_{min,j}, -r_{o_{min}}, \epsilon, S, n) \tag{2.7}$$

where it is constrained below by a minimum distance $r_{o_{min}}$.

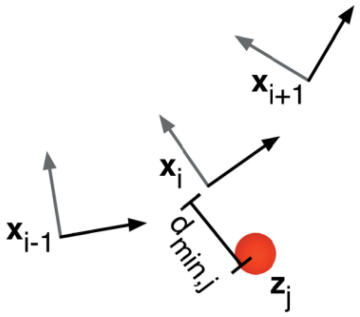


Figure 2.9 Minimal distance between TEB and way point or obstacle [9]

TEB imposes constraints on velocity and acceleration to regulate the robot's movement. These constraints are commonly expressed as mathematical equations, with velocity and acceleration subjected to specific minimum and maximum bounds. For instance, TEB establishes upper and lower bounds on both velocity and acceleration to keep the robot's motion within acceptable ranges throughout its trajectory. These bounds are typically represented as inequalities, where $v(t)$ denotes the robot's velocity at time t, and $a(t)$ epresents its acceleration. The parameters v_{min} and v_{max} signify the minimum and maximum allowable velocities, while a_{min} and a_{max} indicate the minimum and maximum allowable accelerations, respectively.

Additionally, TEB utilizes optimization techniques to determine the most suitable velocity profile for the robot along its path. Optimization algorithms adjust the robot's velocity to minimize a cost function, which quantifies the deviation from desired velocity profiles while adhering to imposed constraints. This optimization process facilitates efficient and effective navigation, considering environmental constraints and specified targets.

By combining these approaches, TEB ensures that the robot operates within safe and efficient velocity and acceleration ranges throughout its trajectory. This integrated framework enables the robot to navigate complex environments while adhering to predefined motion constraints, ultimately enhancing its overall performance and safety.

TEB incorporates kinematic and dynamic constraints to shape velocity and acceleration behavior according to the robot's physical limitations. These constraints encompass factors like maximum slope, turning radius, and collision avoidance. By considering these constraints during velocity and acceleration calculations, TEB ensures that the robot operates within its physical capabilities, promoting stable and effective motion in diverse environments.

The objective function in TEB often includes terms related to the shortest path, aiming to minimize the distance traveled or time taken to reach the destination. By incorporating the concept of the shortest path into the objective function, TEB seeks to optimize the robot's trajectory, ensuring that it follows the most direct route while avoiding obstacles and adhering to velocity and acceleration limits. It can be expressed as:

$$f_k = (\sum_{i=1}^{n} \triangle T_i)^2 \tag{2.8}$$

TEB process is visually presented in Algorithm 4.

Algorithm 4 Control Flow of TEB Algorithm

```
Initialization
while goal is not reached do
    while not best hyper-graph do
        Re-Initialization
        Insert/Delete TEB states
        Associate TEB states with waypoints/obstacles
        Generate hyper-graph
        Optimize hyper-graph
    end while
    Verify trajectory
    Calculate control variables v, w
    if Obstacle is detected then
        Go to state 6
    end if
end while
```

2.2.3 Dynamic Window Approach - DWA

DWA aims to find the ideal velocity (v, w) from a search space as shown in Figure 2.10 containing the obstacles-safe areas under the dynamic constraints of the robot and find the maximum velocity in this space using objective function.

To get the velocity search space as shown in the example Figure 2.11, circular trajectories (curvatures) consisting of pairs (v, w) of translational and rotational velocities are determined. Then, this space is constrained by the admissible velocity at which it can stop without reaching the closed obstacle on the corresponding curvature. Finally, this space is constrained by the dynamic window, which limits the admissible velocities to those achievable within a short time interval given the robot's limited accelerations. The objective function as described in Equation 2.9 is maximized [8].

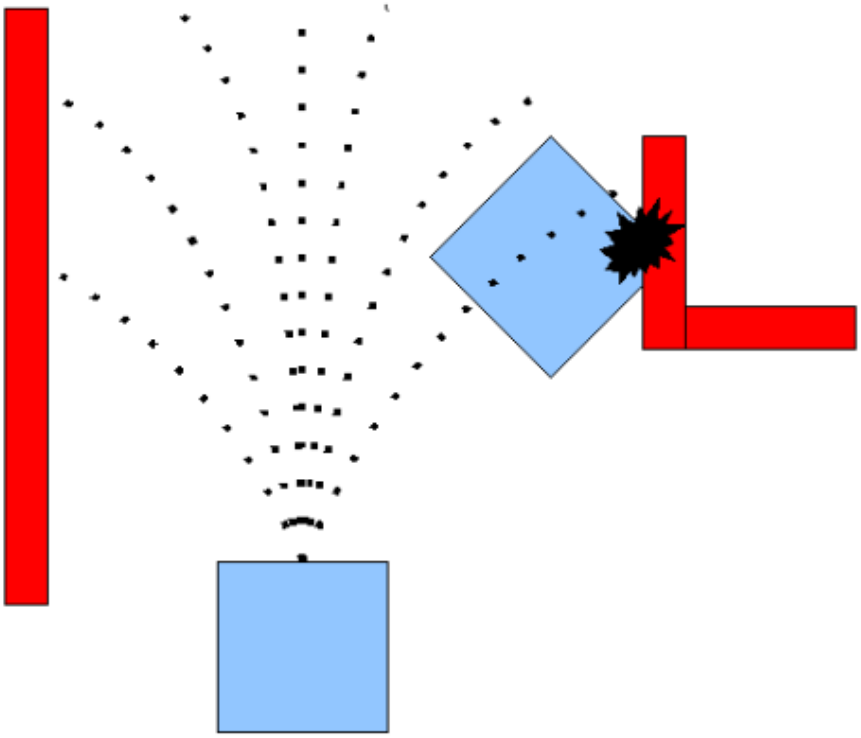


Figure 2.10 Trajectory roll-out based on v, w pair selection [34]

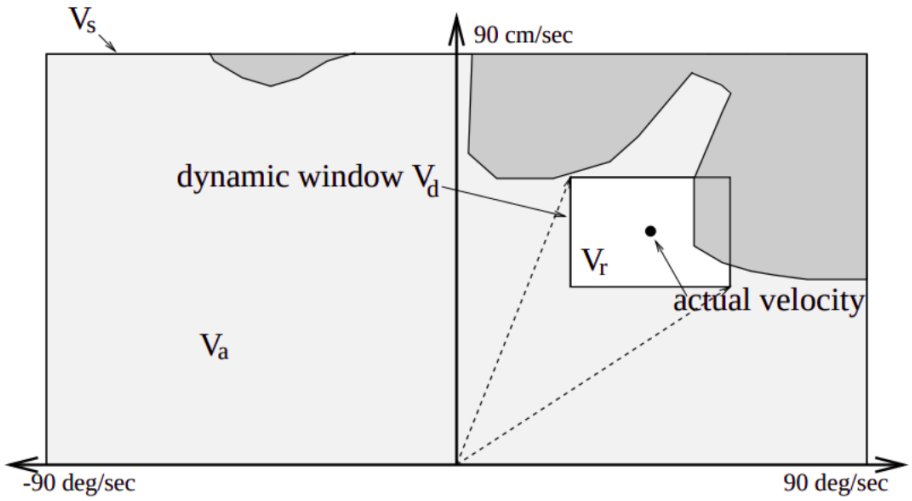


Figure 2.11 Velocity Space [8]

$$G(v, w) = \sigma(\alpha \cdot heading(v, w) + \beta \cdot dist(v, w) + \gamma \cdot vel(v, w)) \tag{2.9}$$

The target heading, $heading(v, w)$, measures the alignment of the robot with the target direction. It is maximal if the robot moves directly towards the target. The clearance, $dist(v, w)$, is the distance to the closest obstacle that intersects with curvature. $vel(v, w)$ function represents the forward velocity of robot. The DWA process is visually presented in Algorithm 5.

Algorithm 5 Dynamic Window Approach Algorithm

1: Define the initial position of robot and the goal position.
2: **while** the goal is not reached **do**
3: Calculate a valid velocity space based on the robot's current position and the goal position.
4: The dynamic window of the robot determines a valid velocity space based on the current position and the goal position.
5: Adjust this window to avoid collisions with obstacles considering their positions.
6: Choose the most suitable velocity from the adjusted velocity space.
7: Calculate the new position of the robot using the selected velocity.
8: **end while**

2.2.4 Reinforcement Learning: Overview

Finite Markov Decision Processes (finite MDPs) play a central role in the field of RL. RL is a type of machine learning paradigm in which an agent learns to interact with an environment to achieve a certain goal or maximize a cumulative reward. MDPs provide the mathematical framework for modeling the environment's dynamics, as well as the agent's interactions within it. MDPs consists of several key components: states, actions, rewards, transition probabilities and policies.

States S represent the different situations or conditions within the system, capturing relevant information about the environment. Actions A denote the possible decisions that can be made at each state, like moving, stopping, or turning. Rewards R are associated with state-action pairs, providing feedback on the immediate outcome of taking an action in a given state. These rewards guide the learning process towards achieving a specific goal, shaping the agent's decision-making. Transition probabilities p specify the likelihood of moving from one state to another after taking a particular action, reflecting the system's uncertainty and evolution over time. Policies π determine the action to take in each state, aiming to maximize cumulative rewards over time. These policies can be deterministic or stochastic, offering strategies for decision-making.

In RL, as described in [35], the agent and environment interact over a sequence of discrete time steps $t = 0, 1, 2, 3, ...$ At each time step t, the agent receives information about the current state of environment,denoted as $S_t \in S$ and selects an action $A_t \in A(S)$. One time step later, as a result of action, the agent receives a numerical reward $R_{t+1} \in R$ and transitions to a new state S_{t+1} as shown in Figure 2.12.

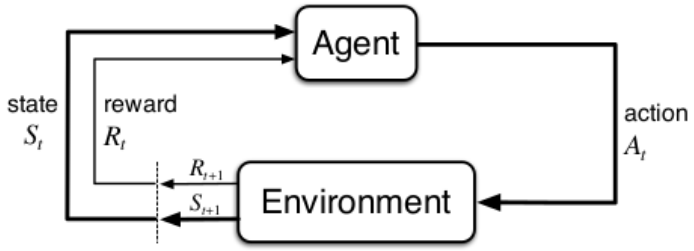


Figure 2.12 The agent–environment interaction in a Markov decision process. [35]

The agent's objective is typically to maximize the expected return. Return G is the cumulative sum of rewards over time, that is that it accounts for both immediate rewards and future rewards discounted by a factor γ. It represents the cumulative sum of all rewards obtained over a sequence of actions and states.Mathematically, this can be expressed as:

$$G_t = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k \cdot R_{t+k+1} \tag{2.10}$$

where γ , $0 \leq \gamma \leq 1$, is a discount rate. Therefore, the agent interacts with its environment, selecting actions based on received rewards to maximize its expected return. In this process, the agent often develops a strategy that takes into account future rewards, as sometimes long-term rewards might be preferred over short-term gains.

The state-value function, denoted as $v_\pi(s)$, represents the expected return that an agent can achieve from a given state s under a specific policy. Mathematically, the state-value function is defined as:

$$v_\pi(s) = E_\pi \left[G_t | S_t = s \right] = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k \cdot R_{t+k+1} \middle| S_t = s \right] \tag{2.11}$$

where $E_\pi[.]$ denotes expected value under policy π.

The Bellman equation for the state-value function relates the value of a state to the

values of its possible successor states under a given policy. Mathematically, it can be expressed as:

$$\begin{aligned} v_\pi(s) &= E_\pi\left[G_t|S_t = s\right] \\ &= E_\pi\left[R_{t+1} + \gamma \cdot G_{t+1}|S_t = s\right] \\ &= E_\pi\left[R_{t+1} + \gamma \cdot v_{t+1}|S_t = s\right] \end{aligned} \tag{2.12}$$

for all $s \in S$, where $\pi(a|s)$ probability of policy.

The action-value function, denoted as $Q(s, a)$, represents the expected return that an agent can achieve by taking action a in state s, followed by adhering to a specific policy thereafter. Unlike the state-value function, the action-value function considers both the state and the action. Mathematically, the action-value function is defined as:

$$q_\pi(s, a) = E\left[G_t|S_t = s, A_t = a\right] = E\left[\sum_{k=0}^{\infty} \gamma^k \cdot R_{t+k+1} \middle| S_t = s, A_t = a\right] \tag{2.13}$$

MDPs are typically solved by identifying an optimal policy that dictates the best action to take in each state to maximize expected return. Mathematically, the optimal state-value function v_* is defined as:

$$v_*(s) = \max_\pi \left\{v_\pi(s)\right\} \tag{2.14}$$

for all $s \in s$.

Mathematically, the optimal action-value function q_* is defined as:

$$q_*(s, a) = \max_\pi \left\{q_\pi(s, a)\right\} \tag{2.15}$$

for all $s \in s$ and $a \in A(s)$. To express $q_*(s, a)$ in terms of v_*, the optimal state-value function, we can utilize the relationship between $q_*(s, a)$ and $v_*(s)$ known as the Bellman equation. Specifically, this relationship can be stated as follows:

$$q_*(s,a) = E\left[R_{t+1} + \gamma \cdot v_*(S_{t+1}) | S_t = s, A_t = a\right] \tag{2.16}$$

Equation 2.14 can be rewritten referring to the above mentioned Equation 2.12, as follows:

$$v_*(s) = \max_a E\left[R_{t+1} + \gamma \cdot v_*(S_{t+1}) | S_t = s, A_t = a\right] \tag{2.17}$$

This Equation is the form of the Bellman optimality equation for v_*.

$$q_*(s,a) = E\left[R_{t+1} + \gamma \cdot \max_a q_*(S_{t+1}, a') | S_t = s, A_t = a\right] \tag{2.18}$$

This Equation also is the form of the Bellman optimality equation for q_*.

The estimation of the optimal value function is commonly done using Monte Carlo Methods or Temporal Difference (TD). In reinforcement learning, two prominent TD-based algorithms are SARSA (State-Action-Reward-State-Action) and Q-learning. SARSA and Q-learning are well-suited for solving complex problems in dynamic environments, where the iterative methods of dynamic programming, such as the value iteration algorithms, become impractical. These algorithms are particularly effective in scenarios where the agent interacts with the environment, updating its policy based on the observed states, actions, and rewards over time. SARSA focuses on on-policy learning, ensuring that the learned policy aligns with the current behavior strategy, while Q-learning is an off-policy algorithm, emphasizing the learning of an optimal policy irrespective of the agent's current strategy.

2.2.5 Q-learning

Q-learning is aims to help agents make optimal decisions in various environments such as robotic control, game playing, financial trading, and other dynamic systems where adaptive decision-making is crucial. At its core, Q-learning involves the concept of Q-values, which represent the expected rewards for taking a particular action in a specific state. The algorithm enables an agent to learn by interacting with the environment, choosing actions, receiving rewards, and updating Q-values based on these experiences as described in Equation 2.19. Through this iterative process of trial and error, the agent refines its decision-making abilities, ultimately selecting actions that lead to the highest cumulative rewards in different states of the environment.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a \{Q(s_{t+1}, a_t)\} - Q(s_t, a_t)) \quad (2.19)$$

where α is a learning rate ($0 < \alpha \leq 1$).

Selecting the current best action in the Q-learning algorithm often tends to approach learning to a local minimum/maximum. Therefore, exploration by the agent is also necessary. However, continuous exploration with a fixed probability reduces the learning performance. Hence, the probability of exploration needs to decrease over time [36].

2.2.6 Deep Q-Network - DQN

The Deep Q-Network (DQN) algorithm is a deep reinforcement learning method introduced by DeepMind in 2015 [21]. It is a powerful algorithm that combines principles of deep learning with Q-learning to tackle complex decision-making tasks in environments with high-dimensional state spaces.

Optimal Value function can be solved by dynamic programming (iterative methods) such as the value iteration algorithms, but it is impractical especially when the environment is complex. However, recently, utilizing neural networks which are nonlinear function approximator for estimation has become even more prevalent. In DQN, neural network function approximator with weights θ is used as Q-network. However, nonlinear approximator may cause RL to be unstable or even to diverge. As a solution to this, replay memory is used that updates iteratively the target values periodically towards action-values (Q). It can be implemented as a method, called experience replay, by saving the agent's experimences $e_t = (s_t, a_t, r_t, s_{t+1})$, at every step t into dataset $D_t = e_1, ..., e_t$. Thoughtout the Q-network's learning process, samples (or minibatches) are drawn randomly and uniformly from the dataset, denoted as $U(D)$. This mitigates the issues of correlated samples and allows the agent to learn from past experiences more effectively. Target networks are used to stabilize the learning process by providing fixed targets for the Q-value predictions. Periodically, the parameters of the target network are updated to match those of the primary network. The Q-network is trained by minimising a sequence of loss function $L_i(\theta_i)$ [37] that changes at each iteration i,

$$L_i(\theta_i) = E_{(s,a,r,s') \; U(D)}[(y_i - Q(s, a; \theta_i))^2] \quad (2.20)$$

where $y_i = E_{s'}[r + \gamma * max_{a'} Q(s^{'}, a^{'}; \theta_{i-1})|s, a]$. The parameters from previous iteration θ_{i-1} are held fixed when optimising the lost function $L_i(\theta_i)$. The loss function is derived from the Bellman Optimality Equation, which measures the difference between the predicted Q-values and the target Q-values. The target Q-values are obtained by bootstrapping from the target network and are calculated as the sum of the immediate reward and the discounted maximum Q-value for the next state.

DQN uses gradient descent to minimize the loss function. The gradients are computed with respect to the Q-network parameters, and the weights are updated to reduce the discrepancy between the predicted Q-values and the target Q-values as described in Equation 2.21.

$$\nabla_{\theta_i} L_i(\theta_i) = E_{(s,a,r,s')\ U(D)}[(r+\gamma*max_{a'} Q(s^{'}, a^{'}; \theta i - 1) - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i)] \tag{2.21}$$

where $\nabla_{\theta_i} L_i(\theta_i)$ denotes the gradient of the loss function $L_i(\theta_i)$ with respect to the parameters θ_i of the `i-th` critic network. $\nabla_{\theta_i} Q(s, a; \theta_i)$ denotes the gradient of the Q-value with respect to the parameters θ_i of the `i-th` critic network.

Instead of calculating complete expectations within the given gradient, it's frequently more computationally efficient to enhance the loss function using stochastic gradient descent. If the weights are updated after each time-step, and the expectations are substituted with single samples, it results in the well-known Q-learning algorithm.

To balance exploration and exploitation, DQN typically employs an ϵ-greedy strategy. With probability ϵ, the agent selects a random action to explore the environment, while with probability $1 - \epsilon$, it selects the action with the highest Q-value predicted by the Q-network.

By training the Q-network to approximate the optimal action-value function, DQN learns to make effective decisions in various environments, including playing Atari games, controlling robotic systems, and navigating complex environments. DQN has demonstrated impressive performance and has been influential in advancing the field of deep reinforcement learning.

The DQN process is presented in Algorithm 6.

In the DQN, a variety of different deep learning network architectures have been

Algorithm 6 DQN with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
for episode=1,M **do**
    Initialize sequence $s_1 = \{x_1\}$ and pre-processed sequence $\phi_1 = \phi(s_1)$
    for t=1,T **do**
        With probability ϵ select a random action a_t
        Otherwise select $a_t = argmax_a Q(\phi(s_t), a; Q)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and pre-process $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_j, a_j, r, \phi_{j+1})$ in D
        Sample random minibatch of transitions $(\phi_j, a_j, r, \phi_{j+1})$ from D
        Set $y_j = \begin{cases} r_j, & \text{if episode terminates at step } j+1 \\ r_j + \gamma \cdot \max_{a'} \hat{Q}(\phi_{j+1}, a'; Q^{-}), & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; Q))^2$ with respect to the network parameters Q
        Every C steps reset $\hat{Q} = Q$
    end for
end for

preferred. In some studies, a single-layer network architecture is used, while in others, much larger and advanced architectures are preferred. In Study [38], a fully connected neural network with one hidden layer of 80 neurons has been utilized. In another study [39], a fully-connected 2-layer neural network with 50 rectified linear units (ReLU) in each layer has been employed on 50 bootstrapped samples from the data. This architecture was chosen to capture complex patterns and relationships within the dataset. The first hidden layer, consisting of 50 ReLU units, is responsible for extracting relevant features from the input data. These features are then processed by the second hidden layer, also comprising 50 ReLU units, to further refine the learned representations. By using bootstrapped samples, the model is trained on multiple subsets of the data, allowing it to learn robust and generalized patterns. In the research conducted by Mnih et al. (2015) [21], the following architecture was utilized: The neural network takes as input an 84x84x4 image. The first hidden layer applies 32 filters of size 8x8 with a stride of 4 to the input image, followed by a rectifier nonlinearity. Similarly, the second hidden layer applies 64 filters of size 4x4 with a stride of 2, also followed by a rectifier nonlinearity. This is followed by a third convolutional layer with 64 filters of size 3x3 and a stride of 1, followed by a rectifier. The final hidden layer is fully connected, consisting of 512 rectifier units. The output layer is a fully connected linear layer with a single output corresponding to each valid action.

In DQN, it's important to emphasize that the action space is discrete. This means that the agent can only choose from a finite set of actions at each time step. Unlike continuous action spaces, where actions are represented by real-valued vectors, discrete action spaces involve selecting from a predefined set of options.

In the context of DQN, the discrete action space is typically associated with environments where actions correspond to specific discrete decisions or movements. For example, in Atari games, the agent's actions might represent pressing different buttons (e.g., move left, move right, jump) on the game controller.

This discrete action space simplifies the learning process, as the agent only needs to estimate the value of each action for a given state, rather than learning a complex policy for selecting from a continuous range of actions. It also allows for more straightforward implementation of Q-learning, as the Q-network outputs a separate Q-value for each possible action.

By discretizing the action space, DQN can effectively learn optimal strategies for decision-making in a wide range of environments, contributing to its success in various reinforcement learning tasks.

2.2.7 Deep Deterministic Policy Gradient - DDPG

The Deep Deterministic Policy Gradient (DDPG) algorithm, introduced by Lillicrap et al. in 2016 [40], stands as a pivotal method in the realm of deep reinforcement learning, specifically tailored to address challenges posed by continuous action spaces. DDPG adopts an actor-critic architecture, which consist of two neural networks: an actor network $\mu(s|\theta^{\mu})$ responsible for learning the policy—i.e., the action selection strategy—and a critic network tasked with learning the action-value function to evaluate actions chosen by the actor. Unlike stochastic policies, DDPG's actor network outputs deterministic actions given states, streamlining the learning process.

To enhance data efficiency, DDPG employs experience replay which involves storing past experiences in a replay buffer$(s_t, a_t; r_t, s_{t+1})$ and uniformly sampling batches during training. Furthermore, target networks are employed to stabilize training, with periodic updates of target network parameters based on the online networks. Loss functions are defined to train both actor and critic networks: the actor aims to maximize expected return by adjusting parameters to increase the critic's estimated value as described in Equation 2.23, while the critic minimizes temporal difference error between predicted and target Q-values using the Bellman equation as described in Equation 2.22.

$$L(\theta^Q) = E_{s_t \sim p^\beta, a_t \sim \beta, r_t \sim E}[(Q(s_t, a_t|\theta^Q) - y_t)^2] \tag{2.22}$$

where $L(\theta^Q)$ denotes the loss function associated with the critic network, parameterized by θ^Q. $E_{s_t \sim p^\beta, a_t \sim \beta, r_t \sim E}$ denotes the expectation over the states s_t sampled from the state distribution p^β, actions a_t sampled from the policy distribution β, and rewards r_t sampled from the environment E. $Q(s_t, a_t|\theta^Q)$ denotes the output of the critic network, representing the estimated value of taking action a_t in state s_t, parameterized by θ^Q. y_t denotes the target value for the critic, which is typically computed using a temporal-difference (TD) learning algorithm. It represents the sum of the current reward and the estimated value of the next state-action pair.

$$\begin{aligned} \nabla_{\theta^\mu} J &\approx E_{s_t \sim p^\beta} \left[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)} \right] \\ &= E_{s_t \sim p^\beta} \left[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|_{s=s_t} \right] \end{aligned} \tag{2.23}$$

where $\nabla_{\theta^\mu} J$ denotes the gradient of the loss function with respect to the parameters of a policy network denoted by μ. Here, J represents the loss function. $E_{s_t \sim p^\beta}$ denotes an expectation. The state s_t is sampled according to the distribution p^β. Here, p^β represents the distribution of the replayed experiences. $Q(s, a|\theta^Q)$ represents a value function corresponding to state-action pairs. It depends on a value network with parameters θ^Q. $\mu(s|\theta^\mu)$ represents a policy network that produces an action corresponding to a state. It has parameters θ^μ. $\nabla_{\theta_\mu} \mu(s|\theta^\mu)$ denotes the gradient of the value function with respect to the parameters θ^μ. $\nabla_a Q(s, a|\theta^Q)$ denotes the gradient of the value function with respect to the action.

Moreover, to facilitate exploration in continuous action spaces, DDPG frequently introduces noise sampled from the noise process into the actions selected by the actor as described in Equation 2.24.

$$\mu^{'}(s_t) = \mu(s_t|\theta_t^\mu) + N \tag{2.24}$$

where $\mu^{'}(s_t)$ denotes action conditioned on the state generated by a new policy network. $\mu(s_t|\theta_t^\mu)$ denotes action conditioned on the state generated by the original policy network, with parameters denoted by θ_t^μ. N denotes a noise or randomness

component. This can be used to enhance the exploration capability by adding some kind of noise to the output of the policy network.

An Ornstein-Uhlenbeck process [41] is employed to produce temporally correlated exploration as noise. An Ornstein-Uhlenbeck process is a stochastic process that models the behavior of a particle undergoing Brownian motion subject to friction. It is often used in the context of stochastic differential equations to model the dynamics of a system influenced by random noise and damping effects. The Ornstein-Uhlenbeck process is characterized by its mean-reverting behavior, where it tends to return to a central value over time due to the damping force.

The DDPG process is presented in Algorithm 7.

Algorithm 7 DDPG algorithm

1: Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
2: Initialize target network $Q^{'}$ and $\mu^{'}$ with weights $\theta^{Q^{'}} \leftarrow \theta^Q$, $\theta^{\mu^{'}} \leftarrow \theta^\mu$
3: Initialize replay buffer R
4: **for** episode=1,M **do**
5: Initialize a random process N for action exploration
6: Receive initial observation state s_1
7: **for** t=1,T **do**
8: Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise
9: Execute action a_t and observe reward r_t and observe new state s_{t+1}
10: Store transition $(s_i, a_i, r_i, s_{i+1)}$ in R
11: Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
12: Set $y_i = r_i + \gamma \cdot Q^{'}(s_{i+1}, \mu^{'}(s_{i+1}|\theta^{Q^{'}})$
13: Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
14: Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta_\mu} \mu(s|\theta^\mu)|s_i)$$

15: Update networks:

$$\theta^{Q^{'}} \leftarrow \tau \cdot \theta^Q + (1-\tau) \cdot \theta^{Q^{'}}$$
$$\theta^{\mu^{'}} \leftarrow \tau \cdot \theta^\mu + (1-\tau) \cdot \theta^{Q\mu^{'}}$$

16: **end for**
17: **end for**

2.2.8 Twin Delayed Deep Deterministic Policy Gradient - TD3

The Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm is an extension of the Deep Deterministic Policy Gradient (DDPG) algorithm, designed to further improve stability and sample efficiency in reinforcement learning tasks. Introduced by Fujimoto et al. in 2018 [42], TD3 builds upon the success of DDPG and addresses some of its limitations.

One crucial key enhancement introduced by TD3 is the utilization of twin critics. In traditional DDPG, a single critic is used to estimate the action-value function, which can sometimes lead to overestimation or underestimation of Q-values, resulting in suboptimal policies. TD3 addresses this issue by employing two separate critic networks, each independently estimating the Q-value. During training, the minimum Q-value from these twin critics is used as the target. This reduces the potential for overestimation or underestimation and improves the robustness of the learned policy, as described in Equation 2.25

$$y_1 = r + \gamma \min_{i=1,2} Q_{\theta_i'}(s^{'}, \pi_{\phi_1}(s^{'})) \tag{2.25}$$

where y_1 denotes represents the target value for updating the critic network. $\min_{i=1,2}$ denotes taking the minimum value among Q values from two twin critics. $\gamma \min_{i=1,2} Q_{\theta_i'}(s^{'}, \pi_{\phi_1}(s^{'}))$ represents the estimated action-value function for the next state $s^{'}$ and the action produced by the actor network. Here, $\theta_i^{'}$ are the target network parameters for the critic, and $\pi_{\phi_1}(s^{'})$ is the action produced by the actor network with parameters ϕ_1.

Another notable improvement introduced by TD3 is the use of target policy smoothing as described in Equation 2.26. This technique adds noise to the target policy during training, helping to regularize the learning process and prevent the policy from getting stuck in local optima. By smoothing the target policy, TD3 reduces the variance in Q-value estimates and improves the stability of the training process.

$$a^{'}(s^{'}) = clip(\mu_{\theta_{targ}}(s^{'}) + clip(\epsilon, -c, c), a_{\text{Low}}, a_{\text{High}}), \quad \epsilon \sim N(0, \sigma) \tag{2.26}$$

where $a^{'}(s^{'})$ denotes action produced by the target policy network, clipped within the range specified by Low and High. $\mu_{\theta_{targ}}(s^{'})$ denotes Action produced by the target policy network, with parameters denoted by θ_{targ} for the state $s^{'}$. ϵ denotes

Random noise sampled from a normal distribution with mean 0 and standard deviation σ. clip(x,min,max) denotes a function that clips the value x to be within the range specified by min and max. $-c$ and c denote lower and upper bounds for clipping the noise term ϵ, respectively.

Moreover, TD3 incorporates delayed updates for the policy and target networks. Instead of updating the policy and target networks at every time step, TD3 updates them less frequently, which helps to decorrelate the updates and stabilize learning. As a result, this delayed update mechanism enables TD3 to outperform traditional DDPG in terms of performance and sample efficiency.

The TD3 process is presented in Algorithm 8.

Algorithm 8 TD3

Initialize critic networks Q_{θ_1}, Q_{θ_1} and actor network π_ϕ with random parameters θ_1, θ_2 and ϕ
Initialize target networks $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer B
for t=1,T **do**
    Select action $a \sim \pi_\phi(s) + \epsilon, \quad \epsilon \sim N(0, \sigma)$ and observe reward r and new state s'
    Store transition tuple (s, a, r, s') in B
    Sample a random mini-batch of N transitions (s, a, r, s') from B
    $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim clip(N(0, \tilde{\sigma}), -c, c)$
    $y \leftarrow r + \gamma \cdot min_{i=1,2} Q_{\theta_i'}(s', \tilde{a})$
    Update critics $\theta_i \leftarrow argmin_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
    if t mod d **then**
        Update ϕ by the deterministic policy gradient:

$$\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$$

        Update networks:

$$\theta_i' \leftarrow \tau \cdot \theta_i + (1 - \tau) \cdot \theta_i'$$
$$\phi_i' \leftarrow \tau \cdot \phi_i + (1 - \tau) \cdot \phi_i'$$

    end if
end for

2.2.9 Curriculum Learning - CL

A basic approach for reinforcement based training is to create random targets without considering the difficulty level of the target and evaluating generated episodes. However while this approach enables exploring larger areas of the state

space, it also lengthens the generalization process of the learner. CL is proposed to sort the difficulty level of experience an agent acquires over time, in order to accelerate and improve learning time and generalization [43]. Although there are multi-agent automatic sequencing CL approaches our study relies on task-level CL. In task-level CL multiple tasks are defined and the agent is exposed to these tasks in a ascending difficulty order. In our study task-levels are defined as: path following for direct line of sight goal point, path following for goal point that is not in direct line of sight, path following for direct line of sight goal point with unseen obstacle during path planning is introduced and finally path following for goal point that is not in direct line of sight with unseen obstacle during path planning is introduced.

3
PROPOSED METHOD

Classical motion planning methods like Teb and DWA have some disadvantages and challenges. These methods rely on the assumption that all dynamics and obstacles in the environment are fully known. However, in the real world, this information may be incomplete or variable, which can reduce the reliability of plans. Additionally, these methods can be computationally intensive, making their use challenging in real-time applications. Effectively navigating obstacles is also among the challenges of these methods; algorithms like TEB and DWA may provide limited flexibility for maneuvering around obstacles, making it difficult to navigate effectively in complex, densely obstructed environments. DQN offers a way to overcome these challenges by leveraging deep learning techniques.

The objective of this research is to develop an innovative local planner for autonomous mobile robots using the DQN, DDPG and TD3 methods. Towards this goal, the initial step involved the establishment of the foundational components of the state space, followed by the evolutionary refinement of our model. In this process, inspiration was drawn from the angle calculation method in the Pure Pursuit algorithm. Consequently, the angle θ as shown in Figure 3.2a between the vector drawn from the robot's current position to the target point and the robot's orientation was identified as the initial element of the state space.

Route tracking can only be achieved with angle information. However, for a robot with constant linear velocity, the required angular velocity varies as it moves away from its path to reach the look ahead point. Similarly, keeping the angular velocity constant results in varying linear velocity. This circumstance has prompted the addition of the information about the robot's closest distance to its route alongside the angle value to the state space. Let's elaborate on an example scenario using Figure 3.1.

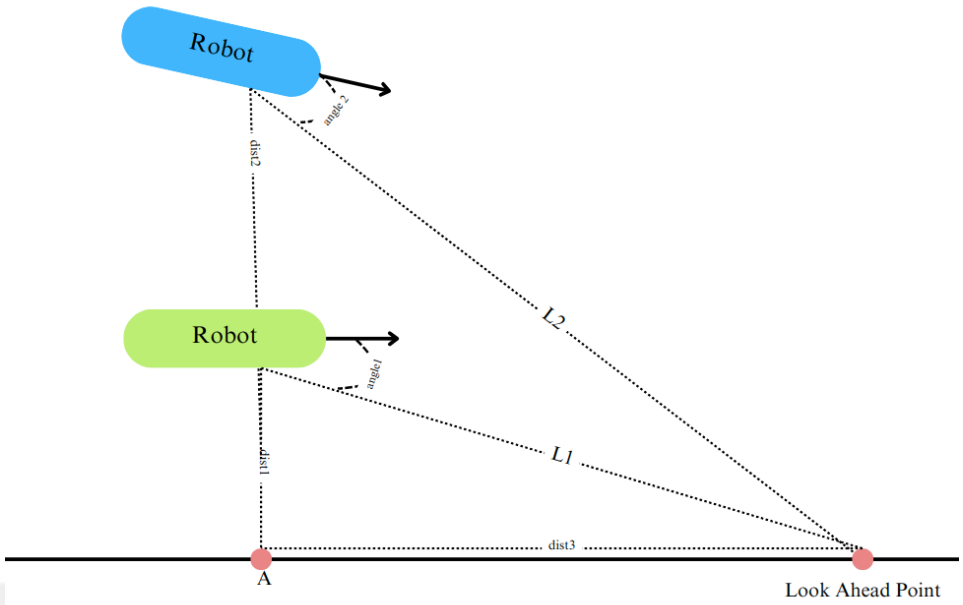


Figure 3.1

$$L1 = \sqrt{dist1^2 + dist3^2}$$
$$L2 = \sqrt{dist2^2 + dist3^2} \tag{3.1}$$

Let's define the variables:

- "dist1": The distance between the green robot and point A.
- "dist2": The distance between the blue robot and point A.
- "dist3": The distance between point A and the look ahead point.
- "angle1": The angle value, which is the first element of the state space for the green robot.
- "angle2": The angle value, which is the first element of the state space for the blue robot.
- "L1": The value calculated using Equation 3.1 in Figure 3.1.
- "L2": The value calculated using Equation 3.1 in Figure 3.1.

Let's create a scenario as follows:

- Both robots have a linear velocity of 1m/s.
- $angle1 = angle2$ and is set to 30 degree.
- $dist1 = 1m$, $dist2 = 0.3m$ and $dist3 = 0.4m$

It can be calculated the time it takes for the green robot to reach look ahead point with a certain angular velocity. A robot moving at a constant speed along a straight line will cover more distance over time.

Firstly, using Equation 3.1, $L1$ is calculated as $L1 = \sqrt{1^2 + 0.4^2}$, resulting in $1.077m$. We can calculate the time required for the robot to reach look ahead point by defining the vector that represents the movement from green robot's center to point look ahead point with a magnitude of $1.077m$. To find the time required to complete the path:

$$t = \frac{x}{v} = \frac{1.077m}{1m/s} = 1.077s \tag{3.2}$$

To complete the path from green robot's center point to look ahead point, the robot needs $1.077s$.

For the robot to complete its path from center point to look ahead point, it will need to perform a turning motion at a specific angle. To find the angular velocity of this turning motion, we can calculate the angle covered during the turning motion and the time required to cover this angle.

Angular velocity w is related to the angle covered (θ) in a unit of time.

$$\theta = w.t \tag{3.3}$$

The robot needs to turn $30\,\mathrm{deg}$ to reach look ahead point. Therefore,

$$w = \frac{\theta}{t} = \frac{30\,\mathrm{deg}}{1.077} = 27.855\,\mathrm{deg}\,/s \tag{3.4}$$

The robot needs to have an approximate angular velocity of $27.855\,\mathrm{deg}\,/s$ to reach look ahead point. Similarly, when calculating the angular velocity for the blue robot, it is $60\,\mathrm{deg}\,/s$.

As a consequence, it's observed that for identical angles, the requisite angular velocities vary across different positions. This discrepancy underscores the need for the robot to be responsive to such conditions. In response, an enhancement has

been introduced wherein the shortest distance to its designated route is incorporated as the second element of the state space, denoted as L as shown in Figure 3.2b. By integrating this parameter, the robot gains an enhanced awareness of its proximity to the intended path, thereby facilitating more adaptive and context-sensitive navigation strategies.

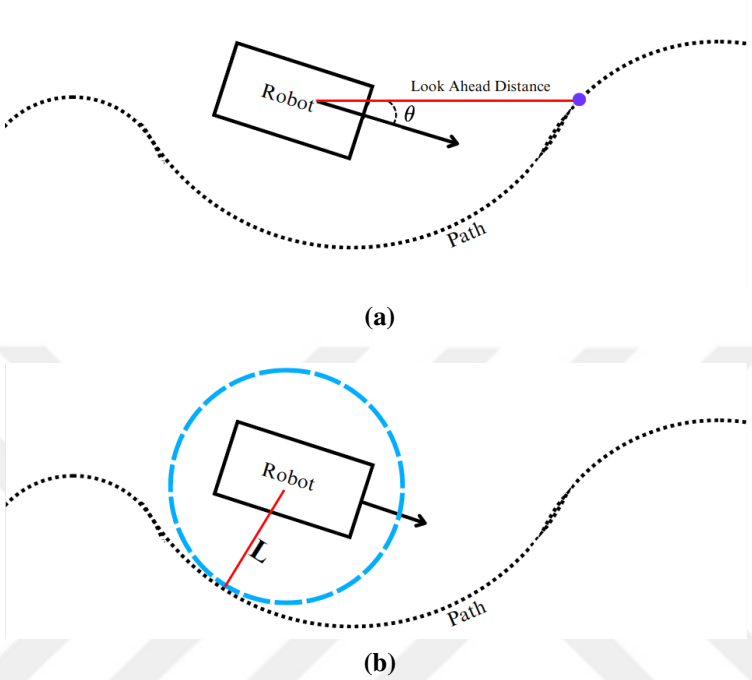


Figure 3.2 Two of the state space parameters a) angle difference between look ahead distance and robot heading, b) the distance (L) of the robot to its closest point on the global path

Using these two parameters, the robot can navigate alongside path towards its respective look-ahead points. However, their inability to detect obstacles along the path results in collisions. In order to prevent such collisions, an enhancement has been made by incorporating scan data into the state space. This augmentation enables the robots to perceive their surroundings comprehensively, thereby facilitating obstacle avoidance during route tracking.

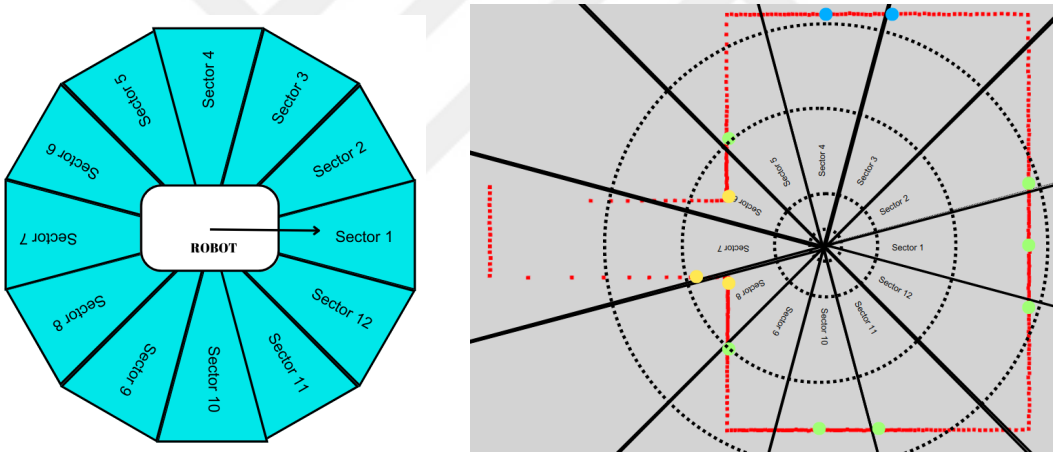
We have established the fundamental elements of the state space. Now let's examine the evolution and changes of our model throughout the process.

3.1 Model Evolution Process

In our initial model, called DQNS14A3, we established a state space defined across a 14-dimensional domains. To facilitate computation and decision-making, both Angle (θ) and Distance (L) have been discretized from a continuous state space. The discretization process involves sampling at a specific resolution, with the angle

resolution set at 0.35 radians and the distance resolution at 0.2 meters. These discretized parameters enable the robot to effectively perceive and navigate its environment with enhanced precision and efficiency.

The scan which is the fundemental of state stape refers to the sensor data obtained from the preprocessed LiDAR scans. In order to simplify and reduce the complexity of the state space, the 360-degree LiDAR data is segmented into 12 sectors, as shown in Figure 3.3a. Each sector encapsulates the most significant data point, which is typically the one closest to the robot's current position. This means that for each sector, the data representing it corresponds to the nearest obstacle or feature, as shown in Figure 3.3b. The resolution of these distances is determined based on the specifications outlined in Table 3.1. Consequently, the model gains the ability to refine its decisions with greater precision as it approaches obstacles. This adjustment allows the local planner to prioritize nearer obstacles when generating velocity commands, thus enhancing the robot's ability to navigate effectively and safely in its environment.



(a) **(b)**

Figure 3.3 a) sector tiling around the robot, b) smallest distances of LiDAR data on each sector

Table 3.1 The resolution table by distance

Min Distance(m)	Max Distance(m)	Resolution(m)
0.0	0.2	0.05
0.2	1.0	0.1
1.0	5.0	0.5
5.0	15.0	1.0
15.0	max range	5.0

Additionally, the model is equipped with three predefined actions: (i) moving forward at 1 m/s linear velocity, (ii) slowly turning right at -0.5 rad/s angular

velocity while moving forward at 0.3 m/s linear velocity, (iii) slowly turning left at 0.5 rad/s angular velocity while moving forward at 0.3 m/s linear, velocity. The agent is rewarded with an inverse ratio to the closest distance to the global path, an inverse ratio to the angular difference and a reward if the robot reaches the goal. However, a negative reward is given if it is 3m away from the global path, if it got too close to an obstacle or if the angle difference is more than 2.44 radians. Expect for the first two rewards, the episode ends in other positive/negative reward cases.

Finally, a network architecture for the DQNS14A3 agent consists of three fully connected layers, each with a rectified linear activation function. These layers consist of 64, 128, 32 and 3 output layers neurons as shown in Figure 3.4.

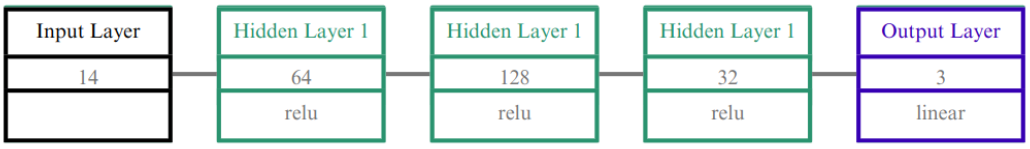


Figure 3.4 Network architecture of the DQNS14A3 model

After completing its training, the DQNS14A3 model gained the ability to successfully navigate obstacles and achieve the target with an 80% success rate. This achievement serves as a significant indicator of the fundamental capabilities of the model. However, upon conducting a detailed analysis, we noticed that this model exhibited low sensitivity when encountering obstacles and had a limited action space. This limitation restricted the model's ability to respond effectively to complex or unexpected obstacles. Therefore, we initiated a new endeavor to further enhance the performance of our current model and enable it to respond more effectively to complex scenarios with a broader action space. The aim of this new model was to effectively navigate obstacles and possess a wider action space. These efforts represent a crucial step towards enabling the robot to better understand its surroundings and overcome obstacles more effectively.

In the DQNS14A3 model, the angle and distance values perceived by the robot were processed with a certain resolution, and these values were represented in a discrete space. For instance, the perceived angles were grouped into certain intervals, and any value within these intervals was assigned to the corresponding interval. Similarly, distance values were processed with a specific resolution, meaning they were divided into certain intervals, and a value was assigned to each interval. However, we realized that this approach had limited accuracy and precision.

In our new model, called DQNS363A15, we changed this approach and began processing angle and distance values continuously. This means that the angles and

distances perceived by the robot are now processed seamlessly without any interval or grouping. This allows us to create a more precise and accurate model because instead of limiting any value to specific intervals, we can process the entire range of values. The scan data was segmented into sectors, and the data closest to the robot was used in each sector. Later, these data were discretized in a discrete space. In the DQNS363A15 model, continuous 360-degree scan data is used in continuous space. This allows the entire 360-degree scan data to be directly evaluated by the model without the need for any sampling process. Nevertheless, these data are constrained within a 3-meter range. This implies that the robot does not process scan data beyond 3 meters, thereby examining obstacles in its vicinity with greater detail within this 3-meter range. This approach minimizes data loss and enables the model to assess environmental information more accurately. Using scan data in continuous space allows the robot to perceive its surroundings more precisely and make accurate decisions, thus enhancing navigation performance.

Additionally, a new parameter has been added to the state space. Based on observations made in the DQNS14A3 model, it was noted that the robot's primary objective is not simply to reach the final destination point, but rather to track a way point known as the "look ahead point." This observation led to the realization that the robot tends to perceive its ultimate goal as akin to an intermediate point. The model architecture can be shown in Figure 3.5. Therefore, in the DQNS363A15 model, a new parameter has been introduced to ensure that as the robot approaches the final destination, its behavior adapts and it focuses on the target. This parameter allows the robot to assess its distance and proximity to the final destination, enabling it to adjust its behavior accordingly. Consequently, by enhancing its focus on the final destination, the robot can navigate more effectively and efficiently.

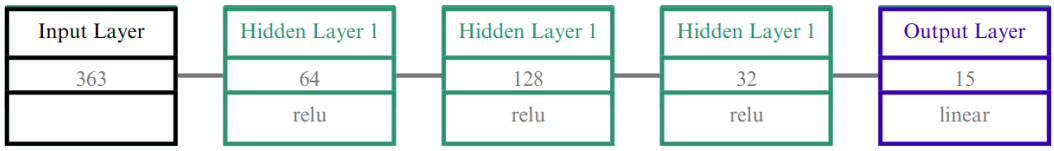


Figure 3.5 Network architecture of the DQNS363A15 model

Furthermore, the reward and penalty mechanisms have been restructured to further enhance the model's performance. In the event that the robot successfully reaches its destination, a significant positive reward of +10000 is bestowed upon it. This generous reward is specifically designated to underscore the importance of the robot successfully completing its task and reaching the intended destination. Conversely, if the robot collides with an obstacle, it incurs a penalty of -4000. Similarly, exceeding the predefined limits for parameters such as L or theta also results in a penalty of -4000. These penalties are implemented to deter the robot from engaging

in careless or erroneous behavior and to promote safe navigation. In addition, as the robot deviates from the path, a reward/punishment mechanism similar to the one shown in Figure 3.6 has been incorporated. In this system, the amount of reward or penalty varies depending on how far the robot deviates from the path. For instance, if the robot strays a certain distance from the path, the reward decreases or the penalty increases accordingly. This ensures that appropriate incentives or deterrents are provided to prevent the robot from deviating from the path and to encourage it to maintain its course.The reward is given according to the function in Equation 3.5. The minimum value that L can take is 0 meters. For this value, a reward of +1 is given. The maximum value, on the other hand, is 1.5 meters, and a penalty of -6.5 is applied for this value. In between these values, the reward or penalty varies linearly. This reward/punishment system helps guide the robot more accurately and effectively towards its goal by minimizing deviations from the path.

$$f(x) = 10 \cdot (-0.5 \cdot x + 0.1) \tag{3.5}$$

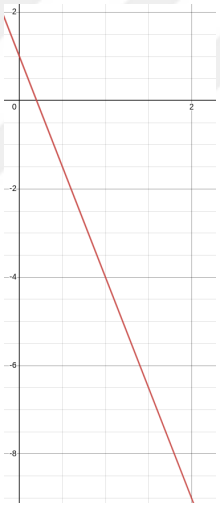


Figure 3.6 Rewad graph of the Distance(L)

The angle value ranges from $-\pi$ to $+\pi$. Within this range, the closer the angle value approaches 0, the greater the reward it receives.The reward is given according to the function in Equation 3.6. As shown in Figure 3.7, at the 0 point, the maximum reward value is obtained, which is +0.27. However, at the minimum point of the angle value (-1.57), which corresponds to the negative maximum angle, a penalty of -2.195 is incurred. Additionally, as the angle value approaches 0, the reward amount increases, incentivizing the robot to progress in the correct direction and stay closer to the path.

$$f(x) = (-x)^2 + 0.27 \tag{3.6}$$

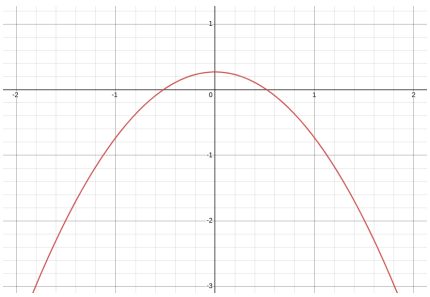


Figure 3.7 Rewad graph of the Theta(θ)

As the robot approaches the final target, it earns a reward as shown in Figure 3.8.The reward is given according to the function in Equation 3.7. If the robot gets within 0.3 meters of the target, it is considered to have reached the goal. Hence, the minimum value for this parameter is set to 0.3, while the maximum is 1.5. A reward of +10 is granted for reaching the minimum value, whereas +2 is awarded for reaching the maximum value.

$$f(x) = \frac{2 \cdot \text{look_ahead_dist}}{x} \tag{3.7}$$

Expanding upon the 3-action space in the DQNS14A3 model, we have enlarged the action space in the DQNS363A15 model to encompass 15 distinct actions. This expanded action space enables the robot to execute more precise and varied movements at different speeds and angles. We have designated various values for linear speed, such as 0.33, 0.66, and 1 m/s, and for angular speed, including -0.4, -0.2, 0, 0.2, and 0.4 rad/s. These diversified speed values enhance the robot's maneuverability, aiding it in coping with more complex scenarios.

After defining the linear and angular speed values, we generated 15 unique actions by combining these values in various combinations. These actions represent different permutations of linear and angular speed values, allowing the robot to respond more effectively to its surroundings. Consequently, the robot gains a broader range of motion to navigate obstacles effectively and reach its target with greater speed and accuracy. This expanded action space empowers the robot to explore its environment more flexibly and adaptively, facilitating efficient movement in diverse scenarios.

Finally, the network architecture of the DQNS363A15 model remains the same as the DQNS14A3 model. However, the input and output values have been adjusted

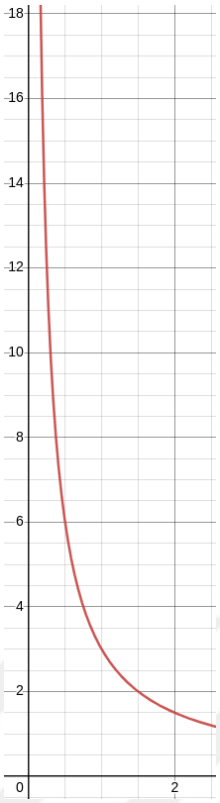


Figure 3.8 Rewad graph of the new parameter of state space(θ)

to align with the state space and action space. The network architecture for the DQNS363A15 agent consists of three fully connected layers, each with a rectified linear activation function. These layers consist of 64, 128, 32 and 15 output layers neurons.

Positive results have been obtained from the DQNS363A15 model, which has led us to focus more on the continuous action space. The aim was to train methods capable of managing continuous action spaces such as DDPG and T3D under the same conditions. However, slight adjustments in reward values were made due to the sensitivity of these methods to hyperparameters. Instead of the +10000 positive reward given upon reaching the target, a reward of +400 is now provided. Similarly, instead of the -4000 penalty for collision, a penalty of -50 is applied. Additionally, when exceeding the limits of L and Theta, a penalty of -200 is imposed instead of the previous -4000 penalty. We named the DQN model trained with these parameters as DQNS363A15V2.

In methods DDPG and TD3, the actions are continuous, allowing for the adjustment of parameters required for robots to perform specific actions. In both of these methods, the action space consists of two primary dimensions: linear velocity and angular velocity.

Linear velocity represents a robot's ability to move in a straight line. This could indicate the forward or backward movement speed of a robot, for instance. The range of values for this dimension of the action space typically falls between 0.1 m/s and 1 m/s. This range ensures that the robot moves at a sufficient but controllable speed, preventing it from moving too quickly or too slowly.

Angular velocity, on the other hand, represents a robot's rotational movement capability. This could indicate the rotation speed or the rate of change of orientation of a robot. The second dimension of the action space usually ranges from -0.4 units per second to 0.4 units per second. Negative and positive values represent the robot's ability to turn in opposite directions. This range allows the robot to turn in the desired direction while minimizing the risk of excessive rotation or unstable movement.

In this way, algorithms like DDPG and TD3 control the robot's movement through these continuous parameters in the action space, ensuring that it behaves safely and stably within the specified limits.

In these methods, two different networks are used: actor network and critic network. The hidden layers of both networks are similar to those of the DQNS363A15V2 model, consisting of 64, 128, and 32 neurons, respectively. The 'relu' activation function is used in all hidden layers.

The input layer of the critic network includes both the state space and the actions. With a state space of 363 dimensions and 2 actions, this results in a total of 365 neurons in the input layer. The output layer predicts the Q values and hence consists of only 1 neuron. The 'linear' activation function used in output layer. The critic network of DDPG and TD3 can be shown in Figure 3.10.

The input layer of the actor network receives only the state space, containing 363 neurons. The output layer predicts the actions, hence it has 2 neurons. A 'tanh' activation function is used in the output layer, which produces values between -1 and +1. To ensure compliance with the limits of our actions, the function in equation 3.8 has been applied to the output layer. The actor network of DDPG and TD3 can be shown in Figure 3.9.

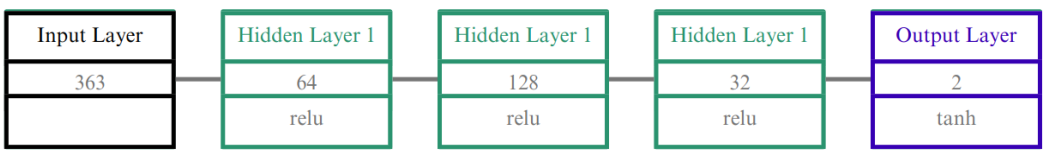


Figure 3.9 Actor Network architecture of the DDPG and TD3 model



Figure 3.10 Critic Network architecture of the DDPG and TD3 model

$$f(x) = \text{lower_bound} + 0.5 \cdot (x + 1.0) \cdot (\text{upper_bound - lower_bound}) \quad (3.8)$$

where x denotes output layer data and lower_bound, upper_bound denote action limits.

4
EXPERIMENTAL SETUP

In this chapter, the features of the simulation environment and the real robot platform used for testing the models will be described. Additionally, the training processes of the models will be examined in depth. Finally, the necessary scenarios and evaluation metrics for assessing the models will be discussed.

4.1 Simulation Environment

The simulation environment is a critical component for testing the models. This environment is used to simulate real-world conditions and evaluate the performance of artificial intelligence models. In the simulation environment, various environments/worlds can be created, allowing us to test the generalization capabilities of the models. Compared to a real robot platform, the simulation environment is faster and more cost-effective. Additionally, it allows for the easy creation and repetition of various scenarios.In our study, the Gazebo simulator, mentioned in the Chapter 2, has been preferred for both model training and testing. Additionally, there is a need for a robot operating in the simulation environment to train and test the model. In this study, the waffle model of Turtlebot3 has been preferred as shown in Figure 4.1.

The Turtlebot3 robot is a type of differential wheeled robot, utilizing two wheels independently driven and located on opposite sides of its body to facilitate movement. By adjusting the rotation rate of each wheel independently, it can effectively change its direction. The drive control is carried out by `gazebo_ros_diff_drive` plugin with built in Proportional Integral Derivative (PID) speed controllers. There is a LiDAR sensor on the robot that can scan its surroundings at 1-degree intervals, covering 360 degrees. It's footprint can be given as the following rectangle [[-0.205, -0.155], [-0.205, 0.155], [0.077, 0.155], [0.077, -0.155]], where [0,0] can be considered as the geometric center and center of mass of the robot. Messages passed together with source and sinks for those messages

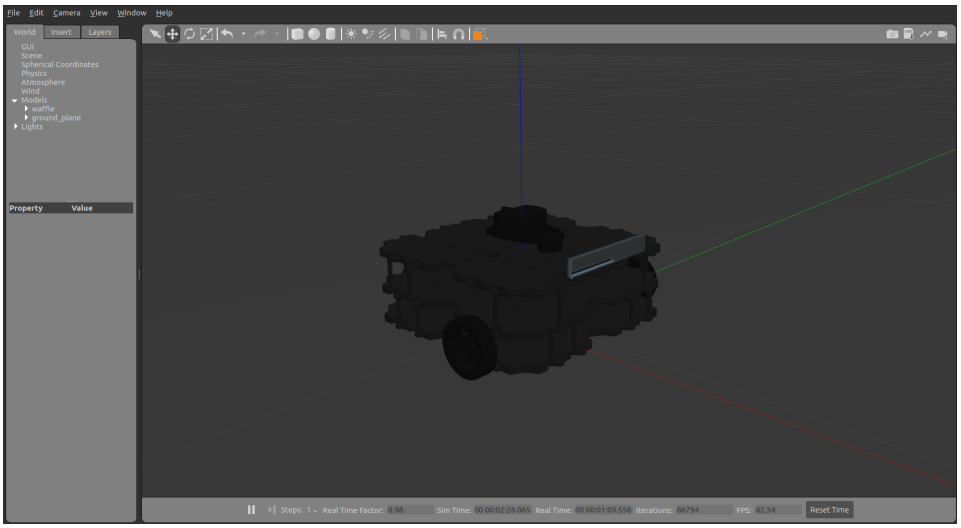


Figure 4.1 Waffle model of Turtlebot3

covering the whole simulation environment and the ROS nodes can be summarized with `rqt_graph` tool as shown in Figure 4.2.

The Turtlebot3 robot is both physically available and has a model in the gazebo environment. It is also compatible with ROS. ROS, which we also use as our experimental setup, is a widely used open-source software for mobile robots. It allows complex algorithms developed for topics that form the basis of robotics such as mapping, localization, navigation used in robots to communicate with each other. In our study, we used `gmapping` for mapping, `amcl` for localization and `move_base` for navigation. As mentioned in Chapter 2, `move_base` is a ROS package that combines both global planning and local planning functionalities, allowing the robot to generate a general path plan to reach a destination (global planner) and enabling it to navigate adaptively to environmental conditions by executing this plan (local planner). With reference to this page[44], it is possible to add global and local planners in ROS. After completing the training of our model, we added it as a ROS plugin.

Built-in A* in ROS is used as the global planner for robot navigation. Figure 4.3 shows a path generated using `A*`, depicted in green. TEB, DWA, and our own models have been used as local planners.

We utilized `gym`, the standard API for RL, to observe the environment and determine the next action. The fundamental features of the "gym" library include providing various environments for training and evaluating various artificial intelligence algorithms. These environments simulate spaces where agents such as artificial intelligence models or robots can interact. Additionally, there is a "Reset" method that resets the "gym" environment to its initial state, ensuring that

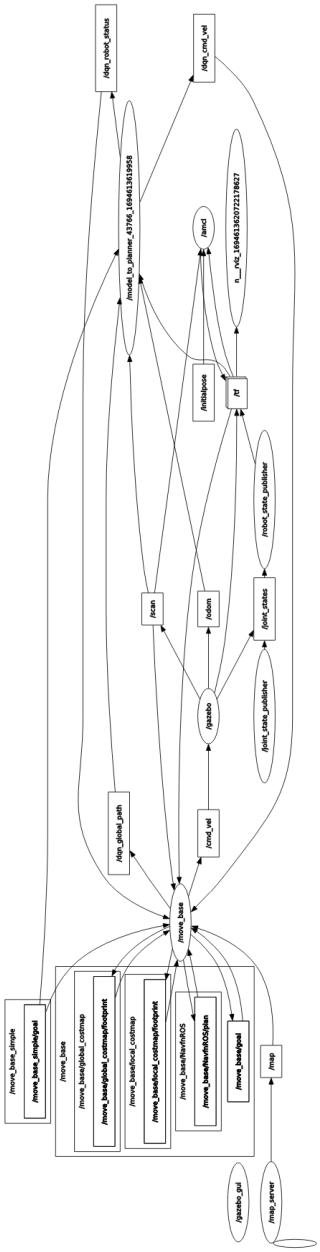


Figure 4.2 ROS nodes and messages summarized for the simulation environment

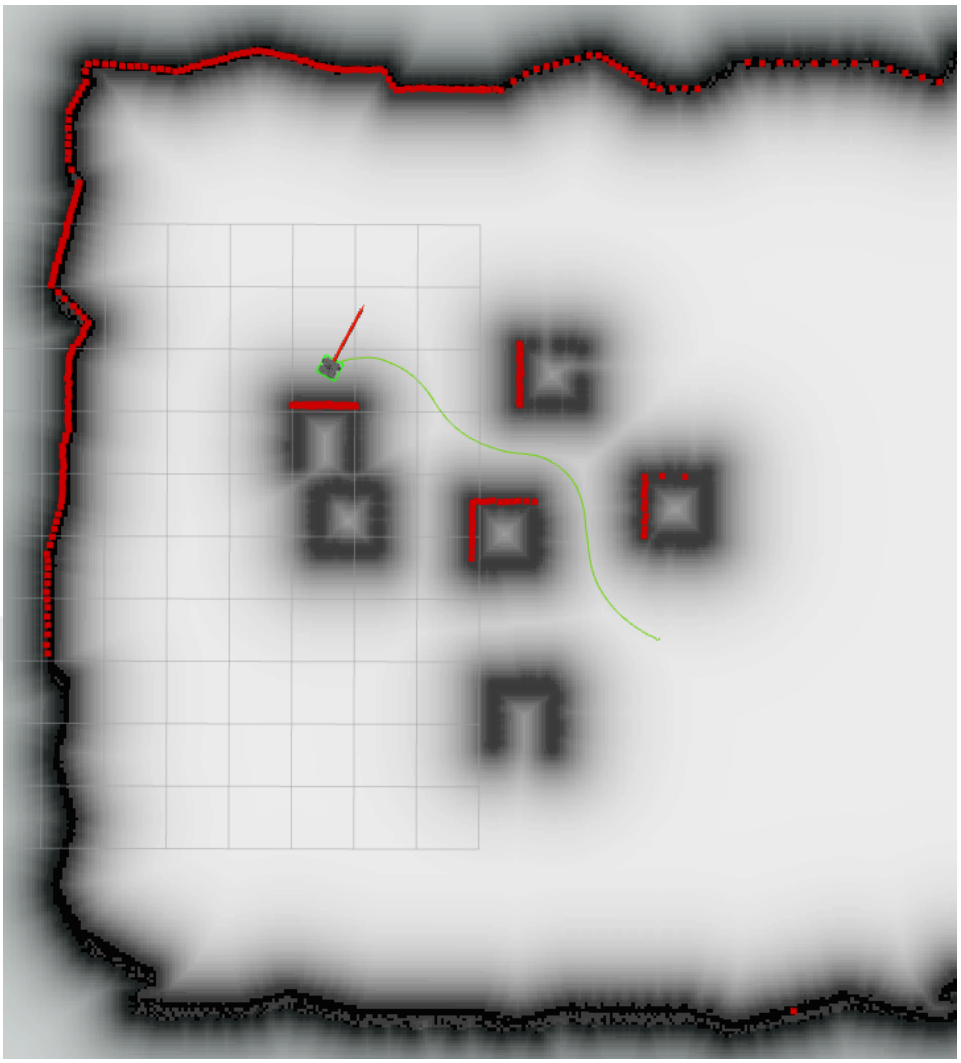


Figure 4.3 An example output of A* as viewed in RViz

the environment returns to its starting state at the beginning of each training loop. Another essential feature is the "Step" function, which enables an agent to take actions within the environment. This function allows the agent to interact with the environment by selecting and executing actions. The "Render" function enables the visualization of the environment and can be used for visualizing training progress or results. Finally, each "gym" environment has specific action and observation spaces. These spaces define how the agent interacts with the environment and the information it receives from the environment. Known as the Action Space and Observation Space, these spaces determine the actions the agent can take and the information it can perceive to make decisions.

Additionally, We employed `openai_ros` [45] package to facilitate communication between `gym` and ROS. This package facilitates the use of various machine learning algorithms in robotic projects, enabling robots to perceive their surroundings, move, and interact. For example, it can be used to train a robot to recognize and manipulate objects, map its environment, or navigate to a target.

We used the `TensorFlow Keras` library to create neural network. TensorFlow Keras is a component of the TensorFlow library and is a high-level deep learning API. Keras is an easy-to-use deep learning library that simplifies tasks such as model creation, training, evaluation, and deployment.

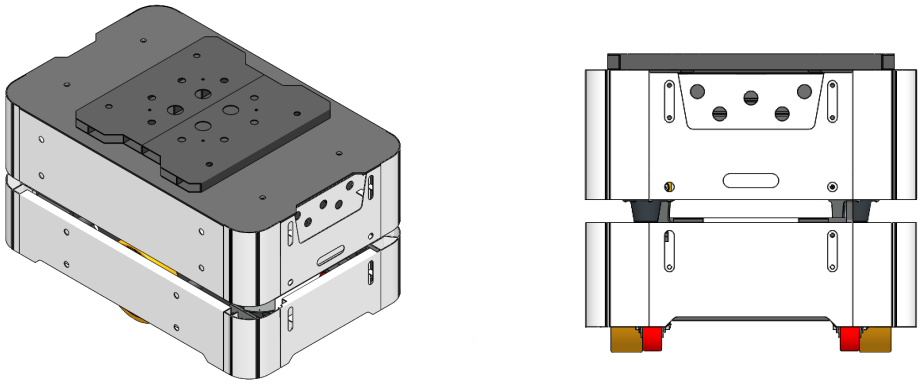
TensorFlow Keras provides a set of predefined layers and model structures for building artificial neural networks and deep learning models. These layers can be used to create successive neural network layers and easily define different types of neural network architectures.

Being integrated with TensorFlow, Keras can leverage TensorFlow's powerful computing capabilities and distributed training features. Additionally, Keras can accelerate model training by utilizing high-performance computing resources such as GPUs and TPUs.

4.2 Real Platform and Environment

The real platform is a commercially available carry robot. Just like the simulated Turtlebot3, real platform has a differential drive model as shown in Figure 4.4. It has two drive wheels on its longer edge centers and four caster wheels at its corners. The robot is equipped with two LiDAR sensors each having 270-degree field of view and they are placed at the two diagonal corners as shown in Figure 4.5. Data from these sensors is combined relative to the robot's center and integrated into the

system as a single LiDAR data covering the whole 360-degree field of view. It's footprint can be given as the following rectangle [[0.4068, -0.24], [0.4068, 0.24], [-0.4068, 0.24], [-0.4068, -0.24]], where [0,0] can be considered as the geometric center. It can reach a maximum speed of 1.5 m/s.



(a) (b)

Figure 4.4 Models of robot used in real platform a) Perspective, b) Front side

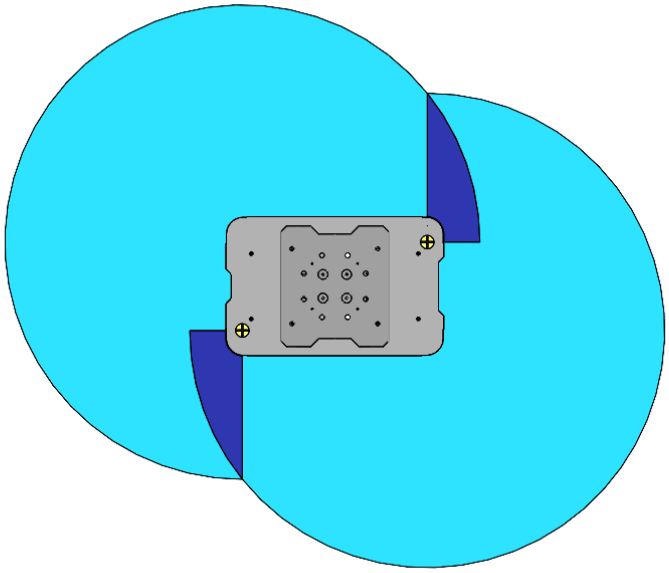


Figure 4.5 Visualization of scan sensor's FOV(Field of View)

The environment as shown in Figure 4.6 is an office environment with work desks and chairs distributed orderly. In addition, foam obstacles have been placed in the environment to force the robot to perform complex paths.

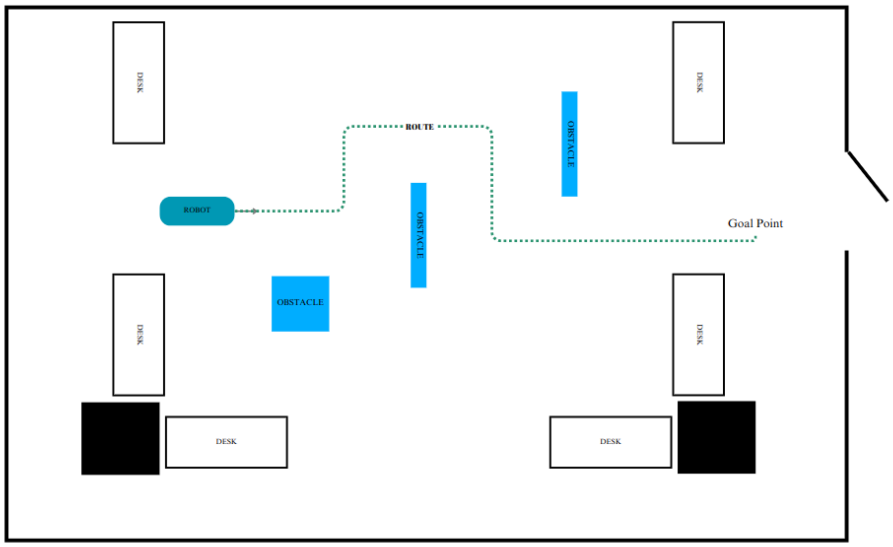


Figure 4.6 Office environment (representative)

4.3 Training

In the Chapter 3, we examined two different DQN models operating in a discrete action space, DDPG and TD3 methods capable of producing outputs in a continuous action space. Each method was detailed regarding its state space, action space, reward structure, and network architecture. Now, we will delve into the training processes of these models one by one.

In the DQNS14A3 model, as discussed in the Section 2.2.9 task-level learning starting from fixed position of goal and obstacle to randomly selected in an area that expanded over time is carried out in order to have the learner converge faster. A (1,3,1) size box is placed in the center of the empty space as an obstacle. In each episode, the robot is started from the point (-3.9, 0) and is expected to navigate to the goal at (3,0). In this way, the model is trained for about 4000 episodes. At the end of 4000 episodes, it is observed that the robot mostly navigates to the goal point. As the second stage, the location of the obstacle and goal point is changed in each episode. The obstacle is randomly generated between (-0.5, 0.5) on the x-axis and (-2,2) on the y-axis. Path given to system as a global path is the straight line between the start point and the target point with 0.2 meters way-points. In this way, the model is trained for about 2000 more episodes. Finally, the obstacle and goal area is expanded and the model is trained for about 4000 more episodes. The trained model is integrated into ROS as local planner with the name `dqn_local_planner`. To generalize the training, the DQNS14A3 model is augmented with the replay memory feature as mentioned Section 2.2.6. In the model, the replay memory size is set to 2000, and training is conducted at the end of each episode. Scenarios given

visually in Figure 4.7 and step by step in Table 4.1 are created to test our model and to compare it with `teb_local_planner` and `dwa_local_planner`. The test scenario given in Figure 4.7a introduces a static obstacle at different locations that was unseen during path planning. The scenario given in Figure 4.7b introduces a dynamic obstacle, which was unseen during path planing and also that moves continuously perpendicular to the planned path. The scenario given in Figure 4.7c introduces a dynamic obstacle, which was unseen during path planing and also moves towards the robot. Finally the scenario given in Figure 4.7d introduces a dynamic obstacle, which was unseen during path planing and also moves away from the robot.

Table 4.1 Test scenarios

Scenario Name	Scenario Steps
Scenario1	Place the robot at point (-3.9,0.0) Give a point a) (2.0,0.0) as goal b) (4.0,0.0) as goal c) (11.0,0.0) as goal Trigger global path planning using 'navfn' Place a (1.0,3.0,1.0) sized box at (0.0, 0.0) as a fixed obstacle
Scenario2	Place the robot at point (-3.9,0.0) Give point (4.0,0.0) as goal Trigger global path planning using 'navfn' Place a (1.0,3.0,1.0) sized box at random $y \in [-2.0, 2.0]$ and fixed x=0.0 and let the obstacle move repeatedly between –2.0 and 2.0 in y-axis with a speed of 0.2 m/s
Scenario3	Place the robot to point (-3.9,0.0) Give point (10.0,0.0) as goal Trigger global path planning using 'navfn' Place a (1.0,3.0,1.0) sized box at random $x \in [-2.0, 8.0]$ and fixed y=0.0 and let the obstacle first move away from the robot and then repeatedly move between –2.0 and 8.0 in x-axis with a speed of 0.1 m/s
Scenario4	Place the robot at point (-3.9,0.0) Give point (10,0.0) as goal Trigger global path planning using 'navfn' Place a (1.0,3.0,1.0) sized box at random $x \in [0.0, 8.0]$ and fixed y=0.0 and let the obstacle first move towards the robot and then repeatedly move between 8.0 and 0.0 in x-axis with a speed of 0.1 m/s

Upon achieving successful results with the DQNS14A3 model, we aimed to enhance it by expanding both the state and action spaces. Consequently, our new

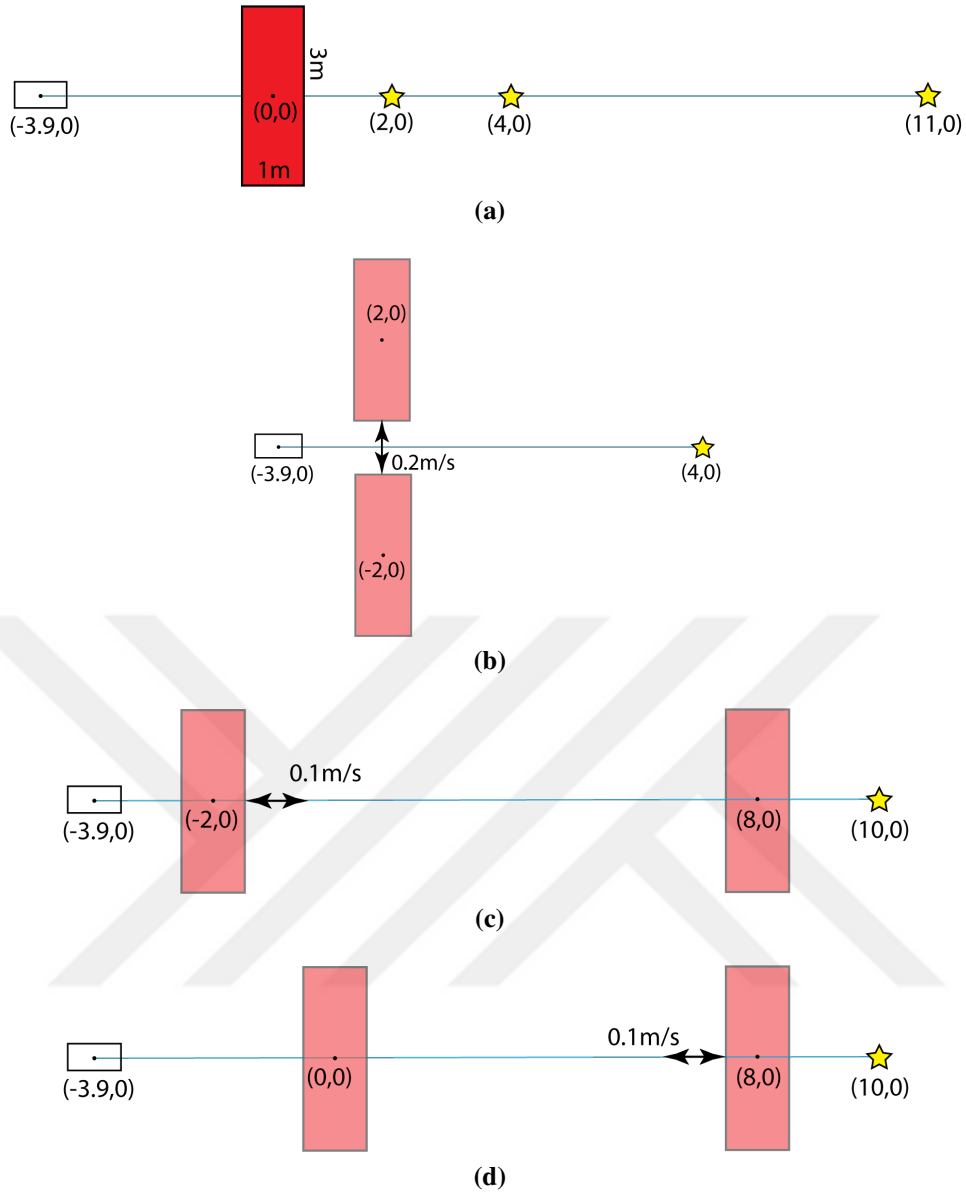


Figure 4.7 2D representation of scenarios a) Scenario1, b) Scenario2, c) Scenario3, d) Scenario4

model, DQNS363A15, adopted a more complex structure. However, we noticed that the training process became considerably slower due to this complexity. To address this issue, we decided to perform training after each step rather than after each episode. Additionally, we adjusted the batch size to 64 for training. Moreover, to facilitate generalization, we increased the size of the replay memory to 50,000. These adjustments were made to improve the efficiency and effectiveness of our model's training process. In classic DQN code, during the training process, predictions for the next action are made for each element in the minibatch using the target network, followed by training the model after a series of operations. However, we noticed that this prediction and training procedure significantly slowed down the process. Therefore, to perform the prediction and training steps in a single step, we provided the elements in the minibatch to the deep learning network as an array. As a result, our process became at least 64 times faster.

Firstly, we trained the DQNS363A15 model in a square area of 10x10m^2 as seen in Figure 4.8 for approximately 750 episodes. In each episode, the robot started from the position (-4.0, 0.0) and attempted to reach the position (4.0, 0.0). Since there were no obstacles in the environment, its path was straight.

Next, we continued the training of the model in a 20x20m^2 square area as shown in Figure 4.9. In each episode, we placed the robot at the position (-8.0, 0.0) and set the position (8.0, 0.0) as the target. The aim of this training was to measure the path following performance over a longer route. We continued this training for approximately 1300 episodes.

In the third phase, we randomly assigned a target point as shown in Figure 4.10 within the range [7.0, 9.0] on the x-axis and [-8.0, 8.0] on the y-axis for each episode. We trained the model in this manner for approximately 750 episodes.

In the last phase, we trained the model in an environment where there was an obstacle with a width of 1m along its path as shown in Figure 4.11. At the beginning of each episode, the robot started from the position (0.0, 0.0) and attempted to reach the position (13.0, 0.0). We continued this training for approximately 250 steps. We observed that the robot successfully reached the target without colliding with the obstacle, and thus concluded the training.

Finally, training was conducted for the DDPG and TD3 models, and the DQNS363A15V2 model was trained based on the parameters of these models. Training was performed with all parameters used in these three models being the same. The simulation environment and processes used during the training of these three models were identical. In the world as shown in Figure 4.11, the robot was

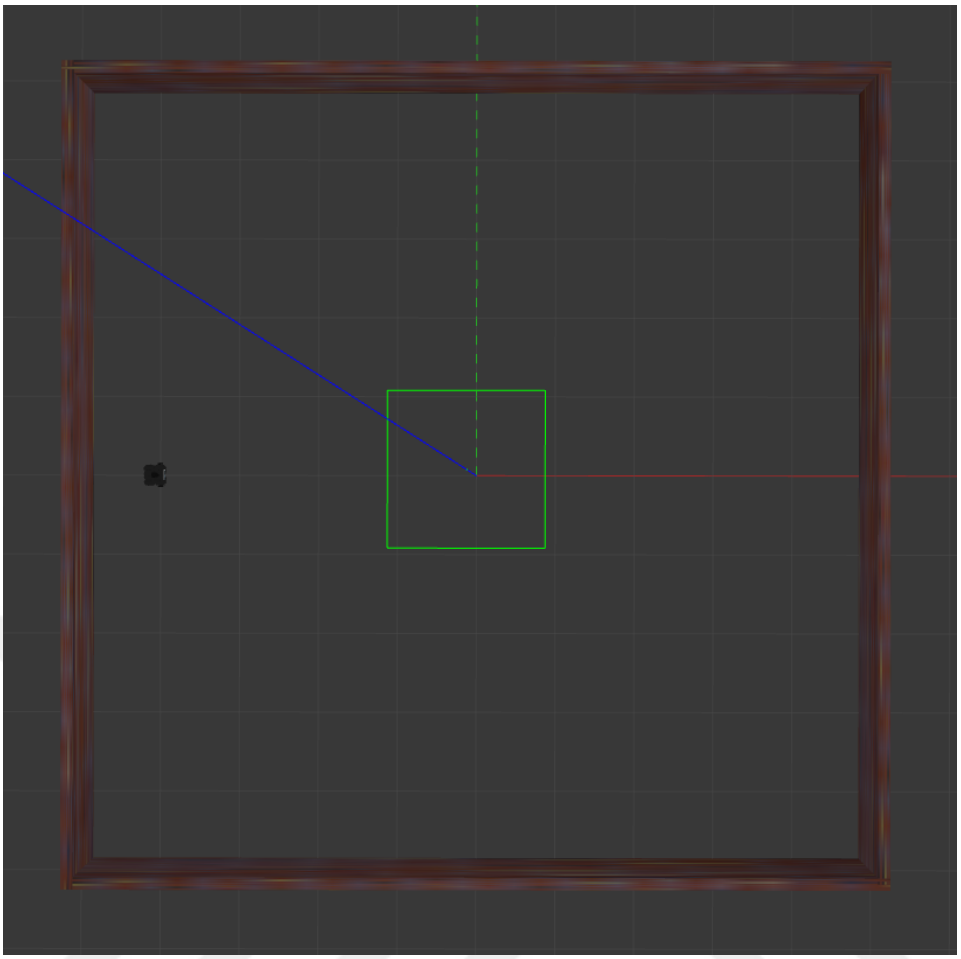


Figure 4.8 The 10x10 m^2 Gazebo world

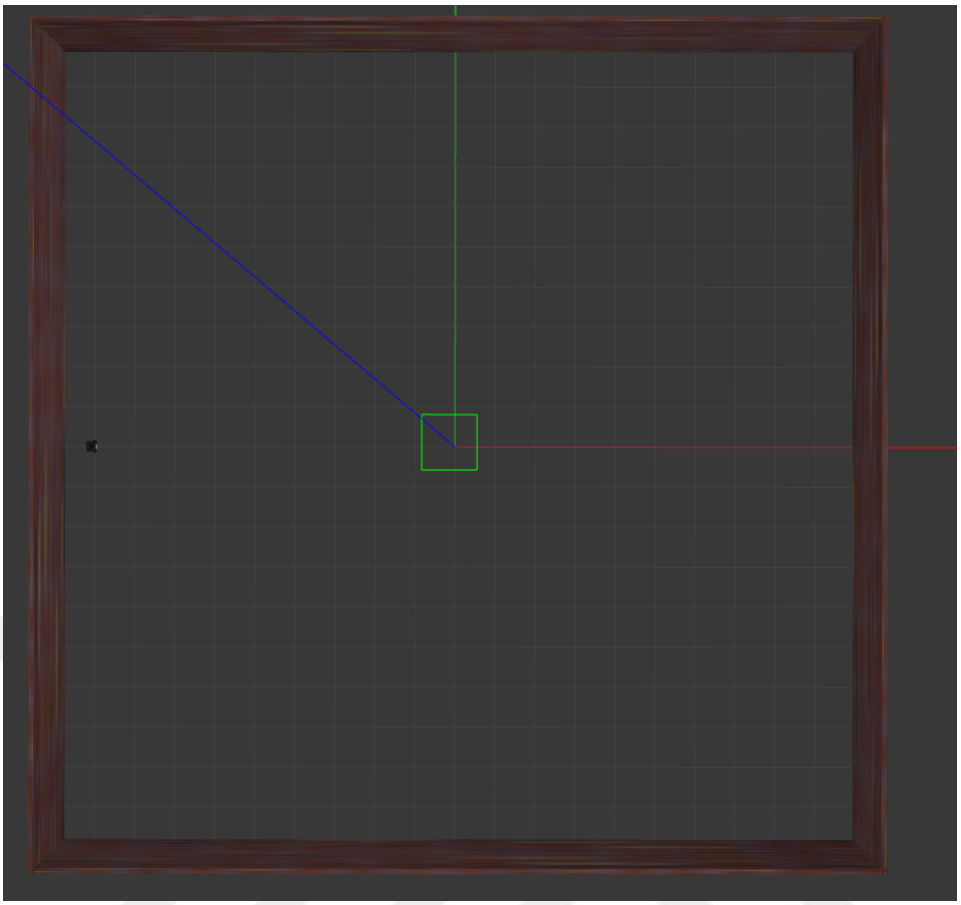


Figure 4.9 The 20x20 m^2 Gazebo world

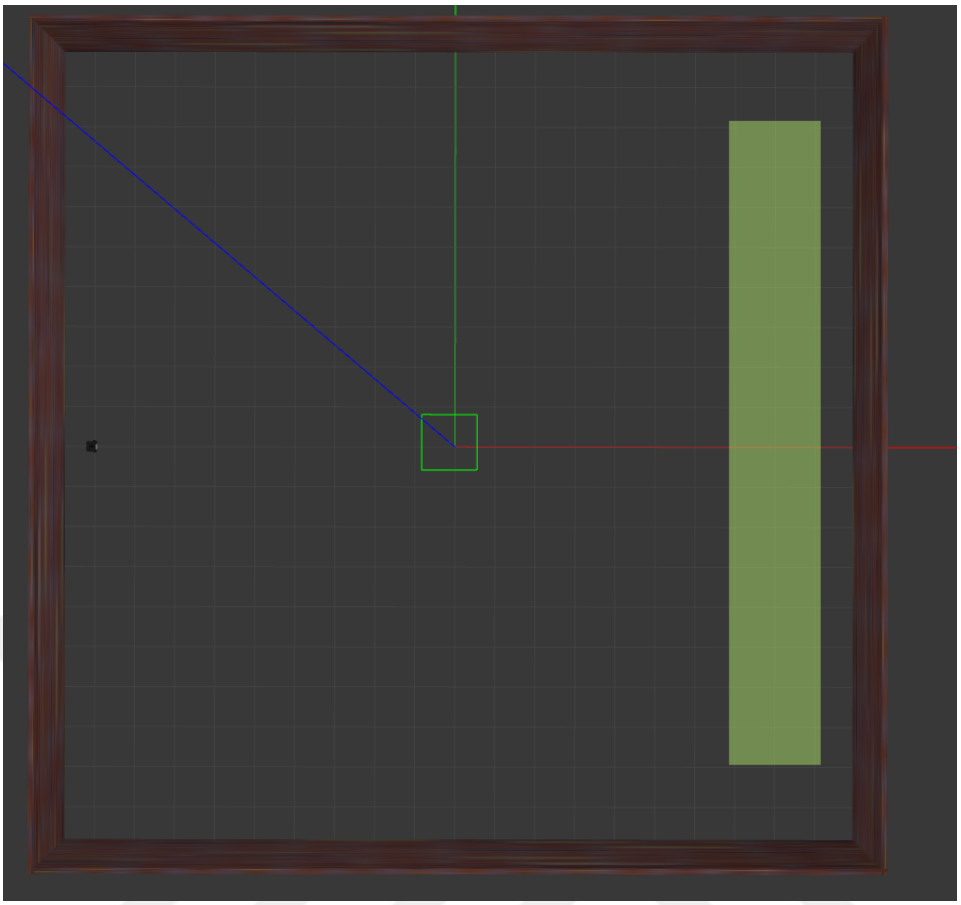


Figure 4.10 The 20x20 m^2 Gazebo world showing the possible target locations.

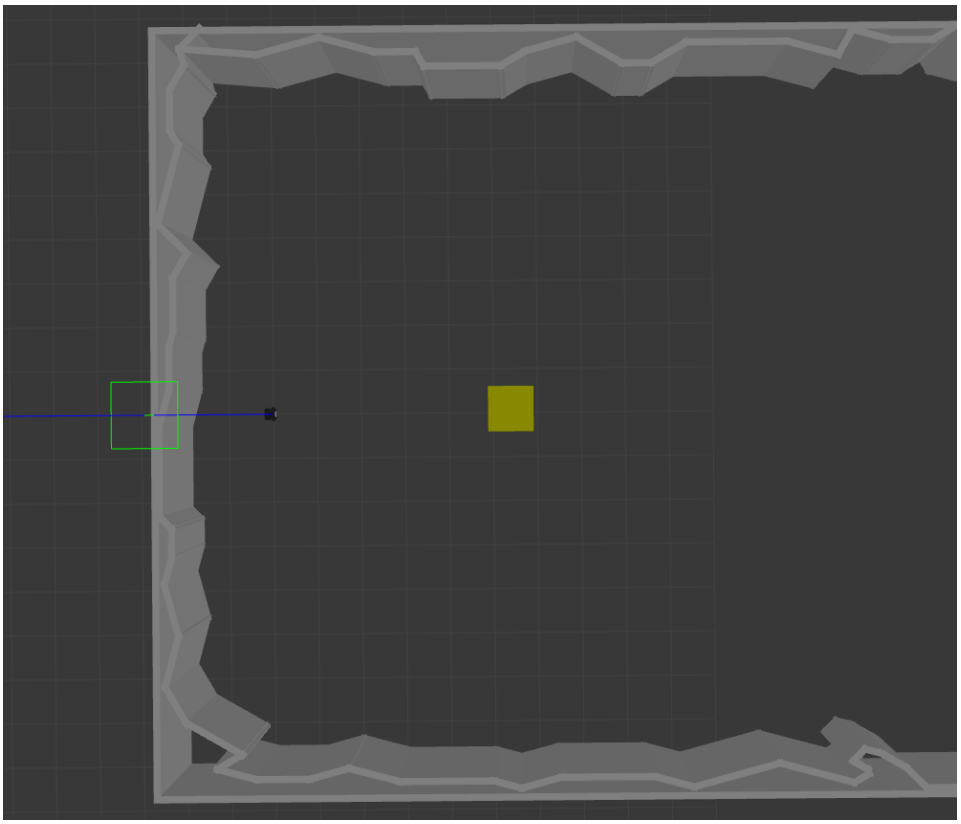


Figure 4.11 The Gazebo world containing obstacle

initially placed at the (0,0) point and aimed to reach the point (13,0). A $1m^3$ cube was placed between the starting point and the target at the point (6,0). However, the global planner does not perceive this obstacle, meaning the route is a straight line. All three models were trained for 2500 steps, and the results were reported.

4.4 Metrics

Local planners are compared based on the number of times the goal is successfully achieved, the total execution time and the sum of absolute areas between the planned path and the executed path metrics. Experiments are repeated 5 times, then averages and standard deviations are reported. Reaching a 0.2 meters radius circle with the goal in the center is classified as an achieved goal. Final heading angle is not considered.

5
RESULTS AND DISCUSSION

The graph in Figure 5.1 depicts the reward during the training of the DQNS14A3 model. The model is trained in 10050 episodes. At around episode 4000, our model has learned to go to a fixed goal in a static environment. Then the obstacles and the goal point are randomly generated in a large area and the model is trained for about 2000 more episodes. The drop on reward observed around episode 5000 is due to change in policy, but the robot quickly adapts to the unfamiliar environment. During the training, it is also observed that there was usually no obstacle between the robot and its goal. So, this area is narrowed. Therefore, the model training decreased again after around episode 6000, but the robot quickly adapts again. It is observed that the model success does not increase after around episode 10050, so the model training is terminated here.

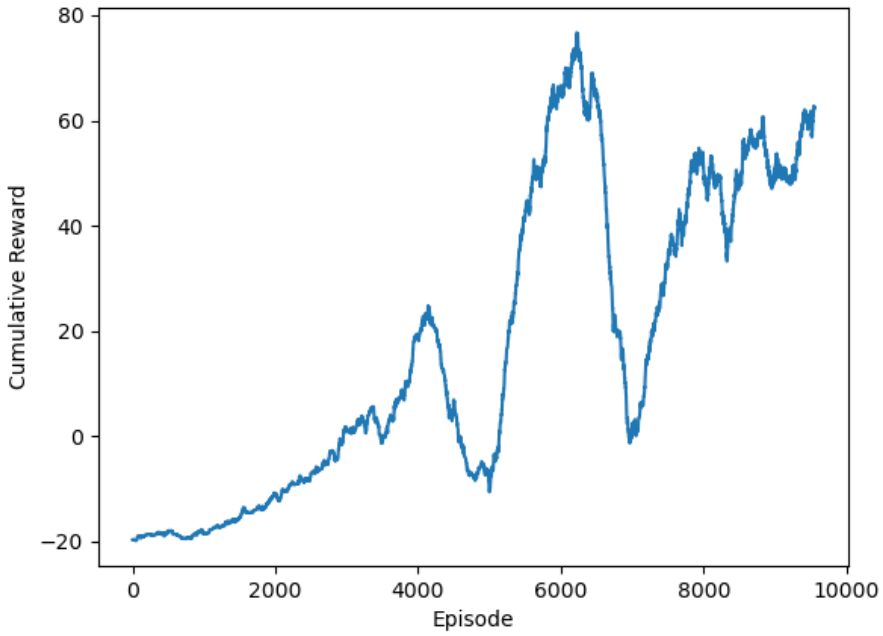


Figure 5.1 The cumulative reward graph during the training phase of the DQNS14A3 model

The values in the all cumulative rewards and all done reasons graphs were obtained

as follows: Each episode ended either by reaching the goal, colliding with an obstacle, exceeding the angle limit, or exceeding the distance limit. At the end of each episode, a list was created for each of these outcomes, assigning a value of 1 to the occurred event and 0 to the others. Then, the values in these lists were obtained using the moving average method as described in Equation 5.1. A window size of 500 steps was selected for the moving average calculation. s

$$\text{moving_average} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{5.1}$$

This method allowed us to track the performance of the agent over time, smoothing out fluctuations in the raw data and providing a clearer understanding of the agent's behavior and learning trends throughout the training process. Additionally, choosing a window size of 500 steps for the moving average calculation effectively captured general trends while preserving details of short-term fluctuations in performance.

After obtaining satisfactory results from the DQNS14A3 model, the state and action space of the model were expanded. It is observed that the new model, DQNS363A15, performs better in route tracking and obstacle avoidance. The agent reached the goal approximately 1384 times in about 3000 episode. During the first 1500 episodes, the focus was solely on route tracking. Similar to Figure 5.2, it can be seen that the cumulative reward steadily increases but starts to decline after around 1500 episode. This decline can be attributed to changes in the training environment. From this point onwards, the agent began to learn obstacle avoidance.

Finally, the DDPG and TD3 models, capable of working in continuous action spaces, were compared with the DQNS363A15V2 model.

When we look at Table 5.1, we see that the DQNS363A15V2 model has reached the goal 242 times, the DDPG model 169 times, and the TD3 model 125 times. All models has struggled the most with exceeding distance limit.

Table 5.1 Done reasons of advanced DQN, DDPG and TD3 models

Model	Goal Reached	Obstacle Collided	Distance Exceeded	Angle Exceeded
DQN	242	796	1189	345
DDPG	169	492	1462	326
TD3	125	267	1384	673

Figure 5.3 illustrates the cumulative reward obtained by the DQNS363A15V2 model throughout its training process. This plot provides insights into how the

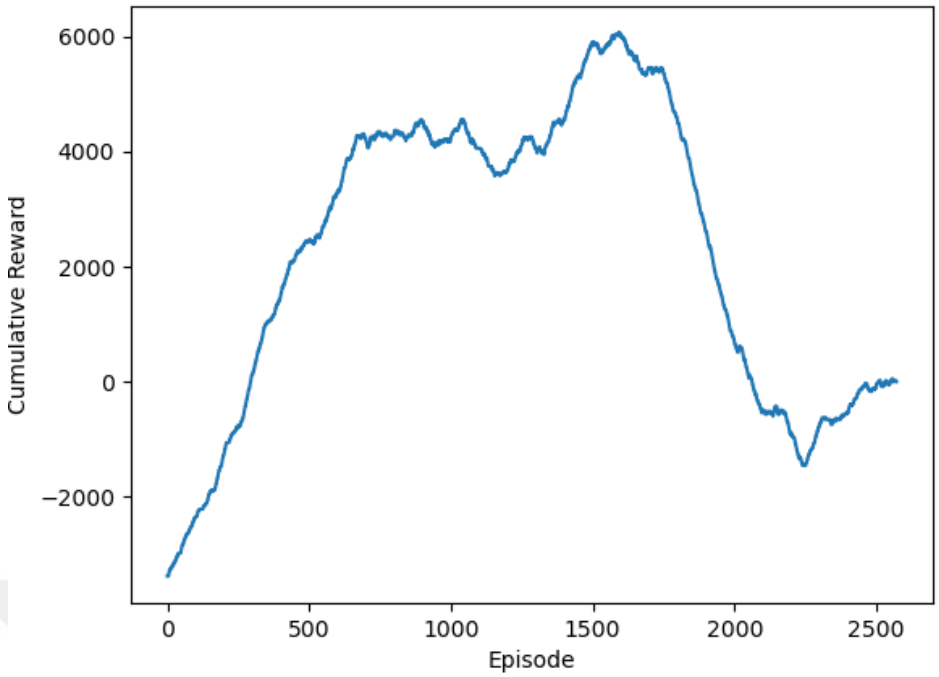


Figure 5.2 Cummulative reward of the DQNS363A15 model

model's performance evolves over time, reflecting its ability to maximize rewards while interacting with the environment.

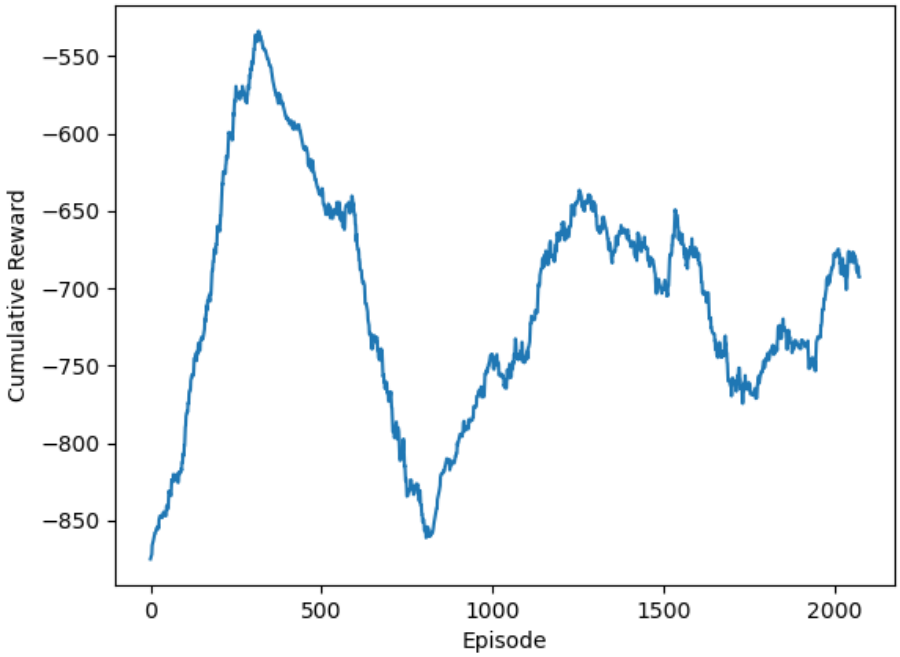


Figure 5.3 Cumulative reward of the DQNS363A15V2 model trained under the same conditions as DDPG and TD3 models

In Figure 5.4, we observe the cumulative reward trajectory of the DDPG model. This graph offers a visualization of the reward accumulation pattern specific to the

DDPG algorithm, demonstrating its effectiveness in optimizing the agent's policy.

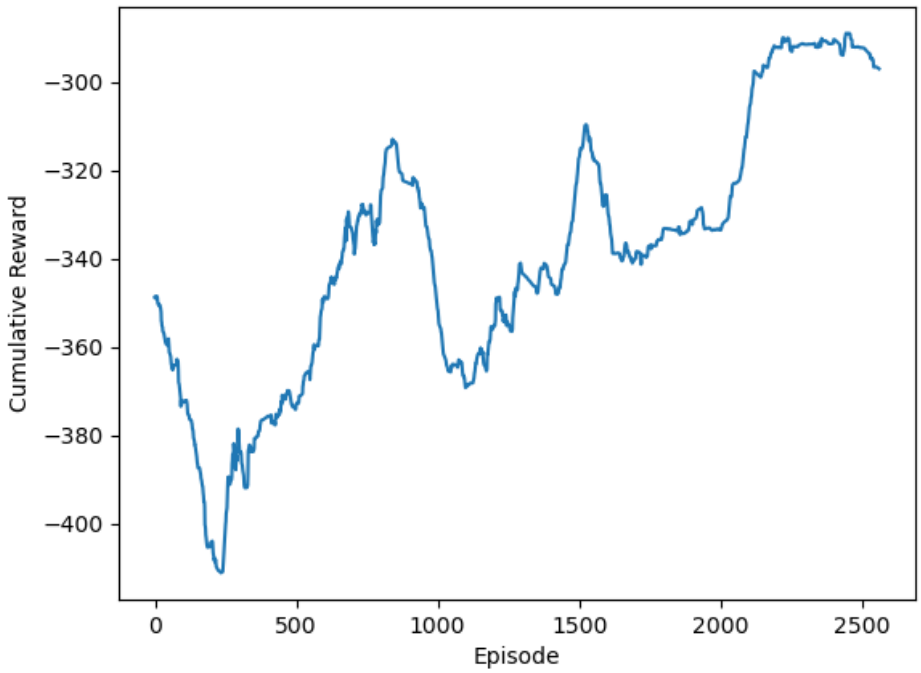


Figure 5.4 Cumulative reward of the DDPG model

Figure 5.5 presents the cumulative reward progression of the TD3 model. By examining this plot, we can discern how the TD3 algorithm impacts the agent's reward acquisition over successive iterations, highlighting its proficiency in reinforcement learning tasks.

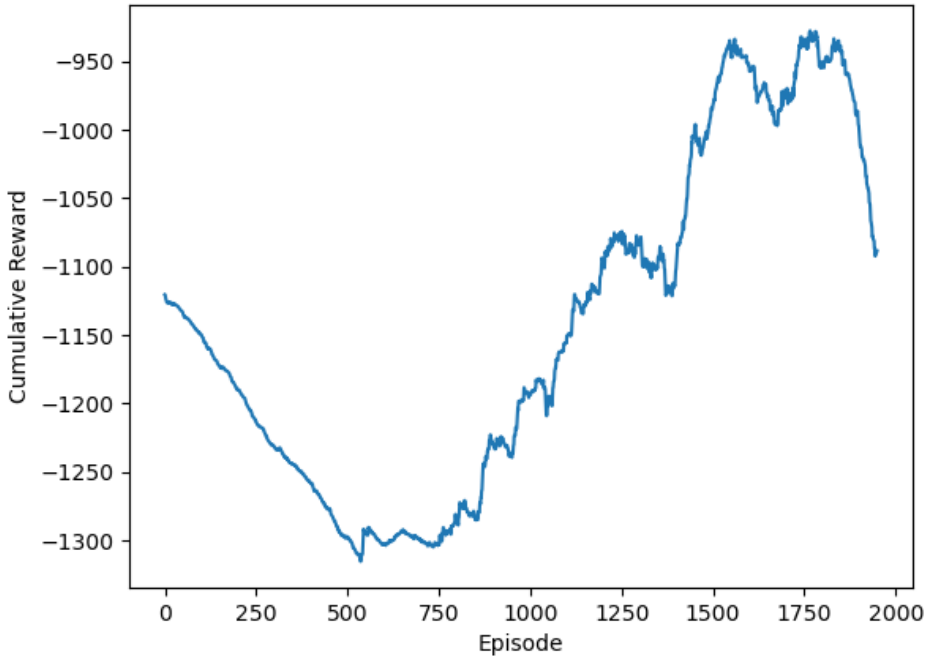


Figure 5.5 Cumulative reward of TD3 the model

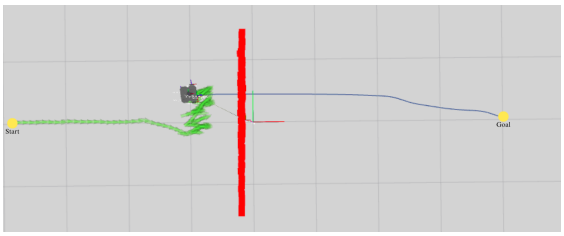
When we look at the training processes of the DQNS363A15V2, DDPG, and TD3 agents for the same scenarios, we observe that DQNS363A15V2 performs better by reaching the goal more frequently. However, we notice that the performance of DDPG and TD3 agents is relatively lower. We believe that this could be attributed to the sensitivity of these models to hyperparameter selection, as mentioned in Chapter 1. These models heavily rely on the choice of hyperparameters, and selecting the appropriate ones can significantly impact their performance.

Moreover, considering the success of the DQNS363A15 model, we have observed that satisfactory results can also be achieved in discrete spaces concerning route tracking and obstacle avoidance by the robot. This demonstrates that even simplified approaches like working in discrete spaces and using algorithms like DQN can be effective in handling complex tasks. Therefore, we believe that with careful hyperparameter tuning and proper training, the success of more complex algorithms can be enhanced.

5.1 Simulation Environment

In the figures below, the blue line represents the global path, green arrows represent the robot's movement and red dots represent the LiDAR sensor readings from the obstacle. Within the scope of scenario 1, the robot is given (4, 0) as goal point. The scenario is repeated 5 times. As shown in Figure 5.6a and Figure 5.6b, `teb_local_planner` and `dwa_local_planner` respectively, fails to pass the obstacle in any of the trials. `teb_local_palnner` and `dwa_local_planner` can trigger the `move_base` to recalculate the path in the case of suddenly encountering an obstacle, however DQN does not have this feature. Normally, there is a parameter in the ROS navigation stack for the global planner to detect dynamic obstacles as well. But here we have turned off that parameter to observe the performance of the pure local planner. As shown in Figure 5.6c, the `dqn_local_planner` with DQNS14A3 reaches the goal in an average of 33.59 ± 3.468 seconds. The robot has moved in an average of 12.032 ± 1.121 m away from its path during its entire movement. After passing the obstacle in each attempt, it does not immediately enter the path and goes directly to the goal point.

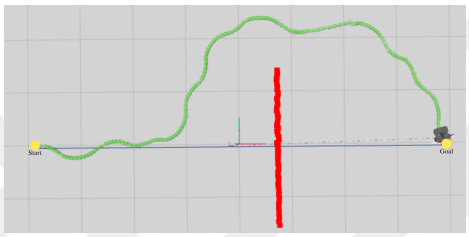
However, as shown in Figure 5.7a, when the goal point is given as (2,0), the robot could not reach the goal in all trials. If the projection of the robot's position to the path at any time becomes the goal point, it cannot reach the goal. When the goal point is given as (11,0) as shown in Figure 5.7b, the `dqn_local_planner` with DQNS14A3 reaches the goal in an average of 57.395 ± 1.492 seconds. The robot has



(a)



(b)



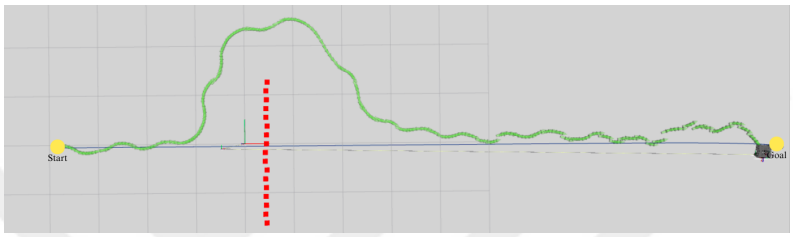
(c)

Figure 5.6 Scenario 1b results for a) teb_local_planner, b) dwa_local_planner, c) dqn_local_planner

moved an average of 14.18 ± 1.876 m away from its path during its entire movement. The robot enters the path after passing the obstacle and continues to follow the path. In one of the experiments, the robot passed the goal without reaching the goal, so the experiment was terminated.



(a)



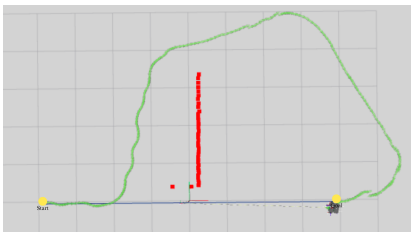
(b)

Figure 5.7 dqn_local_planner behaviours for a) scenario 1a, b) scenario 1c

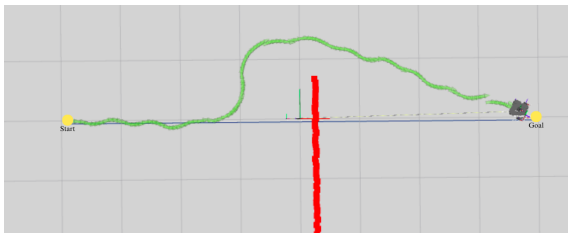
In scenario 2, the target is given when the obstacle is in various positions. The local path differs because the obstacle was a moving object as shown in Figure 5.8a and Figure 5.8b. The `dqn_loca_planner` with DQNS14A3 reached the target with an average of 38.88 ± 11.040 seconds. The robot has moved an average of 13.07 ± 16.86 m away from its path during its entire movement. It encounters an obstacle once and the alternative path cannot be reproduced. After the obstacle moves away from the robot, the robot continues its path as shown in Figure 5.8c.

As shown in Figure 5.9a, the `teb_local_planner` reaches the goal in an average of 25.9 ± 1.1 seconds. The robot has moved an average of 1.71 ± 0.244 m away from its path during its entire movement. When an obstacle is in front of the Robot, `teb_local_planner` waits until it gets out of the way. After the obstacle is removed, it continues its movement. As shown in Figure 5.9b, the `dwa_local_planner` reaches the goal in an average of 25.13 ± 4.995 seconds. The robot has moved an average of 1.41 ± 1.374 m away from its path during its entire movement. DWA waits when an obstacle is in front of it and continues after it is removed. However, while the obstacle is moving towards the robot from the side of the robot, it cannot react adequately, and the obstacle hits the robot.

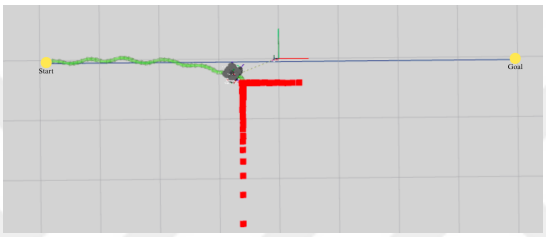
In scenario 3, the `teb_local_planner` and `dwa_local_planner` could not pass the obstacle as in the first scenario. As shown in Figure 5.10, the



(a)

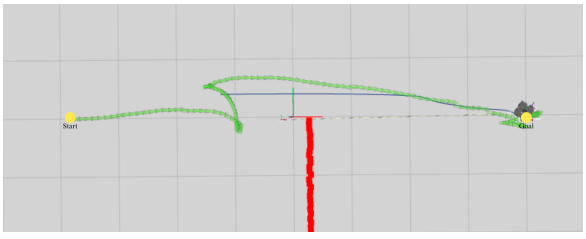


(b)

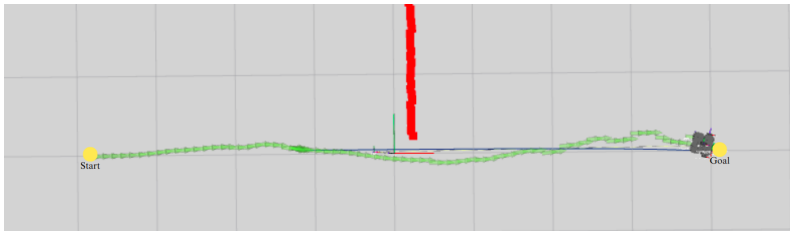


(c)

Figure 5.8 dqn_local_planner behaviours from different runs for scenario 2 when obstacle is encountered at, a) $y > 0$, b) $y \cong 0$, c) $y < 0$



(a)



(b)

Figure 5.9 Scenario 2 behaviours for, a) teb_local_planner, b) dwa_local_planner

`dqn_local_planner` with DQNS14A3 reached the goal in an average of 49.23 ± 0.959 seconds. The robot has moved an average of 22.02 ± 5.17 m away from its path during its entire movement. `dqn_local_planner` with DQNS14A3 can easily pass slower objects in the same direction.

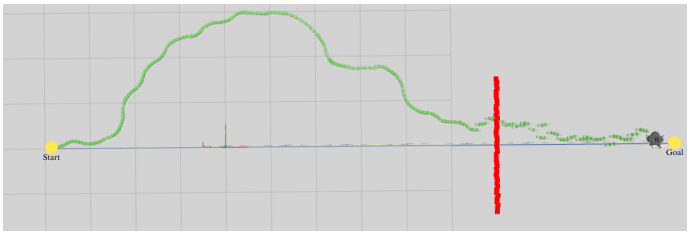


Figure 5.10 Scenario 3 behaviour of dqn_local_planner

Scenario 4 gave equivalent results to scenario 3. Thanks to the robot `dqn_local_planner` with DQNS14A3, it easily passed an obstacle coming towards it with a speed of 0.1 m/s.

Table 5.2 summarizes the results of the proposed method by means of number of successfully achieved goals, execution times and area between the planned and executed paths, where global path lengths set an anchor for base comparison.

To summarize the outputs of all scenarios, although `teb_local_planner` and `dwa_local_planner` give much better results when there are no obstacles or small obstacles on the path, `dqn_local_planner` with DQNS14A3 gives better results if there are large obstacles on the robot's path. However, in the current model, if the robot accidentally passes the goal point while following the path, it is exceedingly difficult to return and reach the goal point again. It is usually stuck at the local minimum point as shown in Figure 5.7a. The robot may also not take the shortest path while avoiding obstacles. If the robot is too close to the obstacle, it waits until the obstacle disappears and continues its movement. If the obstacle moves to the right in the y-axis forever and the robot encounters the obstacle, if it turns to pass to the right of the obstacle in the first step, they will go together forever in this way. The robot prefers not to go left at any given moment.

The training environment illustrated in Figure 4.11 was utilized to evaluate the performance of the advanced DQNS363A15V2, DDPG and TD3 models. In each trial, the robot commenced from point (0,0) and was tasked with reaching a target positioned at (13,0). This procedure was repeated 10 times. Observations revealed that the DQN model encountered the "angle_exceeded" error twice, resulting in the target being bypassed. However, it successfully reached the target in the remaining 8 attempts. On average, the DQN model achieved the goal in 23.4 ± 1.91 seconds, deviating by an average of 6.36 ± 0.35 meters from its intended path during its

Table 5.2 Comparison of all simulation environment scenarios for DQN

	Scenario 1					
Metrics	a	b	c	Scenario 2	Scenario 3	Scenario 4
Global plan length (m)	5.9	7.9	14.9	7.9	13.9	13.9
The number of times the goal is successfully achieved	0	4	5	4	5	5
Total execution time (s)	-	33.59 ± 3.47	57.39 ± 1.49	38.88 ± 11.04	49.23 ± 0.96	49.34 ± 0.93
Sum of absolute areas between the planned path and the executed path (m)	-	12.03 ± 1.12	14.18 ± 1.88	13.07 ± 16.86	22.02 ± 5.17	20.58 ± 3.59

movement as shon in Table 5.3.

In contrast, both the DDPG and TD3 models consistently reached the target in every trial. The DDPG model attained the goal in 137.52 ± 0.48 seconds on average, with an average deviation of 6.05 ± 0.039 meters from its intended path. Similarly, the TD3 model achieved the goal in 136.4 ± 0.13 seconds on average, deviating by an average of 6.27 ± 0.05 meters from its intended path.

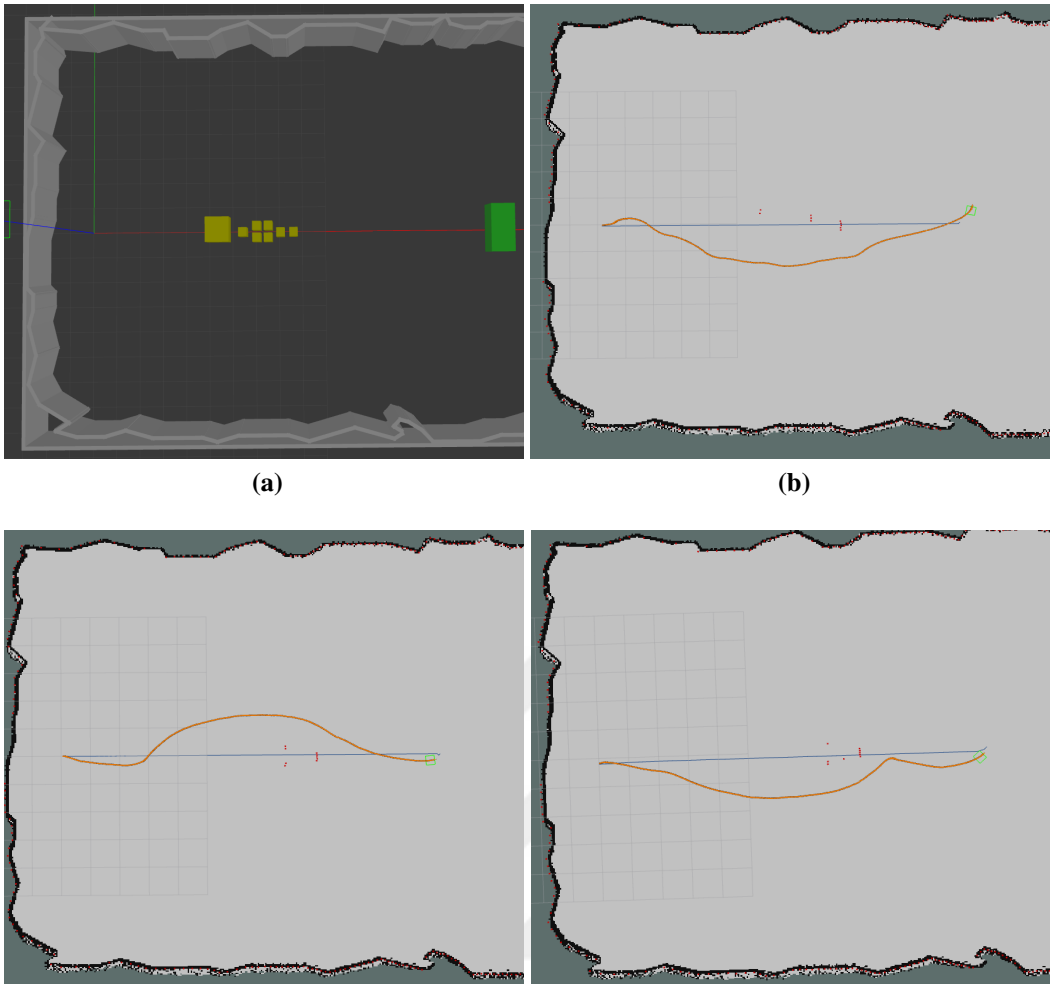
Table 5.3 The results of the DQN, DDPG, and TD3 models in the simulation environment.

Metrics	DQN	DDPG	TD3
The number of times the goal is successfully achieved	8	10	10
Total execution time (s)	23.4 ± 1.91	137.52 ± 0.48	136.4 ± 0.13
Sum of absolute areas between the planned path and the executed path (m)	6.36 ± 0.35	6.05 ± 0.039	6.27 ± 0.05

Furthermore, it has been tested in more complex environments as shown in Figure 5.11a, and it has been observed that the robot successfully reached its target. In this environment, there are obstacles that were not present during the training stage. The results are shown in Figure 5.11b, Figure 5.11c and Figure 5.11d.

If we were to elaborate, during the training process of the DQNS363A15V2 model, there are situations within the training space that remain unlearned. This allows the model to take actions such as deviating from the path, as indicated by the first arrow in Figure 5.12. However, as the robot approaches the goal, we observe another state space becoming active, indicating the robot's orientation towards the goal , as indicated by the second arrow in Figure 5.12. The blue line in the Figure 5.12 represents the global path, the orange line represents the local path, and the red line represents obstacles, with the green rectangle representing the robot.

Finally, robot was instructed to perform slalom movement. An environment similar to the one depicted in Figure 5.13a was set up with a target at point (15,0). DQN successfully reached the target as shown in Figure 5.13b, and DDPG reached the target as shown in Figure 5.13c. However, TD3, as illustrated in Figure 5.13d, managed to pass the first obstacle but failed to pass the second one. It stopped due to approaching the obstacle too closely.



(a) **(b)**

(c) **(d)**

Figure 5.11 The simulation test in complex environment, a) gazebo environment, b) DQN result, c) DDPG result, d) TD3 result

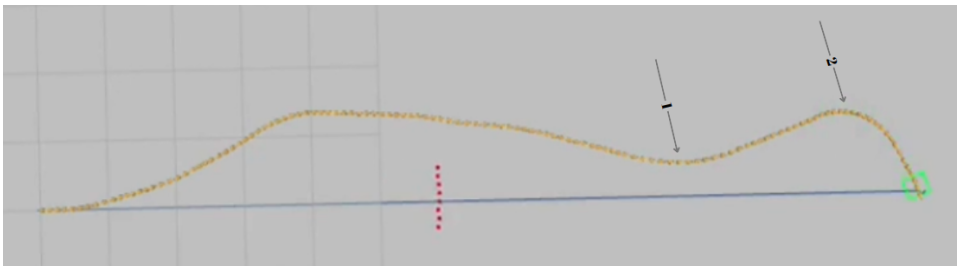
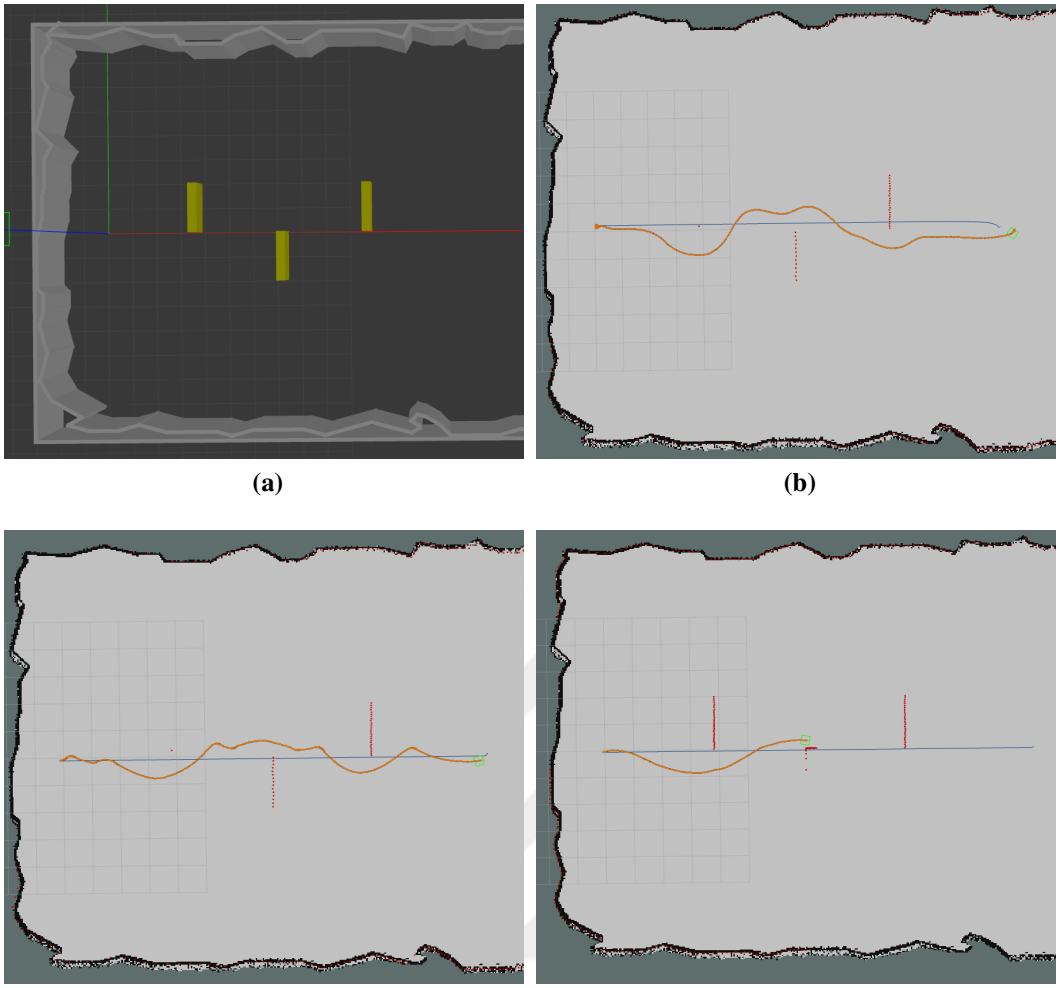


Figure 5.12 Simulation test example of the DQNS363A15V2 model



(a) **(b)**

(c) **(d)**

Figure 5.13 The simulation test for slalom movement, a) gazebo environment, b) DQN result, c) DDPG result, d) TD3 result

5.2 Real Platform

Real platform's performance is evaluated based on both calculated path following capabilities and avoidance of unseen static obstacles during path planning. DQN, DDPG and TD3 models have been trained using Turtlebot3 Waffle. Therefore, when testing on a real robot, laser data is shifted to simulate the behavior of the Turtlebot3.

The real platform's path following capability is demonstrated in the following case (Figure 5.14) where a rather complex path is calculated and passed to DQN for execution. The blue line represents the calculated path with A*, whereas the red line around the calculated path represents the path executed by the robot in Figure 5.14. The mapping result is depicted with light gray unoccupied cells, black obstacles and dark gray unknown cells. The robot's footprint is represented by the green rectangle reaching the goal point. Finally the LiDAR sensor readings are shown with red dots.

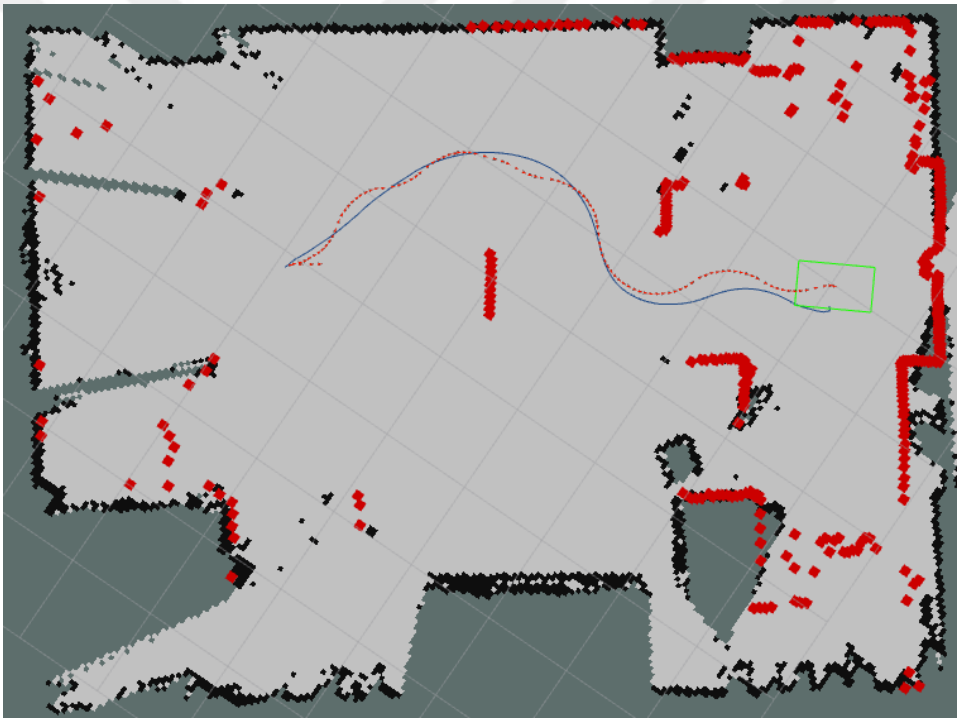


Figure 5.14 Real platform path following test Rviz output for DQNS14A3 model

The execution of the calculated path, as depicted in Figure 5.14, is represented frame by frame in Figure 5.15. Although minor oscillations are observed during the task completion, it should be noted that these are expected results due to the three-valued action space of forward, left-forward, and right-forward commands generated by DQNS14A3.

The obstacle avoidance capability of the DQN-based local planner is demonstrated when an unseen static obstacle is introduced during navigation. This can be observed on the real platform in Figure 5.16 and Figure 5.17. The blue line in these

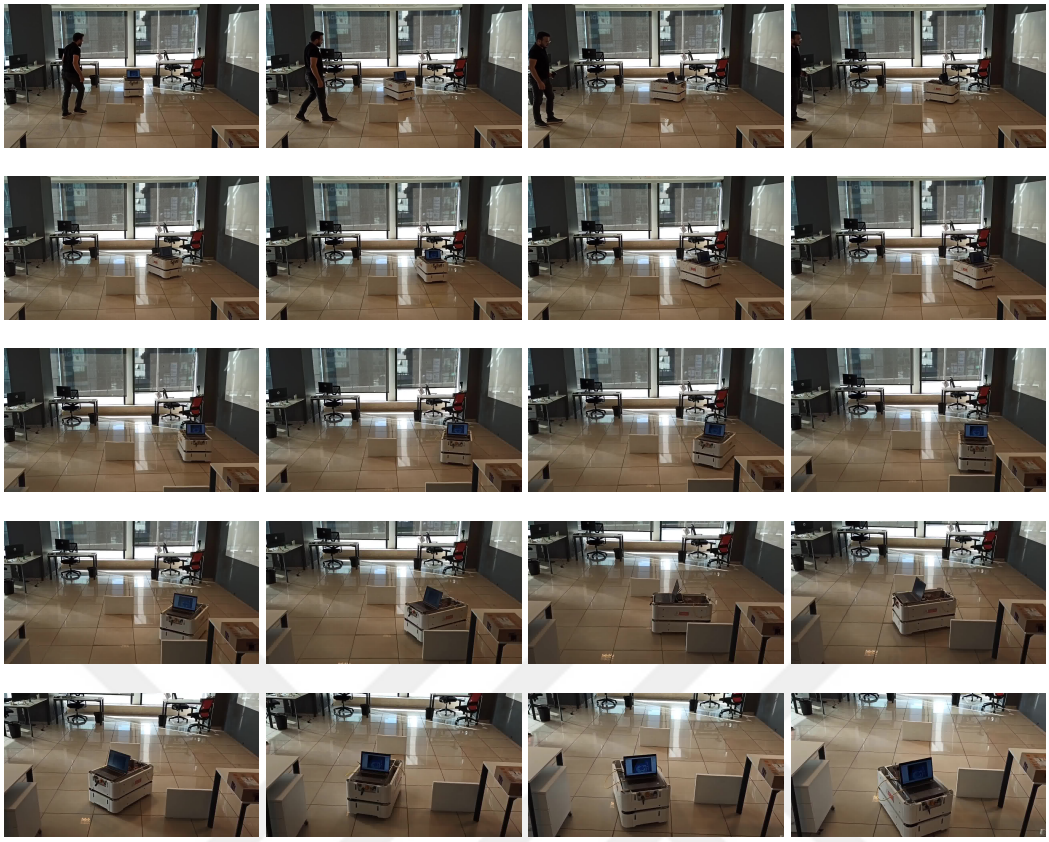


Figure 5.15 Real platform path following test for DQNS14A3 model (in left to right, top to bottom order)

figures represents the calculated path with A*, while the yellow line represents the path executed by the robot. As seen from Figure 5.16, the calculated path is not aware of the static obstacle, yet the DQN local planner with DQNS14A3 is able to execute an obstacle-avoiding trajectory.

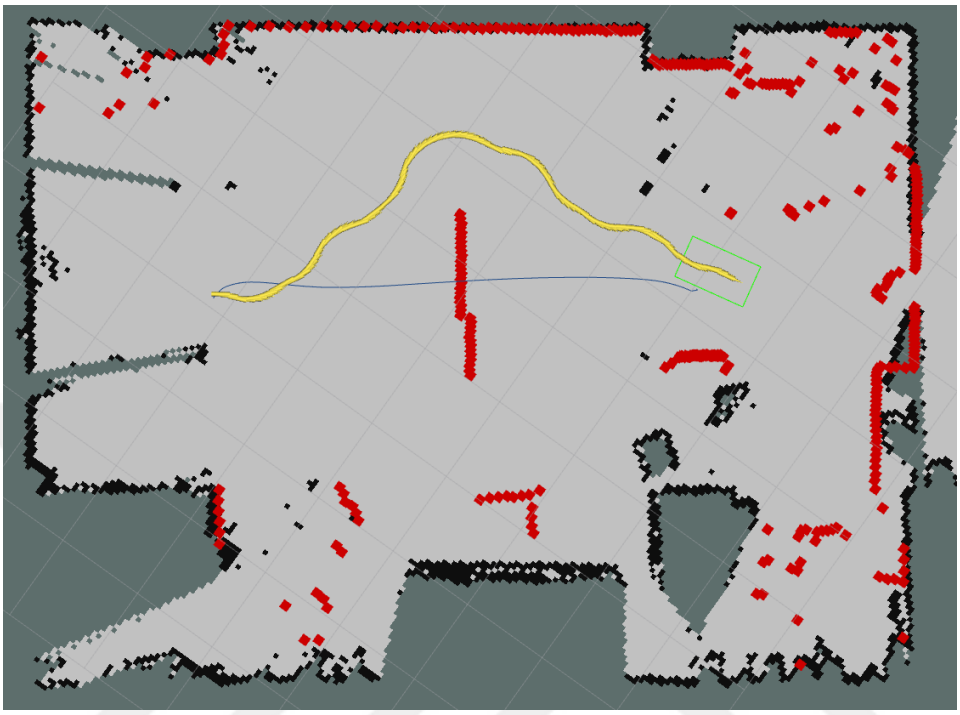


Figure 5.16 Real platform obstacle avoidance Rviz output for DQNS14A3 model

The execution of the calculated path given in Figure 5.16 is represented frame by frame in Figure 5.17.

Similarly, the enhanced DQNS363A15 model with expanded state and action spaces demonstrates the DQN-based local planner's obstacle avoidance capability, as shown in Figure 5.18 and Figure 5.18. The blue line represents the calculated path with A*, while the orange line represents the path executed by the robot. As seen from Figure 5.18, the calculated path is not aware of the static obstacle, yet the advanced DQN local planner with DQNS363A15 is able to execute an obstacle-avoiding trajectory.

The execution of the calculated path in Figure 5.18 is represented frame by frame in Figure 5.19.

The obstacle avoidance capability of the DDPG-based local planner is demonstrated in Figure 5.20 and Figure 5.20. The blue line represents the calculated path with A*, while the orange line represents the path executed by the robot. As seen from Figure 5.20, the calculated path is not aware of the static obstacle, yet the DDPG-based local planner is able to execute an obstacle-avoiding trajectory.

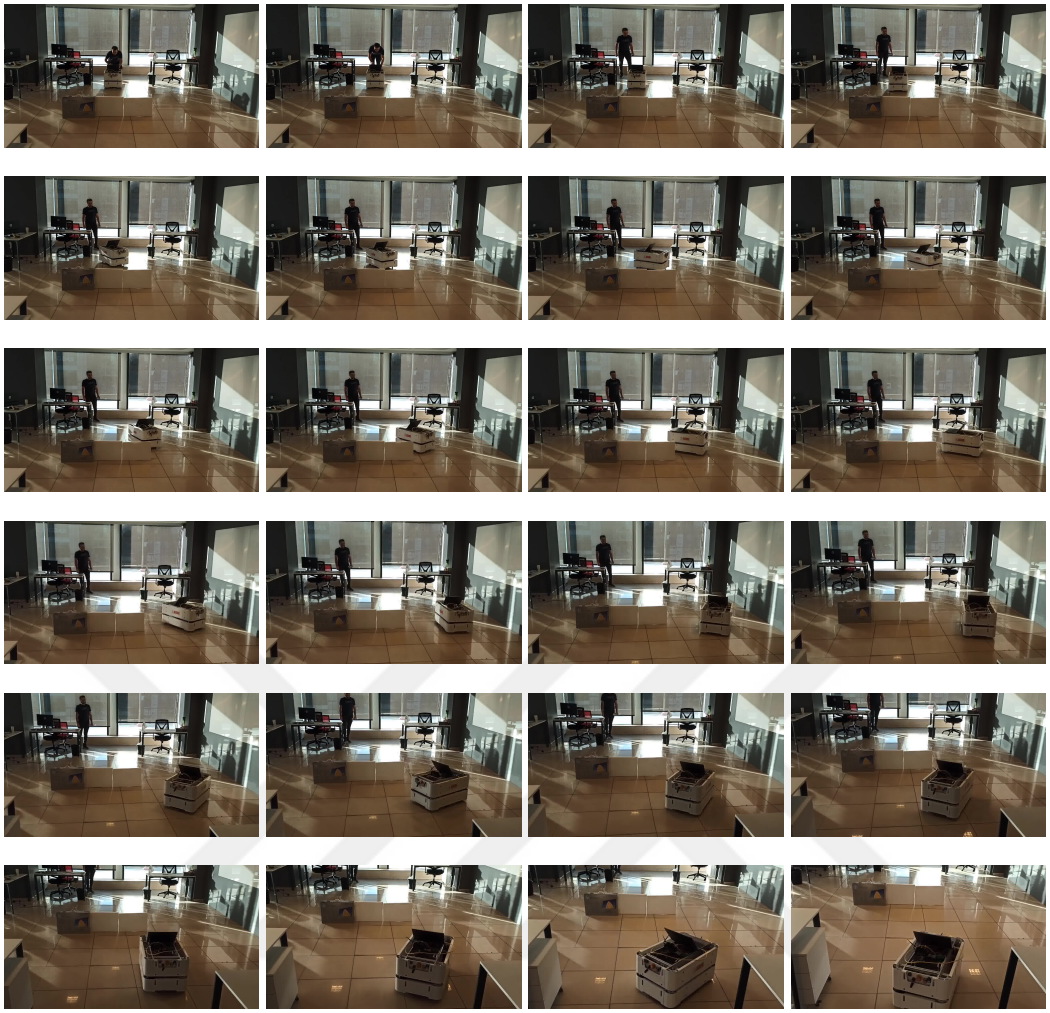


Figure 5.17 Real platform obstacle avoidance test for DQNS14A3 model (in left to right, top to bottom order)

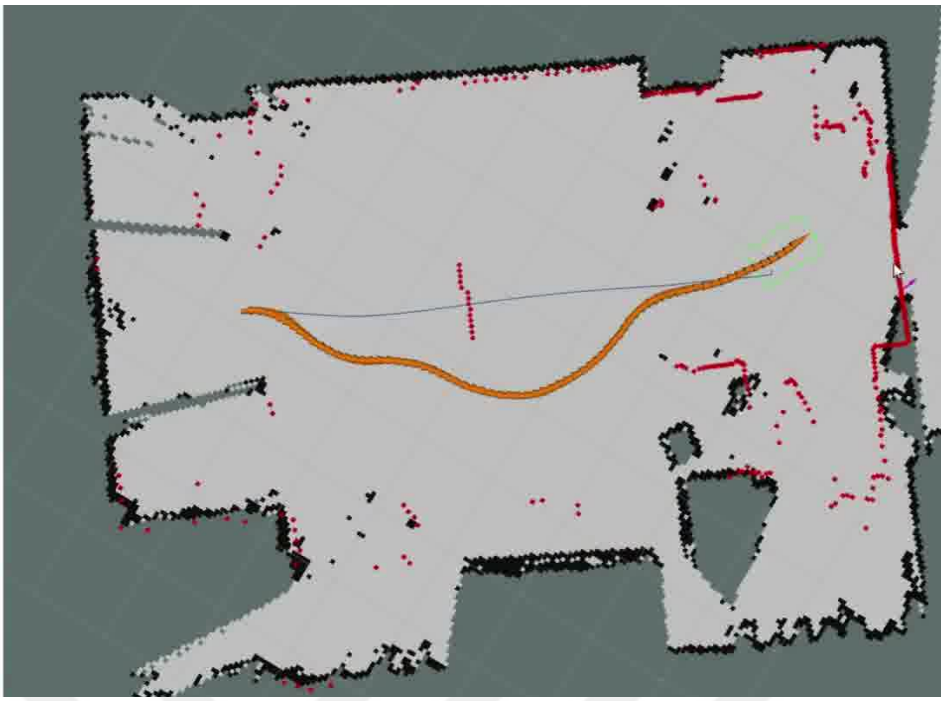


Figure 5.18 Real platform obstacle avoidance Rviz output for DQNS363A15 Model

The execution of the calculated path in Figure 5.20 is represented frame by frame in Figure 5.21.

Finally, the obstacle avoidance capability of the TD3-based local planner is demonstrated in Figure 5.22 and Figure 5.22. The blue line represents the calculated path with A*, while the orange line represents the path executed by the robot. As seen from Figure 5.22, the calculated path is not aware of the static obstacle, yet the TD3 local planner is able to execute an obstacle-avoiding trajectory.

The execution of the calculated path in Figure 5.22 is represented frame by frame in Figure 5.23.

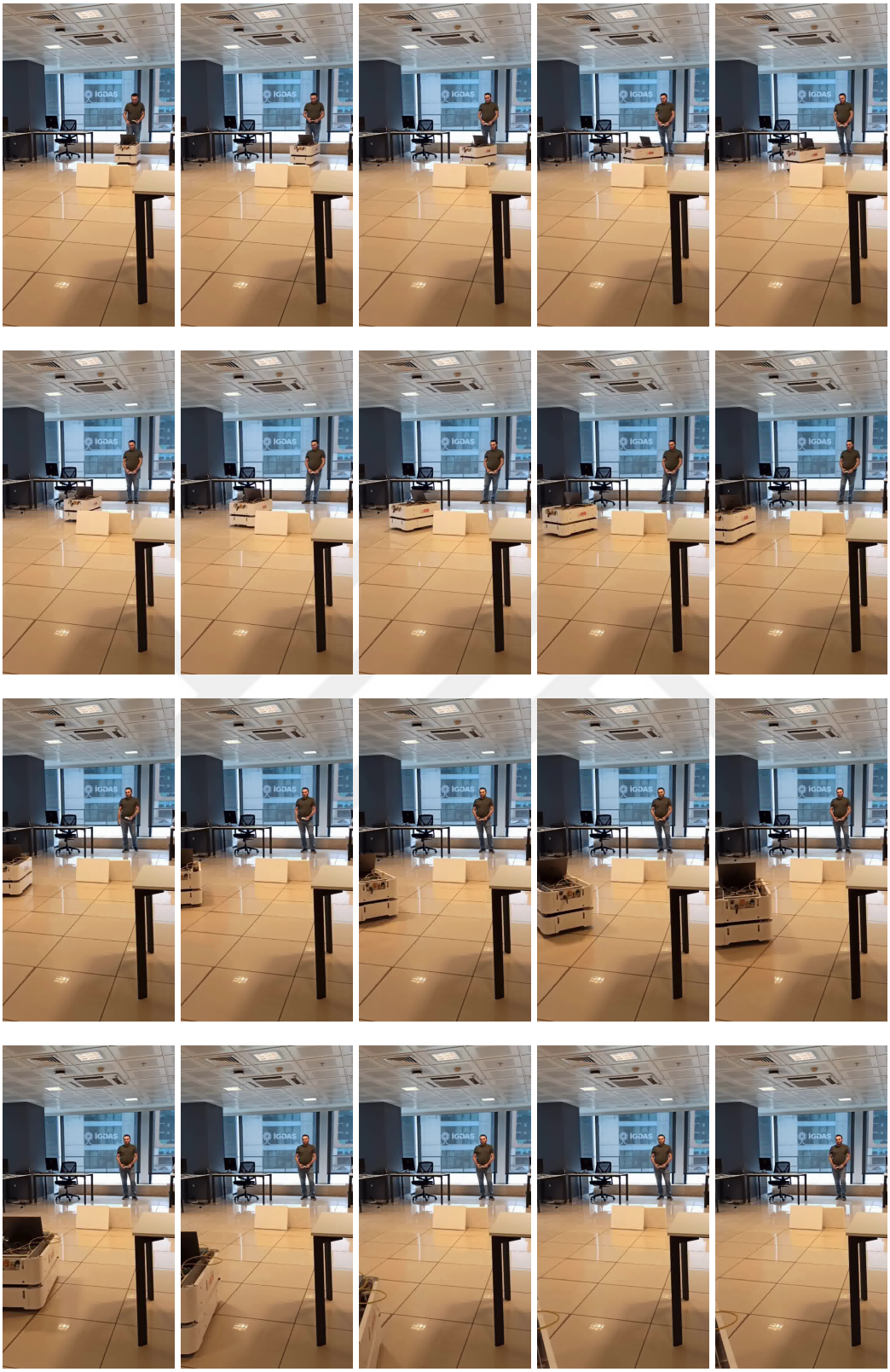


Figure 5.19 Real platform obstacle avoidance test for DQNS363A15 model(in left to right, top to bottom order)

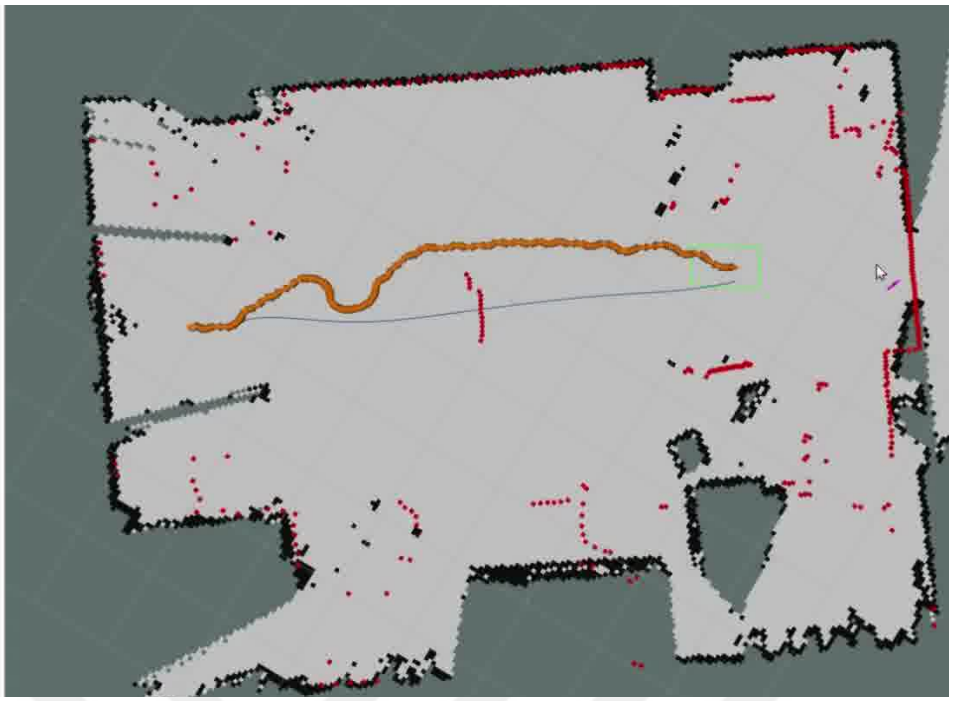


Figure 5.20 Real platform obstacle avoidance Rviz output for DDPG Model

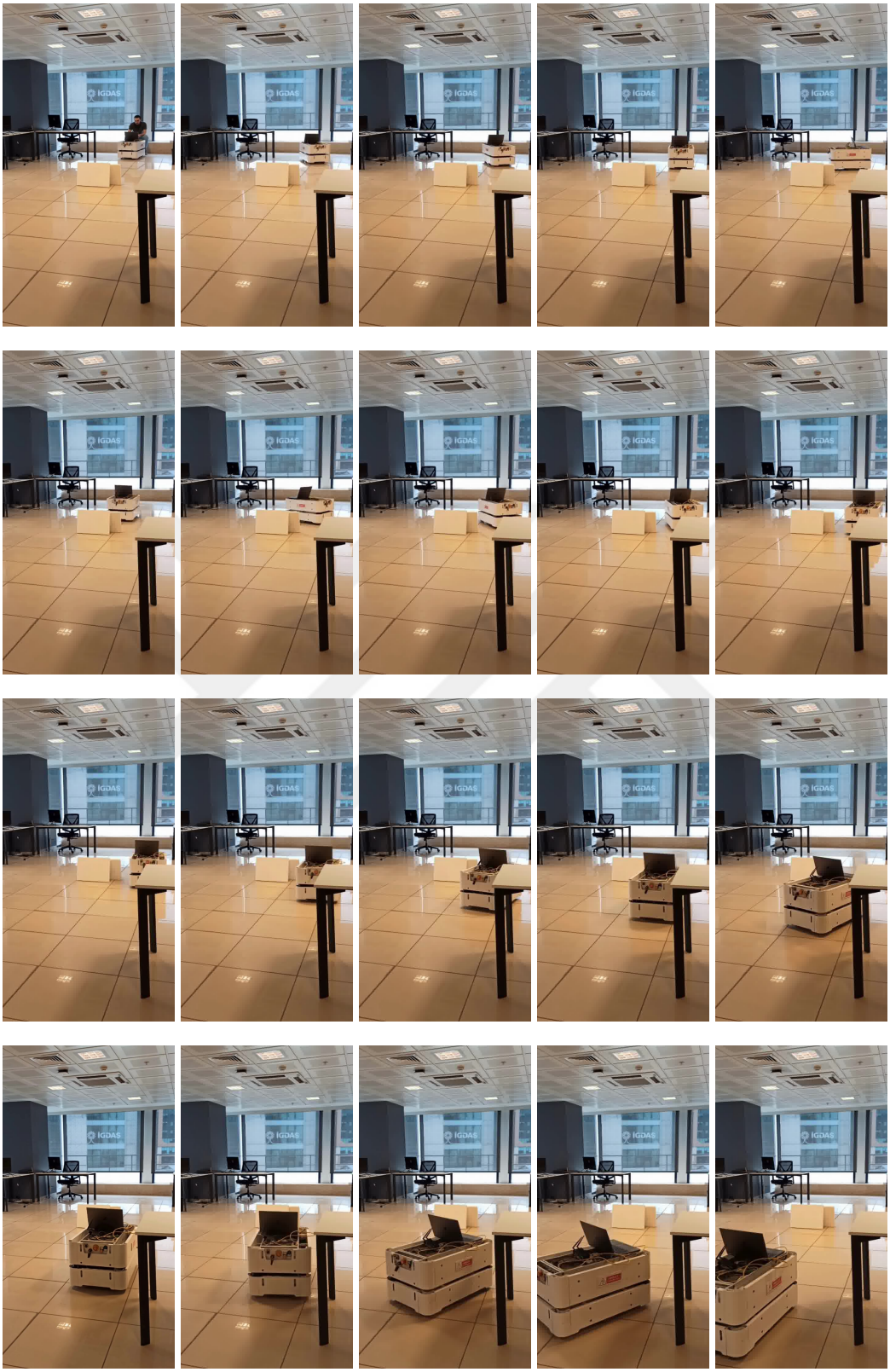


Figure 5.21 Real platform obstacle avoidance test for DDPG model(in left to right, top to bottom order)

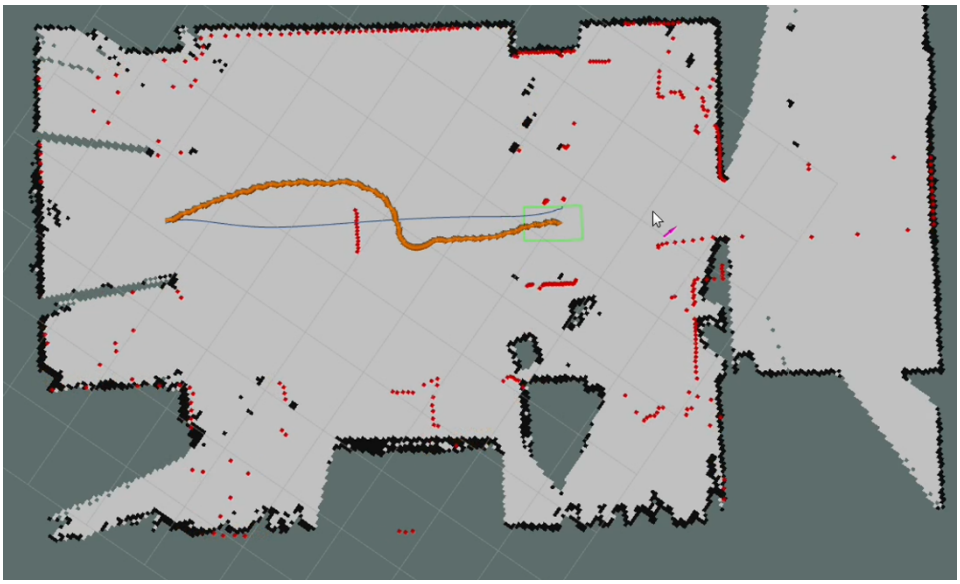


Figure 5.22 Real platform obstacle avoidance Rviz output for TD3 Model

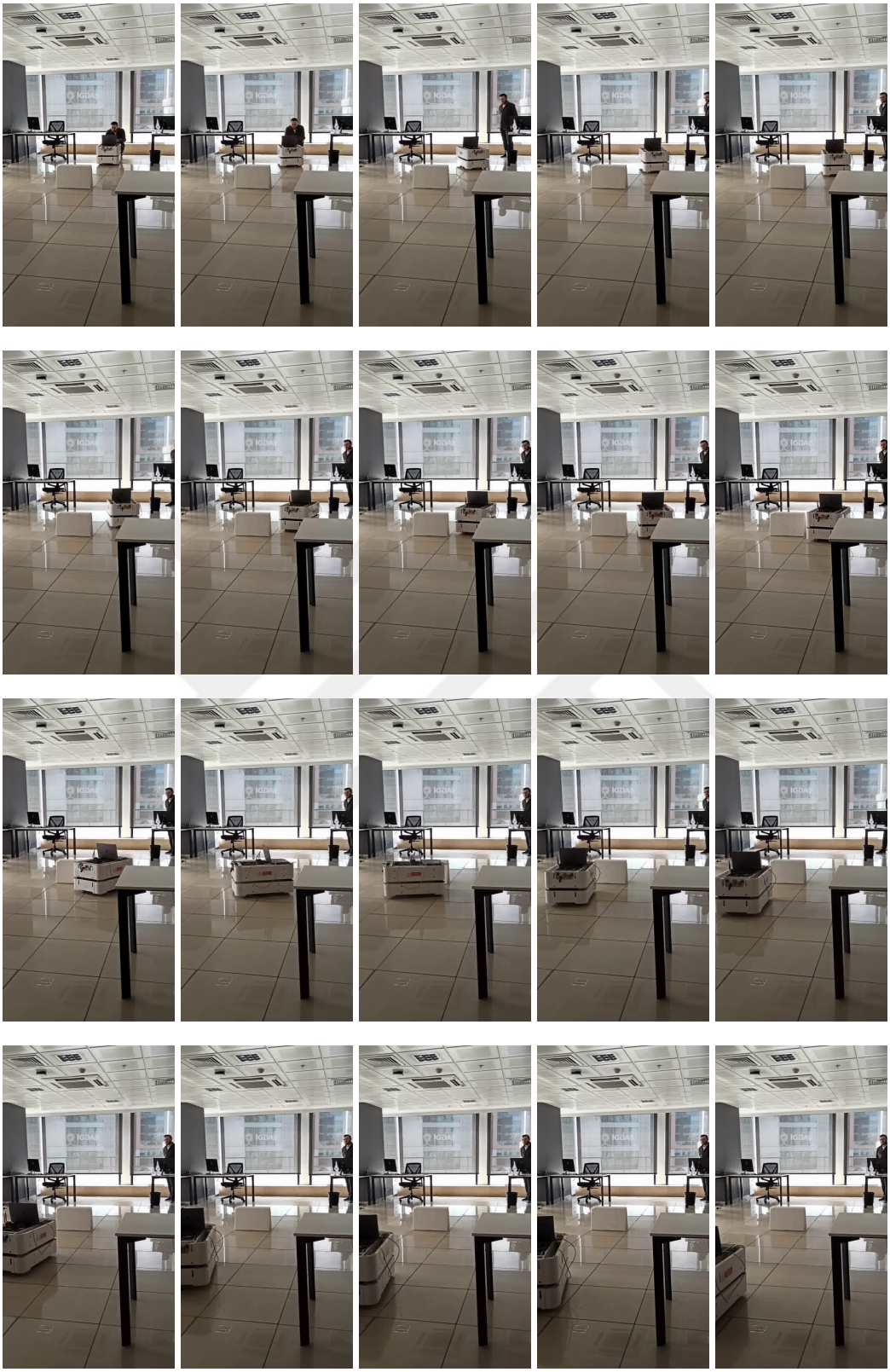


Figure 5.23 Real platform obstacle avoidance test for TD3 model (in left to right, top to bottom order)

6
CONCLUSION

In conclusion, a new method developed with DQN model called DQNS14A3, which is a RL method, is proposed for the local planner. In this method, the environment is first taught to the robot by using sensor data, robot position and global path. For testing the model, fixed, and moving (toward the robot, moving away from the robot, moving vertically) objects that are not in the environment map are added and it is observed whether the robot could reach the goal by avoiding obstacles. The same experiments are also performed with TEB and DWA. To test the performance of the pure local planner, the obstacles that are subsequently introduced into the environment are not added to the global path. While TEB and DWA planners are not successful in avoiding obstacles, our model is able to pass the obstacle easily in all scenarios. The model trained in the simulation environment is also tested with the real platform in an office environment. Firstly, a complex path that requires several mild rotations is created to test path tracking. Secondly, obstacle avoidance on a straight path is tested. Just like in the simulation environment, the robot successfully executed the obstacle avoiding path even on the planned straight path.

Additionally, the DQNAS14A3 method has been extended to continuous state space and a 15-action space, renamed as DQNS363A15, and its performance has been observed under complex environmental conditions. Moreover, DDPG and TD3 models, capable of operating in continuous action spaces, have been trained and compared with the DQN model. This comparison encompasses critical performance metrics such as each model's ability to handle obstacles, time taken to reach the goal, and accuracy in path following. The results indicate that while the DQN method can effectively operate in complex environments, DDPG and TD3 models, designed for continuous state space, exhibit even more enhanced performance. Such findings provide insights into the comparative analyses of different RL methods for local planning problems, aiding in determining the most suitable approach for various scenarios.

If the models do not learn the entire space, local minimum points may occur. In

addition, if the robot and the obstacle start parallel movement in the same direction and at a similar speed, they will move together forever. However, our current models is trained in a specific environment and so works in a small world. More complex environments can be selected for training in later models. Moreover, only a fully connected network is used in our models. While teaching more complex spaces, training can be continued with networks such as Convolutional Neural Network (CNN), Long Short-Term Memory (LSTM) network. As another future work it is planned also to include the dynamic objects' direction and speed values in the state space, in order to enable the robot to learn not to move in a crossing path with the dynamic object and let the robot move away from the obstacle in an efficient way.

REFERENCES

[1] G. Fragapane, R. De Koster, F. Sgarbossa, J. O. Strandhagen, "Planning and control of autonomous mobile robots for intralogistics: Literature review and research agenda," *European Journal of Operational Research*, vol. 294, no. 2, pp. 405–426, 2021. doi: https://doi.org/10.1016/j.ejor.2021.01.019.

[2] H. Durrant-Whyte, T. Bailey, "Simultaneous localization and mapping: Part i," *IEEE robotics & automation magazine*, vol. 13, no. 2, pp. 99–110, 2006. doi: 10.1109/MRA.2006.1638022.

[3] M. Betke, L. Gurvits, "Mobile robot localization using landmarks," *IEEE transactions on robotics and automation*, vol. 13, no. 2, pp. 251–263, 1997.

[4] J. J. Leonard, H. F. Durrant-Whyte, "Mobile robot localization by tracking geometric beacons," *IEEE Transactions on robotics and Automation*, vol. 7, no. 3, pp. 376–382, 1991. doi: 10.1109/70.88147.

[5] J. Biswas, M. Veloso, "Depth camera based indoor mobile robot localization and navigation," in *2012 IEEE International Conference on Robotics and Automation*, IEEE, 2012, pp. 1697–1702. doi: 10.1109/ICRA.2012.6224766.

[6] P. E. Hart, N. J. Nilsson, B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[7] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," *Research Report 9811*, 1998.

[8] D. Fox, W. Burgard, S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics & Automation Magazine*, vol. 4, no. 1, pp. 23–33, 1997.

[9] C. Rösmann, W. Feiten, T. Wösch, F. Hoffmann, T. Bertram, "Trajectory modification considering dynamic constraints of autonomous robots," in *ROBOTIK 2012; 7th German Conference on Robotics*, VDE, 2012, pp. 1–6.

[10] X. Xiao, B. Liu, G. Warnell, P. Stone, "Toward agile maneuvers in highly constrained spaces: Learning from hallucination," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1503–1510, 2021.

[11] A. Pokle *et al.*, "Deep local trajectory replanning and control for robot navigation," in *2019 international conference on robotics and automation (ICRA)*, IEEE, 2019, pp. 5815–5822.

[12] B. Rubí, B. Morcego, R. Pérez, "A deep reinforcement learning approach for path following on a quadrotor," in *2020 European Control Conference (ECC)*, IEEE, 2020, pp. 1092–1098.

[13] S. Zhou, B. Li, C. Ding, L. Lu, C. Ding, "An efficient deep reinforcement learning framework for uavs," in *2020 21st International Symposium on Quality Electronic Design (ISQED)*, IEEE, 2020, pp. 323–328.

[14] Q. Zhang, J. Lin, Q. Sha, B. He, G. Li, "Deep interactive reinforcement learning for path following of autonomous underwater vehicle," *IEEE Access*, vol. 8, pp. 24 258–24 268, 2020.

[15] M. M. Ejaz, T. B. Tang, C.-K. Lu, "Vision-based autonomous navigation approach for a tracked robot using deep reinforcement learning," *IEEE Sensors Journal*, vol. 21, no. 2, pp. 2230–2240, 2020.

[16] X. Ruan, D. Ren, X. Zhu, J. Huang, "Mobile robot navigation based on deep reinforcement learning," in *2019 Chinese control and decision conference (CCDC)*, IEEE, 2019, pp. 6174–6178. doi: 10.1109/CCDC.2019.8832393.

[17] E. Marchesini, A. Farinelli, "Discrete deep reinforcement learning for mapless navigation," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2020, pp. 10 688–10 694.

[18] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, J. Pineau, "An introduction to deep reinforcement learning," *Foundations and Trends® in Machine Learning*, vol. 11, no. 3-4, pp. 219–354, 2018. doi: 10.1561/2200000071.

[19] H. Zhang, T. Yu, "Taxonomy of reinforcement learning algorithms," *Deep Reinforcement Learning: Fundamentals, Research and Applications*, pp. 125–133, 2020.

[20] C. J. Watkins, P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992. doi: 10.1007/BF00992698.

[21] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015. doi: 10.1038/nature14236.

[22] M. Quigley *et al.*, "Ros: An open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[23] D. Thomas, W. Woodall, E. Fernandez, "Next-generation ROS: Building on DDS," in *ROSCon Chicago 2014*, Mountain View, CA: Open Robotics, Sep. 2014. doi: 10.36288/ROSCon2014-900183. [Online]. Available: https://vimeo.com/106992622.

[24] TullyFoote. "Ros tutorials." Accessed: October 22, 2022. (2009), [Online]. Available: https://wiki.ros.org/ROS/Tutorials.

[25] G. Grisetti, C. Stachniss, W. Burgard, "Improved techniques for grid mapping with rao-blackwellized particle filters," *IEEE transactions on Robotics*, vol. 23, no. 1, pp. 34–46, 2007. doi: 10.1109/TRO.2006.889486.

[26] P. Pfaff, W. Burgard, D. Fox, "Robust monte-carlo localization using adaptive likelihood models," in *European robotics symposium 2006*, H. I. Christensen, Ed., Springer, 2006, pp. 181–194, isbn: 978-3-540-32689-2. doi: 10.1007/11681120_15.

[27] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, K. Konolige, "The office marathon: Robust navigation in an indoor office environment," in *International Conference on Robotics and Automation*, 2010, pp. 300–307. doi: 10.1109/ROBOT.2010.5509725.

[28] E. Marder-Eppstein. "Move_base." Accessed: January 1, 2024. (2009), [Online]. Available: https://wiki.ros.org/move_base/.

[29] N. Koenig, 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)*, IEEE, vol. 3, 2004, pp. 2149–2154. doi: 10.1109/IROS.2004.1389727.

[30] M. Erdogan. "Gazebo worlds models and maps." Accessed: February 21, 2024. (2024), [Online]. Available: https://github.com/mlherd/Dataset-of-Gazebo-Worlds-Models-and-Maps.

[31] G. Brockman *et al.*, "Openai gym," 2016. eprint: arXiv:1606.01540.

[32] M. Towers *et al.*, *Gymnasium*, Mar. 2023. doi: 10.5281/zenodo.8127026. [Online]. Available: https://zenodo.org/record/8127025 (visited on 07/08/2023).

[33] R. C. Coulter *et al.*, *Implementation of the pure pursuit path tracking algorithm*. Carnegie Mellon University, The Robotics Institute, 1992.

[34] S. space. "Dynamic window approach (dwa)." Accessed: February 11, 2024. (2024), [Online]. Available: https://salihspace.home.blog/2020/12/07/dynamic-window-approach/.

[35] R. S. Sutton, A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[36] M. Guo, Y. Liu, J. Malec, "A new q-learning algorithm based on the metropolis criterion," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 34, no. 5, pp. 2140–2143, 2004.

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

[38] O. Anschel, N. Baram, N. Shimkin, "Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning," in *International conference on machine learning*, PMLR, 2017, pp. 176–185.

[39] I. Osband, C. Blundell, A. Pritzel, B. Van Roy, "Deep exploration via bootstrapped dqn," *Advances in neural information processing systems*, vol. 29, 2016.

[40] T. P. Lillicrap *et al.*, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[41] G. E. Uhlenbeck, L. S. Ornstein, "On the theory of the brownian motion," *Physical review*, vol. 36, no. 5, p. 823, 1930.

[42] S. Fujimoto, H. Hoof, D. Meger, “Addressing function approximation error in actor-critic methods,” in *International conference on machine learning*, PMLR, 2018, pp. 1587–1596.

[43] S. Narvekar, B. Peng, M. Leonetti, J. Sinapov, M. E. Taylor, P. Stone, “Curriculum learning for reinforcement learning domains: A framework and survey,” *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 7382–7431, 2020.

[44] ROS.org. “Writing a global path planner as plugin in ros.” Accessed: February 17, 2024. (2021), [Online]. Available: http://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20ROS.

[45] ROS.org. “Openai_ros.” Accessed: February 17, 2024. (2021), [Online]. Available: http://wiki.ros.org/openai_ros.

Conference Papers

1. Küçükyılmaz, M., Uslu, E. (2024). ROS Compatible Local Planner and Controller Based on Reinforcement Learning. In: Şen, Z., Uygun, Ö., Erden, C. (eds) Advances in Intelligent Manufacturing and Service System Informatics. IMSS 2023. Lecture Notes in Mechanical Engineering. Springer, Singapore. https://doi.org/10.1007/978-981-99-6062-0_37