Computing&AI Connect
2024, Vol. 1, Cite as: doi:10.x/journal.x.x.x

SCIFINITI PUBLISHING
Research Article

Challenging Conventions Towards Reliable Robot Navigation Using Deep Reinforcement Learning

Amjad Yousef Majid,✉[1] **Tomas van Rietbergen**,[2] **R Venkatesha Prasad**,[2]

[1] Martel Innovate;
[2] Delft University of Technology;

Article History
Submitted: May 18, 2024 Accepted: XXXX XX, 2024 Published: XXXX XX, 2024

Abstract
Effective indoor navigation in the presence of dynamic obstacles is crucial for mobile robots. Previous research on deep reinforcement learning (DRL) for robot navigation has primarily focused on expanding neural network (NN) architectures and optimizing hardware setups. However, the impact of other critical factors, such as backward motion enablement, frame stacking buffer size, and the design of the behavioral reward function, on DRL-based navigation remains relatively unexplored. To address this gap, we present a comprehensive analysis of these elements and their effects on the navigation capabilities of a DRL-controlled mobile robot. In our study, we developed a mobile robot platform and a Robot Operating System (ROS) 2-based DRL navigation stack. Through extensive simulations and real-world experiments, we demonstrated the impact of these factors on the navigation of mobile robots. Our findings reveal that our proposed agent achieves state-of-the-art performance in terms of navigation accuracy and efficiency. Notably, we identified the significance of backward motion enablement and a carefully designed behavioral reward function in enhancing the robot's navigation abilities. The insights gained from this research contribute to advancing the field of DRL-based robot navigation by uncovering the influence of crucial elements and providing valuable guidelines for designing robust navigation systems.

Keywords
deep reinforcement learning; robot; navigation.

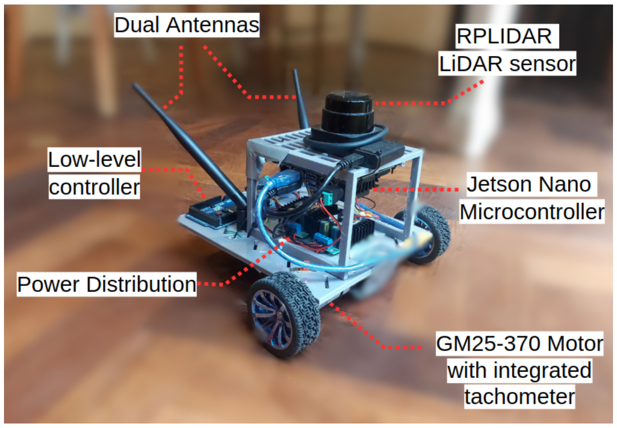
1. Introduction

It could be deduced that in an environment where autonomous robots are becoming integral to various sectors, the development of sophisticated and robust robotic navigation systems is imperative. Traditional motion planners and localization techniques such as Simultaneous Localization and Mapping (SLAM) [11], Dynamic Window Approach (DWA) [14], and Adaptive Monte Carlo Localization (AMCL) [54], rely heavily on predefined feature extraction and prior environmental maps. Despite being effective, these methods have considerable limitations as they require extensive parameter tuning and have difficulty adapting to new environments without further modifications. [33,57]. Additionally, these traditional systems are compartmentalized, with separate modules for tasks such as vision, planning, and control, resulting in suboptimal overall performance due to isolated optimization of each module [58].

Recently, the application of deep learning in autonomous navigation of unmanned ground vehicles (UGVs) has surged [3,20,24,37,49]. Deep learning's ability to translate raw inputs into precise steering commands enables an integrated, end-to-end system, which is proficient in both motion planning and obstacle avoidance. Compared to traditional methods, classical machine learning tech-

* **Corresponding Author:**
A.Y. Majid, Martel Innovate, amjad.majid@martel-innovate.com;
Tel.: +31-616-955-224

Figure 1: An overview of the robot hardware.



niques, such as supervised learning, reduce the need for hand-tuning and adapt better to unfamiliar environments. However, these methods also have limitations since data collection for supervised learning is labor-intensive, and the final system's effectiveness relies heavily on the quality of the dataset. [38,43].

Deep reinforcement learning (DRL) addresses these limitations by providing a framework to train models through direct environmental feedback, eliminating the need for extensive data collection. DRL has gained significant traction in recent years and is widely used in autonomous navigation research for both LiDAR and vision-based systems [42,45,58]. Despite its benefits, DRL encounters challenges due to the costly and potentially hazardous trial-and-error nature of data collection in real-world environments. Therefore, agents are first trained in simulations for efficient learning before being deployed on physical robots. However, this approach introduces a 'reality gap,' as current simulators fail to perfectly replicate the complexities of the physical world, leading to potential discrepancies in agent behavior between simulated and real environments [23].

A prevalent observation in most prior DRL navigation research is the lack of exploration into the effects of various design decisions on mobile UGVs [58]. Many studies adhere to standard practices regarding hyperparameters, sensor configuration, reward design, frame stacking, and direction of motion. In contrast, this work challenges these norms by analyzing the impact of these design elements on the navigational performance of a UGV DRL agent. To develop a reliable, autonomous navigation agent with dynamic obstacle avoidance capabilities, we assess these elements within our novel framework.

Our framework, built on the Robot Operating System 2 (ROS 2) middleware suite [32] and the Gazebo simulator [27], extends standard forward-only motion to a full range of motion and evaluates the benefits of backward motion, particularly for dynamic obstacle evasion. By implementing various frame stacking configurations, we assess their influence on dynamic obstacle avoidance. Additionally, we introduce unique reward components to mitigate undesirable behaviors, such as 'swaying,' thereby enhancing real-world applicability. The efficacy of our system is validated in both novel simulation environments and on a custom-built, physical mobile robot.

With this work, our primary goal is to provide the research community with insightful design guidelines for creating reliable DRL UGV navigation systems. To this end, we make the following contributions:

1. We develop an end-to-end framework for ROS 2 DRL-based UGV navigation, incorporating three off-policy DRL algorithms.
2. We explore the impact of different off-policy algorithms, hyperparameter configurations, and reward functions on UGV navigation.
3. We investigate the utility of frame stacking and backward motion for effective dynamic obstacle avoidance.
4. We validate the system in real-world scenarios, employing a custom-built mobile robot (Figure 1) navigating in challenging environments with fast-moving obstacles.

2. Related Work

In this section, we discuss the work related to the use of DRL for robot navigation and obstacle avoidance.

2.1. Navigation

Significant advancements have been noted in the domain of Deep Reinforcement Learning (DRL)-based navigation. Among the initial research in this area, Duguleana et al. were pioneers, combining Q-learning with a neural network to develop a Deep Q-Network (DQN) motion planner, capable of executing three distinct actions i.e., moving forward, turning left, or turning right [10]. To reduce the state space, the environment was segmented into eight angular regions. This approach successfully managed navigation in simple environments and established a foundation for more complex DRL navigation research.

The following year, Tai et al. introduced a low-cost, mapless DRL-based navigation system for mobile robots, which mapped sparse Light Detection and Ranging (LiDAR) distance readings into continuous actions [47]. Experiments in unseen environments with static obstacles confirmed the model's real-world transferability. However, due to a low laser count and a simplistic training environment, the model's performance was impaired when encountering dynamic obstacles.

Further innovation came from Choi et al., who replaced the traditional 360° LiDAR with a 90° depth camera and a state-of-the-art Long Short-Term Memory (LSTM) network [6]. This combination outperformed agents with a wide Field of View (FOV) but no memory. To bridge the simulation-reality gap, they employed dynamics randomization, adding noise to scan readings, robot velocity, and control frequency. This approach made the model robust against unpredictable real-world dynamics.

Surmann et al. took a different approach, designing a system to simultaneously train multiple DRL agents using a lightweight 2D simulation, each in a unique environment [46]. However, their work did not include dynamic obstacles and was not entirely collision-free. In contrast, our work demonstrated that Gazebo could effectively train an autonomous navigation agent with robust generalization capabilities.

Nguyen Van et al. demonstrated the versatility of DRL-based LiDAR navigation by applying it to different robotic models, such as a four-wheel omnidirectional robot [40]. Their simulation experiments demonstrated the agent's capability to navigate between waypoints while avoiding stationary obstacles.

Recently, Weerakoon et al. developed a navigation robot using 3D LiDAR and elevation maps for reliable trajectory planning in uneven outdoor environments [52]. Their network used a Convolutional Block Attention Module to identify regions with reduced robot stability. Although their method had a high success rate compared to the dynamic window approach, the robot struggled with steep ditches and surface boundaries, requiring additional depth sensors and RGB cameras.

2.2. Obstacle Avoidance

In the rapidly evolving field of autonomous navigation, a variety of methodologies employing deep reinforcement learning (DRL) for obstacle avoidance have emerged. One novel approach by Wang et al. [50] developed Deep Max-Pain, a modular DRL method with separate policies for reward and punishment, inspired by the operational mechanisms of the animal brain. They proposed a state-value dependent weighting scheme based on a Boltzmann distribution to balance the ratio between the two signals for the final joint policy. This approach used both LiDAR scans and RGB-camera images, achieving superior performance compared to DQN in both simulations and real robots.

DRL navigation approaches incorporating global information from high-level planners have been proposed by Jin et al. [25], Kato et al. [26] and Gao et al. [16]. Kato et al. [26] developed a long-range DRL navigation system by pairing a local DDQN agent with a topological map global planner. Gao et al. [16] proposed a similar approach, combining TD3 with Probabilistic Road Maps (PRM) to create an indoor long-range motion planner capable of generalizing to larger, unseen environments. To improve DRL agents from a practical standpoint, offline DRL has been proposed [41,42].

However, in complex environments characterized by continuous and large-scale obstacles, DRL navigation may struggle to escape local optima. In such scenarios, Liu et al. [30] proposed integrating structural RNNs into a PPO-based neural network to handle unpredictable human trajectories in dense crowds. This system, incorporating two separate RNNs for spatial and temporal relations with nearby humans, demonstrated superior performance compared to ORCA, handling a significant number of humans. However, a comparison with other DRL approaches has yet to be conducted.

Simultaneously, Gao et al. [17] focused on irregular obstacle detection by fusing laser scan and RGB camera data. Their method used a novel depth slicing technique to acquire pseudo-laser data encoding both depth and semantic information, maximizing the advantages of both data types.

Efficiency improvements in the training phase for depth camera D3QN-based autonomous navigation were proposed by Ejaz et al. [12]. They employed techniques such as layer normalization and the injection of Gaussian noise into the fully connected layers to reduce computational costs and stimulate exploration, leading to reduced training times. Recent work also explores the use of event cameras to reduce latency [18] and improve night vision [29].

The integration of domain-expert knowledge into the DRL training process for mapless navigation has been demonstrated by Corsi et al. [7]. This approach enhances performance and mitigates undesired behaviors by incorporating scenario-based programming (SBP) constraints into the cost function, allowing explicit constraints to be directly embedded into the policy optimization process.

In contrast to the complex architectures proposed by Hoeller et al. [21], Liu et al. [30] and Gao et al. [17], this work seeks to establish a DRL indoor navigation system that balances reliable performance and simplicity. Similar to Corsi et al. [7], we aim to enforce specific desired behaviors, but instead of using manually designed SBP constraints, we focus on developing an effective reward function to address complex navigation scenarios.

This work also seeks to address the limitations in real-world applications identified by Choi et al. [6], Liu et al. [30], and Gao et al. [17], particularly regarding the disabling of backward motion and its impact on pedestrian movement speed and obstacle configurations. Our robot is equipped with 360° LiDAR scans, enabling continuous monitoring of its full surroundings and anticipating obstacles from any direction, including those outside the robot's field of vision or approaching from behind. Through real-world demonstrations, we aim to show that this comprehensive sensing approach allows the robot to navigate safely, even in the presence of fast-moving or unpredictably moving obstacles.

3. System Overview

Here we provide an overview of the theory behind the system and describe the various tools and methods used for the experiments[1].

3.1. Local Navigation

Our goal is to develop a reliable, mapless, and decentralized motion planner for mobile robots. The robot must navigate from the starting location to the goal while maintaining a safe distance from any obstacles. Essentially, we aim to find the following optimal translation function:

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

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, directly mapping 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 $[0, 1]$. 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 $[-1, 1]$.

[1]The associated code can be accessed via https://github.com/amjad-majid/ROS2-DRL-Turtlebot3-like-LIDAR-Robot.

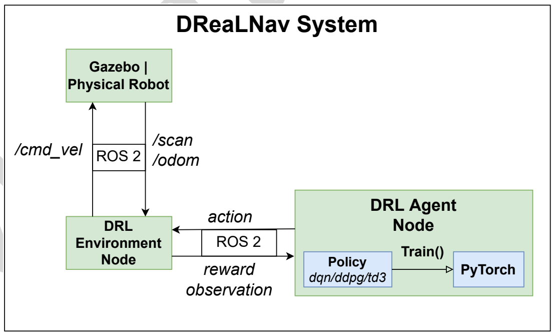
Depending on the model configuration, the linear velocity ranges from zero velocity to maximum forward velocity or from maximum backward velocity to maximum forward velocity. The angular velocity always ranges from maximum clockwise velocity to maximum counter-clockwise velocity.

3.2. Materials & Methods

This section describes the tools and methods used for the experiments in simulation and discusses the system architecture as a whole (Figure 2).

3.3. Simulator

Figure 2: 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.

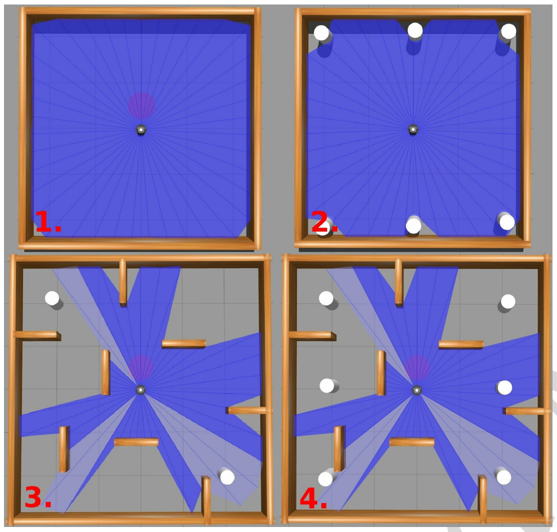


Agents are trained in simulation to largely automate and accelerate the training process without the risk of damaging equipment. However, contemporary simulators cannot perfectly model the complexity of physical properties, leading to the simulation-to-reality gap [56]. This gap can be partially mitigated by introducing noise into the simulation to lower the dependency of the policy on the precision of input data. Selecting a simulator involves trade-offs between performance, accuracy, and flexibility. We chose the Gazebo simulator [27] for its balance between performance, accuracy, and implementation speed. Gazebo is a popular open-source 3D robotic simulator with a robust physics engine and wide support for ROS, mobile robots, and optical sensors.

3.3.1. Environment

Four different training and testing scenarios were built, as showcased in Figure 3. The first scenario involves an area of approximately 4.2×4.2 meters surrounded by walls with no dynamic obstacles, serving 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 includes seven additional static walls placed across the area and two dynamic obstacles, forming the main scenario used in most experiments due to its balanced difficulty. The fourth scenario is similar to the third but with four additional dynamic obstacles, functioning as the final and most challenging test for the best-performing policies. Goal positions are designated randomly from valid locations with a sufficient distance margin to any obstacle.

Figure 3: 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).



3.3.2. 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 evaluation, the following metrics are collected for all 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.
- **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 experiments, the following additional metric is also collected:

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

3.4. Physical System

As part of the DRL autonomous navigation platform, an actual physical robot was developed to employ the policies trained in simulation for real-world autonomous navigation. This section provides an overview of the robot's hardware configuration, designed with two key characteristics in mind: cost and customizability. The goal is to create an inexpensive robot while maintaining the ability to scale its computational capabilities according to the task's demands. Below are the details of the hardware components:

Mainboard. The main controller for our system is the Jetson Nano board developed by Nvidia, running Linux Ubuntu 20.04. The Jetson Nano features a dedicated 128-core Maxwell GPU capable of efficiently running neural networks while preserving battery power. The board has sufficient computational power to run the trained DRL policies and is available for less than €100 per unit. Additionally, the Jetson module offers flexibility as it can be easily swapped out for other modules from the Nvidia Jetson family.

LiDAR. The current setup includes 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 expensive at approximately €600, it can be replaced with cheaper variants such as the RPLIDAR A1, sold at €100. The less expensive RPLIDARs offer sufficient range, accuracy, and sampling frequency for indoor navigation. For our application, we only require 40 scan samples at a sampling frequency of 10 Hz with a maximum range of 3.5 meters. We modified the RPLIDAR driver software to reduce the overall system latency by only using the 40 samples of interest.

Low-level controller. To simplify the design process, an Arduino Mega 2560 is added as a second controller for the robot's low-level functions. The Arduino Mega handles the PWM signals for motor control and manages the interrupt handling for the tachometers. It is connected to the Jetson Nano via a UART channel, over which ROS messages are sent using the Rosserial Arduino ROS package.

Motors and tachometer. The robot uses two Chihai Gm25-370 300 RPM DC gear motors with integrated interrupt-based magnetic encoders functioning as tachometers. The tachometers provide odometry to the robot based on kinematic calculations as described in [9]. The motors are connected to the system through an L298N DC motor driver module

, which regulates the motors' speed and direction.

Power and Chassis. The entire system is powered by three rechargeable 4.2 V Li-Po batteries connected to two DC-DC boost converter modules, providing a 5V power source for the Jetson Nano and a 12V power source for the Arduino Mega and L298N motor driver module. The chassis is 3D printed.

4. Results

4.1. Navigation Performance

This section examines the impact of different hyperparameter and laser scan configurations on navigation performance. It also compares various off-policy algorithms and evaluates the best-performing policy across different stages to demonstrate the system's generalization capability.

5. Hyperparameter Tuning

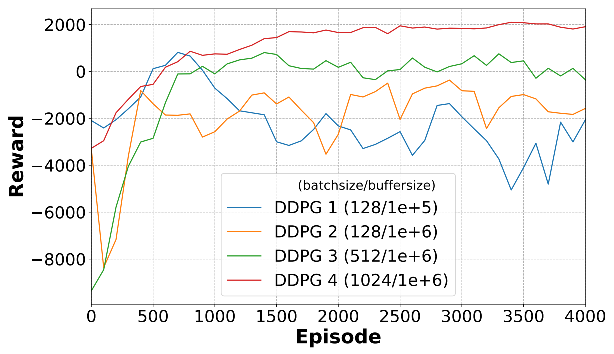
As DDPG forms the basis for most off-policy actor-critic algorithms, we start our experiments using the DDPG algorithm to evaluate and select hyperparameter values that are common to all actor-critic algorithms. While finding the exact optimal hyperparameter configuration is difficult and forms a research field of its own [2,8,53], 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 preceding results and then retrain the policy under the same conditions. Table 1 shows the results of the first experiment conducted in stage 3 with the DDPG hyperparameter configurations and corresponding evaluation metrics. Additionally, Figure 4 shows the reward scores over time during training, averaging over 100 episodes. The initial parameters for model DDPG 0 were based on [47] and the baseline implementations by OpenAI[2].

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, which contains static and dynamic obstacles. In this stage, the model achieved only a 69% success rate, with most failures being static collisions. Evidently, the current configuration is insufficient for our environment setup, necessitating further investigation of the hyperparameters to improve performance. We increased the laser scan density from 10 to 40 individual samples for DDPG 1, as the initial configuration's laser scan samples were too sparse, impeding the agent's ability to detect obstacles in time. Although this adjustment initially showed no improvement, we hypothesize that as the other hyperparameters become more finely tuned, the increased scan samples will enhance performance.

Replay Buffer Size. Analyzing the reward graph for DDPG 1 in Figure 4, we observe that while the agent learns a viable policy, it remains unstable throughout the training session, with large oscillations and drops in the reward curve. The first significant drop in performance occurs around the 1000-episode mark, coinciding with the experience replay buffer filling up. Once the replay buffer is full, old experiences are replaced by newer entries. If the replay buffer is too small, the policy may forget important previous experiences and rely only on the most recent data, potentially causing over-fitting and catastrophic forgetting [1]. To address this, we increased the size of the replay buffer for DDPG 2 to stabilize the learning process and prevent significant policy relapses. Figure 4 suggests that the increased buffer size positively affects training stability, as the reward curve for DDPG 2 shows reduced oscillations and drops, along with a higher overall reward. Table 1 confirms the improved performance, with a significant reduction in static collisions for DDPG 2. We do not further increase the buffer size, as it should be limited to reduce the probability of storing irrelevant experiences. A larger replay buffer may store older, less relevant experiences from earlier, less efficient policy iterations, ultimately slowing down the training process.

Batch Size.

Figure 4: The average reward per 100 episodes for different batch size and replay buffer size configurations in stage 3 (Figure 3).



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

Table 1: Turtlebot3 navigation performance for DDPG configurations in stage 3. BS=Batch Size, RB=Replay Buffer Size, LR=Learning Rate, CS=Collision, CD=Collision Dynamic, TO=Timeout

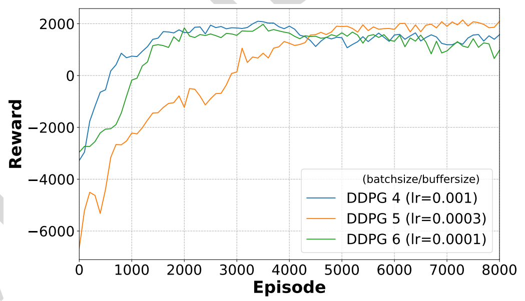
Training					Evaluation						
Model	**BS**	**RB**	**Discount**	**LR**	**Success**	**CS**	**CD**	**TO**	**Dist**	**Time**	**Speed**
DDPG 1 in stage 1	128	1e+5	0.99	1e-3	100	0	0	0	3.45	20.11	0.78
DDPG 1	128	1e+5	0.99	1e-3	67	31	0	2	3.44	16.22	0.96
DDPG 2	128	1e+6	0.99	1e-3	75	13	7	5	3.94	17.12	0.99
DDPG 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**

The batch size is another crucial hyperparameter that affects both the speed and stability of the training process. A larger batch size increases computational parallelism, allowing more samples to be processed per second, and provides a better estimate of the error gradient by processing more samples per step [4,35]. Consequently, larger batch sizes typically result in better optimization of the objective function and enhanced stability. However, this comes at the expense of slower convergence due to the increased number of samples being processed. However, larger batch sizes can also lead to poor generalization, require a larger memory footprint, and necessitate additional processing power to maintain smooth training.

In their 2017 work, Tai et al. [47] configured a batch size of 128 per training step. Since then, the computational capacity of machines has improved, and machine learning software libraries have been further optimized, making it worthwhile to explore larger batch sizes for improved stability and convergence. Figure 4 shows the average reward during training for different batch sizes. The graph reveals that a higher batch size significantly reduces fluctuations in the reward curve, improving training stability. The highest batch size, DDPG 4, results in better performance and a more favorable reward curve compared to the other configurations, as it processes more samples per timestep, guiding it toward better optimization. We fix the batch size at 1024 for the remaining experiments.

Learning Rate. Lastly, the learning rate and tau parameter are critical factors affecting 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 [4]. A larger learning rate allows the model to learn faster but risks converging early to a sub-optimal solution or not converging at all, whereas a smaller learning rate provides more stability at the expense of longer training times.

Figure 5: The average reward per 100 episodes for different learning rates in stage 3.



Upon examining the reward graph for the different learning rates in Figure 5, it is not immediately clear which model performs best, as some models show a downward trend in the reward graph later in the training process. However, DDPG appears to deliver the most stable results and achieves the highest average reward. Although a lower learning rate might eventually lead to better optimization with more training time, part of the objective is to balance performance with training duration. Therefore, we have constrained the training period to 40 hours. During evaluation, DDPG 4 achieved a success rate of 97%. Lowering the learning rate slightly further optimized the solution for DDPG 5, resulting in a success rate of 99% and lower averages for both distance and time. Further decreasing the learning rate for DDPG 6 did not yield any additional performance improvements. Thus, the learning rate and tau parameters are fixed at 3e−4.

5.0.1. LiDAR Configuration

After establishing a set of hyperparameters with acceptable performance, we turned our attention to the number of scan samples and their effect on obstacle avoidance. Table 2 shows the evaluation results for several DDPG policies

Table 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

Figure 6: The average reward per 100 episodes for different DDPG laser scan densities in stage 3.

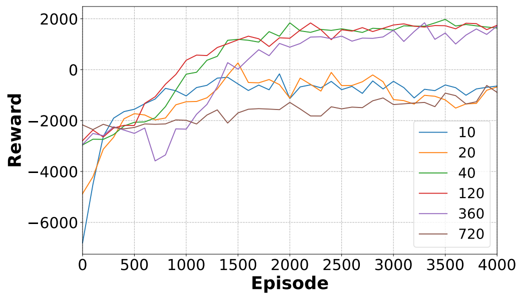
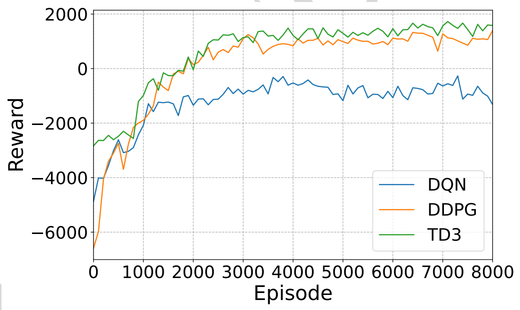


Figure 7: The average reward per 100 episodes for different DRL algorithms in stage 4.



with different laser scan densities tested in stage 3. The laser scans are distributed evenly across 360° around the robot. For the current environment, configurations with fewer than 40 scan samples severely deteriorate the agent's performance. This issue arises from the shapes and dimensions of the obstacles used in the simulation. Without a sufficiently high scan density, the agent cannot reliably detect the corners of static obstacles, often resulting in collisions when attempting to maneuver around the endpoints of walls. Additionally, lower scan densities may cause dynamic obstacles to fall between adjacent scan points, rendering them undetectable until they are in close proximity to the agent, thus complicating obstacle avoidance.

With 40 laser scan samples, the agent can detect obstacles early enough to avoid collisions and successfully navigate 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, resulting in either no significant difference or slightly worse performance. Configurations with fewer scan samples reduce the number of inputs for the neural network, simplifying the learned policy. Among the best-performing models, the configuration with 40 scan samples also gives the best performance in terms of average distance and time per episode. Figure 6 shows the average reward graphs for the DDPG models with different scan densities, which align with the evaluation results. Around the 1500-episode mark, the difference in performance becomes evident as the rewards for policies with lower scan densities stagnate. The other policies follow a fairly similar curve, with the 40-sample configuration reaching the highest average reward.

It is important to note that the minimum detectable obstacle size depends 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. Different environment settings with various obstacle sizes might require different scan sample densities to achieve optimal performance. However, given that additional laser scans provide 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.

6. Algorithm Selection

Off-policy algorithms like DQN [51], DDPG [28,44], and TD3 [15] decouple the experience collection and training process. This allows the agent to explore and optimize simultaneously, reusing previous experiences for greater efficiency. DQN uses a deep neural network (DNN) to approximate the Q-function for complex problems. DDPG, designed for continuous environments, features two DNNs:

Table 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

an actor proposing actions and a critic determining their quality. TD3 improves on DDPG by introducing a second Q-function and adding noise to target actions to reduce the overestimation of Q-values. It also updates the policy network less frequently, allowing Q-function minimization before generating Q-values for policy updates.

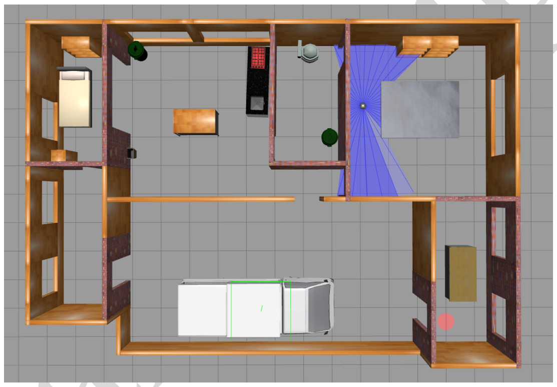
To determine which off-policy DRL algorithm is best suited for our application, we trained separate policies for DQN, DDPG, and TD3 with the same configuration and compared the results. Table 3 shows the evaluation results for the three different algorithms. As expected, DQN performs the worst since it is designed for discrete action spaces, whereas the navigation problem corresponds to a continuous domain. DDPG achieves a much higher success rate as it is tailored 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 an improved iteration of DDPG. However, TD3 takes a slightly more conservative approach, with a lower speed and greater average distance traveled.

Figure 7 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 appears slightly more unstable, as the average reward drops significantly at some points. Given that TD3 demonstrates the best performance without increasing training time, we chose it as the algorithm for the remaining experiments.

7. Generalization

After completing the reward function design and tuning the hyperparameters, the best-performing TD3 policy is evaluated in an unseen scenario with different dimensions and features to validate the model's generalizability. Table 4 shows the performance of the TD3 policy in different scenarios. Stage 5 (Figure 8) simulates a realistic house environment in an area of 15 $\times$ 10 meters, significantly larger than the training stage of 6 $\times$ 6 meters. Additionally, stage 5 features multiple differently shaped static obstacles resembling common household objects.

Figure 8: Stage 5 (15×10 m) features larger dimensions than seen during training to verify the generalizability of the agent.



During the evaluation, the policy achieved a 94% success rate in stage 5. Of the failed trials, 82% were terminated due to timeouts, while only 18% were due to collisions. The high incidence of timeouts can be attributed to the larger area of stage 5, which includes extended contiguous obstacles that sometimes cause the robot to become ensnared in a loop, repeatedly navigating the same region. Since the robot lacks memory capability, it does not recognize the repeating trajectory, causing it to repeat the same circular route until timeout. Additionally, during training, the LiDAR sensor's range is limited to a maximum of 3.5 meters, a distance rarely exceeded in the training stage. In stage 5, however, 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 paper.

7.1. Dynamic Obstacles

In the previous section, we demonstrated that the trained policy can achieve a success rate of up to 97% in stage 3. According to the evaluation results, most of the remaining failures are due to collisions with dynamic obstacles. In fact, Table 4 shows that the TD3 policy can achieve a 100% success rate in stage 2 when all dynamic obstacles are removed. It is important to 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 while attempting to avoid a dynamic one.

Table 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 [3]	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

Dynamic obstacles present a more challenging problem for obstacle avoidance as they require the agent to have a temporal understanding of the environment and consider the trajectory of moving obstacles. Observations of the agent in action suggest that dynamic collisions are likely due to the agent's inability to process the relationship between consecutive scan frames. Since the agent processes only a single frame of LiDAR distance readings per step, it cannot distinguish between moving and static obstacles or deduce the velocity and direction of a dynamic obstacle. Moreover, interactions with dynamic obstacles generally occur less frequently than with static obstacles, causing the policy to be more biased towards handling static obstacles. As a result, when approaching a dynamic obstacle, the agent tends to steer away at the last moment, similar to its strategy for avoiding static walls. This approach works for static obstacles, but dynamic obstacles require a different strategy due to their added velocity, giving the agent less time to respond and increasing the probability of collision.

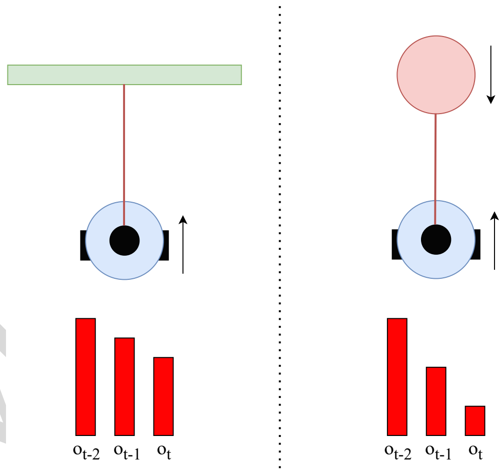
To address this, we move from stage 3 to stage 4 (Figure 3), which includes four additional dynamic obstacles to increase the number of interactions with dynamic obstacles and better train the policy for dynamic obstacle avoidance.

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

7.1.1. Frame Stacking

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

Figure 9: Frame stacking enables the agent to distinguish between static obstacles (green) and oncoming dynamic obstacles (red).



static and moving obstacles as the velocity of a moving obstacle influences the distance readings as shown in Figure 9. Frame stacking has been used before in DRL navigation systems [13,17,31] for UGVs and other applications [5,39]. 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 10 illustrates how frame stacking is implemented and how multiple frames are used as input for the neural network.

7.1.2. 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. However, 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

Table 5: 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	0	6	1	3.04	15.09	0.92
TD3 S6	10	0	95	0	2	3	3.46	17.67	0.89

Figure 10: Frame skipping and frame stepping depicted with $s_d = 3$ and $s_\epsilon = 3$.

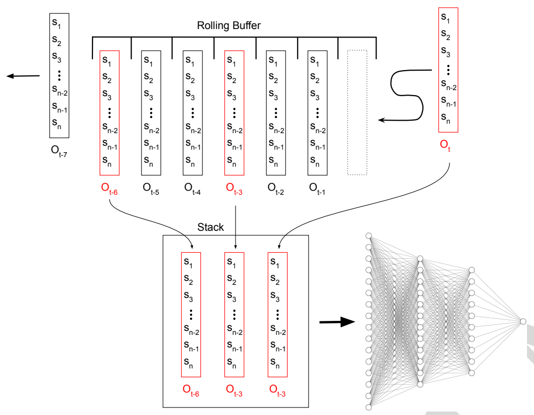
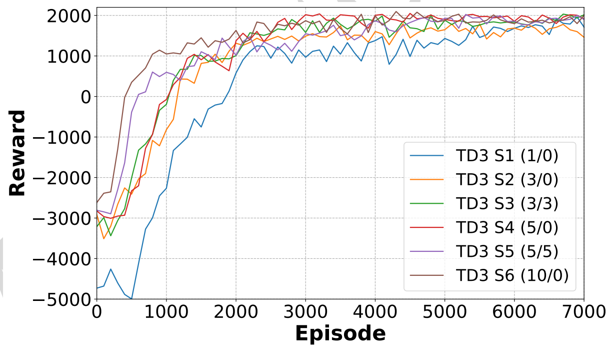


Figure 11: The average reward per 100 episodes for different frame stacking/stepping configurations in stage 2.

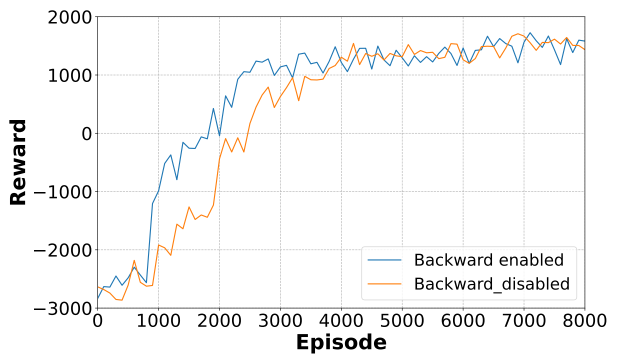


approach is to insert an artificial delay to ensure enough time has passed in between steps, but this would significantly increase the reaction time of the agent. Frame stepping enables the agent to insert any amount of time in between two subsequent input frames, without heavily affecting the model 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 10 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.

8. Simulation Results

To analyze the effect of frame stacking and frame stepping on dynamic obstacle avoidance, we trained 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 results. Table 5 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 aligns 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. Conversely, employing frame stepping on top of frame stacking did not seem to improve dynamic obstacle avoidance. A possible explanation is that the time between steps is already sufficiently large, and additional time between consecutive input frames delays the reaction speed of the agent. While frame stepping did not benefit our training setup, it may improve frame stacking performance on more powerful machines with less processing time between steps.

Figure 12: The average reward per 100 episodes showing the effect of backward motion in stage 4.



The corresponding reward graph in Figure 11 shows 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 others. This is also reflected in the evaluation results, where 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. Increasing the stack depth beyond five frames for TD3 S6 resulted in slightly worse performance in terms of success rate and path efficiency. TD3 S6 also suffered more from timeout failures, likely due to the larger number of frames being processed as input, making it harder for the agent to extract the correct information. A larger stack depth retains each frame in memory longer, including frames where 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.

8.1. Backward Motion

In most previous works on DRL UGV navigation, the agent is restricted to moving only in the forward direction and cannot move backward [58]. However, the ability to move backward can significantly improve the robot's ability to avoid obstacles, especially dynamic ones. The main arguments for omitting backward motion are twofold: 1) The LiDAR scan only needs to cover the front half 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, particularly in situations where the robot needs to react quickly, as depicted in Figure 16, where an obstacle suddenly approaches from around the corner and the robot may not have sufficient time to maneuver around it. In such scenarios, the only option is to quickly move backward to avoid a collision.

Table 6 highlights the difference in performance between configurations with backward motion disabled and enabled. While both policies 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 allows the agent to avoid collisions in almost every situation, although timeouts can still occur. The increase in success rate comes at a cost: the backward-enabled (BE) agent travels a longer distance on average. This increase in distance traveled and episode duration is expected, as the agent generally moves backward to avoid an obstacle and then 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.

The reward graph in Figure 12 shows 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 navigate around obstacles without moving backward, which is a more complex behavior. Although the BE policy yields better results during evaluation, both policies eventually 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 choice.

Additionally, incorporating backward motion enables the agent to learn more sophisticated maneuvers. Since the reward component for linear motion remains unchanged (equation 6), 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.

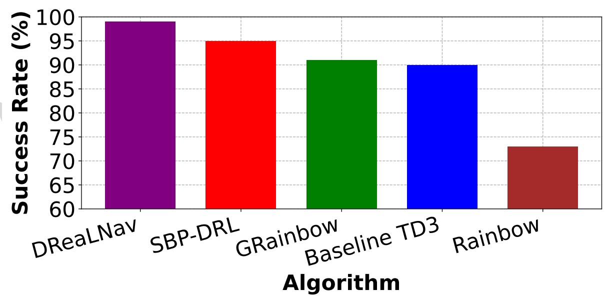
One of the challenges of training the agent with backward motion enabled is that during the exploration stage, taking purely random actions can result in the agent oscillating around its starting position as it alternates forward and backward movement commands, leading to low-quality replay buffer samples. 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.

Table 6: The effect of backward motion on navigation performance in stage 4. BD=Backward Disabled and BE=Backward Enabled

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

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 13 shows the success rates for each of these algorithms. The baseline TD3 is implemented by Gao et al. [16] as a local planner for their long-range navigation system. GRainbnow [34] combines Genetic Algorithms with DRL to reduce the sensitivity to hyperparameter tuning. The same authors have also evaluated the standard Rainbow method [19] which achieved the lowest score out of all algorithms. Corsi et al. [7] 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 and dynamic obstacles and is generally more challenging than the environments in the other works presented in Figure 13. 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 13: The success rate of different mapless navigation algorithms in simulation from the following papers: Rainbow [19], Baseline TD3 [16], GRainbow [34], SBP-DRL [7].



8.2. Reward Design & 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.

8.2.1. Reward Components

A well-designed reward function is essential for achieving good navigation performance within a feasible training time [55]. 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 [22]. Through reward shaping [36] 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 [58] 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. [47] and Long et al. [31]. 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:

$$r_{d_1} = c_d * (d_t - d_{t-1}) \tag{2}$$

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:

$$r_{d_2} = c_d * \frac{d_t - d_{t-1}}{v_{l_{max}} * (t_i - t_{i-1})} \tag{3}$$

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:

$$r_{d_3} = c_d * \frac{2 * d_0}{d_0 + d_t} \tag{4}$$

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.

$$r_\alpha = -\alpha_g \tag{5}$$

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.

$$r_{v_l} = -c_l * (v_{l_{max}} - v_l)^2 \tag{6}$$

Steering Velocity. Long et al. [31] and Choi et al. [6] 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. By employing swaying, the agent moves forward while continuously alternating between high negative and positive angular velocities, leading to a swaying motion. Excessive angular velocities may cause the agent to sway excessively in the opposite direction, resulting in overcompensation. While this behavior does not necessarily impede the agent from achieving satisfactory performance in simulation, it presents significant challenges in real-world scenarios, where physical constraints, mechanical wear, and limited battery capacity are factors. 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:

$$r_{v_a1} = -c_a * v_a^2 \tag{7}$$

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. [31] and Choi et al. [6] 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.

$$r_{v_a2} = \begin{cases} -c_a * |v_s|, & \text{if } v_s > |\pi/4| \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

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:

$$r_{ob_1} = \begin{cases} -20, & \text{if } d_{min} < d_o \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

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.

$$r_{ob_2} = \begin{cases} \dfrac{d_{min} - d_{col}}{d_o - d_{col}} & if d_{min} < d_o \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

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

$$r_{termination} = \begin{cases} 2500, & \text{if } d_t < d_{goal} \\ -2000, & \text{if } d_{min} < d_{collision} \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

8.2.2. Reward Functions

To evaluate the effectiveness of different reward components, we train multiple models using various composite reward functions. Each DDPG model is trained for approximately 40 hours, after which the best-performing iteration is evaluated over 100 trials. The outcome of each episode is recorded in Table 7, along with the average distance covered and episode duration for all successful episodes. The sway index, representing the variance in steering, is calculated by summing the squared differences between consecutive angular actions. A higher sway index indicates undesirable 'swaying' behavior, reducing system stability.

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 2) as seen in the works of Tai et al. [47] and Kato et al. [26].

Next, the reward function R_B is derived from the study by Long et al. [31] 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 obtained insights results in the best-performing reward function R_G with a 95% success rate. Although it's difficult to specify each component's contribution, omitting any component reduces performance. Furthermore, testing alternative components in reward functions R_H, R_I, and R_J showed no significant improvement in any evaluation metrics.

Figure 14 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 part with R_G being the reward function of choice for our experiments.

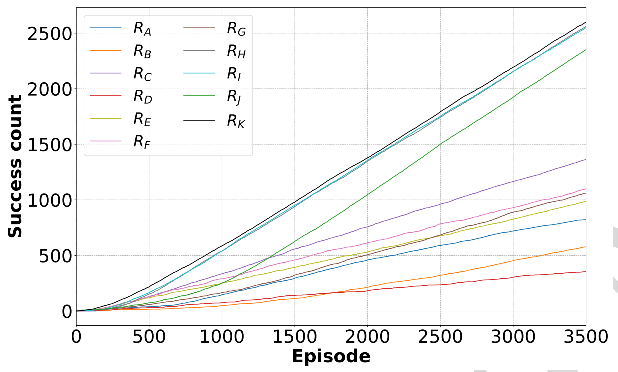
9. Physical System Validation

This section evaluates the performance and robustness of trained policies in various real-world scenarios.

Table 7: Evaluation of different reward functions with varying components in stage 3.

Function	Components	Success	CS	CD	TO	Dist	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

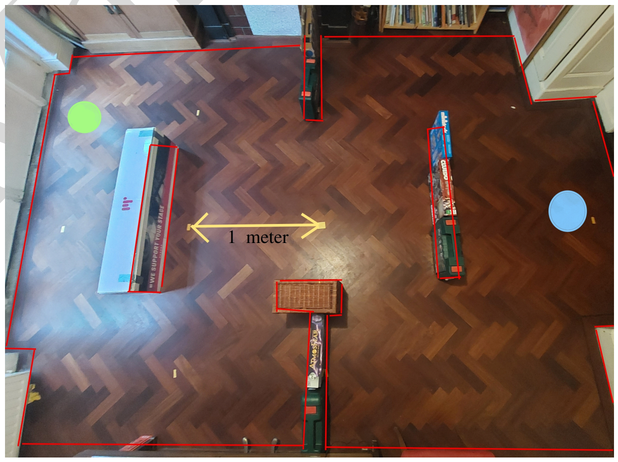
Figure 14: The success rates over time during training for different reward functions.



9.1. Real-World Evaluation

One of the challenges in transferring the navigation policy from simulation to the real world is the precise tuning needed to accurately translate model output actions into 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.

Figure 15: The stage used for the real-world experiment presented in Table 8. The starting position and goal are indicated by the blue and green circles.



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 15. 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 8 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

Table 8: Evaluation on the physical robot over the same trajectory for 20 trials in a real-world stage.

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

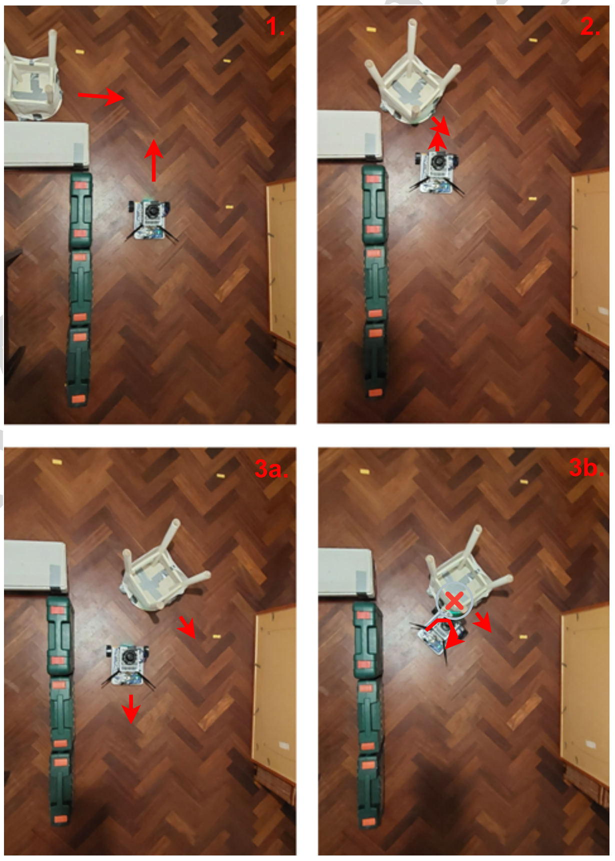
well as inaccuracies in the distance measurements, odometry, and motor output causing deviations from the desired trajectory which are corrected by the robot, all resulting in longer paths. Despite being trained in an environment with different obstacles and dimensions the robot is still able to navigate reliably in the unseen scenario.

9.2. Backward Motion

To demonstrate the viability and benefit of backward motion in the real world, we reproduced a realistic situation in which the agent has to rely on backward motion in order to avoid a collision as shown in Figure 16. 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

In addition, a further demonstration of how backward motion—compared to forward-only motion—to can help the robot respond quickly in tricky situations is made available online https://youtu.be/-eyNCvBohRk.

Figure 16: 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 it. **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.



10. Discussion

Hyperparameter Tuning: The hyperparameter tuning process highlighted the significant impact of parameters such as replay buffer size, batch size, and learning rate on the navigation performance of the Turtlebot3 using the DDPG algorithm. Our experiments demonstrated that increasing the replay buffer size from 1×10^5 to 1×10^6 enhanced the stability of the learning process, reducing oscillations in the reward curve and leading to improved performance in static collision avoidance. This finding aligns with previous research indicating that a larger replay buffer helps mitigate the effects of catastrophic forgetting and overfitting by retaining a more diverse set of experiences [1].

Similarly, the adjustment of the batch size to 1024 significantly improved the stability of training, resulting in smoother reward curves and better overall performance. Larger batch sizes allow for better gradient estimates, which contribute to more stable learning processes [4,35]. However, it is essential to balance batch size and computational resources, as excessively large batch sizes can lead to diminished returns due to increased memory requirements and slower convergence rates.

The learning rate tuning experiments underscored the necessity of a carefully balanced learning rate to achieve optimal performance. A learning rate of 3×10^{-4} provided a good trade-off between training stability and speed, leading to the highest success rate of 99% for the DDPG 5 model. This result is consistent with the literature suggesting that an appropriately tuned learning rate is critical for avoiding premature convergence to suboptimal policies or instability during training [4].

LiDAR Configuration: Our investigation into LiDAR scan densities revealed that a configuration of 40 scan samples provided the best performance for the Turtlebot3 in environments with both static and dynamic obstacles. This configuration allowed the agent to detect obstacles early enough to avoid collisions, achieving a 94

The results emphasize the importance of selecting an appropriate scan density that balances the need for sufficient environmental information with the computational and learning complexity of the neural network. In environments with different obstacle shapes and sizes, it may be necessary to adjust the scan density to achieve optimal performance.

Algorithm Selection: Our comparison of off-policy algorithms, including DQN, DDPG, and TD3, demonstrated that TD3 outperformed the others, achieving the highest success rate of 97

The reward curves for the different algorithms reinforced these results, with TD3 achieving the highest overall scores and the most stable learning process. Given its superior performance and stability, TD3 was selected as the algorithm for the remaining experiments.

Generalization: The generalization capability of the TD3 policy was evaluated in different stages, including a significantly larger and more complex environment in stage 5. The policy demonstrated strong generalization, achieving a 94

Dynamic Obstacles: Dynamic obstacles posed a significant challenge, with most failures in stage 3 attributed to collisions with moving obstacles. Our experiments with frame stacking and frame stepping indicated that frame stacking provided a slight benefit by enabling the agent to develop short-term memory and better anticipate the trajectories of moving obstacles. However, frame stepping did not yield additional improvements, likely due to the increased reaction time required.

Backward Motion: The introduction of backward motion significantly improved the agent's ability to avoid collisions, especially in scenarios involving dynamic obstacles. The backward-enabled (BE) policy achieved a higher success rate and demonstrated more complex maneuvers, such as reversing out of tight situations. The reward graph showed faster initial learning for the BE policy, indicating that the ability to move backward facilitated quicker adaptation to the environment.

Overall, the experiments highlighted the importance of carefully tuning hyperparameters, selecting appropriate LiDAR configurations, and designing effective reward functions to achieve optimal navigation performance in complex environments. The TD3 algorithm, with its superior stability and performance, proved to be the best choice for the Turtlebot3 navigation task, demonstrating strong generalization and robustness in both simulated and real-world scenarios.

11. Conclusion

In this study[4], we developed a robust deep reinforcement learning navigation stack capable of autonomous navigation and obstacle avoidance in both simulation and real-world environments, achieving a 99% success rate in complex, dynamic conditions. Our work demonstrated that incorporating backward motion and 360-degree LiDAR coverage could significantly enhance navigation performance, challenging traditional conventions. Furthermore, we reduced swaying through a carefully designed reward function, resulting in smoother robot trajectories.

However, several factors limit the current system's potential. The robot hardware, which is still at the prototype stage, requires improvements—particularly in the steering mechanism and LiDAR placement. Although the current system avoids convolutional and transformer networks to minimize complexity, these techniques represent promising avenues for future enhancements to the robot's obstacle-avoidance capabilities. Additionally, integrating an RGB camera alongside the LiDAR could provide valuable contextual information for improved dynamic obstacle avoidance, despite the increased complexity and effort required to maintain transferability.

It could be deduced that our study provides a compelling foundation for the development of autonomous robotic systems and prompts further investigation into

[4]This work is derived from the MSc Thesis in [48].

hardware design and the potential use of convolutional and transformer networks.

Authors' Contribution

Conceptualization, methodology, developed the hardware, and participated in software development and writing: Amjad Yousef Majid; writing code, conducting experiments, and participated in writing: Tomas van Rietbergen; Overall guidance: R Venkatesha Prasad.

Availability of Data and Materials

The code for this study can be accessed via https://github.com/amjadmajid/ROS2-DRL-Turtlebot3-like-LIDAR-Robot. No external data has been used.

Consent for Publication

Not applicable

Conflict of Interest

The authors declare no conflicts of interest regarding this manuscript.

Funding

No external funding was received for this research.

Acknowledgment

The authors would like to express their gratitude to Dr. Chris Verhoeven for his valuable input and guidance throughout this research. Additionally, we extend our thanks to the reviewers for their time, effort, and constructive feedback, which have significantly contributed to the improvement of this work.

References

1. Joshua Achiam. Spinning up in deep reinforcement learning. 2018. URL https://spinningup.openai.com/en/latest/index.html.
2. Nesma Ashraf, Reham Mostafa, Rasha Sakr, and M. Rashad. Optimizing hyperparameters of deep reinforcement learning for autonomous driving based on whale optimization algorithm. *PLOS ONE*, 16: e0252754, 06 2021. doi: 10.1371/journal.pone.0252754.
3. Philip J Ball, Laura Smith, Ilya Kostrikov, and Sergey Levine. Efficient online reinforcement learning with offline data. In *International Conference on Machine Learning*, pages 1577–1594. PMLR, 2023.
4. Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *CoRR*, abs/1206.5533, 2012. URL http://arxiv.org/abs/1206.5533.
5. Devendra Singh Chaplot. Transfer deep reinforcement learning in 3 d environments : An empirical study. 2016.
6. Jinyoung Choi, Kyungsik Park, Minsu Kim, and Sangok 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)*, pages 5993–6000, 2019. doi: 10.1109/ICRA.2019.8793979.
7. Davide Corsi, Raz Yerushalmi, Guy Amir, Alessandro Farinelli, David Harel, and Guy Katz. Constrained reinforcement learning for robotics via scenario-based programming, 2022. URL https://arxiv.org/abs/2206.09603.
8. Xingping Dong, Jianbing Shen, Wenguan Wang, Yu Liu, Ling Shao, and Fatih Porikli. Hyperparameter optimization for tracking with continuous deep q-learning. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 518–527, 2018. doi: 10.1109/CVPR.2018.00061.
9. Gregory Dudek and Michael Jenkin. *Computational Principles of Mobile Robotics*. Cambridge University Press, USA, 2nd edition, 2010. ISBN 0521692121.
10. Mihai Duguleana and Gheorghe Mogan. Neural networks based reinforcement learning for mobile robots obstacle avoidance. *Expert Systems with Applications*, 62:104–115, 2016. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2016.06.021. URL https://www.sciencedirect.com/science/article/pii/S0957417416303001.
11. H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *IEEE Robotics & Automation Magazine*, 13(2):99–110, 2006. doi: 10.1109/MRA.2006.1638022.
12. Muhammad Mudassir Ejaz, Tong Boon Tang, and Cheng-Kai Lu. Vision-based autonomous navigation approach for a tracked robot using deep reinforcement learning. *IEEE Sensors Journal*, 21(2):2230–2240, 2021. doi: 10.1109/JSEN.2020.3016299.
13. Tingxiang Fan, Xinjing Cheng, Jia Pan, Dinesh Manocha, and Ruigang Yang. Crowdmove: Autonomous mapless navigation in crowded scenarios. *CoRR*, abs/1807.07870, 2018. URL http://arxiv.org/abs/1807.07870.
14. D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997. doi: 10.1109/100.580977.
15. Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018. URL http://arxiv.org/abs/1802.09477.
16. Junli Gao, Weijie Ye, Jing Guo, and Zhongjuan Li. Deep reinforcement learning for indoor mobile robot path planning. *Sensors*, 20(19), 2020. ISSN 1424-8220. URL https://www.mdpi.com/1424-8220/20/19/5493.

17. Lingping Gao, Jianchuan Ding, Wenxi Liu, Haiyin Piao, Yuxin Wang, Xin Yang, and Baocai Yin. A vision-based irregular obstacle avoidance framework via deep reinforcement learning. *CoRR*, abs/2108.06887, 2021. URL https://arxiv.org/abs/2108.06887.
18. Daniel Gehrig and Davide Scaramuzza. Low-latency automotive vision with event cameras. *Nature*, 629 (8014):1034–1040, 2024.
19. Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. URL http://arxiv.org/abs/1710.02298.
20. Noriaki Hirose, Dhruv Shah, Kyle Stachowicz, Ajay Sridhar, and Sergey Levine. Selfi: Autonomous self-improvement with reinforcement learning for social navigation. *arXiv preprint arXiv:2403.00991*, 2024.
21. David Hoeller, Nikita Rudin, Dhionis Sako, and Marco Hutter. Anymal parkour: Learning agile navigation for quadrupedal robots. *Science Robotics*, 9(88):eadi7566, 2024.
22. Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Curiosity-driven exploration in deep reinforcement learning via bayesian neural networks. *CoRR*, abs/1605.09674, 2016. URL http://arxiv.org/abs/1605.09674.
23. Han Hu, Kaicheng Zhang, Aaron Hao Tan, Michael Ruan, Christopher Agia, and Goldie Nejat. A sim-to-real pipeline for deep reinforcement learning for autonomous robot navigation in cluttered rough terrain. *IEEE Robotics and Automation Letters*, 6(4): 6569–6576, 2021.
24. Wenhui Huang, Yanxin Zhou, Xiangkun He, and Chen Lv. Goal-guided transformer-enabled reinforcement learning for efficient autonomous navigation. *IEEE Transactions on Intelligent Transportation Systems*, 2023.
25. Sheng Jin, Xinming Wang, and Qinghao Meng. Spatial memory-augmented visual navigation based on hierarchical deep reinforcement learning in unknown environments. *Knowledge-Based Systems*, 285:111358, 2024.
26. Yuki Kato, Koji Kamiyama, and Kazuyuki 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)*, pages 1040–1046, 2017. doi: 10.1109/SII.2017.8279360.
27. N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004. doi: 10.1109/IROS.2004.1389727.
28. Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 09 2015.
29. Haoyue Liu, Shihan Peng, Lin Zhu, Yi Chang, Hanyu Zhou, and Luxin Yan. Seeing motion at nighttime with an event camera. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 25648–25658, 2024.
30. Shuijing Liu, Peixin Chang, Weihang Liang, Neeloy Chakraborty, and Katherine Driggs Campbell. Decentralized structural-rnn for robot crowd navigation with deep reinforcement learning. *CoRR*, abs/2011.04820, 2020. URL https://arxiv.org/abs/2011.04820.
31. Pinxin Long, Tingxiang Fan, Xinyi Liao, Wenxi Liu, Hao Zhang, and Jia Pan. Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning. *CoRR*, abs/1709.10082, 2017. URL http://arxiv.org/abs/1709.10082.
32. Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022. doi: 10.1126/scirobotics.abm6074. URL https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.
33. Amjad Yousef Majid, Casper van der Horst, Tomas van Rietbergen, David JohannesZwart, and R Venkatesha Prasad. Lightweight audio source localization for swarm robots. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE, 2021.
34. Enrico Marchesini and Alessandro Farinelli. Genetic deep reinforcement learning for mapless navigation. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '20, page 1919–1921, Richland, SC, 2020. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9781450375184.
35. Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *CoRR*, abs/1804.07612, 2018. URL http://arxiv.org/abs/1804.07612.
36. Maja J Mataric. Reward functions for accelerated learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning Proceedings 1994*, pages 181–189. Morgan Kaufmann, San Francisco (CA), 1994. ISBN 978-1-55860-335-6. doi: https://doi.org/10.1016/B978-1-55860-335-6.50030-1. URL https://www.sciencedirect.com/science/article/pii/B9781558603356500301.
37. Victor RF Miranda, Armando A Neto, Gustavo M Freitas, and Leonardo A Mozelli. Generalization in deep reinforcement learning for robotic navigation by reward shaping. *IEEE Transactions on Industrial Electronics*, 2023.
38. Amjad Yousef Mjaid, Venkatesha Prasad, Mees Jonker, Casper Van Der Horst, Lucan De Groot, and Sujay Narayana. Ai-based simultaneous audio localization and communication for robots. In *Proceedings of the 8th ACM/IEEE Conference on Internet of Things Design and Implementation*, pages 172–183, 2023.
39. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep rein-

forcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

40. Van Nguyen, Tien Manh, Cuong Manh, Dung Tien, Manh Van, Duyen Ha Thi Kim Duyen, and Duy Nguyen Duc. Autonomous navigation for omnidirectional robot based on deep reinforcement learning. *International Journal of Mechanical Engineering and Robotics Research*, pages 1134–1139, 01 2020. doi: 10.18178/ijmerr.9.8.1134-1139.
41. Kwanyoung Park and Youngwoon Lee. Tackling long-horizon tasks with model-based offline reinforcement learning. *arXiv preprint arXiv:2407.00699*, 2024.
42. Seohong Park, Dibya Ghosh, Benjamin Eysenbach, and Sergey Levine. Hiql: Offline goal-conditioned rl with latent states as actions. *Advances in Neural Information Processing Systems*, 36, 2024.
43. Harry A Pierson and Michael S Gashler. Deep learning in robotics: a review of recent research. *Advanced Robotics*, 31(16):821–835, 2017.
44. David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. ICML'14, page I–387–I–395. JMLR.org, 2014.
45. Kyle Stachowicz, Dhruv Shah, Arjun Bhorkar, Ilya Kostrikov, and Sergey Levine. Fastrlap: A system for learning high-speed driving via deep rl and autonomous practicing. In *Conference on Robot Learning*, pages 3100–3111. PMLR, 2023.
46. Hartmut Surmann, Christian Jestel, Robin Marchel, Franziska Musberg, Houssem Elhadj, and Mahbube Ardani. Deep reinforcement learning for real autonomous mobile robot navigation in indoor environments. *CoRR*, abs/2005.13857, 2020. URL https://arxiv.org/abs/2005.13857.
47. Lei Tai, Giuseppe Paolo, and Ming 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)*, pages 31–36, 2017. doi: 10.1109/IROS.2017.8202134.
48. Tomas van Rietbergen. Toward reliable robot navigation using deep reinforcement learning. Master's thesis, The Netherlands, 2022. URL https://repository.tudelft.nl/record/uuid:8dda87ed-5acd-44fc-9a31-e8a60b20f43b.
49. Jiankun Wang, Tianyi Zhang, Nachuan Ma, Zhaoting Li, Han Ma, Fei Meng, and Max Q-H Meng. A survey of learning-based robot motion planning. *IET Cyber-Systems and Robotics*, 3(4):302–314, 2021.
50. Jiexin Wang, Stefan Elfwing, and Eiji Uchibe. Modular deep reinforcement learning from reward and punishment for robot navigation. *Neural Networks*, 135:115–126, 2021. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2020.12.001. URL https://www.sciencedirect.com/science/article/pii/S0893608020304184.
51. Christopher Watkins. Learning from delayed rewards. 01 1989.
52. Kasun Weerakoon, Adarsh Jagan Sathyamoorthy, Utsav Patel, and Dinesh Manocha. TERP: reliable planning in uneven outdoor environments using deep reinforcement learning. *CoRR*, abs/2109.05120, 2021. URL https://arxiv.org/abs/2109.05120.
53. Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, 2020. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2020.07.061. URL https://www.sciencedirect.com/science/article/pii/S0925231220311693.
54. Safdar Zaman, Wolfgang Slany, and Gerald 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)*, pages 1–5, 2011. doi: 10.1109/SIECPC.2011.5876943.
55. Qichen Zhang, Meiqiang Zhu, Liang Zou, Ming Li, and Yong Zhang. Learning reward function with matching network for mapless navigation. *Sensors*, 20(13), 2020. ISSN 1424-8220. doi: 10.3390/s20133664. URL https://www.mdpi.com/1424-8220/20/13/3664.
56. Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. pages 737–744, 2020. doi: 10.1109/SSCI47803.2020.9308468.
57. Kaiyu Zheng. Ros navigation tuning guide. *Robot Operating System (ROS) The Complete Reference (Volume 6)*, pages 197–226, 2021.
58. Kai Zhu and Tao Zhang. Deep reinforcement learning based mobile robot navigation: A review. *Tsinghua Science and Technology*, 26(5):674–691, 2021.