

Mobile robot navigation: A study on various DRL and traditional path planning algorithms

*Report submitted to the Amrita Vishwa Vidyapeetham,
Coimbatore as the requirement for the course*

21AIE401: Deep Reinforcement Learning

Submitted by

Team-5

December 2023



AMRITA
VISHWA VIDYAPEETHAM

SCHOOL OF ARTIFICIAL INTELLIGENCE

COIMBATORE, TAMIL NADU, INDIA – 641112



AMRITA

VISHWA VIDYAPEETHAM

SCHOOL OF ARTIFICIAL INTELLIGENCE
COIMBATORE, TAMIL NADU, INDIA – 641112

Bonafide Certificate

This is to certify that the report titled “**Mobile robot navigation: A study on various DRL and traditional path planning algorithms**” submitted as a requirement for the course, **21AIE401: Deep Reinforcement Learning** for B.Tech. Computer Science and Engineering (Artificial Intelligence) program, is a bonafide record of the work done by **Team -5** during the academic year 2023-24, in the School of Arti, under my supervision.

Signature of Project Supervisor :

Name with Affiliation :

Date :

Project Viva-voce held on _____

Examiner 1

Examiner 2



AMRITA

VISHWA VIDYAPEETHAM

SCHOOL OF ARTIFICIAL INTELLIGENCE

COIMBATORE, TAMIL NADU, INDIA – 641112

Declaration

We declare that the report titled “**Mobile robot navigation: A study on various DRL and traditional path planning algorithms**” submitted by us is an original work done by Team-11 under the guidance of **Dr. Palmani Duraisamy, Associate Professor, School of Artificial Intelligence, Amrita Vishwa Vidyapeetham, Coimbatore** during the seventh semester of the academic year 2022-23, in the **School of Artificial Intelligence**. The work is original and wherever we have used materials from other sources, we have given due credit and cited them in the text of the report. This report has not formed the basis for the award of any degree, diploma, associate-ship, fellowship or other similar title to any candidate of any University.

Team-5

1. **Gorantla V N S L Vishnu Vardhan[CB.EN.U4AIE20019]**
2. **Menta Sai Aashish [CB.EN. U4AIE20039]**
3. **Penaka Vishnu Reddy [CB.EN. U4AIE20048]**
4. **Chandu Janakiram [CB.EN. U4AIE20009]**

Abstract

The realm of mobile robotics has traditionally centered around the development of navigation strategies, typically encompassing localization, map construction, and path planning. However, these conventional methodologies assume an ideal scenario of complete and accurate knowledge of environmental dynamics, a condition rarely met in real-world applications. The dynamic and stochastic nature of practical environments often undermines the efficacy of explicit mapping techniques, necessitating a reevaluation of navigation strategies.

In response to the challenges posed by dynamic and unpredictable environments, a paradigm shift has emerged in the field of mobile robotics. This shift emphasizes the significance of map-less navigation strategies that harness the learning capabilities of robots. The ability to learn and adapt becomes crucial for flexible path planning and effective obstacle avoidance. One particularly promising avenue in this evolving landscape is the application of Reinforcement Learning (RL) techniques, enabling autonomous mobile robots to dynamically adapt and navigate in response to their surroundings.

This project delves into the comparative study of mobile robot navigation by juxtaposing traditional path planning algorithms, exemplified by Dijkstra's and A*, with contemporary Deep Reinforcement Learning (DRL) algorithms, including Q-learning, DQN, TD3, and PPO. Traditional methods, grounded in explicit map construction, such as Dijkstra's and A*, pursue deterministic paths based on a known environment. While these algorithms excel in static settings, their limitations become apparent in dynamic and unpredictable environments where real-time adaptability is crucial. The rigid adherence to predefined maps hinders their ability to respond effectively to unforeseen obstacles and changes in the environment.

In contrast, DRL algorithms represent a transformative shift by allowing robots to learn and adapt autonomously without explicit map information. Q-learning, DQN, TD3, and PPO utilize neural networks for learning from interactions with the environment. This adaptive learning process enables robots to navigate effectively in complex and dynamic environments, making them well-suited for scenarios where explicit maps may be absent or incomplete. However, challenges such as computational complexity, training time, and the impact of training data quality need to be considered. By meticulously exploring both traditional and DRL algorithms, this project aims to offer insights into their respective strengths and limitations. The findings contribute to the ongoing discourse on developing more robust and adaptive robotic navigation systems, synthesizing the advantages of both traditional and modern methodologies.

Signature of the Guide

Name : Dr. Palmani Duraisamy

Table of Contents

Title	Page No.
Bonafide Certificate	ii
Declaration	iii
Acknowledgements	iv
Abstract	v
List of Figures	vii
Abbreviations	viii
1. Introduction	1
1.1. Robot Operating System (ROS)	2
1.2. Rviz	3
1.3. Gazebo	4
1.4. OpenAI Gym	5
1.5. CONCEPTS REQUIRED FOR MOVEMENT OF ROBOT	6
1.5.1 Frames	6
1.5.2 Transformation	7
1.5.3 Rotation matrices	8
1.6. Reinforcement learning	10
1.6.1 RL Algorithm	11
1.6.2 Elements of RL problem	11
1.6.3 Markov Decision Process	12
1.6.4 Model-based Reinforcement Learning	13
1.6.5 Model-free Reinforcement Learning	14
2. Objectives	15
3. Methodology	16
3.1. Navigation	16
3.2. BUILDING A MAP	16
3.3. Mapping procedure	17
3.4. Navigation stack	18
3.5. Mountain Car Environment	21
3.5.1 Q-learning	
3.5.2 DQN	

3.5.3 PPO	
3.5.4 TD3	
3.6. Simulation	30
4. Results and Discussion	36
5. Conclusions	41
6. References	43

List of Figures

Figure No.	Title	Page No.
1	An example of RQT Graph	2
2	Command to trigger Rviz simulation	3
3	Rviz interface	4
4	Gazebo Simulator	5
5	Gym Environment	5
6	Pose estimation of robot	6
7	Odometry frame	7
8	Transformation	7
9	View frames	9
10	tf_monitor	9
11	Tf_echo	10
12	RL Paradigm	11
13	Grid Mapping	17
14	Navigation Algorithms	19
15	DWA PLANNER	20
16	TURTLEBOT3	20
17	Mountain car	21
	Images of robot while simulating and result graphs of rewards	
18-27		
27-33		

Abbreviations

ROS – Robot Operating System

Rviz - Robot Visualization tool

SLAM – Simultaneous Localization and Mapping

CV - Computer Vision

CHAPTER 1

INTRODUCTION

The evolution of mobile robots within the realm of modern technology and automation has been marked by their increasing significance as versatile, autonomous entities adept at navigating unstructured environments. This report offers an in-depth exploration of a meticulously designed mobile robot system, emphasizing its core components, notably featuring the TurtleBot3 platform. Developed in collaboration with ROBOTIS and the Open Source Robotics Foundation, TurtleBot3 emerges as a cost-effective, open-source personal robot kit with a modular design, providing remarkable structural expansion capabilities through DYNAMIXEL technology.

In the context of this project, the TurtleBot3 Waffle model takes center stage. This specific iteration showcases advanced features, including the DYNAMIXEL XM430-W210-T servo, Intel Joule board, Intel RealSense, and laser distance sensors. Such attributes position the Waffle model as an ideal platform for facilitating autonomous navigation, 3D perception, and the creation of innovative applications. The report intricately examines the technical aspects of TurtleBot3, emphasizing its role as a catalyst for autonomy and innovation.

The integration of TurtleBot3 with the Robot Operating System (ROS) for navigation forms a pivotal aspect of the project. ROS, coupled with the capabilities of TurtleBot3, enables the mobile robot to operate autonomously, making decisions based on sensor data and environmental factors. This autonomous functionality proves invaluable, particularly in scenarios where human intervention is impractical or unsafe. The project's emphasis on ROS navigation aligns with the broader industry trend, acknowledging the importance of open-source software in advancing robotics capabilities.

Adding another layer of innovation, the project extends its exploration into Deep Reinforcement Learning (DRL) within a simulated Mountain Car environment. This dimension allows for the evaluation and comparison of traditional and cutting-edge algorithms, including Q-learning, DQN, TD3, and PPO. The fusion of ROS navigation and DRL capabilities showcases the project's commitment to comprehensive research, delving into the synergies between traditional autonomous navigation and the evolving landscape of reinforcement learning methodologies.

Beyond the technical intricacies, the report unfolds the transformative potential of the mobile robot system, particularly in diverse industries. From manufacturing and logistics to healthcare, the autonomous capabilities of TurtleBot3 offer a paradigm shift, enhancing efficiency, productivity, and safety. The interplay between traditional and modern methodologies, as exemplified by the integration of ROS navigation and DRL, underscores the project's contribution to the ongoing discourse on adaptive and robust robotic systems. As the report concludes, it leaves a compelling narrative of the TurtleBot3's role as a pioneering force in

reshaping the landscape of autonomous robotics.

1.1 Robot Operating System (ROS)

ROS (Robot Operating System) is an open-source software, categorized as a “robot framework”, that resembles a meta-operating system for our robot. It provides various tools, libraries, and packages for building, compiling, simulating, and running numerous codes across robots.

“rqt_graph” is one such tool provided by ROS. This tool displays the simulation of how all nodes (processes that are known as nodes) interact and depend on each other. This tool is very useful in debugging code and finding the point of miscommunication between various nodes. The graph represents all the nodes as a workflow chart with arrows pointed based on the message communication across various nodes.

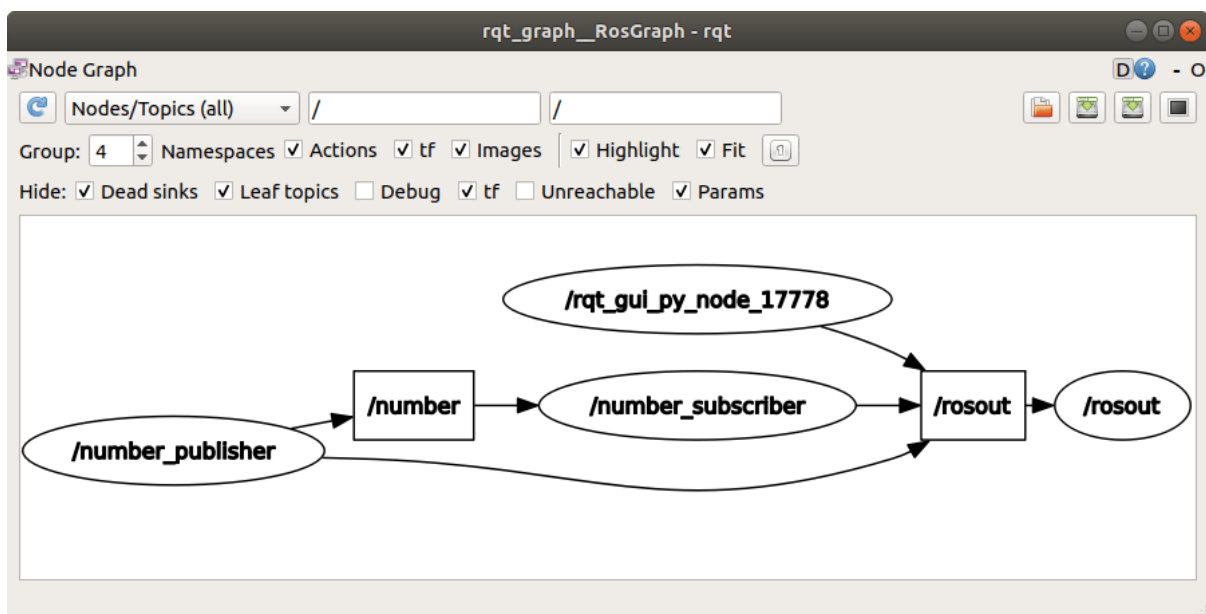


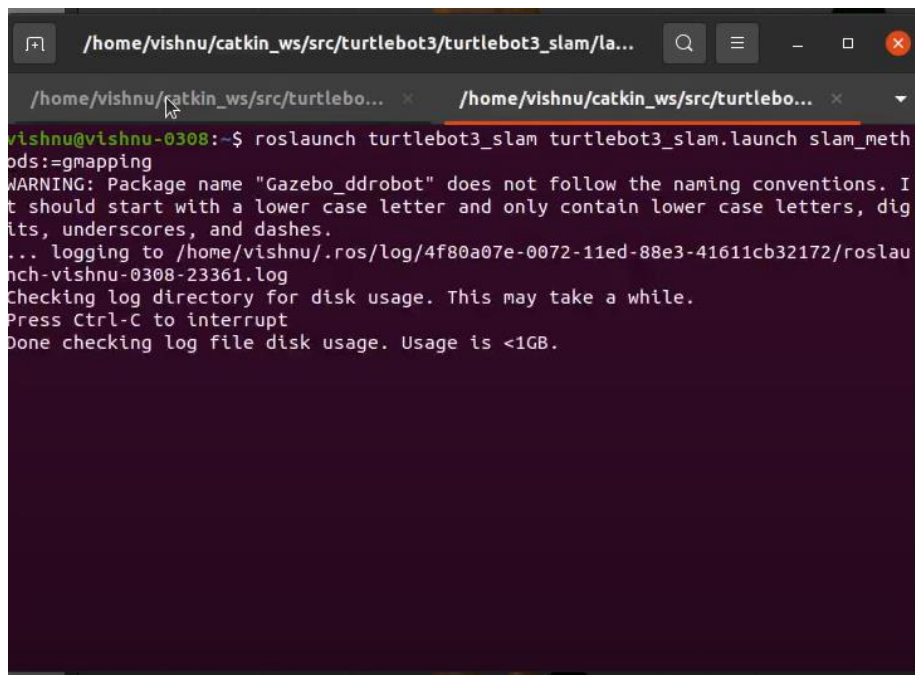
Fig. 1 An example of RQT Graph

Fig. 1 describes a sample RQT graph that visualizes how nodes publish and subscribe messages from other nodes. The */rosout* node at the end of the graph is a standard node for ROS representing the end of a ROS program.

1.2 Rviz

Rviz, or the Robot Visualization tool, is a crucial component in the arsenal of tools provided by the Robot Operating System (ROS) for visualizing robotic data. One of its primary functions is to offer a real-time, three-dimensional representation of the robot's sensor data and state. This includes visualizing sensor information such as camera images, laser scans, and point clouds, allowing developers and engineers to gain a comprehensive understanding of the robot's perception of its environment. Rviz plays a pivotal role in debugging and fine-tuning robotic systems during development, providing an interactive interface for visualizing the impact of sensor configurations and algorithms in real-time.

Moreover, Rviz is instrumental in the mapping process within ROS. It allows users to visualize the robot's path as it navigates through an environment, aiding in the identification of potential issues and fine-tuning navigation algorithms. The tool facilitates the creation of maps by visualizing the sensor data and the robot's trajectory, providing critical insights into the effectiveness of the mapping algorithms being employed. Rviz's visualization capabilities also extend to the creation of YAML files, which store configuration parameters for ROS nodes. This feature enhances the reproducibility of experiments and promotes efficient collaboration among developers by enabling them to share and recreate specific configurations seamlessly. Overall, Rviz proves to be an indispensable tool in the ROS ecosystem, streamlining the development, mapping, and configuration processes for robotic systems.

A terminal window with a dark background and light-colored text. The window title is partially visible as "/home/vishnu/catkin_ws/src/turtlebot3/turtlebot3_slam/la...". The terminal shows a user prompt "vishnu@vishnu-0308:~\$" followed by the command "ros launch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping". The output includes a warning about the package name "Gazebo_ddrobot" not following naming conventions, followed by logging information and a message about checking log directory disk usage, stating "Usage is <1GB.".

```
vishnu@vishnu-0308:~$ ros launch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
WARNING: Package name "Gazebo_ddrobot" does not follow the naming conventions. It should start with a lower case letter and only contain lower case letters, digits, underscores, and dashes.
... logging to /home/vishnu/.ros/log/4f80a07e-0072-11ed-88e3-41611cb32172/roslaunch-vishnu-0308-23361.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
```

Fig. 2 Command to trigger Rviz simulation

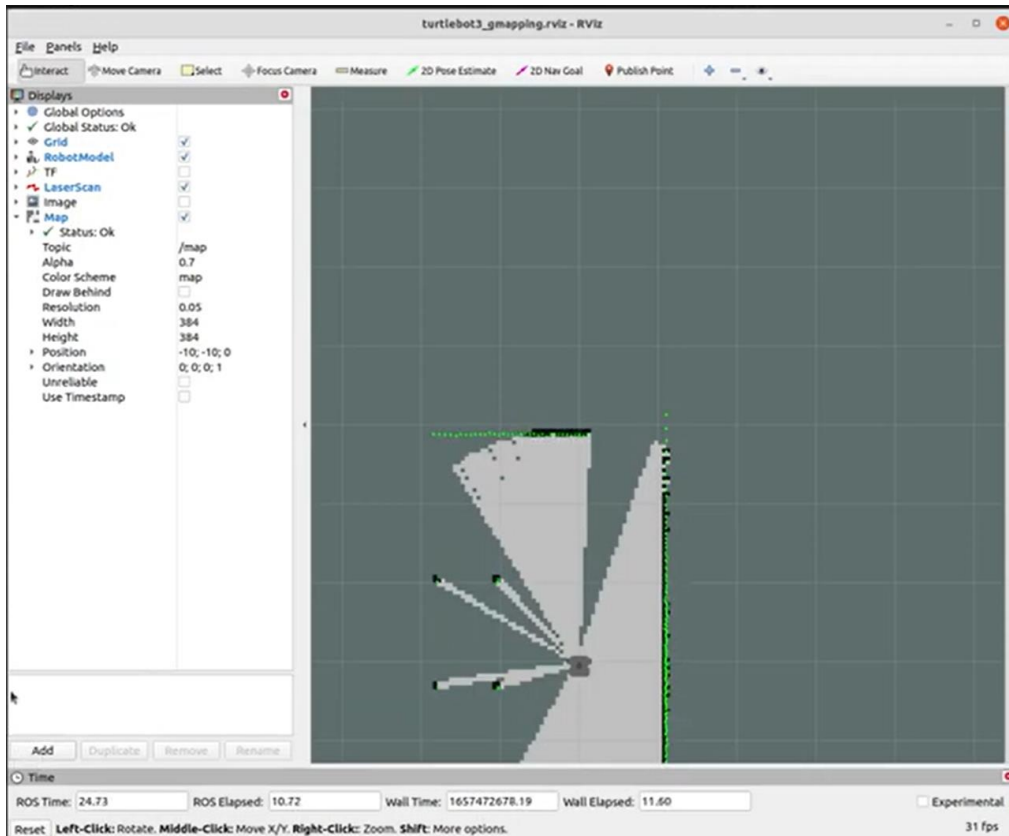


Fig. 3 Rviz interface

1.3 Gazebo

Gazebo stands as a robust simulation software within the Robot Operating System (ROS) ecosystem, providing an indispensable tool for the realistic simulation of robots. The software is equipped with a comprehensive package and a thriving community, offering a plethora of public models and worlds, as well as plugins that mirror real-life components. This extensive repository of resources facilitates developers in creating highly realistic simulations that closely resemble the physical environments in which robots will operate. Gazebo's commitment to accuracy is exemplified by its ability to render nearly any model or world with high fidelity, presenting a realistic and dynamic simulation environment.

One of Gazebo's notable strengths lies in its provision of a diverse set of sensor and noise models, enabling developers to simulate sensor streams with high fidelity. This feature is essential for testing and validating the performance of robotic systems under various conditions. Gazebo leverages the inertial values of robots to simulate external factors that may influence a robot's behaviour, enhancing the realism of the simulation. The simulator's user interface, as depicted in Fig. 3, offers a visual representation of the simulated environment, complete with three axes, a checked-format default plane, and a default light source. Additionally, Gazebo provides critical information such as frame rate, simulation time, real-time execution, and the real-time factor, offering developers valuable insights into the performance of their robotic systems within the simulated environment.

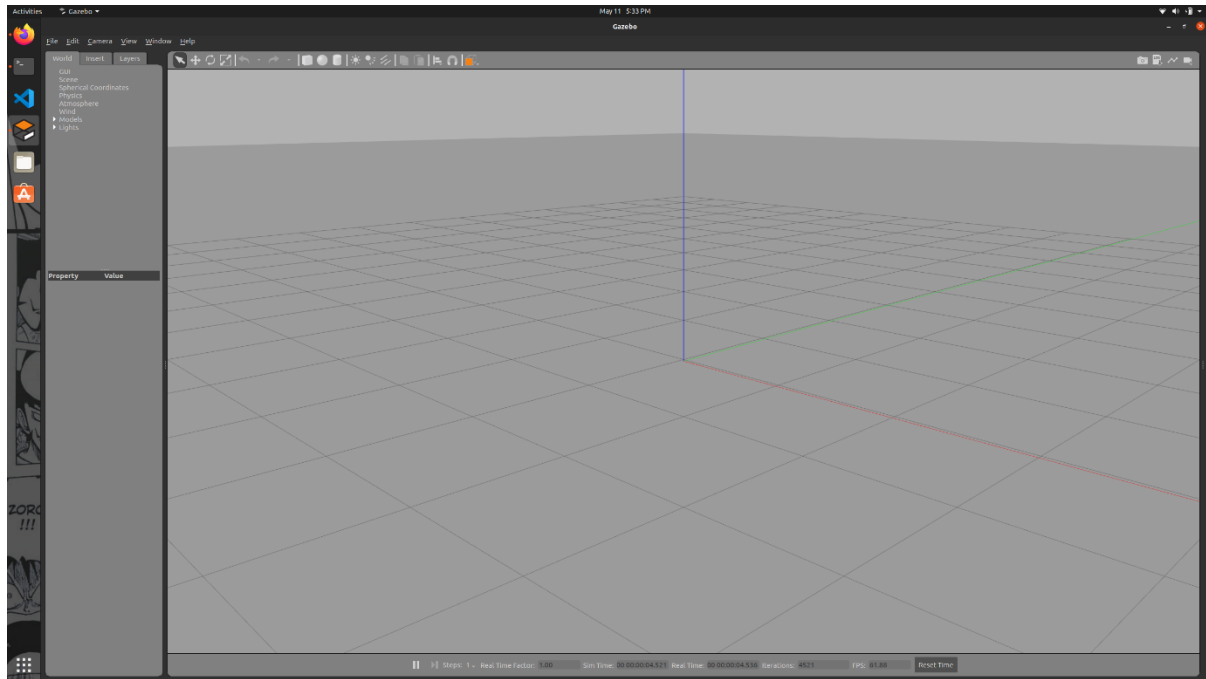


Fig.4 Gazebo Simulator

1.4 OpenAI Gym

OpenAI Gym serves as a powerful toolkit for developing and comparing reinforcement learning algorithms, providing a versatile environment for the simulation and evaluation of various robotic tasks. With an extensive collection of pre-built environments and tasks, OpenAI Gym supports a broad range of learning scenarios, making it a valuable resource for researchers and developers alike. The toolkit's flexibility allows for the creation of custom environments to suit specific project requirements, and it comes equipped with well-defined physics and a dynamic community contributing diverse environments. OpenAI Gym's user-friendly interface and accessible API empower developers to seamlessly integrate their reinforcement learning algorithms, offering a standardized platform for evaluating and benchmarking the performance of robotic systems in simulated environments.

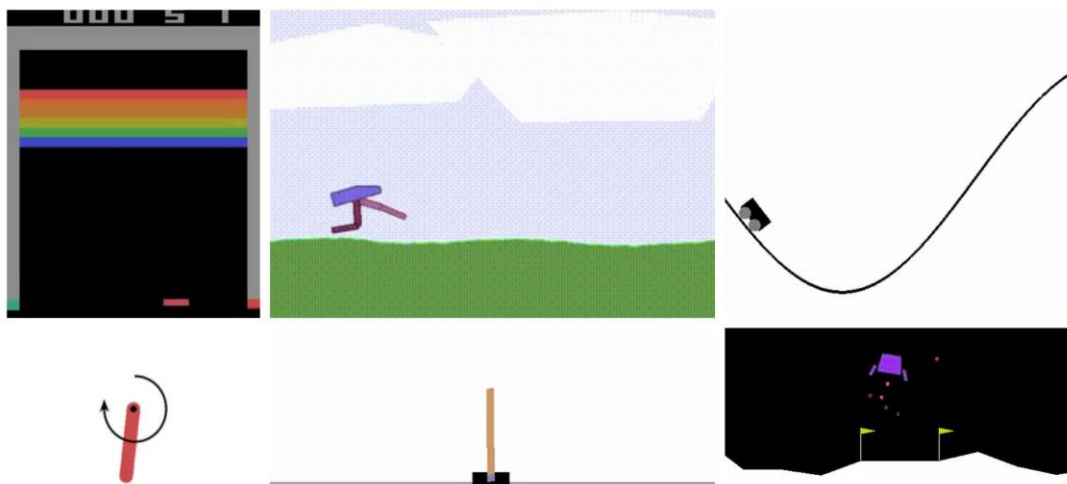


Fig. 5 Gym Environment

1.5 CONCEPTS REQUIRED FOR MOVEMENT OF ROBOT

1.5.1 FRAMES

Frames in a robot define a coordinate system that the robot uses to know where it is and where to go. A frame is comprised of six main components: an X, Y, & Z axis and a rotation about each of these axes. There are generally three types of frames used on a robot: base (world) frame, user frame, and tool frame

Map frame has its origin at some arbitrarily chosen point in the world. This coordinate frame is fixed in the world. Now we will see information about these map frame by typing the command “rostopic echo amcl_pose” in terminal.

“**Amcl_pose**” is responsible topic for finding poses of map and other information related to map frame



```
vishnu@vishnu-0308:~$ rostopic echo amcl_pose
header:
  seq: 33
  stamp:
    secs: 314
    nsecs: 387000000
  frame_id: "map"
pose:
  pose:
    position:
      x: 3.788857581415532
      y: 1.0188280395581324
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: -0.06812447539075775
      w: 0.9976768293654684
  covariance: [0.00379468214151224, 0.0009290744650187754, 0.0, 0.0, 0.0, 0.0, 0.0009290744650183314,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0007911645910745471]
```

Fig.6 Pose estimation of robot

Odom frame has its origin at the point where the robot is initialized. This coordinate frame is fixed in the world. Now we will see information about these odom frame by typing the command “rostopic echo odom” in terminal.

“**odom**” is responsible topic for finding poses of map and other information related to odom Frame

There are many other frames which are related robots these 2 frames map and odom frames are the main frames.

1.5.3 Rotation matrices

GENERAL ROTATION MATRIX IN 3D

$$R_X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$R_Y = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

$$R_Z = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R = R_z(\alpha)R_y(\beta)R_x(\gamma)$$

GENERAL TRANSFORMATION MATRIX IN 3D

$$\begin{bmatrix} {}^{W^1}x \\ {}^{W^1}y \\ {}^{W^1}z \\ 1 \end{bmatrix} = \begin{bmatrix} {}^{W^1}R_{W^2} & t \\ 0_{1 \times 3} & 1 \end{bmatrix} * \begin{bmatrix} {}^{W^2}x \\ {}^{W^2}y \\ {}^{W^2}z \\ 1 \end{bmatrix}$$

TRANSFORMATION PACKAGE

TF stands for transformation library in ROS

It performs computation for transformations between frames.

It allows to find the pose of any object in any frame using transformations

A robot is a collection of frames attached to its different joints

TF Package Nodes

The TF Package has several ROS nodes that provide utilities to manipulate frames and transformations in ROS

view_frames: visualizes the full tree of coordinate transforms.


```

^Cvishnu@vishnu-0308:~$ roslaunch tf view_frames
Listening to /tf for 5.0 seconds
Done Listening
^Cvishnu@vishnu-0308:~$ dot - graphviz version 2.43.0 (0)\n'
Detected dot version 2.43
frames.pdf generated
vishnu@vishnu-0308:~$ evince frames.pdf

```

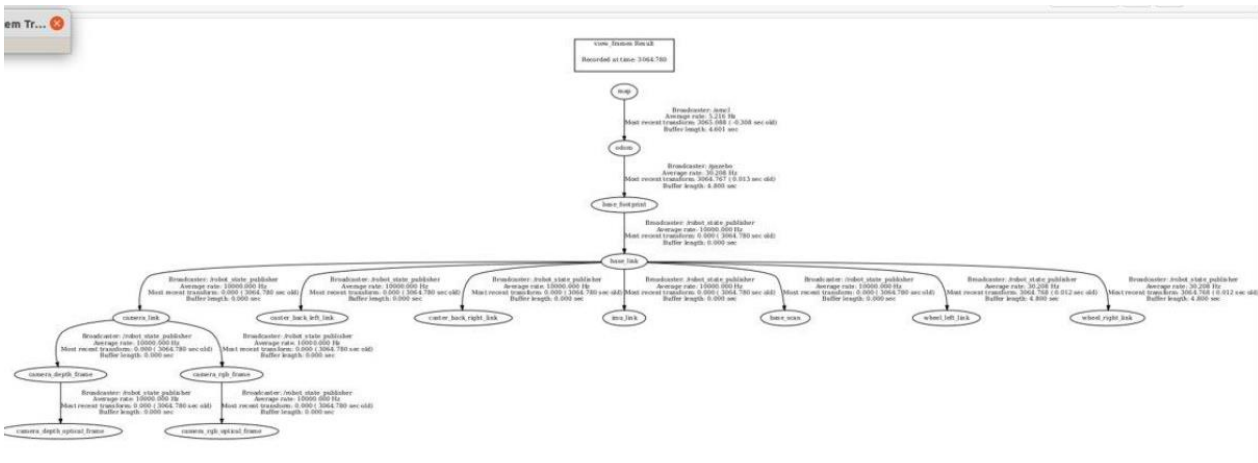


Fig. 9 View frames

tf_monitor: monitors transforms between frames.

```

^Cvishnu@vishnu-0308:~$ roslaunch tf tf_monitor

RESULTS: for all Frames

Frames:
Frame: base_footprint published by unknown_publisher Average Delay: 0.001 Max Delay: 0.001
Frame: base_link published by unknown_publisher(static) Average Delay: 0 Max Delay: 0
Frame: base_scan published by unknown_publisher(static) Average Delay: 0 Max Delay: 0
Frame: camera_depth_frame published by unknown_publisher(static) Average Delay: 0 Max Delay: 0
Frame: camera_depth_optical_frame published by unknown_publisher(static) Average Delay: 0 Max Delay: 0
Frame: camera_link published by unknown_publisher(static) Average Delay: 0 Max Delay: 0
Frame: camera_rgb_frame published by unknown_publisher(static) Average Delay: 0 Max Delay: 0
Frame: camera_rgb_optical_frame published by unknown_publisher(static) Average Delay: 0 Max Delay: 0
Frame: caster_back_left_link published by unknown_publisher(static) Average Delay: 0 Max Delay: 0
Frame: caster_back_right_link published by unknown_publisher(static) Average Delay: 0 Max Delay: 0
Frame: imu_link published by unknown_publisher(static) Average Delay: 0 Max Delay: 0
Frame: wheel_left_link published by unknown_publisher Average Delay: 0.00425 Max Delay: 0.005
Frame: wheel_right_link published by unknown_publisher Average Delay: 0.00425 Max Delay: 0.005

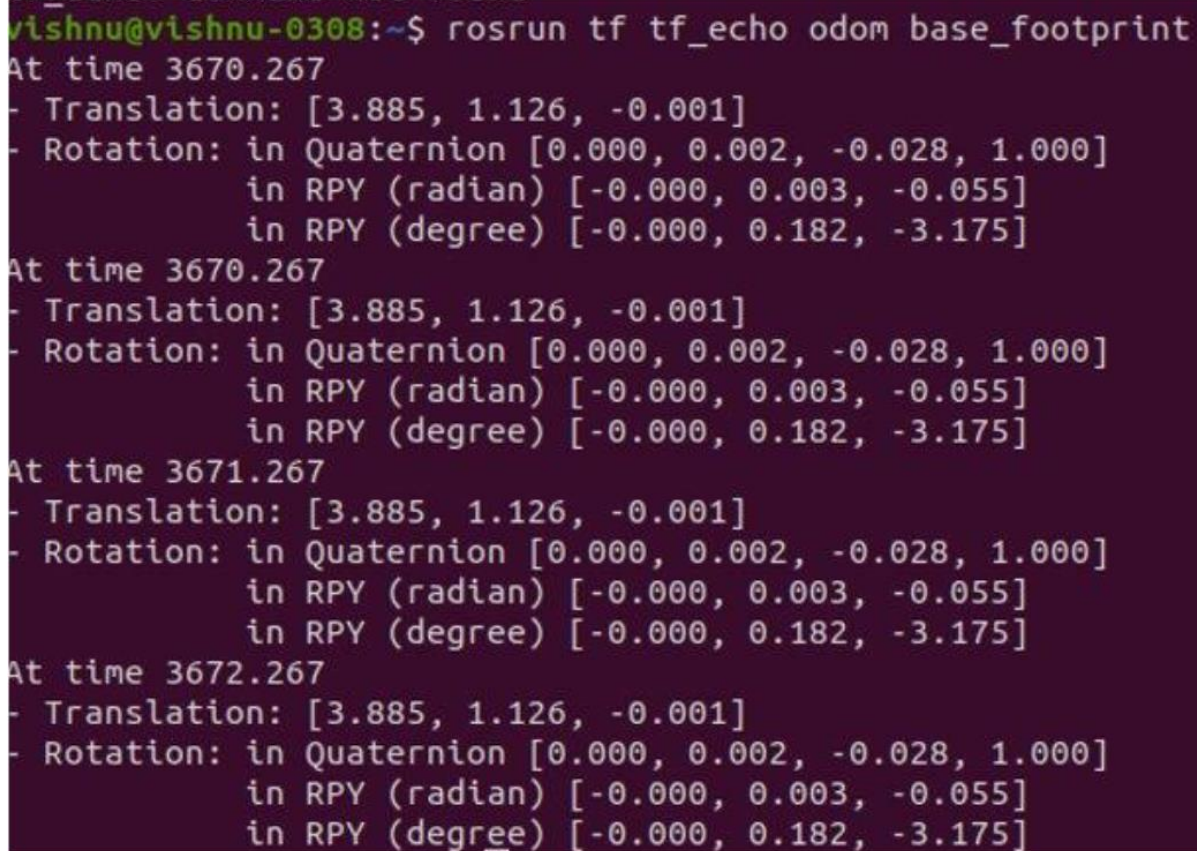
All Broadcasters:
Node: unknown_publisher 77.6699 Hz, Average Delay: 0.002625 Max Delay: 0.005
Node: unknown_publisher(static) 1e+08 Hz, Average Delay: 0 Max Delay: 0

RESULTS: for all Frames

```

Fig. 10 tf_monitor

tf_echo: prints specified transform to screen



```
vishnu@vishnu-0308:~$ rosrn tf tf_echo odom base_footprint
At time 3670.267
- Translation: [3.885, 1.126, -0.001]
- Rotation: in Quaternion [0.000, 0.002, -0.028, 1.000]
           in RPY (radian) [-0.000, 0.003, -0.055]
           in RPY (degree) [-0.000, 0.182, -3.175]
At time 3670.267
- Translation: [3.885, 1.126, -0.001]
- Rotation: in Quaternion [0.000, 0.002, -0.028, 1.000]
           in RPY (radian) [-0.000, 0.003, -0.055]
           in RPY (degree) [-0.000, 0.182, -3.175]
At time 3671.267
- Translation: [3.885, 1.126, -0.001]
- Rotation: in Quaternion [0.000, 0.002, -0.028, 1.000]
           in RPY (radian) [-0.000, 0.003, -0.055]
           in RPY (degree) [-0.000, 0.182, -3.175]
At time 3672.267
- Translation: [3.885, 1.126, -0.001]
- Rotation: in Quaternion [0.000, 0.002, -0.028, 1.000]
           in RPY (radian) [-0.000, 0.003, -0.055]
           in RPY (degree) [-0.000, 0.182, -3.175]
```

Fig. 11 tf_echo

We are finding relation between two frames at different time

roswtf: with the tfwtf plugin, helps you track down problems with tf.

static_transform_publisher is a command line tool for sending static transforms

The above picture shows all topics and nodes .

1.6 Reinforcement Learning

Reinforcement learning revolves around the dynamic process of discerning optimal actions by mapping situational contexts to responses, all with the overarching goal of maximizing a numerical reward signal. In this paradigm, the learner operates in an environment where explicit instructions about which actions to take are absent. Instead, the learner must engage in a continuous process of trial and error, exploring various actions to ascertain which ones lead to the most favorable outcomes. Particularly noteworthy in reinforcement learning are two key characteristics: trial-and-error search and delayed reward. These features encapsulate the essence of learning in scenarios where actions not only impact immediate rewards but also subsequent situations and, consequently, all future rewards.

A distinctive challenge in reinforcement learning, setting it apart from other learning paradigms, is the inherent trade-off between exploration and exploitation. The crux of the matter lies in the agent's need to strike a delicate balance—preferring actions that have proven effective in the past to accrue significant rewards while also delving into unexplored actions to

discover potentially more beneficial strategies. The dilemma becomes apparent; exclusive pursuit of either exploration or exploitation leads to suboptimal task performance. Successful navigation through this challenge necessitates the agent's willingness to try a diverse array of actions, progressively favoring those that demonstrate superior efficacy in achieving optimal outcomes. The essence of reinforcement learning lies in this delicate dance between exploring new possibilities and exploiting known effective actions to navigate the intricate landscape of decision-making.

1.6.1 RL Algorithm

1. First, the agent interacts with the environment by performing an action
2. The agent performs an action and moves from one state to another
3. And then the agent will receive a reward based on the action it performed
4. Based on the reward, the agent will understand whether the action was good or bad
5. If the action was good, that is, if the agent received a positive reward, then the agent will prefer performing that action or else the agent will try performing another action which results in a positive reward. So it is basically a trial and error learning process .

1.6.2 Elements of RL problem

- **Agent:** It is the program or the mind of our robot which is able to intelligently plan and take actions and receive rewards.
- **Model:** in model-based Algorithms, the model is the representation of the environment from the point of view of the agent, in our case the environment is a set of states (positions) representing the obstacle-map in which our agent (robot) interacts with, while navigating to reach a specific target.

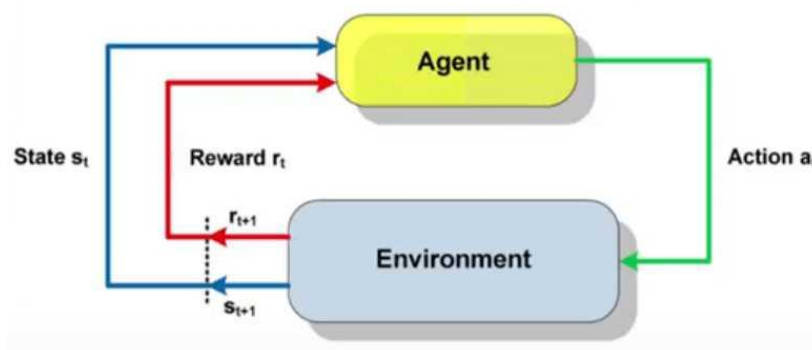


Figure 12 : RL Paradigm

- **Policy:** The policy serves as the guiding framework for the agent's behavior within an environment, outlining how the agent makes decisions and selects actions. In the context of our SSMR robot, the policy corresponds to the planning algorithm determining the directional motion, effectively dictating the course of action the robot takes.

- **Reward Signal:** Acting as a crucial feedback mechanism, the reward signal is a scalar value bestowed upon the agent following its interaction with the environment. This numerical value encapsulates the efficacy of the agent's action, conveying whether the undertaken action was favorable or unfavorable. The overarching objective for the agent is to maximize the cumulative reward over the long term, compelling it to make strategic decisions based on the received feedback. Various reward shaping systems contribute to assigning these values, influencing the learning process.
- **Value Function:** The value function provides an evaluative measure of how advantageous it is for an agent to be in a specific state. Denoted as $v(s)$, the value function is intricately tied to the policy, representing the total expected reward the agent anticipates receiving, starting from the initial state and extending throughout the task.
- **State-Action Value Function:** Also known as the Q function, the state-action value function quantifies the desirability of an agent executing a specific action within a given state under a defined policy. Represented as $Q(s)$, this function illuminates the value associated with taking a particular action in a state, guiding the agent in making informed decisions based on the underlying policy.

1.6.3 Markov Decision Process

The Markov Decision Process (MDP) provides a mathematical framework for solving the reinforcement learning (RL) problem. Almost all RL problems can be modeled as MDP. MDP is widely used for solving various optimization problems. The Markov property states that the future depends only on the present and not on the past. The Markov chain is a probabilistic model that solely depends on the current state to predict the next state and not the previous states, that is, the future is conditionally independent of the past. The Markov chain strictly follows the Markov property. In a finite MDP, the sets of states, actions, and rewards (S , A , and R) all have a finite number of elements. In this case, the random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action. That is, for particular values of these random variables, $s_0 \in S$ and $r \in R$, there is a probability of those values occurring at time t , given particular values of the preceding state and action:

$$p(s', r | s, a) = Pr(S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a)$$

In a Markov decision process, the probabilities given by p completely characterize the environment's dynamics. That is, the probability of each possible value for S_t and R_t depends only on the immediately preceding state and action, S_{t-1} , A_{t-1} , and given them, not at all on earlier states and actions. This is best viewed a restriction not on the decision process, but on the state. The state must include information about all aspects of the past agent–environment interaction that make a difference for the future. If it does, then the state is said to have the Markov property.

The agent is devoted to maximize the sum of the rewards taken in the long run from the environment, so we denote the expected cumulative sum of reward by G_t (the return value)

that is represented by the following equation:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_t$$

Tasks in MDPs can be divided into two categories, episodic and non-episodic tasks. Episodic tasks are the tasks which terminate by reaching a final state and the agent is reset to start all over again, but in non-episodic task the environment lacks the terminal state and the task is continuous. However, in both cases a reward discounting factor is needed to discount the future rewards based on the policy followed by the agent which states the importance of future rewards in respect to immediate ones so the equation 1.3 is updated by adding the discounting factor γ .

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \doteq \sum_{k=0}^{k=\infty} \gamma^k R_{t+1+k}$$

1.6.4 Model-based Reinforcement Learning

By a model of the environment, we mean anything that an agent uses to predict how the environment will respond to its actions. Given a state and an action, a model produces a prediction of the resultant next state and next reward as shown in figure 2. If the model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring. Some models produce a description of all possibilities and their probabilities; these we call distribution models. Other models produce just one of the possibilities, sampled according to the probabilities; these we call sample models. Model based reinforcement learning extends the traditional Q-learning by an online model and solve the problem using Dynamic Programming techniques. We usually assume that the environment is a finite MDP. That is, we assume that its state, action, and reward sets, S , A , and R , are finite, and that its dynamics are given by a set of probabilities $p(s_0, r | s, a)$, for all $s \in S$, $a \in A(s)$, $r \in R$, and $s_0 \in S^+$ (S^+ is S plus a terminal state if the problem is episodic). Although DP ideas can be applied to problems with continuous state and action spaces, exact solutions are possible only in special cases. A common way of obtaining approximate solutions for tasks with continuous states and actions is to quantize the state and action spaces and then apply finite-state DP methods can use.

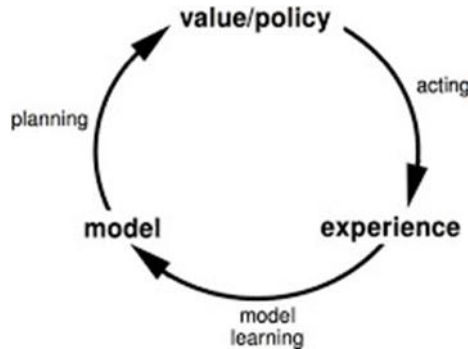


Figure 1.2: Model-Based RL Paradigm

1.6.5 Model-free Reinforcement Learning

Many reinforcement learning algorithms consider the environment as unknown, which makes the interaction between the agent and the environment based on trial and error. These algorithms are called model-free which assumes no information about the future after taking some actions. Model-free algorithms prove that this is an effective way to interact with the environment for learning a specific task. These algorithms depend only on the reward value at the end of each episode for the policy improvement by maximizing the expected total reward.

CHAPTER 2

2.1 OBJECTIVES

Project Objective: The primary objective of this project, titled "Mobile Robot Navigation: A Study on Various DRL and Traditional Path Planning Algorithms," is twofold. First, the project aims to explore and compare different navigation strategies by mapping a house environment using Gmap and SLAM techniques. The navigation within this environment is facilitated through the integration of ROS Gazebo, employing traditional path planning algorithms such as Djikstra's and A*.

Second, the project extends its scope to the realm of reinforcement learning by implementing Q-learning, DQN, TD3, and PPO in a mountain car gym environment. The goal is to assess the performance and efficacy of both traditional and Deep Reinforcement Learning (DRL) algorithms in diverse settings.

Key Project Tasks:

1. Mapping and Navigation in House Environment:
.
2. Reinforcement Learning in Mountain Car Gym Environment:

CHAPTER 3

METHODOLOGY

PART-1

3.1 NAVIGATION

There are two types of navigation used in ROS

1)Map-based Navigation

2)Reactive Navigation

In first part we performed map-based navigation

3.1.1 Map based navigation

Localization: it helps the robot to know where he is

Mapping: the robot needs to have a map of its environment to be able to recognize where he has been moving around so far

Motion planning or path planning: to plan a path, the target position must be well-defined to the robot, which require an appropriate addressing scheme that the robot can understand

3.1.2 Navigation packages

Three main packages of the navigation stack

***move_base*:** makes the robot navigate in a map and move to move to a goal pose with respect to a given reference frame

***mapping*:** creates maps using laser scan data

***amcl*:** responsible for localization using an existing map.

3.2 BUILDING A MAP: SIMULTANEOUS LOCALIZATION AND MAPPING (SLAM)

It is the process of building a map using range sensors (e.g. laser sensors, 3D sensors, ultrasonic sensors) while the robot is moving around and exploring an unknown area

***Sensor Fusion*:** This process uses filtering techniques like Kalman filters or prarticle filters

3.2.1 SLAM APPROACHES

There are several SLAM approaches in ROS

gmapping which contains a ROS wrapper for OpenSlam's Gmapping

cartographer, which is a system developed by Google that provides real-time simultaneous ***localization and mapping (SLAM)*** in 2D and 3D across multiple platforms and sensor configurations.

hector_slam which is an other SLAM approach that can be used without odometry.

3.3 MAPPING PROCEDURE

3.3.1 OCCUPANCY IN GRID MAP

- A map is a grid (matrix) of cell
- A cell can be empty or occupied
- Depending on resolution, cell size can be 5 to 50 cm
- Each cell hold a probability of occupancy 0% to 100%
- Areas that are unknown are marked as -1

3.3.2 MARKING AND CLEARING

The map is built using a SLAM algorithm.

Cell has three possible states:

- Unknown
- Empty
- Occupied

It is based on the process of marking and clearing

Marking: a cell is marked as obstacle

Clearing: a cell is marked as empty

ray-tracing: used to find empty cell.

In the above diagram we can see that black box represents there is an obstacle. The white colour represents there is no obstacle and the grey colour represents the path is not covered.

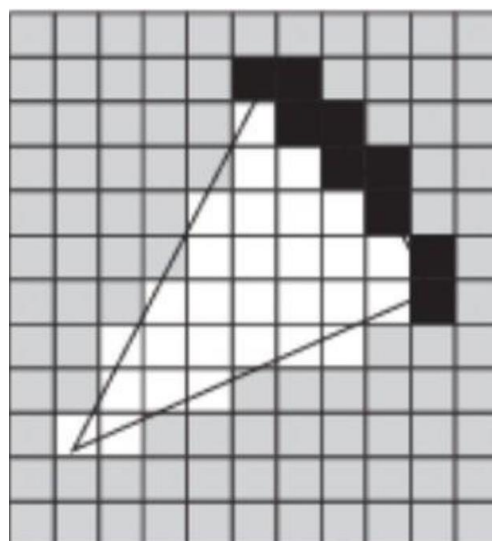


Fig.13 Grid mapping

In the above diagram we can see that black box represents there is an obstacle. The white colour represents there is no obstacle and the grey colour represents the path is not covered.

3.3.3 USAGE OF ACTION SERVICES

- For every movement of the robot the feedback will be published
- Goal will be our target location.
- Result will be either it is completed or not

3.4 NAVIGATION STACK

The navigation stack has two motion planners:

Global Path Planner: plans a static obstacle-free path from the location of the robot to the goal location

Local Path Planner: execute the planned trajectory and avoids dynamic obstacle.

3.4.1 GLOBAL PATH PLANNER

The global path planner is responsible for finding a global obstacle-free path from initial location to the goal location using the environment map

Global path planner must adhere to the `nav_core::BaseGlobalPlanner` interface.

3.4.2 BUILT-IN GLOBAL PATH PLANNER IN ROS

There are three built-in global path planners in ROS:

carrot planner: simple global planner that takes a user specified goal point and attempts to move the robot as close to it as possible, even when that goal point is in an obstacle.

navfn: uses the Dijkstra's algorithm to find the global path between any two locations.

global_planner: is a replacement of navfn and is more flexible and has more options.

In our project we are using **navfn** package for global path planner

3.4.2.1 NAVFN

- navfn uses Dijkstra's algorithm to find a global path
- global_planner is a flexible replacement of navfn
- Support of A*
- Can use a grid path

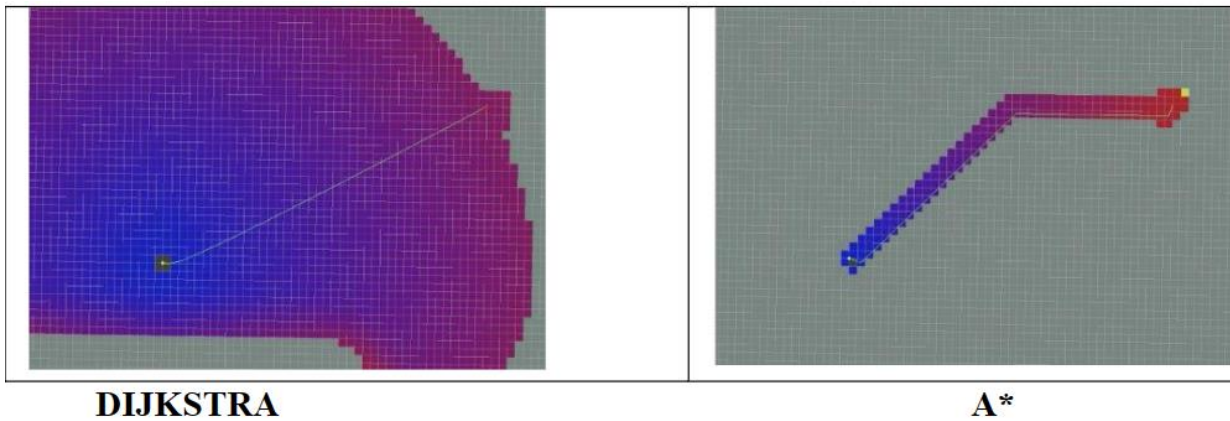


Fig. 14 Navigation Algorithms

3.4.3 LOCAL PATH PLANNER

The local path planner is responsible for executing the static path determined by the global path planner while avoiding dynamic obstacles that might come into the path using the robot's sensors.

• Global path planner must adhere to the `nav_core::BaseLocalPlanner` interface.

The algorithm used for Local path planner is DWA Algorithm

3.4.3.1 DWA ALGORITHM

- Discretely sample in the robot's control space ($dx, dy, d\theta$)
- For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
- Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
- Pick the highest-scoring trajectory and send the associated velocity to the mobile base. Rinse and repeat.

3.4.3.2 DWA PARAMETERS

simulation time: time allowed for the robot to move with the sampled velocities

takes the velocity samples in robot's control space, and examines the circular trajectories represented by those velocity samples, and finally eliminates bad velocities

High simulation times (≥ 5) lead to heavier computation, but get longer paths

Low simulation times (≤ 2) limited performance, especially when the robot needs to pass a narrow doorway, or gap between furnitures, because there is insufficient time to obtain the optimal trajectory that actually goes through the narrow passway

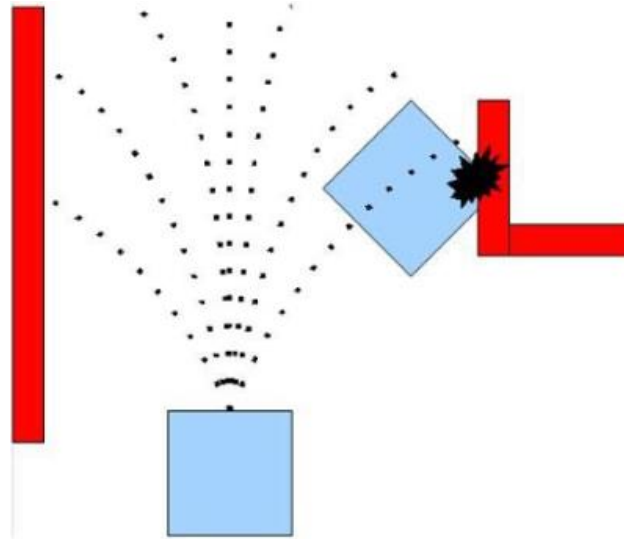


Fig. 15 DWA planner

3.4.4 TRAJECTORY SCORING

cost = path distance bias (distance(m) to path from the endpoint of the trajectory) *
+ goal distance-bias (distance(m) to local goal from the endpoint of the trajectory) *
+ ocddist-scale (maximum obstacle cost along the trajectory in obstacle cost (0-254))

3.4.5 THE TURTLEBOT IN EMPTY WORLD

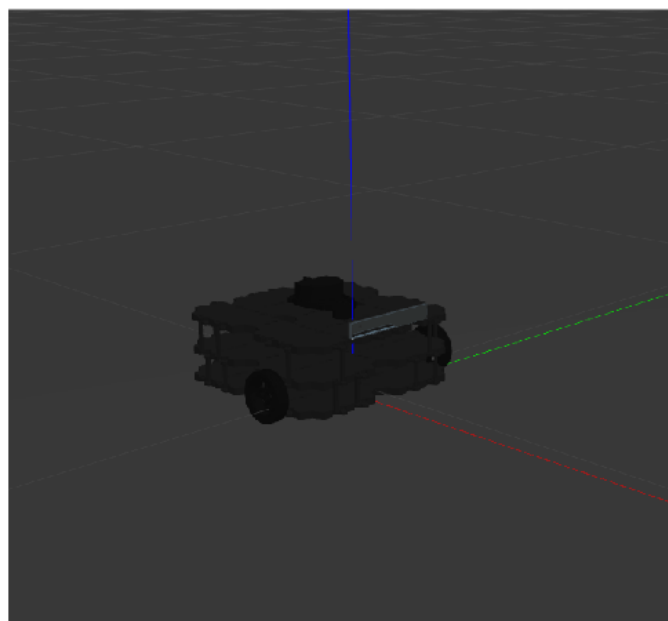


Fig.16 Turtlebot3

PART-II

3.5 Solving Mountain Car Problem with Q-learning , DQN, PPO & TD3

Description:

The Mountain Car MDP is a deterministic MDP that consists of a car placed stochastically at the bottom of a sinusoidal valley, with the only possible actions being the accelerations that can be applied to the car in either direction. The goal of the MDP is to strategically accelerate the car to reach the goal state on top of the right hill. There are two versions of the mountain car domain in gym: one with discrete actions and one with continuous. This version is the one with discrete actions.

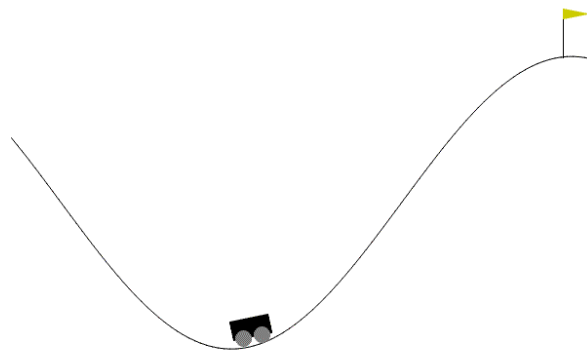


Fig. 17 Mountain Car

Action Space	Discrete(3)
Observation Shape	(2,)
Observation High	[0.6 0.07]
Observation Low	[-1.2 -0.07]
Gym Environment	MountainCar-v0

3.5.1 Q-Learning

Learning within the agent is inherently model-free, characterized by the absence of prior knowledge about expected next states or rewards when the agent interacts with the environment through actions. This sets the stage for a trial-and-error system, where the agent gradually constructs a tabular function mapping each state-action pair to the subsequent state and corresponding reward. Over numerous trials, the agent accumulates a database of state-action values, leading to the eventual construction of a policy based on the learned Q-values—commonly referred to as Q-learning.

The learned action-value function, Q , directly approximates the optimal action-value function, q^* , irrespective of the specific policy being followed. This simplifies the algorithm's analysis and facilitates early convergence proofs. Although the policy influences the state-action pairs visited and updated, Q-learning remains an off-policy algorithm, simultaneously learning about the greedy policy while employing a distinct behavior policy for interacting with the environment and collecting data. Typically, this behavior policy adheres to an ϵ -greedy strategy, selecting the greedy action with probability ϵ and a random action with probability $1-\epsilon$ to ensure comprehensive coverage of the state-action space.

$$Q^\pi(s, a) = E^\pi\left(\sum_{k=0}^{\infty} \gamma^k R_{t+1+k} \mid S_t = s, A_t = a\right)$$

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

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
Loop for each episode:
 Initialize S
 Loop for each step of episode:
 Choose A from S using policy derived from Q (e.g., ϵ -greedy)
 Take action A , observe R, S'
 $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$
 $S \leftarrow S'$
 until S is terminal

The initialization of the state space involves assigning arbitrary Q values, and subsequent approximation of Q values occurs through a sweep along the state-action space using the previously defined equation (3.3.1). An indispensable element in this process is the update rule, crucial for iteratively refining Q values to converge towards the optimal Q value, representing the most accurate evaluation of taking an action A concerning a state S . The introduction of a step-size parameter or learning rate α emphasizes the impact of the newly updated value on the Q value estimate. Additionally, a discount rate γ is fine-tuned to give weight to future rewards, where a value approaching unity prompts the agent to prioritize enhancing future rewards, while a value nearing zero causes the agent to disregard future consequences on current actions. Actions are selected in a greedy fashion, based on the maximum Q value, until an optimal policy is generated.

State and action spaces

The states are the position of the car in the horizontal axis on the range $[-1.2, 0.6]$

and its velocity on the range $[-0.07, 0.07]$. The goal is to get the car to accelerate up the hill and get to the flag. The possible actions are (left, neutral, right). Thus, we have the two-dimensional continuous state space [position x velocity]. and one-dimensional discrete action space with values (0,1,2).

Environment and reward threshold

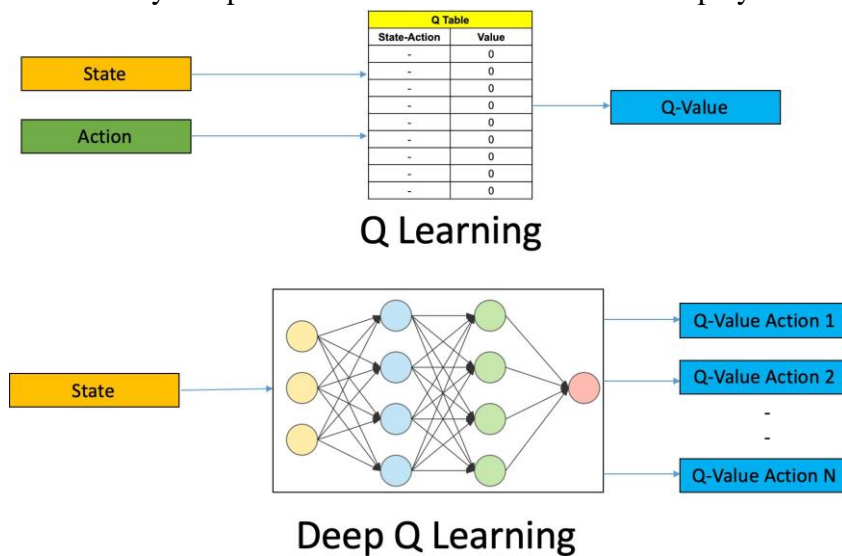
Solving the environment require an average total reward of over -110 on 100 consecutive episodes.

3.5.2 Deep Q learning

The DQN (Deep Q-Network) algorithm was created by **DeepMind** in 2015. By mixing reinforcement learning and deep neural networks at scale, it was able to tackle a wide range of Atari games (some to superhuman levels). The algorithm was created by combining deep neural networks and experience replay to enhance and evolve the Q-learning mechanism. Deep-Q-learning uses the neural networks as function approximators instead of going through a table of Q-values.

$$Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t^-).$$

These function approximators predict the Q-values based on the network weights then the predicted value is compared to the target value provided by the bellman equation and the loss function is calculated. Afterwards, the networks weights are updated using gradient descent and back propagation and the data fitting process takes place. To avoid high correlation, a replay memory is initialized with a preset length saving each transition taken by the agent and for each training, a batch of randomly sampled transitions are taken from the replay memory to train on.



Deep Q Networks

In the realm of Deep Q Networks (DQN), a neural network is employed to estimate Q-values for various actions based on the states received from a given environment. The network's primary goal is to approximate the optimal Q-function, as dictated by the Bellman equation. The loss in the network is computed by contrasting the outputted Q-values with the target Q-values derived from the Bellman equation's right-hand side. This loss is then minimized through Stochastic Gradient Descent and backpropagation. The iterative process continues for each state in the environment until the loss is sufficiently minimized, yielding an approximate optimal Q-function.

$$\underbrace{\text{New } Q(s, a)}_{\text{New Q-Value}} = Q(s, a) + \underbrace{\alpha}_{\text{Learning Rate}} \left[\underbrace{R(s, a)}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount rate}} \underbrace{\max_{a'} Q'(s', a')}_{\text{Maximum predicted reward, given new state and all possible actions}} - Q(s, a) \right]$$

The Network Layers

The network consists of several layers, with the initial layer being the input layer. Nodes in this layer represent individual features from the data set, with each connection bearing a weight that signifies the connection's strength. The input layer processes the state of the environment, passing it to the subsequent layer through weighted connections. The weighted sum in each node is then subjected to an activation function, leading to an output that serves as input for the next layer. This process continues until reaching the output layer, where the number of nodes corresponds to the possible actions.

Network Training

Training the model involves optimizing weights to accurately map input data to the correct output class. Stochastic Gradient Descent (SGD) is a widely used optimizer, adjusting weights based on the error or difference between the predicted and ideal values. By repeatedly feeding data to the network and optimizing weights, the network progressively learns to align predictions with optimal values.

Experience Replay and Replay Memory

Agent experiences at each time step are stored in a dataset known as replay memory, comprising state, action, reward, and next state tuples. These experiences are collected over all episodes and kept within a finite-sized memory. After each episode, random samples are extracted from the memory to train the network, avoiding high data correlation. The replay memory process involves initializing memory capacity, training the network with random weights, and sampling random batches for training.

$$\delta_j = R_j + \gamma_j Q_{target}(S_j, \operatorname{argmax}_a Q(S_j, a)) - Q(S_{(j-1)}, A_{j-1})$$

3.5.3 PPO-Proximal Policy Optimization

With supervised learning, we can easily implement the cost function, run gradient descent on it, and be very confident that we'll get excellent results with relatively little hyperparameter tuning. The route to success in reinforcement learning isn't as obvious—the algorithms have many moving parts that are hard to debug, and they require substantial effort in tuning in order to get good results. PPO strikes a balance between ease of implementation, sample complexity, and ease of tuning, trying to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small.

The idea with Proximal Policy Optimization (PPO) is that we want to improve the training stability of the policy by limiting the change you make to the policy at each training epoch: we want to avoid having too large policy updates.

For two reasons:

We know empirically that smaller policy updates during training are more likely to converge to an optimal solution.

A too big step in a policy update can result in falling “off the cliff” (getting a bad policy) and having a long time or even no possibility to recover.

So with PPO, we update the policy conservatively. To do so, we need to measure how much the current policy changed compared to the former one using a ratio calculation between the current and former policy. And we clip this ratio in a range $[1-\epsilon, 1+\epsilon]$, meaning that we remove the incentive for the current policy to go too far from the old one (hence the proximal policy term).

$$L^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta))\hat{A}_t, \operatorname{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)]$$

The Ratio Function

$$L^{CLIP}(\theta) = \hat{E}_t\left[\min(\boxed{r_t(\theta)}\hat{A}_t, \operatorname{clip}(\boxed{r_t(\theta)}, 1 - \epsilon, 1 + \epsilon)\hat{A}_t)\right]$$

This ratio is calculated this way:

$$r_t(\theta) = \frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)}$$

It's the probability of taking action a-t at state s-t in the current policy divided by the previous one.

As we can see, $R_t(\theta)$ denotes the probability ratio between the current and old policy:

If

- If $R_t(\theta) > 1$, the action a_t at state s_t is more likely in the current policy than the old policy.
- If $R_t(\theta)$ is between 0 and 1, the action is less likely for the current policy than for the old one.

So this probability ratio is an easy way to estimate the divergence between old and current policy.

The unclipped part of the Clipped Surrogate Objective function

This ratio can replace the log probability we use in the policy objective function. This gives us the left part of the new objective function: multiplying the ratio by the advantage.

$$L^{CPI}(\theta) = \mathbb{E}_t \left[\frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)} \hat{A}_t \right] = \mathbb{E}_t [r_t(\theta) \hat{A}_t]$$

However, without a constraint, if the action taken is much more probable in our current policy than in our former, this would lead to a significant policy gradient step and, therefore, an excessive policy update.

The clipped Part of the Clipped Surrogate Objective function

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right]$$

Consequently, we need to constrain this objective function by penalizing changes that lead to a ratio away from 1 (in the paper, the ratio can only vary from 0.8 to 1.2).

By clipping the ratio, we ensure that we do not have a too large policy update because the current policy can't be too different from the older one.

To do that, we have two solutions:

- TRPO (Trust Region Policy Optimization) uses KL divergence constraints outside the objective function to constrain the policy update. But this method is complicated to implement and takes more computation time.
- PPO clip probability ratio directly in the objective function with its Clipped surrogate objective function.

This clipped part is a version where $r_t(\theta)$ is clipped between $[1-\epsilon, 1+\epsilon]$.

With the Clipped Surrogate Objective function, we have two probability ratios, one non-clipped and one clipped in a range (between $[1-\epsilon, 1+\epsilon]$, epsilon is a hyperparameter that helps us to define this clip range (in the paper $\epsilon=0.2$)).

Then, we take the minimum of the clipped and non-clipped objective, so the final objective is a lower bound (pessimistic bound) of the unclipped objective.

Taking the minimum of the clipped and non-clipped objective means we'll select either the clipped or the non-clipped objective based on the ratio and advantage situation.

The final Clipped Surrogate Objective Loss for PPO Actor-Critic style looks like this, it's a combination of Clipped Surrogate Objective function, Value Loss Function and Entropy bonus.

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t [L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)]$$

c1 and c2 are coefficients.

Squared-error value loss: $(V_\theta(s_t) - V_t^{\text{targ}})^2$

Add an entropy bonus to ensure sufficient exploitation.

Algorithm 1 PPO-Clip

- 1: Input: initial policy parameters θ_0 , initial value function parameters ϕ_0
- 2: **for** $k = 0, 1, 2, \dots$ **do**
- 3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
- 4: Compute rewards-to-go \hat{R}_t .
- 5: Compute advantage estimates, \hat{A}_t (using any method of advantage estimation) based on the current value function V_{ϕ_k} .
- 6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg \max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.

- 7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg \min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T (V_\phi(s_t) - \hat{R}_t)^2,$$

typically via some gradient descent algorithm.

- 8: **end for**
-

3.5.4 TD3-Twin-Delayed deep deterministic policy gradient

TD3 is the successor to the Deep Deterministic Policy Gradient (DDPG)(Lillicrap et al, 2016). Up until recently, DDPG was one of the most used algorithms for continuous control problems such as robotics and autonomous driving. Although DDPG is capable of providing excellent results, it has its drawbacks. Like many RL algorithms training DDPG can be unstable and heavily reliant on finding the correct hyper parameters for the current task (OpenAI Spinning Up, 2018).

This is caused by the algorithm continuously over estimating the Q values of the critic (value) network. These estimation errors build up over time and can lead to the agent falling into a local optima or experience catastrophic forgetting. TD3 addresses this issue by focusing on reducing the overestimation bias seen in previous algorithms.

This is done with the addition of 3 key features:

- Using a pair of critic networks (The twin part of the title)
- Delayed updates of the actor (The delayed part)
- Action noise regularisation

During training, a TD3 agent:

- Updates the actor and critic properties at each time step during learning.
- Stores past experiences using a circular experience buffer. The agent updates the actor and critic using a mini-batch of experiences randomly sampled from the buffer.
- Perturbs the action chosen by the policy using a stochastic noise model at each training step.

Algorithm 1 TD3

1	Initialize critic networks $Q_{\theta_1}, Q_{\theta_2}$, and actor network π_ϕ with random parameters θ_1, θ_2, ϕ Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$
2	Initialize replay buffer \mathcal{B}
	for $t = 1$ to T do
3	Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
4	Store transition tuple (s, a, r, s') in \mathcal{B}
5	Sample mini-batch of N transitions (s, a, r, s') from \mathcal{B} $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$ $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$ Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
6	if $t \bmod d$ then Update ϕ by the deterministic policy gradient: $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a) _{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
7	Update target networks: $\theta'_i \leftarrow \tau \theta_i + (1 - \tau) \theta'_i$ $\phi' \leftarrow \tau \phi + (1 - \tau) \phi'$
	end if
	end for

Fig. 4 Methodology of TD3

1. Initialise networks
2. Initialise replay buffer
3. Select and carry out action with exploration noise
4. Store transitions
5. Update critic
6. Update actor
7. Update target networks
8. Repeat until sentient

Actor and Critic Functions

To estimate the policy and value function, a TD3 agent maintains the following function approximators:

- Deterministic actor $\pi(S;\theta)$ — The actor, with parameters θ , takes observation S and returns the corresponding action that maximizes the long-term reward.
- Target actor $\pi(S;\theta_t)$ — To improve the stability of the optimization, the agent periodically updates the target actor parameters θ_t using the latest actor parameter values.
- One or two Q-value critics $Q_k(S,A;\phi_k)$ — The critics, each with different parameters ϕ_k , take observation S and action A as inputs and returns the corresponding expectation of the long-term reward.
- One or two target critics $Q_{tk}(S,A;\phi_{tk})$ — To improve the stability of the optimization, the agent periodically updates the target critic parameters ϕ_{tk} using the latest corresponding critic parameter values. The number of target critics matches the number of critics.

Both $\pi(S;\theta)$ and $\pi(S;\theta_t)$ have the same structure and parameterization.

For each critic, $Q_k(S,A;\phi_k)$ and $Q_{tk}(S,A;\phi_{tk})$ have the same structure and parameterization.

When using two critics, $Q_1(S,A;\phi_1)$ and $Q_2(S,A;\phi_2)$, each critic can have a different structure, though TD3 works best when the critics have the same structure. When the critics have the same structure, they must have different initial parameter values.

During training, the agent tunes the parameter values in θ . After training, the parameters remain at their tuned value and the trained actor function approximator is stored in $\pi(S)$.

Key Equations

Target policy smoothing.

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

Actions used to form the Q-learning target are based on the target policy. Target policy smoothing essentially serves as a regularizer for the algorithm.

It addresses a particular failure mode that can happen in DDPG: if the Q-function approximator develops an incorrect sharp peak for some actions, the policy will quickly exploit that peak and then have brittle or incorrect behaviour. This can be averted by smoothing out the Q-function over similar actions, which target policy smoothing is designed to do.

Clipped double-Q learning. Both Q-functions use a single target, calculated using whichever of the two Q-functions gives a smaller target value:

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{i,\text{targ}}}(s', a'(s')),$$

and then both are learned by regressing to this target

$$L(\phi_1, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}} \left[\left(Q_{\phi_1}(s, a) - y(r, s', d) \right)^2 \right],$$

$$L(\phi_2, \mathcal{D}) = \mathbb{E}_{(s,a,r,s',d) \sim \mathcal{D}} \left[\left(Q_{\phi_2}(s, a) - y(r, s', d) \right)^2 \right].$$

Using the smaller Q-value for the target, and regressing towards that, helps fend off overestimation in the Q-function.

Lastly: the policy is learned just by maximizing Q_{ϕ_1} :

$$\max_{\theta} \mathbb{E}_{s \sim \mathcal{D}} [Q_{\phi_1}(s, \mu_{\theta}(s))],$$

which is pretty much unchanged from DDPG. However, in TD3, the policy is updated less frequently than the Q-functions are. This helps damp the volatility that normally arises in DDPG because of how a policy update changes the target.

3.6 Simulation

A robot that mimics the real robot has been designed in the Gazebo environment, and all the codes were tested on it. The Gazebo environment contains regular environment variables, and Sun is the only light source.

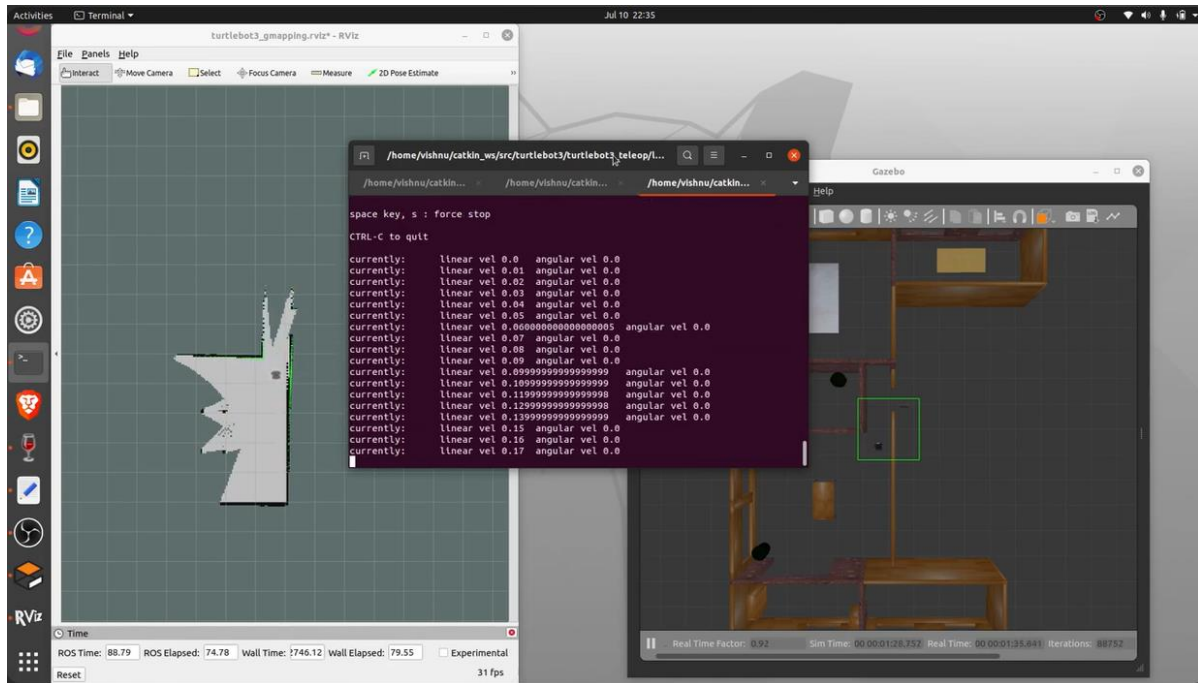


Fig. 18 World Mapping through Rviz

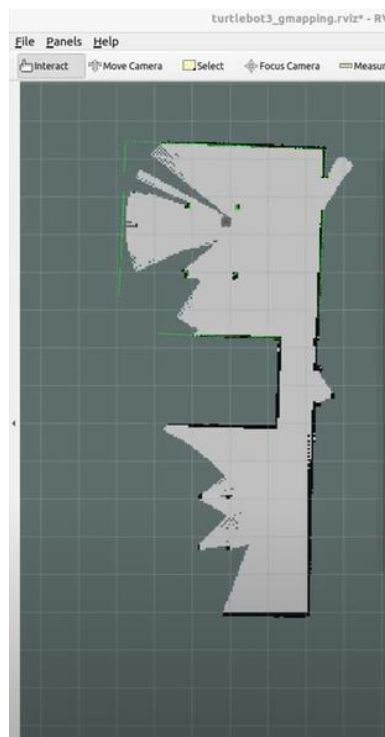


Fig. 19 Mapping progress

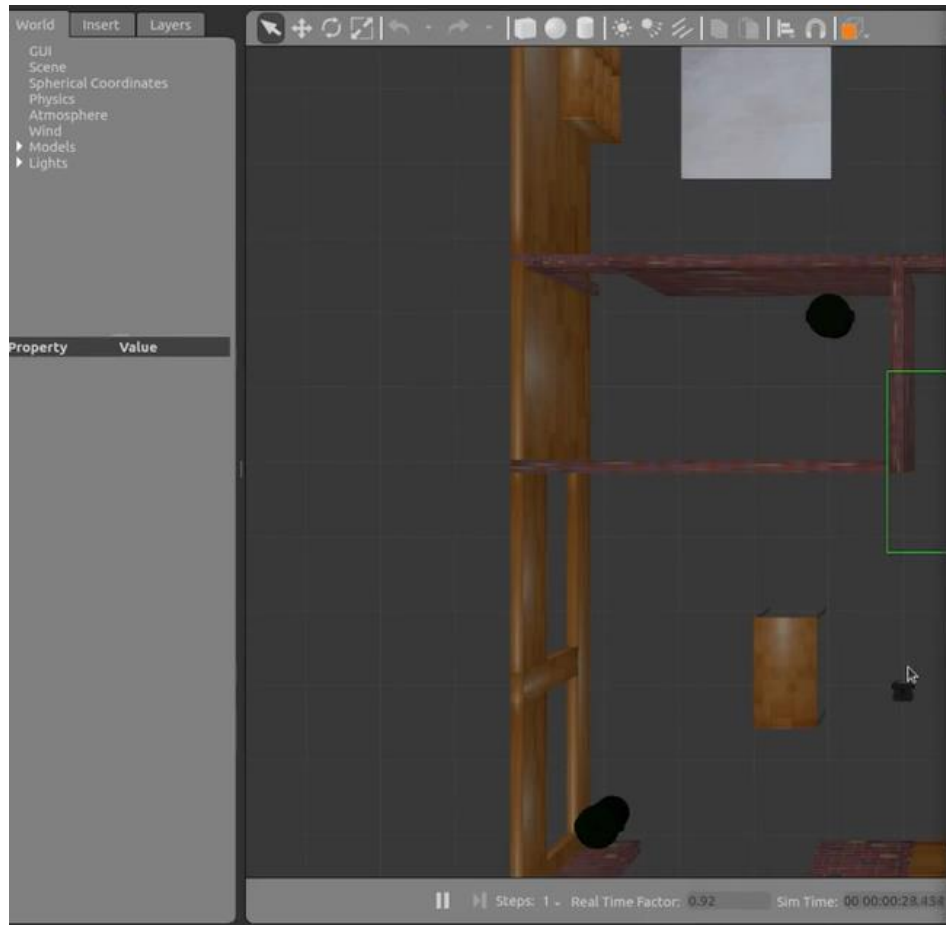


Fig. 20 Robot spawned in house world

Fig. 7 describes the robot spawned in the custom world designed with house on the ground plane. Fig. 5 illustrates the robot navigation along the house for mapping. Fig. 6 shows the grid mapping from every location based on obstacle detection using lidar sensor. In Fig. 8 Fully mapped environment ready for movement where pink line shows destination. Fig 9. Shows The process of planning robot is doing to reach destination

```
vishnu@vishnu-0308:~$ rosrn turtlebot3_example turtlebot3_pointop_key

control your Turtlebot3!
-----
Insert xyz - coordinate.
x : position x (m)
y : position y (m)
z : orientation z (degree: -180 ~ 180)
If you want to close, insert 's'
-----

| x | y | z |
0.4 0.2 0
□
```

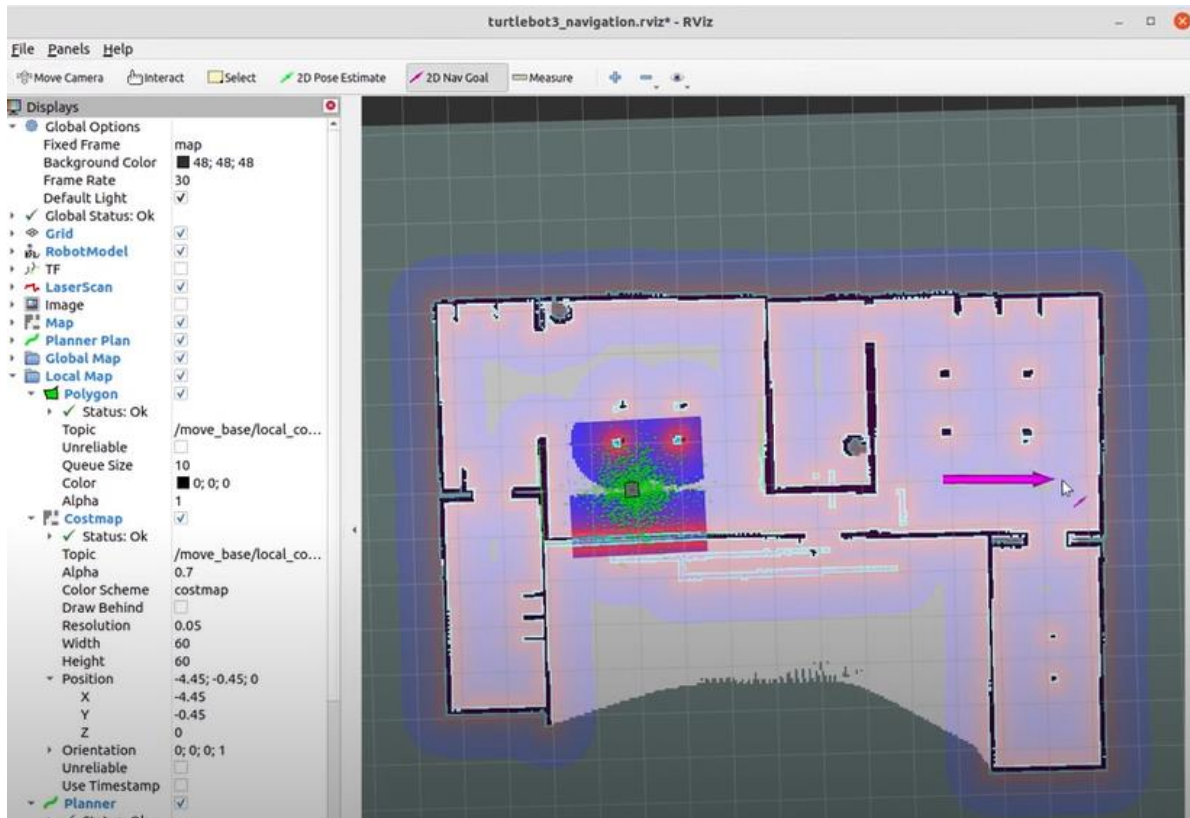



Fig.21 Pointing the goal destination

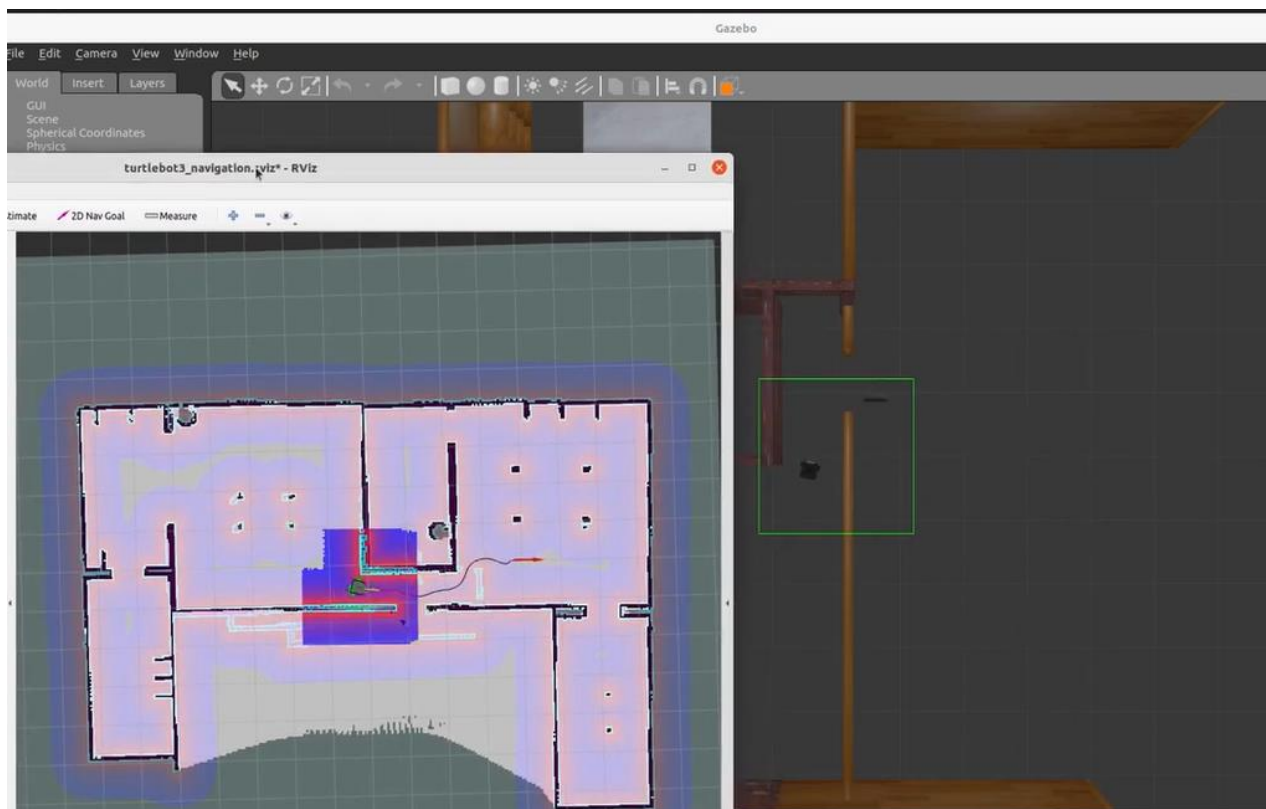


Fig. 22 Robot navigation with respect to the location and obstacles

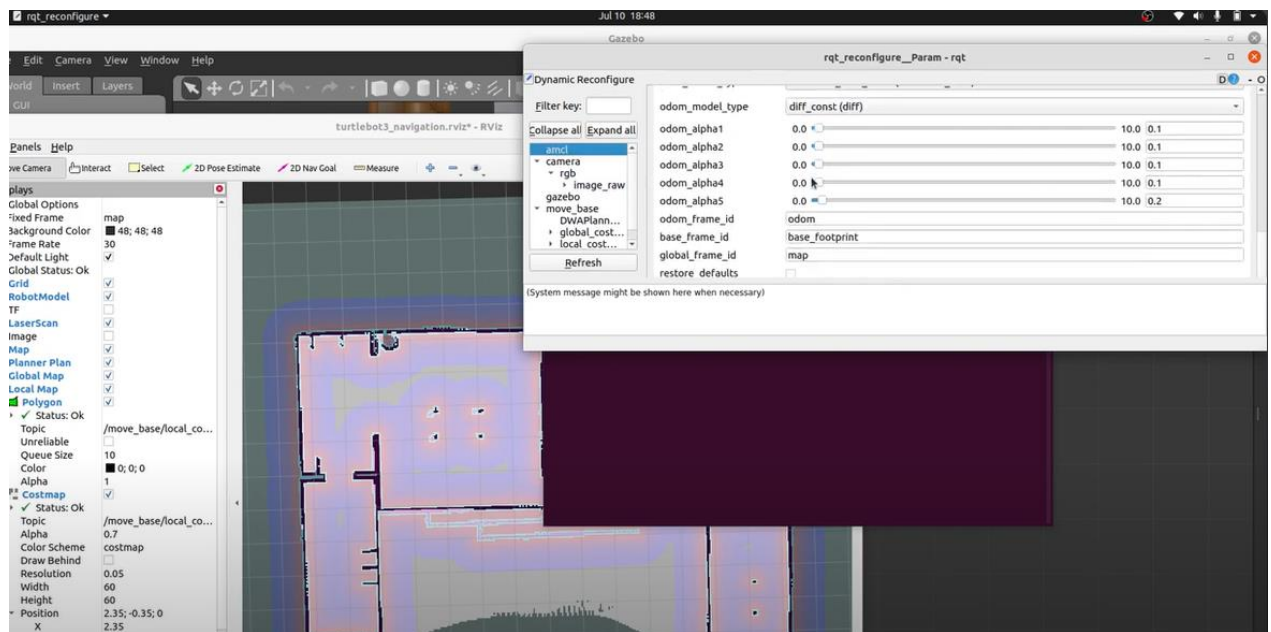


Fig 23. Reconfiguration of variables manually

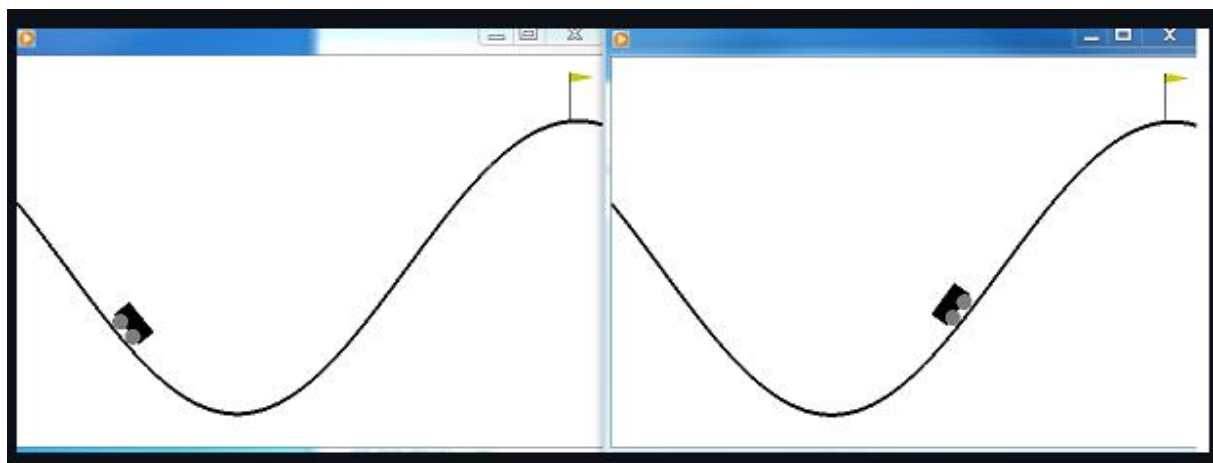


Fig 24. Simulation for Q-learning



Fig 25. Simulation for DQN

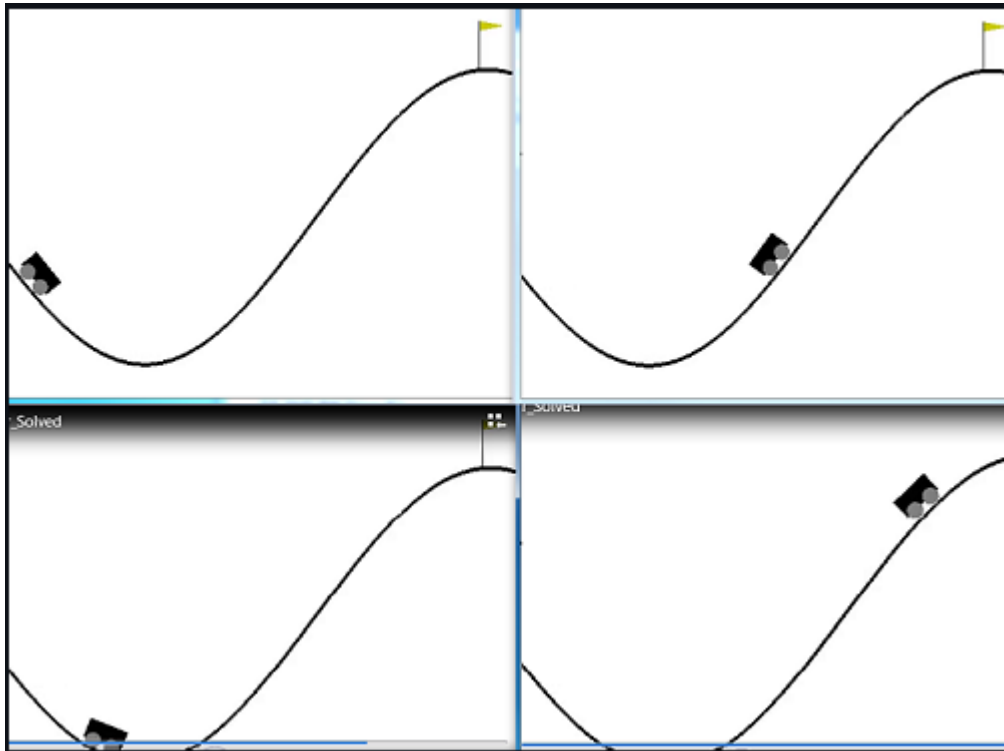


Fig 26. Simulation in PPO

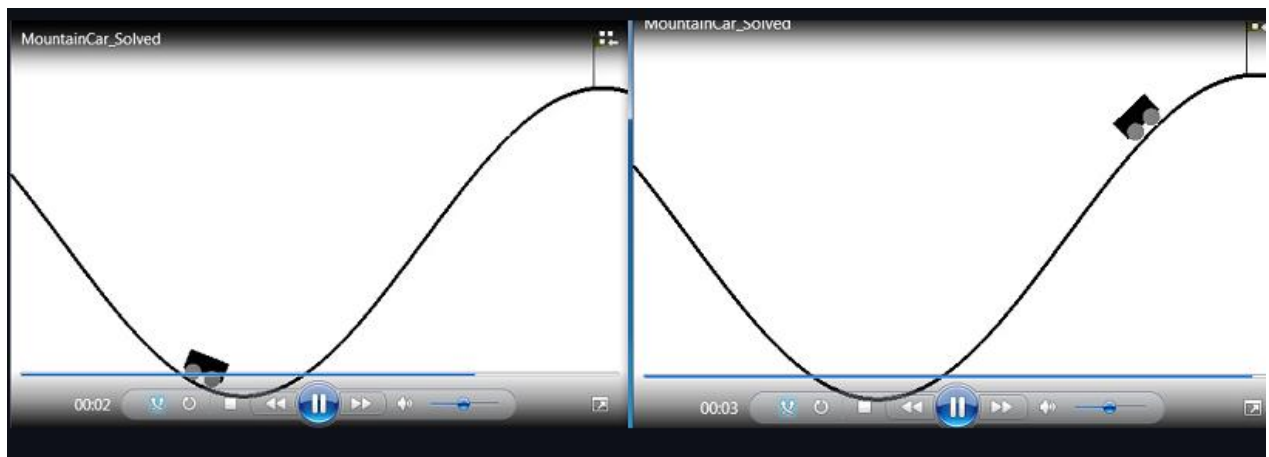


Fig 27. Simulation in TD3

Here in above 4 figures snapshots are taken from different episodes where trying to reach towards Goal flag.

CHAPTER 4

RESULTS AND DISCUSSION

1.5 Results For Ros simulation

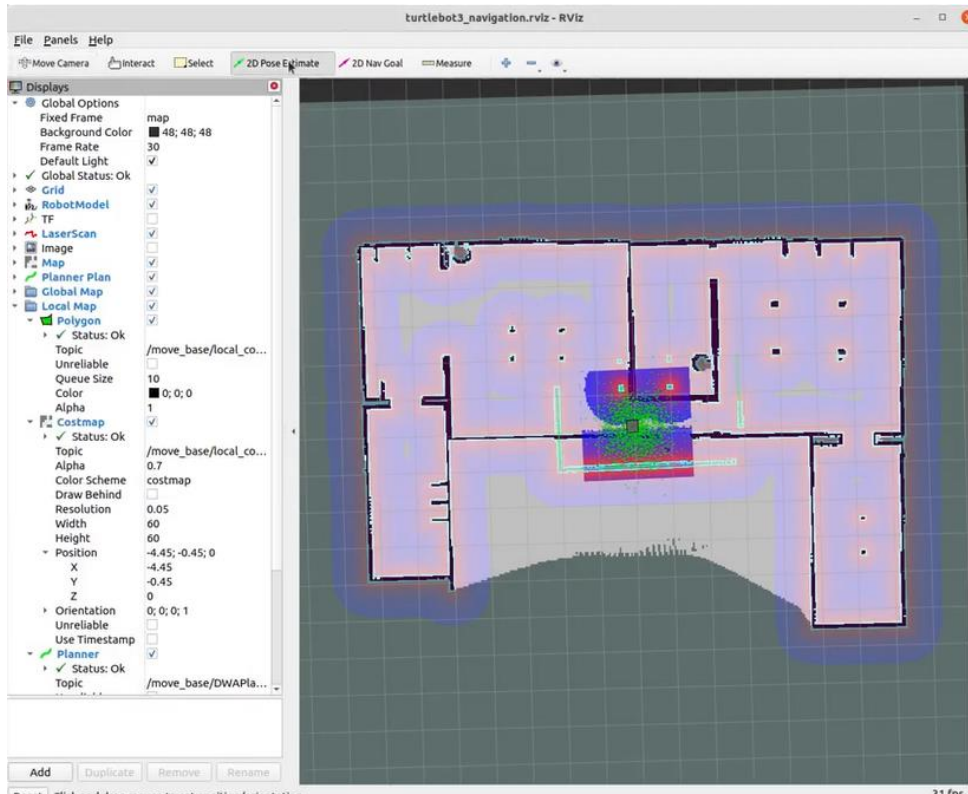


Fig. 28 Screenshot from robot while mapping is complete

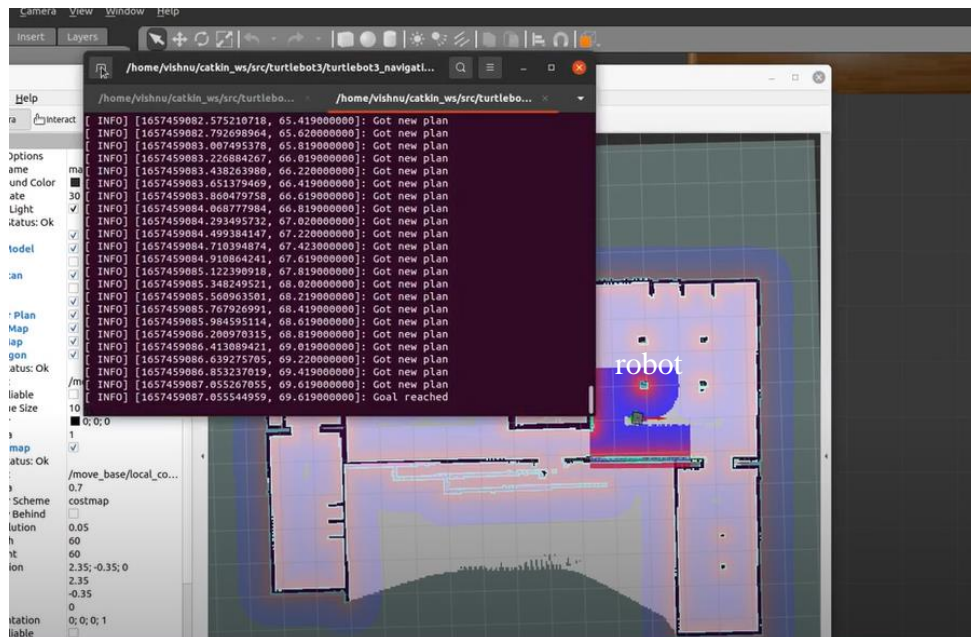


Fig 29 Robot Reached Goal Successfully

1.6 Results of Q-learning

Since gravity is stronger than the car's engine, even at full throttle, the car cannot simply accelerate up the steep slope. The car is situated in a valley and must learn to leverage potential energy by driving up the opposite hill before the car is able to make it to the goal at the top of the rightmost hill. The domain has been used as a test bed in various Reinforcement Learning papers.

Environment and reward threshold

Solving the environment require an average total reward of over -110 on 100 consecutive episodes.

By using the Q-learning algorithm we solve MountainCar-v0 environment in 283600 episodes in 22 minutes !.

Q-learning is the core of the DQN (Deep Q-Network) algorithm, but it is not Deep Learning, since we do not use neural networks!

Discretization

The state space [position x velocity] is discretized into 12x12 buckets.

```
def discretize_state(self, obs):
    discretized = list()
    for i in range(len(obs)):
        scaling = (obs[i] + abs(self.lower_bounds[i])) / (self.upper_bounds[i] - self.lower_bounds[i])
        new_obs = int(round((self.buckets[i] - 1) * scaling))
        new_obs = min(self.buckets[i] - 1, max(0, new_obs))
        discretized.append(new_obs)
    return tuple(discretized)
```

Training Score

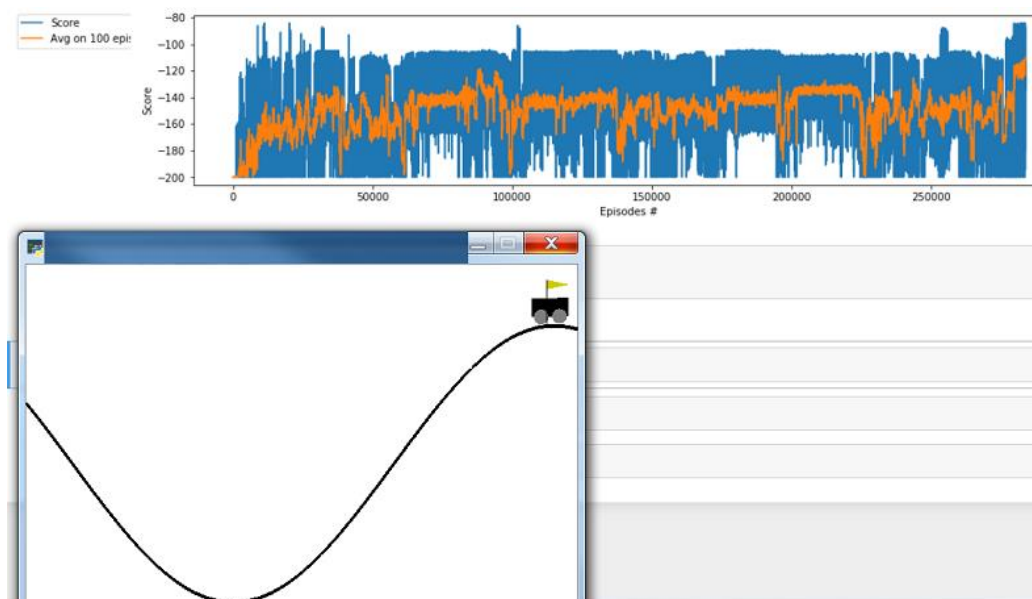


Fig.30 Q-Learning

The last few lines from the log

Time: 00:21:57

Episode: 281200, Timesteps: 135, Score: -135.0, Avg.Score: -120.35, eps-greedy: 0.01,

Time: 00:21:59

Episode: 281600, Timesteps: 116, Score: -116.0, Avg.Score: -122.64, eps-greedy: 0.01,

Time: 00:22:00

Episode: 282000, Timesteps: 97, Score: -97.0, Avg.Score: -116.98, eps-greedy: 0.01,

Time: 00:22:02

Episode: 282400, Timesteps: 89, Score: -89.0, Avg.Score: -121.86, eps-greedy: 0.01,

Time: 00:22:04

Episode: 282800, Timesteps: 113, Score: -113.0, Avg.Score: -119.09, eps-greedy: 0.01,

Time: 00:22:05

Episode: 283200, Timesteps: 85, Score: -85.0, Avg.Score: -115.96, eps-greedy: 0.01,

Time: 00:22:07

Episode: 283600, Timesteps: 114, Score: -114.0, Avg.Score: -109.86, eps-greedy: 0.01,

Time: 00:22:08

Environment solved in 283600 episodes! Average Score: -109.86

Finished training!

1.7 Results of DQN

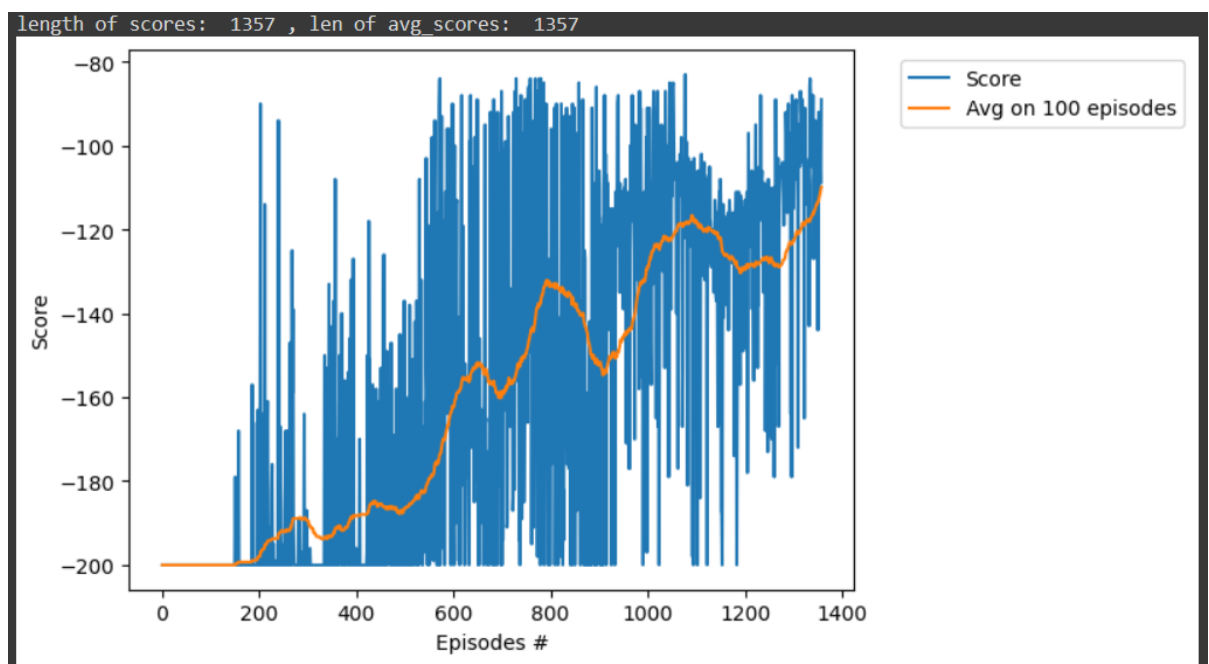


Fig.31 DQN

The last few lines from the log

Episode: 1270 Score: -112.0 Avg.Score: -128.87, eps-greedy: 0.010 Time: 00:12:02

Episode: 1280 Score: -108.0 Avg.Score: -126.99, eps-greedy: 0.010 Time: 00:12:06
 Episode: 1290 Score: -106.0 Avg.Score: -123.44, eps-greedy: 0.010 Time: 00:12:09
 Episode: 1300 Score: -106.0 Avg.Score: -122.31, eps-greedy: 0.010 Time: 00:12:13
 Episode: 1310 Score: -103.0 Avg.Score: -120.87, eps-greedy: 0.010 Time: 00:12:17
 Episode: 1320 Score: -102.0 Avg.Score: -118.33, eps-greedy: 0.010 Time: 00:12:21
 Episode: 1330 Score: -110.0 Avg.Score: -117.58, eps-greedy: 0.010 Time: 00:12:24
 Episode: 1340 Score: -127.0 Avg.Score: -116.30, eps-greedy: 0.010 Time: 00:12:28
 Episode: 1350 Score: -107.0 Avg.Score: -113.35, eps-greedy: 0.010 Time: 00:12:32

Environment solved in 1356 episodes! Average Score: -109.86

Solving the environment require an average total reward of over -110 on 100 consecutive episodes.

We solve the MountainCar environment in 1356 episodes, in 1.75 hours.

By usage of the Q-learning algorithm, the environment is solved in 283600 episodes in 22 minutes!

1.8 Result of TD3

Solving the environment require an average total reward of over 90 over 100 consecutive episodes.

The environment is solved in 1156 episodes in 10 hours by usage of the Twin Delayed DDPG (TD3) algorithm.

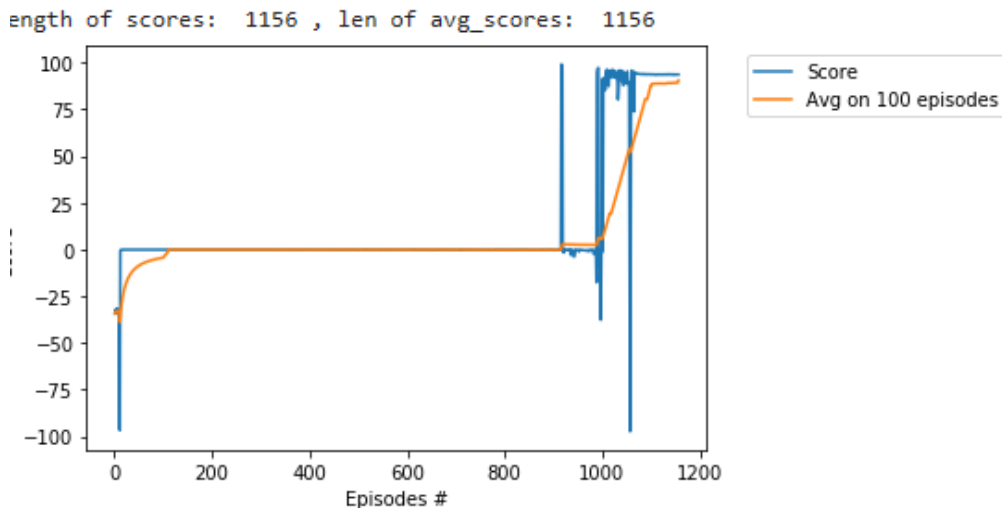


Fig. 32 TD3

The last few lines from the log

Ep. 1110, Timestep 1016360, Ep.Timesteps 70, Score: 93.55, Avg.Score: 88.831, Max.Score: 96.26, Time: 10:14:05

Ep. 1120, Timestep 1017028, Ep.Timesteps 66, Score: 93.75, Avg.Score: 88.867, Max.Score: 95.93, Time: 10:14:30

Ep. 1130, Timestep 1017700, Ep.Timesteps 68, Score: 93.64, Avg.Score: 88.817,

Max.Score: 95.93, Time: 10:14:55

Ep. 1140, Timestep 1018362, Ep.Timesteps 65, Score: 93.77, Avg.Score: 89.098, Max.Score: 95.83, Time: 10:15:20

Ep. 1150, Timestep 1019031, Ep.Timesteps 68, Score: 93.64, Avg.Score: 89.277, Max.Score: 95.83, Time: 10:15:44

Ep. 1156, Timestep 1019431, Ep.Timesteps 66, Score: 93.68, Avg.Score: 90.408, Max.Score: 95.83, Time: 10:15:59

Environment solved with Average Score: 90.40762049331501

1.9 Result of PPO

Environment:

Usually, solving the environment require an average total reward of over the threshold over 100 consecutive episodes.

However, in this case the solution is achieved very fastly: in 21 episodes in 4 minutes !

This is due to the fact

that there are 16 processes in use in this PPO implementation notebook. We can think that the real number of episodes

is $21 \times 16 = 336$.

Vectorized Environments (in our case there are 16 environments) is a method that means that the agent is trained in 16 environments simultaneously.

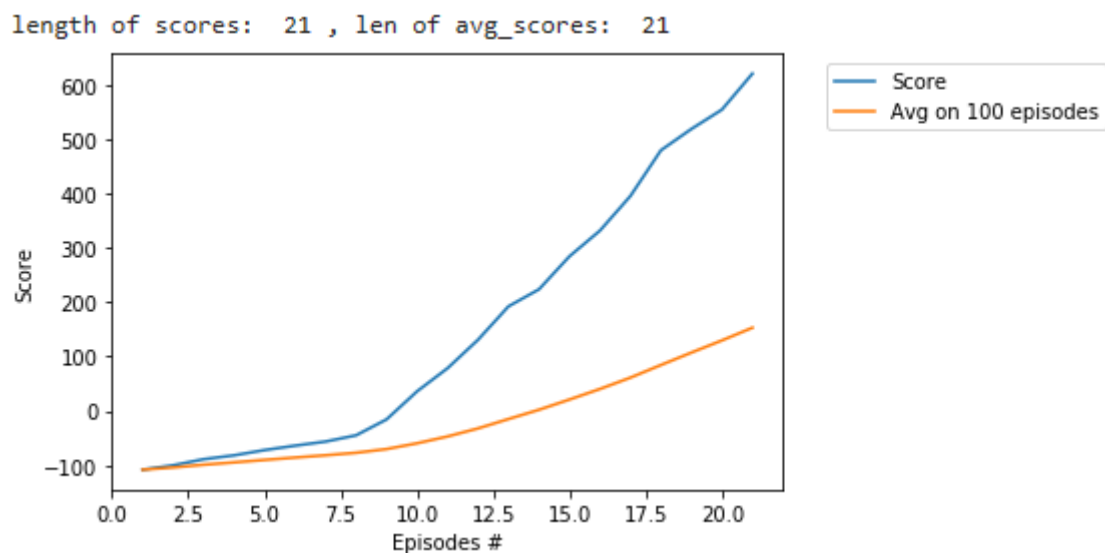


Fig 33 PPO

Log

Ep. 17, Timesteps 999, Score.Agents: 479.05, Avg.Score: 84.20, Time: 00:01:10, Interval: 00:04

Ep. 18, Timesteps 999, Score.Agents: 518.25, Avg.Score: 107.04, Time: 00:01:14,

Interval: 00:04

Ep. 19, Timesteps 999, Score.Agents: 553.75, Avg.Score: 129.38, Time: 00:01:18,

Interval: 00:04

Ep. 20, Timesteps 999, Score.Agents: 619.99, Avg.Score: 152.74, Time: 00:01:22,

Interval: 00:04

Environment solved with Average Score: 152.7411231610043

CHAPTER 5

CONCLUSIONS

This project also delved into the realm of mobile robotics, specifically focusing on the TurtleBot 3 in ROS Gazebo. The integration of Simultaneous Localization and Mapping (SLAM) techniques, including Dijkstra's algorithm, A*, and the Dynamic Window Approach (DWA) planner, allowed the TurtleBot 3 to navigate autonomously within a simulated environment.

The mapping phase involved creating a detailed map of the environment using SLAM techniques, and the navigation phase utilized path planning algorithms such as Dijkstra's and A* to find optimal paths for the robot. The DWA planner facilitated real-time reactive navigation, enabling the TurtleBot 3 to dynamically adjust its trajectory based on the immediate surroundings.

The map-based navigation, employing algorithms like Dijkstra's and A*, focused on pre-mapping the environment and planning paths beforehand. This approach is beneficial when the entire environment is known in advance and allows for optimal path selection. It is particularly useful in structured environments like warehouses, where the robot can plan paths based on the pre-existing map.

On the other hand, reactive navigation, implemented through the DWA planner, operates in real-time, making instantaneous decisions based on sensor data without relying on a pre-built map. This approach is advantageous in dynamic and changing environments where the robot must adapt quickly to obstacles or unforeseen changes.

The project's dual focus on reinforcement learning for the MountainCar-v0 environment and mobile robotics with TurtleBot 3 demonstrates the versatility of robotic systems across different applications. The combination of these approaches provides a comprehensive understanding of navigation strategies, showcasing the strengths and trade-offs between map-based and reactive navigation in robotic systems.

The project also involved exploring and implementing various reinforcement learning algorithms, including Q-learning, Deep Q Networks (DQN), Twin Delayed DDPG (TD3), and Proximal Policy Optimization (PPO), to tackle the challenging problem of navigating the MountainCar-v0 environment. Each algorithm was assessed in terms of its effectiveness and efficiency in solving the specified task. Here are the key findings:

- Q-learning: The traditional Q-learning algorithm, while not a deep learning approach, demonstrated satisfactory performance by solving the environment in 283,600 episodes within 22 minutes. The discretization of the state space into 12x12 buckets played a crucial role in achieving the desired results.
- DQN (Deep Q Networks): Employing a neural network to approximate the optimal Q-

function, DQN showcased notable efficiency, solving the environment in 1,356 episodes over 1.75 hours. The utilization of deep learning principles contributed to the algorithm's ability to generalize and learn complex patterns.

- TD3 (Twin Delayed DDPG): TD3, a state-of-the-art deep reinforcement learning algorithm, exhibited a longer training time, solving the environment in 1,156 episodes over 10 hours. The Twin Delayed DDPG algorithm is known for its stability and robustness.
- PPO (Proximal Policy Optimization): PPO, leveraging vectorized environments with 16 processes, demonstrated exceptional speed in solving the environment. Achieving the solution in 21 episodes within 4 minutes, PPO's efficiency can be attributed to the parallelization of training processes.

In summary, each algorithm presented unique strengths and trade-offs. Q-learning, despite its simplicity, provided a baseline for comparison. DQN showcased the benefits of deep learning in reinforcement learning tasks, while TD3 emphasized stability at the cost of training time. PPO, with its parallelization strategy, delivered remarkable efficiency. The choice of algorithm depends on specific project requirements, considering factors such as computational resources, training time, and task complexity. This project contributes valuable insights into the comparative performance of reinforcement learning algorithms in solving a challenging navigation problem.

Future Scope:

Unified Metric Development:

Explore the creation of a standardized metric for evaluating both map-based and reactive navigation systems. This metric should consider factors like efficiency, adaptability, and computational overhead to provide a comprehensive assessment.

Integration of SLAM with DRL:

Investigate the integration of Simultaneous Localization and Mapping (SLAM) techniques with Deep Reinforcement Learning (DRL) to enhance spatial awareness and adaptability in novel environments.

Exploration of Self-Adaptive Critic (SAC) in DRL:

Research the application of Self-Adaptive Critic (SAC) mechanisms in Deep Reinforcement Learning for robotic navigation. SAC introduces dynamic adjustments to the critic network during training, potentially improving learning efficiency and adaptability.

Advanced Learning Techniques:

Focus on incorporating advanced learning techniques, such as SAC, to enhance the adaptability and intelligence of robotic navigation systems. This could lead to more efficient learning processes and improved performance in diverse environmental

Signature of the Guide

Name : Dr. Palani Duraisamy

CHAPTER – 6

REFERENCES

- [1] Mohammed Alhawary. Reinforcement-learning-based navigation for autonomous mobile robots in unknown environments. Master's thesis, University of Twente, 2018.
- [2] Hee Rak Beom and Hyung Suck Cho. A sensor-based navigation for a mobile robot using fuzzy logic and reinforcement learning. *IEEE transactions on Systems, Man, and Cybernetics*, 25(3):464–477, 1995.
- [3] XIA Chen and Abdelkader El Kamel. A reinforcement learning method of obstacle avoidance for industrial mobile vehicles in unknown environments using neural net-work. In *Proceedings of the 21st International Conference on Industrial Engineering and Engineering Management 2014*, pages 671–675. Springer, 2015.
- [4] Mohammad Abdel Kareem Jaradat, Mohammad Al-Rousan, and Lara Quadan. Reinforcement based mobile robot navigation in dynamic environment. *Robotics and Computer-Integrated Manufacturing*, 27(1):135–149, 2011.
- [5] Lazhar Khriji, Farid Touati, Kamel Benhmed, and Amur Al-Yahmedi. Mobile robot navigation based on q-learning technique. *International Journal of Advanced Robotic Systems*, 8(1):4, 2011.
- [6] Krzysztof Kozłowski and Dariusz Pazderski. Modeling and control of a 4-wheel skid-steering mobile robot. *International journal of applied mathematics and computer science*, 14:477–496, 2004.
- [7] Maxim Lapan. *Deep Reinforcement Learning Hands-On*. Packt Publishing, Birmingham, UK, 2018.
- [8] Madson Rodrigues Lemos, Anne Vitoria Rodrigues de Souza, Renato Souza de Lira, Carlos Alberto Oliveira de Freitas, Vanderlei João da Silva, and Vicente Ferreira de Lucena. Robot training and navigation through the deep q-learning algorithm. In *2021 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–6, 2021.
- [9] Kristijan Macek, Ivan Petrović, and N Perić. A reinforcement learning approach to obstacle avoidance of mobile robots. In *7th International Workshop on Advanced Motion Control. Proceedings (Cat. No. 02TH8623)*, pages 462–466. IEEE, 2002.
- [10] Enrico Marchesini and Alessandro Farinelli. Discrete deep reinforcement learning for mapless navigation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10688–10694, 2020.