PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

Ministry of Higher Education and Scientific Research

University of Amar Telidji - Laghouat

Faculty of Technology

Department of Electronics

DOMAIN: Science & Technology

FIELD: Electronic

SPECIALTY: Instrumentation

A Master's Thesis Titled:

Deep Q-learning based motion planning for mobile robot navigation

Presented by

Brighet Ahmed Issam | Ramdani NourEddin

Jury members:

Djerfaf Fatima	Prof	President
Oubatti Khalil	MCB	Examiner
Chouireb Fatima	Prof	Supervisor

Academic year 2024/2025

Acknowledgements

First and foremost, we wish to express our profound gratitude to Almighty God, who blessed us with health, patience, and perseverance, guiding us through each step of this endeavor and making its successful completion possible.

We would like to extend our deep and sincere appreciation to our supervisor, Mme. Chouireb Fatima, for her constant support, insightful guidance, and constructive criticism. Her expertise, encouragement, and patience were an unfailing source of motivation, shaping this thesis into a more rigorous and polished piece of work. It has truly been a privilege and an honor to learn under her supervision.

Furthermore, we wish to express our warmest and heartfelt thanks to our families, whose love, understanding, and encouragement sustained us through all the obstacles we faced. To our parents, sisters, and brothers — you believed in us when we doubted ourselves, supported us through difficult moments, and remained a constant source of happiness and stability. Without your sacrifices, patience, and moral support, this accomplishment would not have been possible.

We are also sincerely grateful to the members of the jury for honoring us by evaluating this thesis and offering their constructive comments and suggestions. We appreciate their expertise and the time they have devoted to reviewing our work.

Ultimately, this accomplishment is not ours alone; it reflects the combined encouragement, patience, and love of all those who stood by us.

We sincerely thank you

Abstract

This Master thesis implements two deep-RL planners DDPG and TD3 for real-time motion planning of a mobile robot in unknown environments. Agents generate continuous linear and angular velocity commands to reach randomized goals in ROS/Gazebo simulations on Pioneer 3-DX and TurtleBot3 platforms. We train each agent across diverse obstacle layouts and analyze learning stability, sample efficiency, trajectory smoothness, and goal-reaching success. The results offer actionable insights into integrating deep-RL algorithms for robust, autonomous navigation in dynamic settings.

الملخص

في رسالة الماستر هذه نقوم بتطبيق منهجين في التخطيط التنبؤي للحركة باستخدام التعلم المعزّز العميق DDPG و TD3 لروبوت متنقل في بيئات مجهولة. يولّد الوكيل أوامر سرعات خطية وزاوية مستمرة للوصول إلى أهداف عشوائية داخل محاكاة ROS/Gazebo على منصتي Pioneer 3-DX وTurtleBot3. ندرّب كل خوارزمية عبر ترتيبات عوائق متنوعة ثم نحلّل ثبات التعلم، وكفاءة العيّنات، وسلاسة المسارات، ونسبة النجاح في الوصول. تقدّم النتائج توصيات عملية لدمج هذه الخوارزميات في نظام ملاحة ذاتي قوي في ظروف ديناميكية.

Résumé

Cette thèse de Master implémente deux planificateurs deep-RL DDPG et TD3 pour la planification de mouvement en temps réel d'un robot mobile dans des environnements inconnus. Les agents génèrent des commandes de vitesses linéaires et angulaires continues afin d'atteindre des objectifs aléatoires dans des simulations ROS/Gazebo sur les plateformes Pioneer 3-DX et TurtleBot3. Nous entraînons chaque agent sur divers agencements d'obstacles et analysons la stabilité d'apprentissage, l'efficacité par échantillon, la fluidité des trajectoires et le taux de réussite à atteindre la cible. Les résultats fournissent des recommandations concrètes pour intégrer ces algorithmes deep-RL dans une navigation autonome robuste en milieux dynamiques.

Table of Contents

List of Figures

List of Tables

List of abbreviations

CPU	Central Processing Unit
DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q Network
DRL	Deep Reinforcement Learning
FOV	Field of View
Gazebo	3D robot simulation environment
GPU	Graphics Processing Unit
Hz	Hertz (sampling frequency)
IMU	Inertial Measurement Unit
LDS	Laser Distance Sensor
LiDAR	Light Detection and Ranging
MDP	Markov Decision Process
MPC	Model Predictive Control
OS	Operating System
RAM	Random Access Memory
RL	Reinforcement Learning
ROS	Robot Operating System
SARSA	State Action Reward State Action
SLAM	Simultaneous Localization and Mapping
TD3	Twin Delayed Deep Deterministic Policy Gradient
WMR	Wheeled Mobile Robot

GENERAL Introduction

In recent years, mobile robotics has transitioned from structured, pre-mapped environments toward truly unstructured navigation, enabled by advances in deep reinforcement learning (DRL). The Deep Deterministic Policy Gradient (DDPG) agent and its improved variant, Twin Delayed Deep Deterministic Policy Gradient (TD3) can be trained end-to-end using raw 2D LiDAR scans, odometry, and random goal coordinates in unknown simulated environments. By integrating these algorithms with a ROS/Gazebo simulation framework, the agent learns continuous linear and angular velocity commands to achieve both obstacle avoidance and goal-reaching without relying on an explicit map. Furthermore, we study the impact of DDPG and TD3 on the agent's learning and behavioral performance, emphasizing how each algorithm impacts navigational ease and generalization to new environments.

- **Chapter 1** presents categories of mobile robots, including legged robots for rough terrains, space and warehouse robots leveraging reinforcement learning for autonomy, and self-driving vehicles using deep RL. Differential drive robots like the Pioneer 3-DX and TurtleBot3 are emphasized due to their simplicity and relevance in research.
- **Chapter 2** introduces the fundamentals of reinforcement learning, from tabular Q-learning to deep actor-critic methods (DQN, DDPG, TD3), and explains why DDPG and TD3 are well suited for continuous control in robotics.
- **Chapter 3** this section details our reinforcement learning implementation, including the hardware and software stack, the design of DDPG and TD3 algorithms (state/action spaces, reward shaping, and network architecture), Gazebo simulation setups, and comprehensive training and testing results for both navigation and motion planning tasks.
- Finally, we conclude with a general conclusion and propose future perspectives for extending this research.

Chapter 1: Mobile Robotics and Sensors

1.1 Introduction

Chapter 1 provides a comprehensive overview of the diverse landscape of mobile robotic platforms, beginning with legged robots capable of traversing uneven and complex terrain, and extending to specialized explorers in space and multi-agent systems in warehouse automation. We then discuss self-driving vehicles that leverage deep reinforcement learning (DRL) for end-to-end policy learning from raw sensor inputs.

A detailed focus follows on differentially driven wheeled robots specifically the Pioneer 3-DX and TurtleBot3 emphasizing their mechanical simplicity, non-holonomic constraints, and suitability for indoor RL research. Finally, we introduce the key hardware and sensor suites of these platforms, setting the stage for the DRL-based navigation and motion-planning experiments presented in subsequent chapters.

1.2 Different types of mobile robots

1.2.1 Legged Robots

Legged robots are mobile systems designed to move using limbs, typically in the form of articulated legs, powered by electric or hydraulic actuators.

These robots may have two (bipedal), four (quadrupedal), or even six or more legs, allowing them to navigate a variety of terrains that would be difficult or impossible for wheeled or tracked robots. Their design is often inspired by biological organisms, enabling them to maintain mobility over irregular, uneven, or soft surfaces such as stairs, rubble, or natural landscapes. The multi-joint configuration of their legs provides high adaptability and maneuverability. However, these advantages are balanced by challenges including complex motion planning, dynamic balance control, and increased power consumption. Advances in control algorithms and lightweight materials have helped mitigate some of these issues, making legged robots increasingly practical for exploration, inspection, and disaster response tasks [1].

Figure 1.1 - Examples of a legged robot

1.2.2 Space Exploration Robots

Autonomous robotic explorers employed in space missions increasingly leverage reinforcement-learning algorithms to adaptively plan trajectories, optimize scientific data collection, and negotiate unstructured extraterrestrial terrains. By modeling rover navigation as a Markov decision process, RL agents learn policies that balance exploration of unknown regions with the exploitation of high-value scientific targets. For example, deep Q-learning has been used to enable Mars rovers to autonomously select safe traversals while maximizing information gain under communication delays and energy constraints [2]. These methods reduce reliance on pre-programmed sequences and allow on-board adaptation to unforeseen obstacles, radiation effects, and extreme temperature variations.

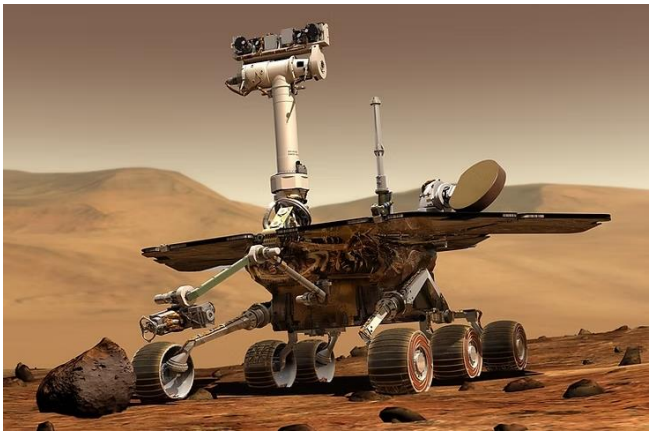


Figure 1.2 - Space Exploration Robots

1.2.3 Self-Driving Cars

In autonomous driving (see figure 1.3), reinforcement learning provides a framework for end-to-end policy learning, mapping raw sensor inputs (LiDAR, camera, radar) directly to control commands. Deep RL algorithms, such as DDPG and TD3, have been applied to learn continuous steering and acceleration policies that optimize safety, comfort, and fuel efficiency in simulation before real-world deployment [3]. By training in realistic driving simulators, RL agents can experience rare events (e.g., sudden pedestrian crossings) at scale, improving robustness. Moreover, hierarchical RL architectures enable decomposition of complex driving tasks—such as lane changing, intersection handling, and highway merging—into sub-policies that are learned and switched among dynamically.

Figure 1.3 - Self Driving Cars

1.2.4 Warehouse Robots

Modern warehouse automation employs multi-agent reinforcement learning to coordinate fleets of mobile robots for tasks including goods picking, shelving, and order consolidation. Each robot is modeled as an RL agent that learns to navigate dynamic warehouse layouts (figure 1.4), avoid collisions, and optimize throughput under time and energy constraints. Centralized training with decentralized execution—using value-decomposition networks or MADDPG—allows agents to learn cooperative behaviors, such as dynamic task allocation and traffic management, leading to significant improvements in order-processing rates and

resource utilization [4]. RL-based controllers adapt on-the-fly to changes in inventory distribution and order patterns without manual reprogramming.

Figure 1.4 - Warehouse Robots

1.2.5 Differentially Driven Wheeled Mobile Robots (WMRs)

Differentially driven WMRs, such as the P3-DX and TurtleBot3, are widely used in robotics research due to their mechanical simplicity and suitability for indoor tasks. These robots use two independently driven wheels and are often equipped with sensors like LiDAR and encoders for navigation. Reinforcement learning (RL) methods, particularly deep RL, have proven effective in enabling autonomous behaviors such as obstacle avoidance and path following without relying on detailed environmental models. Platforms like ROS and Gazebo facilitate training RL agents in simulation before deploying them to real robots, improving safety and efficiency [5].

Figure 1.5 - P3-DX mobile robot (in the right) and Turtlebot3 (in the left)

Non-holonomic WMR

This type of WMR has limited movements in 2D environment, it can be only driven straight (forward/ backward), on a curve and turning around spot. However, non-holonomic WMRs have many types depending on wheel's types and their position on the robot. Figure 1.6 shows different designs of non-holonomic WMR [6].

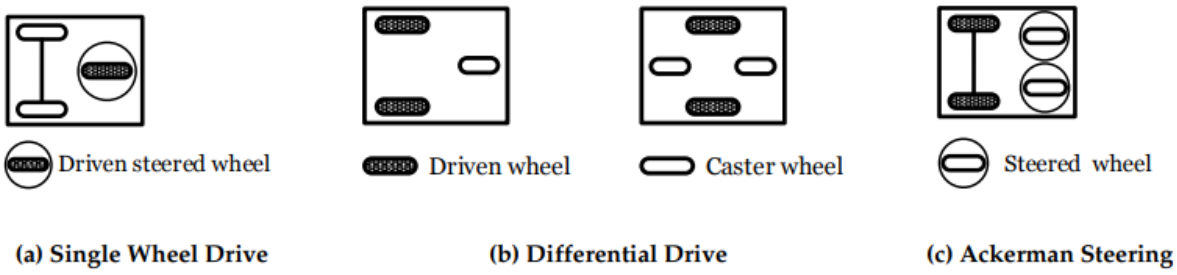


Figure 1.6 - Non-holonomic WMR types

a) Single Wheel drive

The mechanical design of this kind is shown in figure 1.6.a, the robot consists of three wheels; two free wheels and one driven steered wheel. This kind of robot is very simple to program, but the robot cannot turn around the center of the robot.

b) Differential Drive

It is the most common type of WMR, it consists of two driven wheels and one or two free wheels depending on the position of driven wheels. Figure 1.6b displays the two design of differential drive WMR. The main advantage of this design is the capability of rotating around spot. Besides driving it is straightforward by controlling velocity of left and right wheels. Table 1.1 presents the three different cases of driving differential drive WMR and the relation between velocities of left and right wheels for each case.

c) Ackerman Steering

this style like cars, has four wheels. The front wheels are for steering, while the rear ones for driving. Figure 1.6c presents Ackerman steering design [7].

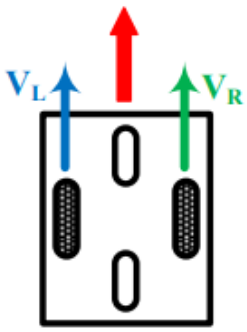
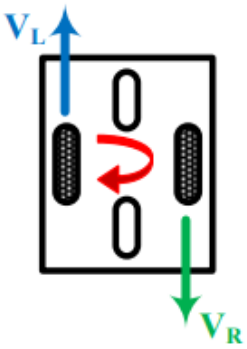
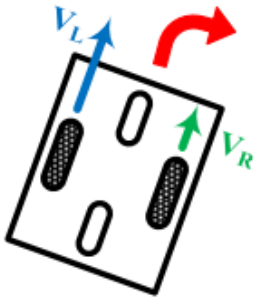
Movement	Straight	Turning	On Curve
Robot			
Velocity	$V_L = V_R$	$V_L = - V_R$	$V_L > V_R$

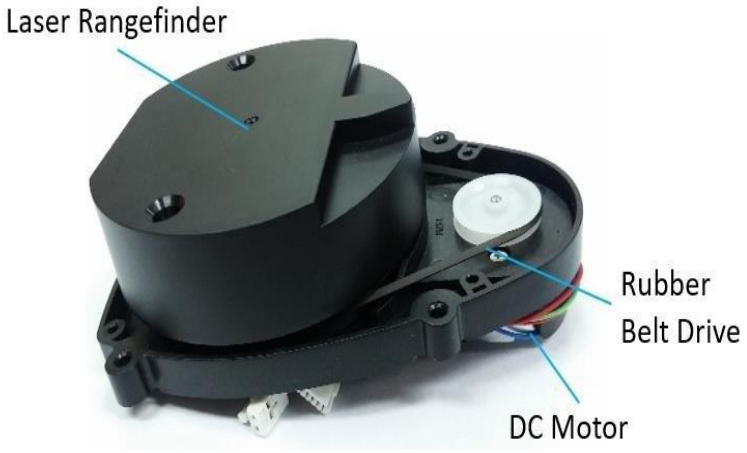
Table 1.1 - Driving Differential Drive WMR [24].

1.3 Turtlebot3 components

As previously mentioned, the Turtlebot3 is a widely used differential drive mobile robot in research. Its model in Gazebo was selected as a platform for our work. It includes the following components.

LIDAR Sensor (LDS-01)

The LDS-01 is a two-dimensional scanning LiDAR unit that provides 360° range measurements by emitting and receiving laser pulses at up to 5.5 Hz. With an angular resolution of 0.25° and a maximum detection distance of 12 m, it enables high-fidelity mapping and obstacle detection for mobile robots. The sensor's output—polar distance measurements—is often converted into Cartesian coordinates and used by SLAM and path-planning algorithms to construct occupancy grids and local cost maps. Its lightweight design and USB interface make it straightforward to integrate with ROS-based platforms for both simulation and real-world experiments [8].



Items	Specifications
Operating supply voltage	5V DC ±5%
Light source	Semiconductor Laser Diode(λ=785nm)
LASER safety	IEC60825-1 Class 1
Current consumption	400mA or less (Rush current 1A)
Detection distance	120mm ~ 3,500mm
Interface	3.3V USART (230,400 bps) 42bytes per 6 degrees,
Ambient Light Resistance	10,000 lux or less
Sampling Rate	1.8kHz
Dimensions	69.5(W) X 95.5(D) X 39.5(H)mm
Mass	Under 125g

Figure 1.7 - LIDAR Sensor (LDS-01)

Wheel Encoders (Dynamixel Motors):

Each wheel of the TurtleBot3 is driven by a Dynamixel smart actuator (e.g., XL430-W250), which incorporates a high-resolution magnetic encoder. These encoders report the motor shaft position with a resolution of 4096 ticks per revolution, enabling precise measurement of wheel rotations. By differentiating successive encoder readings and knowing the wheel radius, the robot computes linear and angular velocities for odometry. Encoder feedback closes the velocity loop in motor control and provides critical state information for reinforcement-learning agents that rely on accurate motion estimation [9].

Figure 1.8 - Dynamixel Motors

Raspberry Pi 3

The Raspberry Pi 3 Model B serves as the primary onboard computer for TurtleBot3, running ROS nodes and high-level algorithms. It features a 1.2 GHz 64-bit quad-core ARM Cortex-A53 CPU, 1 GB of RAM, Ethernet, and built-in Wi-Fi/Bluetooth connectivity. This computing platform strikes a balance between performance, power consumption, and cost, allowing real-time sensor processing (e.g., LiDAR scanning at 5.5 Hz) and deep-learning inference for policy evaluation. Its Linux-based environment simplifies deployment of Python and C++ reinforcement-learning libraries within ROS ecosystems [10].

Figure 1.9 - Raspberry Pi 3 board

OpenCR Board

The OpenCR 1.0 (Open-source Control Module for ROS) is the dedicated microcontroller board responsible for low-level motor and sensor interfacing. Based on a 32-bit ARM Cortex-M7 MCU running at 200 MHz, it integrates power management, IMU (MPU-9250), and Dynamixel bus drivers. OpenCR provides API support to ROS through the ros_opencr package, allowing seamless bridging of high-level ROS topics (e.g., /cmd_vel) to Dynamixel motor commands and IMU readings. Its modular design and open-source firmware facilitate real-time control loops and sensor polling essential for reinforcement-learning experiments in both simulation and hardware-in-the-loop setups [11].

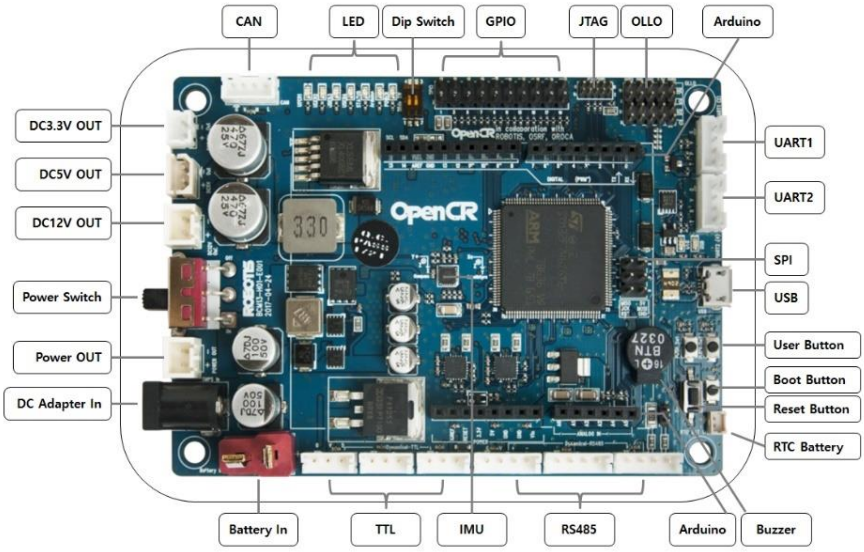


Figure 1.10 - OpenCR Board

1.4 Pioneer 3-DX components

The Pioneer 3-DX is also a widely used differential drive mobile robot in research. It was chosen as an additional platform for our work due to its availability in the LTSS research laboratory, with the intention of using it for experimental validation in future studies. It includes the following components.

Drive System (Motors & Wheels)

The P3-DX uses a differential drive system composed of two independently controlled wheels powered by DC motors. This setup allows the robot to maneuver by varying the speed and direction of each wheel, enabling forward, backward, and turning movements.

Onboard Computer

The P3-DX is typically equipped with an onboard microcontroller or embedded computer (such as an ARIA/ARIA-compatible controller) that handles motor control, sensor data processing, and communication with external systems via serial or USB [12].

Power Supply

The robot is powered by a rechargeable 12V lead-acid battery, which provides sufficient energy for extended indoor operation. The battery powers both the motors and the onboard electronics.

Sonar Sensors

The P3-DX is equipped with an array of ultrasonic rangefinders (sonar sensors) placed around the front and rear of the robot. These sensors detect obstacles and help in mapping and collision avoidance during autonomous navigation.

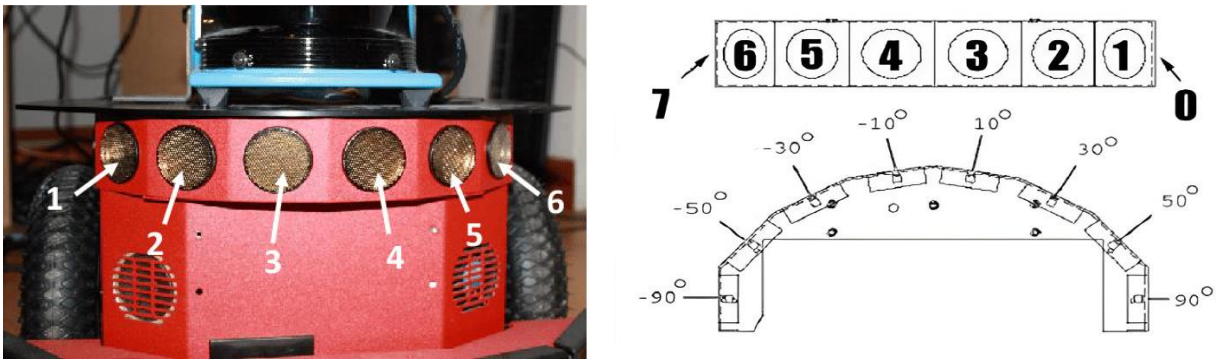


Figure 1.11 - Pioneer 3-DX mobile robot and arrangement of sonar sensors

Other Sensors and Accessories

The platform supports various add-on sensors, including LiDAR, cameras, and GPS modules, making it suitable for research in SLAM, autonomous navigation, Encoders, and sensor fusion.

1.5 Conclusion

Chapter 1 has surveyed the major classes of mobile robots, the prevalence of differentially driven wheeled robots specifically the Pioneer 3-DX and TurtleBot3 owing to their mechanical simplicity and suitability for indoor RL experiments. The chapter also introduced non-holonomic constraints and various drive configurations (single-wheel, differential, and Ackerman steering), underscoring trade-offs between maneuverability and control complexity. Finally, we detailed the hardware and sensor suites of TurtleBot3 and Pioneer 3-DX.

Chapter 2: Fundamentals of Reinforcement Learning (RL)

2.1 Introduction

Chapter 2 lays the theoretical foundation of reinforcement learning (RL) by first defining the agent–environment interaction loop and contrasting RL with supervised learning. We then classify RL algorithms into model-based and model-free approaches, introduce core concepts such as policies, value functions, and the exploration–exploitation trade-off, and detail the Q-learning algorithm with illustrative examples. Building on tabular methods, we present the transition to deep reinforcement learning, covering Deep Q-Networks (DQN), Deep Deterministic Policy Gradient (DDPG), and Twin Delayed DDPG (TD3), including their architectures, training mechanisms (replay buffer, target networks), and relevant hyperparameters. Throughout, we emphasize how these methods enable agents to learn optimal behaviors in high-dimensional and continuous action spaces

2.2. Reinforcement Learning (RL)

Reinforcement Learning is a branch of machine learning where an agent learns to make decisions by interacting with an environment. Through trial and error, the agent takes actions in different states of the environment and receives feedback in the form of rewards. The goal of the agent is to learn an optimal policy a strategy that dictates which action to take in each state to maximize the cumulative reward over time.

In contrast supervised learning, which relies on labelled data, RL focuses on learning from experience without explicit instruction. It is particularly well suited for sequential decision-making problems, such as robotics and control tasks, where the consequences of actions may unfold over time. RL methods are categorized into value based, policy based, and actor-critic approaches, with deep reinforcement learning combining RL with deep neural networks to handle high dimensional state and action spaces.

2.2.1 Agent-environment interaction

Reinforcement Learning refers to the dynamic process where an agent interacts with its environment in a loop to learn **optimal behavior.** At each time step, the agent observes the current state of the environment, takes an action based on a decision-making policy, and in return, the environment provides a new state and a reward signal as shown in figure 2.1. This feedback allows the agent to adjust its actions over time with the goal of maximizing cumulative rewards. This continuous cycle constitutes the basis of learning in reinforcement-based systems [13]

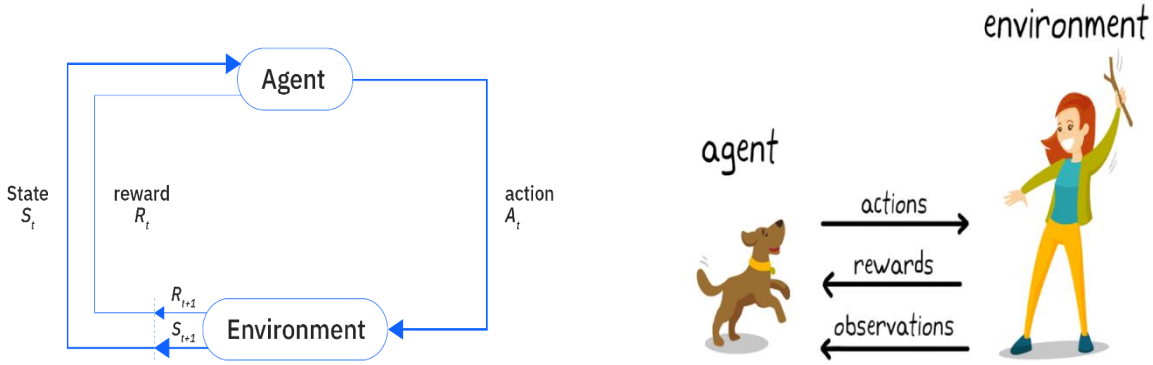


Figure 2.1 - Illustration of Agent-Environment Interaction in Reinforcement Learning

2.2.2 Reinforcement Learning Algorithms: Model-Based and Model-Free Approach

Reinforcement Learning (RL) algorithms can generally be classified into two main approaches as we can see in figure 2.2.

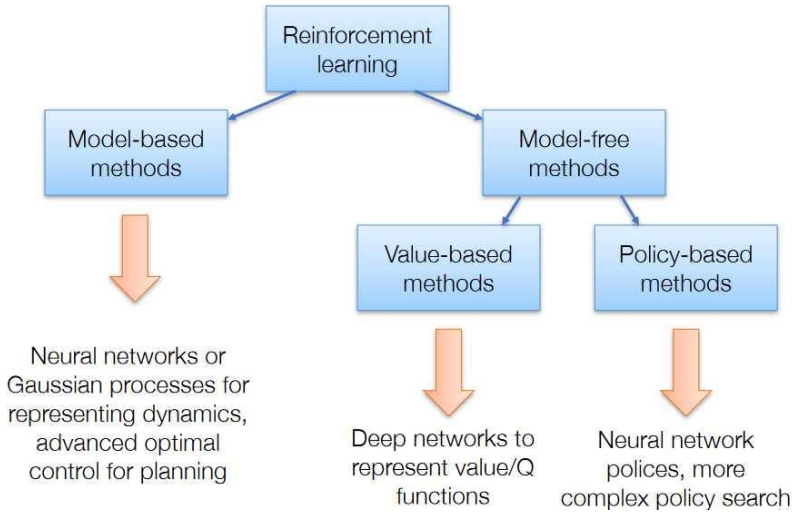


Figure 2.2 - Reinforcement Learning Algorithms

Model-Based Methods

- A model-based method builds a model of the environment, which predicts the next state and reward given a current state and action.
- The agent learns or is given a model to simulate future states.
- Model errors can lead to poor decisions.
- Learning an accurate model in complex environments is difficult.

Model-Free Methods

- Learn directly from experience without an explicit model. It learns the policy or value function directly from interaction with the environment.
- Subdivided into:
    - Value-Based Methods: Learn a value function to guide decisions (e.g., Q-Learning, Deep Q-Networks (DQN), DDPG).
    - Policy-Based Methods: Learn the policy directly (e.g., REINFORCE) [14].

2.3 Fundamental Concepts in RL

Action-Value Function $Q(s, a)$

The action-value function, often denoted as $Q(s, a)$, is a fundamental concept in reinforcement learning. It represents the expected return (cumulative reward) an agent can obtain by taking an action a in a given state s, and thereafter following a specific policy π.

What is a Policy (π)?

In RL, a policy (denoted by π) is the strategy or rule that the agent follows to choose actions based on the current state.

There are two main types of policies:

Deterministic Policy

A deterministic policy is a strategy in which the agent always chooses the same action for a given state, there is no randomness involved. It is defined as a function:

$$\pi(s) = a \tag{2.1}$$

meaning that in state s, the policy will always return the same action a (figure 2.3). Deterministic policies are commonly used in continuous action spaces (like in DDPG), where directly sampling from a probability distribution can be inefficient or unnecessary. These policies are simpler and faster to evaluate since they make fixed decisions, but they might not explore the environment as thoroughly without added exploration mechanisms.

Figure 2.3 - Deterministic Policy

Stochastic Policy

A stochastic policy, on the other hand, assigns a probability distribution over actions for each state. It is defined as:

$$\pi(a|s) \tag{2.2}$$

where the agent chooses actions based on probabilities. For example, in a certain state s, the agent might choose action a_1 with 60% probability and action a_2 with 40% probability. This type of policy is useful in environments with uncertainty or where exploration is crucial. Stochastic policies are commonly used in algorithms like A3C, which benefit from the randomness in action selection to explore more diverse strategies during training [14].

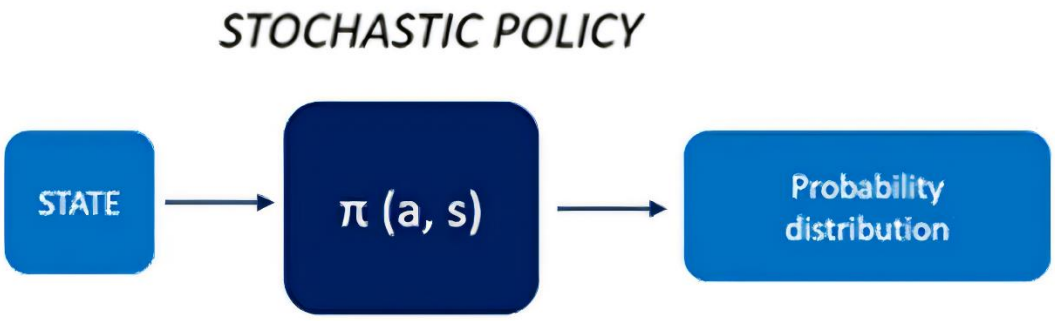


Figure 2.4 - Stochastic Policy

Exploration vs. Exploitation

- **Exploration**: Trying new actions to discover their potential rewards.
- **Exploitation**: Choosing the action that has the highest known reward so far.
- Q-learning balances these through strategies like **ε-greedy**, which picks a random action with probability ε and the best-known action with probability 1–ε.

2.4 Q-Learning

Q-Learning is a widely-used, value-based reinforcement learning (RL) algorithm as we said before. It aims to find the optimal policy for decision-making tasks. It enables an agent to learn the best action to take in a given state by interacting with an environment and receiving feedback in the form of rewards. The agent maintains a Q-value table (also called a Q-table), where each entry $Q(s, a)$ estimates the expected cumulative reward for acting a in state s, and then following the optimal policy thereafter.

The update rule for Q-values in Q-Learning is defined as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[R + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{2.3}$$

Where:

$Q(s, a)$ is the current Q-value for state s and action a.

$\alpha \in [0,1]$ is the learning rate, determining how much new information overrides the old.

R: Immediate reward received after taking action a in state s.

γ: Discount factor, $0 \leq \gamma < 1$, which weighs the importance of future rewards.

$\max_{a'} Q(s', a')$ is the maximum predicted Q-value for the next state s', across all possible actions a'.

Through iterative updates, Q-Learning helps the agent converge towards an optimal policy that maximizes the expected long-term reward. One of the key advantages of Q-Learning is that it is off-policy, meaning it learns the value of the optimal policy independently of the agent's current actions.

This method is particularly effective in discrete state and action spaces and forms the basis for more advanced algorithms like Deep Q-Networks (DQN), which use neural networks to estimate Q-values in high-dimensional environments [15].

A Q-Learning example

In this example a rat must navigate through a maze to reach a designated goal point, the big pile of cheese. The maze contains hidden bombs, and the rat can move only one tile at a time. Stepping on a bomb results in failure, while reaching the goal in the shortest number of steps defines success. The rat must learn an optimal path that avoids the bombs and minimizes the number of steps.

Figure 2.5 – The rat environment

The reward system goes like this:

- **+0:** Going to a state with no cheese (the yellow lightning icon) in it.
- **+1:** Going to a state with a small cheese in it.
- **+10:** Going to the state with the big pile of cheese.
- **-10:** Going to the state with the bomb and thus dying.
- **+0** If we take more than five steps.

The rat has four possible actions at each non-edge tile: (↑, ↓, →, or ←) (move up, down, right, or left). The state is the position of the rat in the maze.

To train our agent to have an optimal policy (a policy that goes right, right, down), we will use the Q-Learning algorithm.

- The episode ends if we step on a bomb, eat the big pile of cheese, or if we take more than 5 steps for example.
- The learning rate is α= 0.1.
- The discount rate (gamma) is γ= 0.99.

Step 1: Initialize the Q-table

Initialize Q arbitrarily (e.g., $Q(s, a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q($ terminal-state, $\cdot) = 0$) as shown in Table 2.1.

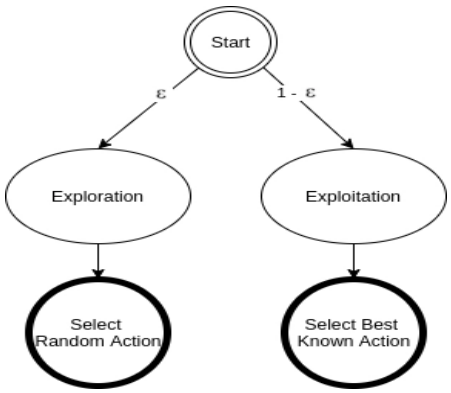
So, for now, our Q-table is useless we need to train our Q-function using the Q-Learning algorithm. Let's do it for 2 training timesteps.

	←	→	↑	↓
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

Table 2.1 - Initialization of the Q-table

Training timestep 1:

Step 2: Choose an action using the Epsilon Greedy Strategy:



Initially, because epsilon is big (ε = 1.0), our rat knows nothing about the environment (maze), so it will choose a random action (exploration), say right:

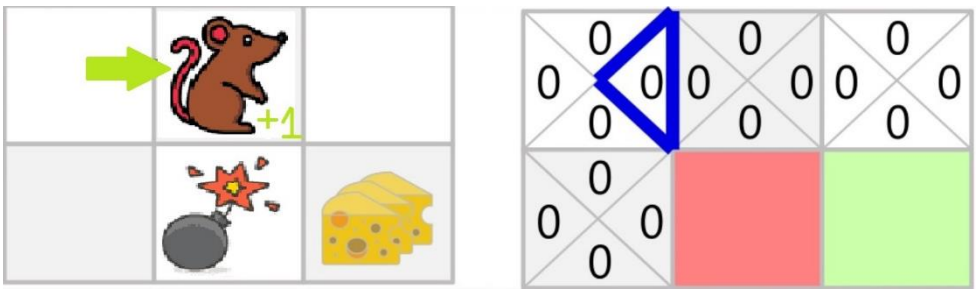


Figure 2.6 - Random action taken by the rat

Step 3: Perform action $_t$, get reward R_{t+1} and go to new state S_{t+1}:

By going right, the rat gets a small cheese, so $R_{t+1} = +1$ and he is in a new state as shown in the

figure 2.6 above (Take action A_t and observe R_{t+1} and S_{t+1}).

Step 4: Update Q (S_t , A_t):

We can now update Q (S_t , A_t) using the formula blow from the Algorithm:

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

New Q-value estimation | Former Q-value estimation | Learning Rate | Immediate Reward | Discounted Estimate optimal Q-value of next state | Former Q-value estimation

TD Target

TD Error

$$Q(Initial\ state, Right) = 0 + 0.1 * [1 + 0.99 * 0 - 0]$$

$$Q(Initial\ state, Right) = 0.1$$

Then the Q table becomes as follow:

	←	→	↑	↓
	0	0.1	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

Table 2.2 - the Q value updated

Training timestep 2:

Step 2: Choose an action using the Epsilon Greedy Strategy

We take a random action again, since epsilon=0.99 is big (Notice we decay epsilon a little bit because, as the training progress, we want less and less exploration).

We took the action 'down'. This is not a good action since it leads to the bomb.

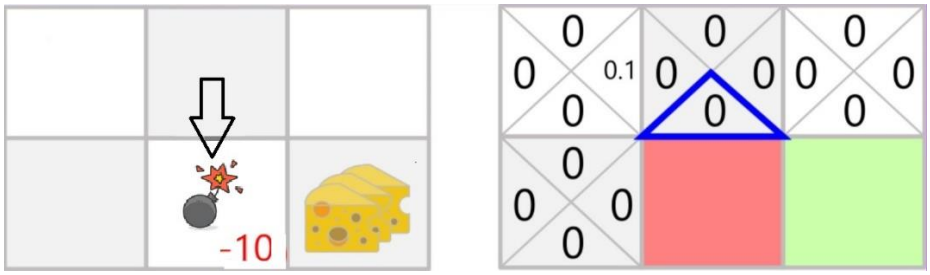


Figure 2.7 - We took a random action (exploration)

Step 3: Perform action A_t, get R_{t+1} and S_{t+1}:

Since the rat stepped on a bomb, it receives a negative reward of $R_{t+1} = -10$ and the episode terminates, as the rat is considered to have failed. the agent takes an action A_t, then observes the resulting reward R_{t+1} and transitions to the next state S_{t+1}, which in this case is a terminal state due to the failure.

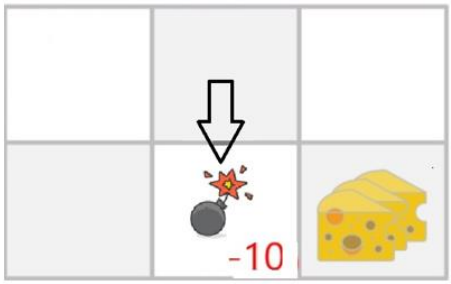


Figure 2.8 - Action penalty taken by a rat

Step 4: Update $Q(S_t, A_t)$:

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

$$Q(State\ 2, Down) = 0 + 0.1 * [-10 + 0.99 * 0 - 0]$$

$$Q(State\ 2, Down) = -1$$

The updated Q-Table becomes:

	←	→	↑	↓
	0	0.1	0	0
	0	0	0	-1
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

Table 2.3 - the Q value updated after penalty

Because the rat is dead, we start a new episode. But what we see here is that, with two explorations steps, our agent became smarter.

As the agent continues to explore and exploit the environment while updating the Q-values using the Q-learning update rule $Q(S_t, A_t)$, the Q-table progressively provides a more accurate approximation of the expected cumulative rewards. Over time, with sufficient interactions

and updates, the Q-table converges toward an estimate of the optimal Q-function, which represents the best possible action to take in each state to maximize the cumulative rewards.

	←	→	↑	↓
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

Training

	←	→	↑	↓
	0	10.8	0	0
	0	9.9	0	-10
	0	0	0	10
	0	-10	0	0
	0	0	0	0
	0	0	0	0

Table 2.4 - the Q table after training

The result:

After training, the Q-table guides the agent to follow the **optimal policy** (Right, Right, and Down), now he agent learns to avoid dangers and reach the goal (cheese).

The Algorithm 1 below summarizes how Q-learning works:

Algorithm 1: Q-learning Algorithm(off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$;
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ arbitrarily, except $Q(\text{terminal}, \cdot) = 0$; ← Step 1
foreach *episode* **do**
  Initialize S;
  foreach *step of episode* **do**
    Choose A from S using policy derived from Q (e.g., $\varepsilon - greedy$); Step2
    Take action A, observe R, S'; Step 3
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$; Step4
    $S \leftarrow S'$;
  end
  until S is terminal
end

2.5 Deep Reinforcement Learning

Traditional reinforcement learning (RL) algorithms often experience slow convergence, which can limit their effectiveness, particularly in complex environments. To address these challenges, deep reinforcement learning (DRL) combines RL principles with deep learning techniques. This integration enhances the ability to manage large-scale, dynamic systems more efficiently. In this section, DRL-based methods are introduced, as they form the basis for the algorithms developed and employed in this thesis [16].

2.5.1 Deep Q Network (DQN)

A Deep Q-Network is a type of reinforcement learning algorithm that integrates Q-learning with deep neural networks to estimate optimal action values (Q-values). It is designed to handle environments with high-dimensional state spaces, where traditional Q-learning methods become impractical. Instead of maintaining a Q-table, DQN uses a neural network to approximate the Q-function, mapping states to action values. By doing so, it enables agents to learn effective policies directly from raw input data such as images or sensor readings. DQN is particularly useful in complex and dynamic environments where the state-action space is too large for conventional tabular approaches [17].

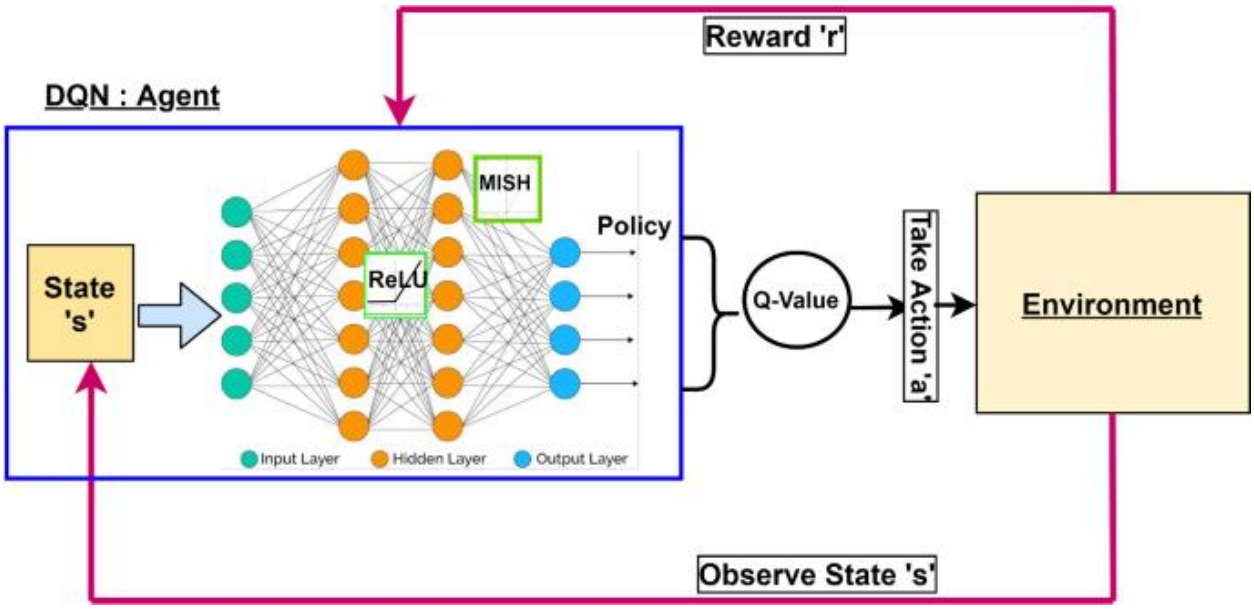


Figure 2.9 - Illustration of a DQN agent

As shown in (Figure 2.9), the Deep Q-Network (DQN) receives the current observed system states as its input. These input states are processed through several layers of a neural network, each associated with a set of parameters (denoted as θ), which are adjusted during training. Initially, these weights are assigned random values and are progressively refined through iterative updates to improve the network's predictions. A loss function is employed to evaluate the prediction error, and gradient descent is used to minimize this error. The DQN produces a Q-value for each possible action, corresponding to one output node per action the agent can choose from. The agent selects the next action based on the highest Q-value equation (2.4) giving the link between Q-values and Policy. Finding an optimal Q-values function leads to having an optimal policy. Ultimately, the objective of DQN is to learn optimal weight parameters from past experiences in order to make effective decisions.

$$\pi(s) = \arg max_{\mathrm{a}}\, Q\ (s, a) \tag{2.4}$$

DQN learns the optimal policy by using a deep neural network to estimate the Q-values, a replay buffer to store past experiences, and a target network to prevent overestimating Q-values. The agent uses an epsilon-greedy exploration strategy during training and selects the action with the highest Q-value during testing.

To clearly illustrate DQN in action, we often refer to the cart-pole balancing task as is shown if figure 2.10. The system under consideration is characterized by four state variables: the cart position x, the cart velocity $\dot{x}$, the pole angle θ, and the pole angular velocity $\dot{\theta}$. The control mechanism involves two discrete actions: applying a force to move the cart to the left (action $A = 0$) or to the right (action $A = 1$). The primary control objective is to determine a sequence of actions that maintains the pole in an upright (vertical) position.

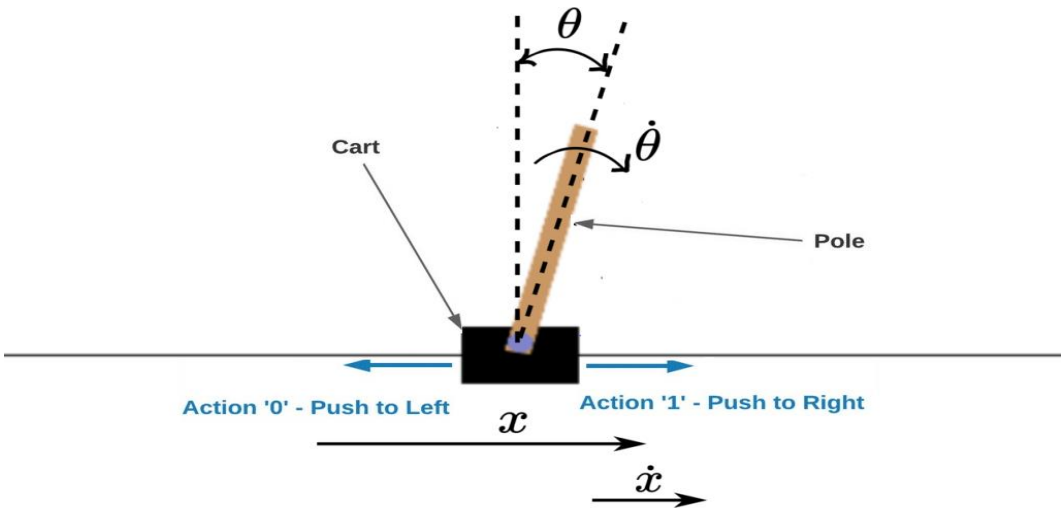


Figure 2.10 - Cart-Pole balancing problem

To achieve this objective, it is necessary to approximate a function that maps the state vector $S = [x, \dot{x}, \theta, \dot{\theta}]^T$ to action-value functions corresponding to each possible action. In other words, the goal is to learn a function that can estimate the value of taking each action in a given state.

This mapping is represented as:

$$\begin{bmatrix} Q(S, A = 0) \\ Q(S, A = 1) \end{bmatrix} = F(S) = F\left(\begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}\right) \tag{2.5}$$

Here, F is the function (potentially nonlinear) to be approximated through learning. It takes the state vector S as input and outputs the estimated action-value functions for each action. These values guide the decision-making process in selecting the optimal action either $A = 0$ (move left) or $A = 1$ (move right) to stabilize the pole in an upright position.

The main idea of the deep Q networks is to use a neural network as the mapping F. This is shown in the figure below. The objective is to train this neural network so that it can accurately approximate the desired mapping. Once the training process is complete, the network's predictions will be utilized in conjunction with a greedy policy to select the optimal control actions at each state.

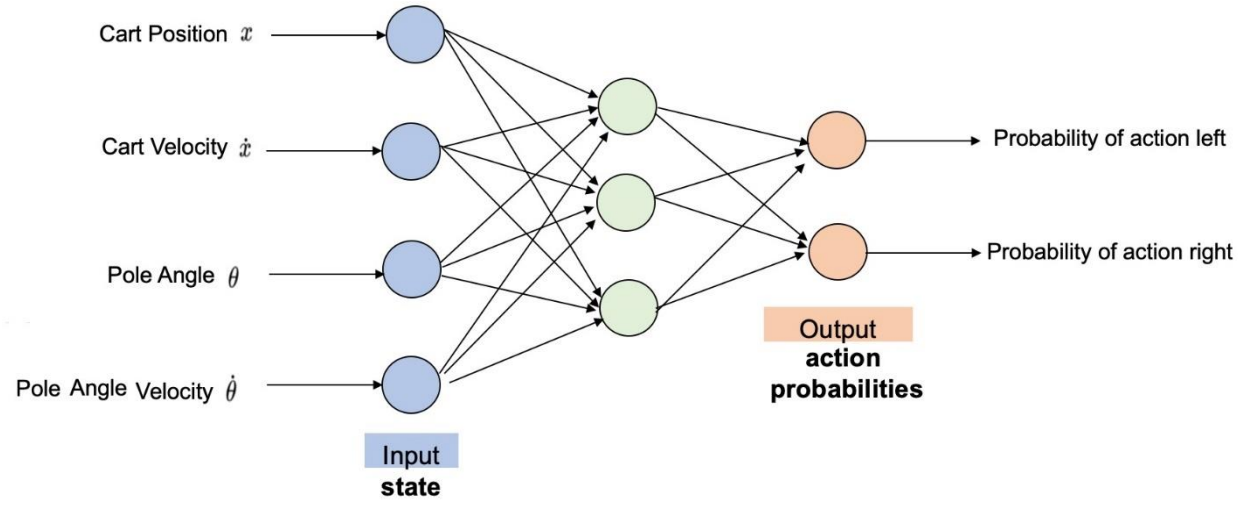


Figure 2.11 - Neural network diagram

To train the neural network effectively, it is essential to use the replay buffer (also referred to as experience replay). This mechanism plays a crucial role in stabilizing and improving the learning process.

Consider a sequence of states and actions from the cart-pole system, as illustrated in Figure 2.11. Let the initial state of episode 1 be denoted as S_{11} . At this point, a control action A_{11} is selected using a certain policy and applied to the environment. As a result, a reward R_{11} is received, and the system transitions to a new state S_{12} . This procedure is then repeated: an action A_{12} is computed based on state S_{12} , applied to the environment, leading to a reward R_{12} and a transition to state S_{13}.

This cycle continues throughout the episode until a terminal state is reached. Let the total number of time steps in episode 1 be N_1, such that the terminal state is denoted by S_{1N1} .

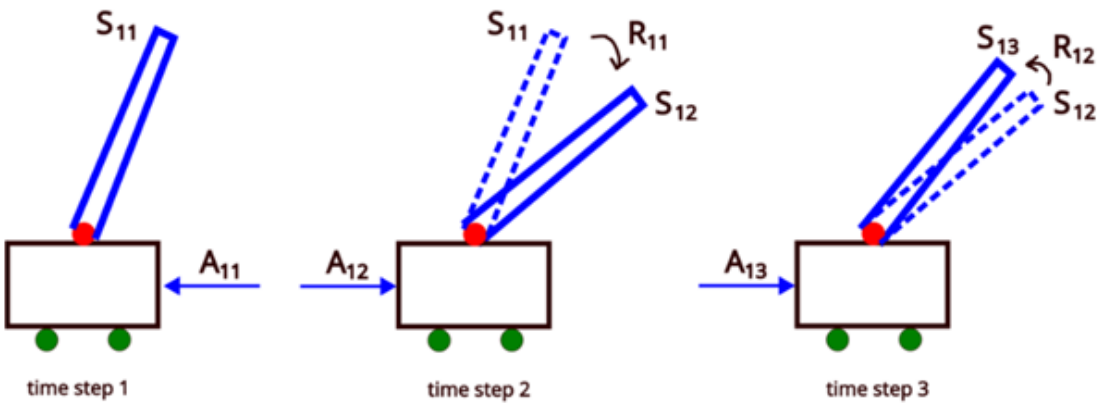


Figure 2.12 - the state transitions.

Subsequently, the second episode begins with an initial state denoted by S_{21} . An action A_{21} is selected and applied to the environment, resulting in a reward R_{21} and a transition to the next state S_{22} . This process is then iteratively repeated at each subsequent state. At state S_{22} , an action A_{22} is taken, yielding a reward R_{22} and a new state S_{23}, and so on, until the terminal state S_{2N2} is reached, where N_2 denotes the total number of time steps in episode 2.

The same procedure is applied across all subsequent episodes (third, fourth, etc.), each producing a sequence of states, actions, rewards, and next states. All these transitions are systematically stored in the Replay Buffer, as illustrated in the accompanying figure 2.13. This buffer serves as a memory bank that allows the learning algorithm to sample past experiences during the training process, thereby enhancing learning efficiency and stability [18].

Replay Buffer and Random Sampling for Training

To implement the replay buffer, we store the current state, the action taken in the current state, the reward received, the next state, and the terminal state indicator (denoted as "IsTerminal") in a tuple. The boolean variable "IsTerminal" indicates whether the next state is a terminal state (True) or not (FALSE). We store these tuples sequentially, adding each new tuple on top of the previous ones, until the end of the episode. Afterward, we repeat this process for a specified number of episodes. This method allows us to collect states, actions, and rewards across various episodes.

The queue data structure is used to manage the replay buffer. For instance, assume that the buffer contains N cells for storing tuples. As we add a new tuple, the oldest tuple in the buffer is removed, ensuring that the buffer always retains the most recent N tuples.

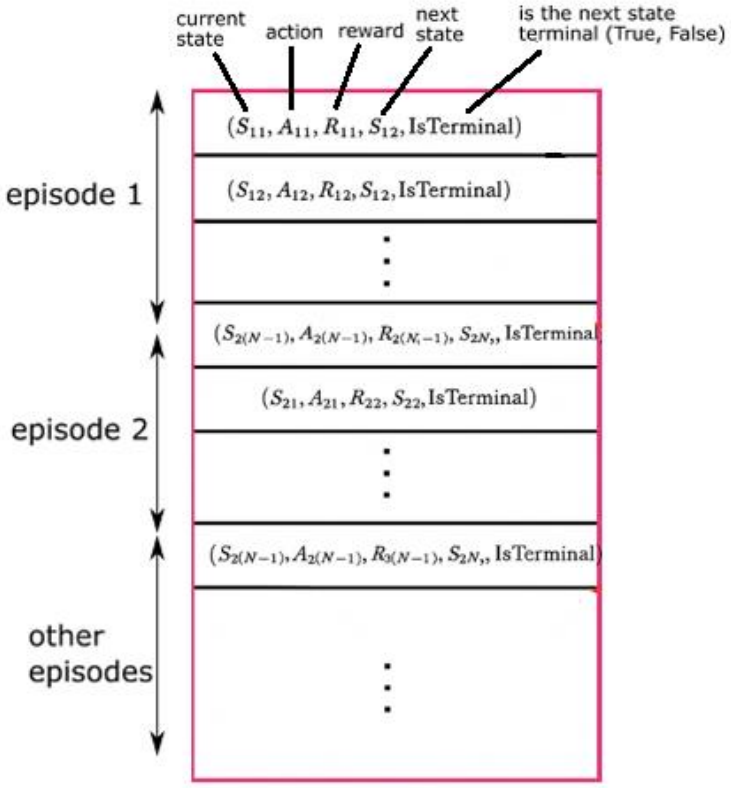


Figure 2.13 - The replay buffer

When training the network, instead of using the entire replay buffer (which would involve processing the full batch of data), we randomly sample tuples from the buffer to form a smaller batch used for training. This process is illustrated in the figure 2.14 below.

- **Random Sampling**: We randomly choose tuples from the replay buffer to form a batch for training. The states in these sampled tuples may be highly correlated, and therefore, we avoid using consecutive tuples to minimize correlations, leading to a more efficient learning process.

This approach speeds up the training by reducing the correlation between consecutive tuples in the buffer. As a result, we obtain more diverse training data that accelerates the model's learning process.

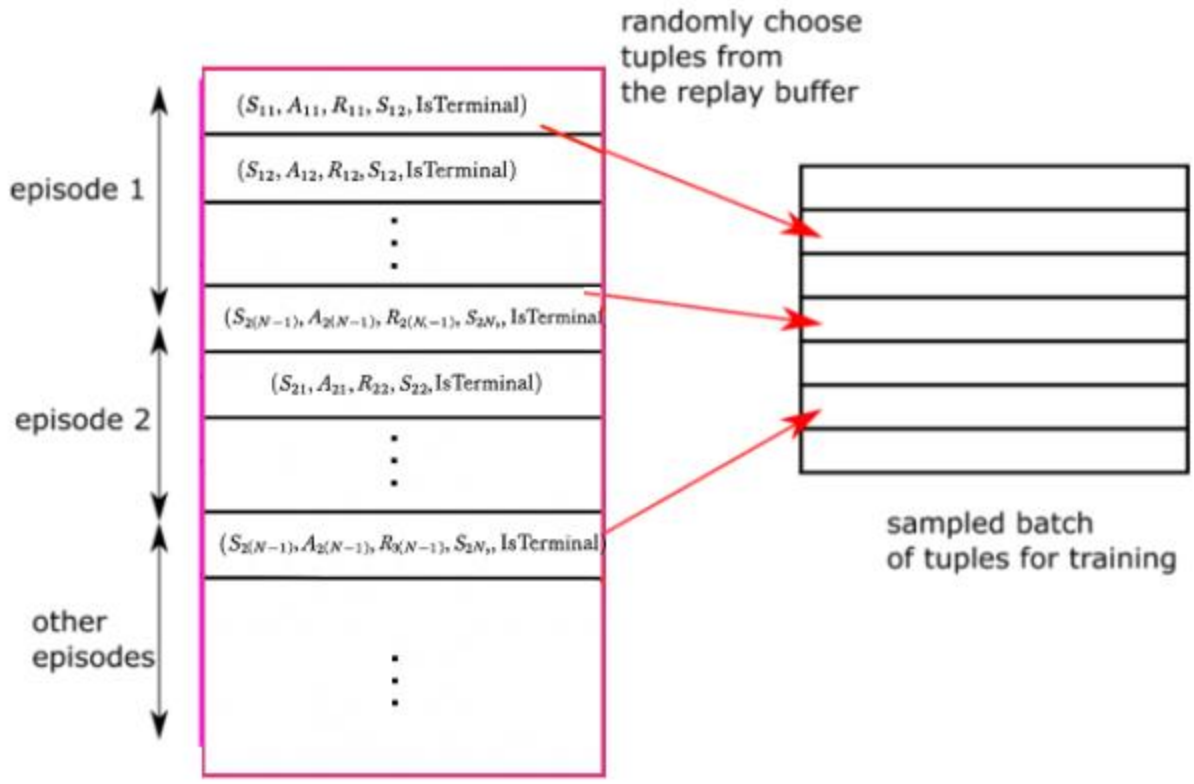


Figure 2.14 - Random sampling from the replay buffer to form a batch of data for training

Online Network and Target Network

- **The Online Network**, also referred to as the main network, is the network we continuously update during the training process. It is responsible for predicting the action-value functions and, once the training is completed, forming the greedy policy. The predictions made by this network are represented by: $Q(S, A, \theta)$ (Online network action value prediction), where θ represents the vector of parameters of the neural network. The training algorithm updates θ continuously during the training process.
- **The Target Network** is employed to compute the target output samples used for training the network. This network produces predictions based on both the network's current outputs and the rewards obtained, forming the target values for the training process. The

parameters of this network are updated as well, but less frequently than the online network. For example, we may update the target network every 100-time steps (this number can be larger depending on the setup). The predictions of the target network are denoted as: $Q(S, A, \tilde{\theta})$ (Target network action value prediction), where $\tilde{\theta}$ represents the parameters of the target network.

At the beginning of the training process, we initialize the target network's parameters to match those of the online network: $\tilde{\theta} \leftarrow \theta$. After a certain number of time steps (for example, after 100 or more-time steps), we update the target network by setting $\tilde{\theta} \leftarrow \theta$, essentially copying the parameters from the online network to the target network.

With an understanding of both the online and target networks, we now define the cost function for the Q-learning model. We aim to minimize the following cost function:

$$L = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - Q(S_i, A_i, \theta)\right)^2 \tag{2.6}$$

Where y_i is given by the following equation if the next state S_i' is NOT the terminal state

$$y_i = R_i + \gamma Q_{max}\left(S_i', \tilde{\theta}\right) \tag{2.7}$$

and

$$Q_{max}(S_i') = \max_A Q\left(S_i', A, \tilde{\theta}\right) \tag{2.8}$$

and given by equation (2.8) if the next state S_i' is the terminal state:

$$y_i = R_i \tag{2.9}$$

Here, S_i' refers to the next state, and γ is the discount factor. The term y_i represents the target value computed based on the current reward R_i and the maximum predicted value from the next state S_i', as determined by the target network.

To effectively implement the learning algorithm, it is important to define key components of the training process. Let N denote the total number of tuples in the mini-batch sampled from the replay buffer. Each tuple is of the form: (S_i, A_i, R_i, S_i', IsTerminal), where:

- S_i is the current state.

- A_i is the action taken in S_i.
- R_i is the reward received.
- S_i' is the resulting next state.

The goal is to compute a target value y_i, which serves as the training signal for the Q-network.

$\tilde{\theta}$ in equation (2.7) and (2.8) represents the parameters of the target network. To evaluate Q_{max}, the next state S_i' is passed through the target network to obtain the action-value predictions Q (S_i', A =0, $\tilde{\theta}$) and Q (S_i', A =1, $\tilde{\theta}$) The maximum of these two values is taken as Q_{max} (S_i', $\tilde{\theta}$) which contributes to computing the target y_i [18].

Summary of the Deep Q-Learning Algorithm

***Preliminary Steps*:**

Before starting the training process, we perform the following initialization steps:

- Define the architecture of the online Q-network. The target network must share the same structure.
- Initialize the parameter vector θ of the online network.
- Copy the parameters of the online network to the target network, i.e., set $\tilde{\theta} \leftarrow \theta$.

***Training Loop Over Episodes*:**

STEP 1: Initialize the environment and observe the initial state. This resets the environment and sets the initial state S of the current episode.

STEP 2: For each time step within the episode, repeat the following until a terminal state is reached:

- **STEP 2.1**: In the current state, select an action A using the ε-greedy strategy. At the beginning of training, actions can be selected randomly to ensure sufficient
- exploration. The current online network is used to evaluate the Q-values for the greedy part of the policy.

- **STEP 2.2**: Apply the selected action A to the environment and observe the resulting reward R and the next state S'. Store the tuple (S, A, R, S', IsTerminal) in the Replay Buffer.
- **STEP 2.3**: If the replay buffer is not yet full (i.e., contains fewer elements than a predefined maximum), skip this step and proceed to **STEP 2.4**. Otherwise:
    - Sample a mini-batch of experience tuples from the replay buffer.
    - Create samples of input and output data necessary to form the cost function (2.6).
    - Train the online network on this batch.
    - Periodically (e.g., every fixed number of steps), update the target network parameters by copying them from the online network: $\tilde{\theta} \leftarrow \theta$.
- **STEP 2.4**: Set the current state to the next state $S \leftarrow S'$ and repeat the loop starting from **STEP 2.1**. If the terminal state is reached, return to **STEP 1** for a new episode.

Once the training is complete and the online Q-network is sufficiently accurate, it can be used to derive an optimal policy. During deployment or evaluation, the agent selects actions by passing the current state through the trained Q-network and choosing the action that maximizes the predicted Q-value this constitutes the greedy policy.

2.5.2 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is a model-free, off-policy actor–critic algorithm (see figure 2.15) specifically designed to operate in continuous action spaces. DDPG merges key ideas from both Deep Q-Networks (DQN) and Deterministic Policy Gradients (DPG) to extend reinforcement learning beyond discrete action environments. Originally proposed by Lillicrap et al. (2015) [19], DDPG is well-suited for solving complex control tasks in domains

such as robotic manipulation, inverted pendulum stabilization, autonomous driving, and energy management systems.

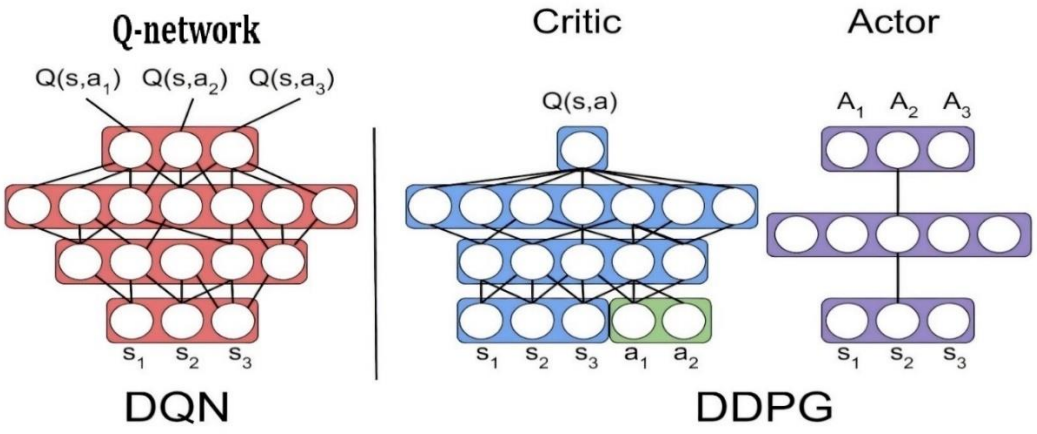


Figure 2.15 - Comparison Between DQN and DDPG Architectures: Critic-Only vs. Actor-Critic Framework

DQN and DDPG are both deep reinforcement-learning algorithms, but they differ fundamentally in how they represent policies and value functions. DQN is a *value-based* method designed for discrete action spaces, using a single Q-network to estimate the action-value function for all possible discrete actions. In contrast, DDPG is an *actor-critic* method for **continuous** action spaces: it uses an **actor network** (policy) to output continuous actions and a **critic network** to evaluate those actions by estimating $Q(s, a)$.

Figure 2.15 illustrates he comparison between DQN et DDPG: the DQN side shows one neural "Q-network" that takes state as input and outputs Q-values for each discrete action, while the DDPG shows two networks (actor and critic) and it outputs a **single Q-value** for the **specific action** a provided.

Also, DDPG lies in its use of two core techniques introduced by DQN: the **replay buffer** and the **target networks**.

The **target networks** consist of two additional neural networks:

- **Target critic network** Q′ with parameters $\theta^{Q'}$.
- **Target actor network** μ′ with parameters $\theta^{\mu'}$.

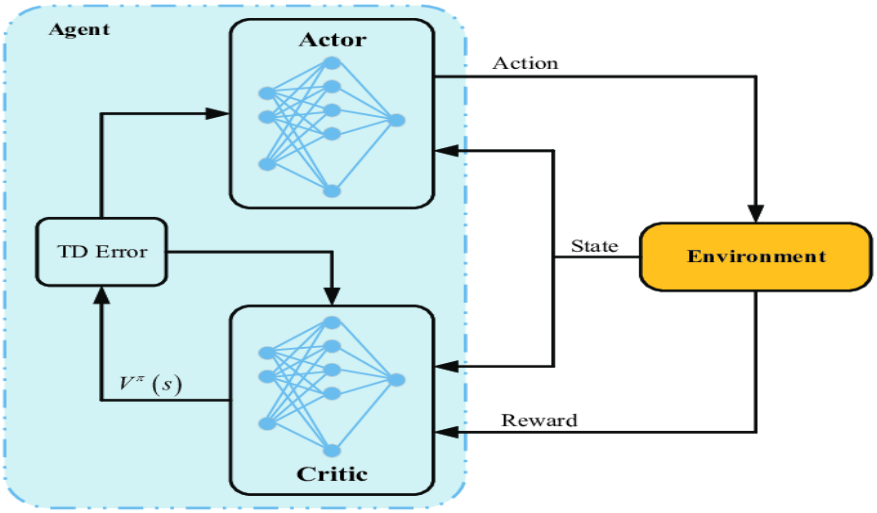


Figure 2.16 - The structure of deep actor-critic reinforcement learning

Actor-critic methods

the actor-critic framework, a reinforcement learning architecture that combines policy learning with value estimation. The **actor** selects actions according to a policy, while the **critic** evaluates those actions using a value function as shown in Figure 2.16. The critic computes the temporal-difference **(TD) error**, which guides updates to both the value function and the policy. The environment provides feedback through rewards, helping to improve future decisions.

a) Actor Network

In the Deep Deterministic Policy Gradient (DDPG) algorithm, the **Actor Network** is responsible for learning the **policy function** that maps each state to a specific action in **continuous action spaces**. Unlike stochastic policies that output action distributions [19], the actor in DDPG outputs a **deterministic action**:

$$a = \mu(s \mid \theta^{\mu}) \tag{2.8}$$

Where:

- s: the current state.
- a: the selected action.

- μ: the deterministic policy function.
- θ^{μ} : the actor network parameters.

The Actor Network defines what action to take in a given state based on its current understanding of the environment.

b) Exploration in DDPG

Since the actor outputs deterministic actions, DDPG adds external noise to encourage exploration during training. This is essential because a deterministic policy by itself doesn't explore the environment effectively.

$$a = \mu(s \mid \theta^{\mu}) + N \tag{2.9}$$

Where N is a noise term, commonly:

- **Ornstein-Uhlenbeck Process** (for temporally correlated noise),
- or **Gaussian Noise** (uncorrelated, simple to implement).

This noise is only added **during training**, not during evaluation or testing.

c) Critic Network

In DDPG, the Critic Network is responsible for learning the action-value function (Q-value), which estimates how good a given action is in a particular state under the current policy.

The critic is represented as a function:

$$Q(s, a; \theta^{Q}) \tag{2.10}$$

Where:

- θ^{Q}: the parameters (weights) of the Critic Network.
- $Q(s, a; \theta^{Q})$: The predicted Q-value (expected return) of taking action a in state , as estimated by the Critic.

The critic learns to evaluate actions suggested by the actor. This evaluation is then used to train the actor.

d) Hyperparameters

It is important to distinguish between the model parameters and model hyperparameters. The parameters are all the configuration variables that are internal to the model and whose value can be estimated from data (i.e. the weights of the neural networks).

The model hyperparameters are the ones that help with the learning process. They are set manually before starting the training and they have an important impact on the performance of the model, so it is important to choose appropriate values.

The hyperparameters used in the model are:

- **Learning rate:** it controls how much the model changes in response to the estimated error each time the weights are updated during the training, i.e. the learning rate controls the speed at which the model learns. A too small value may result in a very long training and if the learning rate is too large the model may not converge.

- **Batch size:** this hyperparameter indicates how many tuples are chosen randomly when training the model. It is used a mini-batch gradient descent configuration (batch size is set to more than one but less than the total number of tuples).

- **Replay buffer max length:** this hyperparameter tells the maximum capacity of samples that can be stored in the replay buffer.

- **Discount factor:** this hyperparameter is used when doing the transition to the next step. It determines how much importance is to be given to the immediate rewards and future rewards. Lower values place more emphasis on immediate rewards, while values close to 1 give more importance to future rewards.

- **Hidden layers**: they define the size and number of layers of the neural network. They represent the capacity to learn the optimal behaviour, so as more complex the

problem, the more nodes and layers will be needed. However, if they neural network is much larger than needed, it will lead to a problem of overfitting.

• **Initial collect steps:** it tells the number of steps done before starting the training with a random policy. It is done to collect some data that can be useful to get information about states that may not be reached during the training.

• **Maximum training steps:** it tells the maximum number of iterations that the simulation will last. This hyperparameter loses importance when studying the convergence time of the algorithm, as the simulation will usually stop before reaching this value [20].

2.5.3 Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm

TD3 is an off-policy actor-critic algorithm for continuous action spaces that builds on Deep Deterministic Policy Gradient (DDPG) by addressing key sources of function-approximation error. It introduces three core innovations clipped double Q-learning, delayed policy updates, and target policy smoothing that together yield substantially more stable and sample efficient learning across benchmark continuous control tasks [21].

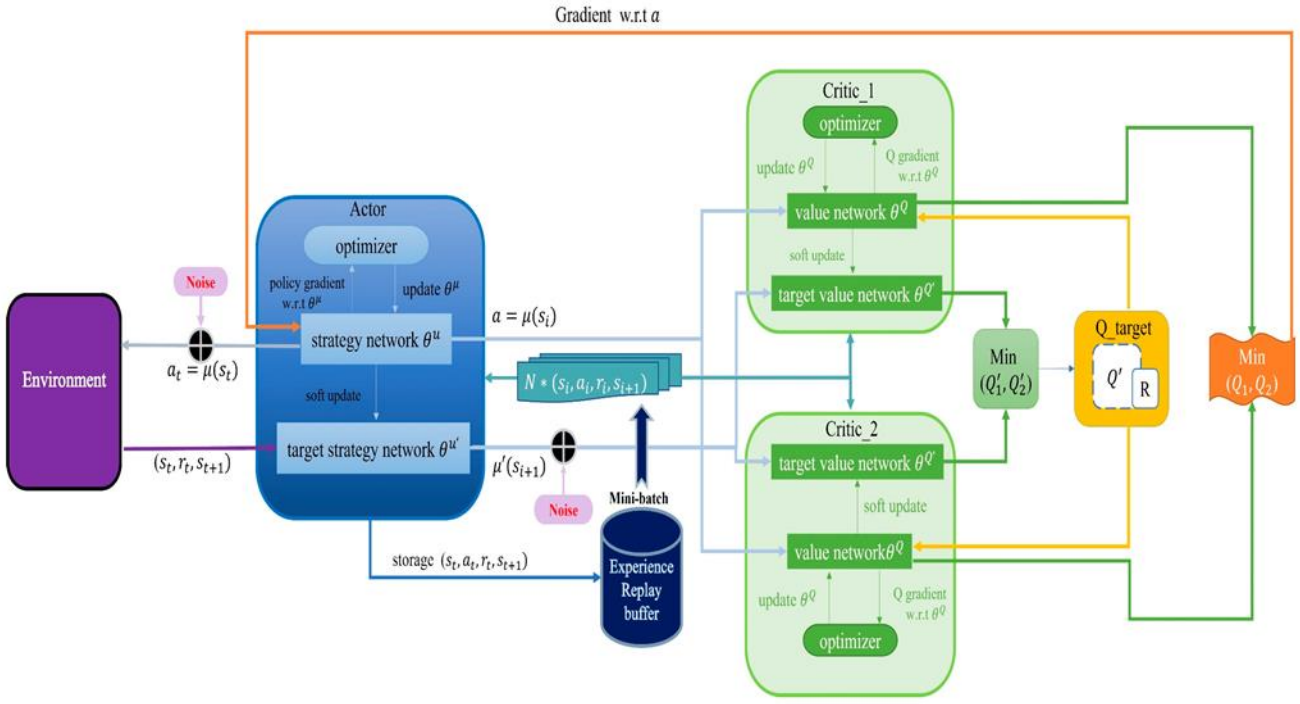


Figure 2.17 - Schematic diagram of the TD3 structure

As shown in the (figure 2.17) TD3 algorithm begins by instantiating an actor network that learns to map states to actions and two independent critic networks that each estimate the value of a given state-action pair, as well as corresponding “target” copies of each network. During training.

the agent interacts with the environment by selecting actions from the actor plus a small amount of random noise for exploration, and it stores each experience state, action, reward, next state in a replay buffer. At each training step, a batch of these experiences is sampled and the target actor network is used to generate “smoothed” next actions by adding clipped noise, which helps prevent the critics from overfitting to narrow peaks.

Both target critics then evaluate those actions, and the smaller of the two values is used to form a stable target for training the main critic networks. The critics are updated every step by minimizing the difference between their current estimates and this reference target. Meanwhile, the actor network and all target networks are updated less frequently only every few critic updates to further stabilize learning. When these delayed updates occur, the actor is adjusted to improve the critic’s estimated value of its chosen actions, and each target network is softly moved toward its main counterpart rather than being replaced outright. By combining twin critics to reduce overestimation, target policy smoothing to avoid reacting to outliers, and delayed updates to maintain stability.

2.5.4 The Differences between TD3 and DDPG

- Critic Architecture: DDPG uses a single critic, which tends to overestimate Q-values. TD3 uses two critics and selects the minimum Q-value during updates to reduce overestimation bias.
- Target Policy Smoothing: TD3 adds clipped noise to the target action before calculating target Q-values. This prevents the agent from exploiting Q-value spikes and encourages smoother learning. DDPG does not apply this technique.

- Policy Update Frequency: In DDPG, the actor and critic are updated at the same frequency. TD3 delays actor updates, typically updating the actor every two critic updates, which improves stability.
- Stability: TD3 is more stable and less sensitive to hyperparameter tuning than DDPG, which often suffers from divergence during training [22].

2.6 Conclusion

In this chapter, we established the fundamental principles of Reinforcement Learning (RL) and examined how they apply to robotic control tasks:

- We formalized the agent–environment paradigm, defining states, actions, transition dynamics, and reward signals, and contrasted model-based versus model-free approaches.
- Through a tabular Q-learning example, we saw how value estimates are iteratively updated and how exploration–exploitation trade-offs affect learning.
- We introduced Deep Q-Networks (DQN), demonstrating how neural function approximators can handle high-dimensional state spaces, and discussed stability challenges such as overestimation bias.
- To address continuous action domains, we explored actor-critic methods. We detailed the Deep Deterministic Policy Gradient (DDPG) algorithm its network architectures, replay buffer usage, and noise-based exploration and then presented Twin Delayed DDPG (TD3) enhancements that mitigate function approximation errors.

Overall, this chapter equips us with both the theoretical and practical insights needed to implement modern RL algorithms. These concepts directly inform our design choices in Chapter 3, where we build and train a DDPG and TD3 agent for mapless navigation in ROS/Gazebo.

Chapter 3: RL Implementation for Navigation and Motion Planning

3.1 Introduction

Autonomous navigation in unstructured environments is a core problem in mobile robotics, requiring the robot to perceive its surroundings, plan motions, and execute trajectories safely. Classical planners often depend on pre-built maps, limiting adaptability to novel scenarios. In this work, we develop an approach based on Deep Reinforcement Learning (DRL), enabling a differential-drive robot to learn continuous control policies directly from sensor observations. We implement both DDPG and TD3 in simulation, demonstrating their effectiveness in goal-reaching and obstacle avoidance tasks.

This chapter:

- describes the RL framework and agent–environment loop.
- details hardware and software configurations.
- presents the DDPG and TD3 algorithms, state-action design, reward, network architectures, and training protocol.
- introduces the Gazebo worlds used for training and testing.
- reports quantitative and qualitative results.
- summarizes and discusses the findings.

3.2 System Overview

In any reinforcement learning (RL) problem, there are two core components (see figure 3.1):

- The **Agent** (which learns the policy—in our case, the DDPG agent),
- The **Environment** (which simulates the world the agent interacts with).

In our case, the environment consists of the mobile robot equipped with its sensors, as well as the surrounding space in which it operates. The DDPG agent serves as the controller that

learns to navigate the robot through an unknown environment, avoiding obstacles and reaching a target position.

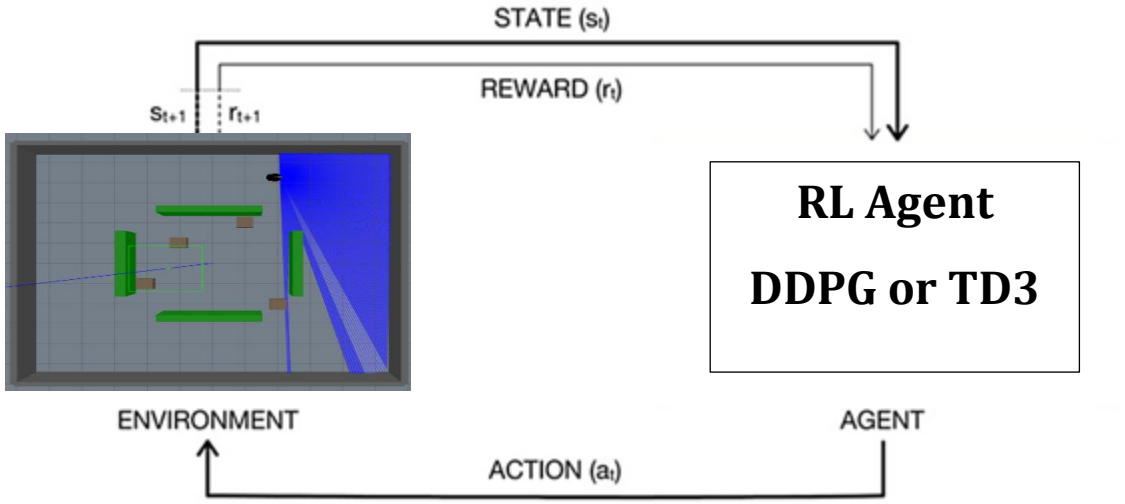


Figure 3.1 - Agent–Environment Interaction Loop

3.3 The Used Hardware and Software

3.3.1 Hardware Description

- In our work, we used the model of the simulated TurtleBot3 Burger differential-drive and the model of the Pioneer 3-DX robot considered for future real-world transfer.

- As sensors, the robot uses laser range data from an LDS-01, which has a field of view (FOV) of 180° and an angular resolution of 0.25°. Its scanning range is from 0.05 m to 3.5 m in real-world implementations; however, in our simulation model in Gazebo, we increased the maximum range to 10 m to accommodate larger environments. We used also wheel encoders for linear and angular velocity estimation.

3.3.2 Software Configuration

We conducted our work in Ubuntu 20.04 LTS. Our DDPG and TD3 networks were trained in the Gazebo 11 simulator and controlled by the Robot Operating System (ROS) commands. ROS

provides standard communication, tools, and libraries for developing and integrating robot software. RViz was used for point cloud and pose visualization, reducing rendering overhead. We implemented our approach using Python 3.8, leveraging a range of development libraries to support simulation, learning, and analysis. The libraries and tools include:

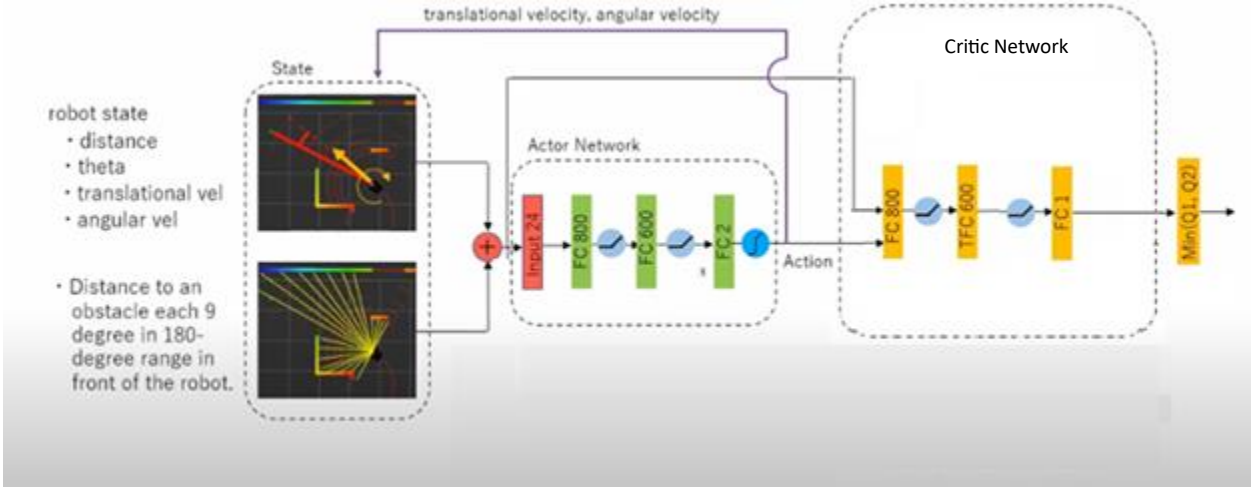
- Python 3.8 with rospy and custom Gym–ROS wrapper for integrating reinforcement learning with robotic control.
- NumPy for numerical processing.
- PyTorch 1.10 for building and training neural networks [4].
- TensorBoard for monitoring reward and loss curves.
- Additional supporting libraries for data processing, visualization, and simulation utilities.

3.4 DRL Implementation

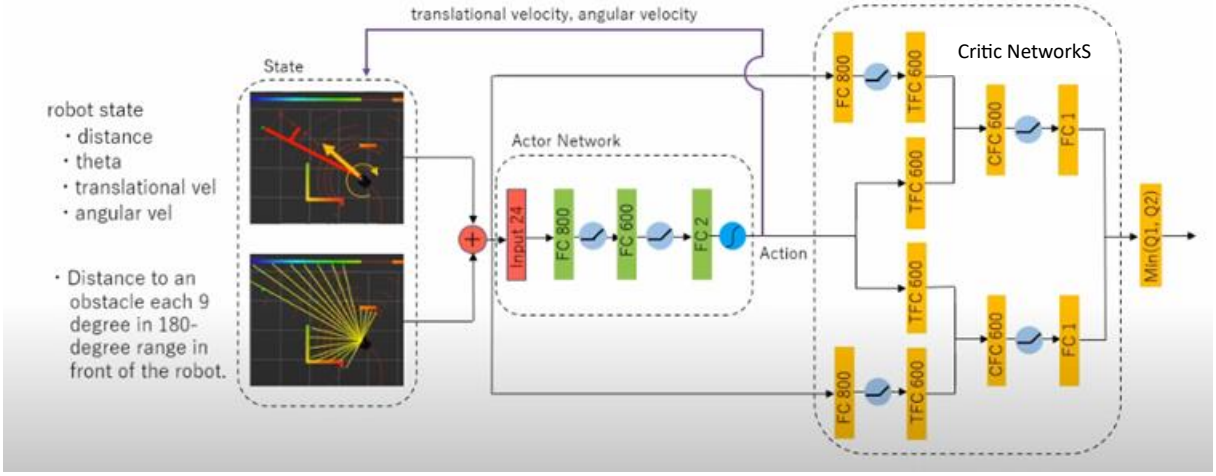
The Deep Reinforcement Learning DRL (DDPG or TD3) algorithm is specifically designed for continuous action spaces by learning a deterministic mapping from states to actions. Its actor–critic architecture stabilizes training, allowing the agent to refine smooth control signals such as velocities for real-time robotic tasks. Figures 3.2 below details the full system and neural networks architecture in the two cases: DDPG agent (figure 3.2.a) or TD3 agent (figure 3.2.b).

3.4.1 Overview of the DDPG and TD3 Algorithm

- **State ($\mathbf{s_t}$)**: 20-dim LiDAR sector averages (180° (degree Field of View) split into 20 bins), the distance and angle between the robot and the goal (2 inputs), Transitional v and angular ω velocities in the last time step (2 inputs); in total 24 inputs.
- **Action ($\mathbf{a_t}$)**: continuous linear velocity $v \in [-0.5, +0.5]$ m/s and angular velocity $\omega \in [-2.84, 2.84]$ rad/s.
- **Reward ($\mathbf{r_t}$)**: combination of goal progress, obstacle clearance, motion smoothness, and collision penalty.



(a)



(b)

Figure 3.2 - System Implementation. (a) shows DDPG implementation and (b) shows the TD3 implementation.

3.4.2 Reward Function

- **+100:** Reaching the goal (when distance to goal < 0.5 m).
- **–100:** Collision (when any LIDAR reading < 0.2 m).
- **0:** Episode timeout (when exceeding 500 steps without goal or collision).

- **+1× (v/0.5):** Forward motion reward, proportional to linear speed which encourages faster straight-line progress. Here we normalized the linear velocity by dividing it by v_{max}=0.5m/s.
- **−0.3 × |ω/2.84|:** Turning penalty, proportional to angular speed magnitude which discourages excessive rotation. Here we normalized the angular velocity by dividing it by ω_{max}=2.84rad/s.

3.4.3 Network Architectures & Exploration

As shown in Figure 3.2, after 3 fully-connected neural network layers, the input vector (State of size:24 inputs) is transferred to the linear and angular velocity commands of the mobile robot via the Actor network. To constrain the range of the linear and angular velocities in [-1, 1], a hyperbolic tangent function (tanh) is used as the activation function.

For the critic-network (2 critic networks in the case of the TD3 Agent), the Q-value of the state and action pair is predicted. 3 fully-connected neural network layers were used to process the input state. The action is merged in the first fully-connected neural network layers. The Q-value is finally activated through a linear activation function. In the case of TD3, the Q-value is taken as the minimum of the two outputs from the critic networks.

The exploration strategy in DDPG and TD3 is typically based on adding noise to the deterministic actions output by the policy network. Since DDPG operates in continuous action spaces, exploration cannot rely on discrete action sampling (like in DQN); instead, it uses action noise during training. The standard deviation of the noise decays linearly over episodes:

$$\text{noise\_level} = 0.5 \times \left(1 - \frac{\text{ep}}{\text{episodes}}\right)$$

where *ep* is the index of the current episode during training. The noise level Starts at 0.5 and approaches 0 as training progresses to discouraging exploration.

3.4.4 Hyperparameters

The hyperparameters we used in the training of our DRL Agent are listed in the table below.

Parameter	**DDPG**	**TD3**	**Description**
max_steps	500	500	Max steps per episode
batch_size	64	40	Sampled batch size from replay buffer
replay_buffer_size	1,000,000	1,000,000	Experience replay memory size
Gamma (γ)	0.99	0.99999	Discount factor for future rewards
Tau(τ)	0.001	0.005	Soft update factor for target networks
Actor α	1e-4	1e-3	Learning rate for the actor optimizer
Critic α	1e-3	1e-3	Learning rate for the critic optimizer
Initial noise	0.5	1	Initial exploration noise

Table 3.1 - DRL Hyperparameters for DDPG and TD3

3.5 Simulation Environments

The training procedure of our DRL agents was implemented in three virtual environments simulated by Gazebo. We built two 10x10 m^2 indoor environments Env1 and Env2 with walls around them, as shown in Figure 3.3. The environment Env2 is the same as Env1 but it includes more obstacles. Both models of these two environments were learned from scratch. The third environment is the world used in the paper [23]. The target is represented by a green sphere object. In fact, it cannot be rendered by the laser sensor mounted on the TurtleBot.

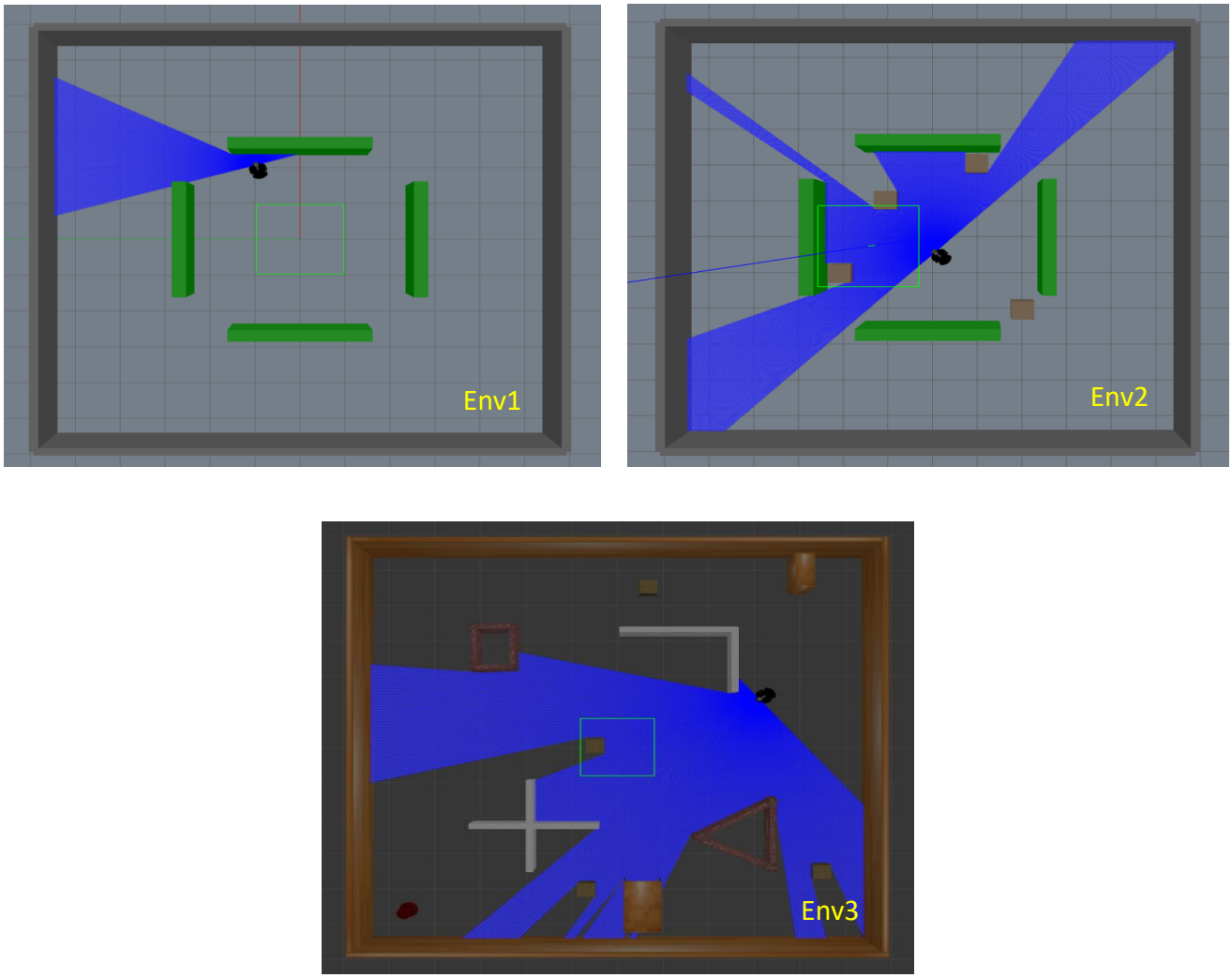


Figure 3.3: The virtual training environments

3.6 Results and Discussion

Our simulations were performed using CPU-only training on a computer equipped with Intel i5-8365U, 16 GB RAM. Our networks were trained in the Gazebo simulator and controlled by the Robot Operating System (ROS) commands. In our simulations, we adopted two scenarios described in the sections below. The training in the second scenario for example ran for 7500 episodes which took approximately ~1.75 days. Each training episode concluded when a goal was reached (< 0.2 meters), a collision was detected or 500 steps were taken. v_{max} and ω_{max} were set as 0.5 meters per second and 2.84 radian per second, respectively. The training was carried out in simulated 10x10 meter-sized environments as explained before.

3.6.1 Scenario 1

In the first simulation, we used the DDPG algorithm. We considered an initial robot position at the center (0,0) of environment ENV1 and a fixed goal at position (2, 1). Figure 3.4 shows a screenshot of RViz at the beginning of the training process. The robot's trajectory can be seen starting from the initial position and moving away from the goal represented as a green sphere at position (2, 1). This behavior occurs because the agent has not yet learned an optimal policy. At this early stage, actions are selected randomly due to the exploration phase of the learning process.

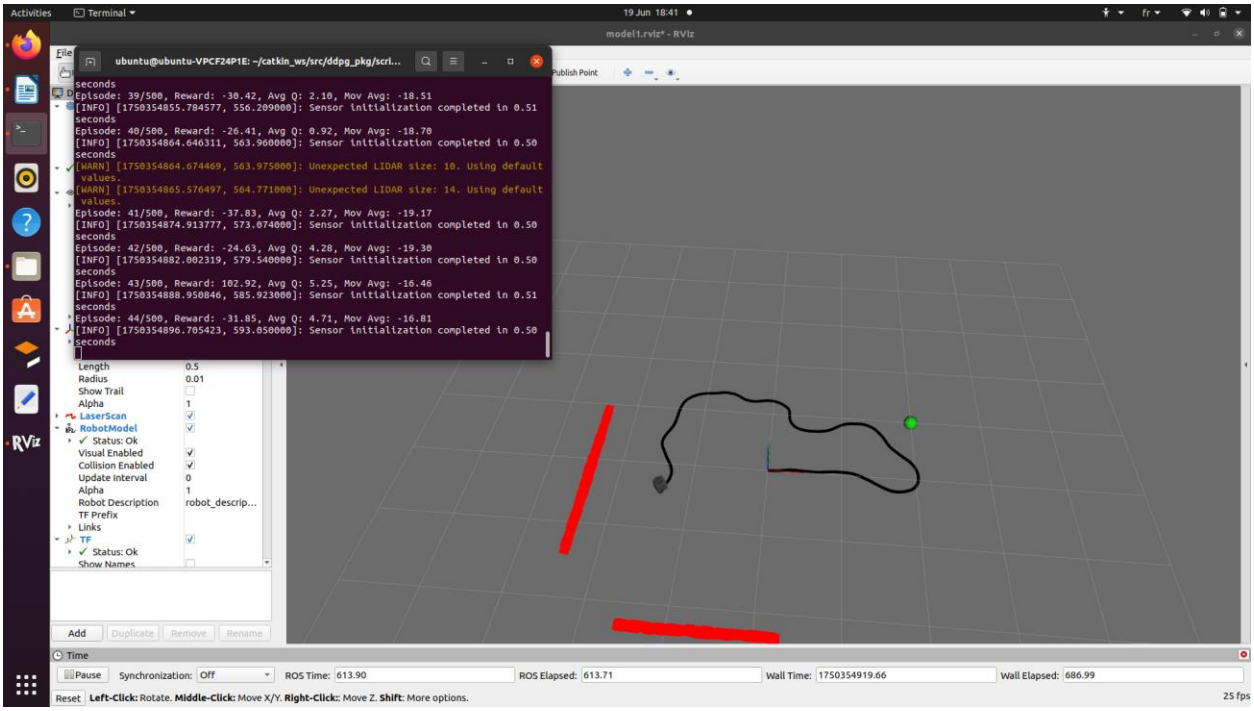


Figure 3.4: The beginning of the 1st scenario training process

As we said before, each training episode finished when a goal was reached, as shown in figure 3.5 a collision was detected and in figure 3.6 the robot reaching the goal (LIDAR scans are displayed in red in Rviz).

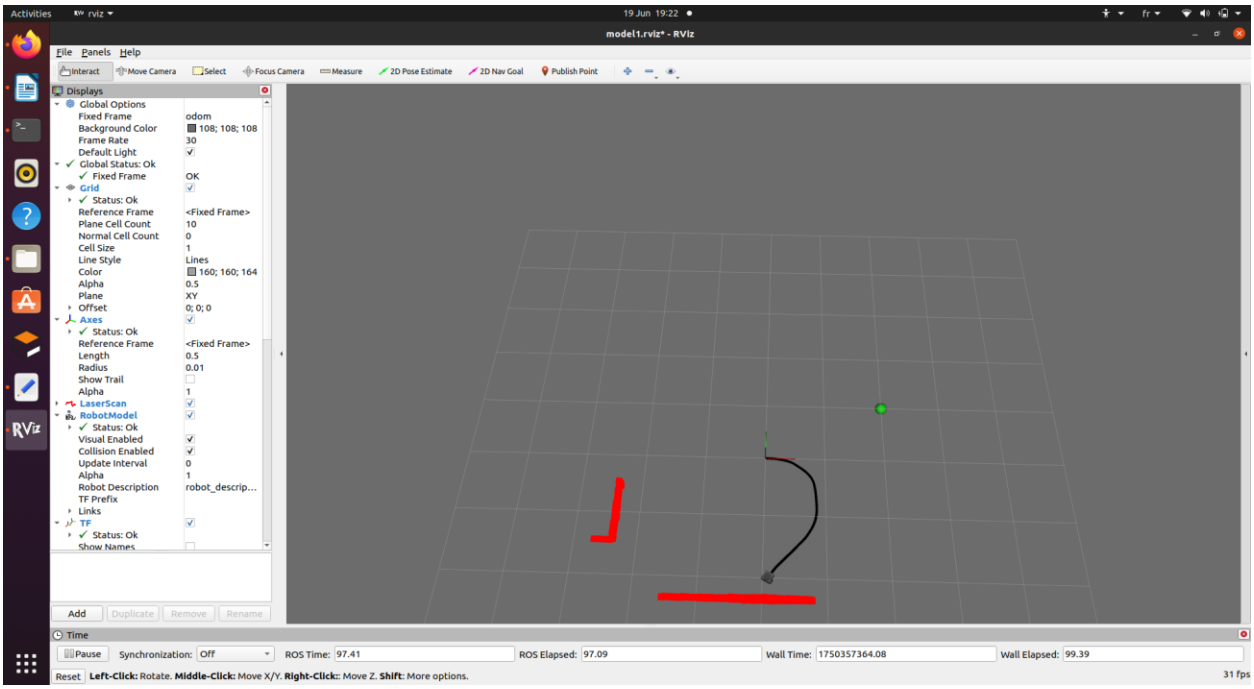


Figure 3.5 - Illustration of the detection of a collision

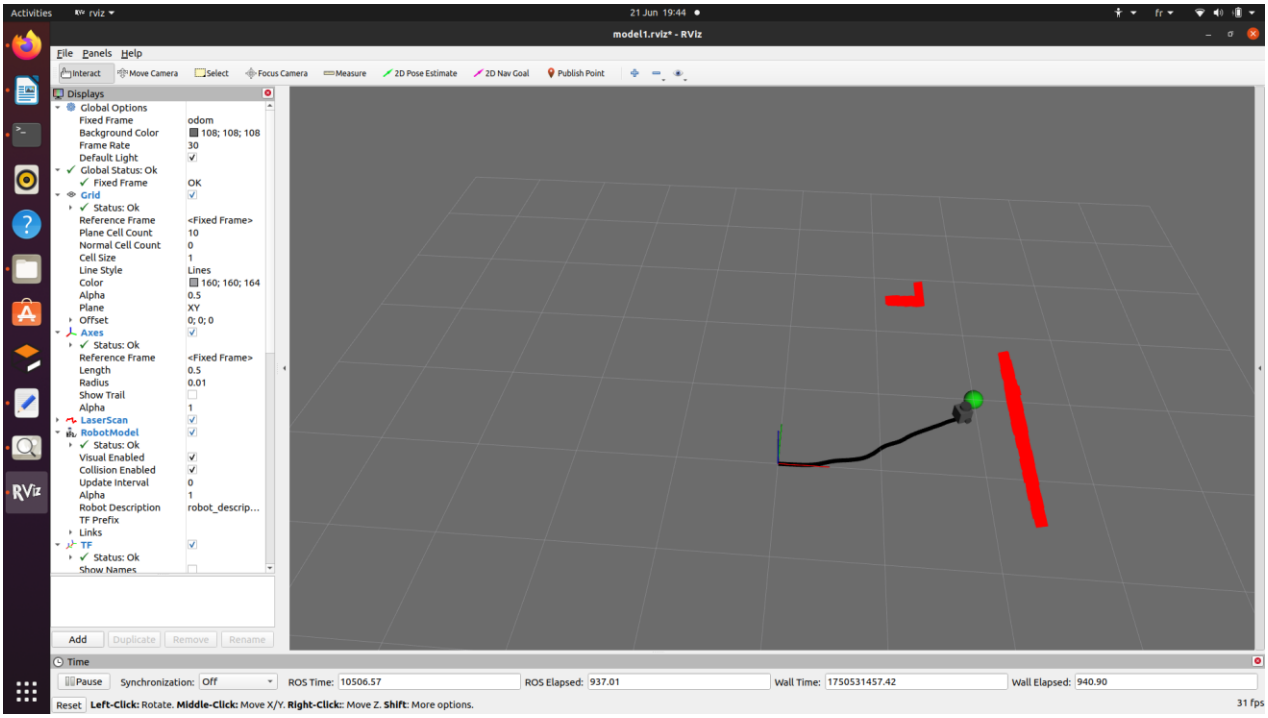


Figure 3.6 - The end of an exploration episode, showing the robot reaching the goal. After that, the robot resets to the initial position at the origin.

Figure 3.7 shows the curves of the Reward and the Q-value for the first 300 episodes of the training process. We can observe that the reward curve starts with low values but gradually increases as the number of training episodes grows. However, it exhibits significant fluctuations, which is expected during the exploration phase where actions are selected randomly to allow the robot to discover its environment. The Q-value curve also shows an upward trend, which is a positive indication that the DDPG agent is learning an effective policy. The fluctuations in the Q-value curve are likely caused by obstacles encountered by the robot along its path.

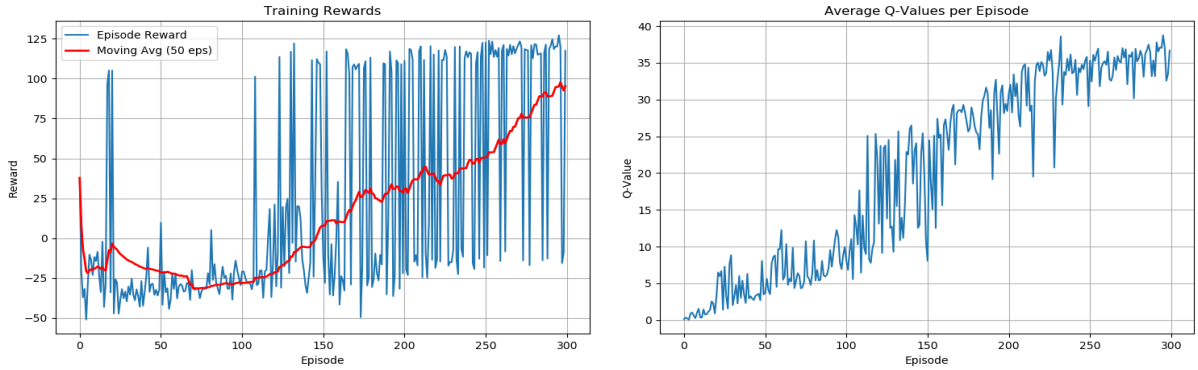


Figure 3.7: Training Rewards and Average Q-values per Episode

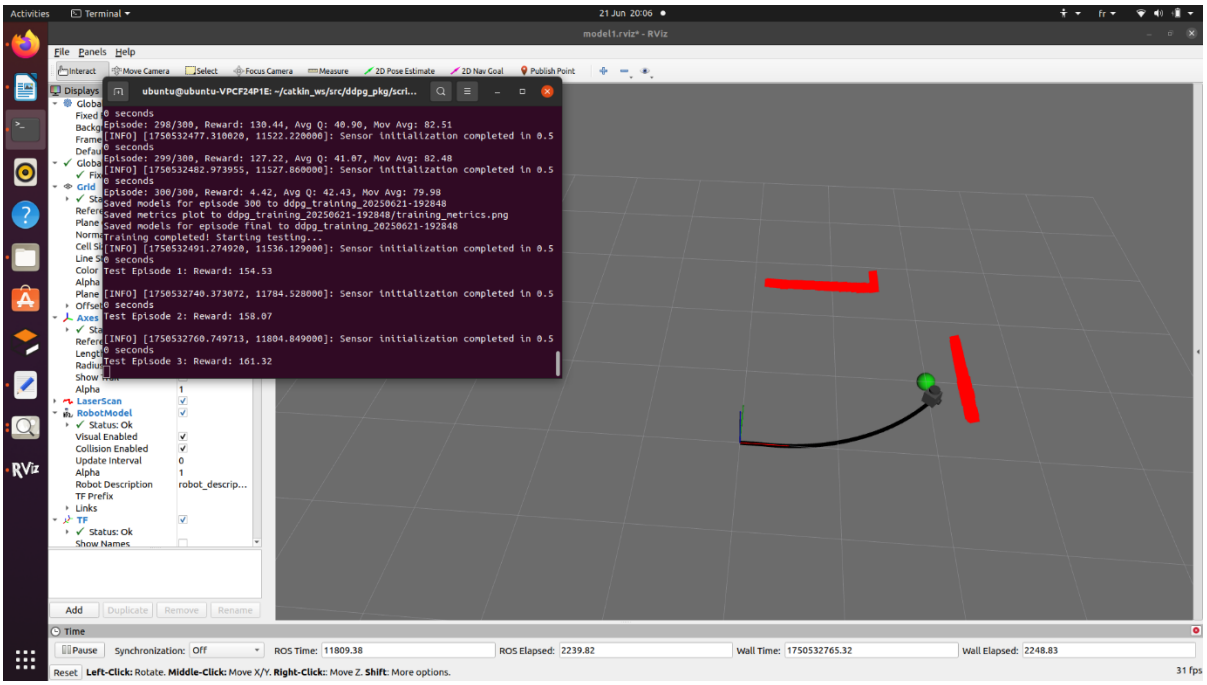


Figure 3.8: Results of the test phase once the training process finished

Once the training process was completed, we tested our trained DDPG agent over three episodes. In each test episode, the robot reached the goal smoothly, as shown in Figure 3.8, with a reward of 154.53 in the first episode, 158.07 in the second, and 161.32 in the third, as also displayed in the terminal output illustrated in Figure 3.8.

3.6.2 Scenario 2

In this experiment, the initial and target positions were initialized randomly in every episode within the whole area and guaranteed to be collision-free with other obstacles. We conducted this experiment in the third Environment ENV3 using the TD3 algorithm. ENV3 includes dynamic obstacles (the four boxes in ENV3) that change their positions randomly. Figure 3.9 highlights smooth obstacle avoidance and goal convergence toward the end of the training process.

The average Q-value curve shows the following evolution over the episodes:

- Episodes 1 – 1 000: The average Q drops sharply from 0 down to around –40, reflecting the agent's initial struggle with high variability (random obstacle placements and reward resets).
- Episodes 1 000 – 2 000: Recovery phase where avg Q climbs back to 0 as the agent learns basic avoidance and reward-collection patterns.
- Episodes 2 000 – 7 500: Steady increase to approximately 20, then plateaus, indicating that once the core behaviors are mastered, further gains are marginal despite continued training.

The Maximum Q-Value shows these details:

- Episodes 1 – 1500: Peaks around 0 until the agent first discovers high-reward trajectories.
- Episodes 1500 – 7500: Rapid jump to approximately 120 by episode 1500, and then consistently maintains that peak, suggesting occasional runs find near-optimal paths even as the average performance stays around 20.

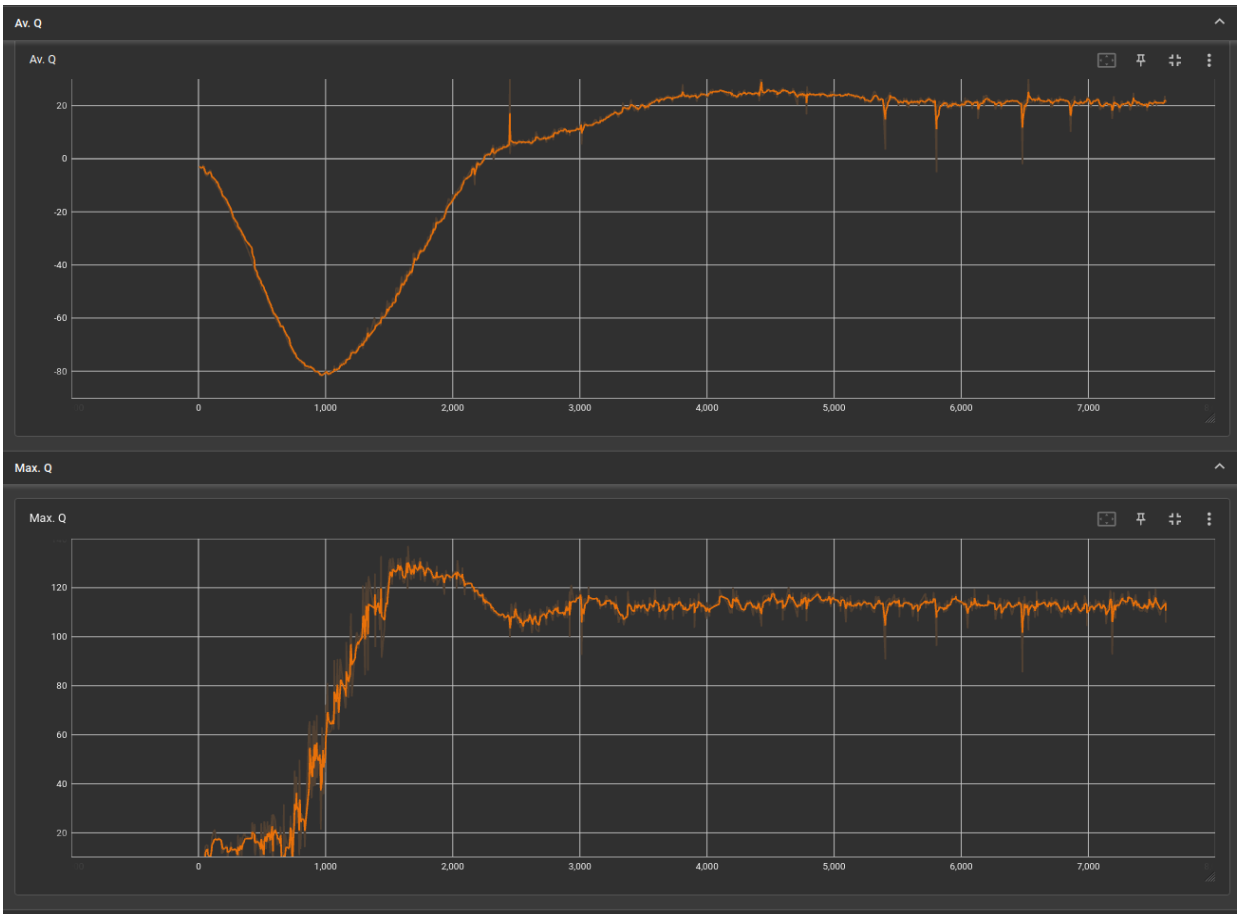


Figure 3.9 - Learning curves (avg. Q-value & max. Q-value) across 7000 episodes in ENV3

To evaluate the generalization performance of the TD3 policy under consistent environmental conditions but randomized internal variables, we conducted 10 evaluation episodes in the same test environment. Figure 3.10 shows the obtained results in the 1st and 7th test episodes. Each episode began with a random robot position and orientation, and the goal location was also randomized. This setup tests the agent's ability to adapt its behavior to varying initial conditions within a fixed map layout.

The agent achieved a success rate of 80% (8 out of 10 episodes), meaning it was able to reach the goal without collision or timeout in most trials. This high success rate demonstrates that the TD3 policy reliably handles spatial variability in initialization and reward signals, even without environmental structure changes. The two failed attempts were due to early

collisions caused by tight cornering near randomized box placements highlighting the policy's remaining sensitivity to specific geometric configurations.

(a) – Episode 1: Initial State

(b) – Episode 1: Near Goal

(c) – Episode 7: Initial State

(d) – Episode 7: Near Goal

Figure 3.10: Results for the 1st and 7th test episodes

3.7 Conclusion

In this chapter, we presented the end-to-end implementation of Deep Deterministic Policy Gradient (DDPG) and Twin Delayed DDPG (TD3) agents for mapless navigation in ROS/Gazebo. We detailed our system setup, including hardware specifications, software stack, and environment configurations and described the design of state and action spaces, network architectures, exploration strategies, and reward functions. By systematically testing across three Gazebo environments (four-wall map, static-box map, and map with dynamic obstacles) and visualizing robot behavior in both Gazebo and RViz, we demonstrated practical integration of perception and control in simulation.

Our learning-curve analyses further revealed distinct Q-value dynamics across environments. TD3 outperformed DDPG in convergence speed (20–30 % faster), final reward levels, and training stability, confirming the benefits of its clipped double-Q updates and delayed policy updates.

Overall, Chapter 3 confirms that off-policy DRL agents can be reliably trained for continuous mapless navigation, and it identifies concrete strategies to close the gap between average and peak performance. These insights lay the groundwork for future work on sim-to-real transfer, multi-agent coordination, and integration of adaptive exploration schedules in both simulation and real-world deployments.

General Conclusion

In this work, we developed and evaluated two state-of-the-art, off-policy deep reinforcement learning algorithms: Deep Deterministic Policy Gradient (DDPG) and Twin Delayed DDPG (TD3) for continuous, mapless navigation of a mobile robot within the ROS/Gazebo simulation framework. Chapters 1 and 2 laid the groundwork by surveying robotic motion-planning fundamentals, classical and learning-based approaches, and the necessary software and hardware components. In Chapter 3, we detailed our complete pipeline: defining observation and action spaces, crafting network architectures, designing reward functions, and conducting extensive multi-map experiments with side-by-side Gazebo and RViz visualizations.

Our quantitative results reveal that both agents can learn effective obstacle avoidance and goal-reaching behaviors, but TD3 consistently outperforms DDPG across key metrics: it converges 20–30 % faster, achieves higher and more stable average and maximum Q-values, and exhibits approximately an 8 % higher success rate on unseen test scenarios. Notably, our learning-curve analysis showed environment-dependent Q-value dynamics: in highly randomized arenas, agents recovered from early performance drops to maintain positive average Q-values, whereas in more structured or static settings they tended to converge prematurely to suboptimal policies. These insights motivated practical enhancements such as adaptive exploration schedules, curriculum learning, and prioritized experience replay—that can help preserve high-reward behaviors in complex environments.

Looking ahead, integrating these strategies should further narrow the gap between average and peak performance and bridge simulation-to-real transfer challenges. Future work will also explore model-based reinforcement learning for improved sample efficiency, multi-agent coordination for collaborative navigation tasks, and the incorporation of partial mapping or SLAM feedback to enrich the agent's state representation. Together, these avenues promise to advance the robustness and adaptability of DRL-based navigation in real-world mobile robotics.

References

[1] Raibert, M. H. *Legged Robots That Balance*. MIT Press, 2008.

[2] M. M. Zhang, A. S. Huang, and J. B. Homer, "Deep Q-Learning for Autonomous Mars Rover Navigation," in *Proc. IEEE Int. Conf. Robotics and Automation*, Stockholm, Sweden, May 2024, pp. 5123–5130.

[3] L. K. Chen and P. R. Venkatesh, "Continuous Control for Autonomous Driving Using TD3," *IEEE Trans. Vehicular Technology*, vol. 73, no. 2, pp. 1456–1467, Feb. 2024.

[4] S. T. Nguyen, H. Saddik, and Y. L. Li, "Cooperative Multi-Agent RL for Warehouse Automation," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, Detroit, MI, USA, Oct. 2023, pp. 2891–2898. [10]

[5] J. Xie and K. Wang, "End-to-End Navigation for TurtleBot3 Using DDPG with Sensor Fusion," *IEEE Access*, vol. 9, pp. 148233–148243, 2021.

[6] T. Bräunl, Embedded robotics: mobile robot design and applications with embedded systems. Springer Science & Business Media, 2008.

[7] A. Souliman, "Pose Estimation of Indoor Mobile Robot Based on Sensor Fusion," M.Sc. thesis, Mechatronics, University of Siegen, Siegen, Germany, Aug. 2018.

[8] Slamtec, *LDS-01 Scanning Laser Range Finder Technical Manual*, 2015.

[9] ROBOTIS, *Dynamixel XL430-W250-T Product Manual*, ver. 1.1, 2017.

[10] Raspberry Pi Foundation, *Raspberry Pi 3 Model B Product Brief*, 2016.

[11] ROBOTIS, *OpenCR1.0 Technical Overview*, 2017.

[12] MobileRobots Inc., "Pioneer 3-DX Mobile Robot," *MobileRobots Inc.* [Online]. Available: https://www.mobilerobots.com/ResearchRobots/PioneerP3DX.aspx

[13] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.

[14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., Cambridge, MA, USA: MIT Press, 2018.

[15] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529–533.

[16] "Deep reinforcement learning," *Wikipedia*, Jun. 2025. [Online]. Available: https://en.wikipedia.org/wiki/Deep_reinforcement_learning

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[18] A. Haber, "Deep Q-Networks (DQN) in Python from Scratch by Using OpenAI Gym and TensorFlow – Reinforcement Learning Tutorial," *AleksandarHaber.com*, [Online].

[19] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[20] V. Garcia Cruz, "Deep Reinforcement Learning based Approaches for Capacity Sharing in Radio Access Network Slicing," M.S. thesis, Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona, Universitat Politècnica de Catalunya, Barcelona, Spain, Jul. 2020

[21] T. P. Lillicrap *et al.*, "Addressing Function Approximation Error in Actor-Critic Methods," *arXiv:1802.09477*, Oct. 2018.

[22] S. Fujimoto, H. Van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," in *Proc. 35th Int. Conf. Machine Learning (ICML)*, Stockholm, Sweden, 2018, pp. 1582–1591.

[23] R. Cimurs, *DRL-robot-navigation*. [Online]. Available:

https://github.com/reiniscimurs/DRL-robot-navigation

[24] A. Souliman, *Pose Estimation of Indoor Mobile Robot Based on Sensor Fusion*, M.Sc. thesis, University of Siegen, 2018. [Online]. Available: https://doi.org/10.13140/RG.2.2.12965.35041