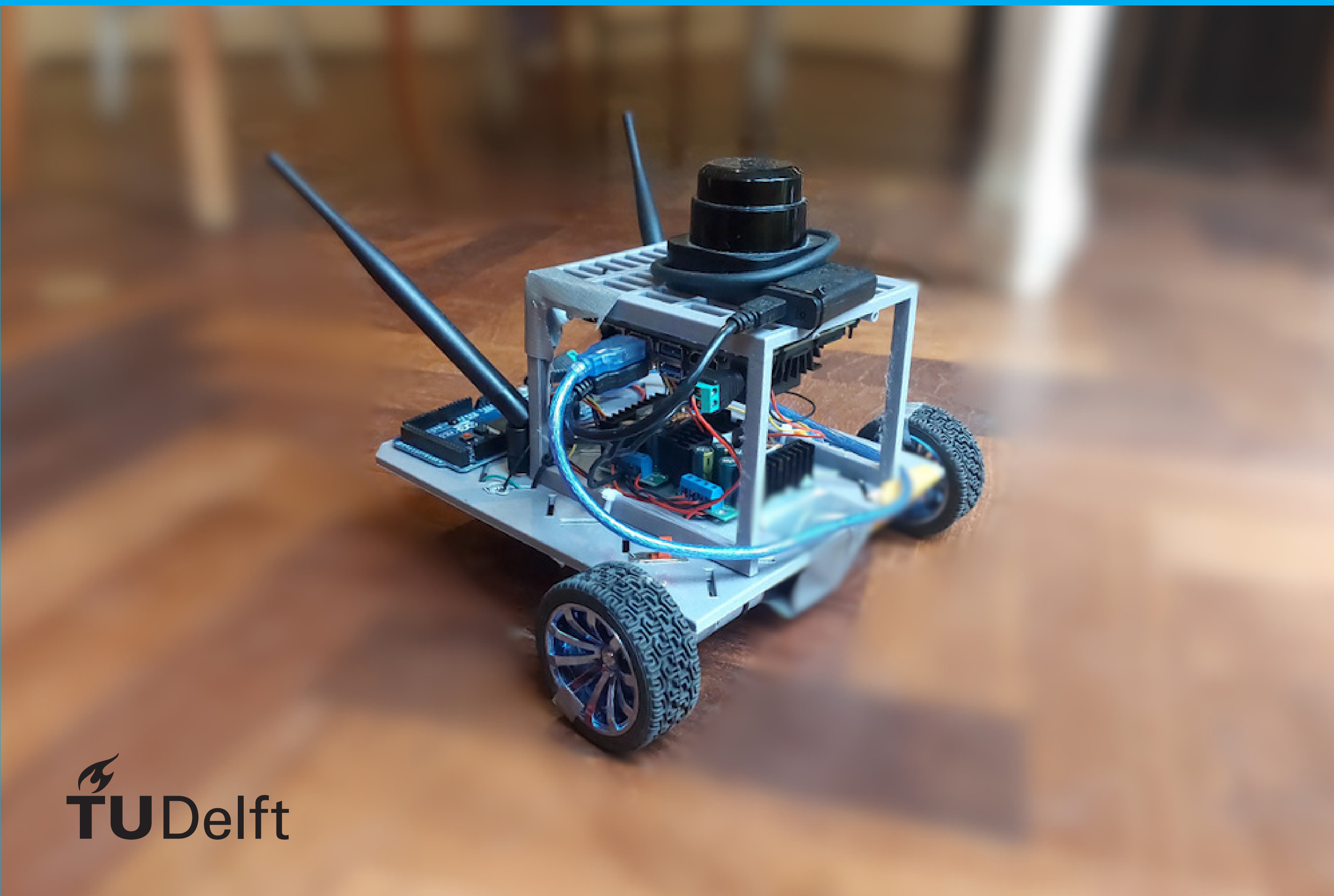
Toward Reliable Robot Navigation Using Deep Reinforcement Learning

Thesis Report

Tomas van Rietbergen



TU Delft

Toward Reliable Robot Navigation Using Deep Reinforcement Learning

Thesis Report

by

Tomas van Rietbergen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday November 7, 2022 at 10:00 AM.

Student number:	5179505	
Project duration:	January 14, 2021 – November 7, 2022	
Thesis committee:	Prof. dr. R. V. Prasad,	TU Delft, supervisor
	Prof. dr. ir. Babuska,	TU Delft
	Dr. A. Y. Majid,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Preface

With this thesis, I conclude my educational period of studying at the TU Delft. I am grateful for having had the chance to pursue a master's degree in the Embedded Systems program through which I have obtained a great deal of technical knowledge and academic experience.

This period has certainly not been without hardships and would not have been the same without the help of the people who supported me. I would like to thank my daily supervisor Dr. Amjad Y. Majid for his guidance and feedback throughout the thesis process and for encouraging me to get the most out of this research experience. I would also like to thank Prof. Ranga R. V. Prasad for his valuable advice and help during the graduation process.

Lastly, I would like to thank all my friends and family who have always been there for me without question and supported me through each and every hardship including the pandemic. In particular, I want to express my gratitude to my father, Bert, my mother, Terry, and my dear friends, Karim, Ion, and Vince, without whom the completion of this master thesis would have been a great deal more arduous. Lastly, I would also like to thank my brother Juraj for allowing me to make use of his machine to perform model training.

Thank you.

Contents

Abstract

Reliable indoor navigation in the presence of dynamic obstacles is an essential capability for mobile robot deployment. Previous work on robot navigation focuses on expanding the network structure and hardware setup leading to more complex and costly systems. The accompanying physical demonstrations are often limited to slow-moving agents and simplistic obstacle configurations. In this thesis, we develop an end-to-end navigation system with a focus on real-world transferability and produce a low-cost and customizable robot platform. Instead of expanding the network structure, we rely on external capabilities such as backward motion, frame stacking, and behavioral reward design to improve performance while preserving transferability. By convention, these methods have been largely disregarded in previous works on deep reinforcement learning (DRL) for unmanned ground vehicle (UGV) navigation. We analyze the effect on performance in simulation of different off-policy algorithms with hyperparameter and reward function configurations. Experimental results show that our agent can achieve state-of-the-art performance in challenging and unseen simulated environments. In addition, physical robot demonstrations show that our system is capable of dealing with fast-moving and unpredictable agents in a real-world environment.

1

Introduction

1.1. Motivation

The increasing demand for autonomous robot workers calls for reliable and robust robotic navigation systems. Traditional motion planners and localization methods such as Simultaneous Localization and Mapping (SLAM) [1], Dynamic Window Approach (DWA) [2], and Adaptive Monte Carlo Localization (AMCL) [3] depend on manually designed feature extraction and a priori maps of the environment. These classical methods require extensive parameter tuning and do not generalize well to new environments without repeating the tuning process. Multiple independent modules are each responsible for a single subtask such as vision, planning, or control. Each module tries to optimize for its assigned subtask without considering the resulting joint performance of the system as a whole. In recent years, deep learning has become a prevalent approach for autonomous navigation of unmanned ground vehicles (UGVs). With the ability to map raw input directly to steering commands, deep learning systems can form a single cohesive end-to-end system for motion planning and obstacle avoidance. Classic machine learning methods such as supervised learning show improvement over traditional methods reducing the amount of hand-tuning and providing better generalizability to new environments. However, the data set collection process for supervised learning is time-consuming and the resulting performance is highly dependent on the quality of the data set.

Deep reinforcement learning (DRL) promises to alleviate these shortcomings with the ability to train models using only direct feedback from the environment without the need for laborious data collection. DRL has been the go-to approach for the majority of recent autonomous navigation research works including both LiDAR and vision-based systems. Data is gathered through a trial-and-error process which is costly and potentially dangerous in a real-world environment. Therefore, agents are first trained in simulation with improved efficiency after which the system can be deployed on a physical robot. However, training in simulation brings along new challenges as contemporary simulators cannot perfectly mimic the physical world causing the agent to behave differently between simulation and real environments. The accuracy of a simulation generally decreases as the system becomes more complex due to the increased number of assumptions and computations involved.

The design factors studied in this thesis involving backward motion, full-range vision, frame stacking, and reward design can be applied to both camera and LiDAR-based approaches. Since it is harder to conduct this study using a camera-based system we focus solely on LiDAR sensors and leave camera implementations for future investigation. One of our goals is to show that a differential-drive system can perform effective navigation using only LiDAR distance measurements and odometry information thereby limiting system complexity. Limiting complexity reduces the risk of overfitting during training and facilitates the transfer of policies from simulation to the real world. Cameras are harder to simulate compared to simple laser distance measurements from LiDAR. Including camera information in the input can result in more deviations between simulation and reality. Also, by omitting cameras the system utilizes only a single, inexpensive LiDAR sensor to observe its surroundings keeping costs to a minimum. Part of the future vision for this work is to create a real-world platform that can be used for swarm robot experimentation involving a large number of agents making individual robot cost an important factor.

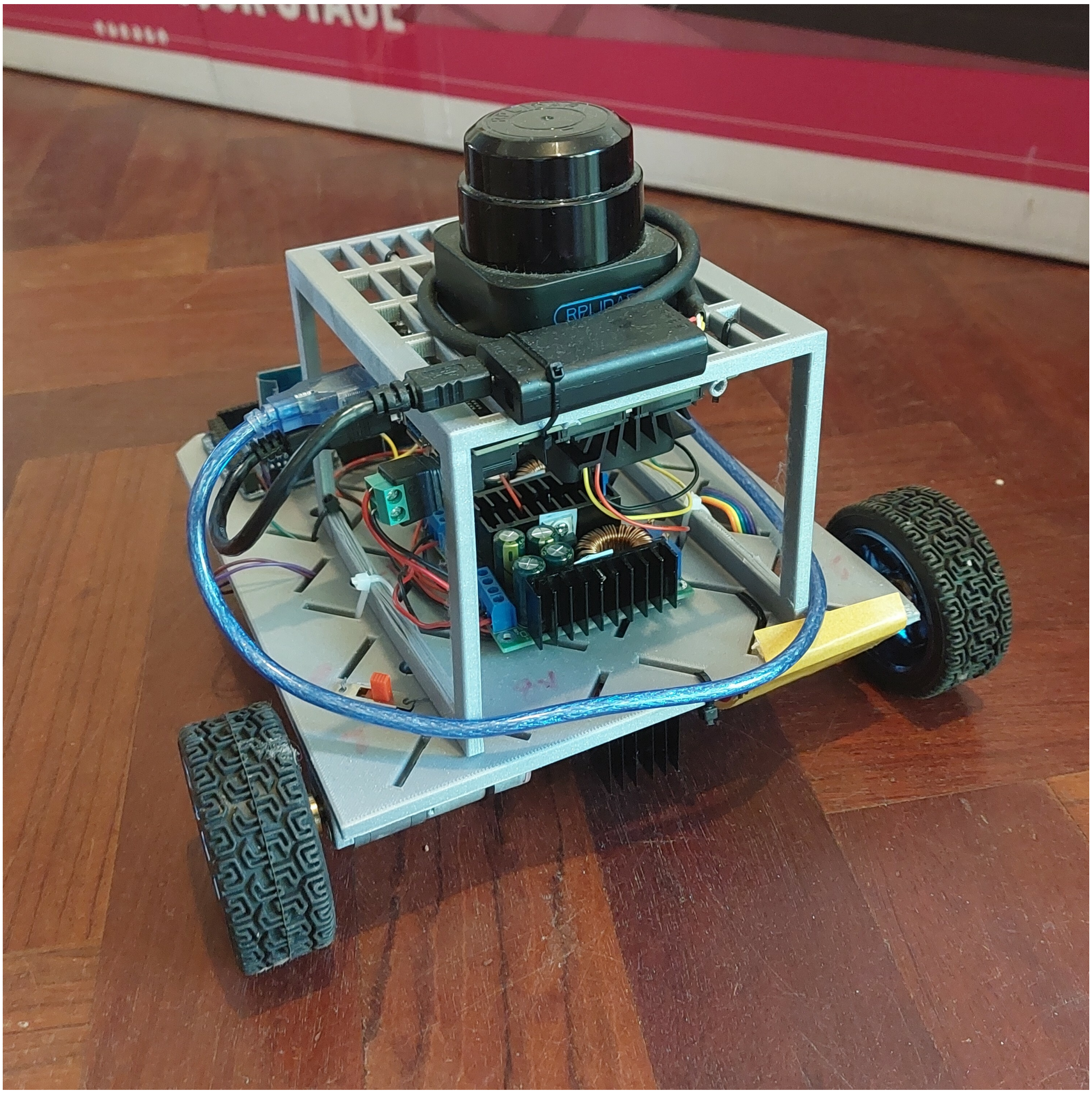


Figure 1.1: The robot platform developed for this thesis.

Most previous works on DRL navigation for mobile UGVs do not consider the effect of certain design decisions and follow common conventions regarding hyperparameters, sensor configuration, reward design, frame stacking, and the direction of motion [4]. In this work, we question these conventions by analyzing the effect of these design factors on the navigation performance of a UGV DRL agent. We compare the results to develop a reliable, mapless, decentralized, autonomous navigation agent with dynamic obstacle avoidance within our new framework. Our developed framework relies on the Robot Operating System 2 (ROS2) middleware suite [5] and the Gazebo simulator [6]. Instead of limiting the robot to forward motion, we expand the sensor range to cover every direction and investigate the usability of backward motion, especially with regard to dynamic obstacle avoidance. We also implement different frame stacking configurations to analyze their effect on dynamic obstacle avoidance. Furthermore, we try to limit certain undesired behaviors such as 'swaying' by introducing specific reward components aimed toward increasing real-world transferability. The resulting system is successfully demonstrated in unseen environments in simulation and on a self-developed physical mobile robot that operates fully independently without a centralized entity. The ultimate goal of this thesis is to provide the research community with design guidelines for developing DRL UGV navigation systems.

1.2. Research Objectives

The following research questions represent the core objectives that this thesis aims to achieve:

- How can we develop a DRL stack for reliable real-world differential drive UGV navigation while limiting system complexity?
- To what extent do different hyperparameters, off-policy algorithms, and reward functions influence training and navigation performance for UGVs?
- How can we design a reward function to restrain undesired behavior to facilitate the transfer to a physical system?
- What are the effects of frame stacking and backward motion on the overall performance and dynamic obstacle avoidance capability of the system?

1.3. Contributions

In this thesis we present the following key contributions:

1. We developed an end-to-end framework for DRL-based UGV navigation including three off-policy DRL algorithms.
2. We investigated the effect of different off-policy algorithms, hyperparameter configurations, and reward functions for UGV navigation.
3. We examined frame stacking and backward motion for dynamic obstacle avoidance.
4. We demonstrate the system in the real world using a self-developed mobile robot with fast-moving obstacles and challenging environments to validate the system.

1.4. Thesis Overview

The remainder of the thesis is divided as follows. Chapter 2 discusses the related work on DRL autonomous navigation and dynamic obstacle avoidance. Chapter 3 discusses the problem setting and gives a system overview describing both the software and hardware. Chapter 4 discusses the results of experiments in simulation regarding different agent configurations and discusses generalizability. Chapter 5 discusses frame stacking and backward motion for dynamic obstacle avoidance. Chapter 6 defines the different reward components and discusses their effect on performance and undesired behavior. Chapter 7 lays out the results of real-world experiments and showcases the capabilities of the agent through demonstration. Finally, Chapter 8 concludes the thesis with a summary and discusses limitations and future directions for the work.

2 Related Work

2.1. Deep Reinforcement Learning Navigation

The navigation problem can generally be described as a point-to-point movement in which the robot has to travel from start to origin while avoiding obstacles. To simplify the problem, most research has focused on a two-dimensional setting in which the robot moves along a flat plane.

Deep Reinforcement learning (DRL) has been widely applied for local navigation and obstacle avoidance because of its strong ability to represent and solve complex nonlinear environments. When compared to classical navigation methods, DRL also has the advantage of being mapless and not needing to rely on high sensor accuracy. This makes DRL navigation relatively robust and allows it to generalize to different environments without the need for extensive parameter tuning.

Pfeiffer et al. [7] presented target-based autonomous navigation combining a deep neural network with a Convolutional Neural Network (CNN) using dense laser scans for supervised learning. Training data was collected in simulation using the ROS navigation package combined with expert demonstrations. Their robot was able to navigate unknown scenarios but was tested only in smaller environments and required interventions during real-world evaluation. Moreover, the performance of the agent relied heavily on the quality of the labeled training sets.

One of the earlier DRL navigation research works stems from 2016 in which Duguleana and Mogan [8] combined Q-learning with a neural network to form a DQN motion planner which could output one of three discrete actions: moving forward, turning left, or turning right. In order to reduce the state space the environment was divided into eight angular sections. Their work was able to navigate in simple environments and paved the way for more complex DRL navigation research.

In 2017, Tai et al. [9] developed low-cost mapless DRL-based navigation for mobile robots mapping sparse LiDAR distance readings to continuous actions. Successful experiments in unseen environments with static obstacles confirmed the transferability of the model to the real world. However, due to the low laser count and simplistic training environment, the model shows weak performance on dynamic obstacles.

Choi et al. [10] replaced the conventional 360° LiDAR with a 90° depth camera and novel Long Short-Term Memory (LSTM) network which outperforms the wide FOV memoryless agents. In addition, to close the gap between simulation and the real world the authors implemented dynamics randomization by adding noise to the scan readings, robot velocity, and control frequency. The trained model inherently learns these uncertainties and as a result, is more robust to the unpredictable dynamics of the real world.

Surmann et al. [11] implemented a system to train many DQN agents in parallel using a lightweight 2d simulation each with a unique environment. However, their work does not include dynamic obstacles and is still not entirely collision-free. Contrary to the authors' claims, our work shows that Gazebo can be used to effectively train an autonomous navigation agent with good generalization capabilities.

Nguyen et al. [12] demonstrated that DRL-based LiDAR navigation can also be applied to different robotic models such as a four-wheel omnidirectional robot. Simulation experiments show that the agent is able to move between waypoints while avoiding static obstacles.

Weerakoon et al. [13] developed a navigation robot based on 3D LiDAR and elevation maps to perform reliable trajectory planning in uneven outdoor environments. Their network makes use of a Convolutional Block Attention Module to identify the regions in the visible environment on which the robot has reduced stability. While their method achieves a high success rate compared to the dynamic window approach, the robot still struggles with steep ditches and boundaries between different surfaces for which it needs to rely on depth sensors and RGB cameras.

2.2. Dynamic Obstacle Avoidance

To achieve better performance on dynamic obstacles, Long et al. [14] developed a PPO-based multi-robot motion planner which feeds three consecutive frames from a 180° scanner into a CNN network. In addition, the authors make use of transfer learning between incremental stages to accelerate learning and achieve better final performance. The resulting system can effectively avoid collisions in a simulated multi-robot scenario without centralized orchestration. Their work is evaluated by Fan et al. [15] showing good performance in crowded real-world environments. However, the setup requires fairly expensive hardware and the robot is not able to move backward to actively escape high-velocity obstacles.

In 2018, Everett et al. [16] developed a multi-agent GA3C-based collision avoidance algorithm that does not make any assumptions about the behavior of other agents. Their method makes use of an LSTM module to deal with the variable number of up to ten other agents in the environment, where the closest agent will have the biggest influence. In addition, instead of training a single agent the experiences of multiple agents are collected into the training batch each episode to maximize the joint reward.

Similarly, Liu et al. [17] incorporated structural RNNs into a PPO-based neural network to deal with the unpredictability of human trajectories in dense crowds. Two separate RNNs are each designated to either the spatial or the temporal relation with the nearby humans visible on camera allowing the agent to reason about their trajectories. Their system outperforms other navigation approaches such as ORCA and can handle a large number of humans, but is not compared with other deep reinforcement learning approaches.

Wang et al. [18] demonstrated a novel approach to autonomous navigation using Deep MaxPain, a modular DRL method involving two different policies for reward and punishment akin to the animal brain. The authors propose a state-value dependent weighting scheme based on a Boltzmann distribution that automatically mixes the ratio between the two separate signals for the final joint policy, which was previously done by hand. The scale between the reward weight and punishment weight also determines the balance between exploration and exploitation. Both LiDAR scans and RGB-camera images are used as input for the neural network. The robot was trained and simulated both in simulation and on a real robot achieving safe navigation and outperforming DQN.

It is important to note that with complex environments that involve large continuous obstacles DRL navigation can get stuck in a local range. To be successful in such environments DRL navigation requires global information which can be provided by a high-level planner based on more classical techniques.

Kato et al. [19] developed such a long-range DRL navigation system by combining a local DDQN agent with a topological map global planner. While the resulting agent was able to successfully navigate among people in the real world, the speed of pedestrians was limited to 0.5 m/s. Moreover, the use of a discrete algorithm requires a fixed set of outputs to be defined while the variable speed of pedestrians requires more flexibility. Similarly, Gao et al. [20] combined TD3 with Probabilistic Road Maps (PRM) to create a long-range motion planner for indoor environments. Their simulation results confirm that the trained policy can generalize to larger, unseen environments.

Gao et al. [21] introduced a new method focused on detecting irregular obstacles by fusing laser scan data and RGB camera data. The fused data is then passed through a newly developed depth slicing method to obtain pseudo-laser data which encodes both depth information and semantic information combining the advantages of both. By making use of an LSTM module the field of view can be reduced to 90° without a significant performance decrease which allows for the use of an inexpensive depth camera. While the system shows decent performance during evaluation, the tested environments are relatively simple and contain few dynamic obstacles.

Ejaz et al. [22] improved the training efficiency for depth camera D3QN-based autonomous navigation by applying layer normalization and injecting Gaussian noise into the fully connected layers. Layer normalization reduces the computational cost as it prevents domination by gradients from larger parameters. The factorized Gaussian noise is injected directly into the weights and biases of the linear layers to stimulate better exploration leading to shorter training times. Their method outperformed several DQN variants in simulation and was validated on a real robot.

Corsi et al. [23] incorporated domain-expert knowledge into the DRL training process for mapless navigation to avoid undesired behaviors, such as swaying, and improve performance. This is implemented by connecting scenario-based programming (SBP) constraints to the cost function. Whenever the agent selects an action that is mapped to a blocked SBP event the cost is increased which allows explicit constraints to be injected directly into the policy optimization process. While their method achieves good performance in simulation, the SBP events need to be designed manually and require expert domain knowledge.

In this thesis, we aim to develop a DRL indoor navigation system that provides reliable performance while also preserving simplicity. Liu et al. [17] and Gao et al. [21] achieved good performance for indoor navigation but required a complex architecture consisting of multiple modules. Unlike the works by Everett et al. [16], Fan et al. [15], and Gao et al. [21], we omit the use of a camera to minimize the simulation-to-reality gap and focus solely on LiDAR distance readings. By making full use of off-policy algorithms through hyperparameter tuning and reward function customization our agent can achieve a high success rate without the need for additional modules or a camera. Similarly to Corsi et al. [23], we aim to enforce specific desired behaviors upon the agent. However, instead of relying on manually designed SBP constraints, we make use of simple reward components to achieve a similar effect.

The works by Choi et al. [10], Liu et al. [17], and Gao et al. [21] include real-world experiments but these are limited to slow-moving pedestrians and simple obstacle configurations. Furthermore, their robots are unable to handle nearby obstacles that have escaped the robot's field of vision and cannot deal with obstacles coming from behind. Instead of relying on short-term memory and limited FOV we equip the robot with $360°$ LiDAR scans to continually monitor its full surroundings and anticipate obstacles coming from any direction. We also aim to show that having a full range of sensing enables the robot to safely move backward allowing it to deal with fast-moving obstacles and unexpected situations. The included real-world demonstrations will show that backward movement allows the robot to react quickly when an obstacle suddenly appears or changes direction.

2.3. Background

2.3.1. Reinforcement Learning

Reinforcement Learning (RL) is a computational approach to understanding and automating goal-directed learning and decision-making [24]. The goal of an RL agent is to select a policy that maximizes the total reward it receives when interacting with an environment [25]. The training process of reinforcement learning is often modeled as a Markov decision problem (MDP). An MDP is defined by the tuple (S, A, P, R, ρ_0), where:

- S denotes the state space, the set of all valid states;
- A is the action space, the set of all valid actions;
- $P(s'|s,a)$ is a transition probability function that defines the probability of transitioning from the current state s to the next state s' after an agent takes action a;
- $R(s,a,s')$ is the reward function that defines the immediate reward r that the agent observes after taking action a and the environment transitions from s to s'.
- ρ_0 is the state distribution at the start.

The total *return* (or reward) starting from time t until the end of the interaction between the agent and its environment is expressed as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.1}$$

where r_t is a random variable that models the immediate reward and $\gamma \in [0, 1)$ is a discount factor that weights future rewards [26]. *Value functions* represent the expected return of being in a state or taking a particular action. The state-value function $v_\pi(s)$ gives the expected return from state s following policy π:

$$V^\pi(s) = E_\pi[G_t | s = s_t] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s = s_t\right] \tag{2.2}$$

The action-value function (or Q-function) $q^\pi(s, a)$ gives the expected return of taking action a in state s and following policy π thereafter. The action-value function can be expressed in terms of the state-value function using the Bellman equation as follows:

$$Q^\pi(s, a) = \sum_{s' \in S} P(s'|s, a)\left[R(s, a, s') + \gamma V^\pi(s')\right] \tag{2.3}$$

The action selection process of an agent is governed by its policy, which in the general stochastic case yields an action according to a probability distribution over the action space conditioned on a given state $\pi(s, a)$.

The ultimate goal of the agent is to find the optimal policy π^* which is the policy that maximizes the total cumulative reward for the system. To do so the agent needs to find the optimal state-action value function that indicates the maximum possible reward for each action given a specific state. The maximum reward given a specific state is equal to the expected reward when taking the best possible action from that state:

$$\max_a Q^*(s, a) = V^*(s) \tag{2.4}$$

Finally, combining equation 2.3 and equation 2.4 the optimal state-action value function that the agent aims to find can be defined as:

$$Q^*(s, a) = \sum_{s' \in S} P(s'|s, a)\left[R(s, a, s') + \gamma \max_a Q^*(s', a')\right] \tag{2.5}$$

2.3.2. Off-Policy Algorithms

In recent years numerous different types of DRL algorithms with specific properties and applications have emerged from research work. These algorithms can be divided into different categories, including the category of on-policy and off-policy algorithms. on-policy algorithms, such as A3C, TRPO, and PPO [27], update the Q-function based on the action taken according to the current policy. Off-policy algorithms, such as DQN, DDPG, and TD3, update the Q-function according to the best possible action to be taken in the current state, regardless of which action the current policy proposes. With off-policy algorithms, the experience collection and training process can essentially be decoupled from each other and run separately. Stored experiences can be used to train the current policy regardless of which policy they were collected by. This allows the agent to perform exploration while simultaneously optimizing the current policy. In addition, off-policy algorithms allow for an experience replay buffer so that previous experiences can be reused resulting in greater sample efficiency [28]. For this reason, we will consider only the off-policy algorithms which are commonly used for mobile robot navigation [4].

Deep Q-Network Deep Q-network (DQN) is a value-based DRL algorithm that combines the concept of tabular Q-learning with a deep neural network to allow for problems with much higher complexity [29]. The most important limitation of the original Q-learning method is that every possible state-action pair needs to be represented as an entry in the Q-table rendering it infeasible for solving complex problems [30]. Thanks to recent advancements in neural networks this limitation has largely been solved by replacing the Q-table with a deep neural network which instead approximates the Q-function allowing for problems with much larger dimensionality. DQN forms the most basic variant of a model-free off-policy DRL algorithm and we use it as the starting point for the autonomous navigation system.

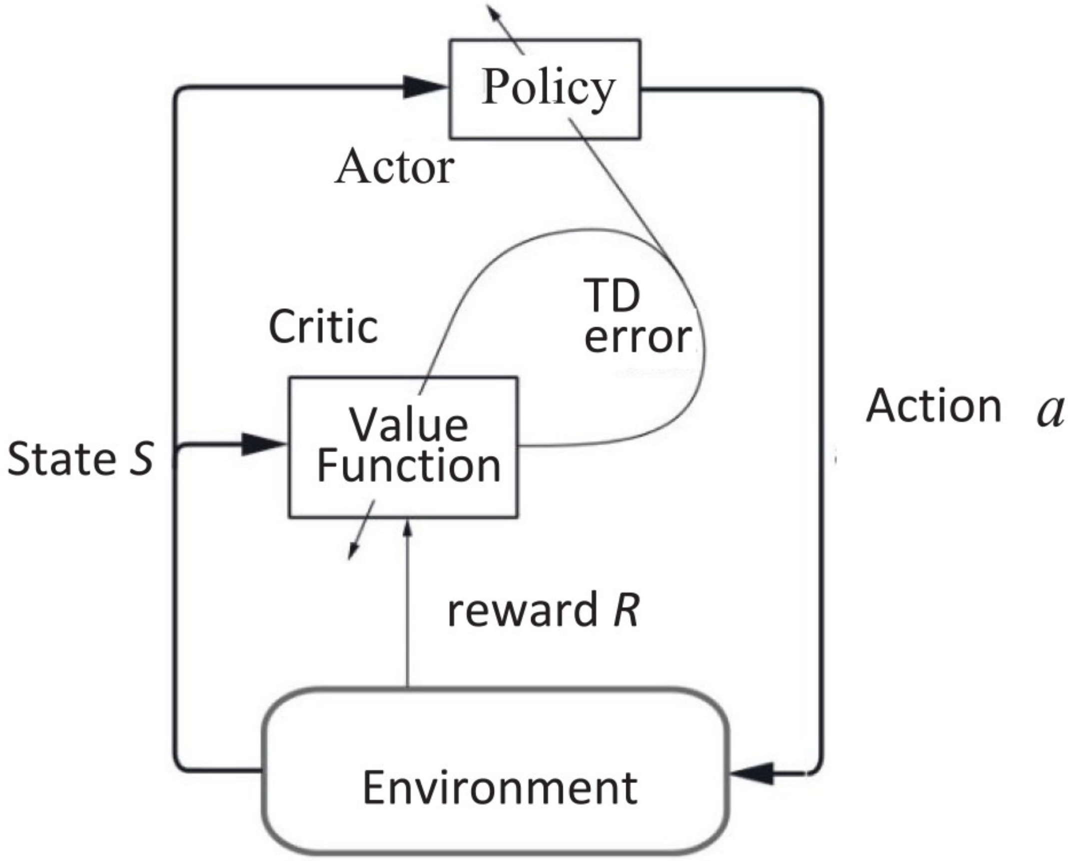


Figure 2.1: A schematic overview of the actor-critic pattern present in DDPG, taken from [31].

Deep Deterministic Policy Gradient Deep Deterministic Policy Gradient (DDPG) is a model-free off-policy actor-critic algorithm designed for continuous environments [32, 33] making it a more suitable candidate for autonomous navigation compared to DQN. Unlike DQN, DDPG can output actions with a continuous range allowing for more precise movement control. Figure 2.1 presents a schematic diagram for the DDPG algorithm. Instead of a single network, DDPG features two separate neural networks: a policy (actor) network and a value (critic) network which makes it an actor-critic algorithm. At each time step, the actor network takes the environment state as input and outputs the most appropriate action according to the current policy. The critic network concatenates the state observation with the action proposed by the actor to output a Q-value which indicates the quality of that specific action given the current state. The policy is then optimized according to the value assigned to the current state-action pair. However, compared to DQN DDPG requires more tuning of hyperparameters to achieve optimal performance as will be discussed in Section 4.

Twin Delayed Deep Deterministic Policy Gradient Twin Delayed Deep Deterministic Policy Gradient (TD3) is a subsequent iteration of DDPG which introduces three improvements that address the Q-values overestimation problem of DDPG which can lead to policy breaking [34]. The first and largest improvement is the introduction of a second Q-function (critic network) which learns separately from the first Q-function. At each training step, the smaller of the two Q-values is used as the target in the Bellman error loss function which reduces the probability of overestimating the Q-value. Secondly, noise is added to the target actions to smoothen out the Q-function over similar actions, essentially serving as a regularizer for the algorithm. Without this adjustment, the function approximator error can cause the critic to develop sharp incorrect peaks for certain actions which the policy will try to exploit leading to incorrect behavior as the true value of these actions is much lower than estimated. By explicitly forcing similar actions to have a similar value the negative effect of the function approximator error can be reduced. Lastly, the policy network is made to update less frequently than the Q-function.

This effectively allows the Q-function to minimize the approximation error before generating Q-values to update the policy. The delayed policy update is implemented on top of the target networks which prevents the model from chasing ever-changing targets that would lead to instability.

3

System overview

The following chapter gives an overview of the theory behind the system and describes the different tools and methods used for the experiments.

3.1. Local Navigation

The goal of this thesis is to develop a reliable, mapless, and decentralized motion planner for an unmanned mobile ground vehicle. The robot needs to navigate from the starting location to the goal while maintaining a safe distance margin to any obstacle. Essentially, we attempt to find the following optimal translation function:

$$v_t = f(o_t, p_t, v_{t-1}) \tag{3.1}$$

where for each time step t, o_t is the current set of readings from raw sensor data, p_t is the current estimated position of the robot, and v_{t-1} is the velocity of the robot during the previous time step. The model is trained to approximate this optimal translation function and directly maps the input observations to output action v_t, which holds the next velocity target for the robot.

Observation space The observation space O_N consists of N LiDAR distance readings spaced evenly over 360° around the robot, where N is specified for each model. To increase the robustness of the policy we add Gaussian noise to the laser distance readings to reduce the simulation-to-reality gap. The laser scan inputs are normalized to fall within the range of $[0, 1]$. The other inputs are normalized to the range $[-1, 1]$. Furthermore, the distance and angle to the goal are derived from odometry information and concatenated with the LiDAR readings. Lastly, the previous linear and angular velocities of the robot are included to form the entire observation set.

Action space The action space A is a two-dimensional vector that defines both the desired linear velocity and angular velocity of the robot at each time step. Before being sent to the motor control unit, both output velocities are fed through a tanh cell to obtain the normalized range $[-1, 1]$. Depending on the model configuration, the linear velocity ranges either from zero velocity to maximum forward velocity or from maximum backward velocity to maximum forward velocity. The angular velocity always ranges from maximum clockwise velocity to maximum counter-clockwise velocity.

3.2. Method

This section describes the tools and methods used for the experiments in simulation and also discusses the system architecture as a whole.

3.2.1. Simulator

Agents are trained in simulation to largely automate and accelerate the training process without the risk of damaging equipment. However, contemporary simulators cannot perfectly model the complexity of

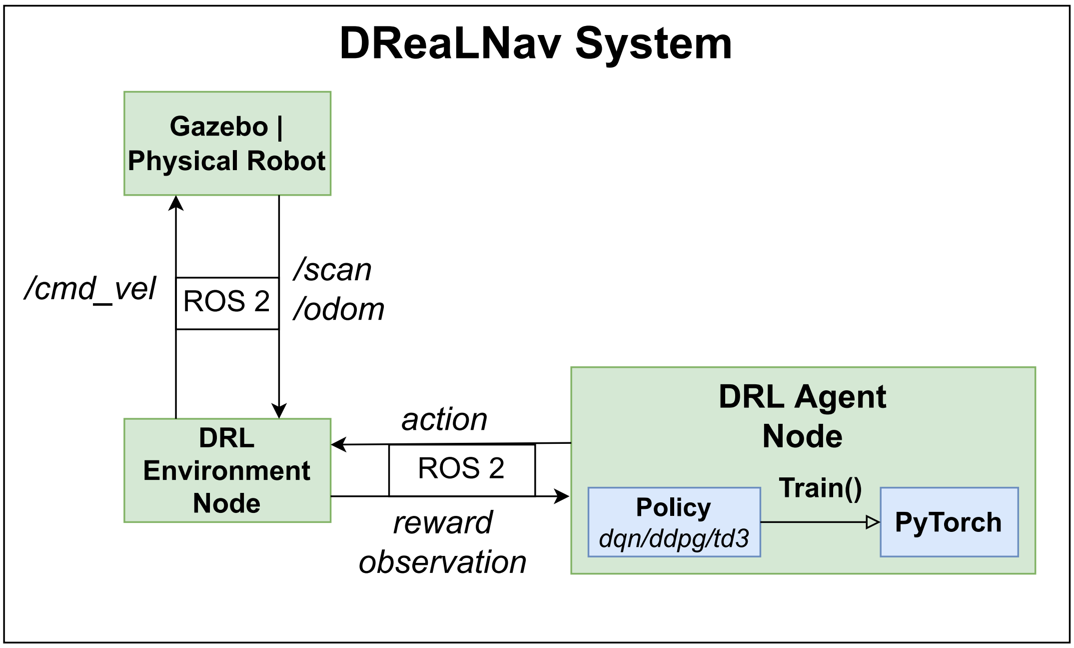


Figure 3.1: System architecture for the navigation stack. The nodes and communication layer are implemented using ROS2. The DRL environment node provides an interface to facilitate switching between the Gazebo simulation and the physical robot.

the physical properties of reality and thus introduce discrepancies leading to the simulation-to-reality gap. [35]. The simulation-to-reality gap can be partially amended by introducing noise to the simulation to lower the dependency of the policy on the precision of input data. Selecting a simulator is a non-trivial decision as there is generally a trade-off between performance, accuracy, and flexibility. The simulation experiments in this thesis are performed using the Gazebo simulator[6], which is also the most frequent choice among the prior works cited in Chapter 2. Gazebo is a popular open-source 3D robotic simulator with a robust physics engine and wide support for ROS, mobile robots, and optical sensors. For our use case, Gazebo offers the best balance between performance, accuracy, and speed of implementation.

3.2.2. Environment

Four different training and testing scenarios were built as showcased in Figure 3.2. The first scenario involves an area of approximately 4.2×4.2 meters surrounded by walls with no dynamic obstacles and mainly serves as a control scenario. The second scenario involves no static obstacles other than the perimeter walls and six dynamic obstacles to test how well an agent can deal with dynamic obstacles. The third scenario involves seven additional static walls placed across the area and two dynamic obstacles and forms the main scenario used during the majority of the experiments as it provides a good balance in difficulty. The fourth and last scenario is similar to the third scenario with four additional dynamic obstacles. The last scenario functions as the final and most difficult test for the best-performing policies. Goal positions are designated randomly from valid locations with a sufficient distance margin to any obstacle.

3.2.3. Training Setup

The policies are implemented using PyTorch and trained on a computer equipped with an AMD Ryzen 9 5900HX and CUDA-enabled RTX 3050 ti GPU for approximately 40 hours. During the training process, the outcome per episode and the average reward per episode are recorded and visualized in a graph.

Each trained policy is evaluated for 100 episodes on the same stage it was trained on unless specified otherwise, during which the following evaluation metrics are collected for all of the experiments:

- **Success Rate** - The percentage of trials in which the robot reaches the goal.
- **Collision (Static)** - The percentage of trials in which the robot collides with a static obstacle.

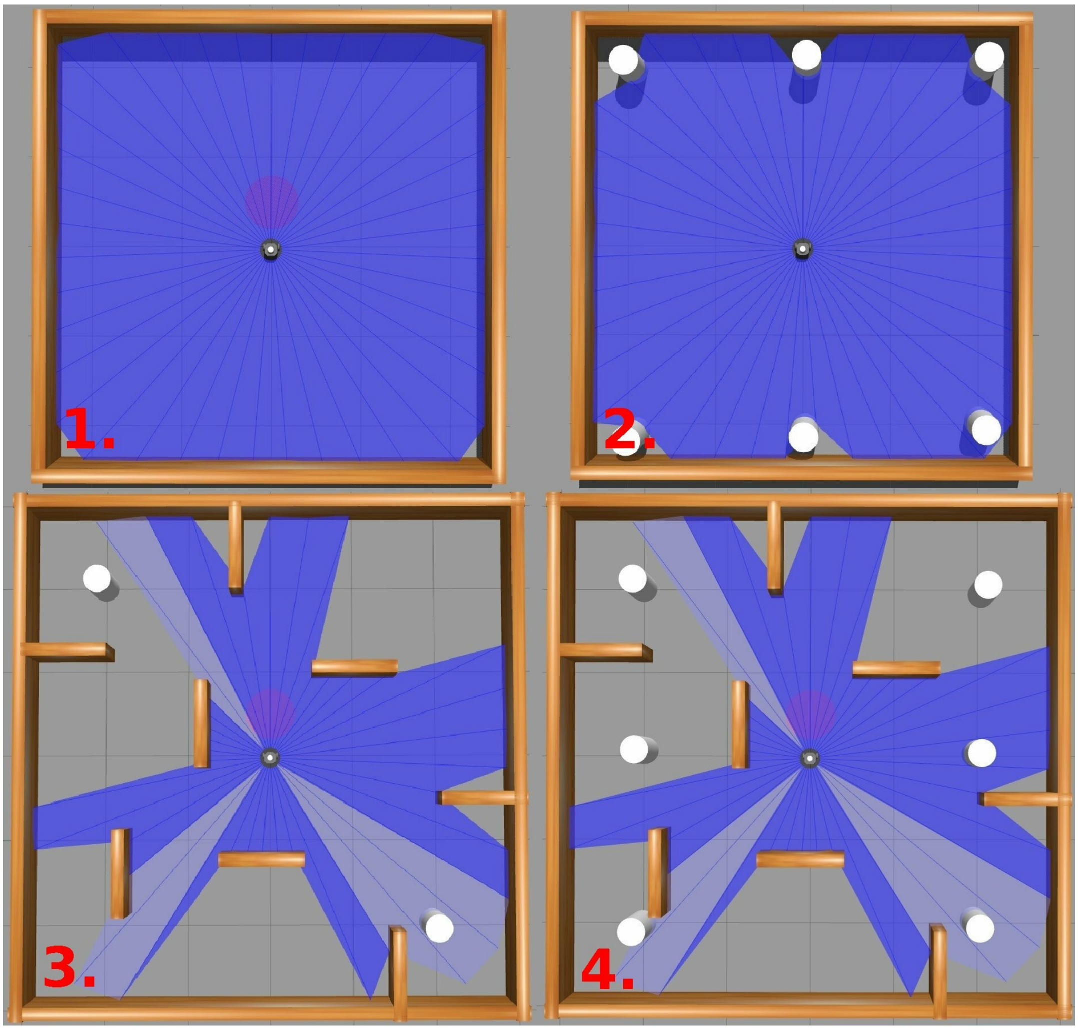


Figure 3.2: The different stages used during training and evaluation in simulation, **TL**: Stage 1, no obstacles (4.2×4.2 m), **TR**: Stage 2, six dynamic obstacles (4.2×4.2 m), **BL**: Stage 3, Static and two dynamic obstacles (6×6 m), **BR**: Stage 4, Static and six dynamic obstacles (6×6 m).

- **Collision (Dynamic)** -The percentage of trials in which the robot collides with a dynamic obstacle.
- **Timeout** - The percentage of trials in which none of the other outcomes happen within the specified time limit.
- **Average Distance** - The traveled distance in meters averaged over all successful trials.
- **Average time** - The elapsed time in seconds from start to goal averaged over all successful trials.

For some of the experiments, the following additional metric is also collected:

- **Sway Index** - A measure of how frequently the robot changes its angular velocity output possibly causing the robot to sway.

3.3. Physical System

As part of the DRL autonomous navigation platform, an actual physical robot was developed which can employ the policies trained in simulation to perform real-world autonomous navigation. This section gives an overview of the hardware configuration for the robot. The robot is designed with two key characteristics in mind: cost and customizability. The aim is to create an inexpensive robot while maintaining the ability to scale its computational capabilities according to the demands of the task at hand.

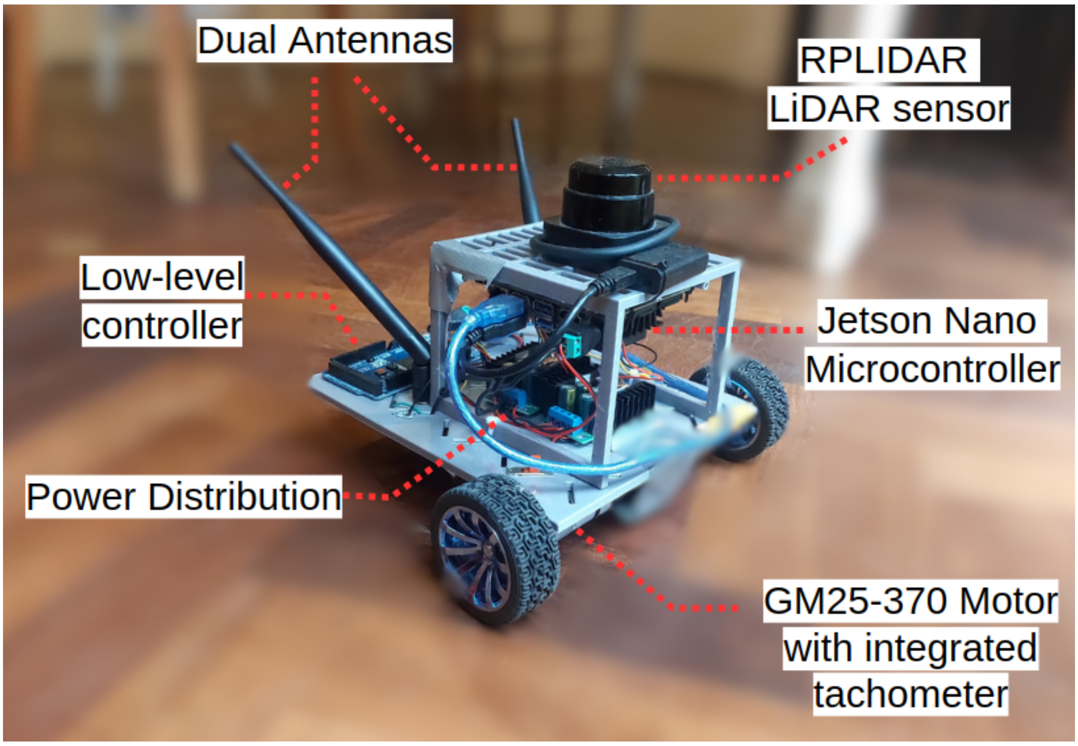


Figure 3.3: An overview of the robot hardware.

In the following section, we elaborate on the details of the hardware description.

Mainboard As the main controller for our system, we selected the Jetson Nano board developed by Nvidia running Linux Ubuntu 20.04 as its operating system. The Jetson Nano features a dedicated 128-core Maxwell GPU with the capability to run neural networks efficiently while preserving battery power. The board possesses sufficient computational power to run the trained DRL policies and is available for less than €100 per unit. In addition to being affordable, the Jetson module gives a lot of flexibility as the board can easily be swapped out for any of the other modules from the Nvidia Jetson family.

LiDAR The current setup is equipped with the S1 RPLIDAR laser range scanner from Slamtec. The S1 RPLIDAR offers up to 720 scan samples distributed over 360° around the robot at a maximum frequency of up to 15 Hz. While the S1 RPLIDAR is relatively costly at approximately €600, it can easily be exchanged for one of the cheaper liDAR variants from Slamtec such as the RPLIDAR A1 which is sold at €100. While the less expensive RPLIDARs are less performant, they all offer a sufficient range, accuracy, and sampling frequency for indoor navigation. For our application, we only require a total of 40 scan samples at a sampling frequency of 10 Hz with a maximum range of 3.5 meters. We modified the RPLIDAR driver software so that only the 40 samples that are of interest are given to reduce the overall latency of the system.

Low-level controller In order to simplify the design process an Arduino Mega 2560 is added as a second controller for the low-level functions of the robot. The Arduino mega handles the PWM signals for motor control and takes care of the interrupt handling for the tachometers. To receive motor control commands and forward the tachometer counts the Arduino is connected to the Jetson Nano through a UART channel over which ROS messages are sent using the Rosserial Arduino ROS package.

Motors and tachometer For the motors, we use two Chihai Gm25-370 300 RPM DC gear motors with an integrated interrupt-based magnetic encoder that functions as a tachometer. The tachometers provide odometry to the robot based on kinematic calculations as seen in [36]. The motors are connected to the rest of the system through an L298N DC motor driver module which regulates the speed and direction of the motors.

Power and Chassis The entire system is powered by three rechargeable 4.2 V Li-Po batteries which are connected to two DC-DC boost converter modules to provide a 5V power source for the Jetson Nano and a 12V power source for the Arduino Mega and L298N motor driver module. The plastic chassis and frame on top of which the LiDAR is mounted are produced using a 3D printer.

4

Navigation Performance

This section will present the results of the simulation experiments concerning different hyperparameter and laser scan configurations to showcase the effect on navigation performance. Furthermore, the different off-policy algorithms are compared and the best-performing policy is evaluated in different stages to show the generalization capability of the system.

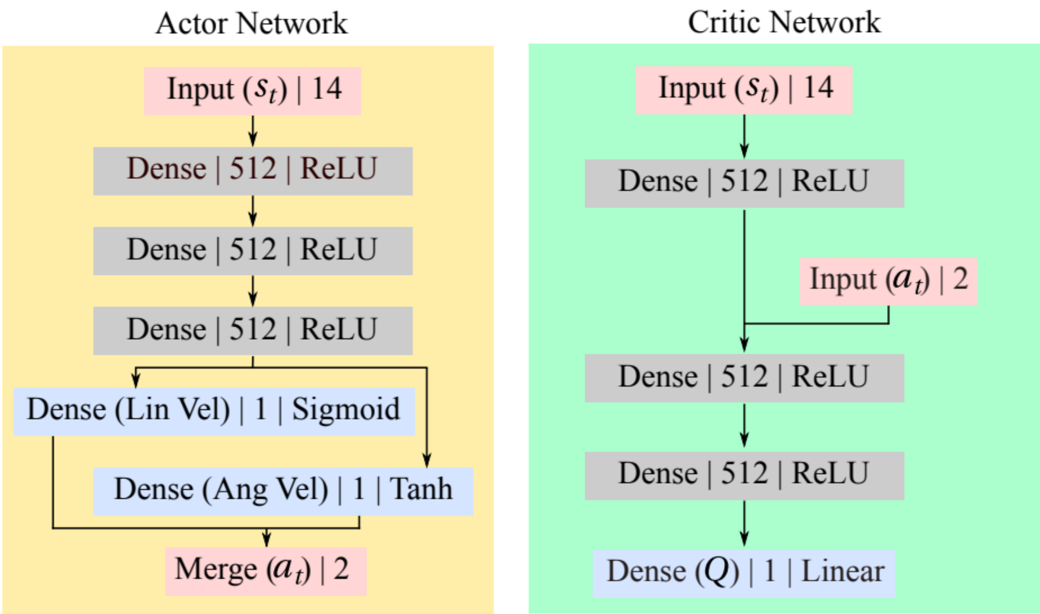


Figure 4.1: The initial network structure for the base DDPG model is taken from [9]. The dense layers represent fully connected linear layers.

4.1. Hyperparameter Tuning

As DDPG forms the basis for most off-policy actor-critic algorithms, we start our experiments using the DDPG algorithm to evaluate and select hyperparameter values that are common to all actor-critic algorithms. While finding the exact optimal hyperparameter configuration is difficult and forms a research field of its own [37, 38, 39], by heuristically selecting between extreme values and evaluating the effect on the agent's navigation performance we can achieve a sufficiently high success rate. We gradually adjust each parameter based on the preceding results and then retrain the policy under the same conditions. Figure 4.1 presents the initial structure for the actor and critic networks which is based on the work by [9]. Table 4.1 shows the results of the first experiment conducted in stage 3 with the DDPG hyperparameter configurations and corresponding evaluation metrics. In addition, Figure 4.2 shows the

Table 4.1: Turtlebot3 navigation performance for DDPG configurations in stage 3.

Training					Evaluation						
Model	**BS**[a]	**RB**[b]	**Discount**	**LR**[c]	**Success**	**CS**[d]	**CD**[e]	**TO**[f]	**Avg. Dist**	**Avg. Time**	**Avg. Speed**
DDPG 1[g]	128	1e+5	0.99	1e-3	100	0	0	0	3.45	20.11	0.78
DDPG 1	128	1e+5	0.99	1e-3	67	31	0	2	3.44	16.22	0.96
DDPG 2	128	1e+6	0.99	1e-3	75	13	7	5	3.94	17.12	0.99
DDPG 3	512	1e+6	0.99	1e-3	92	1	6	1	3.54	16.11	0.99
DDPG 4	1024	1e+6	0.99	1e-3	97	0	3	0	3.31	16.35	0.92
DDPG 5	1024	1e+6	0.99	3e-4	**99**	0	1	0	3.02	15.67	0.88
DDPG 6	1024	1e+6	0.99	1e-4	94	0	5	1	4.05	18.97	0.97
DDPG 7	1024	1e+6	0.999	3e-4	88	1	11	0	4.94	21.13	**1.00**

[a] Batch Size
[b] Replay Buffer Size
[c] Learning Rate
[d] Collision Static
[e] Collision Dynamic
[f] Timeout
[g] Evaluated in stage 1

reward scores over time during training averaged over 100 episodes. The initial parameters for model DDPG 0 were based upon [9] and the baseline implementations by OpenAI [1]. First, DDPG 0 is evaluated in stage 1 with no obstacles as a control condition where it successfully reaches the goal in 100% of the trials, confirming the expected behavior. Next, the model is evaluated in stage 3 containing static and dynamic obstacles on which it was trained resulting in only a 69% success rate with most failures being static collisions. Evidently, the current configuration is not sufficient for our environment setup and further investigation of the hyperparameters is needed to improve performance. We increased the laser scan density from 10 to 40 individual samples for DDPG 1 since the laser scan samples were too sparse for the initial configuration, impeding the ability of the agent to detect obstacles on time. While this adjustment initially gives no improvement, we hypothesize that as the other hyperparameters become more well-tuned the increase in scan samples will start to take effect on performance as will be discussed in this section.

Replay Buffer Size Looking at the reward graph for DDPG 1 in Figure 4.2 we see that while the agent is able to learn a viable policy it remains unstable over the entire training session with large oscillations and drops in the reward curve. The first large drop in performance appears approximately around the 1000 episodes mark which is approximately the number of episodes it takes for the experience replay buffer to fill up. Once the replay buffer is full old experiences will start to be ejected and replaced by newer entries. When the size of the replay buffer is too small the policy will forget important previous experiences and use only the most recent data which can cause overfitting and may lead to catastrophic forgetting [25]. Therefore, we increase the size of the replay buffer for DDPG 2 in an attempt to stabilize the learning process and prevent large relapses in the policy. Figure 4.2 suggests that the increase in buffer size has a positive effect on training stability as the oscillations and drops in the reward curve for DDPG 2 are less severe and the overall reward is higher. Table 4.1 confirms the improvement in performance as the number of static collisions is reduced significantly for DDPG 2. We do not increase the buffer size any further as it should be limited to reduce the probability of storing irrelevant experiences. A larger replay buffer stores older experiences from a previous and usually less performant iteration of the policy when training is stable. Experiences that are too old provide little value ultimately slowing down the training process.

[1] https://github.com/openai/baselines

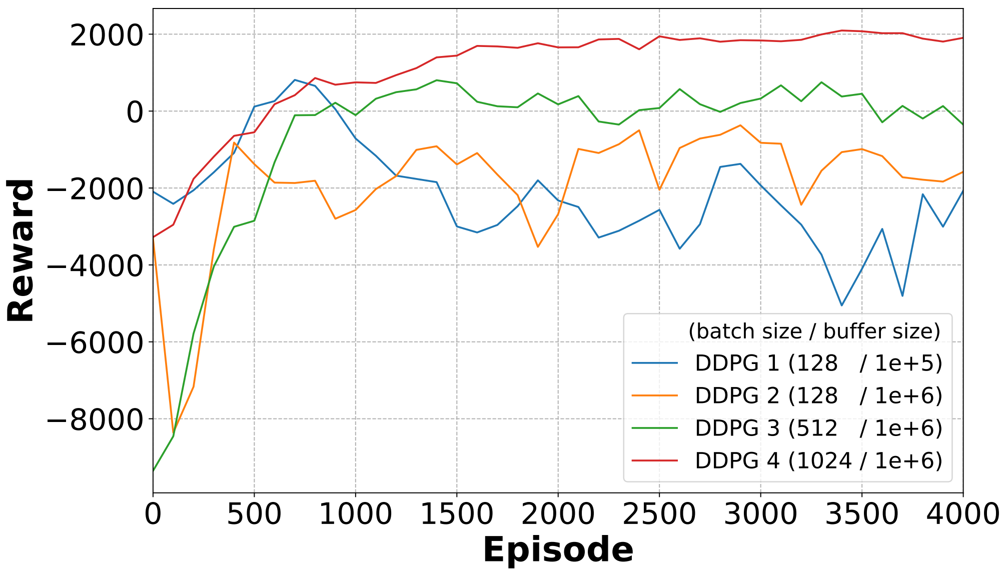


Figure 4.2: The average reward per 100 episodes for different batch size and replay buffer size configurations in stage 3.

Batch Size The batch size is another important hyperparameter that can affect both the speed and stability of the training process. A larger batch size increases the available computational parallelism, increasing the number of samples processed per second, and gives a better estimate of the error gradient as more samples are processed per step. [40, 41]. Consequently, larger batch sizes generally lead to better optima of the objective function and better stability at the cost of slower convergence as more samples need to be processed. However, larger batch sizes can also lead to poor generalization, carry a larger memory footprint and require additional processing power to limit the processing time per step and keep the training process running smoothly. In their work from 2017 Tai et al. [9] configured a batch size of 128 per training step. Since then, the computational capacity of machines has improved and the software libraries for machine learning have been further optimized making it worthwhile to explore larger batch sizes for improved stability and better convergence. Figure 4.2 shows the average reward during training for the evaluated batch sizes. From the graph, it becomes evident that a higher batch size significantly reduces the amplitude of fluctuations in the reward curve improving the stability of the training process. With the highest batch size, DDPG 4 results in a better performance and reward curve compared to the other two configurations as it processes more samples per timestep guiding it toward better optima. We leave the batch size fixed at 1024 for the remaining experiments.

Learning Rate Lastly, the learning rate and tau parameter are two more important similar factors that affect stability and training speed. It is generally recommended to start with a larger learning rate and gradually decrease it until the best result is achieved [40]. With a larger learning rate, the model learns faster but might converge early to a suboptimal solution or not converge at all, whereas a smaller learning rate can provide more stability at the cost of longer training times. Upon observing the reward graph for the different learning rates in Figure 4.3 it does not become immediately clear which model performs best as some of the models feature a downward trend in the reward graph later on in the training process. However, DDPG 5 appears to provide the most stable result and also reaches the highest average reward. A lower learning rate may eventually recover and lead to better optima given more training time. However, part of the objective is to find a balance between performance and training time so we limit the training duration to 40 hours. During the evaluation, DDPG 4 achieved a success rate of 97%. Lowering the learning rate seems to give a slightly more optimal solution for DDPG 5 with a success rate of 99% and lower averages for distance and time. Decreasing the learning rate yet again for DDPG 6 gives no further improvement in performance, hence the learning rate and tau parameters are fixed at $3\mathrm{e}{-4}$.

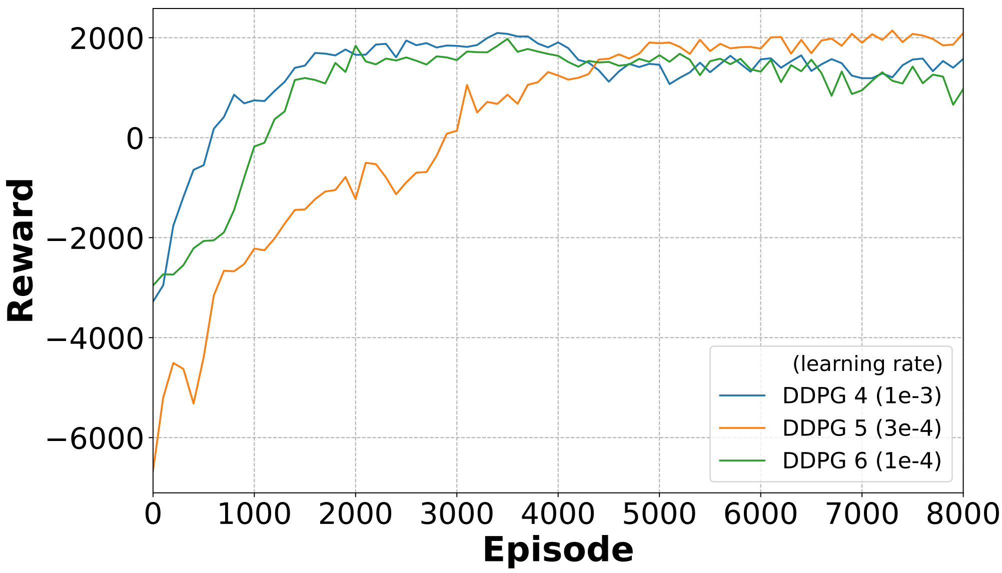


Figure 4.3: The average reward per 100 episodes for different learning rates in stage 3.

4.2. LiDAR Configuration

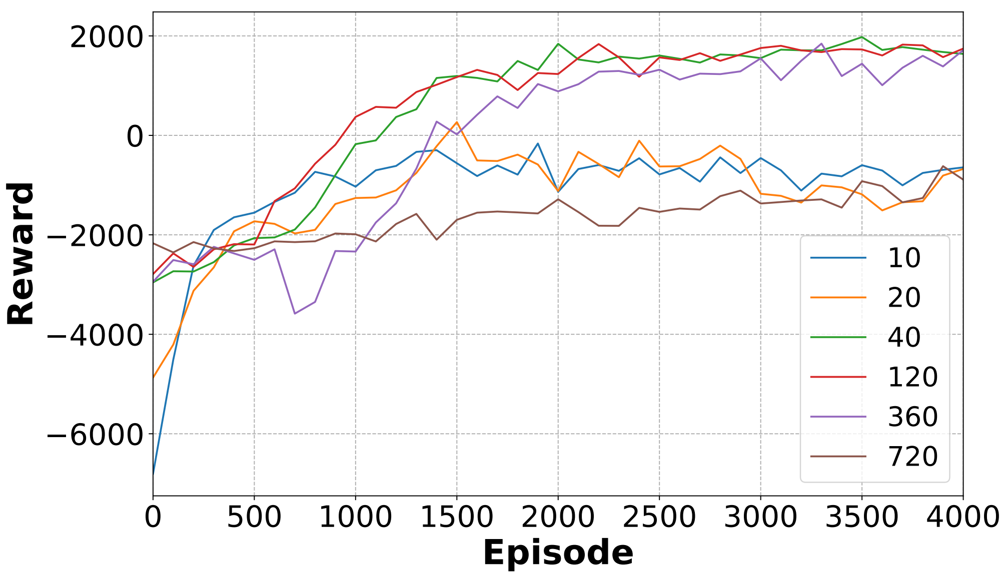


Figure 4.4: The average reward per 100 episodes for different DDPG laser scan densities in stage 3.

After establishing a set of hyperparameters with an acceptable performance we turn our attention to the number of scan samples and their effect on obstacle avoidance. Table 4.2 shows the evaluation results for several DDPG policies with different laser scan densities tested in stage 3. The laser scans are distributed evenly across 360° around the robot. For the current environment, a configuration with less than 40 scan samples severely deteriorates the performance of the agent. This can be explained by the shapes and dimensions of the obstacles that are used in the simulation. Without a high enough scan density, the agent is not able to reliably detect corners of the static obstacles often resulting in a collision when the agent attempts to maneuver around the endpoints of a wall. In addition, with lower scan densities dynamic obstacles might fall in between two adjacent scan points rendering them undetectable until in close proximity to the agent making it harder to perform obstacle avoidance. At 40 laser scan samples, the agent can detect obstacles early enough to avoid collisions and successfully navi-

Table 4.2: Turtlebot3 navigation performance for DDPG configurations in simulation stage 3.

Training		Evaluation					
Model	**Scans**	**Success**	**CS**	**CD**	**TO**	**Avg. Dist**	**Avg. Time**
s10	10	44	36	20	0	2.11	13.51
s20	20	54	37	9	0	2.62	14.82
s40	40	**94**	0	5	1	4.05	19.13
s120	120	91	0	9	0	4.32	20.46
s360	360	91	1	7	1	4.63	21.86
s720	720	87	9	3	1	4.22	19.56

gate the environment resulting in a 94% success rate during evaluation. For the current environment setting, configurations with more than 40 scan samples do not seem to benefit the agent anymore resulting in either no significant difference or slightly worse performance. Configurations with fewer scan samples reduce the number of inputs for the neural network which simplifies the learned policy. Out of the best-performing models, s40 also gives the best performance in terms of average distance and time per episode. Figure 4.4 shows the average reward graphs for the DDPG models with different scan densities, which are in line with the evaluation results. Around the 1500 episodes mark the difference in performance starts to become evident as the rewards for policies with lower scan density policies stagnate. The other policies follow a fairly similar curve with the 40-sample configuration reaching the highest average reward. Note that the minimum detectable obstacle size is dependent on the number of scan samples. With the current configuration, some smaller obstacles such as table legs might not be detected when the robot is still far away. Other environment settings with differently sized obstacles might require more or less scan sample density to achieve optimal performance. However, given that additional laser scans seem to give no significant benefit for our current environment setting, we continue with the 40-scan sample configuration to keep the network input dimensions to a minimum.

4.3. Algorithm Selection

Table 4.3: Navigation performance for different DRL algorithms in stage 4

Algorithm	**Success**	**CS**	**CD**	**TO**	**Avg. Dist**	**Avg. Time**	**Speed**
DQN	79	1	10	10	4.01	20.27	0.90
DDPG	94	0	6	0	**3.93**	**19.93**	**0.90**
TD3	**97**	0	2	1	4.83	25.07	0.88

In order to determine which off-policy DRL algorithm is best suited for our application we train a separate policy for DQN, DDPG, and TD3 with the same configuration and compare the results. Table 4.3 shows the evaluation results for the three different algorithms. As expected DQN performs worst as it is designed for discrete action spaces while the navigation problem corresponds to a continuous domain. DDPG achieves a much higher success rate as it is meant for continuous action spaces and allows for finer movement control resulting in fewer dynamic collisions and timeouts. Unsurprisingly, TD3 achieves the highest success rate as it is essentially an improved iteration of DDPG. However, TD3 does take a slightly more conservative route as it has a lower speed and travels a greater distance on average. Figure 4.5 shows the reward curve for each of the algorithms collected during training. The reward curves correspond to the evaluation results with TD3 achieving the highest overall score, although the difference with DDPG is small. DQN training also seems to be slightly more unstable as the average reward drops significantly at some points. We choose TD3 as the algorithm for the remaining experiments given that it shows the best performance without increasing training time.

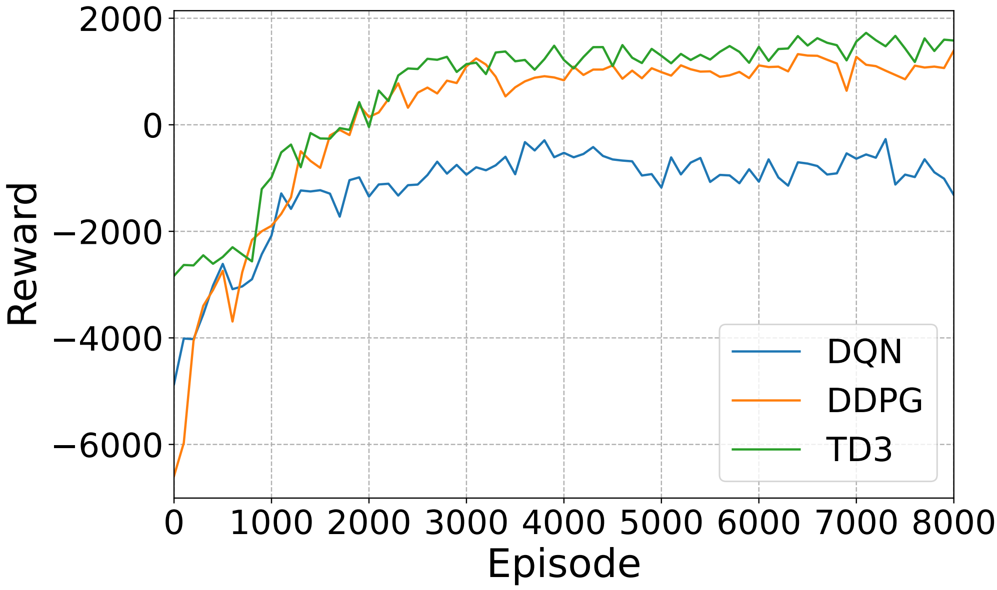


Figure 4.5: The average reward per 100 episodes for different DRL algorithms in stage 4.

4.4. Generalization

Table 4.4: Navigation performance for the best-performing TD3 policy trained in stage 4 and tested in different stages.

Stage	Success	CS	CD	TO	Avg. Dist	Avg. Time	Avg. Speed
4 [a]	100	0	0	0	3.90	19.00	0.93
4	97	0	2	1	4.83	25.07	0.88
2	94	0	5	1	2.95	14.53	0.92
5	94	1	0	5	11.78	55.51	0.96

[a]With no dynamic obstacles

After completing the reward function design and tuning the hyperparameters, the best-performing TD3 policy is evaluated on an unseen scenario with different dimensions and features to validate the generalizability of the model. Table 4.4 shows the performance of the TD3 policy in different scenarios. stage 5 (Figure 4.6) simulates a realistic house environment in an area of 15×10 meters, significantly larger than the training stage with 6×6 meters. In addition, stage 5 features multiple differently shaped static obstacles that resemble common household objects. During the evaluation, the policy achieved a 94% success rate in stage 5 where out of all failed trials 82% were terminated due to timeout and only 18% of the total failures were caused by a collision. The relatively high number of timeouts can be explained by the fact that stage 5 spans a larger area and contains longer contiguous obstacles, occasionally causing the robot to get trapped in a loop continuously traversing the same area. As the robot has no memory capability it does not recognize the repeating trajectory causing it to repeat the same circular route until timeout. Also, during training, the range of the LiDAR sensor is limited to a maximum of 3.5 meters which is rarely exceeded in the training stage while in stage 5 distance readings often reach larger values. Therefore, additional work is required to ensure optimal performance in larger environments which is beyond the scope of this thesis.

Figure 4.6: Stage 5 (15×10 m) features larger dimensions than seen during training to verify the generalizability of the agent.

5

Dealing with Dynamic Obstacles

In the previous section, we showed that the trained policy can reach a success rate of up to 97% in stage 3. According to the evaluation results, the majority of the remaining failures are caused by collisions with dynamic obstacles. In fact, Table 4.4 shows that the TD3 policy can achieve a 100% success rate in stage 2 with all dynamic obstacles removed. Note that static collisions are more likely to occur in scenarios that include dynamic obstacles as they may cause the robot to steer into static obstacles as the agent tries to avoid one. We reason that dynamic obstacles pose a more difficult challenge for obstacle avoidance as they require the agent to have a temporal understanding of the environment to consider the trajectory of dynamic obstacles. Observing the agent in action indeed suggested that the dynamic collisions are likely caused by the fact that the agent has no way to process the relationship between subsequent scan frames. Since the agent only processes a single individual frame of LiDAR distance readings per step it is unable to distinguish between moving and static obstacles and cannot deduce the velocity and direction in which a dynamic obstacle is moving. In addition, interactions with dynamic obstacles generally occur less frequently than interactions with static obstacles meaning the policy is more biased toward dealing with static obstacles. As a result, when moving toward a dynamic obstacle the agent will, similarly to how it avoids the static walls, steer away from the obstacle only at the last moment. While this strategy works for static obstacles, dynamic obstacles require a different strategy as the added velocity gives the agent less time to respond increasing the probability of collision. Therefore, we move from stage 3 to stage 4 (3.2) which adds four additional dynamic obstacles to increase the number of interactions with dynamic obstacles and gear the policy more toward dynamic obstacle avoidance.

In an attempt to reduce the number of dynamic collisions, we investigate different methods and evaluate their effect on dynamic obstacle avoidance and overall performance as discussed below.

5.1. Frame Stacking

Table 5.1: Navigation performance for different DRL stacking algorithms in stage 2.

Training			Evaluation						
Model	**Stack Depth**	**Frame Skip**	**Success**	**CS**	**CD**	**TO**	**Avg. Dist**	**Avg. Time**	**Speed**
TD3 S1	1	0	94	0	5	1	2.95	14.53	0.92s
TD3 S2	3	0	**96**	0	3	1	3.23	16.15	0.91
TD3 S3	3	3	93	1	5	1	2.90	14.40	0.92
TD3 S4	5	0	**96**	0	3	1	3.11	15.58	0.91
TD3 S5	5	5	93	0	6	1	3.04	15.09	0.92
TD3 S6	10	0	95	0	2	3	3.46	17.67	0.89

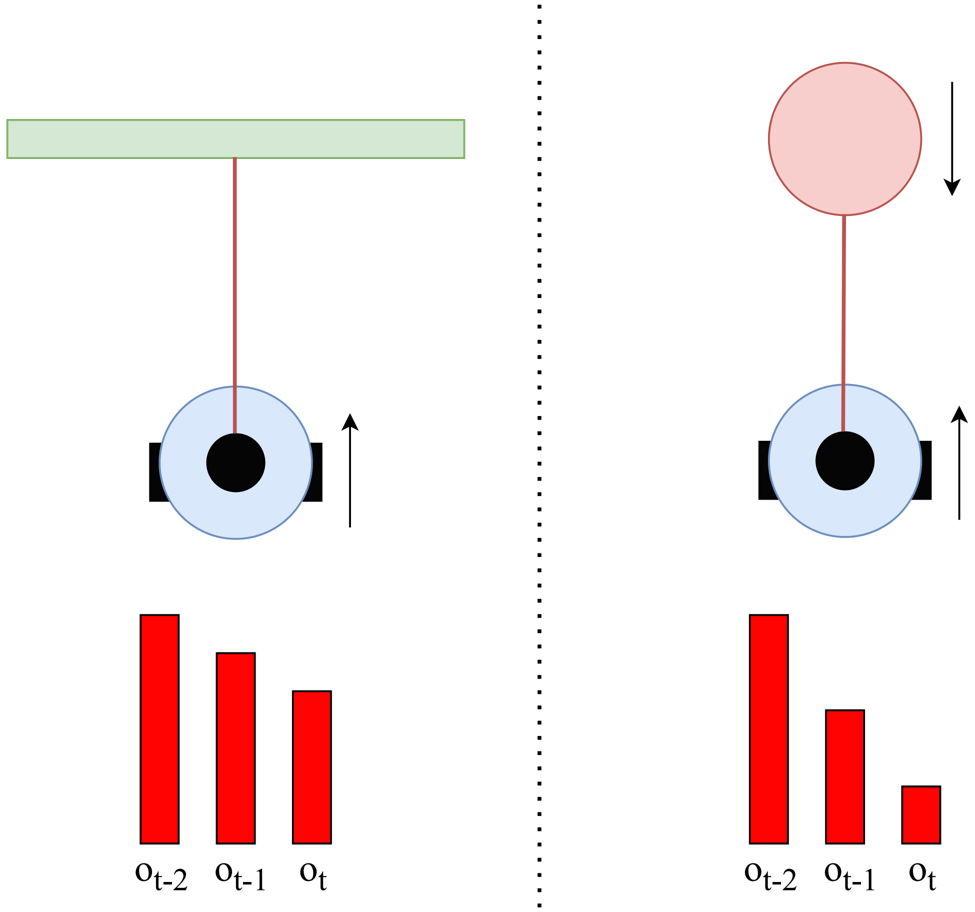


Figure 5.1: Frame stacking enables the agent to distinguish between static obstacles (green) and oncoming dynamic obstacles (red).

With frame stacking the agent processes the last s_d observation sets at every step instead of only the current observation, where s_d is known as the *stack depth*. This is achieved by multiplying the input dimension by the stack depth resulting in a total input size of $O * s_d = O_d$. Essentially, this gives the agent the ability to develop short-term memory and approximate the velocity of visible obstacles. By combining the recent history of velocity commands and scan frames the agent can compare the different values and detect how fast an obstacle is moving and in which direction. This enables the agent to distinguish between static and moving obstacles as the velocity of a moving obstacle influences the distance readings as shown in Figure 5.1. Frame stacking has been used before in DRL navigation systems [14, 15, 21] for UGVs and other applications [29, 42]. However, to the best of our knowledge, the effect of frame stacking on navigation performance and collision avoidance has not been extensively studied before. Figure 5.2 illustrates how frame stacking is implemented and how multiple frames are used as input for the neural network.

5.2. Frame Stepping

The upper range for s_d is limited by computational capabilities as the input dimension grows proportionally to s_d. However, with small values for s_d the subsequent frames will be very near to each other in time providing little valuable information as the environment has shifted only minimally. A simple way to increase the time range in which the agent can observe each step is to set a larger s_d. How-

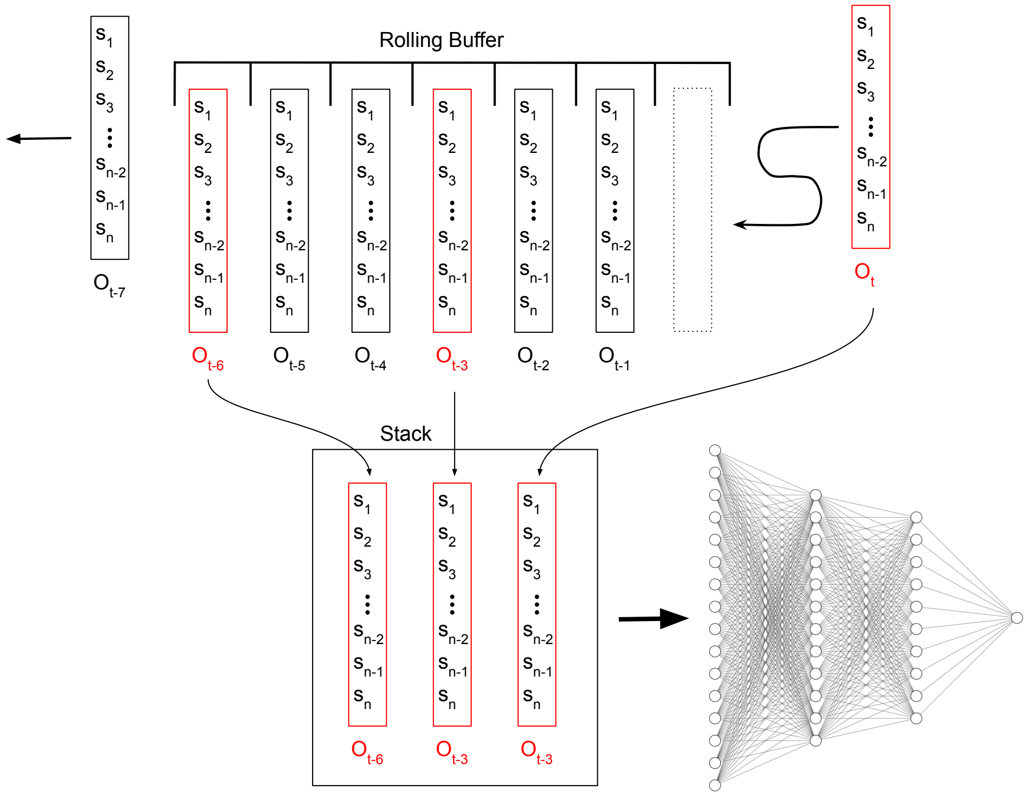


Figure 5.2: Frame skipping and frame stepping depicted with $s_d = 3$ and $s_\epsilon = 3$.

ever, as s_d increases the number of input nodes to the network grows which can quickly increase the complexity of the model making it more difficult to train. Another approach is to insert an artificial delay to ensure enough time has passed in between steps, but this would significantly increase the reaction time of the agent. Frame stepping enables the agent to insert any amount of time in between two subsequent input frames, without heavily affecting the model complexity or responsiveness of the agent. By keeping an active history of a number of the most recent observations, the agent can recall and concatenate any of the samples stored in memory with the most recent observation. Figure 5.2 shows how frame stepping is implemented for our system using a rolling buffer principle. At every timestep t the robot takes s_d observations at a step interval s_ϵ as input, which gives the observation set $O_t, O_{t-(s_\epsilon)*1}, O_{t-(s_\epsilon)*2}, ..., O_{t-(s_\epsilon)*(s_d-1)}$. This is achieved by storing the last $N = s_\epsilon * s_d$ observations in a FIFO buffer of size $B = O_n * s_d * s_\epsilon$ that is updated with the most recent observation at every time step. Effectively, this enables the agent to look back in time for s_d frames at steps of exactly s_ϵ frames at every time step without the need for an artificial delay.

5.3. Simulation Results

To analyze the effect of frame stacking and frame stepping on dynamic obstacle avoidance, we train multiple policies with different frame stacking/stepping configurations in stage 2 which contains only dynamic obstacles to ensure that static obstacles do not influence the result. Table 5.1 shows the evaluation results for the different frame stacking and frame stepping configurations. The results indicate that policies with frame stacking provide a slight benefit over non-stacking policies which is in line with our expectations. The short history of scan observations and velocity commands allows the agent to better anticipate the trajectory of moving obstacles and navigate around them. Contrarily, employing frame stepping on top of frame stacking did not seem to improve dynamic obstacle avoidance. A possible explanation is that the time in between steps is already sufficiently large in which case the additional time in between consecutive input frames delays the reaction speed of the agent. While frame stepping did not seem to provide any benefit for our training setup, it can still improve frame stacking performance on more powerful machines which may leave less processing time in between

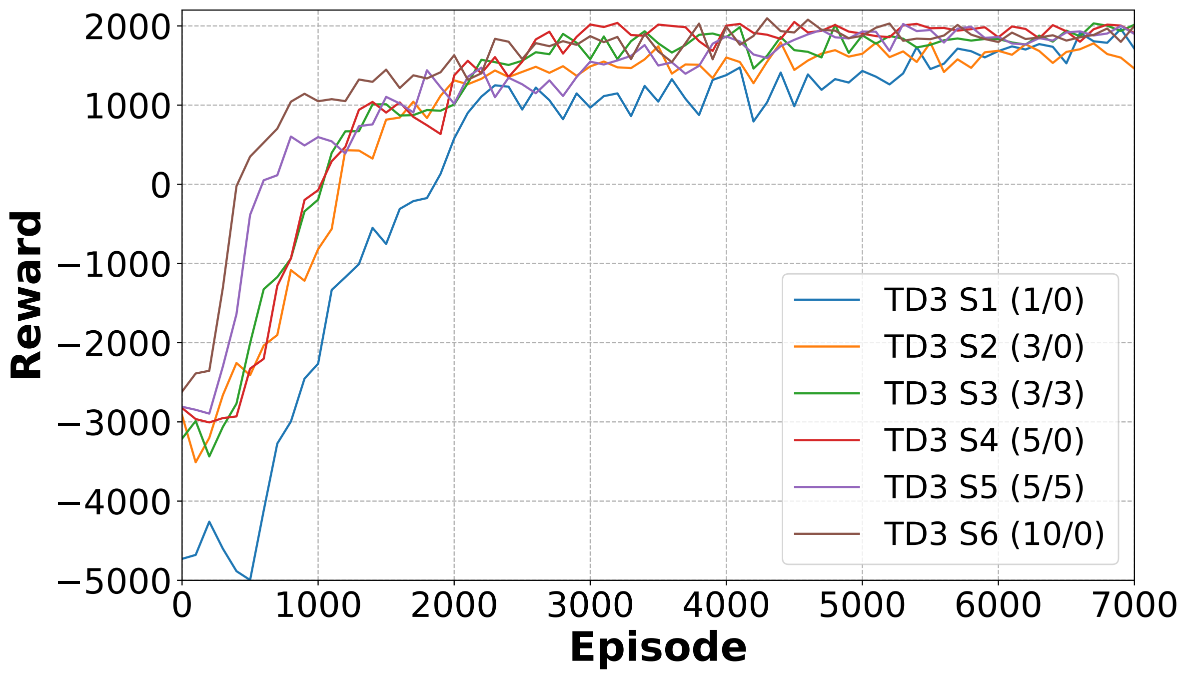


Figure 5.3: The average reward per 100 episodes for different frame stacking/stepping configurations in stage 2.

steps. Looking at the corresponding reward graph in Figure 5.3 we see that the non-stacking model learns slower and scores lower than most other models, but eventually reaches a similar reward value as TD3 S2. The difference in average reward between the different stacking policies is relatively small, with TD3 S4 performing slightly better than the other policies. This is also what we see in the evaluation results where in terms of the average distance and time per trial TD3 S4 navigates more efficiently compared to TD3 S2 with a similar success rate. In both cases, TD3 S2 and TD3 S4 performed better than their counterparts with frame stepping. Lastly, increasing the stack depth beyond five frames for TD3 S6 resulted in slightly worse performance, both in terms of success rate and path efficiency. TD3 S6 also suffered more from timeout failures which can likely be explained by the larger number of frames being processed as input making it harder for the agent to extract the correct information. A larger stack depth results in each frame being retained in memory for a longer period, including frames in which obstacles were near. This may cause the agent to take a more conservative and less optimal route as the presence of obstacles influences the network input over a larger number of steps.

5.4. Backward Motion

In the majority of previous works on DRL UGV navigation, the agent is restricted to moving in the forward direction only and unable to move backward [4]. However, the ability to move backward can significantly improve the ability of the robot to avoid obstacles, especially dynamic obstacles. The main argument for omitting backward motion is that it simplifies the model due to two factors: 1. The LiDAR scan only needs to cover the front half portion of the robot. 2. The agent does not need to learn how to effectively employ action commands for backward motion. However, we argue that backward motion can provide a real benefit to navigation performance, especially in situations where the robot needs to react quickly as depicted in Figure 7.3 where an obstacle suddenly approaches from around the corner and the robot may not have sufficient time to maneuver it. In such a scenario the only option is to quickly move backward until the collision can be averted. Table 5.2 highlights the difference in performance for the same configuration except with backward motion both disabled and enabled once. While both policies are able to achieve a near 100% success rate, the policy with backward motion enabled performed slightly better and did not suffer from a single collision. Backward motion enables the agent to avoid a collision in almost every situation although timeouts can still occur. The increase in success rate does come at a cost as the backward-enabled (BE) agent travels longer on average.

Figure 5.4: The average reward per 100 episodes showing the effect of backward motion in stage 4.

The increase in distance traveled and episode duration are not surprising as the agent generally moves backward to avoid an obstacle and deviates from the path to the goal. Afterward, this deviation needs to be corrected resulting in a longer path whereas other agents would have simply crashed. Looking at the reward graph in Figure 5.4 it appears that the BE policy learns faster than its counterpart at the beginning of training. This could be explained by the fact that the backward-disabled (BD) policy has to first learn how to get around obstacles without moving backward which is a more complex behavior. Although the BE policy gives better results during evaluation, both policies eventually seem to converge to a similar reward value. The total reward per episode is based on multiple factors and decreases in value as time progresses. The higher success rate is likely compensated by the increased distance and time required for the BE, resulting in a similar reward curve. However, since the success rate and collision count are the most important metrics for this study the BE agent is the preferred agent. In addition, Using backward motion the agent is able to learn more complex maneuvers. Since the reward component for linear motion remains unchanged (equation 6.5) the robot is discouraged from moving in the backward direction unless necessary to avoid a collision. This gives rise to new behaviors such as a turning maneuver similar to a three-point turn commonly seen in real-world driving to reverse the heading direction. One of the challenges of training the agent with backward motion enabled is that during the observation stage taking purely random actions will result in the agent oscillating around its starting position as it alternates forward and backward movement commands resulting in low-quality replay buffer samples. Therefore, we bias the random linear actions toward forward movement to generate higher-quality samples as the robot interacts with a larger part of the environment. The effect of backward motion will also be demonstrated on the physical robot in the corresponding section.

Finally, we compare the simulation results of our best-performing agent with different mapless indoor navigation algorithms that were evaluated in a similar setting. Figure 5.4 shows the success rates for each of these algorithms. The baseline TD3 is implemented by Gao et al. [20] as a local planner for their long-range navigation system. GRainbnow [43] combines Genetic Algorithms with DRL to reduce the sensitivity to hyperparameter tuning. The same authors have also evaluated the standard Rainbow method [44] which achieved the lowest score out of all algorithms. Corsi et al. [23] tested their SBP-based DRL algorithm and compared it with a standard PPO implementation. All of these algorithms were evaluated over 100 episodes using the Turtlebot3 simulation platform. Our implementation with backward motion achieves the highest success rate out of the tested algorithms. Although the evaluated environments are not completely identical, our environment includes both dense static

Table 5.2: The effect of backward motion on navigation performance in stage 4.

Training						Evaluation						
Model	**BS**	**RB**	**Discount**	**LR**	**Tau**	**Success**	**CS**	**CD**	**TO**	**Avg. Dist**	**Avg. Time**	**Speed**
TD3 BD [a]	1024	1e+6	0.99	3e-4	3e-4	97	0	2	1	**4.83**	**25.07**	0.88
TD3 BE [b]	1024	1e+6	0.99	3e-4	3e-4	**99**	0	0	1	5.57	27.44	**0.92**

[a] Backward Disabled
[b] Backward Enabled

and dynamic obstacles and is generally more challenging than the environments in the other works presented in Figure 5.4. Furthermore, our system shows consistent performance in a variety of environments with different characteristics. Therefore, it is reasonable to assume that the performance of our algorithm will not decrease significantly in the slightly different environments tested by the other papers.

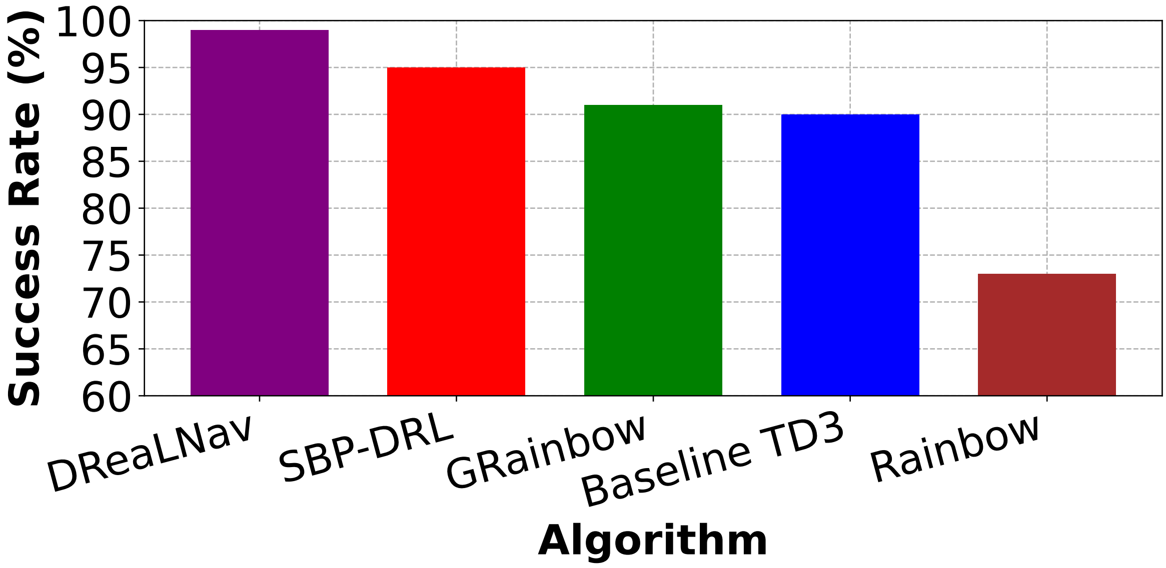


Figure 5.5: The success rate of different mapless navigation algorithms in simulation from the following papers: Rainbow [44], Baseline TD3 [20], GRainbow [43], SBP-DRL [23].

6

Reward Design and Behavior Shaping

The following section will describe a set of reward components and their corresponding variants which can be combined into a composite reward function. The goal for each of the reward components is to facilitate the training process or to restrain certain undesired behavior. The resulting composite reward functions are evaluated and compared in simulation. The goal is to find an efficient reward function that learns a satisfactory policy within a reasonable amount of training time.

6.1. Reward Components

A well-designed reward function is essential for achieving good navigation performance within a feasible training time [45]. In an environment where rewards are sparse, extra steps need to be taken to accelerate the learning process of the agent, especially at the start of training [46]. Through reward shaping [47] the agent is given incremental rewards with every step guiding it toward the final goal. Although reward shaping usually requires hand-crafted solutions based on expert knowledge, it is still widely used in recent works as even simple rules can significantly boost performance.

The autonomous navigation problem generally suffers from the sparse reward problem as the only outcomes associated with a true reward are reaching the goal or ending in a collision with many steps in between. The majority of works on autonomous navigation discussed in the related work make use of a composite reward function consisting of several combined reward components [4] in order to guide the agent toward the goal. By combining different reward components and adjusting the corresponding scaling factors different types of behavior can be elicited from the agent. For example, penalizing the agent for repeatedly adjusting its steering direction can reduce the amount of swaying and smoothen the trajectories. In this section, we analyze the different reward components often used for 2d LiDAR autonomous navigation and their effect on the training time and performance of the agent.

Distance The most common auxiliary reward component is based on the distance from the agent to the goal. d_t represents the distance from the agent to the goal at time step t. One version of this component is defined by the difference in distance to the goal between two consecutive time steps, rewarding the agent for moving closer to the goal, as seen in the works by Tai et al. [9] and Long et al. [14]. The difference is usually multiplied by a scaling factor c_d to keep the overall reward balanced and account for differences in step frequency between machines. This results in the following formula:

$$r_{d_1} = c_d * (d_t - d_{t-1}) \tag{6.1}$$

Note that here the bounds of $r_{distance}$ are defined by the maximum velocity of the robot and c_d. This can make it difficult to select the optimal value for c_d to find suitable reward limits.

For this reason, we propose a normalized version that takes into account the maximum amount of distance the agent could have covered given the time difference $t_i - t_{i-1}$. Let $v_{l_{max}}$ denote the maximum linear speed of the robot:

$$r_{d_2} = c_d * \frac{d_t - d_{t-1}}{v_{l_{max}} * (t_i - t_{i-1})} \tag{6.2}$$

Using this approach the fraction part of the formula is bound to the range $[-1, 1]$, effectively limiting the range of the entire component to $[-c_d, c_d]$ which facilitates clear bound selection in proportion to other components.

Other approaches consider only the current distance and initial distance to the goal d_t without taking into account each previous time step:

$$r_{d_3} = c_d * \frac{2 * d_0}{d_0 + d_t} \tag{6.3}$$

Effectively, this creates an attraction field in which the agent is rewarded for being in closer proximity to the goal rather than being rewarded for actively moving toward the goal. While this approach simplifies defining bounds for the component it has the downside of rewarding the agent for circling close around the goal rather than terminating the episode. Therefore, it is best to ensure that the total reward outcome per step can be at most 0 to avoid positive reward stacking.

Heading The heading component is not strictly required for a good function model as it partly overlaps with the distance component, but it can accelerate the process, especially at the start of training. Given the simple relation between the angle to the goal and the output of the reward, heading toward the goal is often the first thing the agent learns.

$$r_\alpha = -\alpha_g \tag{6.4}$$

Forward Velocity To encourage the robot to move forward, especially early on during the training, a reward can be applied based on the linear velocity. By taking the square of the difference between current linear velocity v_l and maximum possible linear velocity $v_{l_{max}}$ the system penalizes slower velocities exponentially. c_l is the scaling constant for the component which can be adjusted to vary the weight of the penalty.

$$r_{v_l} = -c_l * (v_{l_{max}} - v_l)^2 \tag{6.5}$$

Steering Velocity Long et al. [14] and Choi et al. [10] give a penalty for larger angular velocities to encourage the agent to follow a smooth trajectory. During training, it was observed that without this component the agent might exhibit so-called 'swaying' or 'spinning' behavior. With swaying the agent moves forward while constantly alternating between high negative and positive angular speeds causing it to sway. Large angular velocities may cause the agent to sway too much into the opposite angular direction resulting in overcompensation. While this behavior does not make it impossible for the agent to reach acceptable performance in simulation, it is especially problematic when moving to the real world where physical constraints, mechanical wear, and limited battery capacity play a role. Therefore, it is desirable to have a stable navigation system that moves efficiently along smooth trails with minimal excessive turning. Another hazard of omitting the angular velocity penalty is the occurrence of 'spinning' behavior in which the agent continuously spins in a single angular direction either in place or with minimal linear velocity making little to no progress toward the goal. With an improper training configuration, the agent can be stuck in this detrimental cycle for a long period without making progress toward an optimal policy. One method is to assign an exponentially increasing penalty to larger angular velocities to encourage the agent to turn as little as possible:

$$r_{v_a 1} = -c_a * v_a^2 \tag{6.6}$$

However, the steering penalty may lead to sub-optimal path planning as the agent avoids making large turns resulting in less flexible path planning. Long et al. [14] and Choi et al. [10] take a different approach by applying a steering penalty only at larger velocities to allow the agent to turn moderately but avoid large turns. This prevents the planned routes from becoming too stiff while producing smoother trajectories.

$$r_{v_a 2} = \begin{cases} -c_a * |v_s|, & \text{if } v_s > |\pi/4| \\ 0, & \text{otherwise} \end{cases} \tag{6.7}$$

Obstacle Avoidance Another common practice is the use of safety margins to encourage the robot to proactively avoid obstacles and keep a certain distance from them. The simple approach is to assign a static negative reward when the smallest distance reading d_{min} crosses a safety threshold d_o and the robot enters the 'danger zone' of the obstacle:

$$r_{ob_1} = \begin{cases} -20, & \text{if } d_{min} < d_o \\ 0, & \text{otherwise} \end{cases} \tag{6.8}$$

A more refined approach involves gradual danger zones in which the penalty increases as the robot moves closer to the obstacle after entering the danger zone.

$$r_{ob_2} = \begin{cases} \frac{d_{min} - d_{col}}{d_o - d_{col}} & \text{if d}_{min} < d_o \\ 0, & \text{otherwise} \end{cases} \tag{6.9}$$

Termination Lastly, when the robot reaches a terminating state it receives a reward based on the type of event which terminated the session. If the distance to the goal d_t is smaller than the required minimum distance to the goal d_{goal} the robot receives a large positive reward. Contrarily, if the smallest detected distance value d_{min} is less than the minimum allowed distance to any obstacle before collision $d_{collision}$ the robot receives a large negative reward.

$$r_{termination} = \begin{cases} 2500, & \text{if } d_t < d_{goal} \\ -2000, & \text{if } d_{min} < d_{collision} \\ 0, & \text{otherwise} \end{cases} \tag{6.10}$$

6.2. Reward Functions

To evaluate the effectiveness of different reward components we train multiple models using different composite reward functions. All of the DDPG models are trained for approximately 40 hours after which the best-performing iteration is evaluated over 100 trials. The type of outcome for each episode is recorded in Table 6.1 together with the average distance covered and episode duration over all the successful episodes. The sway index represents the variance in steering and is calculated by taking the squared sum of the difference in two consecutive angular actions. A higher sway index indicates that the robot exhibits undesirable 'swaying' behavior which makes the system less stable.

First, reward functions from various LiDAR-based DRL navigation papers are reimplemented and evaluated in our environment. Next, the reward functions are modified incrementally based on results and our hypotheses to improve performance and reduce training times. The first and most straightforward reward function R_A considers only the difference in distance to the goal between consecutive time steps (equation 6.1) as seen in the works of Tai et al. [9] and Kato et al. [19].

Next, reward function R_B is derived from the study by Long et al. [14] and includes a penalty for angular velocities. As discussed before, this is necessary in order to limit 'sway' behavior. From the results, we see that models trained without the angular velocity component have a significantly higher sway index.

After observing the robot in action it became clear that the robot had too little incentive for moving forward. The robot often came to a standstill when approaching walls to avoid the collision penalty. This often resulted in a timeout and slowed down the training process to such an extent that it did not learn to maneuver around obstacles. Therefore, in R_D the linear velocity component is introduced to encourage the robot to move forward. This eliminates the number of timeouts but results in a much higher number of collisions as the robot does not continue to move forward even when close to obstacles.

Consequently, a non-terminating penalty is introduced which penalizes the robot for moving close to any obstacles. This motivates the robot to turn away from the obstacle at an earlier time step giving it more time to explore alternative routes and avoid a collision.

Combining all of the obtained insights results in the best-performing reward function R_G with a 95% success rate. While it is hard to exactly define the degree to which each component contributes to the success rate, they all play a role as leaving any component out leads to worse performance.

In addition, we test the effect of exchanging certain components with their alternative versions for reward functions R_H, R_I, and R_J. However, none of these functions seem to give a significant improvement for any of the evaluation metrics.

Figure 6.1 shows the number of successful episodes over the first 3500 episodes. This graph indicates how fast each model learns the desired behavior which translates into the efficiency of the underlying reward function. We see that R_G, R_H, and R_J show a similar success rate where small discrepancies can be explained by small differences in starting conditions. The success rates remain fairly constant after the first 3500 episodes showing little further improvement. Note that in theory, all reward functions could eventually converge to a similar performance level given an infinite amount of time. However, since part of our goal is to optimize the training duration we limit the amount of training time. The evaluation of different reward functions concludes this chapter with R_G being the reward function of choice for our experiments.

Table 6.1: Evaluation of different reward functions with varying components in stage 3.

Function	Components	Success	CS	CD	TO	Avg. Dist	Avg. Time	Sway index
R$_A$	$r_{d_1} - 1$	35	53	12	0	3.06	9.15	0.031
R$_B$	$r_{d_1} + r_{v_l} + r_{v_a 1} - 1$	35	60	5	0	4.48	14.49	0.011
R$_C$	$r_{d_1} + r_{\alpha_1} + r_{v_l} - 1$	33	57	21	0	3.05	11.19	0.045
R$_D$	$r_{d_1} + r_{v_a 1} + r_{ob_1} - 1$	22	13	6	59	3.50	9.93	0.012
R$_E$	$r_{d_1} + r_{\alpha_1} + r_{v_l} + r_{v_a 1} - 1$	32	57	11	0	2.75	9.75	0.011
R$_F$	$r_{d_1} + r_{\alpha_1} + r_{v_a 1} + r_{ob_1} - 1$	22	1	29	48	2.19	19.50	0.005
R$_G$	$r_{d_1} + r_{v_l} + r_{v_a 1} + r_{ob_1} - 1$	44	6	19	31	5.28	18.51	**0.003**
R$_H$	$r_{d_1} + r_{\alpha_1} + r_{v_l} + r_{v_a 1} + r_{ob_1} - 1$	**93**	2	3	0	4.30	23.01	0.012
R$_I$	$r_{d_2} + r_{\alpha_1} + r_{v_l} + r_{v_a 1} + r_{ob_1} - 1$	93	1	6	0	4.18	22.08	0.013
R$_J$	$r_{d_1} + r_{\alpha_1} + r_{v_l} + r_{v_a 1} + r_{ob_2} - 1$	93	1	6	0	3.99	18.12	0.005
R$_K$	$r_{d_2} + r_{\alpha_1} + r_{v_l} + r_{v_a 1} + r_{ob_2} - 1$	90	0	10	0	3.77	20.04	0.013

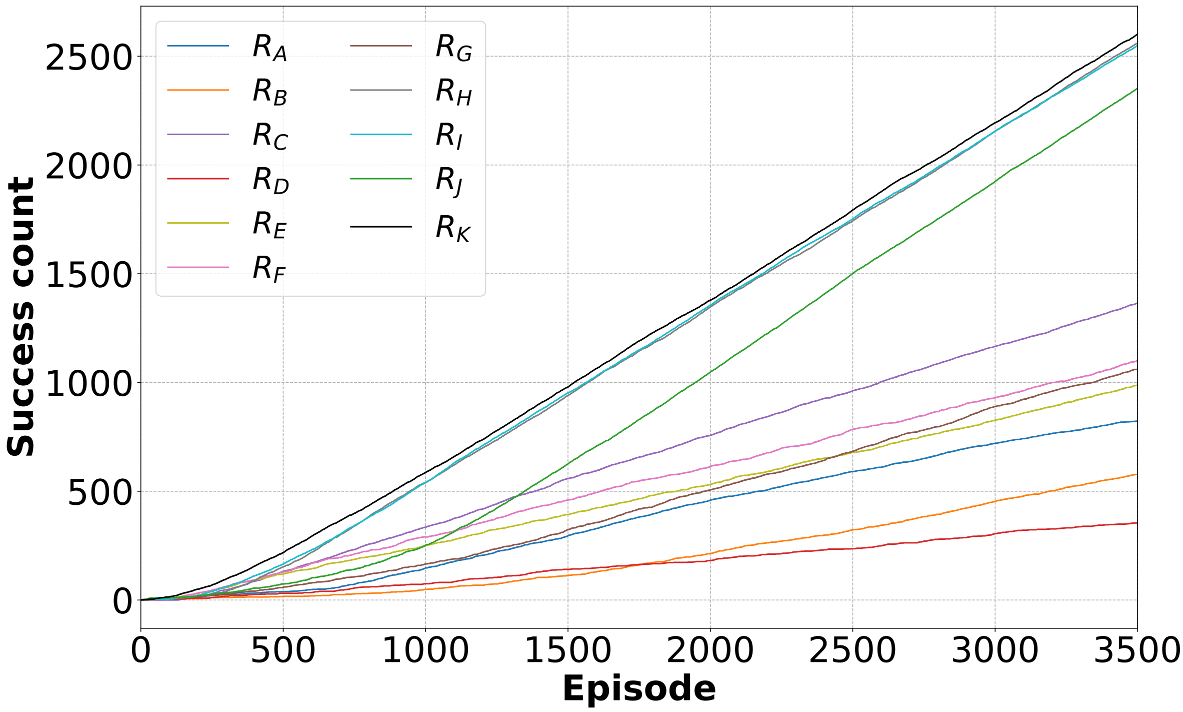


Figure 6.1: The success rates over time during training for different reward functions.

7 Physical System Validation

In order to validate the developed platform we demonstrate and evaluate the trained policies in multiple real-world scenarios.

7.1. Real-World Evaluation

One of the challenges of transferring the navigation policy from simulation to the real world is the tuning required to accurately map model output actions to physical motor control commands. In simulation, the resulting forward velocity of the robot always scales linearly with the action output at any value. However, for the physical robot factors such as inertia, rolling resistance, and imperfect motor and tachometer hardware cause the real-world motion to deviate from the behavior in simulation. To account for the inertia and rolling resistance of the wheels a tuned base speed value is added to the motor PWM signal to prevent the robot from remaining stationary when the model outputs lower velocity commands. Furthermore, the maximum values for linear and angular velocity need to be tuned to the appropriate values for the PWM signal to the motors. Lastly, a low-level PID controller is implemented on the Arduino board using tachometer input to provide fast feedback to the motor actuators which requires tuning according to the specifications of the robot model. A video demonstration of the progress and effect of the tuning process is made available online 7.2.

Table 7.1: Evaluation on the physical robot over the same trajectory for 20 trials in real-world stage I.

Algorithm	Success	CS	Timeout	Avg. Dist (m)	Avg. Time (s)	Max speed ratio
TD3	18	2	0	6.83	22.15	0.88

To demonstrate the transferability of the trained policy to the real world a physical experiment is conducted in which the robot is repeatedly tasked with navigating from the starting location to the goal in the environment shown in Figure 7.1. The experiment consists of 20 trials during which the outcome, distance traveled and duration are recorded. In addition, we also measure the portion of the total trial during which the robot is moving at maximum linear velocity. Table 7.1 shows that the robot is able to achieve a 90% success rate during the performed experiment. The shortest path for the current scenario can be measured to be approximately 5 meters long while the robot traveled 6.83 meters on average during the experiment. The additional distance can be explained by the fact that the robot maintains a safety margin to any obstacles as well as inaccuracies in the distance measurements, odometry, and motor output causing deviations from the desired trajectory which are corrected by the robot, all resulting in longer paths. Despite being trained in an environment with different obstacles and dimensions the robot is still able to navigate reliably in the unseen scenario.

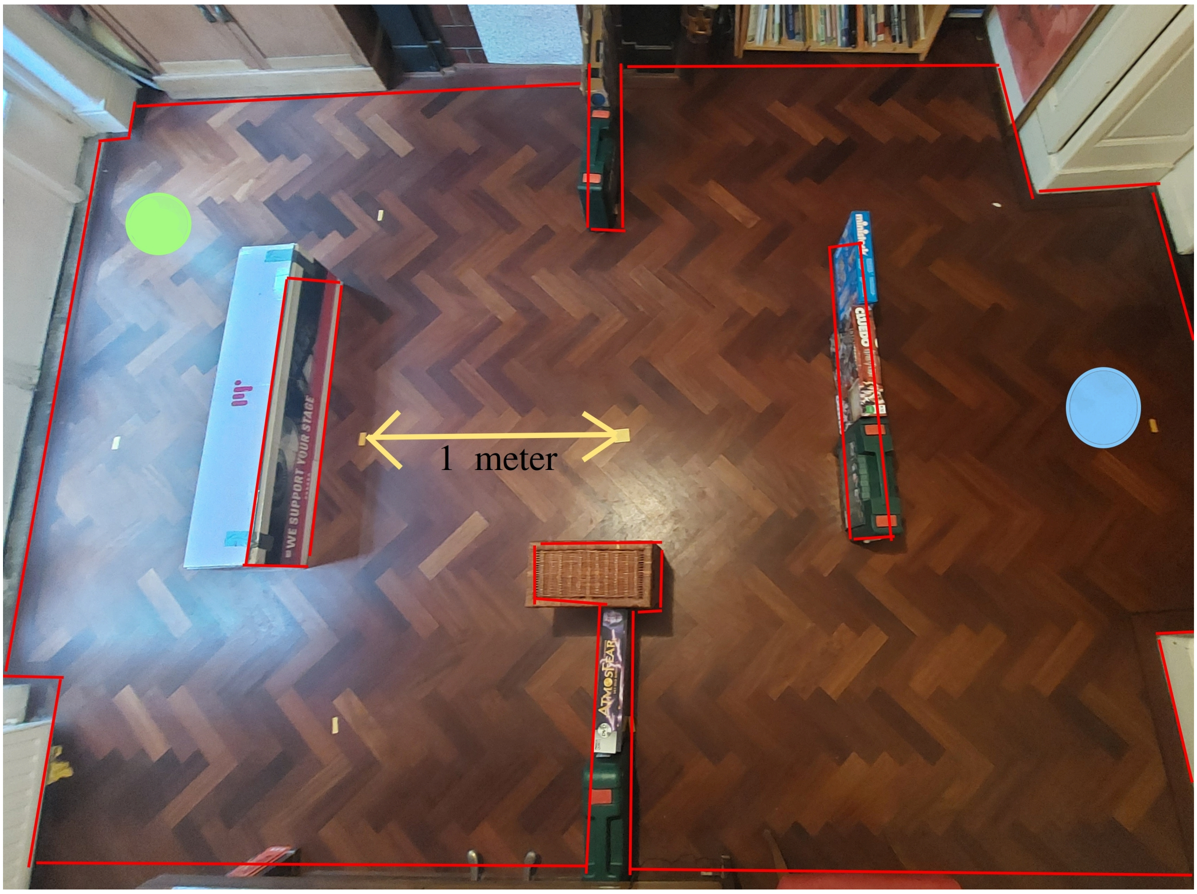


Figure 7.1: Stage I used for the real-world experiment presented in Table 7.1. The starting position and goal are indicated by the blue and green circles.

Figure 7.2: QR code to video to tuning process
(https://www.youtube.com/watch?v=WJtpsJvSsEg).

7.2. Backward Motion

To demonstrate the viability and benefit of backward motion in the real world we reproduce a realistic situation in which the agent has to rely on backward motion in order to avoid a collision as shown in Figure 7.3. The objective of the robot in this scenario is to move forward toward its goal position while avoiding any obstacles. The top left image shows the starting situation where both the robot and dynamic obstacle are approaching the same corner without a line of sight between them. The top right image shows the moment shortly after the robot detects the obstacle and starts to decelerate in response to the approaching obstacle. The obstacle is turning toward the direction of the robot giving the robot too little time to maneuver around the obstacle. The bottom left image features a robot with backward motion. In this case, the robot is able to quickly react and keep a safe distance from the obstacle even as it moves toward the robot. After the path has been cleared again the robot continues to move forward toward its destination. On the contrary, the bottom right image shows a robot without backward motion attempting to steer away from the robot which results in a collision. Note that in this situation the robot could not simply avoid a collision by remaining stationary as the obstacle is moving in the direction of the robot. Moreover, due to inertia, the robot will still move forward slightly immediately after giving the stop command and will come to a halt closer to the obstacle as compared to in simulation. A video demonstration of the depicted situation is made available online [1].

In addition, a further demonstration of how backward motion can help the robot respond quickly in tricky situations is made available online (7.5).

[1] https://youtu.be/-eyNCvBohRk

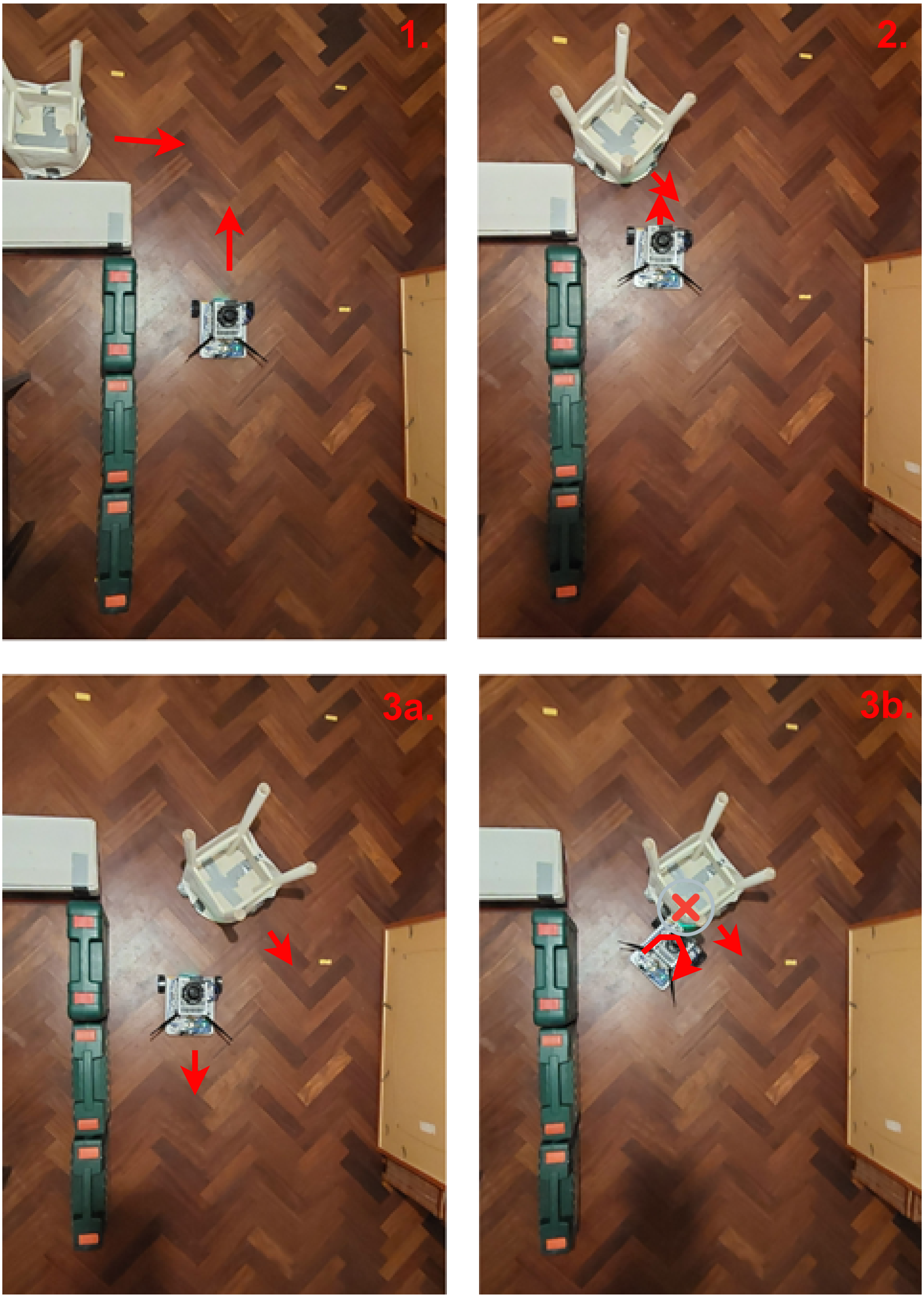


Figure 7.3: Backward motion enables the robot to avoid a collision when a dynamic obstacle suddenly appears. **TL (1):** The robot approaches the corner with no vision of the obstacle. **TR (2):** The robot detects the obstacle moving toward the robot. **BL (3a):** the BE policy evades the oncoming obstacle by moving backward. **BR (3b):** The BD policy attempts to turn away and fails to avoid a collision. Video: https://youtu.be/-eyNCvBohRk).

Figure 7.4: QR code to a video comparing the obstacle avoidance capability of a robot with backward motion and a robot with only forward motion.
(https://pyoutu.be/-eyNCvBohRk)

Figure 7.5: QR code to a video demonstrating the capabilities of backward motion
(https://youtu.be/rEkPuWZlLK0)

8 Conclusions and Limitations

8.1. Conclusions

For this thesis, a complete DRL navigation stack, including both software and hardware, has been developed and demonstrated to be able to train and reliably transfer policies trained in simulation to a physical robot to navigate previously unseen environments. The robot is able to handle both navigation and obstacle avoidance by itself and operates independently without the need for a central entity. The performance of the trained policies is validated through experiments both in simulation and in a real-world environment. Furthermore, video demonstrations showcase the capability of the system when dealing with fast-moving and unexpected obstacles. After evaluating different off-policy DRL algorithms with different hyperparameter settings, LiDAR configurations, and reward functions, the resulting agent is able to reach a 99% success rate in simulation in a challenging environment with uncooperative dynamic entities. We have also evaluated the concept of frame stacking and frame stepping and its effect on dynamic obstacle avoidance in simulation. In the real world, the robot can navigate independently in an unseen environment without extensive tuning. The robot is also capable of reacting to fast-moving obstacles and human agents despite being trained on only a single type of moving obstacle. The amount of swaying is reduced through the reward function design to smoothen the trajectories of the robot. We have shown that backward motion can improve the performance of the robot and gives it the ability to handle tricky real-world situations which would otherwise result in a collision. The $360°$ LiDAR range allows the robot to make full use of backward motion and avoid obstacles from any direction. Therefore, we conclude that the previously unquestioned conventions which largely disregarded the above-mentioned factors are worth reconsidering as this can significantly improve the training process and navigation performance.

8.2. Limitations

While the resulting system can successfully perform autonomous navigation in a real-world environment, several identifiable factors limit the performance of the current setup. Firstly, the robot hardware is a prototype that suffers from several defects that require modification in order to improve reliability. Currently, the robot is equipped with only two drive wheels in the front and a third caster wheel in the back which limits maneuverability as the robot is not able to smoothly turn in place. A better hardware design with higher quality joints or possibly a four-wheel drive system would improve the steering ability and overall performance of the robot. Also, the LiDAR sensor is placed relatively high, hindering the ability of the robot to detect obstacles closer to the ground. This could be amended by modifying the mechanical design such that the LiDAR can be placed closer to the ground just above the wheels.
Next, we have chosen to avoid convolutional and LSTM networks as they are not required for the system to navigate successfully and to limit system complexity. This is not to say that there is no value to be gained from these techniques and they pose a promising future direction for further improvement of the obstacle avoidance capabilities of the robot. Furthermore, while we believe that a LiDAR sensor is essential for robust real-world navigation, a camera could still be a valuable addition to the system for applications that require context-sensitive information. Whereas the LiDAR provides reliable $360°$ distance readings, an inexpensive RGB camera can provide the agent with additional context and identify

different types of obstacles to improve dynamic obstacle avoidance. This would however once again result in a more complex system requiring additional effort to retain transferability.

Bibliography

[1] H. Durrant-Whyte and T. Bailey, “Simultaneous localization and mapping: part i,” *IEEE Robotics Automation Magazine*, vol. 13, no. 2, pp. 99–110, 2006.

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

[3] S. Zaman, W. Slany, and G. Steinbauer, “Ros-based mapping, localization and autonomous navigation using a pioneer 3-dx robot and their relevant issues,” in *2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC)*, 2011, pp. 1–5.

[4] K. Zhu and T. Zhang, “Deep reinforcement learning based mobile robot navigation: A review,” *Tsinghua Science and Technology*, vol. 26, no. 5, pp. 674–691, 2021.

[5] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074

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

[7] M. Pfeiffer, M. Schaeuble, J. I. Nieto, R. Siegwart, and C. Cadena, “From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots,” *CoRR*, vol. abs/1609.07910, 2016. [Online]. Available: http://arxiv.org/abs/1609.07910

[8] M. Duguleana and G. Mogan, “Neural networks based reinforcement learning for mobile robots obstacle avoidance,” *Expert Systems with Applications*, vol. 62, pp. 104–115, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417416303001

[9] L. Tai, G. Paolo, and M. Liu, “Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 31–36.

[10] J. Choi, K. Park, M. Kim, and S. Seok, “Deep reinforcement learning of navigation in a complex and crowded environment with a limited field of view,” in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 5993–6000.

[11] H. Surmann, C. Jestel, R. Marchel, F. Musberg, H. Elhadj, and M. Ardani, “Deep reinforcement learning for real autonomous mobile robot navigation in indoor environments,” *CoRR*, vol. abs/2005.13857, 2020. [Online]. Available: https://arxiv.org/abs/2005.13857

[12] V. Nguyen, T. Manh, C. Manh, D. Tien, M. Van, D. Ha Thi Kim Duyen, and D. Nguyen Duc, “Autonomous navigation for omnidirectional robot based on deep reinforcement learning,” *International Journal of Mechanical Engineering and Robotics Research*, pp. 1134–1139, 01 2020.

[13] K. Weerakoon, A. J. Sathyamoorthy, U. Patel, and D. Manocha, “TERP: reliable planning in uneven outdoor environments using deep reinforcement learning,” *CoRR*, vol. abs/2109.05120, 2021. [Online]. Available: https://arxiv.org/abs/2109.05120

[14] P. Long, T. Fan, X. Liao, W. Liu, H. Zhang, and J. Pan, “Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning,” *CoRR*, vol. abs/1709.10082, 2017. [Online]. Available: http://arxiv.org/abs/1709.10082

[15] T. Fan, X. Cheng, J. Pan, D. Manocha, and R. Yang, “Crowdmove: Autonomous mapless navigation in crowded scenarios,” *CoRR*, vol. abs/1807.07870, 2018. [Online]. Available: http://arxiv.org/abs/1807.07870

[16] M. Everett, Y. F. Chen, and J. P. How, “Motion planning among dynamic, decision-making agents with deep reinforcement learning,” *CoRR*, vol. abs/1805.01956, 2018. [Online]. Available: http://arxiv.org/abs/1805.01956

[17] S. Liu, P. Chang, W. Liang, N. Chakraborty, and K. D. Campbell, “Decentralized structural-rnn for robot crowd navigation with deep reinforcement learning,” *CoRR*, vol. abs/2011.04820, 2020. [Online]. Available: https://arxiv.org/abs/2011.04820

[18] J. Wang, S. Elfwing, and E. Uchibe, “Modular deep reinforcement learning from reward and punishment for robot navigation,” *Neural Networks*, vol. 135, pp. 115–126, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608020304184

[19] Y. Kato, K. Kamiyama, and K. Morioka, “Autonomous robot navigation system with learning based on deep q-network and topological maps,” in *2017 IEEE/SICE International Symposium on System Integration (SII)*, 2017, pp. 1040–1046.

[20] J. Gao, W. Ye, J. Guo, and Z. Li, “Deep reinforcement learning for indoor mobile robot path planning,” *Sensors*, vol. 20, no. 19, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/19/5493

[21] L. Gao, J. Ding, W. Liu, H. Piao, Y. Wang, X. Yang, and B. Yin, “A vision-based irregular obstacle avoidance framework via deep reinforcement learning,” *CoRR*, vol. abs/2108.06887, 2021. [Online]. Available: https://arxiv.org/abs/2108.06887

[22] M. M. Ejaz, T. B. Tang, and 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, 2021.

[23] D. Corsi, R. Yerushalmi, G. Amir, A. Farinelli, D. Harel, and G. Katz, “Constrained reinforcement learning for robotics via scenario-based programming,” 2022. [Online]. Available: https://arxiv.org/abs/2206.09603

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

[25] J. Achiam, “Spinning up in deep reinforcement learning,” 2018. [Online]. Available: https://spinningup.openai.com/en/latest/index.html

[26] A. Y. Majid, S. Saaybi, T. van Rietbergen, V. François-Lavet, R. V. Prasad, and C. J. M. Verhoeven, “Deep reinforcement learning versus evolution strategies: A comparative survey,” *CoRR*, vol. abs/2110.01411, 2021. [Online]. Available: https://arxiv.org/abs/2110.01411

[27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[28] L.-J. Lin, “Reinforcement learning for robots using neural networks,” Ph.D. dissertation, USA, 1992, uMI Order No. GAX93-22750.

[29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[30] C. Watkins, “Learning from delayed rewards,” 01 1989.

[31] Z. Wang, S. Zhang, X. Feng, and Y. Sui, “Autonomous underwater vehicle path planning based on actor-multi-critic reinforcement learning,” *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, vol. 235, no. 10, pp. 1787–1796, 2021. [Online]. Available: https://doi.org/10.1177/0959651820937085

[32] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” *CoRR*, 09 2015.

[33] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic policy gradient algorithms,” ser. ICML’14. JMLR.org, 2014, p. I–387–I–395.

[34] S. Fujimoto, H. van Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” *CoRR*, vol. abs/1802.09477, 2018. [Online]. Available: http://arxiv.org/abs/1802.09477

[35] W. Zhao, J. P. Queralta, and T. Westerlund, “Sim-to-real transfer in deep reinforcement learning for robotics: a survey,” pp. 737–744, 2020.

[36] G. Dudek and M. Jenkin, *Computational Principles of Mobile Robotics*, 2nd ed. USA: Cambridge University Press, 2010.

[37] X. Dong, J. Shen, W. Wang, Y. Liu, L. Shao, and F. Porikli, “Hyperparameter optimization for tracking with continuous deep q-learning,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 518–527.

[38] L. Yang and A. Shami, “On hyperparameter optimization of machine learning algorithms: Theory and practice,” *Neurocomputing*, vol. 415, pp. 295–316, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231220311693

[39] N. Ashraf, R. Mostafa, R. Sakr, and M. Rashad, “Optimizing hyperparameters of deep reinforcement learning for autonomous driving based on whale optimization algorithm,” *PLOS ONE*, vol. 16, p. e0252754, 06 2021.

[40] Y. Bengio, “Practical recommendations for gradient-based training of deep architectures,” *CoRR*, vol. abs/1206.5533, 2012. [Online]. Available: http://arxiv.org/abs/1206.5533

[41] D. Masters and C. Luschi, “Revisiting small batch training for deep neural networks,” *CoRR*, vol. abs/1804.07612, 2018. [Online]. Available: http://arxiv.org/abs/1804.07612

[42] D. S. Chaplot, “Transfer deep reinforcement learning in 3 d environments : An empirical study,” 2016.

[43] E. Marchesini and A. Farinelli, “Genetic deep reinforcement learning for mapless navigation,” in *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS ’20. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2020, p. 1919–1921.

[44] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, “Rainbow: Combining improvements in deep reinforcement learning,” *CoRR*, vol. abs/1710.02298, 2017. [Online]. Available: http://arxiv.org/abs/1710.02298

[45] Q. Zhang, M. Zhu, L. Zou, M. Li, and Y. Zhang, “Learning reward function with matching network for mapless navigation,” *Sensors*, vol. 20, no. 13, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/13/3664

[46] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. D. Turck, and P. Abbeel, “Curiosity-driven exploration in deep reinforcement learning via bayesian neural networks,” *CoRR*, vol. abs/1605.09674, 2016. [Online]. Available: http://arxiv.org/abs/1605.09674

[47] M. J. Mataric, “Reward functions for accelerated learning,” in *Machine Learning Proceedings 1994*, W. W. Cohen and H. Hirsh, Eds. San Francisco (CA): Morgan Kaufmann, 1994, pp. 181–189. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558603356500301