ក្រសួងអប់រំ យុវជន និងកីឡា

វិទ្យាស្ថានបច្ចេកវិទ្យាកម្ពុជា

ដេប៉ាតឺម៉ង់ ទេពកោសល្យឧស្សាហកម្ម និងមេកានិច

គម្រោងសញ្ញាបត្រវិស្វករ

ប្រធានបទ: **ការរុករកស្វយ័តនៃរ៉ូបូតចល័ត ដោយប្រើ** Twin-Delayed Deep Deterministic Policy Gradient (TD3)

និស្សិត : **ស្រឿន សុធារិទ្ធ និង ថោង រិទ្ធី**

ឯកទេស : **ទេពកោសល្យឧស្សាហកម្ម និងមេកានិច**

គ្រូទទួលបន្ទុក : **សាស្ត្រាចារ្យជំនួយ បណ្ឌិត ស្រង់ សារ៉ុត**

ឆ្នាំសិក្សា : **២០២៤-២០២៥**

MINISTERE DE L'EDUCATION, DE LA JEUNESSE ET DES SPORTS

INSTITUT DE TECHNOLOGIE DU CAMBODGE

DEPARTEMENT DE GENIE INDUSTRIEL ET MECANIQUE

MEMOIRE DE FIN D'ETUDES

Titre: **Autonomous Navigation System for Mobile Robot by using Twin-Delayed Deep Deterministic Policy Gradient (TD3)**

Etudiants : **SEOURN Sothearrith et THORNG Rithy**

Spécialité : **Génie Industriel et Mécanique**

Tuteur de stage : **Asst. Prof. Dr. SRANG Sarot**

Année scolaire : **2024-2025**

ក្រសួងអប់រំ យុវជន និងកីឡា
វិទ្យាស្ថានបច្ចេកវិទ្យាកម្ពុជា

ដេប៉ាតឺម៉ង់ ទេពកោសល្យឧស្សាហកម្ម និងមេកានិច

គម្រោងសញ្ញាបត្រវិស្វករ

បេស់និស្សិត: ស្មឿន សុធារិទ្ធ និង ថោច រិទ្ធី

កាលបរិច្ឆេទការពារនិក្ខេបបទ: ថ្ងៃទី ខែ កក្កដា ឆ្នាំ ២០២៥

អនុញ្ញាតឲ្យការពារគម្រោង

នាយកវិទ្យាស្ថាន: _____

ថ្ងៃទី ខែ ឆ្នាំ ២០២៥

ប្រធានបទ: **ការរុករកស្វយ័តនៃរ៉ូបូតចល័ត ដោយប្រើ** Twin-Delayed Deep Deterministic Policy Gradient (TD3)

សហគ្រាស: មន្ទីរពិសោធន៍ឌីណាមិច និងប្រព័ន្ធបញ្ជា (DCLab)

ប្រធានដេប៉ាតឺម៉ង់: សាស្ត្រាចារ្យជំនួយ បណ្ឌិត ចាន់ សារិទ្ធ

គ្រូទទួលបន្ទុក: សាស្ត្រាចារ្យជំនួយ បណ្ឌិត ស្រង់ សារ៉ុត

អ្នកទទួលខុសត្រូវក្នុងសហគ្រាស: បណ្ឌិត ស្រង់ សារ៉ុត

រាជធានីភ្នំពេញ

MINISTERE DE L'EDUCATION,
DE LA JEUNESSE ET DES SPORTS

INSTITUT DE TECHNOLOGIE DU CAMBODGE

DEPARTEMENT DE GENIE INDUSTRIEL ET MECANIQUE

MEMOIRE DE FIN D'ETUDES

DE M. SEOURN Sothearrith et M. THORNG Rithy

Date de soutenance: le juillet 2025

« Autorise la soutenance du mémoire »

Directeur de l'Institut: _____

Phnom Penh, le 2025

Titre: **Autonomous Navigation System for Mobile Robot by using Twin-Delayed Deep Deterministic Policy Gradient (TD3)**

Etablissement du stage: **Dynamics and Control Laboratory (DCLab)**

Chef du département: **Asst. Prof. Dr. CHAN Sarin**

Tuteur de stage: **Asst. Prof. Dr. SRANG Sarot**

Responsable de l'établissement: **Asst. Prof. Dr. SRANG Sarot**

PHNOM PENH

ACKNOWLEDGMENTS

First and foremost, we would like to express our deepest gratitude to **H.E. Prof. Dr. PO Kimtho**, General Director of the Institute of Technology of Cambodia, for his outstanding leadership and unwavering commitment to fostering academic excellence. His collaborative efforts with local, regional, and international partner universities have significantly contributed to the continuous improvement of educational quality at ITC.

We also would like to express our gratitude to **Asst. Prof. Dr. CHAN Sarin**, Head of the Department of Industrial and Mechanical Engineering, for his outstanding leadership and for fostering a positive and encouraging environment that greatly supported our academic journey.

This thesis would not have been possible without the invaluable guidance and mentorship of our respectful advisor, **Asst. Prof. Dr. SRANG Sarot**, Head of Dynamics and Control Laboratory. We are immensely grateful for his continuous support, expert advice, and encouragement throughout our final-year internship and research journey. His knowledge and dedication have been a cornerstone of our academic and professional growth.

We extend our heartfelt appreciation to our beloved parents, whose constant moral encouragement and financial support have been fundamental to our educational journey. Their unconditional love and sacrifices have empowered us to overcome challenges and pursue our goals with confidence.

Finally, we would like to thank all our colleagues and friends at ITC and the Dynamics and Control Laboratory. Their camaraderie, assistance, and collaborative spirit have been invaluable, especially during times of difficulty throughout our internship and research process.

សេចក្តីសង្ខេប

បច្ចេកវិទ្យារុករកផ្លូវនិងគេចឧបសគ្គដើម្បីទៅដល់គោលដៅគឺជាផ្នែកមួយសំខាន់សម្រាប់ការសិក្សាស្រាវជ្រាវ និងការអភិវឌ្ឍនៅក្នុងវិស័យរ៉ូបូត។ បច្ចេកវិទ្យាទាំងនេះចាំបាច់ត្រូវមានសមត្ថភាពយល់ដឹងពីបរិស្ថានជុំវិញខ្លួនរបស់វា ដើម្បីអាចគេចចេញពីឧបសគ្គដ៏ស្មុគស្មាញ និងឆ្ពោះទៅរកគោលដៅដោយស្វ័យប្រវត្តិគ្មានការបញ្ជាពីមនុស្ស។ នៅក្នុងនិក្ខេមបទនេះ ពួកយើងបង្រៀន TD3 algorithm ដែលជើរតួនាទីជាខួរក្បាលរបស់រ៉ូបូតក្នុងការរិភាគរកផ្លូវហើយបញ្ជាឲ្យវាធ្វើដំណើរក្នុងល្បឿនសមស្របដោយពឹងផ្អែកតែទៅលើស្ថានភាពនៃបរិស្ថានជុំវិញខ្លួន ដៀសវាងការបុកទង្គិចដើម្បីឆ្ពោះទៅរកគោលដៅដោយស្វ័យប្រវត្តិ ដូចទៅនឹងរបៀបដែលមនុស្សដើរទៅរកគោលដៅណាមួយ។

ដំណើរការនៃការបង្រៀនបញ្ញាសិប្បនិម្មិត (AI) នេះ បែងចែកជាដំហានៗ។ ដំហានទីមួយ បរិស្ថានសិប្បនិម្មិតត្រូវបានបង្កើតឡើងដោយប្រើកម្មវិធីឈ្មោះថា Isaac Sim ហើយក្នុងនោះមានរូបតកង់បីដែលមានរូបរាង 3D យ៉ាងលំអិតមានបំពាក់ឧបករណ៍វាស់ចម្ងាយ (Laser rang scanner) ត្រូវបានដាក់បញ្ចូល។ ដំហានបន្ទាប់ បញ្ញសិប្បនិម្មិត TD3 ត្រូវបានបង្រៀននៅក្នុងបរិស្ថាន(Environment)នេះ។ រ៉ូបូតត្រូវបានបង្វឹកឲ្យធ្វើដំណើរពីចំណុចចាប់ផ្តើម ទៅកាន់ទីតាំងគោលដៅជាច្រើនដែលបានកំណត់ជាមុន ដោយមានសមត្ថភាពគេចចេញពីឧបសគ្គនៅតាមគន្លងដែលវាធ្វើចរាចរណ៍។

ដើម្បីបង្វឹករ៉ូបូតនេះបាន យើងប្រើប្រាស់ទិន្នន័យសិប្បនិម្មិត (dataset) បង្កើតឡើងដោយកម្មវិធី Isaac Sim ដែលជាព័ត៌មាននៃបរិស្ថាន រួមមានដូចជាស្ថានភាពរបស់រ៉ូបូត ហើយនិងចម្ងាយពីវាទៅឧបសគ្គដើម្បីជានាថារ៉ូបូតអាចរៀននិងយល់ច្បាស់ពីបរិស្ថានជាក់ស្តែង។

លទ្ធផលនៃការតេស្តសាកល្បងបន្ទាប់ពីបង្រៀនហើយ បានបង្ហាញឲ្យឃើញពីប្រសិទ្ធភាពនៃបច្ចេកវិទ្យាបញ្ញាសិប្បនិម្មិត TD3 ក្នុងការរៀនធ្វើដំណើរដើម្បីទៅរកគោលដៅដែលបានដាក់ឲ្យ ដោយគ្មានការប៉ះទង្គិចជាមួយវត្ថុជុំវិញវា។ ក្នុងកំឡុងពេលវារៀន យើងឃើញថាបញ្ញាសិប្បនិម្មិតនេះធ្វើសកម្មភាពកាន់តែប្រសើរឡើងៗ ទៅតាមពេលវេលាដែលវារៀន។ ក្នុងការតេស្តលើកចុងក្រោយ បានបង្ហាញថារ៉ូបូតអាចធ្វើដំណើរដោយជោគជ័យចេញពីទីតាំងដើមទៅកាន់គោលដៅដែលបានដាក់ឲ្យ ដោយមិនបុកទង្គិចជាមួយឧបសគ្គ។

ABSTRACT

Autonomous navigation systems represent a fundamental area of research and development in the domain of mobile robotics. These systems must be capable of accurately perceiving their environment and avoiding the obstacles, thereby enabling robots to operate and traverse autonomously without continuous human intervention. In this thesis, we explore the application of the Twin-Delayed Deep Deterministic Policy Gradient (TD3) algorithm for autonomous navigation of a mobile robot.

The development process involves multiple stages. First, a simulated environment is constructed using Isaac Sim, where a detailed 3D model of the Carter differential drive mobile robot equipped with laser range sensors is deployed. Subsequently, the TD3 algorithm is integrated into the environment to enable autonomous decision-making and path planning. The robot is trained to navigate from a starting location to a sequence of predefined goal positions, avoiding obstacles along the way.

The training process utilizes a dataset comprising simulated laser scan data and robot state information, ensuring that the learning model is exposed to realistic and diverse scenarios. The TD3 model is trained using the Isaac Lab reinforcement learning framework. During training, critical performance indicators such as average reward, actor loss, and critic loss are monitored to assess the model's learning progress. The optimal weights obtained during training are saved for further testing and potential deployment.

Experimental results highlight the TD3 algorithm's effectiveness in learning to navigate complex environments. Over time, the model demonstrated significant improvements in path efficiency and obstacle avoidance. The final testing phase, conducted within the Isaac Sim environment, validated the robustness and generalization capability of the trained model. The robot successfully navigated from its initial position to multiple goal points placed at 3-meter intervals along the x-axis, with random y-axis values, completing the task without collisions or crashes.

ABBREVIATIONS AND SYMBOLS

Adam	Adaptive Moment Estimation.
AI	Artificial Intelligent.
ANN	Artificial Neural Network.
BCE	Binary Cross Entropy.
DDPG	Deep Deterministic Policy Gradient.
DL	Deep Learning.
DP	Dynamic Programming.
DPG	Deterministic Policy Gradient.
DQN	Deep Q-Network.
DRL	Deep Reinforcement Learning.
GPS	Global Positioning System.
MAE	Mean Absolute Error.
MC	Monte Carlo.
MDP	Markov Decision Process.
ML	Machine Learning.
MSE	Mean Square Error.
PPO	Proximal Policy Optimization.
RL	Reinforcement Learning.
RMSProp	Root Mean Squared Propagation.
RNN	Recurrent Neural Network.
ROS	Robot Operating System.
SARSA	State-Action-Reward-State-Action.
SGD	Stochastic Gradient Descent.
SLAM	Simultaneous Localization and Mapping.
SPG	Stochastic Policy Gradient.
TD	Temporal Different.
TD3	Twin Delayed Deep Deterministic Policy Gradient.
USD	Universal Scene Description.
URDF	Unified Robot Description Format.
$\Pr\{X = x\}$	Probability that a random variable X takes on the value x.
$X \sim p$	Random variable X selected from distribution $p(x) = \Pr\{X = x\}$.
$\mathbb{E}[X]$	Expectation of a random variable X.

lnx	Natural logarithm of x.
ε	Probability of taking action a random action in an ε −greedy policy.
α,β	Step-size parameters.
γ	Discount-rate factor.
δ_t	Temporal-different (TD) error at t .
θ	Parameter vector.
π_θ	Policy corresponding to parameter θ.
$J(\theta)$	Performance measure for the policy π_θ.
$\nabla J(\theta)$	Column vector of partial derivatives of $J(\theta)$ with respect to θ.
a	An action.
$\mathcal{A}(s)$	The set of all actions available in state s.
A_t	Action at time t.
$\mathcal{D}$	Replay buffer.
G_t	Return following time t.
$\mathcal{L}$	Loss Function.
$\mathcal{P}$	Transition probability matrix.
$\mathcal{P}_{ss'}^{a}$	Probability of transition to state s', from state s taking action a.
q	Action-value function.
$q_\pi(s,a)$	Value of taking action a in state s under policy π.
$q_\pi(s',a')$	Value of taking next action a' in the successor state s' under policy π.
$q_*(s,a)$	Value of taking action a in state s under the optimal policy.
$q_*(s',a')$	Value of taking the next action a' in the successor state s' under the optimal policy.
Q,Q_t	Array estimates of action-value function q_π or q_*.
r	An immediate reward.
$r(s,a)$	Expected immediate reward from state s after action a.
$r(s,a,s')$	Expected immediate reward on transition from s to s' under action a.
$\mathcal{R}$	The set of all possible reward, a finite subset of $\mathbb{R}$.
R_t	Reward at time t, typically due, stochastically, to S_{t-1} and A_{t-1}.
s	The current state.
s'	The successor state.
$\mathcal{S}$	The set of all nonterminal states.
S_t	State at time t, typically due, stochastically, to S_{t-1} and A_{t-1}.

t	Discrete time step.
v	State value function.
$v_\pi(s)$	Value of state s under policy π.
$v_*(s)$	Value of state s under the optimal policy.
$\hat{v}(s,\theta)$	Approximate value of state s given weight vector θ.
V, V_t	Array estimate of state-value function v_π or v_*.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Background

Autonomous navigation in mobile robots represents an important and rapidly evolving research domain, with many applications spanning from disaster recovery and search-and-rescue operations to self-driving vehicles and delivery systems. Traditional navigation approaches generally depend on pre-constructed maps and external control systems, which constrain their flexibility and responsiveness particularly in dynamic, unpredictable environments. However, recent breakthroughs in Artificial Intelligence (AI) such as Machine Learning (ML) and Deep Learning (DL) have significantly expanded the possibilities for robot autonomy. Among these, Deep Reinforcement Learning (DRL) stands out as a promising paradigm for enabling robots to navigate effectively in unfamiliar and changing setting. Unlike conventional techniques that require substantial manual setup, such as hand-crafted features, environment modeling, or human guidance, DRL empowers robots to autonomously learn efficient navigation behaviors by interacting directly with their environment. By continuously receiving feedback in the form of rewards or penalties, a DRL enables robot refines its decision-making process, improving not only in terms of obstacle avoidance but also in finding more optimal, energy-efficiency, and adaptive paths. This leads to more robust and scalable navigation systems, capable of functioning in real-world scenarios where prior knowledge may be incomplete or entirely unavailable (Mahrous et al., 2022), (Harapanahalli et al., 2019), (Vaidya, 2021), and (Ali, 2023).

1.1.1. Autonomous Navigation

Autonomous navigation is the process by which robot senses its surroundings, localizes itself within a map, plans a path, and moves toward a goal while avoiding obstacles and adapting to dynamic changes in the environment.

There are many core components of autonomous navigation system such as perception, localization, mapping, path planning, obstacle avoidance, and motion control. However, for autonomous navigation with modern approach like DRL the number of core components is reduced to perception, obstacle avoidance, and motion control.

1.1.2. Perception

For mobile robots to navigate safely, they need to sense their surroundings and make decisions based on the perception (Kortenkamp, 1994). To perceive a surrounding environment effectively, a robot requires various sensors such as LiDAR, depth cameras (e.g., Intel

RealSense), ultrasonic sensors, and IMUs. The selection of sensors depends on the specific application and the environment in which the robot operates.

Perception is the process by which a robot collects raw sensor data from its surroundings and transforms it into meaningful information such as detecting obstacles, identifying landmarks, understanding terrain, or recognizing people. This processed information plays a critical role in supporting decision-making during autonomous navigation, enabling the robot to localize itself, build maps, plan paths, and safely interact with dynamic environments.

1.1.3. Localization and Mapping

Simultaneous Localization and Mapping (SLAM) is a core problem in robotics that involves constructing a map of an unfamiliar environment while simultaneously determining the robot's position within that map. This dual-task process is particularly critical in indoor settings, where GPS signals are typically unavailable and no single sensor can provide highly accurate localization on its own. Consequently, a robot must rely on imperfect sensor data to estimate both its trajectory and the layout of its surroundings (Vaidya, 2021).

Accurate localization and up-to-date environmental mapping are essential for enabling autonomous navigation. When a robot can reliably localize itself within a continuously updated map, it gains the situational awareness needed to plan and execute movements effectively.

SLAM is fundamentally a probabilistic problem, often approached through state estimation techniques. Two of the most established frameworks for solving SLAM are based on the Extended Kalman Filter (EKF) and Rao-Blackwellized Particle Filters (RBPF). These techniques estimate the robot's position and map features while managing uncertainty through probabilistic models. The SLAM process typically relies on a suite of onboard sensors. Commonly used sensors include Inertial Measurement Units (IMUs), wheel encoders, LiDAR, and RGB or stereo vision cameras. In practice, sensor fusion combining data from multiple sensor modalities yields more robust and reliable results.

However, implementing SLAM in real-world scenarios presents several challenges. Dynamic environments, where objects or people may move unpredictably, introduce non-static elements that can degrade mapping accuracy. Additionally, sensor noise, drift, and occlusion further complicate reliable localization and mapping. Research continues to explore more resilient and adaptive SLAM algorithms, including those that incorporate deep learning or semantic understanding to enhance perception in complex and changing environments.

1.1.4. Path Planning and Obstacle Avoidance

Path planning is a core component of autonomous robot navigation that involves determining a feasible and optimal path from a robot's starting location to its goal, while avoiding obstacles and satisfying various constraints such as dynamic limitations or safety margins (Song et al., 2023). It enables the robot to reason about the environment and make decisions on how to traverse it efficiently and safely. There are generally two types of path planning:

- Global Path Planning: The robot has full knowledge of the environment (e.g., a predefined map). The algorithms like A^*, Dijkstra's, and RRT (Rapidly-Exploring Random Tree) are commonly used.
- Local Path Planning: The robot reacts to immediate sensor inputs in real-time, dealing with dynamic obstacles and uncertainties. The commonly used algorithms include DWA (Dynamic Window Approach) and APF (Artificial Potential Fields).

1.1.5. Motion Control

Motion control refers to the process of managing and commanding the movement of a robot to follow a desired path, speed, or position. It involves controlling the actuators so that the robot moves accurately and efficiently according to a predefined plan or real-time input. The key components of motion control are:

- Trajectory: The desired position, velocity, or orientation is given either by path planner or human operator.
- Controller: Algorithms (like PID or adaptive control) are used to compute control signals that correct errors between desired and actual motion.
- Actuators: Physical components (e.g., motors) that generate the require movement.
- Feedback system: Sensors (e.g., encoders, IMUs) provide real-time data on the system's actual position, speed, or force, enabling closed-loop control.

1.2. Problem Statement

Navigating complex and dynamic environments remains a significant challenge for mobile robots. To address this, Deep Reinforcement Learning (DRL) has emerged as a powerful solution due to its ability to learn directly from interaction with the environment and to adaptively handle diverse and unpredictable scenarios. As a result, the application of DRL in

autonomous robot navigation is gaining growing attention in both academic and industrial research.

DRL enables robots to develop optimal policies through trial and error, leveraging feedback from the environment to improve over time. However, early DRL methods such as Deep Q-Networks (DQN) are limited in their ability to handle continuous action spaces, primarily because DQN relies on discrete action selection (Mnih et al., 2013). This limitation leads to inefficient exploration and suboptimal decision-making, especially in tasks requiring fine-grained control and smooth transitions between actions.

To address this issue, the Deep Deterministic Policy Gradient (DDPG) algorithm was introduced. DDPG is an off-policy algorithm that splits exploration problem independently from the learning algorithm (Lillicrap et al., 2016). DDPG combines the actor-critic architecture with deterministic policy gradients, allowing it to operate effectively in continuous action spaces. The actor network suggests actions, while the critic evaluates them, facilitating more precise control. Nevertheless, DDPG is prone to overestimation bias where the estimated Q-values tend to be higher than the true expected values, which can degrade learning performance and lead to unstable behaviors.

To mitigate overestimation, the TD3 algorithm was proposed. TD3 enhances DDPG by incorporating twin Q-networks and using the minimum of the two Q-value estimates to reduce bias, a concept inspired by Double Q-learning (Fujimoto et al., 2018). Additionally, TD3 delays the update of the policy network relative to the Q-networks, which helps stabilize learning. It also introduces target policy smoothing, adding noise to target actions during critic updates, which improves robustness against function approximation errors.

With these improvements, TD3 demonstrates superior stability and efficiency in learning policies for complex control tasks. Its robustness makes it particularly well-suited for real-world robotic applications, such as autonomous navigation in cluttered or dynamically changing indoor environments. This thesis aims to harness the capabilities of the TD3 algorithm to develop a reliable and intelligent navigation system for mobile robots, enabling them to operate autonomously with high precision and adaptability.

1.3. Goal and Objective

1.3.1. Goal

Our goal is to develop an autonomous navigation system for a differential mobile robot (Carter) using Reinforcement Learning algorithm.

1.3.2. Objective

- To investigate the effectiveness of the TD3 reinforcement learning algorithm in training a differential drive mobile for goal-directed navigation.

1.4. Scope and Limitation

1.4.1. Scope

- This research is conducted through simulations on the Isaac Sim platform using the Isaac Lab framework.
- The model is trained using virtual data generated within Isaac Sim.

1.4.2. Limitation

- A differential drive mobile robot is used as the primary robot for this study.
- Robot is equipped with laser range scanner (LiDAR).

2. LITERATURE REVIEW

2.1. Isaac Sim

NVIDIA Isaac Sim is a robotics simulation platform built on the NVIDIA Omniverse framework, designed to facilitate the development, simulation, and validation of AI-powered robots within high-fidelity, physics-based virtual environments. It supports the integration of robotic models from widely used formats such as URDF, MJCF, and on shape through the Universal Scene Description (USD), a scalable and extensible 3D scene representation that serves as the core data exchange format (Isaac Sim Documentation, 2025).

A key feature of Isaac Sim is its robust simulation engine, based on NVIDIA PhysX and RTX technologies, which enables real-time rendering and the simulation of various sensors, including cameras, LiDAR, and tactile sensors. This capability allows for the creation of digital twins and comprehensive testing pipelines prior to physical deployment. Isaac Sim also provides tools such as Replicator for generating synthetic datasets, Omnigraph for simulating dynamic environments, and tuning mechanisms to ensure alignment between simulated and real-world behavior. Reinforcement learning is supported through integration with Isaac Lab.

Furthermore, Isaac Sim includes all necessary components for deploying robotic applications to real-world systems. It offers interoperability with ROS and ROS 2, enabling seamless communication between simulated environments and physical robots. The inclusion of NVIDIA Isaac ROS further enhances this by providing optimized ROS 2 packages aimed at accelerating the development of autonomous robotic systems.

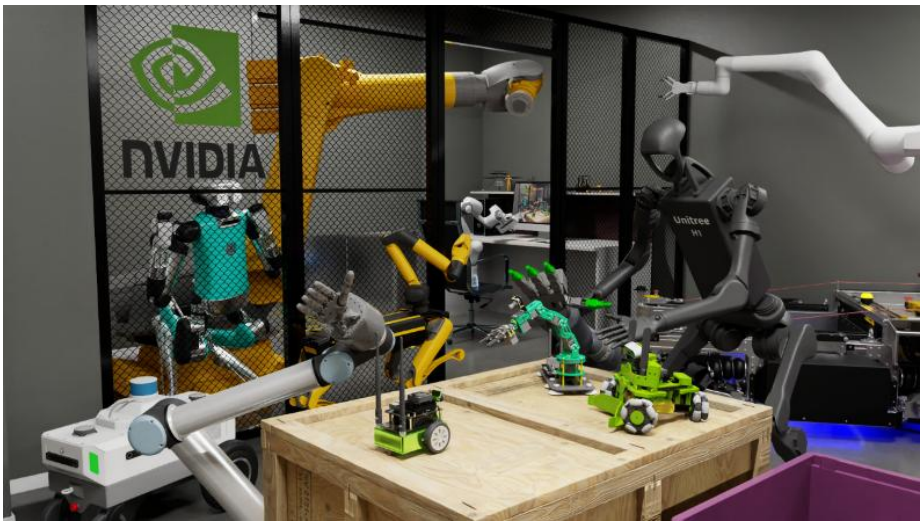

Figure 2.1. Simulated robotics lab environment in Isaac Sim (Isaac Sim Documentation, 2025).

System architecture: The primary objective of Isaac Sim is to facilitate the development of new robotic tools while enhancing and integrating with existing ones. It offers a versatile API compatible with both C++ and Python, allowing for varying levels of integration based on specific project requirements. Rather than acting as a replacement for existing robotics software, Isaac Sim is designed to complement and extend their capabilities. In alignment with this collaborative approach, many of Isaac Sim's components are open source and available for independent use. For instance, users can design a robot using On Shape, simulate its sensor data within Isaac Sim, and manage its operation using ROS or another communication framework. Alternatively, the platform also supports the development of entirely standalone robotics applications within its environment.

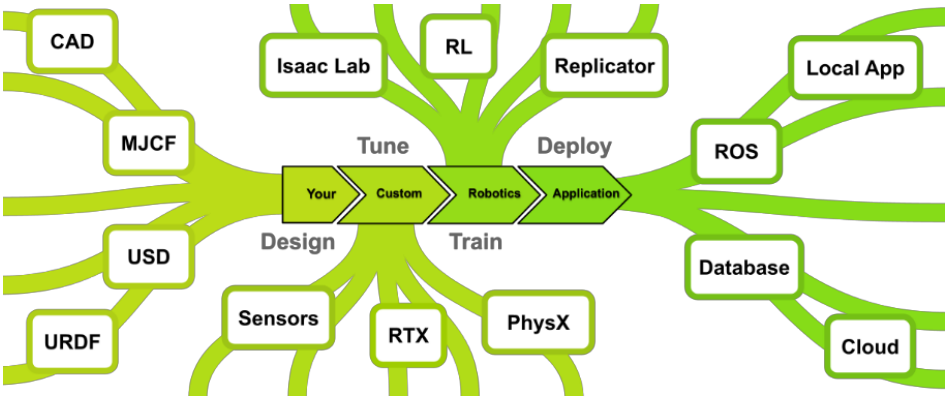

Figure 2.2. Isaac Sim system diagram (Isaac Sim Documentation, 2025).

2.2. Mobile Robot Modeling

A differential drive mobile robot is a common and well-established platform in the field of robotics due to its mechanical simplicity and ease of control. It consists of two main driving wheels mounted on a common axis, each powered by an independent motor (Petrov, 2010). As shown in **Figure 2.3.**, this configuration allows for versatile movement control by varying the speed of each wheel individually. To maintain balance and ensure the robot does not tip over, one or more passive caster wheels are often added to the chassis. These caster wheels are not powered and serve purely for support.

The motion of the differential drive robot is achieved through the coordinated speed control of its two main wheels. When both wheels move forward at the same speed, the robot travels in a straight line. To change direction, the robot varies the speed of one wheel relative to the other. If one wheel moves faster than the other, the robot follows a curved path. If the wheels rotate at equal speed in opposite directions, the robot can rotate in place about its central

axis. This mechanism gives the robot a high degree of maneuverability, particularly useful in environments with limited space.

In robotics, the pose of a differential drive robot refers to its location and orientation within a 2D Cartesian coordinate system. Specifically, this includes the X-coordinate (horizontal position), Y-coordinate (vertical position), and the orientation angle (θ), which describes the robot's heading relative to a reference axis (typically the x-axis).

To describe the motion of the robot in terms of kinematics, two fundamental velocity components are used:

- **Linear Velocity (v)**: This is the rate at which the robot's position changes along its trajectory. It typically refers to the forward or backward motion along the robot's local x-axis.
- **Angular Velocity (ω)**: This indicates how quickly the robot is rotating around its center. It defines the rate of change of the robot's orientation in the plane.

By combining these two components, the complete motion of the robot in a 2D plane can be described. This simple yet powerful control mechanism makes the differential drive model ideal for tasks such as indoor navigation, mapping, and path-following in mobile robot applications.

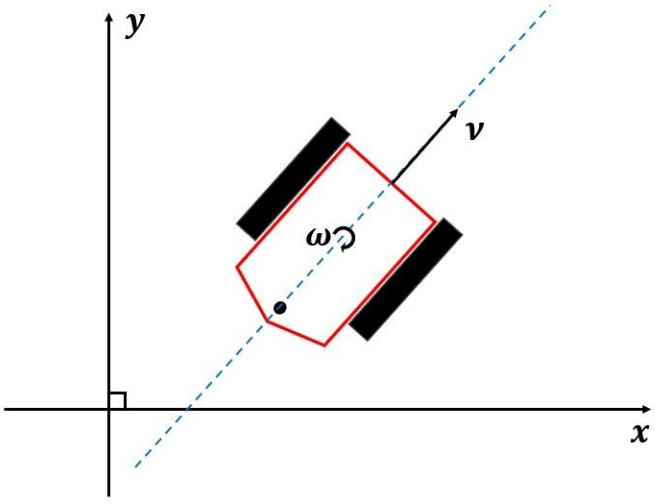

Figure 2.3. The differential drive wheel mobile robot.

2.3. Machine Learning

Machine Learning is a branch of Artificial Intelligence focused on developing software agents that can automatically enhance their performance through experience (Mitchell, 1997).

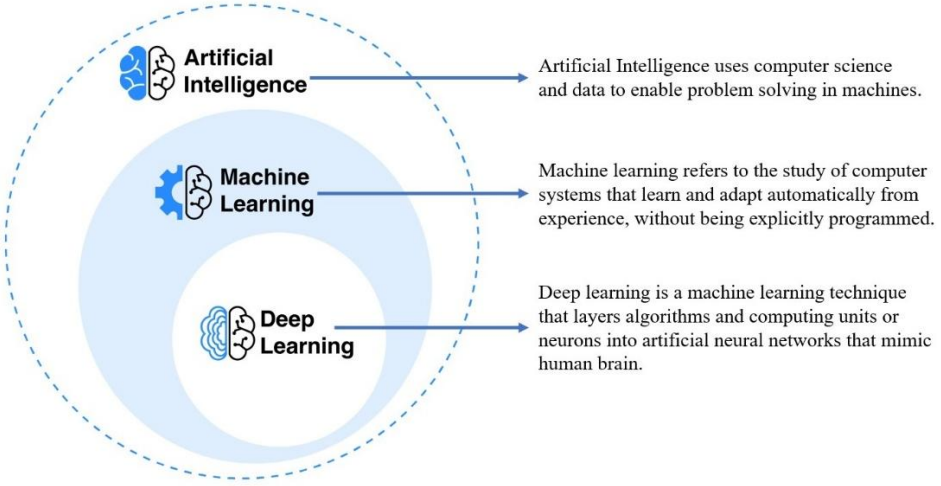

Figure 2.4. Branch of Artificial Intelligence.

It involves training algorithms to learn patterns from data and make decisions without being explicitly programmed. It encompasses various learning paradigms, including supervised, unsupervised and reinforcement learning. The different types of machine learning are depicted in **Figure 2.5.**.

- Supervised learning involves training on a dataset of labeled examples, where a domain expert acts as an external supervisor during the learning process. The main objective is to equip the learning agent with the ability to generalize and make accurate predictions on new or unseen data (Naeem et al., 2020). This method is widely used in tasks like classification and regression, where the agent learns from labeled data to map inputs to correct outputs effectively.
- Unsupervised learning involves discovering hidden patterns and insights within a dataset without any guidance or supervision (Naeem et al., 2020). The agent independently analyzes the data to identify underlying structures, relationships, or clusters, often revealing trends or groupings that are not immediately obvious. This approach is particularly useful for tasks like data compression, anomaly detection, or feature extraction, where the goal is to uncover meaningful information without predefined labels.
- Reinforcement Learning (RL) is a learning approach driven by a specific goal. An agent acquires knowledge by engaging with an unfamiliar environment, typically through a process of trial and error. This mirrors the natural learning style of a child, who acts and observes the outcomes. The agent receives feedback in the form of rewards or penalties

from the environment, using this input to refine its skills and build experience and understanding of the surroundings (Naeem et al., 2020).

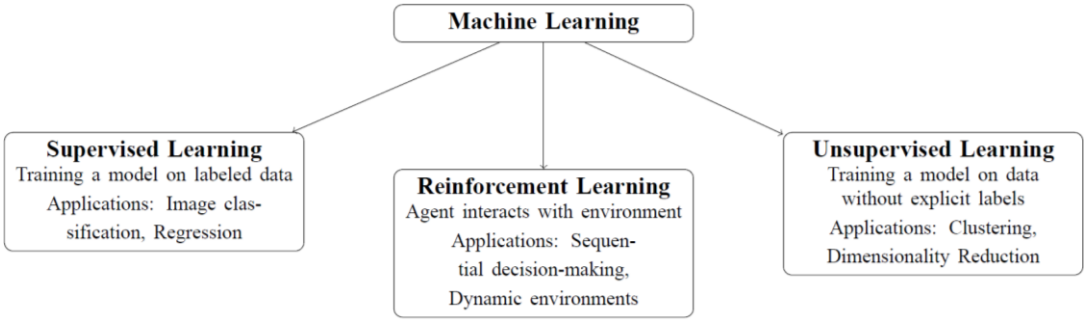

Figure 2.5. Different types of Machine Learning.

2.4. Deep Learning

Deep Learning is a powerful subfield of machine learning that focuses on algorithms modeled after the structure and function of the human brain, commonly referred to as artificial neural networks. Unlike traditional machine learning techniques that often rely on handcrafted features and domain-specific knowledge, deep learning enables systems to automatically learn hierarchical patterns and representations directly from raw data. This is achieved through the use of multiple layers of interconnected nodes or neurons, which form what are known as deep neural networks.

Each layer in a deep neural network transforms the input data into increasingly abstract representations. The lower layers typically learn simple features such as edges or textures in image data, while the higher layers capture more complex structures like objects or semantic meaning. By stacking many such layers, deep learning models can discover intricate patterns that are difficult or impossible to specify manually.

2.4.1. From Biological to Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational models inspired by the structure and functionality of the biological neural networks found in the human brain (Nizam, 2024). The development of ANNs is based on the understanding of how the brain processes information, learns from experience, and adapts over time.

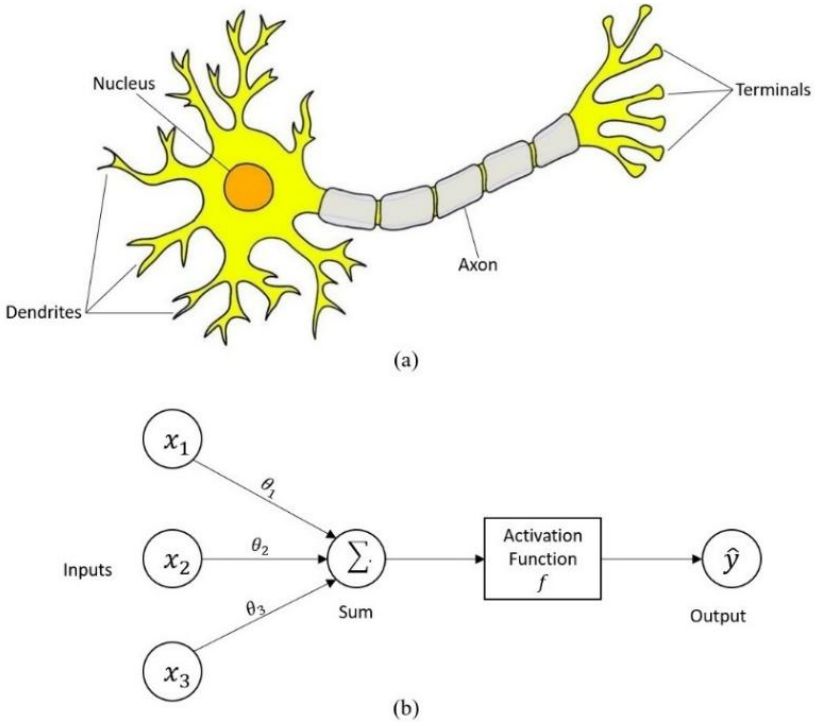

Figure 2.6. Comparison between (a) Biological Neuron and (b) Artificial Neuron.

The comparison between a biological neuron and an artificial neuron, highlighting their structural and functional similarities, is clearly illustrated in **Figure 2.6.**.

A biological neuron consists of dendrites, cell body (Soma), axon, and synapse. Dendrites receive signals from other neurons. Cell body integrates incoming signals. Axon transmits the signal to other neurons. Synapse is the junction where the axon of one neuron connects with the dendrite of another, where learning and memory are stored by adjusting the strength (or weight) of these connections.

Neurons communicate with each other via electrical impulses and chemical signals (Nizam, 2024). Learning occurs through synaptic plasticity, where the strength of the synapse changes depending on the activity. ANNs attempt to mimic this behavior in a simplified, mathematical form. Inputs to artificial neurons are equivalent to dendrites of biological neuron. These inputs are multiplied by weights which is similar to synaptic strength. A summation function aggregates the weighted inputs. An activation function (e.g., sigmoid, ReLU) determines the output that is passed on to neurons in the next layer.

Unlike biological systems, ANNs are trained using algorithms like backpropagation and gradient descent to minimize prediction errors. Though artificial neurons are significantly less complex than their biological counterparts, they have proven effective in solving real-world problems.

2.4.2. The Perceptron Model

The perceptron, a fundamental type of artificial neuron, serves as a building block for more sophisticated neural network architectures (GeeksforGeeks, 2024). It functions by calculating a weighted sum of its input features, where each input is multiplied by a corresponding weight, and then adding a bias term to the result. This sum is subsequently passed through an activation function as illustrated in **Figure 2.7.**, to generate a binary or continuous output, depending on the specific implementation. This process enables the perceptron to make decisions or classify inputs.

The perceptron acquires knowledge from examples via a procedure known as training. Throughout this training phase, the perceptron fine-tunes its weights in response to identified errors, typically utilizing a learning algorithm like the perceptron learning rule or backpropagation method. During training, the perceptron is provided with labeled examples that include known desired outputs. It then assesses its own output against the expected output, adjusting its weights to reduce the difference between the predicted and target results. This training mechanism enables the perceptron to determine the appropriate weights, allowing it to deliver accurate predictions for new, unfamiliar inputs.

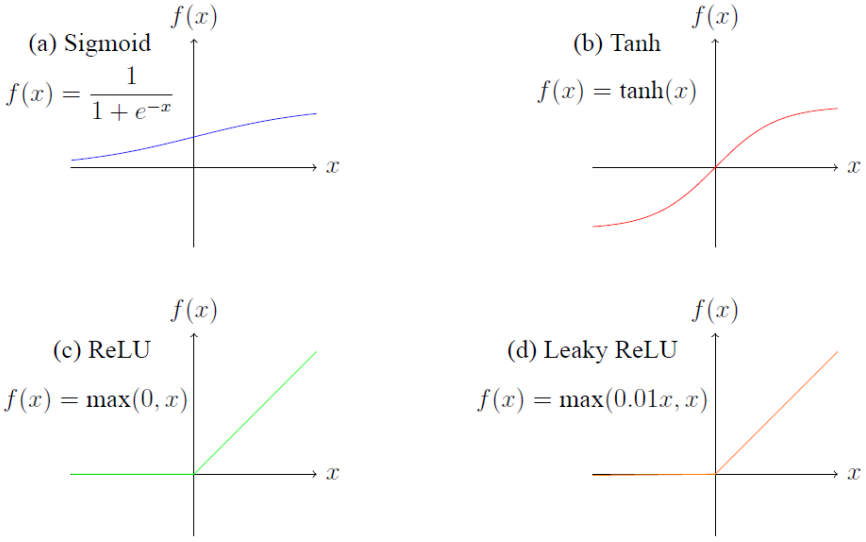

Figure 2.7. Comparison of commonly used activation functions.

- **Sigmoid**: The sigmoid activation function is a mathematical function commonly used in artificial neural networks to introduce non-linearity into the model. It maps any real-valued input to a value between 0 and 1, making it especially useful when

the output is expected to represent a probability (Goodfellow et al., 2016). Mathematically the sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}}. \qquad \textbf{(Eq. 2.1.)}$$

- **Tanh**: is a non-linear activation function commonly used in artificial neural networks, particularly in hidden layers. It transforms real-valued input into a smooth, zero-centered output in the range (–1, 1), making it suitable for models where negative, neutral, and positive signals are meaningful (Goodfellow et al., 2016). Mathematical expression:

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}. \qquad \textbf{(Eq. 2.2.)}$$

- **ReLU (Rectified Linear Unit)**: is a popular and computationally efficient activation function used in deep learning models, especially in convolutional neural networks (CNNs), fully connected networks, and deep residual networks (Goodfellow et al., 2016). Mathematical definition as:

$$\mathrm{ReLU(x)} = \max(0, \mathrm{x}). \qquad \textbf{(Eq. 2.3.)}$$

- **Leaky ReLU**: is a widely used non-linear activation function in artificial neural networks that serves as an improvement over the standard ReLU. It was introduced to address the limitations of ReLU, specifically the issue known as the "dying ReLU" problem (Maas et al., 2013). Mathematical formulation is defined as:

$$\text{Leaky ReLU}(x) = \begin{cases} x, if\ x > 0 \\ \alpha x, if\ x \leq 0 \end{cases}. \qquad \textbf{(Eq. 2.4.)}$$

2.4.3. Training of Neural Networks

Training a neural network involves adjusting its weights based on the error of its predictions. This process typically uses algorithms such as backpropagation and gradient descent to minimize a loss function that quantifies the prediction error (Prince, 2023). The process of training a neural network is iterative. During each iteration, a forward pass through the model's layers calculates an output for each training sample in a batch of data. Then, in a subsequent run through the layers, a gradient with respect to each parameter is computed to propagate how much each parameter impacts the final result.

The training process occurs in repeated cycles known as epochs. In each epoch, a forward pass is executed in which the input data is propagated through the network's layers to generate an output. This output is then compared to the ground truth, and the loss is calculated. Following this, a backward pass (backpropagation) computes the gradient of the loss function with respect to each parameter in the network. These gradients indicate how much each weight contributes to the overall error, enabling the model to update its parameters accordingly through gradient descent or its variants (e.g., Adam, RMSprop).

Over successive iterations, this process allows the network to learn meaningful patterns in the data, gradually improving its performance on the training set and ideally generalizing well to unseen data. The iterative nature of training ensures that the network incrementally refines its predictions, ideally converging toward an optimal or near-optimal set of parameters.

2.4.4. Loss Functions

The loss or cost function yields a single value that indicates the difference between the model's predictions and the true ground-truth (Prince, 2023). In training, we adjust the parameters to minimize this loss, aiming to align the training inputs with the outputs as closely as possible. Common loss functions include binary cross-entropy for classification tasks and mean squared error (MSE), mean absolute error (MAE) for regression tasks as shown in **Table 2.1.**.

Table 2.1. Summary of common Loss Functions.

Loss Function	Equation	Application
Binary Cross-Entropy	$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{p}_i) + (1-y_i)\log(1-\hat{p}_i)]$	Classification
MSE	$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$	Regression
MAE	$L = \frac{1}{N}\sum_{i=1}^{N}\lvert y_i - \hat{y}_i\rvert$	Regression

- **Binary Cross-Entropy** (BCE) is used for binary classification problems, such as spam detection or predicting whether an image contains any specific object (yes or no). It measures the error between predicted probabilities ($\hat{p}_i$) and true binary labels ($y_i \in \{0,1\}$). Here, N is the number of samples, y_i is the true label (0 or 1), and p_i is the predicted probability of the positive class (output of a sigmoid activation function, typically between 0 and 1). The term $y_i \log(\hat{p}_i)$ penalize the model when $y_i =$

1 but $\hat{p}_i$is low (i.e., the model is confident in the wrong prediction). The term $(1 - y_i) \log(1 - \hat{p}_i)$ penalizes the model when $y_i = 0$ but $\hat{p}_i$ is high. The negative sign ensures the loss is positive, and the average over N normalize the loss.

- **Mean Squared Error** (MSE) is used for regression tasks. It calculates the average squared difference between the true value (y_i) and predicted values ($\hat{y}_i$). Here, y_i and $\hat{y}_i$ are continuous values, and N is the number of samples. The squaring ensures the loss is always positive and penalizes larger error more heavily. The average over N normalizes the loss across the dataset.
- **Mean Absolute Error** (MAE) is also used for regression tasks. It computes the average absolute difference between the true value (y_i) and predicted values ($\hat{y}_i$). Like MSE, y_i and $\hat{y}_i$ are continuous, and N is the number of samples. The absolute value ensures the loss is positive and treats all errors equally (linear penalty). The average over N normalizes the loss across the dataset.

2.4.5. Gradient Descent

Gradient Descent is a foundational optimization algorithm used extensively for training deep learning models. It is introduced to minimize a loss (cost) function by iteratively updating the model's parameters (weights and biases) in the direction that reduces the loss between predicted value and true value (Goodfellow et al., 2016).

In deep learning, models (like neural networks) are trained to predict outputs given inputs. The accuracy of predictions is measured using a loss function, such as Mean Squared Error (MSE) or Cross-Entropy. Gradient Descent helps find the set of parameters that minimize this loss.

Mathematically, the weight update rule for a parameter θ is:

$$\theta := \theta - \eta \cdot \nabla_\theta J(\theta), \qquad \textbf{(Eq. 2.5.)}$$

where θ represents the parameters of the model (weights and biases in neural network). η denoted as the learning rate, a small positive number that controls how big each update step is. A smaller value means slower but stable learning. A larger value speed up learning but may overshoot the minimum. $\nabla_\theta J(\theta)$ is the gradient of the loss function $J(\theta)$ with respect to the parameters θ. This tells you the direction and rate of the steepest increase of the function. Since we want to minimize it, we move in the negative direction of the gradient. So, **Eq. 2.5.** means we update the parameters θ by moving them a small step (η) in the direction that reduces the loss.

Common gradient descent algorithms include:

- **Stochastic Gradient Descent (SGD):** This optimization approach adjusts the model's weights incrementally for each individual training sample. Its efficiency stems from processing data one sample at a time, making it highly effective and widely adopted across diverse fields such as machine learning, computer vision, and natural language processing.
- **Adaptive Moment Estimate (Adam)**: An advanced variant of SGD, Adam dynamically calculates an adaptive learning rate for every parameter. This adaptability enhances training efficiency, particularly for large datasets, while requiring less memory compared to traditional methods, thus optimizing performance in complex neural network architectures.
- **Root Mean Squared Propagation (RMSProp)**: This optimization algorithm refines the learning process by scaling the learning rate using an exponentially decaying average of squared gradients. This technique significantly boosts the training performance of neural networks, offering improved convergence and stability, especially in deep learning applications with non-stationary objectives.

Table 2.2. Summary of common Gradient Descent Algorithms.

Algorithm	Update Mechanism	Use case
SGD	Update weights for each training sample	Basic training tasks
Adam	Computes adaptive learning rates for each parameter	Large and complex models
RMSProp	Adjusts learning rate using gradient history	Used for deep networks, especially RNNs

Since this thesis focuses on the Adam optimizer, a detailed exploration is provided to explain its mechanisms and advantages. Adam, short for Adaptive Moment Estimation, is a popular optimization algorithm in deep learning. It builds on traditional stochastic gradient descent (SGD) but introduces two key innovations: the use of momentum (first moment) and adaptive learning rates (second moment). These enhancements help improve the convergence speed and stability of the learning process, especially in complex neural network architectures. Adam dynamically adjusts the learning rate for each parameter during training by leveraging estimates of the first and second moments of the gradient. These moments capture the mean

and uncentered variance, respectively, of the gradients. The following are key aspects of the Adam optimizer:

- **First Moment Estimation (Momentum Component)**: Adam maintains an exponentially weighted moving average of past gradients, which serves as a momentum term. This helps the optimizer smooth out the gradient updates and reduce oscillations, particularly in noisy or sparse gradients.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \qquad \textbf{(Eq. 2.6.)}$$

where m_t is the exponentially decaying average (momentum term), $\beta_1$is decay rate for the moving average of gradient, g_t is gradient at time step t.

- **Second Moment Estimation (Adaptive Learning Rate Component)**: In addition to the first moment, Adam also tracks the exponentially decaying average of the squared gradients, which approximates the second raw moment (variance). This helps adjust the learning rate for each parameter individually, depending on how large or small the gradients are:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \qquad \textbf{(Eq. 2.7.)}$$

where v_t is running average of squared gradients, β_2 is decaying rate for the squared gradients.

- **Bias Correction**: At the beginning of training, both $m_t$and v_t are biased toward zero, especially when the decay rates β_1 and β_2 are close to 1. Adam compensates for this initial bias by computing bias-corrected estimates:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}. \qquad \textbf{(Eq. 2.8.)}$$

This correction ensures that during the early training stages, the moment estimates are accurate and not disproportionately small.

- **Parameter Update**: Finally, the parameter θ (e.g., neural network weights) are updated using the bias-corrected moment estimates. The update rule divides the learning rate by the square root of the second moment estimate to scale the step size, and the first moment guides the direction of the update.

$$\theta_{t+1} = \theta_t - \eta \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t}+\epsilon}, \qquad \textbf{(Eq. 2.9.)}$$

where η is learning rate, $\widehat{m}_t$ is bias-corrected first moments, $\widehat{v}_t$is bias-corrected second moments, ϵ is a small constant to avoid division by zero.

The Adam optimizer brings a range of benefits, particularly through its ability to apply individual learning rates to each parameter. This adaptive behavior enables Adam to tailor updates based on the unique behavior of each parameter, leading to faster convergence and more efficient training. Its adaptability plays a key role in speeding up learning, especially in complex neural networks.

One of Adam's strengths lies in its bias correction mechanism, which helps ensure stable and accurate parameter updates, particularly in the early stages of training when moment estimates can be skewed toward zero. This feature contributes to improved reliability during optimization.

Moreover, Adam is highly effective for working with large-scale datasets and sparse gradients, which makes it an ideal optimizer in many real-world Machine Learning applications, including natural language processing and computer vision tasks.

In practice, the default hyperparameters for Adam are widely accepted and typically include η learning rate of 0.001, $\beta_1$decay rate for the first moment is commonly set to 0.9, β_2 decay rate for the second moment is usually assigned to 0.009, ϵ small constant to prevent division by zero is typically set to 10^{-8}.

2.5. Reinforcement Learning

Reinforcement Learning (RL) is a type of machine learning paradigm where learning is directed toward achieving a specific objective through continuous interaction with a dynamic environment. In this approach, an intelligent agent learns optimal behavior by exploring an initially unknown environment, often through a method resembling trial and error. Much like how a young child learns from their surroundings by experimenting with different actions and observing the results the RL agent gradually builds its knowledge (Naeem et al., 2020).

The agent receives evaluative feedback in the form of rewards (positive reinforcement) or penalties (negative reinforcement), which help guide its decisions. Over time, the agent uses this feedback to develop a strategy or policy that maximizes cumulative reward. This iterative learning process enables the agent to make increasingly informed and effective choices, adapting its behavior to navigate complex environments. Reinforcement learning is especially

valuable in domains where explicit supervision is unavailable and decisions must be made sequentially, such as robotics, game playing, autonomous driving, and real-time decision systems.

2.5.1. Markov Decision Processes

A Markov Decision Process (MDP) is a mathematical framework used to describe the environment in reinforcement learning (Sutton & Barto, 2018). It provides a formalized way to model decision-making situations where outcomes are partly random and partly under the control of an agent. MDPs are essential because they define the structure of the environment that the agent interacts with.

MDP is based on Markov Property which does not consider previous information or history and only the current information. Prediction of the next state doesn't depend on the previous states. Things or physics of the given environment are stationary, and rules do not change. Playing chess is the best example for MDP where the rules are not changed and you no need to remember your previous moves to play the next move (Naeem et al., 2020).

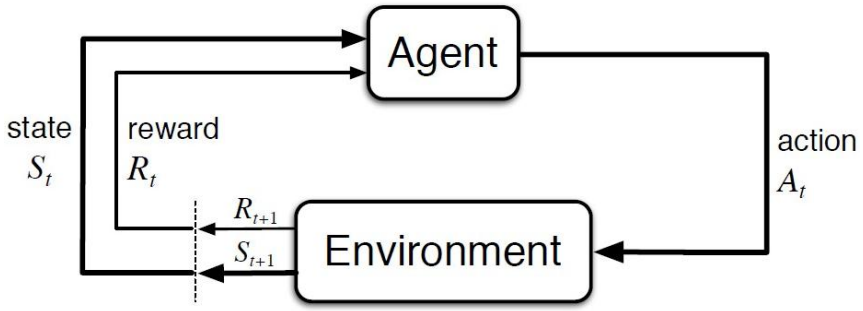

Figure 2.8. The agent-environment interaction in an MDP (Sutton & Barto, 2018).

Figure 2.8. illustrates the agent-environment interaction in a Markov Decision Process (MDP), a mathematical framework used in reinforcement learning. The Agent takes actions and makes decisions based on the current state of the environment. The Environment provides the agent with a state (S_t) and a reward (R_t) at each time step t. The agent selects an action (A_t) and sends it to the environment. In response, the environment transitions to a new state (S_{t+1}) and provides a new reward (R_{t+1}) based on the action taken by the agent.

This process repeats, forming a loop where the agent learns to maximize cumulative rewards over time by choosing optimal actions based on the states it observes. The Markov Properties implies that the next state and reward depend only on the current state and action, not on the previous history.

- **A Markov Decision Process** is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. $\mathcal{S}$ (The set of States) is the set of all possible states the agent can be in. For example, in a robot navigation task, each location the robot can occupy is a state. $\mathcal{A}$ (The set of Actions) is the set of all possible actions which the agent can take. Actions can be discrete action (move left or move right) or continuous action (change speed or direction). $\mathcal{P}(s'|s, a)$ (Transition probability matrix) is the probability of transitioning to state s' when the agent takes action a in state s. This models the stochastic nature of the environment. $\mathcal{R}(s, a)$ (Reward function) is the reward the agent receives after taking action a in state s. It provides feedback to the agent about the quality of an action in a given state. γ (Discount Factor) is A value between 0 and 1 that determines the importance of future rewards. A value closer to 0 makes the agent short-sighted (preferring immediate rewards), while a value closer to 1 makes it far-sighted (valuing future rewards more).
- **Markov Property**: A key feature of MDPs is the Markov property, which states that the future state depends only on the current state and action, not on the sequence of past states. This makes the process memoryless:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \dots, S_t]. \qquad \textbf{(Eq. 2.10.)}$$

- **The policy $\boldsymbol{\pi}$** determines the behavior of our agent, who will take actions $a_t \sim \pi(\cdot|s_t)$. Policies can be derived from an action-value function or can be explicitly parameterized and denoted by π_θ. They can be deterministic, in which case they are sometimes denoted by μ_θ, with $a_t = \mu_\theta(s_t)$.
- **The trajectory $\boldsymbol{\tau}$:** $\tau = (s_0, a_0, s_1, \dots)$ is a sequence of states and actions in the world, with $s_{t+1} \sim P(\cdot|s_t, a_t)$. It is sampled from π if $a_t \sim \pi(\cdot|s_t)$ for each t. Trajectories are also called episodes.
- **The return $\boldsymbol{G_t}$** is the cumulative reward over a trajectory and is the quantity to be maximized by our agent. Return can refer to the finite-horizontal undiscounted return $G_t = R_{t+1} + R_{t+2} + \cdots + R_T$ or the infinite-horizon discounted return $G_t = R_{t+1} + R_{t+2} + \cdots + R_\infty = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.
- **The on-policy value function** is given by:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]. \qquad \textbf{(Eq. 2.11.)}$$

- **The on-policy action-value function** is given by:

$$q_\pi(s,a) = \mathbb{E}_\pi\,[G_t|S_t = s, A_t = a]. \qquad \textbf{(Eq. 2.12.)}$$

The Bellman equations (Bellman , 1957) are essential in reinforcement learning for determining the optimal policy that maximizes expected return. Below are the Bellman equations for the value function $v_\pi(s)$ and the action-value function $q_\pi(s,a)$, as well as their optimal counterparts $v_*(s)$ and $q_*(s,a)$.

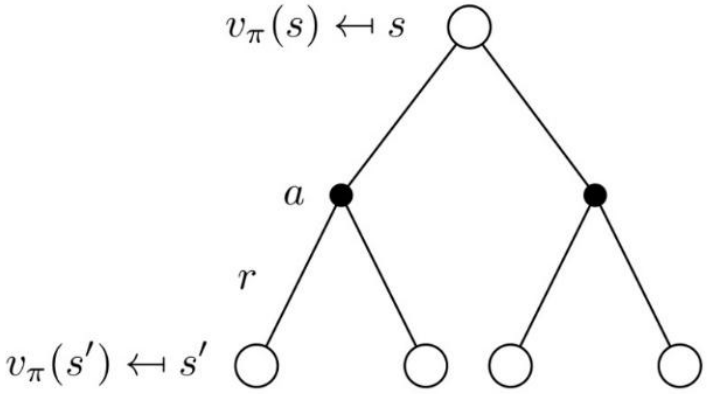

Figure 2.9. Decision tree related to the state-value function $v_\pi(s)$.

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)\big(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')\big), \qquad \textbf{(Eq. 2.13.)}$$

where the **Eq. 2.13.** represents the state-value function $v_\pi(s)$ for a given policy π. It defined the expected return for starting in state s, taking action a according to policy π, and then follow policy π. The term $\mathcal{R}_s^a$ represents the reward received after transitioning from state s to successor state s' by taking action a, and γ is the discount factor.

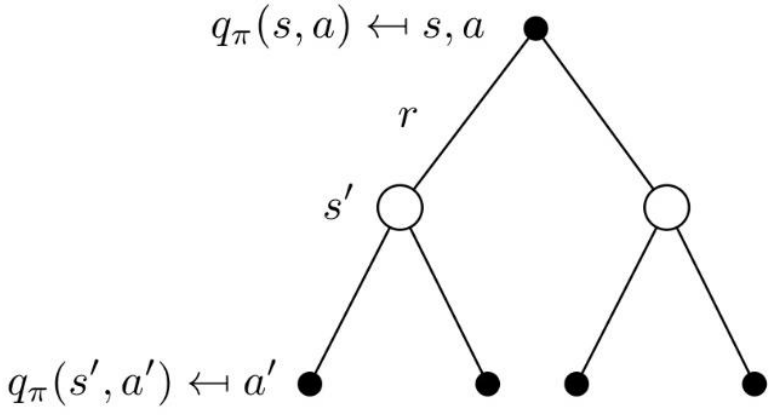

Figure 2.10. Decision tree related to the action-value function $q_\pi(s,a)$.

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a'), \qquad \textbf{(Eq. 2.14.)}$$

where the **Eq. 2.14.** defines the action-value function $q_\pi(s,a)$ for a given policy π. It represents the expected return for starting in state s, taking action a, and then follow policy π. The equation considers the immediate reward $\mathcal{R}_s^a$ and the expected return of the successor state s' and action a' taken according to policy π.

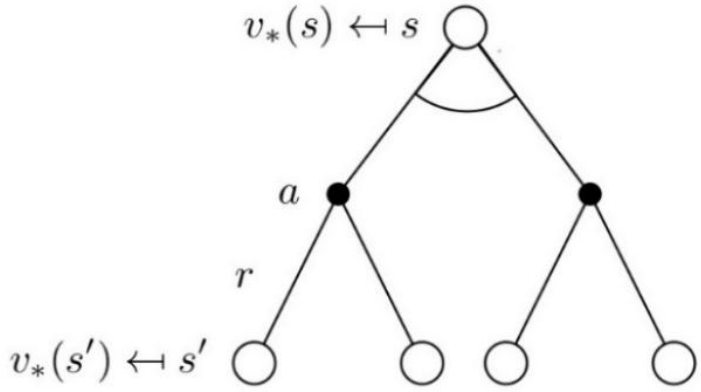

Figure 2.11. Decision tree related to the optimal state-value function $v_*(s)$.

$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s'), \qquad \textbf{(Eq. 2.15.)}$$

where the **Eq. 2.15.** describes the optimal value function $v_*(s)$. It represents the maximum expected return obtainable from state s by taking the best possible action a and then follow the optimal policy. The optimal value function considers the immediate reward and the maximum expected return from the next state s'.

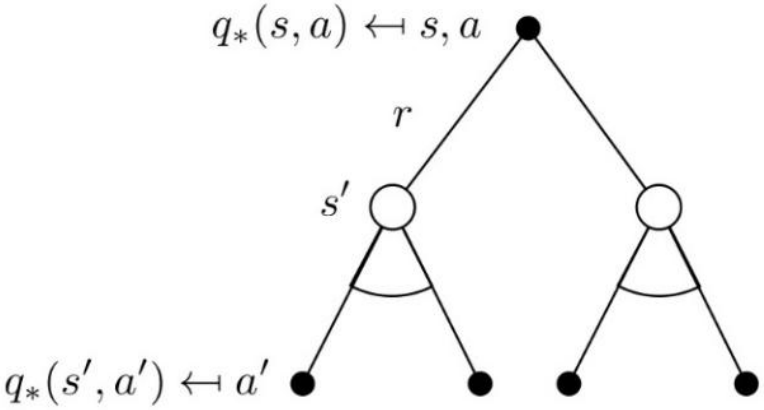

Figure 2.12. Decision tree related to the optimal action-value function $q_*(s,a)$.

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_a q_*(s',a'), \qquad \textbf{(Eq. 2.16.)}$$

where the **Eq. 2.16.** defines the optimal action-value function $q_*(s,a)$. It represents the expected return for starting in state s, taking action a, and then follow the optimal policy. The optimal action-value function considers the immediate reward and the maximum expected return form the next state s' and action a'.

2.5.2. Dynamic Programming

Dynamic programming (DP) is a methodology for solving complicated problems by decomposing them into smaller. In context of Reinforcement Learning, it is used for computing value functions and discovering optimal policies (Bellman , 1957). Even though, this method works only in the case of knowing the transition probability which is not convenient for applying in real world.

The two key characteristics of Dynamic Programming are optimal substructure and overlapping subproblems, both of which are fulfilled by MDP (Naeem et al., 2020).

Policy Evaluation: Policy evaluation is the process of computing the value function $v_\pi(s)$ for a given policy π. The value function estimates how good it is to be in a state s under policy π. The goal of policy evaluation is to find out how good a fixed policy is.

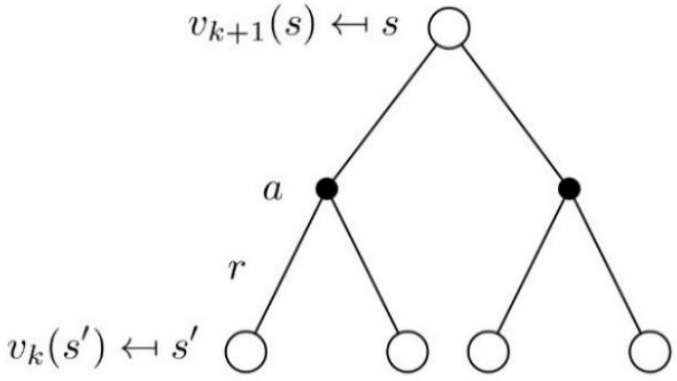

Figure 2.13. Policy evaluation process.

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right),$$
$$\boldsymbol{v}^{k+1} = \mathbf{max}\boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \boldsymbol{v}^k. \qquad \textbf{(Eq. 2.17.)}$$

Policy Iteration: Policy iteration is a method to find the optimal policy π^* by alternating between two steps:

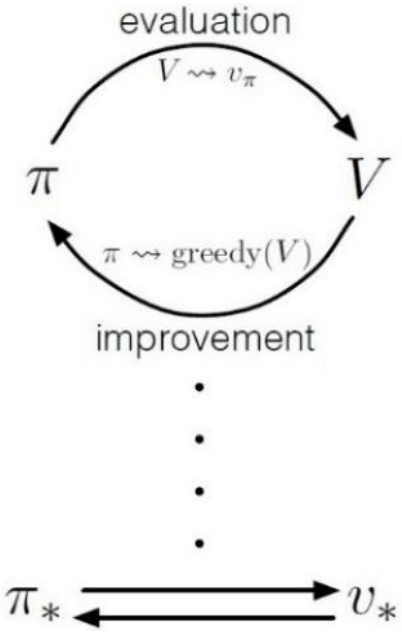

Figure 2.14. Cyclic relationship between policy evaluation and improvement (Sutton & Barto, 2018).

- Policy Evaluation: Evaluate the current policy to compute v_π.
- Policy Improvement: Improve the policy by choosing actions that maximize the expected value.

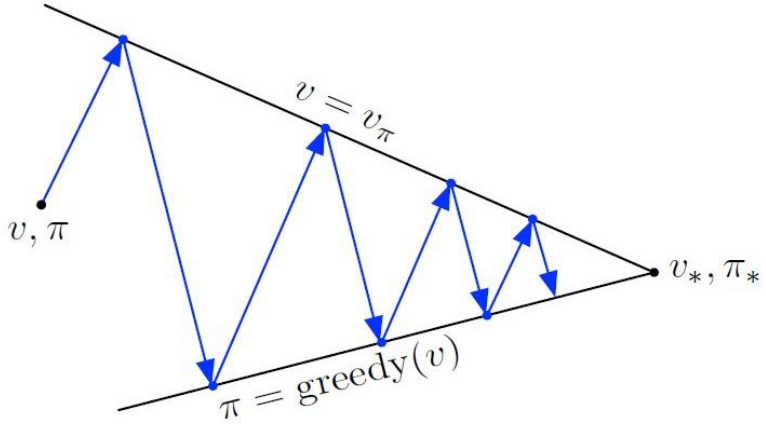

Figure 2.15. Process of policy iteration (Sutton & Barto, 2018).

Value Iteration: Value iteration combines policy evaluation and improvement into a single step. It directly updates the value function using the Bellman Optimality Equation, aiming to find the optimal value function $v_*(s)$.

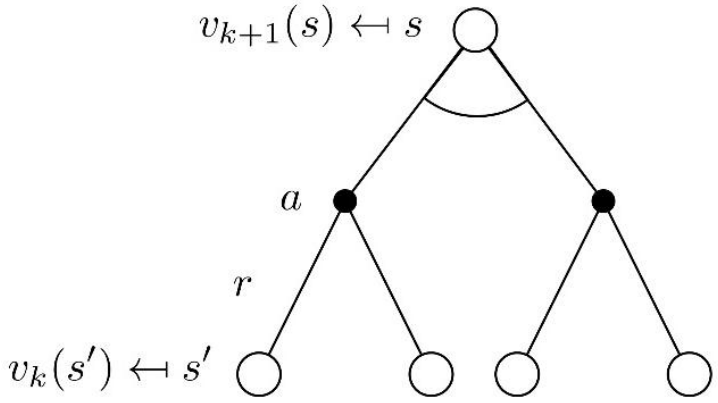

Figure 2.16. Process of value iteration.

$$v_{k+1}(s) = \max_{a \epsilon \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right), \qquad \textbf{(Eq. 2.18.)}$$
$$\boldsymbol{v}^{k+1} = \max_{a \epsilon \mathcal{A}} (\boldsymbol{\mathcal{R}}^{\pi} + \gamma \boldsymbol{\mathcal{P}}^{\pi} \boldsymbol{v}^k).$$

Once $v_*(s)$ is found, the optimal policy π^* is obtained by selecting the action that gives the maximum expected value. The key idea is updating values without fixing a policy and extract the policy at the end.

2.5.3. Monte Carlo Learning

Monte Carlo (MC) methods in RL estimate value functions by averaging returns from sampled episodes. These methods are particularly effective when the model of the environment is unknown or difficult to obtain (Sutton & Barto, 2018). Monte Carlo techniques estimate the value of a state by computing the mean return obtained from that state across multiple complete episodes.

The return G_t from a state s_t is defined as the total discounted reward accumulated over an episode:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T. \qquad \textbf{(Eq. 2.19.)}$$

The state-value function $V(s)$ is then estimated using the average return across all sampled episodes that begin in state s:

$$V(s) = \frac{1}{N}\sum_{i=1}^{N} R_t^i, \qquad \textbf{(Eq. 2.20.)}$$

Where G_t^i is the return obtained from the i-th episode and N is the total number of episodes sampled.

Alternatively, incremental updates can be used to avoid storing all past returns. The update rule is given by:

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)], \qquad \textbf{(Eq. 2.21.)}$$

where $\alpha \in (0,1]$ is the learning rate. This form of update is memory efficient and suitable for online learning.

Similarly, for the action-value function $Q(s_t, a_t)$, Monte Carlo methods estimate the value of taking action a in state s, and then following the policy π:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[G_t - Q(s_t, a_t)]. \qquad \textbf{(Eq. 2.22.)}$$

The policy π is then improved based on the updated actin-value function. A common approach is to make the policy greedy with respect to the current estimate of Q, thereby improving decision making over time.

One limitation of Monte Carlo methods is that they require complete episodes before performing updates. This can be inefficient in environments where episodes are long or may not terminate. Despite this, MC techniques remain a fundamental approach in model-free RL, particularly in episodic environments.

2.5.4. Temporal Difference Learning

Temporal Difference (TD) learning is a reinforcement learning approach that integrates concepts from both Dynamic Programming (DP) and Monte Carlo (MC) methods. It updates value estimates using subsequent state predictions, rather than waiting for the entire episode to finish (Sutton & Barto, 2018). This characteristic makes TD learning particularly well-suited for online and incremental learning scenarios.

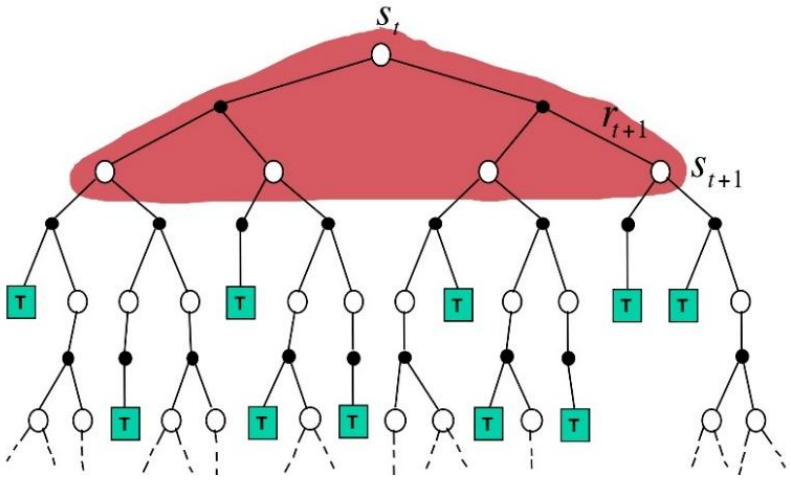

Figure 2.17. Dynamic Programming backup diagram.

Figure 2.17. Visually represents how DP performs updates in RL using full backups. Red-shade region (top part of the tree) represents the backup target of state s_t, showing all the one-step transition from s_t to all possible next state s_{t+1}, and the transitions that follow those. This region shows how DP relies on a complete model of the environment (i.e., transition probabilities and rewards) to compute expected value of future returns. The use of "full tree expansion" allow DP to consider all possible outcomes weighted by their probabilities.

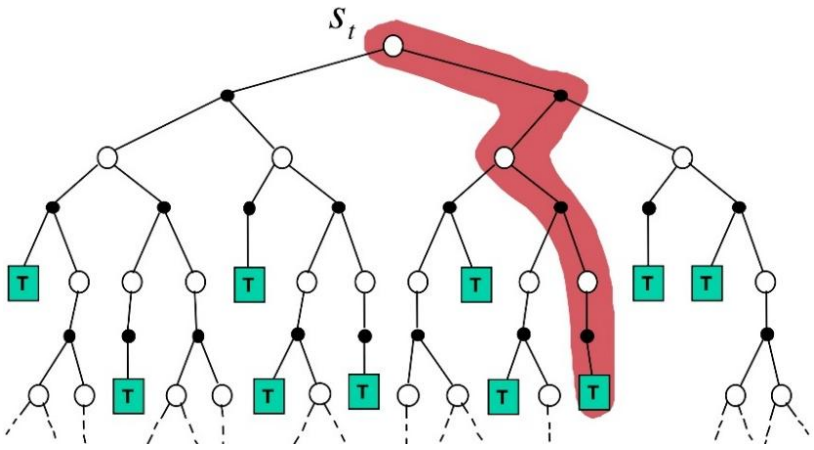

Figure 2.18. Monte Carlo Learning backup diagram.

Figure 2.18. visually illustrates how MC methods perform updates in RL based on sampled episodes rather than full knowledge of the environment. The red path shows one specific trajectory taken from the current state s_t to the terminal state (green T). Unlike DP, which uses a full expectation overall possible paths, MC method samples one complete path and use

that to update the value of s_t. Only visited states and actions along that path contribute to the backup

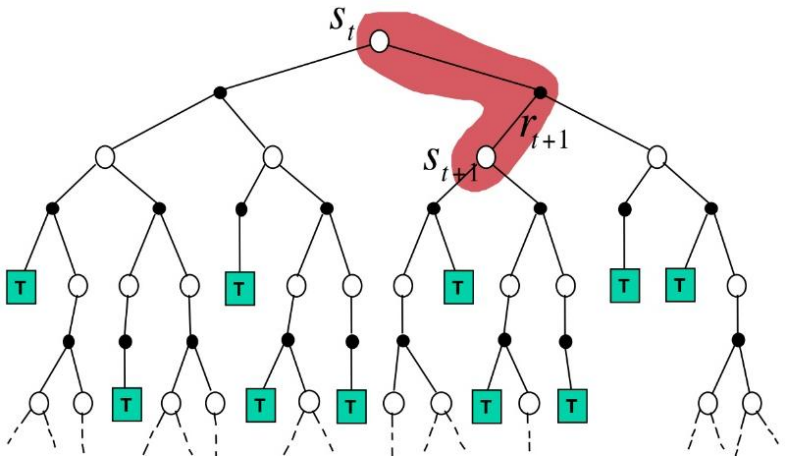

Figure 2.19. Temporal Different Learning backup diagram.

Figure 2.19. Visually illustrates how TD methods perform updates in RL based on partial episode rather than waiting for the full episode to finish. The red path illustrates a transition from the current state s_t to a successor state s_{t+1}, which is different from DP method diagram transitioning from the current state s_t to all possible successor state s_{t+1}. The backup process in TD learning involves updating the value of the current state s_t based on the immediate reward R_{t+1} and the estimated value of the next state s_{t+1}.

The TD update rules for the state-value function $V(s)$ is defined as:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma\, V(s_{t+1}) - V(s_t)], \qquad \textbf{(Eq. 2.23.)}$$

where α is the learning rate, r_{t+1} is the immediate reward after transitioning from current state s_t to successor state s_{t+1}, γ is the discount factor. The term $r_{t+1} + \gamma\, V(s_{t+1}) - V(s_t)$ represents the TD error δ_t, which quantifies the discrepancy between the predicted value and the observed reward plus estimated future return:

$$\delta_t = r_{t+1} + \gamma\, V(s_{t+1}) - V(s_t). \qquad \textbf{(Eq. 2.24.)}$$

This error is minimized over time through updates, allowing the agent to refine its predictions of state values.

TD learning can also be extended to learn the action-value function $Q(s_t, a_t)$, which estimates the expected return for taking action a in state s:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \qquad \textbf{(Eq. 2.25.)}$$

Here, the TD error for the action-value function is:

$$\delta_t = r_{t+1} + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t). \qquad \textbf{(Eq. 2.26.)}$$

Popular TD-based algorithms include SARSA (State-Action-Reward-State-Action) and Q-learning:

- SARSA is an on-policy method that updates the value of the current action based on the action actually taken in the next state:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \qquad \textbf{(Eq. 2.27.)}$$

- Q-learning, in contrast, is an off-policy method that updates using the maximum estimated action value in the next state:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \qquad \textbf{(Eq. 2.28.)}$$

These TD learning approaches enable agents to iteratively improve their policies by learning from raw experience, refining predictions at each step. Although TD methods update more frequently than MC methods, they still rely on sampled transitions, making them both powerful and efficient for real-world learning tasks.

2.6. Deep Reinforcement Learning

Modern Reinforcement learning techniques leverage deep learning to handle high-dimensional state-spaces, enabling more complex and scalable solution as shown in **Figure 2.20.**, the agent, represented by a neural network, takes actions A_t based on the current state S_t. It then receives the next state S_{t+1} and a reward r_t from the environment, using this feedback to improve its policy. These methods incorporate neural networks as function approximators to learn policies and value function, allowing agents to make more sophisticated decisions in dynamic environment (Sutton & Barto, 2018).

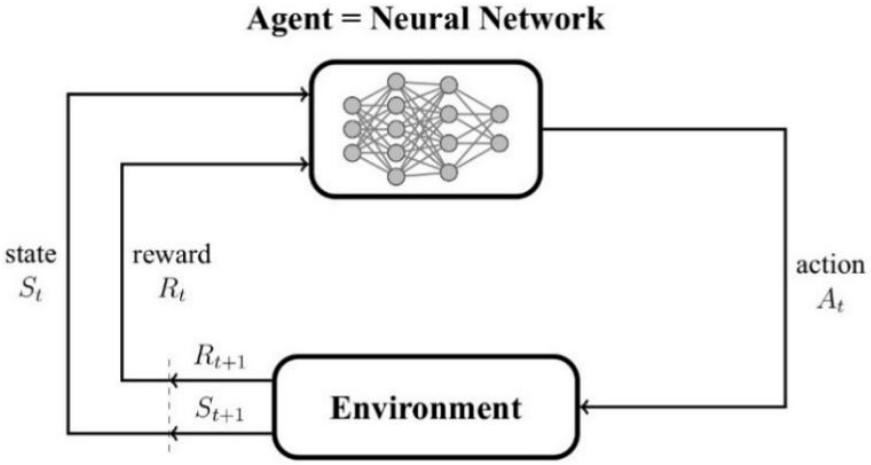

Figure 2.20. Interaction between Agent and Environment in Deep RL.

2.6.1. Function Approximation in RL

In RL, the goal of an agent is to learn an optimal policy that maximizes cumulative rewards by interacting with an environment. A critical component of this process is estimating value functions such as the state-value function $V(s)$ or the action-value function $Q(s,a)$ which help guide the agent's decisions.

In the simple environments with small number of states and actions, these value functions can be stored in a table. However, in most practical applications including robotics, video games, and autonomous systems the state and action spaces are too large or even continuous. In such cases, it combines infeasible to represent the value functions using a table. This is where function approximation becomes essential (Sutton & Barto, 2018).

Function approximation refers to the use parameterized mathematical functions to estimate value functions or policies. Instead of storing a separate value for every state or state-action pair, the agent learns a generalized function that can predict values for unseen or rarely visited states based on patterns learned during training.

There are two types of function approximator:

- Linear function approximation: uses a weighted sum of features extracted from the state or state-action pair.

$$\hat{V}(s,\theta) \approx \theta^T \phi(s), \qquad \textbf{(Eq. 2.29.)}$$

where $\phi(s)$ is a feature vector and θ are learnable weights.

❖ Nonlinear function approximation: uses deep neural networks to model complex, non-linear mappings.

$$\hat{Q}(s, \mathrm{a}; \theta) \approx Q^{*}(\mathrm{s}, \mathrm{a}). \qquad \textbf{(Eq. 2.30.)}$$

Function approximation plays a key role in several modern RL algorithms including DQN, DDPG, and TD3. DQN is a value-based methods that replace Q-table with Q-networks. It uses deep neural networks to estimate action-functions $Q(\mathrm{s}, \mathrm{a})$. DDPG and TD3 are actor-critic methods that use two networks, actor for updating policy, and critic for updating value function.

2.6.2. Deep Q-Network

Deep Q-Network is an extension of traditional Q-Learning that uses deep neural networks to approximate the Q-value function. This allows RL to be applied to environments with very large or high-dimensional state spaces, such as raw image inputs from video games (Naeem et al., 2020). DQN gained widespread attention after achieving superhuman performance in Atari games, where traditional table-based Q-learning fails due to the massive number of possible states.

Even though DQN solves problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces (Lillicrap et al., 2016).

The central idea behind DQN is to use a neural network to estimate the action-value function, denoted as $Q(s, a; \theta)$, where, s is the state, a is the action, and θ represents the parameters (weights) of the neural network. The goal is to train this network so that it accurately predicts Q-value the expected future reward for taking action a and in state s.

The network is trained by minimizing the difference between the predicted Q-value from the network and the target Q-value (based on the reward received and estimated future rewards).

The Q-value update follows this rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right], \qquad \textbf{(Eq. 2.31.)}$$

where α is the learning rate, γ is the discount factor, r_{t+1} is the immediate reward, and $\max_{a'} Q(s_{t+1}, a')$ estimates the best future return.

In DQN, the Q-function is approximated using a deep neural network instead of a table. This network outputs Q-values for each possible action, given the input state.

To improve learning stability, DQN uses a target network which is a copy of the Q-network with parameters θ^-. This network is updated less frequently and is used to compute the target Q-values.

To train the network, we define a loss function that measures the squared error between the predicted Q-value and the target:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s_t,a_t,r_{t+1},s_{t+1})}\left[\left(r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-) - Q(s_t, a_t; \theta)\right)^2\right], \qquad \textbf{(Eq. 2.32.)}$$

where $\mathcal{D}$ is a replay buffer containing past experiences $(s_t, a_t, r_{t+1}, s_{t+1})$.

The target Q-value is:

$$y_t = r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a'; \theta^-). \qquad \textbf{(Eq. 2.33.)}$$

The Q-network parameters θ are updated by minimizing the loss using gradient descent:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta). \qquad \textbf{(Eq. 2.34.)}$$

To improve the stability and performance of training, Deep Q-Networks (DQN) implement two key techniques: experience replay and the use of a target network.

Table 2.3. Deep Q-Learning pseudo code (Ojeda, 2025).

Algorithm 1: Deep Q-Learning

Initialize replay buffer $\mathcal{D}$ with capacity N
Initialize $Q(s_t, a_t)$ action value function with random weights
for episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1,\ T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observer reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_i = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_a Q(\phi_{j+1}, a'; \theta) & \text{for nonterminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $\left(y_i - Q(\phi_j, a_j; \theta)\right)^2$
  end for
end for

- ❖ Experience replay: Instead of learning from consecutive experiences, the agent stores each transition $(s_t, a_t, R_{t+1}, s_{t+1})$ in a replay buffer (a type of memory). During training, random mini-batches are sampled from the buffer. This randomization breaks the strong correlation between sequential experiences to be reused in multiple updates, significantly improving data efficiency and learning performance.
- ❖ Target Network: To reduce oscillations and divergence during training, DQN uses a separate, more stable neural network called the target network. This network has parameter θ^-, which are periodically copied from the main Q-network parameters θ, rather than being updated at every step. By using this slowly-updated target for calculating the TD target y_t, DQN ensures the target value is more consistent across updates, leading to smoother and more reliable learning.

The DQN algorithm can be described by **Table 2.3.**. Deep Q-Learning represents a significant advancement in RL, enabling agents to learn and make decisions in high-dimensional and complex environments effectively.

2.6.3. Policy Gradient Methods

Policy Gradient Methods are the methods learning parameterized policy that can choose actions directly without relying on value function (Sutton & Barto, 2018).

Policy gradient methods are a class of RL algorithms that aim to directly optimize the policy by computing the gradient of the expected return with respect to the parameters of a parameterized policy. These methods are particularly suitable for environments with continuous or high-dimensional action spaces, where value-based methods like Q-learning struggle.

There are two types of policy gradient methods namely, Stochastic Policy Gradient (SPG) method and Deterministic Policy Gradient (DPG) method. Both are used for high dimensional and continuous action spaces, but they approach it differently. For SPG method, policy outputs probability distribution over actions, while DPG method, policy directly maps states to specific actions. This implies that DPG approach can be computed more efficiently and with low variance compared to the SPG method (Silver et al., 2014).

- ❖ **Stochastic Policy Gradient (SPG)**

In policy-based methods, the policy π is modeled as a probabilistic function, parameterized by θ, which maps states s to a distribution over possible action:

$$\pi_\theta(\mathrm{a}|s) \approx \pi(s). \qquad \textbf{(Eq. 2.35.)}$$

The goal is to find the policy parameters θ that maximize the expected return $J(\theta)$, defined as the expected cumulative reward over trajectories τ:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[G(\tau)]. \qquad \textbf{(Eq. 2.36.)}$$

To optimize $J(\theta)$, the PG theorem provides the gradient of the expected return with respect to the policy parameters:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s)\, Q\,(s,a)]. \qquad \textbf{(Eq. 2.37.)}$$

This gradient expression is used to update the policy using gradient ascent. The action-value function $Q(s,a)$ can be estimated using the reward-to-go:

$$G_t = \sum_{k=0}^{T-t} \gamma^k R_{t+k}, \qquad \textbf{(Eq. 2.38.)}$$

this reward-to-go helps reduce variance in the gradient estimates and improves learning stability.

❖ **Deterministic Policy Gradient (DPG)**

While stochastic policies output a distribution over actions, deterministic policy gradient methods aim to learn a deterministic mapping from states to actions:

$$a = \mu_\theta(s), \qquad \textbf{(Eq. 2.39.)}$$

here, the policy μ_θ is directly parameterized by θ, and always outputs the same action for a given state. This approach is especially efficient for continuous action spaces, as it avoids the need to sample actions from a distribution.

The objective remains to maximize the expected return:

$$J(\theta) = \mathbb{E}_{s\sim\rho^\mu}\big[Q^\mu\big(s,\mu_\theta(s)\big)\big]. \qquad \textbf{(Eq. 2.40.)}$$

The DPG theorem provides the gradient of the objective:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\mu}\big[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\big], \qquad \textbf{(Eq. 2.41.)}$$

this formulation enables gradient-based optimization using samples from the state distribution ρ^μ, which is influenced by the policy's interaction with the environment.

❖ **Practical Implementation**

In practice, both the policy $\mu_\theta(s)$ and the value function $Q^\mu(s,a)$ are often approximated using neural networks, which allows the agent to generalize over high-dimensional, continuous state and action spaces. The parameter update for the policy is typically performed using a learning rate α as follows:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta), \qquad \textbf{(Eq. 2.42.)}$$

this iterative update improves the policy by ascending the gradient of the expected return.

Approached that approximate both the policy and the value function are typically known as actor-critic methods, where the 'actor' represents the learned policy and the 'critic' denotes the learned value function.

2.6.4. Deep Deterministic Policy Gradient (DDPG)

A major challenge of learning in continuous action spaces is exploration. DDPG is an off-policy algorithm that splits exploration problem independently from the learning algorithm (Lillicrap et al., 2016). This means that DDPG separate the exploration policy from the learning policy. This separation allows the algorithm to explore the environment independently of the learning process, which is a key advantage. In contrast, on-policy methods (like PPO) tie exploration and learning together, which can make the exploration less flexible.

The Deep Deterministic Policy Gradient (DDPG) algorithm is an advanced model-free, off-policy actor-critic method specifically designed for environments with continuous action spaces, where traditional discrete-action algorithms like DQNs fall short. DDPG addresses this limitation by introducing deterministic policies, allowing the agent to select precise, real-valued actions rather than choosing from a fixed set.

Building upon the foundational principles of DQN, DDPG eliminates the constraint of discrete actions by incorporating two separate neural networks: an actor network and a critic network. The actor network is responsible for learning a deterministic policy that maps each state directly to a specific action. In other words, it decides what action to take based on the current state of the environment. Meanwhile, the critic network serves as an evaluator, estimating the Q-value which is a measure of the expected cumulative reward for the state-action pairs generated by the actor. The combination allows DDPG to efficiently learn control policies for high-dimensional and continuous control tasks, making it particularly useful in robotics and other real-work applications.

The DDPG algorithm is composed of two neural networks:

- Actor Network is parameterized by θ^{μ} and denoted as $\mu_{\theta}(s|\theta^{\mu})$. It takes the current state s and outputs a deterministic action $a = \mu(s|\theta^{\mu})$. The goal of the actor is to maximize the objective function $J(\theta^{\mu})$, which is the estimated value of taking the actor's current actions:

$$J(\theta^{\mu}) = \mathbb{E}_{s\sim\rho^{\beta}}\big[\mathrm{Q}\big(s, \mu(s|\theta^{\mu})\big)\big]. \qquad \textbf{(Eq. 2.43.)}$$

- Critic Network is parameterized by θ^{Q} and denoted as $Q(s, a|\theta^{Q})$. It estimates the action-value function (Q-value), which tells how good a state-action pair is.

The actor is updated using the deterministic policy gradient theorem, which computes how much the policy parameters should change to improve performance.

$$\nabla_{\theta^{\mu}} J(\theta^{\mu}) = \mathbb{E}_{s\sim\rho^{\beta}}\left[\nabla_{a} Q(s, a|\theta^{Q})|_{a=\mu(s|\theta^{\mu})} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})\right], \qquad \textbf{(Eq. 2.44.)}$$

this formular tells us how the actor should adjust its parameters based on how the critic scores its current actions.

The critic is trained using a loss function that minimizes the difference between the predicted Q-value and the target Q-value based on the Bellman equation:

$$\mathcal{L}(\theta^{Q}) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}[(r + \gamma Q(s', \mu(s'|\theta^{\mu})|\theta^{Q}) - Q(s, a|\theta^{Q}))^2], \qquad \textbf{(Eq. 2.45.)}$$

here, $\mathcal{D}$ is the replay buffer that stores past experiences $(s_t, a_t, r_{t+1}, s_{t+1})$.

The exploration is achieved by adding noise N to the actions output by the actor policy $\mu_{\theta}(s|\theta^{\mu})$.

$$\mu'(s_t) = \mu(s_t|\theta^{\mu}) + N, \qquad \textbf{(Eq. 2.46.)}$$

where $\mu'(s_t)$ is the modified (exploration) policy that the agent uses to take actions, $\mu(s_t|\theta^{\mu})$ is the original deterministic policy output by the actor networks for state s_t, N is the noise term added to encourage exploration by introducing randomness in the actions.

Adding noise ensures the agent doesn't always take the same action for a given state, which could lead to getting stuck in suboptimal behaviors. This exploration strategy allows DDPG to try new actions and discover better policies over time, while the learning process (updating the actor and critic networks) focuses on optimizing the policy based on the experience collected.

Table 2.4. DDPG algorithm pseudo code (Lillicrap et al., 2016).

Algorithm 2: DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $\mathcal{D}$
for episode = 1, M **do**
  Initialize a random process N for action exploration
  Receive initial observation state s_1
  for $t = 1,\ T$ **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$
    Sample minibatch of N transition $(s_j, a_j, r_{j+1}, s_{j+1})$ from $\mathcal{D}$
    Set $y_j = r_j + \gamma Q'(s_{j+1}, \mu'(s_{j+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_j \left(y_j - Q\left(s_j, a_j|\theta^Q\right)\right)^2$
    Update the actor policy using the sampled policy gradient:

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

    Update the target networks:

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

  end for
end for

The DDPG algorithm is illustrated in **Table 2.4.**, this algorithm ensures that the policy and value function are iteratively enhanced via actor-critic updates, making it highly effective for continuous control tasks.

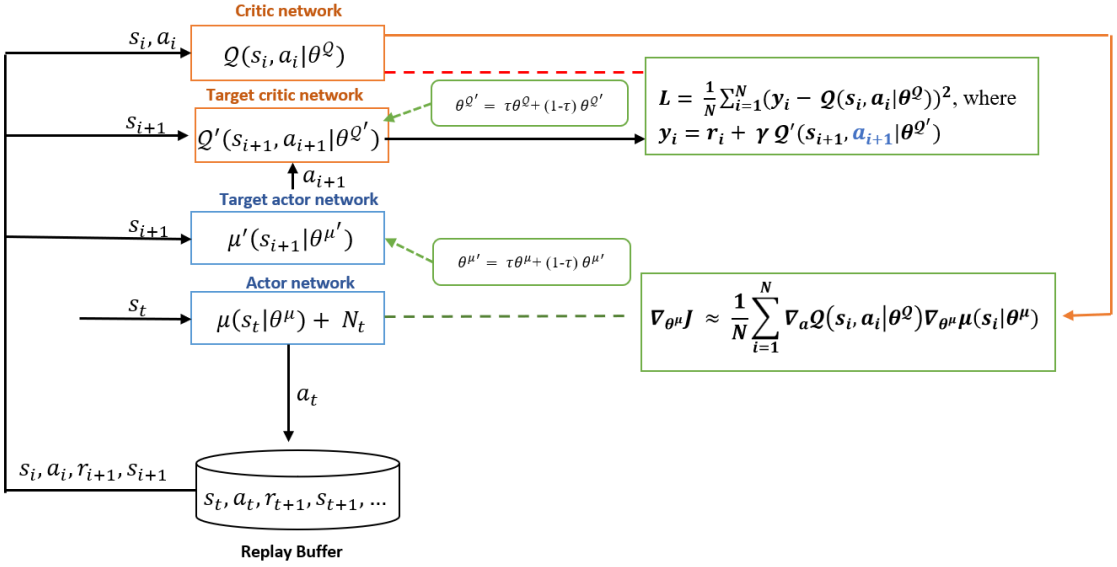

Figure 2.21. TD3 algorithm architecture.

2.6.5. Twin-Delayed Deep Deterministic Policy Gradient (TD3)

The TD3 algorithm enhances the performance of the DDPG by tackling issues related to overestimation bias and training instability. It employs a pair of critics to improve the accuracy of Q-value predictions, introduces delayed policy updates to enhance learning stability, and applies target policy smoothing to reduce variance in the learning process (Fujimoto et al., 2018).

As a result, TD3 demonstrates superior performance in continuous control tasks compared to its predecessors, such as DDPG. TD3 introduces several key innovations including clipped double Q-learning, delayed policy updates, and target policy smoothing which effectively address the overestimation bias and instability issues commonly encountered in actor-critic methods like DDPG.

Table 2.5. TD3 algorithm pseudo code (Fujimoto et al., 2018).

Algorithm 3: TD3 algorithm

Initialize critic networks $Q(s,a|\theta_1^Q)$, $Q(s,a|\theta_2^Q)$, and actor network $\mu(s|\theta^\mu)$ with random parameters θ_1^Q, θ_2^Q, θ^μ

Initialize target network $Q'\left(s,a\middle|\theta_1^{Q'}\right)$, $Q'\left(s,a\middle|\theta_2^{Q'}\right)$ and μ' with weights$\theta_1^{Q'} \leftarrow \theta_1^Q$, $\theta_2^{Q'} \leftarrow \theta_2^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer $\mathcal{D}$

for $t = 1$ **to** T **do**

Select action $a = \mu(s|\theta^\mu) + \epsilon$ according to the current policy and exploration noise

$\epsilon \sim \mathcal{N}(0,\sigma)$ and observe reward r and new state s

Store transition tuple (s,a,r,s') in $\mathcal{D}$

Sample mini-batch of N transition (s,a,r,s') from $\mathcal{D}$

$\tilde{a} \leftarrow \mu'(s'|\theta^{\mu'}) + \epsilon$, $\epsilon \sim \text{clip}(\mathcal{N}(0,\tilde{\sigma}), -c, c)$

$y \leftarrow r + \gamma \min_{i=1,2} Q'\left(s',\tilde{a}\middle|\theta_i^{Q'}\right)$

Update critics $\theta_i^Q \leftarrow \text{argmin}_{\theta^{Q_i}} N^{-1} \sum \left(y - \left(Q(s,a|\theta_i^Q)\right)\right)^2$

if t mod d **then**

Update θ^μ by the deterministic policy gradient:

$\nabla_{\theta^\mu} J(\theta^\mu) = N^{-1} \sum \nabla_a Q(s,a|\theta_1^Q)|_{a=\mu(s|\theta^\mu)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)$

Update target networks:

$\theta_i^{Q'} \leftarrow \tau\theta_i^Q + (1-\tau)\theta_i^{Q'}$

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

end if

end if

The use of two critic networks in TD3 helps mitigate the risk of overestimating Q-values by taking the minimum of the two value estimates when updating the actor, resulting in

more conservative and reliable policy updates. Meanwhile, delayed updates to the actor network prevent premature convergence and promote more stable learning dynamics. Additionally, by adding small noise to the target action (target policy smoothing), TD3 discourages exploitation of narrow peaks in the Q-function, thus enhancing generalization and robustness. Thanks to these architectural improvements, TD3 is not only more sample-efficient but also more resilient to hyperparameter sensitivity and environmental noise, making it highly applicable to real-world problems. Its robustness and ability to learn stable, high-precision policies make it particularly suitable for robotic control, autonomous navigation, manipulation tasks, and other domains where decision-making in high-dimensional, continuous action spaces is required.

In TD3, three key techniques are used to mitigate overestimate bias and stabilize training:

- Clipped Double Q-Learning: TD3 uses two critic networks $Q\left(s,a\middle|\theta_1^Q\right)$ and $Q\left(s,a\middle|\theta_2^Q\right)$ to compute two estimates of the Q-value. The minimum of these two values is used to form the target Q-value:

$$y = r + \gamma \min\left(Q'\left(s',\tilde{a}\middle|\theta_1^{Q'}\right), Q'\left(s',\tilde{a}\middle|\theta_2^{Q'}\right)\right). \qquad \textbf{(Eq. 2.47.)}$$

This reduces the overestimation bias by taking the lower bound of the Q-value estimates.

- Delayed Policy Updates: The policy network is updated less frequently than the critic networks. For every d iteration of the critic update, the actor network is updated once:

$$\theta^{\mu} \leftarrow \theta^{\mu} + \alpha \nabla_{\theta^{\mu}} J(\phi), \qquad \textbf{(Eq. 2.48.)}$$

where the policy gradient is:

$$\nabla_{\theta^{\mu}} J(\theta^{\mu}) = \mathbb{E}_{s\sim\mathcal{D}}\left[\nabla_a Q\left(s,a\middle|\theta_1^Q\right)|_{a=\mu(S|\theta^{\mu})} \nabla_{\theta^{\mu}} \mu(s|\theta^{\mu})\right]. \qquad \textbf{(Eq. 2.49.)}$$

- Target Policy Smoothing: A small amount of noise is added to the action output of the target policy (target actor), making it harder for the policy to exploit errors in the value function approximation:

$$\tilde{a} \leftarrow \mu'(s'|\theta^{\mu'}) + \epsilon,$$
$$\epsilon \sim clip(\mathcal{N}(0, \tilde{\sigma}), -c, c). \qquad \textbf{(Eq. 2.50.)}$$

This technique helps to reduce variance and improve stability in learning.

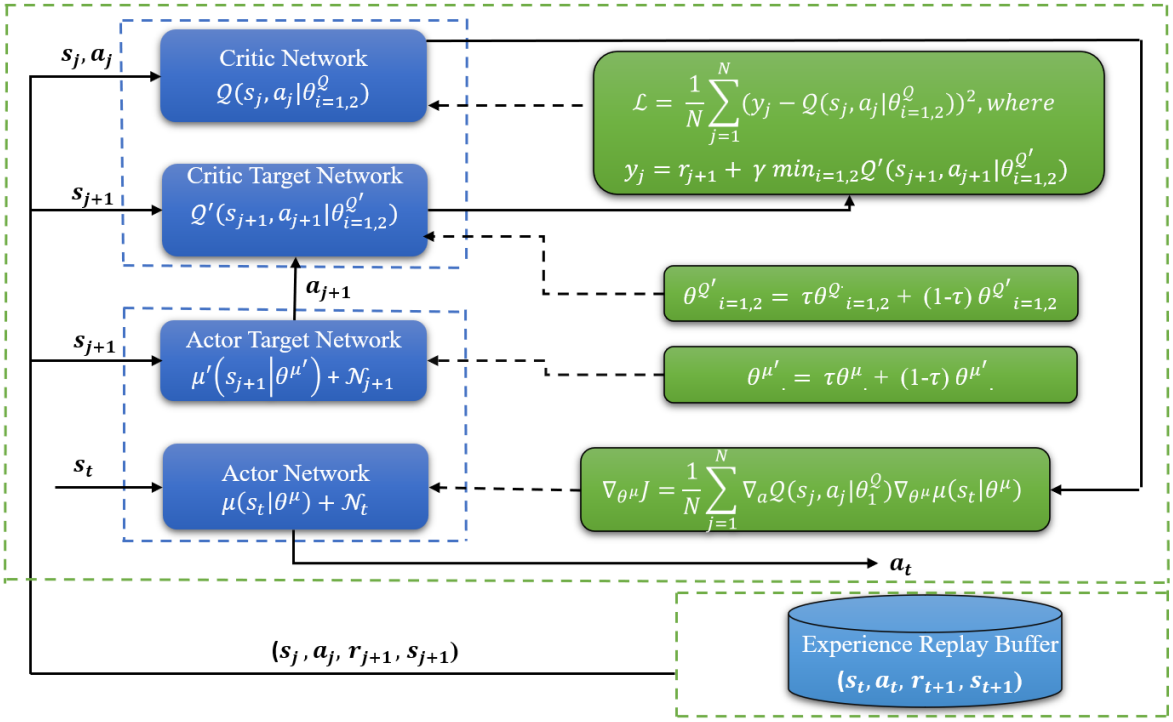

Figure 2.22. TD3 algorithm architecture.

The TD3 algorithm can be described in detail by **Table 2.5.** and **Figure 2.22.**, which illustrate its architecture and key hyperparameters, respectively. TD3 builds upon the foundation of the DDPG algorithm but incorporates several critical enhancements that address common limitations in deep reinforcement learning, particularly in environments with continuous and high dimensional action spaces. One of the major improvements in TD3 is the introduction of clipped double Q-learning, which uses two separate critic networks to estimate the Q-value and takes the minimum of the two during updates. This technique effectively reduces overestimation bias, which often leads to suboptimal policy learning in earlier methods. In addition, TD3 employs a delayed policy update mechanism, where the actor network is updated less frequently than the critics. This delay allows the critics to learn more accurate value estimates before influencing the policy, resulting in more stable and reliable actor updates. Another notable feature of TD3 is target policy smoothing, which introduces small random noise to the actions selected by the target actor network during critic updates. This approach acts as a

regularization method, preventing the policy from exploiting narrow peaks in the estimated Q-function and encouraging smoother and more generalized policies.

These mechanisms, clipping, delayed updates, and target policy smoothing collectively contribute to TD3's enhanced learning stability, sample efficiency, and robust performance. As a result, TD3 is particularly well-suited for complex continuous control tasks, such as robotic locomotion, grasping, and autonomous vehicle control, where precise and consistent decision-making is critical. The parameter configuration in **Table 2.5.** and the visual representation in **Figure 2.22.** serve as essential references for understanding how each component contributes to the overall functionality and effectiveness of the TD3 algorithm.

3. METHODOLOGY

This section presents the overall system framework developed to implement autonomous navigation for a differential-drive mobile robot. The architecture integrates elements from both classical control theory and modern reinforcement learning techniques, effectively combining data-driven decision-making with well-established motion control principles. The framework is designed to operate in a simulation environment provided by Isaac Sim, with the goal of enabling robust and adaptive navigation in complex, obstacle-rich environments.

As depicted in **Figure 3.1.**, the system relies primarily on sensor inputs including LiDAR scan data and odometry information to perceive the environment and estimate the robot's current pose relative to the desired goal location. These sensor readings provide the RL agent with critical spatial information, such as the distance and direction to the target, as well as potential obstacles in the robot's path. The observation space is structured to include a combination of processed laser scan values and pose-related features, which are continuously updated in real time.

The core of the decision-making process lies within the reinforcement learning agent, which employs the TD3 algorithm to generate high-level control commands. Specifically, the agent outputs two continuous control signals: linear velocity (v) and angular velocity (ω). These commands represent the desired forward speed and rotational rate of the robot, respectively, and are selected to maximize cumulative reward while ensuring efficient and collision-free navigation.

Before being executed, the high-level velocity commands are passed through an inverse kinematics module and a classical motion controller. These components are responsible for translating the velocity commands into appropriate motor-level signals, which are then applied to the individual wheels of the robot in the Isaac Sim environment. This hybrid approach ensures that the robot maintains smooth and stable motion, while still benefiting from the adaptive decision-making capabilities of the RL policy.

The entire system operates in a closed-loop configuration, where the robot continuously receives new sensor data, updates its observations, and adjusts its actions based on feedback from the environment. This iterative loop allows the robot to react to dynamic changes, avoid obstacles, and refine its trajectory toward the goal in real time. By integrating learned policy behavior with traditional control logic, the system achieves a balance be-tween flexibility, stability, and responsiveness, which is essential for reliable autonomous navigation in real-world scenarios.

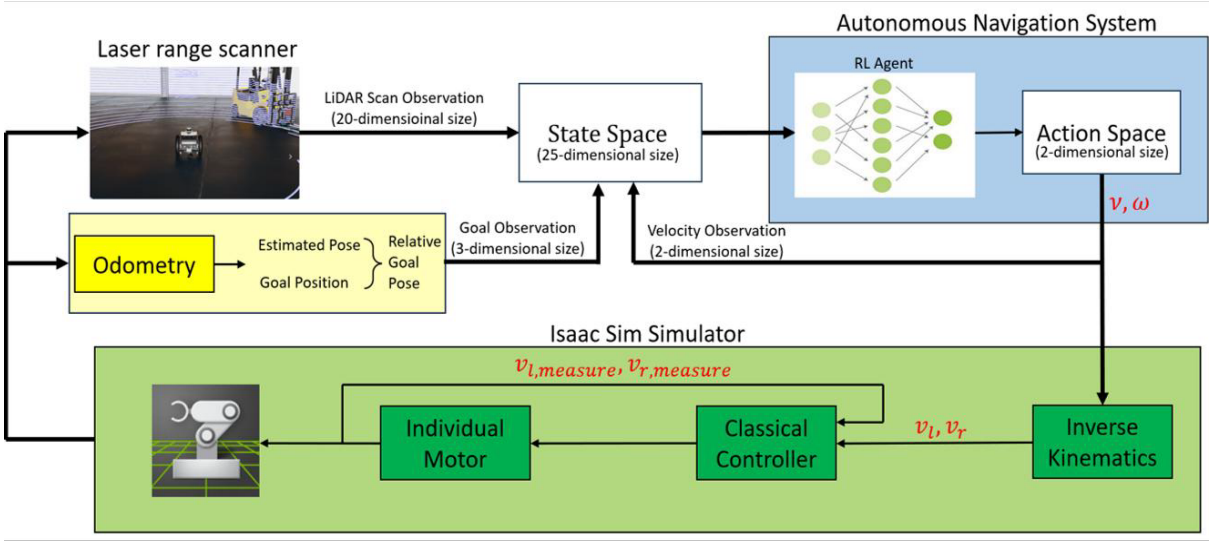

Figure 3.1. Overall system framework.

3.1. System Modeling

3.1.1. Markov Decision Process with mobile robot path planning

Markov Decision Process (MDP) (Sutton & Barto, 2018) is a sequential decision-making mathematical model used to simulate the randomized policies and rewards of intelligent agents in a Markovian environment. A five-elements tuple model $\{\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma\}$ is taken to describe the Markov decision process as hypothesis. The agent-environment interaction in a Markov decision process is illustrated in **Figure 3.2.**.

The decision-making process of mobile robot path planning can be abstracted as a Markov decision process, with continuous interaction between the mobile robot and the environment. The reinforcement learning algorithms can be applied to train the mobile robot about decision making in relation to the rotational speed of their wheels, and the robot can proactively optimal for a trajectory leading to a destination with significant reward value. Consequently, an efficacious and collision-free planned path can be obtained for the mobile robot.

A quintuple model $\{\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma\}$ is considered to depict the decision-making process for the path planning of mobile robot. Within an episode, the mobile robot receives some representation of the environment's state space $\mathcal{S}$, including all previous state information such as radar scanning information, collision status details, etc. Then, the mobile robot selects action space $\mathcal{A}$, including angular velocity of left and right wheel of the robot, where the corresponding Actor Network of the action space updated with algorithm training. The probabilities of the mobile robot to transit to the next state, based on the action $\mathcal{A}$ in the state $\mathcal{S}$, called the state

transition probability $\mathcal{P}$. As time passing, minimizing the impact of long-term benefits on current evaluation is necessary, which is in line with human pursuit of immediate benefits, due to the increasing uncertainty of long-term benefits. For minimizing the impact of subsequent state returns on the current state evaluation, discount factor, γ is introduced to the decision-making process of path planning, which can avoid the algorithm falling into an infinite loop and achieve convergence. As γ approaches 1, future rewards are taken into account more strongly, as γ approaches 0, immediate rewards are concerned more strongly. The mobile robot accepts a numerical reward $\mathcal{R}$, which taken as a consequence of its action.

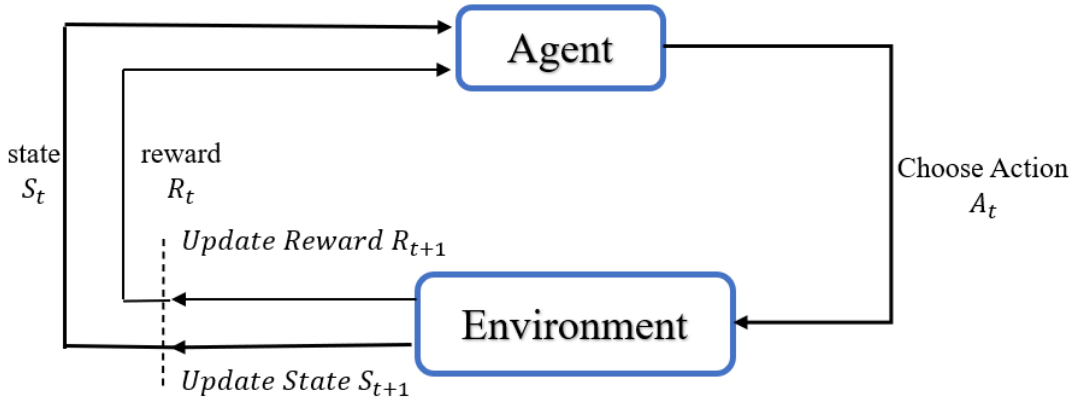

Figure 3.2. State transition graph of Markov Decision Process.

The decision-making process for the mobile robot path planning is based on the deterministic policy, so the state-value equation is expressed as v_π , which can be obtained by the following strategy π beginning from the state s.

$$v_\pi(s) = \mathbb{E}_\pi\,[\,R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T | S_t = s], \qquad \textbf{(Eq. 3.1.)}$$

With the existence of action, **Eq. 3.1.** can be rewritten when the deterministic strategy is applied as action-value equation, and the action is described as $a = \mu(s)$ based on the initial state, as shown in **Eq. 3.2.**.

$$q_\pi(s,a) = \mathbb{E}_\pi\,[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T | S_t = s, A_t = a], \qquad \textbf{(Eq. 3.2.)}$$

With the Bellman expectation equation mentioned above, our goal is to maximize these two equations above, so taking the maximum value of the equation yields the optimal value equation, as shown in equation **Eq. 2.16.** and **Eq. 2.17.**.

In the research of autonomous navigation problem, the Markov property refers to the assumption that the mobile robot selects the optimal action based solely on the current state,

which includes information about its surroundings as well as its own position and orientation. After taking the selected action, the robot receives a reward signal by sensing changes in the environment and its own updated state. It then uses this feedback to adjust its next action. This process continuous iteratively until an optimal policy is formed.

3.1.2. State Space

Mobile robots need to obtain information such as the environment and their own posture to take the optimal action, which is also known as state information, when trained by DRL algorithms. Therefore, we need to design a reasonable state space for the robot and environment to generate input data of neural network. In the autonomous task of this thesis, state information mainly consists of the following two aspects. On the other hand, environmental information includes the distance to target, heading error, linear velocity, angular velocity, left wheel's speed and lidar scan of every 18 degrees. On the other hand, it includes its own posture information and motion status. The selected mobile robot laser radar has measurement range in the range of 30 m in this thesis, and its angle of detection is 360° taken robot front as reference coordinate system, where angle information is divided into 20 dimensions, with date in range of 18° for each. The robot's state will be determined as a collision state when obstacles with a distance of less than 1.1 m and in range of 360° from the robot.

```
[DEBUG] Lidar data shape: (20,), values: [ 7.9602237  9.071642  10.967803   9.001561   8.36887
  9.884851   3.0800335 22.631746  22.464283  23.15792   19.31439
 14.563883  12.108348  11.38569   11.940234  13.209789   9.544542
  8.139847   7.764791 ]
```

Figure 3.3. The robot receives a 1D array of 20 distance readings from its LiDAR sensor at every simulation step.

The LiDAR scan provide an array of range measurements:

$$R = [r_0, r_1, \ \dots, r_{N-1}\ ], \qquad \textbf{(Eq. 3.3.)}$$

where r_i represents the measured distance at index i, and N is the total number of beams in the full scan. The angular resolution of the scan is given by:

$$\Delta\theta = msg.angle\_incremen. \qquad \textbf{(Eq. 3.4.)}$$

To reduce the dimensionality, we sample the LiDAR data every approximately 18 degrees, resulting in 20 beams. The sampling step is computed as:

$$step = \left[\frac{18^{\circ}}{\Delta\theta} + 0.5\right]. \qquad \textbf{(Eq. 3.5.)}$$

The selected indices are calculated using modulo indexing to ensure wrap-around in the circular scan:

$$S = \{(i \cdot step)\ mod\ N | i = 0, 1, \dots, 19\}. \qquad \textbf{(Eq. 3.6.)}$$

The final observation vector representing the distances from the robot to nearby obstacles is:

$$d = [r_{s_0}, r_{s_1}, \ \dots, r_{s_{19}}], s_i \epsilon S. \qquad \textbf{(Eq. 3.7.)}$$

This 20-dimensional distance vector $D \epsilon \mathbb{R}^{20}$ captures spatial obstacle information around the robot at fixed angular intervals. **Figure 3.3.** shows the real-time LiDAR data consisting of 20 beams uniformly spaced at 18° intervals across a 360° scan. The values represent distances from the robot to surrounding obstacles, measured in meters. These data are used as part of the robot's state input for the TD3 policy. Notably, low values such as 3.08 m indicate proximity to nearby objects, which directly influence the reward penalty for collision risk. Based on the principles above, we use the Boolean variable done as the state variable, which is True when collision happened and False in normal conditions, as shown in **Eq. 3.8.**.

$$done = \begin{cases} True, & d_i(t) < 1.1\ m \\ False, & d_i(t) > 1.1\ m \end{cases}. \qquad \textbf{(Eq. 3.8.)}$$

In addition to the 20-dimensional information and collision information of the laser radar mentioned above, the observation space also includes real-time pose-related information of the robot. This includes the **distance to the target** d_t and the **heading angle different** $\Delta\theta$ between the robot's orientation and the direction to the target, as defined in **Eq. 3.9.**:

$$\begin{cases} d_t = \sqrt{(x_t - x)^2 + (y_t - y)^2} \\ \theta_t = arctan2(y_t - y,\ x_t - x) \\ \Delta\theta = arctan2(\sin(\theta_t - \theta), \cos(\theta_t - \theta)) \end{cases}. \qquad \textbf{(Eq. 3.9.)}$$

Rather than directly feeding $\Delta\theta$ into observation space, its sine and cosine values, i.e., $\sin(\Delta\theta)$, $\cos(\Delta\theta)$ are used. This approach ensures smooth input values and avoids discontinuities caused by angle wrapping.

Additionally, the observation includes the robot's linear velocity and angular velocity. In this study, ideal (noise-free) robot state data is used for both training and evaluation purposes.

Among them, the target point's coordinates within the map are (x_t, y_t), and the coordinates of the robot at time t are (x, y). θ_t indicates the angle which formed by the line connected from the robot to the target point and the θ represents the orientation angle of the robot. The full observation space is composed of the following components:

- **20-dimensional LiDAR scan data**, capturing the distance to nearby obstacles.
- **1-dimensional distance to the target** d_t.
- **2-dimensional heading information** represented by $\sin(\Delta\theta)$ and $\cos(\Delta\theta)$.
- **1-dimensional linear velocity** in the robot's local frame (Longitudinal velocity).
- **1-dimensional angular velocity** around the vertical axis.

In total, the observation space has **25 dimensions**, combining sensor data and dynamic state information to support effective decision-making by the reinforcement learning agent. Ideal (noise-free) robot state data is used throughout the training and evaluation processes in simulation.

3.1.3. Action Space

First, the kinematics model of mobile robot is analyzed. Carter robot can be simplified as a two wheels differential robot, as the movement of it at time t shown in **Figure 3.4.**. Obviously, the robot will move in a straight line when the wheels at the same speed, and conversely move in a circular motion. Where, O is seen as the center of the circular motion, θ_1 is the angle that mobile robot bypasses in Δt time, θ_2 is the offset angle perpendicular to the direction of it. l is the distance between the center points of the left and right wheels, and d is the distance of approximate straight-line which is the further travelling length of right wheel than the left. r is the radius of the circular motion trajectory of the center point of the mobile robot. ω is the angular velocity, v_l is the left wheel motion velocity, v_r is the motion velocity of right, and v is the movement velocity of center of mass (the center point of the two wheels).

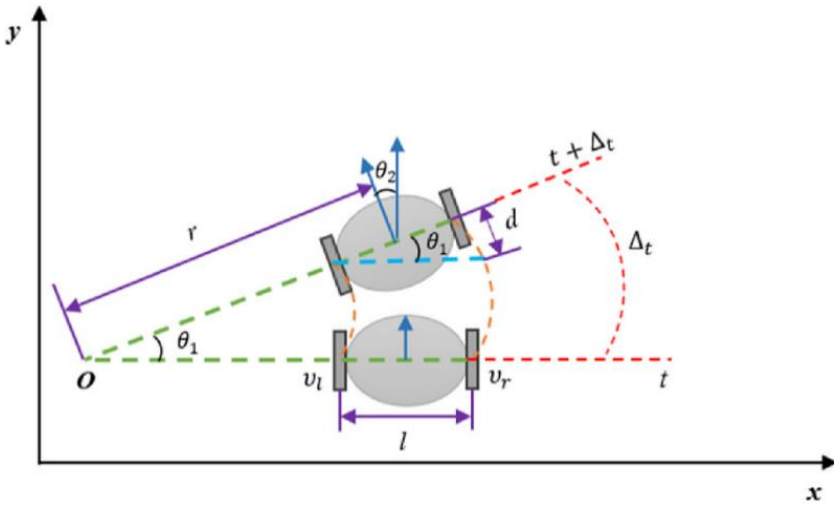

Figure 3.4. Mobile robot motion model (Li et al., 2024).

According to the equation $v = \omega \cdot r$, the relationship between left wheel speed, right wheel speed, and center of mass speed can be obtained, as shown in equation **Eq 3.10.**.

$$\omega = \frac{v}{r} = \frac{v_r}{r + \frac{l}{2}} = \frac{v_l}{r - \frac{l}{2}}, \qquad v = \frac{v_l + v_r}{2}. \qquad \textbf{(Eq. 3.10.)}$$

Based on the geometric principle of similar triangle, the angular velocity can be obtained, as shown in equation **Eq. 3.11.**:

$$\omega = tan\theta_1 = \frac{v_r - v_l}{l}. \qquad \textbf{(Eq. 3.11.)}$$

The radius r of the robot's circular motion can be obtained from **Eq. 3.10.** and **Eq. 3.11.**:

$$r = \frac{v}{\omega} = \frac{l \cdot (v_r + v_l)}{2(v_r - v_l)}. \qquad \textbf{(Eq. 3.12.)}$$

According to equation **Eq. 3.12.**, the motion mode of two-wheel differential mobile robot is determined by the speed difference between the two wheels, and if circular motion is carried out, the center O of the motion can also be determined.

At the same time v_l , v_r can be resolved by analyzing the velocity v of the robot's center of mass, as shown in **Eq 3.13.**:

$$\begin{bmatrix} v_r \\ v_l \end{bmatrix} = \begin{bmatrix} v + \frac{l}{2}\omega \\ v - \frac{l}{2}\omega \end{bmatrix} = \begin{bmatrix} 1 & \frac{l}{2} \\ 1 & -\frac{l}{2} \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}. \qquad \textbf{(Eq. 3.13.)}$$

The equation above is called the inverse kinematics equation. Based on the kinematics equation, the speed of the two driving wheels can be calculated by use of real-time speed information $[v, \omega]^T$ of the robot and the inverse kinematics equation.

The motion information which calculated based on the kinematics model includes the angular velocity value and linear velocity value. To ensure the reasonableness of motion, they are continuous within a certain range and cannot exceed certain thresholds. So, this thesis sets the maximum linear velocity to $0.5\, m/s$, the maximum angular velocity of the robot to $0.3\, rad/s$.

3.1.4. Model Architecture

The TD3 algorithm employs two actor networks, the actor-evaluate and the actor-target. Both networks share the same architecture, typically consisting of fully connected neural layers, but are initialized with different random weights to serve distinct roles during the learning process.

The actor-evaluate network (also referred to as the main actor) is responsible for learning the current policy by mapping the observed states of the environment to specific actions. During training, this network is updated based on the feedback received from the critic networks, which estimate how good a particular action is in a given state (i.e., the Q-value). This enables the actor-evaluate network to gradually improve its decision-making policy over time.

On the other hand, the actor-target network is a slowly updated replica of the actor-evaluate network. It is used in conjunction with the target critics to compute stable target Q-values during training. Instead of updating it at every step, TD3 employs a soft update mechanism, where the target network parameters are slowly adjusted toward those of the main actor using a small update rate (τ). This strategy helps to reduce variance and prevent instability during learning by smoothing sudden changes in the policy.

Together, these two actor networks contribute to the robustness and stability of the TD3 algorithm, allowing it to effectively learn high-quality policies in continuous control environments. The separation of evaluation and target networks is a core principle that underlies TD3's improved performance over earlier methods like DDPG.

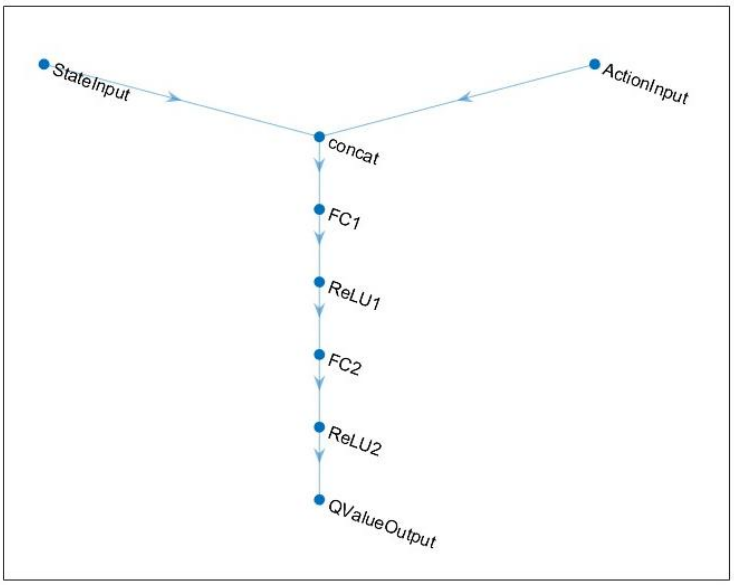

Figure 3.5. Critic network architecture.

The TD3 algorithm incorporates a total of four critic networks comprising two critic-evaluate networks and two critic-target networks. These networks all share the same architecture, typically consisting of fully connected layers designed to approximate the Q-function, but they are initialized with different random weights to perform distinct roles within the learning process.

The critic-evaluate networks are responsible for estimating the Q-values of state-action pairs generated by the actor-evaluate network. Given a current state and the action proposed by the actor, each critic-evaluate network independently predicts a Q-value that represents the expected cumulative reward the agent can achieve from that point forward. The use of two separate critics allows TD3 to apply the clipped double Q-learning technique, where the minimum of the two Q-value estimates is used for updating the actor. This helps mitigate overestimation bias, a common issue in reinforcement learning that can lead to unstable training and poor policy performance.

In parallel, TD3 maintains two critic-target networks, which are slow-moving copies of the critic-evaluate networks. These target critics are updated using a soft update mechanism to ensure stable target Q-value computation during training. They are used when computing the target value for the Bellman update, in conjunction with the action predicted by the actor-target network (with added noise for target policy smoothing). This stable target helps the evaluate critics learn more effectively and avoid oscillations or divergence in value estimates.

By leveraging multiple critic networks in this architecture, TD3 achieves greater robustness and stability in learning, especially in complex, high-dimensional continuous control tasks. The combination of critic redundancy, action smoothing, and delayed policy updates allows the algorithm to learn more reliable policies and generalize better across diverse environments.

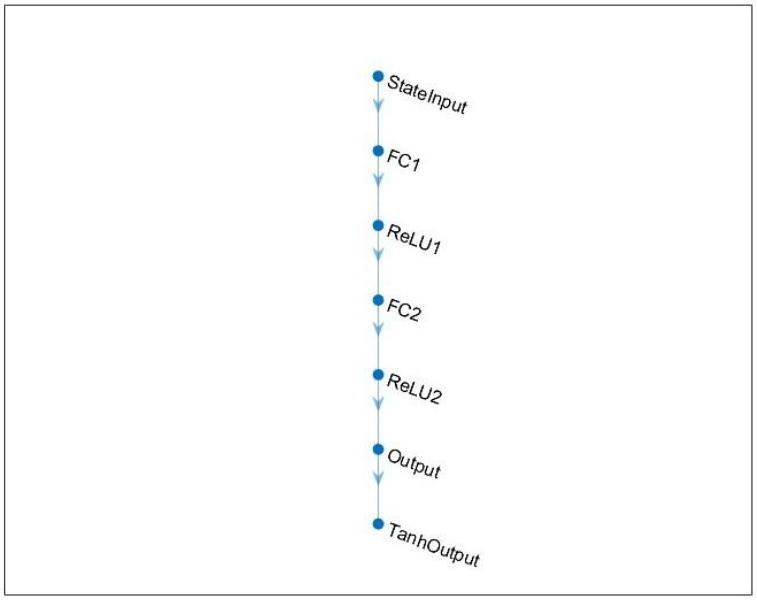

Figure 3.6. Actor network architecture.

3.1.5. Reward Function

The reward function in this project is carefully constructed to guide the mobile robot to navigate safely, efficiently, and smoothly through a simulated warehouse environment toward a dynamic goal location. The design considers multiple behavioral cues, including geometric progress toward the goal, heading alignment, motion smoothness, obstacle avoidance using LiDAR, and reward for reaching the goal. The total reward at each simulation timestep is computed as a composite of both positive and negative feedback signals:

$$R = R_{pos} + R_{head} + R_{goal} + R_{lidar} + R_{smooth} + R_{angular}. \qquad \textbf{(Eq. 3.14.)}$$

The **position progress reward** R_{pos} is defined as the change in Euclidean distance to the goal between the current and previous timestep. It is computed using the formula:

$$R_{pos} = (d_{prev} - d_{current}), \qquad \textbf{(Eq. 3.15.)}$$

where d_{prev} is the distance to the goal at the previous timestep, and $d_{current}$ is the current distance. This term rewards the robot for moving closer to the goal and penalizes it (implicitly) if it moves away.

The **heading alignment reward** R_{head} is designed to encourage the robot to face in the direction of the goal. It is based on the angular error θ_e between the robot's current heading and the vector pointing from the robot to the goal. The reward is given by an exponential decay function:

$$R_{head} = e^{(-\frac{|\theta_e|}{k})}, \qquad \textbf{(Eq. 3.16.)}$$

where θ_e is expressed in radians and k is a tunable coefficient that controls the sensitivity of the reward to misalignment. A smaller θ_e result in a higher reward, promoting direct alignment with the goal.

The **goal-reaching reward** R_{goal} provides a discrete and significant positive reward when the robot reaches within a predefined tolerance of the goal position. It is defined as:

$$R_{goal} = \begin{cases} r_g, & if\ d_{current} < \varepsilon \\ 0, & otherwise \end{cases}, \qquad \textbf{(Eq. 3.17.)}$$

where r_g is a fixed positive constant and ε is a small threshold defining goal proximity. This term reinforces successful navigation episodes.

The **LiDAR-based avoidance penalty** R_{lidar} is introduced to discourage the robot from approaching obstacles too closely. It utilizes processed LiDAR distance readings, where the minimum beam distance d_{min} among all scanned directions is considered:

$$R_{lidar} = \begin{cases} -\alpha, & if\ d_{min} < d_{safe} \\ 0, & otherwise \end{cases}. \qquad \textbf{(Eq. 3.18.)}$$

The **smoothness penalty** R_{smooth} is used to penalize abrupt changes in linear velocity, promoting smooth and stable motion. It is computed based on the change in linear velocity Δu between successive timesteps:

$$R_{smooth} = -\beta \|\Delta u\|^2, \qquad \textbf{(Eq. 3.19.)}$$

where β is a tunable weight, and $\Delta u = u_t - u_{t-1}$ is the vector difference between the current and previous action commands. This term helps the robot develop consistent and energy-efficient driving behavior.

Finally, the **angular velocity penalty** $R_{angular}$ is applied to discourage excessive spinning or oscillatory behavior. It penalizes high yaw angular velocity ω_z:

$$R_{angular} = -\gamma |\omega_z|, \qquad \textbf{(Eq. 3.20.)}$$

where γ is a coefficient that determines how strongly angular motion is penalized. By limiting rotational acceleration, this term contributes to smoother orientation control and enhances overall trajectory stability.

Together, these reward components form a comprehensive structure that balances goal-oriented behavior, safety, and smooth locomotion, enabling the robot to learn efficient path planning in dynamic, obstacle-filled environments.

3.2. Implementation

Table 3.1. Environmental configuration for simulation experiments.

Configuration	Values
Operating System	Ubuntu 22.04
CPU	Intel(R) Xeon(R) Bronze 3206R
GPU	RTX3060
RAM	32
Robot Platform	Carter
Project Programming Language	Python 3.10
Machine Learning Open-Source Library	PyTorch
Reinforcement Learning Experimental Environment Library	Gym

The implementation of an autonomous navigation system for a mobile robot using Twin-Delayed Deep Deterministic Policy Gradient (TD3) involves three key components: building the world environment in Isaac Sim, utilizing pre-built carter robot and simulated LiDAR, and TD3 and Navigation Task Integration. **Figure 3.7.** illustrates the overall workflow of the project.

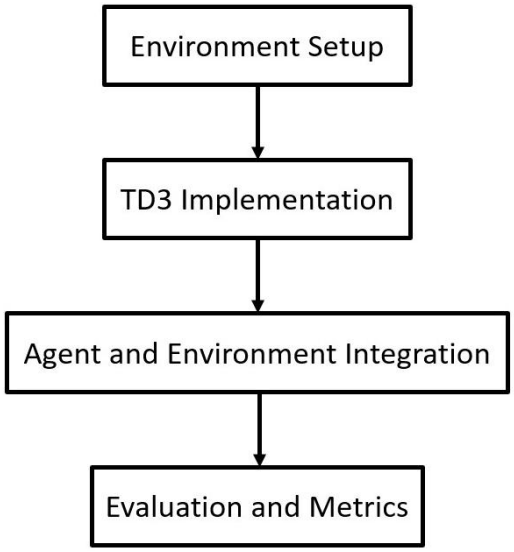

Figure 3.7. Workflow of the TD3-based navigation system.

3.2.1. Building World Environment in Isaac Sim

A realistic warehouse environment was constructed in Isaac Sim to serve as the simelation space for training and evaluating autonomous navigation algorithms. This virtual environment closely mimics real-world industrial settings to provide meaningful training scenarios for the mobile robot.

At the foundation of the scene lies a flat ground plane designed with high-friction surface properties. This setup ensures that the simulated robot's movement reflects realistic traction and collision dynamics, enabling accurate testing of velocity control, turning behavior, and obstacle interaction. To maintain consistent visibility across the entire environment, a dome light is employed to provide uniform illumination from all angles. This global lighting setup ensures that all simulated assets including the robot and surrounding objects are always clearly visible, regardless of their position or orientation within the scene.

A realistic warehouse environment is constructed in Isaac Sim to serve as the simulation space for autonomous navigation training. The virtual scene includes a flat ground plane with high friction properties to ensure accurate robot movement and collision dynamics. The environment is lit evenly from all angles using a dome light so that the robot and objects are always clearly visible, regardless of their location or orientation in the scene.

To enhance realism and create meaningful challenges, the environment is populated with static industrial objects, such as shelving units, forklifts, shop tables, and crates. These elements are carefully arranged to reflect typical warehouse layouts, introducing a variety of navigation difficulties including narrow aisles, cluttered zones, and confined turning areas. In

addition, humanoid models are introduced to simulate the presence of human workers. Although these humanoids are static in the current setup, they represent potential dynamic obstacles and enhance the realism of the simulated workspace.

All elements in the scene are structured using USD assets, which provide a modular and scalable framework for efficient scene composition. This allows for quick adjustments to object positioning, robot initialization, or environmental properties without restarting the simulation making it ideal for iterative training and testing workflows.

The dimensions of the warehouse are approximately **30 meters in length** and **19 meters in width**, providing sufficient space for diverse navigation tasks. This layout supports a variety of scenarios, including long open paths for high-speed movement, narrow corridors requiring precise maneuvering, and areas containing densely arranged obstacles. Together, these features create a controlled yet realistic environment that effectively challenges the robot's path planning, obstacle avoidance, and decision-making capabilities.

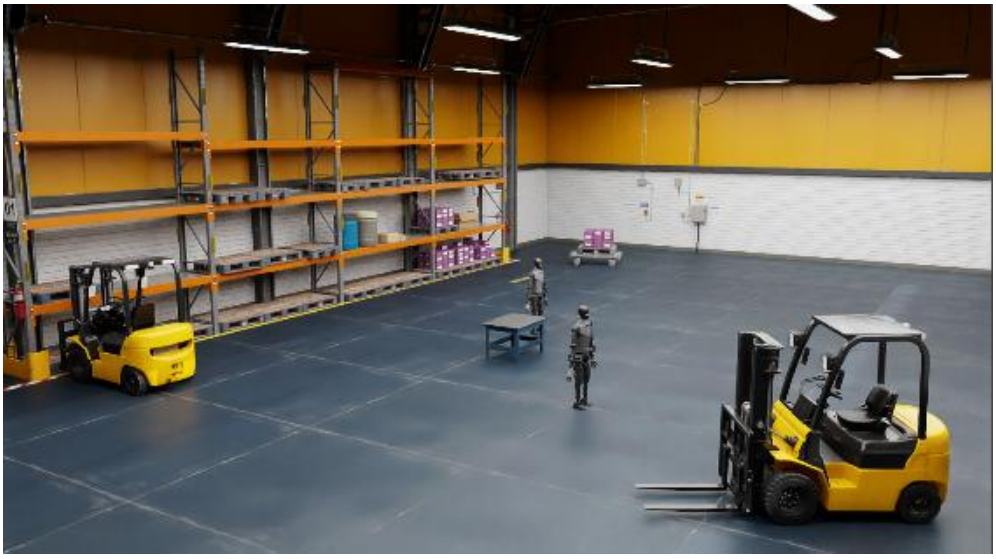

Figure 3.8. Simulated warehouse environment in Isaac Sim.

3.2.2. Utilizing Pre-built Carter Robot and Simulated LiDAR Sensor

To accelerate the development and simulation process, the project leverages the pre-built **Carter robot** as shown in **Figure 3.9.**, provided in NVIDIA Isaac Sim. Carter is a differential drive mobile robot equipped with essential sensing and actuation capabilities, making it suitable for autonomous navigation tasks in complex environments.

Figure 3.9. Carter robot.

The Carter robot includes the following components:

- **Chassis with drive wheels**: Two independently actuated wheels enable differential drive motion.
- **Simulated rotary LiDAR**: A 360-degree LiDAR sensor mounted at the center of the robot to simulate real-time distance measurements.
- **IMU and camera joints**: Although not used in this study, Carter also comes with an IMU and a camera mount for future sensor fusion tasks.

Figure 3.10. shows the Carter robot loaded into the simulation with its articulated components visible in the stage hierarchy.

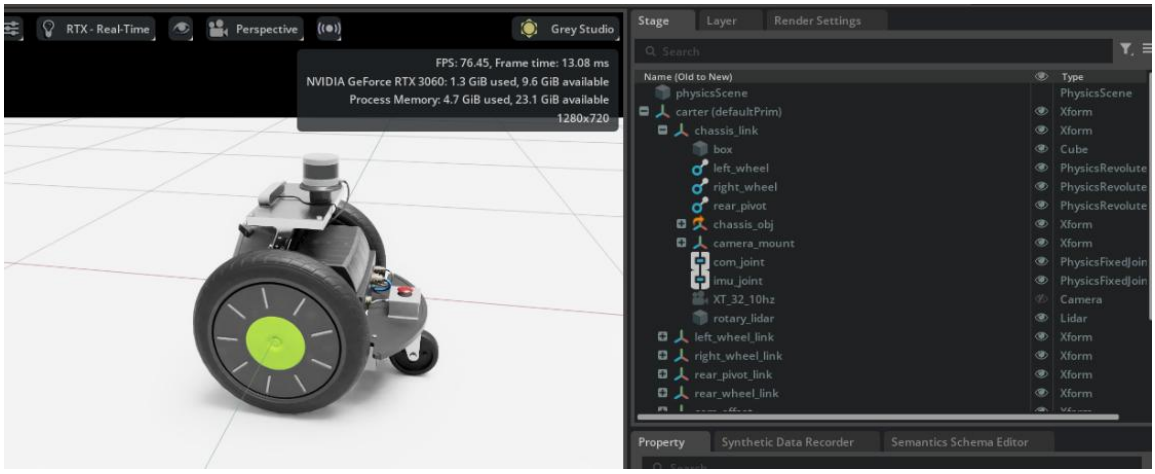

Figure 3.10. Carter robot loaded into simulation with its articulated components visible in the stage hierarchy.

The following Table is the specifications of the Cater robot used for simulation in Isaac Sim.

Table 3.2. Carter robot specifications.

Parameter	Value
Overall Length	0.52 m
Width	0.38 m
Height	0.48 m
Mass	11.5 kg
Wheel Radius	0.09 m
Wheel Base	0.28 m

In this study, the LiDAR sensor plays a critical role in perception. It enables the robot to detect surrounding obstacles by emitting laser beams and measuring reflected distances across a 360-degree field of view. The LiDAR provides a set of distance readings, representing the range from the robot to surrounding obstacles at discrete angular intervals across 360 degrees. These readings are processed as a 1D laser scan array.

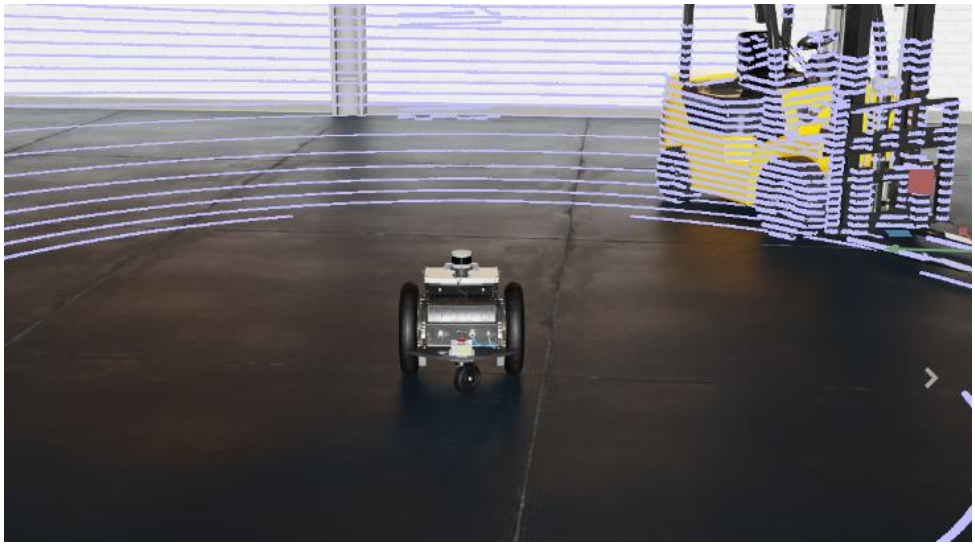

Figure 3.11. Simulated laser scan output from Carter in a warehouse environment.

This laser scan data is streamed via a custom socket interface from a ROS 2 node and used as part of the observation space for the reinforcement learning agent. In this setup, 20 evenly spaced laser beams (one every 18 degrees) are sampled per scan to provide a compact yet informative representation of the robot's surroundings.

By utilizing the Carter robot's simulated laser scanner, the project achieves an efficient and realistic setup for training autonomous navigation agents in complex indoor environment.

3.2.3. Environment Setup

The navigation environment is implemented in NVIDIA Isaac Sim, leveraging GPU accelerated physics for high-fidelity of the Carter differential-drive robot in a warehouse scenario. Key components include:

- **Physics Engine**: Time step = 1/60s (real-time synchronization).
- **Scene**: Custom warehouse USD asset with static/dynamic obstacles (forklifts, humanoid robots).

❖ **Critical Constants**

Three critical constants govern the training dynamics:

- **Goal Reached Tolerance (0.15m)**: Determines how close the robot must be to a waypoint to register success. This threshold balances precision and learnability too strict and the robot struggles to complete episodes, too lenient and it fails to develop accurate navigation.
- **Collision Threshold (1.1m)**: Defined by the minimum LIDAR reading before registering a collision. This 1.1m buffer provides adequate reaction time while preventing overly cautions behavior.

❖ **Robot Initialization and Control**: The Carter robot spawns at a fixed starting position (x=1.0m, y=-8.0m) with randomized orientation ($\pm 30°$) to promote generalization. Differential drive control is implemented through direct wheel velocity commands.

❖ **Perception System**

The robot's environmental awareness comes from two key sensors:

- **20-beam LIDAR**: Provides 360° coverage with18° angular resolution, normalized to [0,1] based on the 30m maximum range. The 20-beam configuration offers a balance between computational efficiency and spatial awareness.

- **Odometry**: Tracks the robot's position (root_pos_w) and velocity (root_lin_vel_b, root_ang_vel_w) at 60Hz, synchronized with the physics engine.

❖ **Navigation Task Design**

The training scenario implements a progressive goal-reaching task:

- Seven sequential waypoints spaced $3m$ apart along the warehouse's longitudinal axis.
- Each waypoint includes random x-axis jitter ($\pm 0.5m$) to prevent simple straight-line solutions.

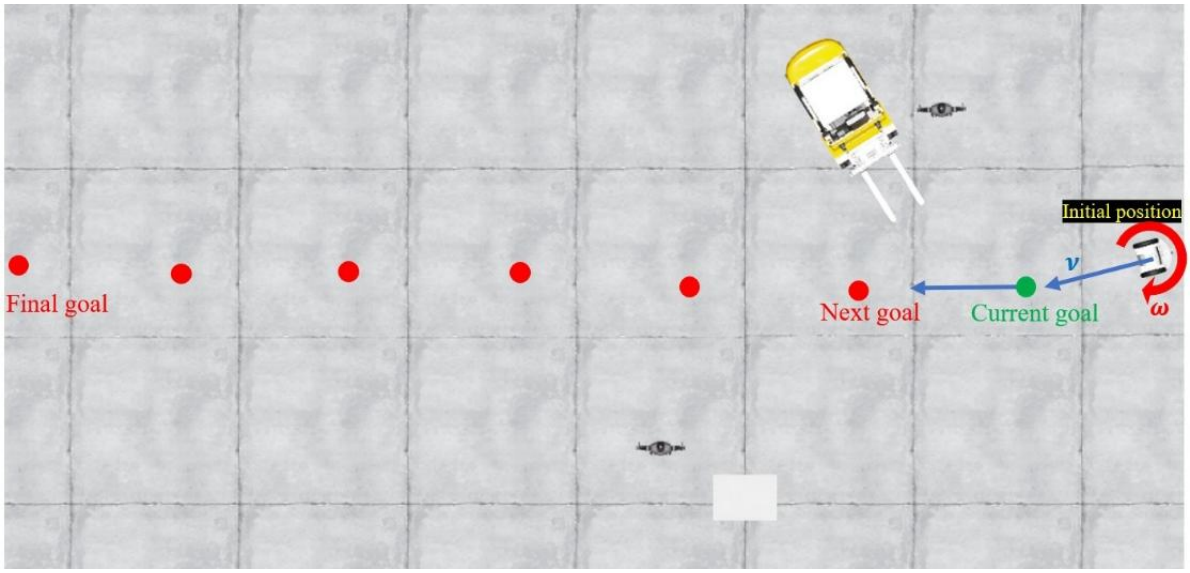

Figure 3.12. The seven sequential waypoints.

❖ **Termination Conditions**

Episodes concludes when:

- All waypoints are reached (position error $< 0.75m$).
- Collision detected (LIDAR reading $< 1.1m$).
- Timeout occurs (20 seconds elapsed).

These conditions create a balanced training regime that rewards efficiency while penalizing unsafe navigation.

- **System Integration:** The simulation interfaces with the TD3 agent through Ros2 topics for sensor data (/scan) and command output (/cmd_vel).
- **Isaac Sim Environment**

The **Isaac Sim Environment** class, illustrated in **Figure 3.13.**, encapsulates the core setup and interaction logic required to simulate the Carter robot within a highly detailed virtual environment. This class serves as the backbone for integrating the robot, sensors, and navigation logic into the simulation platform, enabling effective training and evaluation of reinforcement learning algorithms such as TD3. Below is a detailed explanation of the key components and functions within this environment class:

- **setup_scene**: This function is responsible for initializing all critical elements of the simulation. Specifically, it loads the 3D warehouse environment, spawns the Carter robot, positions humanoid obstacles, and configures the lighting setup to ensure realistic rendering and depth perception. This setup is fundamental for creating a representative and challenging environment for robot navigation.
- **Waypoints**: Waypoints serve as predefined target positions that the robot must navigate to during each episode. This component manages the generation, storage, and visualization of these goals. By providing a sequence of dynamic or fixed target locations, the waypoints system allows for structured training scenarios and clear evaluation of the robot's navigation performance.
- **_reset**: This function is called at the beginning of each episode to reset the simulation state. It repositions the robot to its initial starting point, clears internal buffers, and regenerates new goal locations. This ensures a clean and unbiased training environment across episodes, promoting learning stability and generalization.
- **_pre_physics_step**: This method processes the agent's raw action output before it is applied to the robot. It scales and clamps the throttle (wheel speed) values to remain within safe operational bounds, preventing unrealistic or unstable control signals that could disrupt the simulation or damage the robot in a real-world deployment.
- **_apply_action**: This function sends the processed velocity commands to the robot. It also plays a role in observation construction, combining key sensor inputs and robot state information including the distance to the goal, heading error, current velocity, throttle commands, and real-time LiDAR data into a comprehensive observation vector that is fed back to the learning algorithm.

- **_get_rewards**: This function computes the reward signal used to guide the learning process. It evaluates the robot's behavior based on several criteria: progress toward the goal, alignment of heading, avoidance of collisions, and smoothness of movement. A composite reward is then assigned based on these components to encourage desirable navigation behavior and penalize unsafe or inefficient actions.
- **_get_dones**: This function determines when an episode should be terminated. It checks for several terminal conditions: whether the robot has successfully reached all target goals, has collided with obstacles or walls, or has exceeded a predefined time limit. These conditions help to define the episodic structure of the training and ensure timely resets for continuous learning.

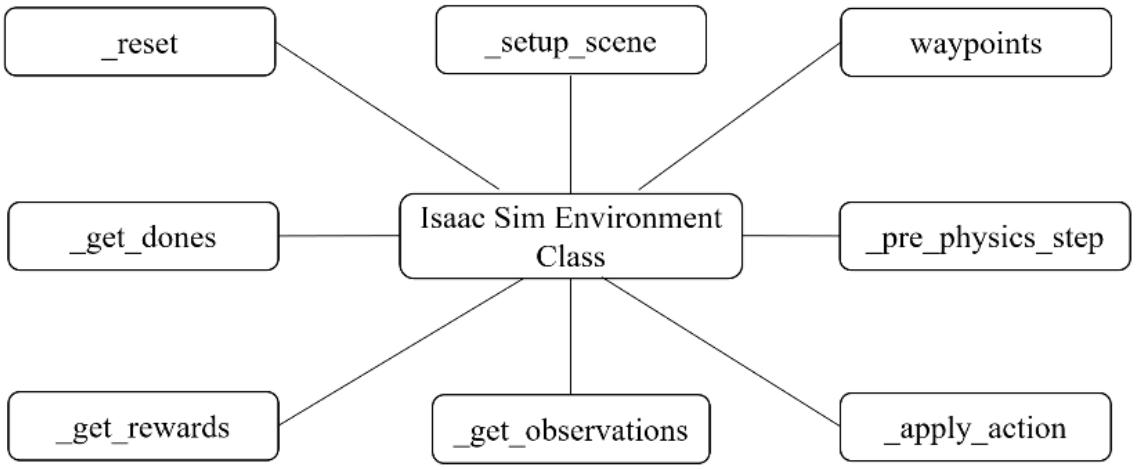

Figure 3.13. Diagram of the Isaac Sim Environment class functions.

Collectively, these functions form a cohesive and modular structure that supports both the training and testing phases of the autonomous navigation system. The flexibility of this design allows for the easy extension of the environment to include additional sensors, goal logic, or dynamic elements as needed for more advanced research or real-world deployment scenarios.

3.2.4. TD3 Implementation

To enable continuous control of the Carter robot in a complex simulated warehouse environment, the Twin-Delayed Deep Deterministic Policy Gradient (TD3) algorithm was implemented. TD3 is a model-free, off-policy actor-critic algorithm designed to improve the stability and performance of the original Deep Deterministic Policy Gradient (DDPG).

❖ **Agent Architecture**

The TD3 agent consists of the following components:

- **Actor Network**: Outputs continuous actions (wheel throttle commands) based on the observed state.
- **Two Critic Networks (Critic 1 & Critic 2)**: Estimate the Q-values for state-action pairs.
- **Target Networks**: Each actor and critic have a corresponding slowly-updated target network to stabilize learning.
- **Replay Buffer**: Stores past transitions (state, action, reward, next_state, done) for off-policy learning.

❖ **Action Selection (choose_action)**

During early training (warm-up), the agent selects actions using Gaussian noise for exploration. After warm-up, the actor network predicts an action given the current state, and a small amount of noise is added to promote continued exploration. The final action is clipped to ensure it stays within the robot's physical limits.

❖ **Experience Storage (remember)**

Each interaction with the environment is stored in the replay buffer to be reused multiple times during training, improving data efficiency.

❖ **Learning Procedure (learn)**

The learning process includes the following steps:

- **Sampling**: A mini-batch of experiences is sampled from the replay buffer.
- **Policy Smoothing**: Target actions are perturbed with clipped noise to reduce overestimation bias in the critics.
- **Critic Update**: Both critics compute Q-value targets using the minimum predicted value from their respective target networks. Mean squared error (MSE) loss is computed between the predicted and target Q-values. Both critics are updated via backpropagation.

- **Actor Update (delayed)**: The actor is updated less frequently than the critics (every 2 steps by default). The actor loss is defined as the negative mean Q-value predicted by Critic 1 for the actor's actions, promoting actions that maximize expected return.
- **Taret Network Update**: The weights of the target networks are updated using soft updates con-trolled by a factor τ, ensuring slow convergence and stability.

3.2.5. Agent and Environment Integration

The integration between the TD3 agent and the Isaac Sim-based Carter environment was performed to enable direct interaction between the policy and the simulation loop. Upon initialization, the environment exposes:

- **An observation space** representing the state vector composed of LiDAR readings, goal-related metrics, and robot dynamics.
- **An action space** defining continuous control over linear velocity and angular velocity of the robot.

The TD3 agent was configured using this information, with its actor and critic networks tailored to match the dimensions of the observation and action spaces. The agent was then continuously fed with observations from the environment, allowing it to select actions using its actor network, receive next state and reward from the environment, store transitions into the replay buffer, and learn from sampled experiences via the learn() function.

This tight integration ensures seamless communication between simulation and learning components, enabling end-to-end training of an autonomous navigation policy.

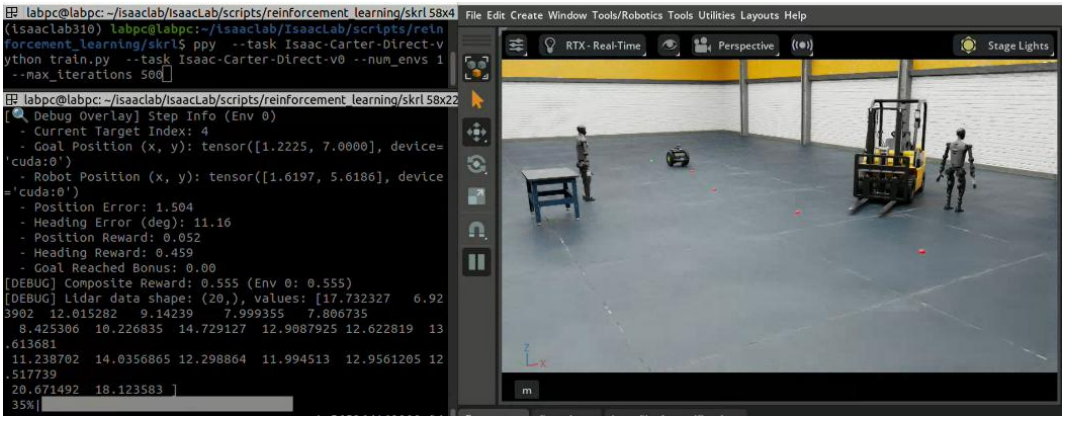

Figure 3.14. Several parameters were recorded during training, including position error, heading error, position progress reward, heading reward, and the goal-reached bonus value.

Figure 3.15. illustrates the data flow and component interactions within the autonomous navigation system designed in this project. The architecture is organized into three conceptual layers simulation, perception, and decision-making each composed of distinct modules that work together in a closed-loop system.

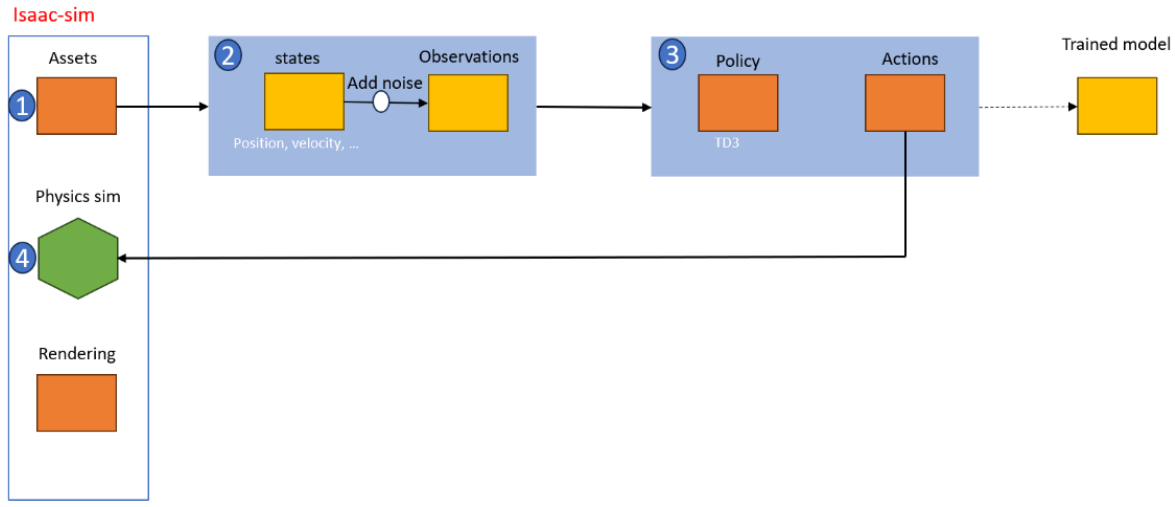

Figure 3.15. Data Flow Diagram of the Autonomous Navigation System.

At the foundation is **Isaac Sim**, which provides the core simulation environment. This includes the 3D **assets** such as the warehouse layout, mobile robot, and obstacles, which define the visual and physical context of the simulation. The **physics simulation** engine handles real-time dynamics, including robot motion, collisions, and friction behavior. Additionally, the **rendering** engine generates RGB frames and visual outputs for observation and recording, ensuring a visually accurate representation of the simulation environment.

Building on top of the simulation layer is the **Isaac Lab** framework, which acts as the interface between the simulated environment and the reinforcement learning agent. Isaac Lab extracts raw **state** information from Isaac Sim, such as joint velocities, robot position, and sensor readings, and transforms it into structured **observations**. These observations form the inputs to the decision-making module and may include processed lidar data, heading errors, or distances to goals.

The decision-making layer consists of components from the **RL Libraries**. This includes the **policy**, which is a neural network trained to map observations to control actions. Once the policy processes the observation, it generates corresponding **actions**, such as linear and angular velocity commands, which are then sent back into the simulation to control the

robot's movement. These actions directly affect how the robot behaves in the environment, thereby influencing the next state of the simulation.

Supporting this loop is the **Trained Models** component, which stores the parameters of the neural networks obtained from the training phase. During evaluation or deployment, the trained models are loaded and used by the policy to make decisions based on previously learned behavior.

The entire system operates in a continuous feedback loop. Isaac Sim generates environmental feedback based on the robot's actions; Isaac Lab processes that feedback into observations; the policy uses these observations to compute new actions; and the cycle repeats. This architecture enables a smooth and modular flow of information and allows for seamless integration of various components, such as different robot assets, sensor types, or reinforcement learning algorithms.

3.2.6. Training Procedure

Table 3.3. Parameters settings for simulation experiments.

Parameters	**Values**
Maximum Training Steps Per Episode	20
Replay Buffer Size	1000000
Batch Size	100
Learning Rate of Actor Network	0.0001
Learning Rate of Critic Network	0.001
Discount Factor	0.99
Exploration Noise	$\mathcal{N}(0,0.1)$
Target Policy Smoothing Noise	$\epsilon \sim clip(\mathcal{N}(0,0.2), -0.5,0.5)$

This section outlines the training process used to develop the TD3 agent for autonomous mobile robot navigation in simulation. The training process was designed to run for **1,600 episodes**, where each episode consists of a full navigation session from the robot's spawn position to a sequence of dynamically generated goals.

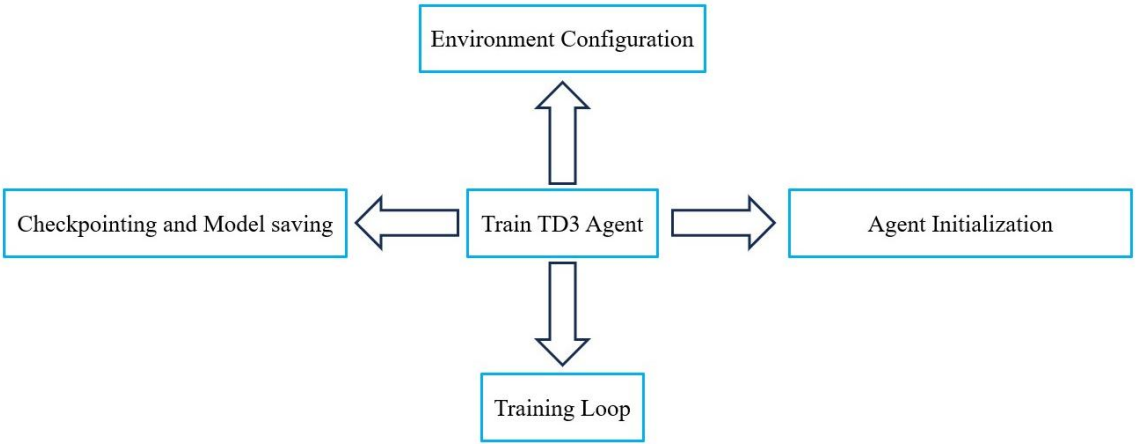

Figure 3.16. Training process outlines.

- Environment configuration: The training was conducted in a simulated warehouse environment using a single-instance Carter mobile robot. The simulation was launched through Isaac Lab, leveraging GPU acceleration (CUDA) when available. The environment provided 26-dimensional state observations that included LiDAR readings, position error, heading error, velocity information, and throttle states. The robot was trained to reach sequentially generated goal positions while avoiding obstacles.

- Agent Initialization: The TD3 agent was initialized with the parameters namely learning rate (Actor, $\alpha = 10^{-4}$, Critic, $\beta = 10^{-3}$), batch size of 100, replay buffer size is assigned to 10^6, soft update coefficient $\tau = 0.005$, discount factor $\gamma = 0.99$, neural network of two hidden layers (400 and 300 units), action dimensions of 2 (linear and angular velocity). The agent used two critic networks and aa delayed actor update strategy in accordance with the TD3 algorithm.

- Training loop

  The training loop follows these steps:

1. **Environment Reset**: At the beginning of each episode, the robot is repositioned, and a new sequence of navigation targets is generated within the warehouse.

2. **Action Selection**: At every step, the TD3 agent selects a throttle command based on the current observation using the choose_action() method.

3. **Environment Step**: The chosen action is applied to the simulation, and the environment returns the next observation, reward, and termination signal.

4. **Experience Storage**: The transition is stored in the replay buffer using remember().
5. **Policy Update**: The agent samples a mini-batch from memory and updates the critic and actor networks using the learn() method, applying TD3's delay and noise-based smoothing strategy.

❖ Checkpointing and Model saving

Every 25 episodes, the current weights of the actor, critic, and all corresponding target networks were saved to disk. This ensured recovery and reproducibility.

❖ Visualization and Analysis

Plots showing episode reward vs. cumulative time steps, running average reward, and per-step reward were saved for visualization.

- Reward per step and episode scores were written to TensorBoard and CSV files for performance monitoring.
- Smoothed average reward over the last 100 episodes was computed to observe learning trends.

3.2.7. Testing Procedure

Following the completion of training, a structured evaluation procedure was designed and executed to rigorously assess the performance of the TD3 agent in the simulated warehouse environment. The primary objective of this testing phase was to observe the learned policy's behavior in a noise-free setting, allowing for a clear analysis of its decision-making capabilities and trajectory efficiency in accomplishing navigation tasks.

Unlike the training phase, where exploration noise is added to encourage learning, the evaluation phase operates in a fully deterministic mode. This ensures that the policy's performance can be assessed purely based on its learned experience, without any random factors influencing its actions.

❖ Evaluation Setup

- Log Directory Initialization: The system automatically locates the most recent training log directory and uses it to store all evaluation outputs, including reward data, rendered frames, and visual results. This streamlines the evaluation pipeline and ensures traceability between training and testing artifacts.

- Environment Initialization: A single-instance simulation environment is initialized using the exact same configuration parameters as used during training. This consistency guarantees that the evaluation results are a fair reflection of the agent's learning.
- Model Loading: The trained TD3 model is loaded from the saved checkpoint directory. The model is then set to evaluation mode using the set_eval_mode() method. This disables any stochastic elements, such as exploration noise, ensuring that the agent behaves deterministically during testing.
- The most recent training log directory is automatically located and used for storing evaluation outputs.
- A single-instance simulation environment is initialized using the same configuration as the training phase.
- The trained TD3 model is loaded from the checkpoint directory and set to evaluation mode using set_eval_mode() to disable any stochastic exploration.

❖ **Execution Setup**

1. **Reset the Environment**: At the beginning of each evaluation episode, the simulation environment is reset to a newly initialized state. This includes randomizing the starting position and re-generating the goal location to evaluate generalization capabilities.
2. **Policy Execution**: The trained actor network selects actions directly based on the observed state. No noise is added to the output, enabling a true representation of the policy's capability under deterministic conditions.
3. **Rendering and Recording**: Each simulation timestep is rendered visually, and RGB frames are captured continuously throughout the episode. These frames are later compiled into animated GIFs, which serve as a visual reference for trajectory analysis and qualitative assessment.
4. **Reward Accumulation**: Throughout each episode, the cumulative reward is tracked. This total reward serves as a quantitative indicator of the agent's navigation performance and is used to compare results across episodes.
5. **Loop Completion**: An evaluation episode concludes when one of the following conditions is met: the robot reaches the final goal, it collides with an obstacle, or the maximum time limit is exceeded.

A total of ten evaluation episodes were conducted under these conditions. The total rewards collected from each episode were recorded, and the results were visualized for comparison. This evaluation process provides both quantitative and qualitative insights into the

policy's reliability, efficiency, and adaptability in goal-oriented navigation within a structured warehouse setting.

3.2.8. Evaluation and Metrics

To objectively assess the performance of the trained agent, the following evaluation metrics were used:

- ❖ Performance Evaluation: the evaluate function is employed to assess the poli-cy's performance over 10 episodes. During evaluation, the model executes a series of episodes without exploration noise to provide a true measure of the learned policy's performance. Key metrics recorded is the average cumulative reward obtained across all evaluation episodes, defined as:

$$\text{Moving Average Reward} = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T} r_t^i, \qquad \textbf{(Eq. 3.21.)}$$

where N is the number of evaluation episodes, T is the number of timesteps per episode, and r_t^i is the reward at timestep t in episode i.

- ❖ Logging and Monitoring: To facilitate real-time monitoring and visualization of the training process, TensorBoard is utilized through the SummaryWriter class. This logging framework captures key training metrics, which include:

- **Value Loss**: The mean squared error (MSE) loss of the critic networks, which measures the difference between the predicted Q-values and the target Q-values. The loss is calculated as the average of the losses from the two critics, Q_1 and Q_2, defined as:

$$\text{Value Loss} = \frac{1}{2B}\sum_{j=1}^{B}\left[\left(Q\left(s_j, a_j|\theta_1^Q\right) - y_j\right)^2 + \left(Q\left(s_j, a_j|\theta_2^Q\right) - y_j\right)^2\right], \qquad \textbf{(Eq. 3.22.)}$$

where B is the batch size, $Q\left(s_j, a_j|\theta_1^Q\right)$ and $Q\left(s_j, a_j|\theta_2^Q\right)$ are the predicted Q-values from the two critics, and y_j is the target Q-value.

- **Actor Loss**: The actor loss is defined as the negative expected Q-value output by the first critic, evaluated at the actor's predicted actions. It reflects how well the policy maximizes the estimated return:

$$\text{Actor Loss} = -\frac{1}{B}\sum_{j=1}^{B} Q\left(s_j, \mu(s_j|\theta^{\mu})|\theta_1^{Q}\right), \qquad \textbf{(Eq. 3.23.)}$$

where B is the batch size, $Q\left(s_j, \mu(s_j|\theta^{\mu})|\theta_1^{Q}\right)$ is the predicted Q-value, and θ^{μ} are the actor network parameters.

4. RESULTS AND DISCUSSION

This section discusses the performance of the proposed autonomous navigation system implemented in a simulated warehouse environment using the Carter mobile robot. The primary objective of this experiment is to evaluate the robot's ability to navigate safely and efficiently toward a sequence of goal positions while avoiding various obstacles present in the environment.

As illustrated in **Figure 4.1.**, the robot begins its navigation task from the initial starting position. The simulation environment includes multiple static obstacles such as forklifts, a workbench, and a humanoid figure positioned in the workspace. These obstacles simulate real-world challenges typically encountered in industrial or warehouse settings, thereby testing the robustness of the path planning and obstacle avoidance strategies employed by the system.

Figure 4.1. Initial position of the robot.

The robot is required to reach seven target positions distributed across the environment. These goal points are visually marked by colored spheres, with the final goal shown in **Figure 4.2.**. The robot successfully navigates through the environment, avoiding collisions and adjusting its trajectory in real time based on LiDAR observations and goal position updates.

The successful arrival at the final goal demonstrates the effectiveness of the trained TD3-based reinforcement learning agent in performing goal-directed navigation under challenging conditions. The trajectory followed by the robot also indicates the agent’s ability to make smooth and adaptive motion decisions while maintaining safe distances from

surrounding obstacles. These results validate the potential of integrating deep reinforcement learning techniques into autonomous navigation systems for real-world applications.

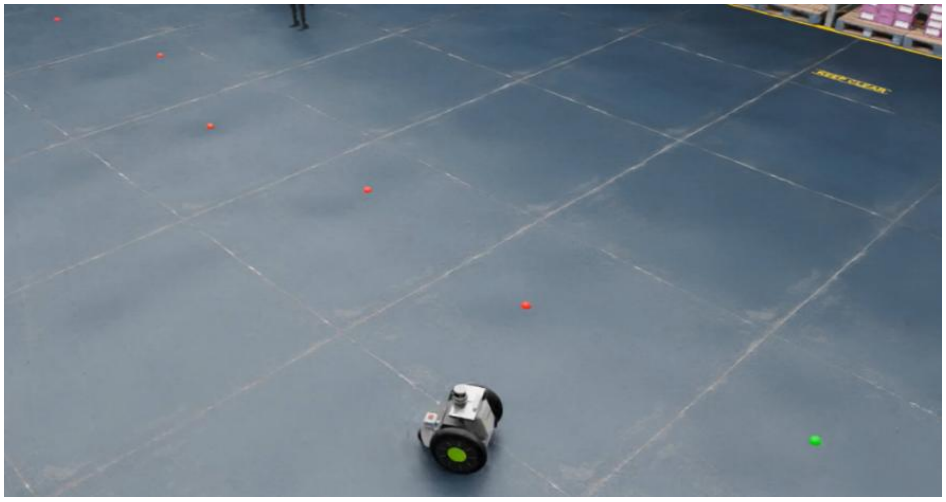

Figure 4.2. Carter robot reaching the final (7^{th}) goal point after successful navigation.

4.1. Training Results

4.1.1. Graph of Average Reward

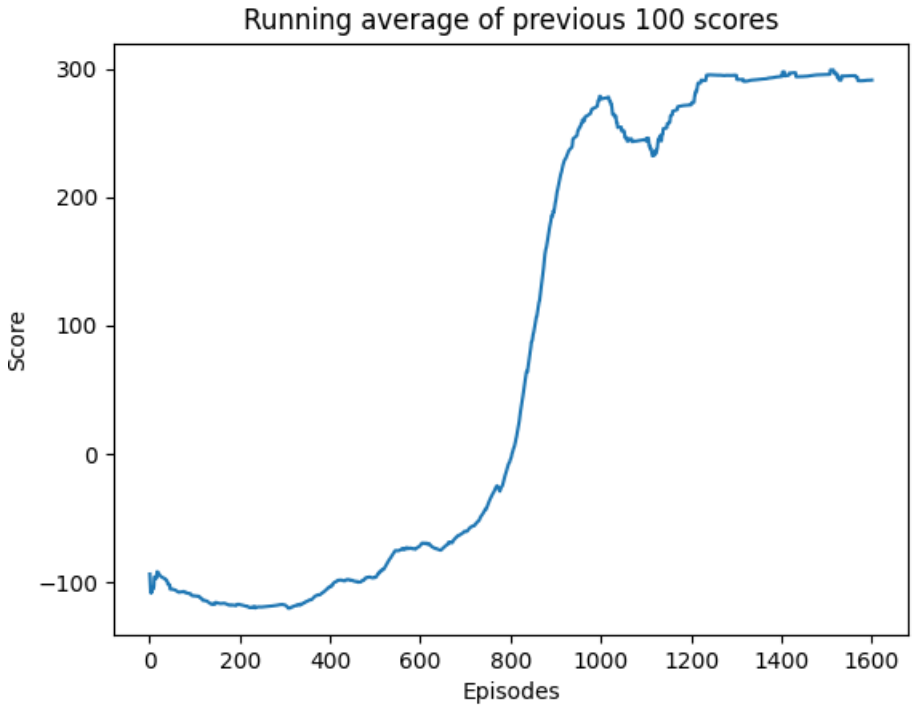

Figure 4.3. Average reward over 1600 episodes.

The reward trend plotted in **Figure 4.3.** illustrates the TD3 agent's learning progress over the course of training. A steady upward trend in both total episode rewards and average rewards per episode is clearly observed, indicating a gradual and consistent improvement in the agent's navigation performance.

This pattern suggests that the TD3 agent was effectively learning from its interactions with the environment. As training progressed, the agent became increasingly proficient at reaching target goals while avoiding obstacles and minimizing inefficient or erratic movements. The increasing rewards reflect enhanced decision-making, improved path planning, and more stable control outputs as the policy matured.

The overall trend confirms that the TD3 algorithm was successful in optimizing the navigation policy, enabling the agent to complete tasks more efficiently and reliably over time. This also validates the design of the reward function and the simulation setup, both of which contributed to shaping the agent's learning behavior in a meaningful way.

4.1.2. Graph of Actor and Critic Loss

The value loss plot displays a clear decreasing trend throughout the training process, signifying that the critic networks successfully minimized the Bellman error over time. This reduction in loss indicates that the Q-value estimations produced by the critic networks became more accurate and consistent as training progressed. Such stability in Q-value learning is a critical component of the TD3 algorithm, as it ensures reliable policy evaluation and contributes to the overall learning stability.

A consistently low value loss suggests that the agent was able to learn the expected return of taking particular actions in given states with increasing precision. This is essential for effective actor updates, as the critic provides the foundation for computing gradients that guide policy improvement. Therefore, the downward trend in value loss directly reflects the critic's growing confidence in estimating long-term rewards, which in turn supports more effective and stable learning of the robot's navigation policy.

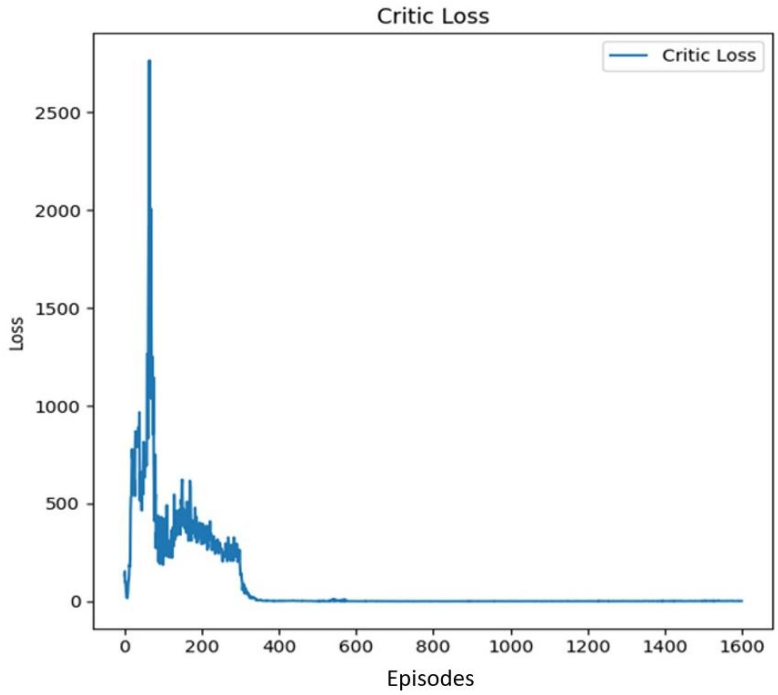

Figure 4.4. Graph of Critic Loss.

The policy loss graph provides insights into the evolution of the actor network during the training process. As training progressed and the value loss of the critic networks began to stabilize, the actor network increasingly relied on the more accurate Q-value estimations to guide its updates. This allowed the policy to make more informed decisions and gradually refine its behavior based on reliable feedback from the critics.

The trend observed in the policy loss curve reflects the actor's effort to improve the expected return by adjusting its action selections. Initially, the actor may exhibit high variability as it explores various actions, but as training continues and the critic's evaluations become more dependable, the actor converges toward a more stable and effective policy. This convergence is a key indicator that the reinforcement learning process is proceeding as intended, with both the actor and critic components reinforcing each other's learning.

Overall, the decreasing and stabilizing pattern of the policy loss signifies that the agent was successfully learning a robust and confident policy for continuous control in the simulated navigation environment.

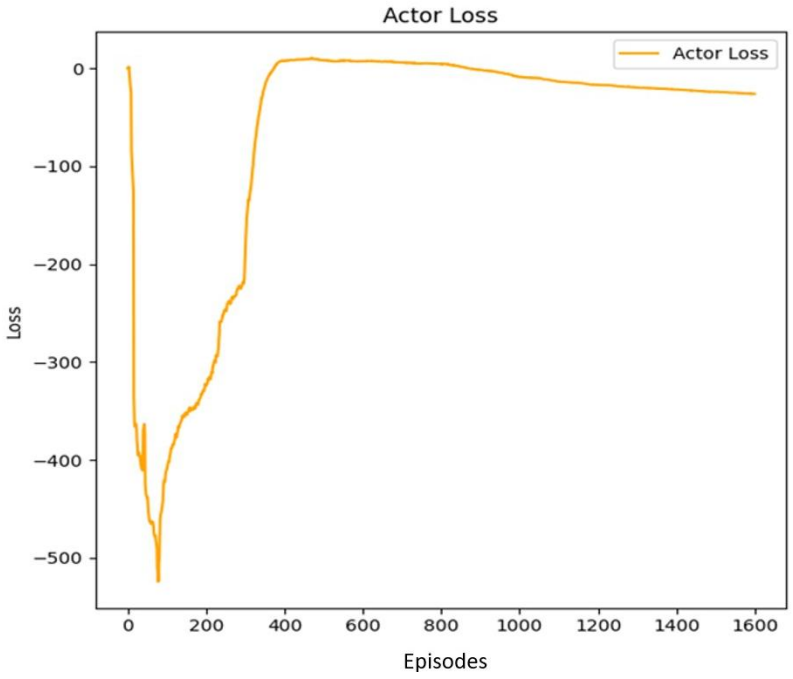

Figure 4.5. Graph of Actor Loss.

Some fluctuations in value or policy loss might point to instabilities during training, possibly caused by sparse rewards, environmental complexity, or imperfect state representations (e.g., LiDAR dropout). These could be addressed by tuning hyperparameters or improving sensory inputs.

4.2. Testing Results

To evaluate the performance of the trained TD3-based navigation policy, the model was tested across multiple episodes without further exploration noise. The evaluation was conducted using a previously saved model checkpoint to ensure consistent policy behavior during inference. The results below reflect the robot's ability to follow the learned navigation policy across four consecutive episodes:

- **Episode 0**: Score = 315.68.
- **Episode 1**: Score = 314.37.
- **Episode 2**: Score = 314.96.
- **Episode 3**: Score = 314.54.

The average score across these episodes gradually stabilized around 314.89, indicating that the policy maintained consistent performance throughout the evaluation. The minimal

variance in episode scores suggests that the agent successfully generalized its behavior and could navigate the environment reliably across multiple trials.

These scores reflect cumulative reward values obtained during each episode, incorporating key aspects of the reward function such as position progress, heading alignment, and successful goal reaching. The high and consistent scores confirm that the TD3 algorithm effectively learned an optimal navigation policy suitable for continuous control in differential-drive robots like Carter.

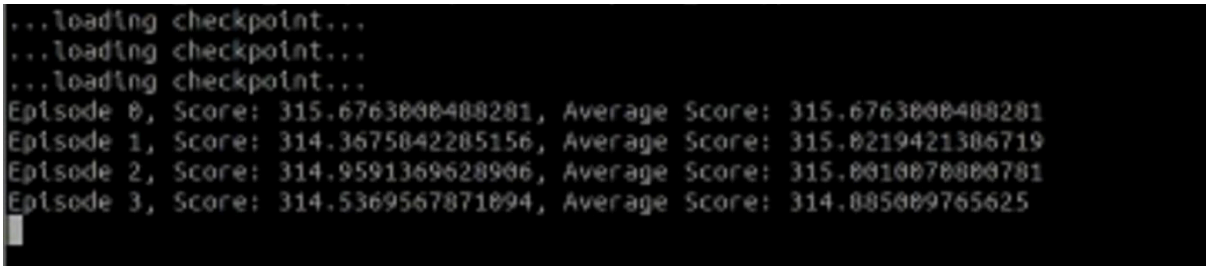

Figure 4.6. Evaluation episode rewards using trained TD3 policy.

In addition to the overall navigation performance, the individual wheel speeds of the Carter robot were observed during the evaluation phase to gain further insights into the agent's motion behavior. The recorded motor speeds exhibited slight fluctuations throughout each episode. These variations are primarily attributed to two key factors: the dynamic positioning of the target goals and the structural complexity of the environment.

As the robot navigated toward varying goal locations, the TD3 policy continuously adjusted the wheel speeds to optimize heading and trajectory, especially when turning, avoiding obstacles, or making fine corrections near goal points. In more constrained or obstacle-rich regions, the control policy often had to perform sharper turns or momentary slowdowns, which contributed to the observed speed variations between the left and right wheels.

Despite these fluctuations, the robot maintained a stable and goal-directed motion pattern across episodes, demonstrating that the TD3 policy was able to handle non-linearities in the control space and adapt effectively to environmental challenges. This behavior reflects the policy's capacity to generalize well to different navigation scenarios, ensuring smooth and responsive control in real time.

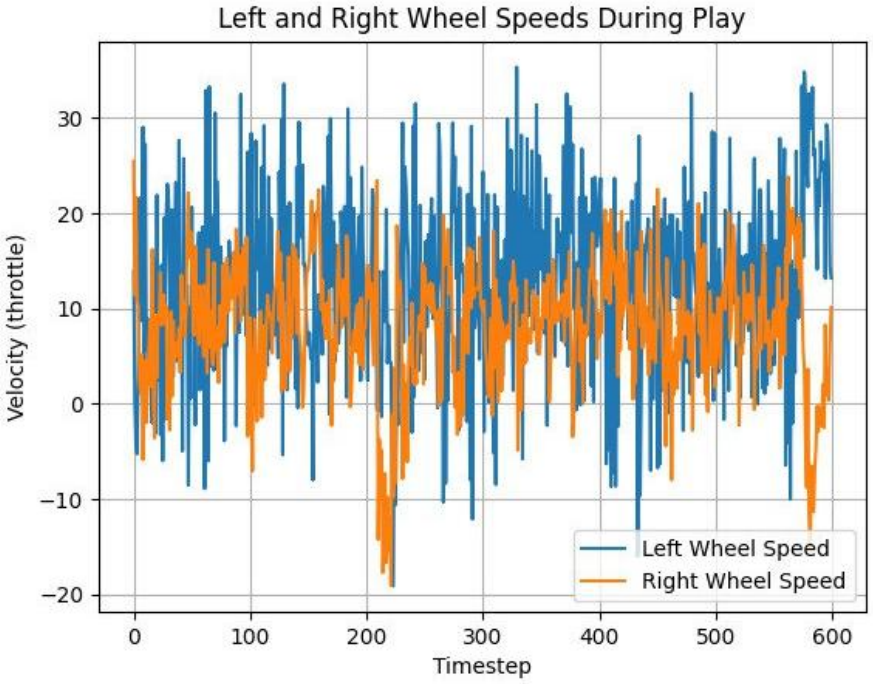

Figure 4.7. Left and right wheel speeds during testing.

5. CONCLUSIONS AND RECOMMENDATIONS

5.1. Conclusions

This project successfully implemented the TD3 algorithm for local path planning in a simulated Carter mobile robot environment using Isaac Sim. Through a series of iterative training episodes, the reinforcement learning agent demonstrated the ability to learn an effective navigation policy that enabled the robot to reach its designated waypoints while minimizing collisions and avoiding inefficient actions such as redundant turns or backtracking.

The training curves show stable convergence patterns for both the actor and critic networks, with the value loss gradually decreasing and the policy loss stabilizing over time. This indicates that the TD3 algorithm effectively balanced exploration and exploitation during learning. Moreover, the use of target networks and delayed policy updates helped mitigate issues such as overestimation bias and policy divergence, which are commonly encountered in value-based reinforcement learning algorithms.

Quantitative evaluation of the agent's performance revealed consistent improvements in cumulative rewards and reduced variance in episode durations, suggesting that the learned policy generalized well across different navigation scenarios within the simulated environment. These improvements validate the robustness and adaptability of the TD3-based controller, highlighting its potential for real-time autonomous navigation tasks in both structured and partially dynamic environments.

Furthermore, this work lays the groundwork for future research in transferring the learned policy to real-world robotic systems, incorporating additional perception modules such as LiDAR and visual sensors, and extending the navigation strategy to handle more complex tasks like dynamic obstacle avoidance and multi-goal planning.

5.2. Future Work

The results of this study confirm that the TD3 algorithm is highly effective for addressing continuous control problems in the field of robotics. Its application to a differential-drive mobile robot, such as Carter, has demonstrated reliable and stable performance in navigation tasks, particularly within environments that require smooth and precise motion control. The ability of TD3 to mitigate overestimation bias and stabilize training through delayed policy updates makes it well-suited for real-world robotic systems that demand robust decision-making in continuous action spaces.

Looking forward, several promising directions can be pursued to enhance and broaden the scope of this research. One potential improvement involves the integration of richer sensory inputs, such as RGB or depth camera data, which can enable more complex perception capabilities and improve the robot's ability to navigate unstructured or visually diverse environments. This could lead to the development of more intelligent behaviors, including obstacle recognition, semantic mapping, and vision-based goal detection.

Another important avenue for future work is the exploration of sim-to-real transfer techniques. While simulation provides a safe and flexible platform for training, deploying the learned policies on physical robots introduces real-world challenges such as sensor noise, latency, and hardware limitations. Investigating domain adaptation methods or employing techniques like domain randomization can help bridge the gap between simulation and real-world performance.

Additionally, a comprehensive comparison between TD3 and other state-of-the-art reinforcement learning algorithms such as Soft Actor-Critic (SAC) and Deep Deterministic Policy Gradient (DDPG) under identical environmental conditions would provide deeper insight into their relative strengths and weaknesses. This comparative analysis can guide algorithm selection for specific robotic tasks and further refine performance benchmarks.

By pursuing these future directions, the applicability and effectiveness of TD3-based autonomous navigation systems can be significantly extended, paving the way for more advanced and reliable robotic solutions in dynamic and complex real-world settings.

5.3. Recommendations

For future development, it is strongly recommended to deploy the TD3-based autonomous navigation system on a physical mobile robot platform. Transitioning from simulation to real-world implementation would provide valuable insights into the practical feasibility and performance of the proposed method under realistic operating conditions.

To facilitate successful deployment, the mobile robot must be equipped with appropriate hardware components, particularly sensors and computational units capable of sup-porting real-time data acquisition and processing. This includes integrating reliable laser range finders or LiDAR sensors to perceive and map the surrounding environment accurately. Additionally, leveraging embedded AI computing platforms, such as the NVIDIA Jetson series, can offer the necessary computational power to run deep reinforcement learning models efficiently while maintaining low latency.

Moreover, extensive field testing in varied and dynamic environments such as indoor warehouses, outdoor terrains, and obstacle-rich areas will be essential to evaluate the system's adaptability and robustness. These experiments will help identify practical challenges such as sensor noise, mechanical limitations, localization errors, or environmental unpredictability, which are often absent in simulation.

Ultimately, implementing and validating the TD3 navigation system on real hardware not only strengthens the contribution of this research but also brings it a step closer to practical applications in autonomous delivery, service robotics, and intelligent transportation systems.

REFERENCES

Ali, R. (2023, December 4-5). A Comparitive Study of TD3 and SAC Algorithms for Exploration and Navigation in Unseen Environments. *World Academy of Science, Engineering and Technology (WASET)*. Retrieved from https://iris.polito.it/retrieve/fec01e42-7735-40f0-8cbf-64b15d056372/robot-exploration-and-navigation-in-unseen-environments-using-deep-reinforcement-learning.pdf

Bellman , R. E. (1957). A markovian decision process. *Journal of Mathematics and Mechanics, 6(5)*, 679–684. https://doi.org/679–684. http://www.jstor.org/stable/24900506

Fujimoto, S., Hoof, H. v., & Meger, D. (2018). Addressing Function Approximation Error in Actor-Critic Methods. *the 35th International Conference on Machine Learning (ICML 2018).* Stockholm, Sweden. Retrieved from http://arxiv.org/pdf/1802.09477.pdf

GeeksforGeeks. (2024, October 21). Retrieved May 22, 2025, from GeeksforGeeks: https://www.geeksforgeeks.org/what-is-perceptron-the-simplest-artificial-neural-network/

Goodfellow, I., Bengio, Y., & Courville, A. (2016). (M. Press, Producer) Retrieved May 22, 2025, from http://www.deeplearningbook.org

Harapanahalli, S., Mahony, N. O., Hernandez, G. V., Campbell, S., Riordan, D., & Walsh, J. (2019). Autonomous Navigation of mobile robot in factory environment. *The 29th International Conference on Flexible Automation and Intelligent Manufacturing (FAIM2019)* (pp. 1524-1531). Elsevier B.V. Retrieved May 4, 2025, from https://www.researchgate.net/publication/339116746_Autonomous_Navigation_of_mobile_robots_in_factory_environment

Isaac Sim Documentation. (2025, April 21). (NVIDIA) Retrieved May 15, 2025, from https://docs.isaacsim.omniverse.nvidia.com/latest/index.html

Kortenkamp, D. (1994). *Perception for mobile robot navigation: A survey of the state of the art.* NASA. Retrieved from https://www.researchgate.net/publication/23875307_Perception_for_mobile_robot_navigation_A_survey_of_the_state_of_the_art

Li, P., Chen, D., Wang, Y., Zhang, L., & Zhao, S. (2024). Path planning of mobile robot based on improved TD3 algorithm. *Heliyon*. https://doi.org/https://doi.org/10.1016/j.heliyon.2024.e32167

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., . . . Wierstra, D. (2016). Continuous control with deep reinforcement learning. International Conference on Learning Representations (ICLR).

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier Nonlinearities Improve Neural Network Acoustic Models. *Rectifier Nonlinearities Improve Neural Network Acoustic Models.* Retrieved May 2025, from https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf

Mahrous, M., Magdy, M., Mahmoud, A., Mohamed, M., Mohamed, T., Mourad, A. F., & Kamal, S. A. (2022). *Autonomous Navigation for Indoor Mobile Robot: Hardware Implementation.* Instute of Aviation Engineering and Technology, Deparment of Electronics and Communication Engineering. Retrieved from https://www.researchgate.net/publication/365800083_Autonomous_Navigation_for_Indoor_Mobile_Robot_Hardware_Implementation

Mitchell, T. M. (1997). *Mahine Learning.* McGraw-Hill Science/Engineering/Math.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). *Playing Atari with Deep Reinforcement Learning.* Retrieved from https://arxiv.org/abs/1312.5602

Naeem, M., Rizvi, S., & Coronato, A. (2020). A Gentle Introduction to Reinforcement Learning and Its Application in Different Fields. *IEEE Access, 8*.

Nizam, M. (2024, February 2). Retrieved May 21, 2025, from https://www.linkedin.com/pulse/neural-networks-from-biological-artificial-muhammed-nizam-pcxwf/

Ojeda, J. (2025, January 26). *Applied Reinforcement Learning III: Deep Q-Networks (DQN)*. Retrieved from Towards Data Science: https://towardsdatascience.com/applied-reinforcement-learning-iii-deep-q-networks-dqn-8f0e38196ba9/

Petrov, P. (2010). Modeling and adaptive path control of a differential drive mobile robot. *12th WSEAS International Conference on AUTOMATIC CONTROL, MODELLING & SIMULATION.* Retrieved from https://www.researchgate.net/publication/228379200

Prince, S. J. (2023). *Understanding Deep Learning.* The MIT Press. Retrieved from http://udlbook.com

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra,, D., & Riedmiller, M. (2014). Deterministic Policy Gradient Algorithms. *The 31st International Conference on Machine Learning*, *32.* Beijing.

Song, X., Gao, H., Ding, T., Gu, Y., Liu, J., & Tian, K. (2023, July 4). A Review of the Motion Planning and Control Methods for Automated Vehicles. *Sensors*. Retrieved from https://www.researchgate.net/publication/372369350_A_Review_of_the_Motion_Planning_and_Control_Methods_for_Automated_Vehicles

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction (Second Edition).* The MIT Press. Retrieved from http://incompleteideas.net/book/the-book-2nd.html

Vaidya, G. Y. (2021). *Deep Reinforcement Learning for Autonomous and Navigation of Mobile Robots in Indoor Environments.* Texas A&M University, Computer Engineering. Retrieved from https://oaktrust.library.tamu.edu/items/3398bbe5-4895-4bc1-9863-935bac717312

W3Schools.com. (2025, February 19). Retrieved May 22, 2025, from https://www.w3schools.com/ai/ai_perceptrons.asp

APPENDIX A PYTHON CODE

1. buffer.py

```
1. import numpy as np
2.
3. class ReplayBuffer():
4.     def __init__(self, max_size, input_shape, n_actions):
5.         self.mem_size = max_size
6.         self.mem_cntr = 0
7.         #self.state_memory = np.zeros((self.mem_size, *input_shape))
8.         self.state_memory = np.zeros((self.mem_size, np.prod(in-
   put_shape)))
9.
10.        self.new_state_memory = np.zeros((self.mem_size, np.prod(in-
   put_shape)))
11.
12.        self.action_memory = np.zeros((self.mem_size, n_actions))
13.        self.reward_memory = np.zeros(self.mem_size)
14.        self.terminal_memory = np.zeros(self.mem_size, dtype=np.bool_)
15.
16.    def store_transition(self, state, action, reward, state_, done):
17.        index = self.mem_cntr % self.mem_size
18.        self.state_memory[index] = state
19.        self.action_memory[index] = action
20.        self.reward_memory[index] = reward
21.        self.new_state_memory[index] = state
22.        self.terminal_memory[index] = done
23.
24.        self.mem_cntr += 1
25.
26.    def sample_buffer(self, batch_size):
27.        max_mem = min(self.mem_cntr, self.mem_size)
28.
29.        batch = np.random.choice(max_mem, batch_size)
30.
31.        states = self.state_memory[batch]
32.        actions = self.action_memory[batch]
33.        rewards = self.reward_memory[batch]
34.        states_ = self.new_state_memory[batch]
35.        dones = self.terminal_memory[batch]
36.
37.        return states, actions, rewards, states_, dones
38.
```

2. nnets_4td3.py

```
1. import os
2. import torch as T
3. import torch.nn as nn
4. import torch.nn.functional as F
5. import torch.optim as optim
6. import numpy as np
7.
8. class CriticNetwork(nn.Module):
9.     def __init__(self, beta, input_dims, fc1_dims, fc2_dims, n_actions,
10.                  name, chkpt_dir='tmp/td3'):
11.         super(CriticNetwork, self).__init__()
12.         self.input_dims = input_dims
13.         self.fc1_dims = fc1_dims
14.         self.fc2_dims = fc2_dims
15.         self.n_actions = n_actions
16.         self.name = name
17.         self.checkpoint_dir = chkpt_dir
18.         os.makedirs(self.checkpoint_dir, exist_ok=True)
19.         self.checkpoint_file = os.path.join(self.checkpoint_dir, name +
    '_td3')
20.
21.         self.fc1 = nn.Linear(np.prod(self.input_dims) + n_actions,
    self.fc1_dims)
22.         self.fc2 = nn.Linear(self.fc1_dims, self.fc2_dims)
23.         self.q1 = nn.Linear(self.fc2_dims, 1)
24.
25.         self.optimizer = optim.Adam(self.parameters(), lr=beta)
26.         self.device = T.device('cuda:0' if T.cuda.is_available() else
    'cpu')
27.         self.to(self.device)
28.
29.     def forward(self, state, action):
30.         q1_action_value = self.fc1(T.cat([state, action], dim=1))
31.         q1_action_value = F.relu(q1_action_value)
32.         q1_action_value = self.fc2(q1_action_value)
33.         q1_action_value = F.relu(q1_action_value)
34.
35.         q1 = self.q1(q1_action_value)
36.         return q1
37.
38.     def save_checkpoint(self):
39.         print('...saving checkpoint...')
40.         T.save(self.state_dict(), self.checkpoint_file)
41.
42.     def load_checkpoint(self):
```

```
        print('...loading checkpoint...')
        self.load_state_dict(T.load(self.checkpoint_file))

class ActorNetwork(nn.Module):
    def __init__(self, alpha, input_dims, fc1_dims, fc2_dims,
                 n_actions, name, chkpt_dir='tmp/td3'):
        super(ActorNetwork, self).__init__()
        self.input_dims = input_dims
        self.fc1_dims = fc1_dims
        self.fc2_dims = fc2_dims
        self.n_actions = n_actions
        self.name = name
        self.checkpoint_dir = chkpt_dir
        os.makedirs(self.checkpoint_dir, exist_ok=True)
        self.checkpoint_file = os.path.join(self.checkpoint_dir, name +
    '_td3')

        self.fc1 = nn.Linear(np.prod(self.input_dims), self.fc1_dims)

        self.fc2 = nn.Linear(self.fc1_dims, self.fc2_dims)
        self.mu = nn.Linear(self.fc2_dims, self.n_actions)

        self.optimizer = optim.Adam(self.parameters(), lr=alpha)
        self.device = T.device('cuda:0' if T.cuda.is_available() else
    'cpu')
        self.to(self.device)

    def forward(self, state):
        prob = self.fc1(state)
        prob = F.relu(prob)
        prob = self.fc2(prob)
        prob = F.relu(prob)
        prob = T.tanh(self.mu(prob))

        return prob

    def save_checkpoint(self):
        print('...saving checkpoint...')
        T.save(self.state_dict(), self.checkpoint_file)

    def load_checkpoint(self):
        print('...loading checkpoint...')
        self.load_state_dict(T.load(self.checkpoint_file))

```

3. Td3_agent.py

```
import os
import torch as T
import torch.nn.functional as F
import numpy as np
from nnets_4td3 import ActorNetwork, CriticNetwork
from buffer import ReplayBuffer
from torch.utils.tensorboard import SummaryWriter
from datetime import datetime

class Agent():
    def __init__(self, alpha, beta, input_dims, tau, env,
                 gamma=0.99, update_actor_interval=2, warmup=1000,
                 n_actions=2, max_size=1000000, layer1_size=400,
                 layer2_size=300, batch_size=100, noise=0.1):
        self.gamma = gamma
        self.tau = tau
        self.max_action = 50.0 # or env.action_space.high
        self.min_action = -50.0 # orenv.action_space.low
        self.memory = ReplayBuffer(max_size, input_dims, n_actions)
        self.batch_size = batch_size
        self.learn_step_cntr = 0
        self.time_step = 0
        self.warmup = warmup
        self.n_actions = n_actions
        self.update_actor_iter = update_actor_interval
        self.noise = noise
        self.eval_mode = False

        self.actor = ActorNetwork(alpha, input_dims, layer1_size,
                                  layer2_size, n_actions=n_actions,
                                  name='actor')
        self.critic_1 = CriticNetwork(beta, input_dims, layer1_size,
                                      layer2_size, n_actions=n_actions,
                                      name='critic_1')
        self.critic_2 = CriticNetwork(beta, input_dims, layer1_size,
                                      layer2_size, n_actions=n_actions,
                                      name='critic_2')

        self.target_actor = ActorNetwork(alpha, input_dims, layer1_size,
                                         layer2_size, n_actions=n_ac-
tions,
                                         name='target_actor')
        self.target_critic_1 = CriticNetwork(beta, input_dims,
layer1_size,
```

```
43.                                             layer2_size, n_actions=n_ac-
   tions,
44.                                             name='target_critic_1')
45.         self.target_critic_2 = CriticNetwork(beta, input_dims,
   layer1_size,
46.                                             layer2_size, n_actions=n_ac-
   tions,
47.                                             name='target_critic_2')
48.
49.         #run_name = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
50.         #base_dir = os.path.dirname(os.path.abspath(__file__))
51.         #log_dir = os.path.join(base_dir, "logs", run_name)
52.         #os.makedirs(log_dir, exist_ok=True)
53.         #self.tb_writer = SummaryWriter(log_dir=log_dir)
54.         #self.loss_log_csv = os.path.join(log_dir, "loss_log.csv")
55.         #with open(self.loss_log_csv, "w") as f:
56.         #    f.write("step,critic1,critic2,critic_total,actor\n")
57.
58.
59.         self.update_network_parameters(tau=1)
60.
61.     def choose_action(self, observation):
62.         def extract_scalar(x):
63.             if isinstance(x, np.ndarray):
64.                 return float(x.flat[0])
65.             return float(x)
66.
67.         min_val = extract_scalar(self.min_action)
68.         max_val = extract_scalar(self.max_action)
69.
70.         if self.eval_mode:
71.             state = T.tensor(observation, dtype=T.float).to(self.ac-
   tor.device)
72.             mu = self.actor.forward(state).to(self.actor.device)
73.             mu_prime = T.clamp(mu, min_val, max_val)
74.             return mu_prime.cpu().detach().numpy()
75.
76.         self.time_step += 1
77.         if self.time_step < self.warmup:
78.             mu = T.tensor(np.random.normal(scale=self.noise,
   size=(self.n_actions,)),
79.                           dtype=T.float, device=self.actor.device)
80.         else:
81.             state = T.tensor(observation, dtype=T.float).to(self.ac-
   tor.device)
82.             mu = self.actor.forward(state).to(self.actor.device)
```

```
83.
84.         noise = T.tensor(np.random.normal(scale=self.noise,
    size=(self.n_actions,)),
85.                          dtype=T.float, device=self.actor.device)
86.         mu_prime = mu + noise
87.         mu_prime = T.clamp(mu_prime, min_val, max_val)
88.         return mu_prime.cpu().detach().numpy()
89.



90.
91.     def remember(self, state, action, reward, new_state, done):
92.         self.memory.store_transition(state, action, reward, new_state,
    done)
93.
94.     def learn(self):
95.         if self.memory.mem_cntr < self.batch_size:
96.             return
97.         # Sample a mini-batch from the replay buffer
98.         state, action, reward, new_state, done = self.memory.sam-
    ple_buffer(self.batch_size)
99.         # Convert to PyTorch tensors
100.                reward = T.tensor(reward,
    dtype=T.float).to(self.critic_1.device)
101.                done = T.tensor(done, dtype=T.bool).to(self.critic_1.de-
    vice)  # Ensure it's a boolean tensor
102.                state_ = T.tensor(new_state,
    dtype=T.float).to(self.critic_1.device)
103.                state = T.tensor(state,
    dtype=T.float).to(self.critic_1.device)
104.                action = T.tensor(action,
    dtype=T.float).to(self.critic_1.device)
105.
106.                # Compute target actions with policy smoothing
107.                target_actions = self.target_actor.forward(state_)
108.                noise = T.tensor(np.random.normal(0, 0.2, target_ac-
    tions.shape), dtype=T.float).to(self.critic_1.device)
109.                target_actions = target_actions + T.clamp(noise, -0.5,
    0.5)  # Clipped noise
110.                #target_actions = T.clamp(target_actions, self.min_ac-
    tion[0], self.max_action[0])  # Ensure valid actions
111.                target_actions = T.clamp(target_actions, self.min_action,
    self.max_action)
112.
113.                # Compute target Q-values
114.                q1_ = self.target_critic_1.forward(state_, target_actions)
```

```
            q2_ = self.target_critic_2.forward(state_, target_actions)

            # Compute current Q-values
            q1 = self.critic_1.forward(state, action)
            q2 = self.critic_2.forward(state, action)

            # Take the min of target Q-values (TD3 trick)
            q1_ = q1_.view(-1)
            q2_ = q2_.view(-1)
            critic_value_ = T.min(q1_, q2_)

            # Apply terminal condition: zero out next state's value when done
            critic_value_[done] = 0.0

            # Compute TD3 target
            target = reward + self.gamma * critic_value_
            target = target.view(self.batch_size, 1)

            # Compute critic loss and update critics
            self.critic_1.optimizer.zero_grad()
            self.critic_2.optimizer.zero_grad()

            q1_loss = F.mse_loss(q1, target)  # MSE loss for critic 1
            q2_loss = F.mse_loss(q2, target)  # MSE loss for critic 2
            critic_loss = q1_loss + q2_loss  # Total critic loss
            critic_loss.backward()
            self.critic_1.optimizer.step()
            self.critic_2.optimizer.step()

            # 🟢 Log critic loss
            #self.tb_writer.add_scalar("Loss/Critic1", q1_loss.item(), self.learn_step_cntr)
            #self.tb_writer.add_scalar("Loss/Critic2", q2_loss.item(), self.learn_step_cntr)
            #self.tb_writer.add_scalar("Loss/CriticTotal", critic_loss.item(), self.learn_step_cntr)
            #with open(self.loss_log_csv, "a") as f:
            #    f.write(f"{self.learn_step_cntr},{q1_loss.item()},{q2_loss.item()},{critic_loss.item()},\n")

            # Delayed policy update
            self.learn_step_cntr += 1
            if self.learn_step_cntr % self.update_actor_iter != 0:
                return
```

```
155.
156.             # Compute actor loss and update actor
157.             self.actor.optimizer.zero_grad()
158.             actor_q1_loss = self.critic_1.forward(state, self.ac-
    tor.forward(state))
159.             actor_loss = -T.mean(actor_q1_loss)  # Gradient ascent on
    Q-value
160.
161.             actor_loss.backward()
162.             self.actor.optimizer.step()
163.
164.             # 🟢 Log actor loss
165.             #self.tb_writer.add_scalar("Loss/Actor", ac-
    tor_loss.item(), self.learn_step_cntr)
166.             #with open(self.loss_log_csv, "a") as f:
167.             #    f.write(f"{self.learn_step_cntr},,,{ac-
    tor_loss.item()}\n")
168.
169.             # Update target networks
170.             self.update_network_parameters()
171.
172.         def update_network_parameters(self, tau=None):
173.             if tau is None:
174.                 tau = self.tau
175.
176.             actor_params = self.actor.named_parameters()
177.             critic_1_params = self.critic_1.named_parameters()
178.             critic_2_params = self.critic_2.named_parameters()
179.             target_actor_params = self.target_actor.named_parameters()
180.             target_critic_1_params = self.target_critic_1.named_param-
    eters()
181.             target_critic_2_params = self.target_critic_2.named_param-
    eters()
182.
183.             critic_1_state_dict = dict(critic_1_params)
184.             critic_2_state_dict = dict(critic_2_params)
185.             actor_state_dict = dict(actor_params)
186.             target_actor_state_dict = dict(target_actor_params)
187.             target_critic_1_state_dict = dict(target_critic_1_params)
188.             target_critic_2_state_dict =  dict(target_critic_2_params)
189.
190.             for name in critic_1_state_dict:
191.                 critic_1_state_dict[name] = tau *
    critic_1_state_dict[name].clone() + \
192.                         (1 - tau) * tar-
    get_critic_1_state_dict[name].clone()
```

```
193.
194.             for name in critic_2_state_dict:
195.                 critic_2_state_dict[name] = tau *
    critic_2_state_dict[name].clone() + \
196.                         (1 - tau) * tar-
    get_critic_2_state_dict[name].clone()
197.
198.             for name in actor_state_dict:
199.                 actor_state_dict[name] = tau * ac-
    tor_state_dict[name].clone() + \
200.                         (1 - tau) * target_ac-
    tor_state_dict[name].clone()
201.
202.             self.target_critic_1.load_state_dict(critic_1_state_dict)
203.             self.target_critic_2.load_state_dict(critic_2_state_dict)
204.             self.target_actor.load_state_dict(actor_state_dict)
205.
206.         # Save models (optional)
207.         def save_models(self):
208.             self.actor.save_checkpoint()
209.             self.target_actor.save_checkpoint()
210.             self.critic_1.save_checkpoint()
211.             self.critic_2.save_checkpoint()
212.             self.target_critic_1.save_checkpoint()
213.             self.target_critic_2.save_checkpoint()
214.
215.         # Load models (optional)
216.         def load_models(self):
217.             self.actor.load_checkpoint()
218.             self.target_actor.load_checkpoint()
219.             self.critic_1.load_checkpoint()
220.             self.critic_2.load_checkpoint()
221.             self.target_critic_1.load_checkpoint()
222.             self.target_critic_2.load_checkpoint()
223.
224.         # Set evaluation mode
225.         def set_eval_mode(self):
226.             self.eval_mode = True
227.             self.actor.eval()
228.             self.target_actor.eval()
229.             self.critic_1.eval()
230.             self.critic_2.eval()
231.             self.target_critic_1.eval()
232.             self.target_critic_2.eval()
233.
234.         # Set training mode
```

```
    def set_train_mode(self):
        self.eval_mode = False
        self.actor.train()
        self.target_actor.train()
        self.critic_1.train()
        self.critic_2.train()
        self.target_critic_1.train()
        self.target_critic_2.train()
```

4. carter_env.py

```
from __future__ import annotations

import torch
from collections.abc import Sequence
import isaaclab.sim as sim_utils
from isaaclab.assets import Articulation, ArticulationCfg
from isaaclab.envs import DirectRLEnv, DirectRLEnvCfg
from isaaclab.scene import InteractiveSceneCfg
from isaaclab.sim import SimulationCfg
from isaaclab.sim.spawners.from_files import GroundPlaneCfg,
spawn_ground_plane
from isaaclab.utils import configclass
from .waypoint import WAYPOINT_CFG
from isaaclab_assets.robots.carter import CARTER_CFG
from isaaclab.markers import VisualizationMarkers

@configclass
class CarterEnvCfg(DirectRLEnvCfg):
    decimation = 4
    episode_length_s = 20.0
    action_space = 2
    observation_space = 7
    state_space = 0
    sim: SimulationCfg = SimulationCfg(dt=1 / 60, render_interval=decima-
tion)
    robot_cfg: ArticulationCfg = CARTER_CFG.re-
place(prim_path="/World/envs/env_.*/Robot")
    waypoint_cfg = WAYPOINT_CFG

    throttle_dof_name = ["left_wheel", "right_wheel"]
    env_spacing = 32.0
    course_length_coefficient = 2.5
    course_width_coefficient = 2.0
```

```
    scene: InteractiveSceneCfg = InteractiveSceneCfg(num_envs=4096,
env_spacing=env_spacing, replicate_physics=True)

class CarterEnv(DirectRLEnv):
    cfg: CarterEnvCfg

    def __init__(self, cfg: CarterEnvCfg, render_mode: str | None = None,
**kwargs):
        super().__init__(cfg, render_mode, **kwargs)
        self._throttle_dof_idx = None
        self.object_state = []
        self.env_spacing = cfg.env_spacing
        self.course_length_coefficient = cfg.course_length_coefficient
        self.course_width_coefficient = cfg.course_width_coefficient
        self._num_goals = 10

        self._target_positions = None
        self._markers_pos = None
        self._target_index = None
        self._position_error = None
        self._previous_position_error = None
        self._throttle_action = None
        self._throttle_state = None

    def _setup_scene(self):
        spawn_ground_plane(
            prim_path="/World/ground",
            cfg=GroundPlaneCfg(
                size=(500.0, 500.0),
                color=(0.2, 0.2, 0.2),
                physics_material=sim_utils.RigidBodyMaterialCfg(
                    friction_combine_mode="multiply",
                    restitution_combine_mode="multiply",
                    static_friction=1.0,
                    dynamic_friction=1.0,
                    restitution=0.0,
                ),
            ),
        )
        self.scene.clone_environments(copy_from_source=False)
        self.scene.filter_collisions(global_prim_paths=[])

        self.carter = Articulation(self.cfg.robot_cfg)
        self.scene.articulations["carter"] = self.carter
        self.waypoints = VisualizationMarkers(self.cfg.waypoint_cfg)

```

```
75.         light_cfg = sim_utils.DomeLightCfg(intensity=2000.0, color=(0.75,
    0.75, 0.75))
76.         light_cfg.func("/World/Light", light_cfg)
77.
78.     def post_reset(self):
79.         self._throttle_dof_idx, _ =
    self.carter.find_joints(self.cfg.throttle_dof_name)
80.
81.         self._throttle_state = torch.zeros((self.num_envs, 2), de-
    vice=self.device)
82.         self.task_completed = torch.zeros((self.num_envs), de-
    vice=self.device, dtype=torch.bool)
83.         self._goal_reached = torch.zeros((self.num_envs), device=self.de-
    vice, dtype=torch.int32)
84.         self._target_index = torch.zeros((self.num_envs), device=self.de-
    vice, dtype=torch.int32)
85.         self._target_positions = torch.zeros((self.num_envs,
    self._num_goals, 2), device=self.device)
86.         self._markers_pos = torch.zeros((self.num_envs, self._num_goals,
    3), device=self.device)
87.
88.         self.position_tolerance = 0.15
89.         self.goal_reached_bonus = 10.0
90.         self.position_progress_weight = 1.0
91.         self.heading_coefficient = 0.25
92.         self.heading_progress_weight = 0.05
93.
94.     def _pre_physics_step(self, actions: torch.Tensor) -> None:
95.         throttle_scale = 10.0
96.         throttle_max = 50.0
97.         self._throttle_action = actions[:, :2] * throttle_scale
98.         self._throttle_action = torch.clamp(self._throttle_action, -
    throttle_max, throttle_max)
99.         self._throttle_state = self._throttle_action
100.
101.          def _apply_action(self) -> None:
102.              self.carter.set_joint_velocity_target(self._throttle_ac-
    tion, joint_ids=self._throttle_dof_idx)
103.
104.          def _get_observations(self) -> dict:
105.              current_target_positions = self._target_posi-
    tions[self.carter._ALL_INDICES, self._target_index]
106.              self._position_error_vector = current_target_positions -
    self.carter.data.root_pos_w[:, :2]
107.              self._previous_position_error = self._position_er-
    ror.clone()
```

```
108.             self._position_error = torch.norm(self._position_er-
    ror_vector, dim=-1)
109.
110.             heading = self.carter.data.heading_w
111.             target_heading_w = torch.atan2(
112.                 self._target_positions[self.carter._ALL_INDICES,
    self._target_index, 1] - self.carter.data.root_link_pos_w[:, 1],
113.                 self._target_positions[self.carter._ALL_INDICES,
    self._target_index, 0] - self.carter.data.root_link_pos_w[:, 0],
114.             )
115.             self.target_heading_error = torch.atan2(torch.sin(tar-
    get_heading_w - heading), torch.cos(target_heading_w - heading))
116.
117.             obs = torch.cat(
118.                 (
119.                     self._position_error.unsqueeze(dim=1),
120.                     torch.cos(self.target_heading_er-
    ror).unsqueeze(dim=1),
121.                     torch.sin(self.target_heading_er-
    ror).unsqueeze(dim=1),
122.                     self.carter.data.root_lin_vel_b[:,
    0].unsqueeze(dim=1),
123.                     self.carter.data.root_lin_vel_b[:,
    1].unsqueeze(dim=1),
124.                     self.carter.data.root_ang_vel_w[:,
    2].unsqueeze(dim=1),
125.                     self._throttle_state[:, 0].unsqueeze(dim=1),
126.                 ),
127.                 dim=-1,
128.             )
129.
130.             if torch.any(obs.isnan()):
131.                 raise ValueError("Observations cannot be NAN")
132.
133.             return {"policy": obs}
134.
135.         def _get_rewards(self) -> torch.Tensor:
136.             position_progress_rew = self._previous_position_error -
    self._position_error
137.             target_heading_rew = torch.exp(-torch.abs(self.tar-
    get_heading_error) / self.heading_coefficient)
138.             goal_reached = self._position_error < self.position_toler-
    ance
139.             self._target_index = self._target_index + goal_reached
140.             self.task_completed = self._target_index >
    (self._num_goals - 1)
```

```
141.             self._target_index = self._target_index % self._num_goals
142.
143.             composite_reward = (
144.                 position_progress_rew * self.position_progress_weight
    +
145.                 target_heading_rew * self.heading_progress_weight +
146.                 goal_reached * self.goal_reached_bonus
147.             )
148.
149.             one_hot_encoded = torch.nn.functional.one_hot(self._tar-
    get_index.long(), num_classes=self._num_goals)
150.             marker_indices = one_hot_encoded.view(-1).tolist()
151.             self.waypoints.visualize(marker_indices=marker_indices)
152.
153.             if torch.any(composite_reward.isnan()):
154.                 raise ValueError("Rewards cannot be NAN")
155.
156.             return composite_reward
157.
158.         def _get_dones(self) -> tuple[torch.Tensor, torch.Tensor]:
159.             task_failed = self.episode_length_buf > self.max_epi-
    sode_length
160.             return task_failed, self.task_completed
161.
162.         def _reset_idx(self, env_ids: Sequence[int] | None):
163.             if env_ids is None:
164.                 env_ids = self.carter._ALL_INDICES
165.             super()._reset_idx(env_ids)
166.
167.             num_reset = len(env_ids)
168.             default_state = self.carter.data.de-
    fault_root_state[env_ids]
169.             carter_pose = default_state[:, :7]
170.             carter_velocities = default_state[:, 7:]
171.             joint_positions = self.carter.data.de-
    fault_joint_pos[env_ids]
172.             joint_velocities = self.carter.data.de-
    fault_joint_vel[env_ids]
173.
174.             carter_pose[:, :3] += self.scene.env_origins[env_ids]
175.             carter_pose[:, 0] -= self.env_spacing / 2
176.             carter_pose[:, 1] += 2.0 * torch.rand((num_reset),
    dtype=torch.float32, device=self.device) * self.course_width_coefficient
177.
178.             angles = torch.pi / 6.0 * torch.rand((num_reset),
    dtype=torch.float32, device=self.device)
```

```
            carter_pose[:, 3] = torch.cos(angles * 0.5)
            carter_pose[:, 6] = torch.sin(angles * 0.5)

            self.carter.write_root_pose_to_sim(carter_pose, env_ids)
            self.carter.write_root_velocity_to_sim(carter_velocities,
env_ids)
            self.carter.write_joint_state_to_sim(joint_positions,
joint_velocities, None, env_ids)

            if self._target_positions is None:
                self._target_positions = torch.zeros((self.num_envs,
self._num_goals, 2), device=self.device)
            if self._markers_pos is None:
                self._markers_pos = torch.zeros((self.num_envs,
self._num_goals, 3), device=self.device)
            if self._target_index is None:
                self._target_index = torch.zeros((self.num_envs),
dtype=torch.int32, device=self.device)
            if self._position_error is None:
                self._position_error = torch.zeros((self.num_envs),
device=self.device)
            if self._previous_position_error is None:
                self._previous_position_error = torch.ze-
ros((self.num_envs), device=self.device)

            self._target_positions[env_ids, :, :] = 0.0
            self._markers_pos[env_ids, :, :] = 0.0

            spacing = 2 / self._num_goals
            target_positions = torch.arange(-0.8, 1.1, spacing, de-
vice=self.device) * self.env_spacing / self.course_length_coefficient
            self._target_positions[env_ids, :len(target_positions), 0]
= target_positions
            self._target_positions[env_ids, :, 1] =
torch.rand((num_reset, self._num_goals), dtype=torch.float32, de-
vice=self.device) + self.course_length_coefficient
            self._target_positions[env_ids, :] += self.scene.env_ori-
gins[env_ids, :2].unsqueeze(1)

            self._target_index[env_ids] = 0
            self._markers_pos[env_ids, :, :2] = self._target_posi-
tions[env_ids]
            visualize_pos = self._markers_pos.view(-1, 3)
            self.waypoints.visualize(translations=visualize_pos)

```

```
            current_target_positions = self._target_positions[self.carter._ALL_INDICES, self._target_index]
            self._position_error_vector = current_target_positions[:, :2] - self.carter.data.root_pos_w[:, :2]
            self._position_error = torch.norm(self._position_error_vector, dim=-1)
            self._previous_position_error = self._position_error.clone()

            heading = self.carter.data.heading_w[:]
            target_heading_w = torch.atan2(
                self._target_positions[:, 0, 1] - self.carter.data.root_pos_w[:, 1],
                self._target_positions[:, 0, 0] - self.carter.data.root_pos_w[:, 0],
            )
            self._heading_error = torch.atan2(torch.sin(target_heading_w - heading), torch.cos(target_heading_w - heading))
            self._previous_heading_error = self._heading_error.clone()

            self.post_reset()

    from isaaclab_tasks.utils.registry import register_agent_cfg

    @register_agent_cfg(name="skrl_td3_cfg_entry_point")
    def build_skrl_td3_cfg():
        import os
        import yaml

        config_path = os.path.join(os.path.dirname(__file__), "agents", "skrl_td3_cfg.yaml")
        with open(config_path, "r") as f:
            cfg = yaml.safe_load(f)
        return cfg

    __all__ = ["CarterEnvCfg", "CarterEnv", "build_skrl_td3_cfg"]

```

5. train_td3_carter_v1.py

```
import os
import sys
import numpy as np
import torch
from torch.utils.tensorboard import SummaryWriter
import matplotlib.pyplot as plt
```

```
from datetime import datetime


# Add Isaac Lab and Isaac Lab Tasks to PYTHONPATH
sys.path.append(os.path.abspath(os.path.join(__file__,
    "../../../../source/isaaclab")))
sys.path.append(os.path.abspath(os.path.join(__file__,
    "../../../../source/isaaclab_tasks")))

from isaaclab.app import AppLauncher
app_launcher = AppLauncher(headless=True)
simulation_app = app_launcher.app

from td3_agent import Agent
from isaaclab_tasks.direct.carter.carter_env import CarterEnv, CarterEn-
    vCfg

# ====================== CONFIG ======================
run_name = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
base_dir = os.path.dirname(os.path.abspath(__file__))
log_dir = os.path.join(base_dir, "logs", run_name)
ckpt_dir = os.path.join(base_dir, "checkpoints")
os.makedirs(log_dir, exist_ok=True)
os.makedirs(ckpt_dir, exist_ok=True)

plot_path = os.path.join(log_dir, "reward_plot.png")
step_plot_path = os.path.join(log_dir, "reward_step_plot.png")
csv_path = os.path.join(log_dir, "rewards.csv")
step_csv_path = os.path.join(log_dir, "reward_steps.csv")

tb_writer = SummaryWriter(log_dir=log_dir)

# ====================== ENV + AGENT ======================
env_cfg = CarterEnvCfg()
env_cfg.scene.num_envs = 1
env_cfg.sim.device = "cuda" if torch.cuda.is_available() else "cpu"
env = CarterEnv(cfg=env_cfg)

agent = Agent(
    alpha=1e-4, beta=1e-3,
    input_dims=env.observation_space.shape,
    tau=0.005, env=env,
    batch_size=100, layer1_size=400, layer2_size=300,
    n_actions=env.action_space.shape[0]
)

```

```
49. # ====================== TRAINING ======================
50. n_games = 1000
51. scores = []
52. step_rewards = []
53. global_step = 0
54. 
55. episode_steps = []
56. cumulative_steps = 0
57. with open(csv_path, "w") as f:
58.     f.write("episode,score,avg100\n")
59. with open(step_csv_path, "w") as f:
60.     f.write("step,reward\n")
61. 
62. for i in range(n_games):
63.     obs, _ = env.reset()
64.     done = False
65.     score = 0
66.     step_count = 0  #  ◇ Initialize for each episode
67. 

68.     while not done:
69.         action = agent.choose_action(obs['policy'])
70.         action_tensor = torch.tensor(action, dtype=torch.float32, de-
    vice=env.device)
71.         if action_tensor.ndim == 1:
72.             action_tensor = action_tensor.unsqueeze(0)
73.         obs_, reward, terminated, truncated, _ = env.step(action_tensor)
74.         done = terminated or truncated
75. 
76.         agent.remember(obs['policy'].cpu().numpy(), action, reward,
    obs_['policy'].cpu().numpy(), done)
77.         agent.learn()
78.         obs = obs_
79.         score += reward
80.         global_step += 1
81. 
82.         # Log per-step reward
83.         step_reward = reward.cpu().item() if torch.is_tensor(reward) else
    reward
84.         step_rewards.append((global_step, step_reward))
85.         tb_writer.add_scalar("Reward/Step", step_reward, global_step)
86.         with open(step_csv_path, "a") as f:
87.             f.write(f"{global_step},{step_reward}\n")
88.         step_count += 1
89.         cumulative_steps += 1
90. 
```

```
91.     episode_steps.append(cumulative_steps)
92.
93.     score_val = score.cpu().item() if torch.is_tensor(score) else score
94.     scores.append(score_val)
95.     avg_score = np.mean(scores[max(0, i-99):])
96.     print(f"Episode {i}, Score: {score_val:.2f}, Avg(100):
    {avg_score:.2f}")
97.
98.     tb_writer.add_scalar("Reward/Episode", score_val, i)
99.     tb_writer.add_scalar("Reward/Avg100", avg_score, i)
100.
101.          with open(csv_path, "a") as f:
102.              f.write(f"{i},{score_val},{avg_score}\n")
103.
104.          if (i + 1) % 25 == 0:
105.              agent.actor.checkpoint_dir = ckpt_dir
106.              agent.critic_1.checkpoint_dir = ckpt_dir
107.              agent.critic_2.checkpoint_dir = ckpt_dir
108.              agent.target_actor.checkpoint_dir = ckpt_dir
109.              agent.target_critic_1.checkpoint_dir = ckpt_dir
110.              agent.target_critic_2.checkpoint_dir = ckpt_dir
111.              agent.save_models()
112.
113.      # ========== Smoothing function ==========
114.      def smooth(data, weight=0.9):
115.          smoothed = []
116.          last = data[0]
117.          for point in data:
118.              smoothed_val = last * weight + (1 - weight) * point
119.              smoothed.append(smoothed_val)
120.              last = smoothed_val
121.          return smoothed
122.
123.      # ====================== PLOT REWARD ======================
124.      # --- Plot Reward per Episode vs Time-Steps ---
125.
126.      plt.figure()
127.      plt.plot(episode_steps, [s.item() if torch.is_tensor(s) else s for
    s in scores], marker='o')
128.      plt.title("Total Reward per Episode vs Time Steps")
129.      plt.xlabel("Cumulative Time Steps")
130.      plt.ylabel("Episode Total Reward")
131.      plt.grid(True)
132.
```

```
133.      plot_vs_steps_path = os.path.join(log_dir, "plots", "re-
   ward_vs_steps.png")
134.      os.makedirs(os.path.dirname(plot_vs_steps_path), exist_ok=True)
135.      plt.savefig(plot_vs_steps_path)
136.      print(f"[☑] Saved reward vs steps plot to: {plot_vs_steps_path}")
137.
138.      # Plot average reward per episode
139.      plt.figure()
140.      avg = [np.mean(scores[max(0, i-99):(i+1)]) for i in
   range(len(scores))]
141.      plt.plot(avg)
142.      plt.title("Running Avg Reward (100 episodes)")
143.      plt.xlabel("Episode")
144.      plt.ylabel("Reward")
145.      plt.grid()
146.      plt.savefig(plot_path)
147.
148.      # Plot reward vs step
149.      steps, rewards = zip(*step_rewards)
150.      plt.figure()
151.      plt.plot(steps, rewards)
152.      plt.title("Reward per Step")
153.      plt.xlabel("Step")
154.      plt.ylabel("Reward")
155.      plt.grid()
156.      plt.savefig(step_plot_path)
157.
158.      # --- Plot Smoothed Episode Reward vs Cumulative Steps ---
159.      smoothed_scores = smooth(scores)
160.
161.      plt.figure()
162.      plt.plot(episode_steps, smoothed_scores, label="TD3",
   color="blue")
163.      plt.title("Smoothed Episode Total Reward vs Time Steps")
164.      plt.xlabel("Cumulative Time Steps")
165.      plt.ylabel("Episode Reward")
166.      plt.grid(True)
167.      plt.legend()
168.
169.      plot_vs_steps_path = os.path.join(log_dir, "plots", "re-
   ward_vs_steps_smoothed.png")
170.      os.makedirs(os.path.dirname(plot_vs_steps_path), exist_ok=True)
171.      plt.savefig(plot_vs_steps_path)
172.      print(f"[☑] Saved smoothed reward vs steps plot to:
   {plot_vs_steps_path}")
173.
```

```
    env.close()
    tb_writer.close()
    simulation_app.close()
```

6. evaluate_td3_carter.py

```
import os
import torch
import numpy as np
import matplotlib.pyplot as plt
import imageio
import cv2

from isaaclab.app import AppLauncher
app_launcher = AppLauncher(headless=True)
simulation_app = app_launcher.app

from td3_agent import Agent
from isaaclab_tasks.direct.carter.carter_env import CarterEnv, CarterEnvCfg

# --- Log directory ---
base_dir = os.path.dirname(os.path.abspath(__file__))
logs_root = os.path.join(base_dir, "logs")

# Get latest modified directory in logs/
run_name = sorted(os.listdir(logs_root), key=lambda x: os.path.getmtime(os.path.join(logs_root, x)), reverse=True)[0]
log_dir = os.path.join(logs_root, run_name)

print(f"[INFO] Using latest log directory: {log_dir}")
os.makedirs(log_dir, exist_ok=True)  # ☑ Ensure the directory exists

gif_path = os.path.join(log_dir, "evaluation.gif")
eval_csv = os.path.join(log_dir, "eval_scores.csv")
eval_plot = os.path.join(log_dir, "eval_plot.png")


# --- Load environment ---
env_cfg = CarterEnvCfg()
env_cfg.scene.num_envs = 1
env_cfg.sim.device = "cuda" if torch.cuda.is_available() else "cpu"
env = CarterEnv(cfg=env_cfg)

# --- Load trained agent ---
agent = Agent(
    alpha=1e-4, beta=1e-3,
```

```
    input_dims=env.observation_space.shape,
    tau=0.005, env=env,
    batch_size=100, layer1_size=400, layer2_size=300,
    n_actions=env.action_space.shape[0]
)
# Override checkpoint paths
ckpt_path = "/home/rith/isaaclab/IsaacLab/tmp/td3"
agent.actor.checkpoint_dir = ckpt_path
agent.critic_1.checkpoint_dir = ckpt_path
agent.critic_2.checkpoint_dir = ckpt_path
agent.target_actor.checkpoint_dir = ckpt_path
agent.target_critic_1.checkpoint_dir = ckpt_path
agent.target_critic_2.checkpoint_dir = ckpt_path

agent.load_models()
agent.set_eval_mode()

# --- Run evaluation ---
eval_episodes = 10
eval_scores = []
frames = []

for ep in range(eval_episodes):
    obs, _ = env.reset()
    done = False
    score = 0

    while not done:
        frame = env.render()
        if frame is not None:
            frames.append(frame)

        action = agent.choose_action(obs['policy'])
        action_tensor = torch.tensor(action, dtype=torch.float32, de-
vice=env.device)
        if action_tensor.ndim == 1:
            action_tensor = action_tensor.unsqueeze(0)

        obs_, reward, terminated, truncated, _ = env.step(action_tensor)
        done = terminated or truncated
        obs = obs_
        score += reward

    score = score.cpu().item() if torch.is_tensor(score) else score
    eval_scores.append(score)
    print(f"[EVAL] Episode {ep}, Score: {score:.2f}")
```

```
84.
85. avg_eval = np.mean(eval_scores)
86. print(f"[EVAL] Average Score over {eval_episodes} episodes:
    {avg_eval:.2f}")
87.
88. # --- Save CSV ---
89. with open(eval_csv, "w") as f:
90.     f.write("episode,score\n")
91.     for ep, s in enumerate(eval_scores):
92.         f.write(f"{ep},{s}\n")
93.     f.write(f"average,{avg_eval}\n")
94. # --- Save Plot ---
95. plots_dir = os.path.join(log_dir, "plots")
96. os.makedirs(plots_dir, exist_ok=True)
97.
98. eval_plot = os.path.join(plots_dir, "eval_plot.png")
99.
100.      plt.figure()
101.      plt.plot(eval_scores, marker='o')
102.      plt.title("Evaluation Episode Scores")
103.      plt.xlabel("Episode")
104.      plt.ylabel("Score")
105.      plt.grid()
106.      plt.savefig(eval_plot)
107.      print(f"[☑] Saved plot to {eval_plot}")
108.
109.      # --- Save GIF ---
110.      if frames:
111.          imageio.mimsave(gif_path, [cv2.cvtColor(f, cv2.COLOR_RGB2BGR)
    for f in frames], fps=15)
112.          print(f"[☑] Saved evaluation GIF to: {gif_path}")
113.      else:
114.          print("⚠ No frames captured to generate GIF.")
115.
116.      # --- Cleanup ---
117.      env.close()
118.      simulation_app.close()
119.
```