Deep Reinforcement Learning-Based Motion Planning for TurtleBot3

Sumukh Porwal, Piyush Thapar, Ameya Phadnis, Srikanth Natarajan
CS525: Reinforcement Learning
Worcester Polytechnic Institute
Robotics Engineering Department
{sporwal, pthapar, aphadnis, snatarajan1}@wpi.edu

***Abstract*—This work aims to develop an efficient motion planning system for the TurtleBot3 robot using Deep Reinforcement Learning (DRL) techniques within a simulation environment. The objective is to train the robot to navigate safely from an initial position to a designated goal while effectively avoiding obstacles. We leverage and compare multiple DRL algorithms, including DDPG, PPO, TD3, and DQN, to analyze their effectiveness in optimizing navigation performance. Special attention is given to designing reward functions that encourage collision-free navigation and efficient path planning. The evaluation focuses on the algorithms' abilities to achieve smooth, stable trajectories and effective obstacle avoidance.**

***Index Terms*—Deep Reinforcement Learning, Motion Planning, TurtleBot3, Robotics, DQN, PPO, TD3, DDPG.**

I. LITERATURE REVIEW

Significant research has been done recently in the field of reinforcement learning and its application to motion planning in robotics. Traditional methods for mobile robot motion planning often rely on techniques such as Rapidly-exploring Random Trees (RRT) and Probabilistic Roadmaps (PRM). While these methods have proven effective in structured environments, they may struggle in dynamic, unstructured, or partially observable settings.

These shortcomings have led to research in several reinforcement learning-based motion planning techniques. For instance:

- Luo et al. [1] proposed a robust planning method that embeds an experience-based planner and self-imitation learning to mitigate data collection issues.
- Bhuiyan et al. [2] suggest a Deep Reinforcement Learning-based approach for industrial robotic manipulators and demonstrate an improvement over traditional techniques like RRT.
- Teheri et al. [3] demonstrate the use of various reinforcement learning-based techniques like DDPG and PPO to perform successful motion planning for a robot surrounded by obstacles.

These studies highlight the rising research in RL-based methods for robot motion planning and their potential to solve complex problems. Inspired by these works, we aim to apply Deep-Reinforcement Learning techniques to motion planning for a TurtleBot3 robot in simulation.

II. PROBLEM DEFINITION

In this project, we confront the challenge of mapless motion planning for mobile ground robots, with the primary objective to develop a robust translation function for determining the next velocity v_t of a robot based on its current state s_t. The state st encompasses several critical components, formalized as:

$$v_t = f(x_t, p_t, v_{t-1}, \theta_t, \alpha_t)$$

Where Sensor Information x_t consists of data from the robot's sensors to understand the environment, and Relative Target Position pt denotes the location of the target relative to the robot. The Previous Velocity v_{t-1} indicates the robot's last recorded speed, aiding in stability. The Yaw Angle θ_t specifies the robot's orientation, and the Rotation Degree α_t is crucial for aligning the robot with the target. Our goal is to model these states into actionable insights, specifically to compute the next velocity v_t, facilitating agile and accurate navigation.

III. METHODOLOGY

A. Static Environment Setup

A customized simulation environment was created in Gazebo as part of the setup, which includes static obstacles. This environment spawns the TurtleBot3, and loads the required Gazebo world as shown in Figure 1 and 2.

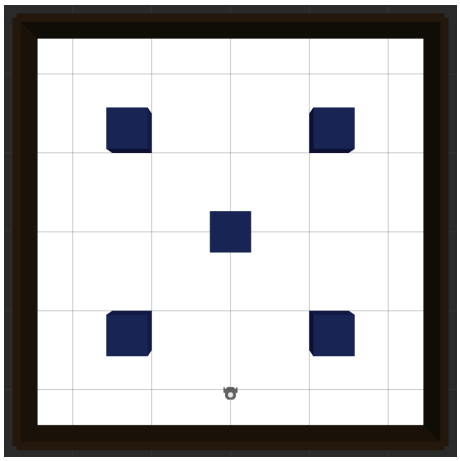

Fig. 1. Top view of the Gazebo World

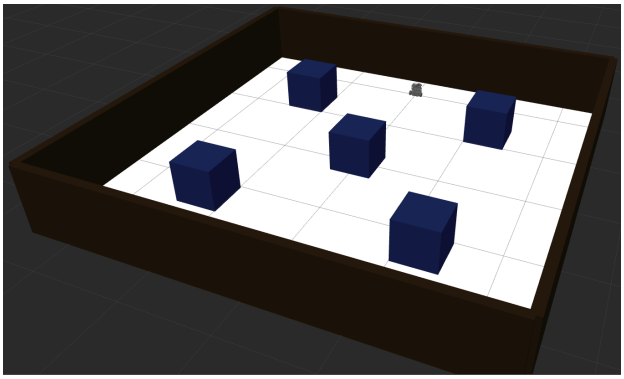

Fig. 2. Orthographic view of the Gazebo World

B. State and Action Representation

- **State**: The input features, forming a **16-dimensional state**, include:
  - **Laser Finding (10 Dimensions)** - Represents sparse laser measurements.
  - **Past Action (2 Dimensions)** - Linear velocity and Angular velocity
  - **Target Position in Robot Frame (2 Dimensions)** - Relative distance and Relative angle (using polar coordinates)
  - **Robot Yaw Angular (1 Dimension)** - Indicates the robot's current yaw angle.
  - **Degrees to Face the Target (1 Dimension)** - The absolute difference between the yaw and the relative angle to the target.
- **Action**: The outputs, forming a 2-dimensional action, consist of Linear Velocity and Angular Velocity.

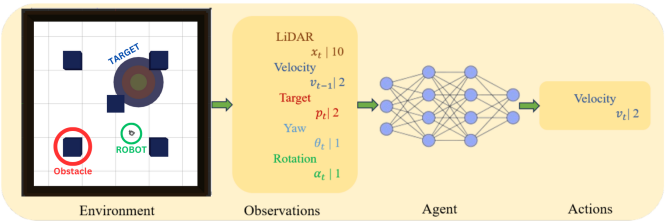

Fig. 3. Overview

C. DRL Algorithms

*1) **Deep Q-Network (DQN) and Double DQN (DDQN):*** In our project, I initially implemented the Deep Q-Network (DQN) [4] algorithm to address the robotic navigation task. However, limitations like overestimation bias and unstable training motivated the transition to Double DQN (DDQN) [5], which provided improved performance through reduced overestimation and enhanced stability.

- **Algorithm Overview**
  - The DQN algorithm learns a Q-function to estimate the expected cumulative reward for state-action pairs.
  - DQN optimizes the Bellman equation:

$$y_t = r_t + \gamma \max_a Q(s_{t+1}, a; \theta')$$

    where θ represents the policy network, and θ' is the target network.
  - DDQN modifies the target calculation to decouple action selection and evaluation, addressing overestimation bias:

$$y_t = r_t + \gamma Q(s_{t+1}, \arg\max_a Q(s_{t+1}, a; \theta); \theta')$$

  - The policy network (θ) selects the action, while the target network (θ') evaluates it, leading to stable updates and more accurate Q-value predictions.
- **Model Architecture**
  - The Q-network consists of fully connected layers to process state-action pairs:
    - **Input Layer:** Encodes the state space into a 128-dimensional vector.
    - **Hidden Layers:** Two hidden layers, each with 128 neurons and `ReLU` activation.
    - **Output Layer:** Produces Q-values for all possible actions.
  - Weight initialization is done using Xavier uniform initialization for better convergence.
- **Hyperparameters**
  - **Episodes:** 100,000
  - **Episode Length:** 2,000 steps
  - **Discount Factor (γ):** 0.99
  - **Epsilon Start:** 1.0
  - **Epsilon End:** 0.05
  - **Epsilon Decay:** 0.995
  - **Learning Rate:** 0.00025
  - **Batch Size:** 256
  - **Replay Buffer Size:** 1,000,000
  - **Target Update Frequency:** 1,000 steps
  - **Loss Function:** Smooth L1 Loss (Huber Loss)
  - **Optimizer:** Adam
- **Reward Function**
  - The reward function incentivizes efficient and safe navigation while penalizing collisions:

$$r(s_t, a_t) = \begin{cases} +400 & \text{if goal reached,} \\ -400 & \text{if collision,} \\ +1 & \text{if closer to goal,} \\ -1 & \text{if moving away.} \end{cases}$$

  - Additional heading alignment reward:

$$r_{\text{heading}} = \max\left(1 - \frac{\text{Heading error}}{\pi}, -1\right),$$

    where the heading error is the angular deviation between the robot's orientation and the target direction.

- **Implementation Details**
  - **Initialization:** The `DDQNAgent` class initializes the policy network, target network, and replay buffer. The **policy network** is responsible for predicting Q-values for actions in a given state, while the **target network** is used to stabilize training by providing more stable targets for the Bellman equation. The **replay buffer** stores transitions of the form $(s, a, r, s', \text{done})$ that the agent experiences during training.
  - **Action Space**
    The action for the robot is discrete and comprises a constant linear velocity and one of five predefined angular velocities. The action is defined as:
    - **Linear Velocity (1 Dimension):** Fixed at 0.12 meters per second.
    - **Angular Velocity (1 Dimension):** Selected from a discrete set of values $[1.5, 0.75, 0.0, -0.75, -1.5]$ radians per second.

    The discrete action space allows the robot to execute predefined angular velocity adjustments while maintaining a constant linear velocity, enabling efficient and predictable navigation.
  - **Action Selection:** The agent uses an ϵ-greedy strategy to balance exploration and exploitation. Initially, ϵ is set to 1 (100% exploration), encouraging the agent to try random actions. Over time, ϵ decays, allowing the agent to exploit its learned knowledge more and explore less.
  - **Experience Replay:** The agent stores transitions $(s, a, r, s', \text{done})$ in the replay buffer. During training, random batches of transitions are sampled to train the neural network. This helps break temporal correlations and stabilize learning.
  - **Optimization:**
    - The policy network is updated by minimizing the temporal difference (TD) error. The loss function is the mean squared error (MSE) between the predicted Q-values $Q(s_t, a_t; \theta)$ and the target Q-values y_t:

      $$\text{Loss} = \mathbb{E}[(y_t - Q(s_t, a_t; \theta))^2].$$

      The target Q-value is computed using the Bellman equation. In DDQN, the policy network is used to select the action, while the target network is used to evaluate it, reducing overestimation bias.
    - The target network is periodically synchronized with the policy network to ensure that the target network's weights remain up-to-date, stabilizing the learning process.
  - **Logging:** Metrics such as loss, average rewards, and success rate are tracked using the `wandb` tool, which helps visualize and monitor training progress.
- **Challenges and Solutions**
  - **Overestimation in DQN:** In standard DQN, the Q-value for a state-action pair is computed using the maximum Q-value for the next state, which can lead to overestimation bias. This bias causes the Q-values to be inflated, resulting in suboptimal policies.
    - **Solution:** The **DDQN** algorithm was introduced to decouple action selection and evaluation. The policy network selects the action, while the target network is used to evaluate it, reducing overestimation bias.
  - **Exploration-Exploitation Tradeoff:** At the start of training, exploration is important to discover new actions, while exploitation is critical in later stages to maximize rewards. Balancing these is a key challenge.
    - **Solution:** The ϵ-decay schedule is carefully tuned to allow thorough exploration in the early episodes and focus on exploitation as training progresses. This helps the agent discover effective actions and then refine its strategy.
  - **Stability Issues:** Stability during training is a common challenge in reinforcement learning. Frequent updates to the Q-values and non-linearities in neural networks can cause instability.
    - **Solution:** Stability is improved by periodically updating the target network and using experience replay. The target network is updated less frequently than the policy network, ensuring more stable learning. Experience replay helps by breaking correlations in the training data and stabilizing updates.

*2) **Proximal Policy Optimization (PPO)**:* In our project, I employed the Proximal Policy Optimization (PPO) algorithm, a well-regarded reinforcement learning technique known for its stability and efficacy in continuous control tasks [6]. PPO iteratively updates the policy to maximize the expected cumulative reward, implementing a constraint to limit policy changes and prevent large, abrupt deviations. This controlled approach to policy updates ensures more stable training progress, making PPO particularly suitable for our mapless motion planning problem. In the Proximal Policy Optimization (PPO) algorithm, the loss function is composed of two integral components: the policy loss and the value loss. The policy loss is crafted to modulate the updates made to the policy, ensuring they remain modest and do not significantly diverge from the current policy. This controlled adjustment is vital for the stability of the training process, enabling a gradual and steady improvement in policy performance. Mathematically, the policy loss can be described as follows:

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

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the likelihood ratio, A_t is the advantage, and ϵ is a clipping parameter.

Here $L^{CLIP}(\theta)$ represents the clipped surrogate objective, θ denotes the parameters of the policy, $r_t(\theta)$ is the probability

ratio of the new policy to the old policy, A_t is the advantage function, and ϵ is a hyperparameter determining the extent of allowed policy changes. The value loss is associated with the critic network and is aimed at minimizing the difference between the predicted value $V(s_t)$ and the actual discounted cumulative reward R_t. The value loss is calculated as:

$$L^{VF}(\theta) = \mathbb{E}[(V(s_t) - R_t)^2]$$

In my implementation of the Proximal Policy Optimization (PPO) algorithm, the actor and critic networks are fundamental components that facilitate the learning process. The actor network is responsible for defining the policy, which specifies the probability distribution of possible actions in a given state. Conversely, the critic network provides an estimate of the value of each state, essentially predicting the expected return from that state. The training of the actor network is guided by the policy loss, which helps in refining the policy to ensure better decision-making in navigating the environment. On the other hand, the critic network is trained to minimize the value loss, aiming to enhance the accuracy of state value predictions. This dual-network approach, where the actor and critic are concurrently trained, is central to the efficiency of the PPO algorithm. By iteratively updating both networks, our system continuously improves in its ability to not only choose optimal actions (via the actor) but also in evaluating the potential future rewards of current states (via the critic). This collaborative training mechanism is particularly effective in our application of mapless motion planning, allowing for a more effective navigation strategy to be developed by the algorithm.

- **Action Space**: The action for the robot is determined by sampling from a Gaussian distribution parameterized by the mean predicted by the actor network and a predefined covariance matrix. The resulting action is a 2-dimensional vector comprising:
  - **Linear Velocity (1 Dimension)**: The sampled value is clipped to lie within the range $[0, 1]$ meters per second.
  - **Angular Velocity (1 Dimension)**: The sampled value is clipped to lie within the range $[-1.0, 1.0]$ radians per second.

  The clipping ensures that the robot's motion remains within realistic physical constraints, enabling smooth and controlled navigation.

- **Actor-Critic Architecture**
  - In my implementation, I have utilized simple neural network architectures for both the actor and critic models, tailored to the needs of efficient robotic navigation.
  - The actor model consists of three fully connected layers. The input layer maps the observation space to 128 neurons, followed by a hidden layer with 64 neurons. The final layer branches into two outputs: one for linear velocity and another for angular velocity. This design enables the model to independently predict these two essential control parameters.
  - The critic model, designed to estimate the state value, follows a straightforward architecture. It features an input layer mapping to 128 neurons, a 64-neuron hidden layer, and a single output neuron that provides the value estimate.
  - Both models employ the `ReLU` activation function in the intermediate layers to introduce non-linearity, facilitating better learning capabilities. In the actor model, the linear velocity output is scaled using a `sigmoid` function, while the angular velocity output is scaled using a hyperbolic tangent (`tanh`) function, ensuring the outputs align with realistic robotic constraints.
  - Weight initialization for all layers in both models is achieved using Xavier uniform initialization. This technique is applied to ensure the weights are well-scaled at the start of training, promoting stable convergence. Biases are initialized to zero to avoid introducing unnecessary bias in the early stages of learning.
  - This approach emphasizes simplicity and computational efficiency, providing a robust framework for effective control policy learning without overcomplicating the model structure.

- **Hyperparameters**
  - **No. of Episodes Trained:** 4,500
  - **Episode Length:** 400 steps
  - **Batch Size:** 4000 steps
  - **Discount Factor (γ):** 0.99
  - **Updates per Iteration:** 50
  - **Total Iterations:** 270
  - **Initial Covariance:** 0.8
  - **Minimum Covariance:** 0.1
  - **Covariance Decay:** 0.95
  - **Learning Rate:** 3×10^{-4}
  - **Clip (ϵ):** 0.2
  - **Critic Loss Function:** MSE Loss
  - **Optimizer:** Adam

- **Reward Function**
  - Our reward function is designed to encourage the agent to move closer to the goal while maintaining an optimal heading angle, penalizing collisions and rewarding goal arrival. It combines dynamic distance and heading angle rewards to achieve balanced behavior.
  - The reward function is calculated as follows:

$$r(s_t, a_t) = \begin{cases} r_{collision} & \text{if } \Delta d < c_o, \\ r_{arrive} & \text{if } d_t < c_g, \\ w_d \cdot r_{\text{distance}} + w_h \cdot r_{\text{heading}} & \text{otherwise.} \end{cases}$$

    where $r_{arrive} = 150$ and $r_{colision} = -100$.
  - The distance reward, r_{distance}, is based on the change in distance to the goal:

$$r_{\text{distance}} = \begin{cases} \Delta d \cdot k & \text{if } \Delta d > 0, \\ \Delta d \cdot 2k & \text{if } \Delta d \leq 0, \end{cases}$$

where $\Delta d = d_{t-1} - d_t$, k is a scaling factor, c_o is near-collision distance, r_{arrive} is granted when the agent is within a critical distance c_g to the target and d_t is the current distance to the goal.

- The heading reward, r_{heading}, is computed based on the heading error:

$$r_{\text{heading}} = 1 - \frac{|((\theta_g - \theta_y + 180) \mod 360) - 180|}{180},$$

where θ_g is the goal angle, θ_y is the current yaw, and the error is normalized between 0 and 1.

- The final reward is a weighted sum of r_{distance} and r_{heading}, where $w_d = 0.7$ and $w_h = 0.3$ are the weights for distance and heading rewards, respectively.
- This reward function effectively balances progress toward the goal with alignment along the correct heading, ensuring robust navigation while discouraging collisions and promoting successful goal completion.

- **Implementation Details**
  - **Initialization:** The `PPO` class initializes hyperparameters such as the learning rate, clipping threshold, and update frequency. Actor and critic networks are instantiated with the respective policy and value function classes. The covariance matrix used for action sampling is set to an initial variance of 0.8.
  - **Training Workflow:** The training is performed via the `learn` function, which iteratively:
    - Collects trajectories using the `rollout` function.
    - Computes the return-to-go and advantage estimates.
    - Updates the actor and critic networks using the surrogate loss objectives.
    - Saves the trained models periodically for checkpointing.
  - **Rollout Procedure:** The `rollout` function collects observations, actions, rewards, and log-probabilities for a batch of trajectories:
    - Actions are sampled from a multivariate Gaussian distribution parameterized by the actor network.
    - Rewards are accumulated, and episodes terminate upon reaching the goal or hitting a predefined time limit.
  - **Optimization and Logging:** Each policy update involves:
    - Normalization of advantages to improve stability.
    - Gradient computation and backpropagation for both actor and critic networks.
    - Logging of losses and performance metrics.

- **Challenges and Solutions**
  - **Balancing Exploration and Exploitation**: Fine-tuning the covariance matrix was essential to strike a balance between exploration and exploitation. This was achieved through iterative testing and parameter adjustments to ensure the agent explores effectively while still converging to optimal policies.
  - **Reward Function Adaptation**: Since the reward function requirements differed across algorithms, convergence for one algorithm did not imply convergence for others. To address this, the reward functions were carefully tailored and dynamically adjusted for each algorithm, ensuring they aligned with the specific learning objectives and behavior requirements.
  - **Handling High-Dimensional Observation Spaces**: The high-dimensional observation spaces typical of TurtleBot navigation posed a challenge. This was mitigated by preprocessing the input data to reduce dimensionality where possible and designing neural network architectures capable of effectively learning from such data while maintaining computational efficiency.

*3) **Deep Deterministic Policy Gradient (DDPG)**:* Deep Deterministic Policy Gradient (DDPG) is a model-free,off policy RL algorithm, It is an advanced reinforcement learning algorithm that extends the deterministic policy gradient apporoach to handle continous action spaces effectively [7]. DDPG is an actor-critic method that combines elements of Deep Q-Networks(DQN) with deterministic policy gradient methods, enabling it to learn a policy directly in high-dimensional action spaces. DDPG was chosen for training the TurtleBot3 due to its effectiveness in continuous action spaces, essential for linear and angular velocity control. The DDPG algorithm was implemented in `ddpg.py` file.

- **Actor-Critic Architecture**: The actor is a policy network that directly outputs the optimal continuous action for a given state. instead of searching through all possible actions(like traditional Q-learning),the actor parameterizes the policy, which maps the state to an action. During training ,it is updated using gradients provided by the critic. [8] the critic evaluates how good the action a is for the given state s, and the actor adjusts itself to maximize the expected Q-value. The critic is a value function approximator that evaluates the quality of the actions suggested by the actor. it estimates the Q-value, it is trained using the Bellman equation to minimize the temporal difference error. The actor depends on the critic to evaluate its proposed actions. The critic evaluates these actions using the Q-value, and its gradients are used to improve the actor's policy. The critic depends on the actor to provide actions for its training, particularly when calculating the target Q-value, Together they both reinforce each other's learning. Both actor and critic utilizes separate neural networks actor network (to decide actions) and the critic network (to evaluate actions). Actor network consists of layer1:Fully connected layer with size (state dim x 400), Layer2: Fully connected layer with size (400 x 300), Layer3(output): Fully connected layer with size (300 x action dim) followed by ReLU, sigmoid and tanh activation. Critic network consists of Layer1(state

processing): Fully connected layer with size (state dim x 400), Layer 2(State+Action): Fully connected layer with size[(400+action dim)x 300] and Layer3(output): Fully connected layer with size (300 x 1), outputs the Q-value.

- **Hyper Parameters:**
  - **Actor Learning Rate:** 1×10^{-4}
  - **Critic Learning Rate:** 1×10^{-3}
  - **Discount Factor (γ):** 0.99
  - **Episodes:** 1500
  - **Soft Update Rate (τ):** 0.005
  - **Exploration Noise:** 0.8
  - **Noise Decay:** 0.92
  - **Steps per Episode:** 1000
  - **Replay Buffer Size:** $1,000,000$
  - **Batch Size:** 128
- **Action Space:**
  - **Linear Velocity** ($action[0]$): Scaled sigmoid activation is applied, defined as:

$$action[0] = 0.2 \cdot \sigma(x)$$

    where $\sigma(x)$ is the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

    This ensures the output lies in the range $[0, 0.2]$.
  - **Angular Velocity** ($action[1]$): Scaled tanh activation is applied, defined as:

$$action[1] = 0.5 \cdot \tanh(x)$$

    where $\tanh(x)$ is the hyperbolic tangent function:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

    This ensures the output lies in the range $[-0.5, 0.5]$.

  The actor network outputs both values and concatenates them to represent the robot's linear and angular velocities.
- **Replay Buffer:** The replay buffer stabilizes training and improves learning efficiency by:
  - Storing transitions from interactions with the environment in the form of tuple. Allowing the agent to learn from a diverse set of experiences by sampling random mini-batches, breaking correlations in sequential data.
  - During each interaction with the environment, the agent's experience is stored in the replay buffer. When the buffer reaches its maximum capacity, the oldest experiences are discarded to make room for new ones. At each training step, a random mini-batch of experiences is sampled from the buffer. These batches are used to compute loss functions and update the agent's network.
- **Soft Target Updates**: Soft target updates refer to a method of updating target network parameters gradually by blending the parameters of the current network with those of the target network.
  - The purpose of soft target updates is to ensure stability during learning. abrupt changes in the target network can lead to significant fluctuations in target Q-value estimates, causing divergent training. By updating the target network slowly and smoothly,the agent receives more consistent targets for computing the temporal difference (TD) error, which stabilizes learning.
  - At the start of training, the target network is initialized with the same weights as the main network. During each training step, after updating the main network using the replay buffer, the target network's weights are updated using the soft update rule. This ensures that the target network evolves slowly over time, closely following the main network but with a lag that helps mitigate large oscillations.
  - The replay buffer helps the agent learn efficiently by providing diverse and de-correlated experiences for training, while soft target updates stabilize the learning process by ensuring smooth and consistent updates to the target networks. Together, these mechanisms play a vital role in of DDPG's stability and efficiency.
- **Reward Function**
  - The reward function enhances efficient and safe navigation while penalizing collisions:

$$r(s_t, a_t) = \begin{cases} +400 & \text{if goal reached,} \\ -400 & \text{if collision,} \\ +1 & \text{if closer to goal,} \\ -1 & \text{if moving away.} \end{cases}$$

  - Additional heading alignment reward:

$$r_{\text{heading}} = \max\left(1 - \frac{\text{Heading error}}{\pi}, -1\right),$$

    where the Heading error is the angular deviation between the robot's orientation and the target direction.
- **Implementation:**
  - **General working:** In the DDPG algorithm, the agent starts in an initial state within the environment, with the goal position and state variables initialized. The actor network predicts a continuous action based on the current state, which is augmented with exploration noise to encourage discovery of better policies. This action is executed in the environment, leading to a new state, a reward signal, and a flag indicating whether the episode has ended (e.g., reaching the goal or colliding with an obstacle). The resulting transition, comprising the current state, action, reward, next state, and termination flag, is stored in a replay buffer, which holds past experiences for training. When sufficient samples are available, the algorithm samples a batch from the replay buffer to train the networks. The critic network is updated by minimizing the error between its predicted Q-values and target Q-values, computed using the target networks [9]. The actor network is updated to maximize the Q-values predicted by the critic, encouraging actions that yield higher cumulative rewards. Soft target updates incrementally adjust the

target networks toward the main networks, ensuring stability during training. The episode ends if the agent reaches the goal, receiving a large reward, or collides with an obstacle, incurring a significant penalty. After each episode, the environment resets, and metrics such as total rewards and success rates are logged to monitor progress. This cycle repeats, enabling the agent to gradually learn an optimal policy through continuous interaction and feedback.

- **Initialization:** Utilizes a deque structure with a fixed maximum size (`BUFFER_SIZE`) to store transitions.
- **Adding Experiences:** Stores each interaction tuple (s, a, r, s', d) in the buffer during the agent's exploration.
- **Sampling:** Retrieves a random batch of `BATCH_SIZE` experiences for training to ensure decorrelated data and stable updates.
- **Training Loop Steps:**
  - Sample a mini-batch of experiences from the replay buffer.
  - Compute the target Q-value using the target networks:

$$Q_{\text{target}} = r + \gamma(1 - \text{done}) \cdot Q'(s', \pi'(s'))$$

    where Q' and π' are the target critic and actor networks.
  - Update the critic network by minimizing the Mean Squared Error (MSE) between the predicted Q-value and the target Q-value.
  - Update the actor network by maximizing the expected Q-value for the actions it predicts:

$$\nabla_\theta J = \mathbb{E}\left[\nabla_a Q(s, a) \nabla_\theta \pi(s)\right]$$

  - Perform soft updates for the target networks:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

    where τ is a small constant controlling the update rate.
- **Implementation Details**
  - **Critic Update:** The critic computes the Q-value for each state-action pair and minimizes the error against the target Q-value, ensuring the predicted Q-value aligns with the expected rewards.
  - **Actor Update:** The actor maximizes the Q-value for the actions it predicts, adjusting its parameters to ensure optimal decisions.
  - **Target Network Updates:** Smooth updates of target networks reduce instability and prevent sudden changes in the policy.

*4) **Twin Delayed Deep Deterministic Policy Gradient (TD3)**:* Twin Delayed Deep Deterministic Policy Gradient (TD3) is a state-of-the-art algorithm for continuous control tasks, designed to address key challenges like overestimation bias and instability in Deep Reinforcement Learning (DRL) [10]. TD3 builds on the foundation of the Deep Deterministic Policy Gradient (DDPG) algorithm and introduces three major improvements: **twin Q-networks**, **delayed policy updates**, and **target policy smoothing**. The algorithm involves two critic networks (Q-networks) to estimate the value of state-action pairs, where the smaller value between the two is used to reduce overestimation errors. The actor-network, which outputs the deterministic action, is updated less frequently than the critic networks to enhance stability. Additionally, TD3 employs target policy smoothing by adding noise to the target action during updates, which makes the algorithm more robust to minor deviations and prevents policy exploitation of value function errors. The training process consists of interacting with the environment to collect experience tuples, storing them in a replay buffer, and using mini-batches from this buffer to update the networks. TD3 effectively learns efficient and stable policies for complex continuous control problems by iteratively improving the policy and value estimates through gradient-based optimization.

This algorithm has been adapted to train a TurtleBot for autonomous navigation in a confined room, focusing on reaching a target position while avoiding obstacles. The implementation details for TurtleBot Navigation are given below:

- **Reward System:** The reward system incentivizes the TurtleBot to reach its goal efficiently and penalizes undesirable behaviors, such as collisions. The following reward function was used in training the TD3 agent:

$$r(s_t, a_t) = \begin{cases} r_{\text{collision}} & \text{if } \Delta d < c_o, \\ r_{\text{arrive}} & \text{if } d_t < c_g, \\ w_d \cdot r_{\text{distance}} & \text{otherwise.} \end{cases}$$

  where, $r_{collision} = -100$ and $r_{arrive} = 500$
  This reward function has the following features :
  - *Collision Penalty*:
    - If the robot's distance to the nearest obstacle, denoted as Δd, falls below a critical threshold c_o, the robot is penalized with $r_{\text{collision}}$.
    - This term heavily discourages the robot from colliding with obstacles, ensuring safety during navigation.
  - *Target Arrival Reward*:
    - If the robot's current distance to the goal, d_t, becomes less than a specified goal threshold c_g, it receives a positive reward r_{arrive}.
    - This term incentivizes the robot to successfully navigate and reach its designated target position.
  - *Continuous Guidance Reward*:
    - For all other scenarios, the reward is computed as the weighted value of the *Distance Component* ($w_d \cdot r_{\text{distance}}$). This term rewards the robot for moving closer to the target. It is proportional to the improvement in the relative distance to the goal (e.g., the difference between the previous and current distances). In our setup, we have kept the

value of w_d as 500, with which we could attain good results.

- **Action Space:** The TurtleBot's action space consists of two continuous actions:
  - *Linear Velocity (v_x)*: Ranges from -0.5 to 0.5 meters per second.
  - *Angular Velocity (ω_z)*: Ranges from -0.5 to 0.5 radians per second.

  The actor network outputs values in the range $[-1, 1]$, which are scaled and clamped to match the required action bounds:

$$v_x = 0.5 \times \text{output}, \quad \omega_z = 0.5 \times \text{output}.$$

- **Neural Network Architecture:**
  Actor Network:
  - *Input Layer*: Takes the 16-dimensional state vector as input.
  - *Hidden Layers*: Two fully connected layers with 400 and 300 neurons, respectively, using ReLU activation functions.
  - *Output Layer*: A fully connected layer with 2 outputs (one for each action), followed by a $\tanh$ activation function to constrain the outputs to $[-1, 1]$.
  - *Scaling*: The output is scaled to the action ranges $[-0.5, 0.5]$.

  Critic Network:
  - *Input Layer*: Takes a concatenated vector of the 16-dimensional state and 2-dimensional action as input.
  - *Hidden Layers*: Two fully connected layers with 400 and 300 neurons, respectively, using ReLU activation functions.
  - *Output Layer*: A single neuron outputting the scalar Q-value.

  Two critic networks (Q_1 and Q_2) are used to estimate Q-values, and the minimum of the two is selected to reduce overestimation bias.
- **Training Process:**
  Critic Update:

$$\mathcal{L}_{\text{critic}} = \text{MSE}(Q(s, a), Q_{\text{target}})$$

  The target Q-value is computed using the minimum of the two critics:

$$Q_{\text{target}} = r + \gamma(1 - \text{done}) \cdot \min(Q_1'(s', a'), Q_2'(s', a')).$$

  Actor Update: The actor is updated to maximize the expected Q-value:

$$\mathcal{L}_{\text{actor}} = -\mathbb{E}[Q_1(s, \pi(s))].$$

  Target Policy Smoothing: Gaussian noise is added to the target actions during training:

$$a' = \pi'(s') + \mathcal{N}(0, \sigma).$$

  Actions are clamped to ensure they remain within bounds.

  Target Network Updates: Soft updates are performed to stabilize learning:

$$\theta_{\text{target}} = \tau\theta + (1 - \tau)\theta_{\text{target}}.$$

- **Advantages of the Implementation:**
  - *Efficient Data Processing*: Streamlined LiDAR data reduces state dimensionality while prioritizing relevant information for safe navigation.
  - *Custom Reward System*: Encourages efficient and goal-oriented behavior while penalizing collisions.
  - *Robust Learning*: TD3's enhancements (dual critics, target smoothing, and delayed updates) ensure stable and reliable learning for continuous control tasks.
  - *Scalable Framework*: Modular implementation allows easy integration of additional sensors or features if needed.
- **Shortcomings:** While the TD3 algorithm is robust and effective, the following limitations arise in the context of this TurtleBot implementation:
  - *Sample Inefficiency:* TD3 requires a large number of interactions with the environment to learn effectively, which can be time-consuming in real-world robotics or high-fidelity simulators like Gazebo.
  - *Sparse Rewards:* The reward system may lead to sparse feedback in situations where the TurtleBot moves but does not significantly reduce the distance to the target. This can slow down learning.
  - *Exploration Challenges:* Gaussian noise may not always suffice to promote exploration in highly complex or cluttered environments, potentially leading to local optima.
  - *Action Smoothness:* While TD3 ensures bounded actions, noise addition may lead to abrupt changes in linear or angular velocity. Such jerky movements could destabilize the robot in certain scenarios.
  - *Computational Overhead:* Training neural networks (actor and critics) frequently and maintaining dual critics doubles the computational requirements compared to simpler algorithms.
- **Hyperparameters:**
  - **Learning Rate:** 3×10^{-4}
  - **Discount Factor (γ):** 0.99
  - **Soft Update Rate (τ):** 0.005
  - **Policy Noise:** 0.2
  - **Noise Clipping:** 0.5
  - **Policy Update Frequency:** 2
  - **Replay Buffer Size:** $1,000,000$
  - **Batch Size:** 256
  - **Episodes:** 4000

Overall, the TD3 algorithm was effectively customized for TurtleBot navigation using a carefully designed state representation, LiDAR preprocessing, reward system, and action space. This implementation ensures efficient training and robust navigation, enabling the TurtleBot to navigate complex

environments and reach its targets safely while avoiding obstacles.

IV. SIMULATION

The training of our model was conducted in virtual environments, using the Robot Operating System (ROS2) [11] combined with the Gazebo [12] simulator. These platforms provided a realistic and customizable setting for the experiments.

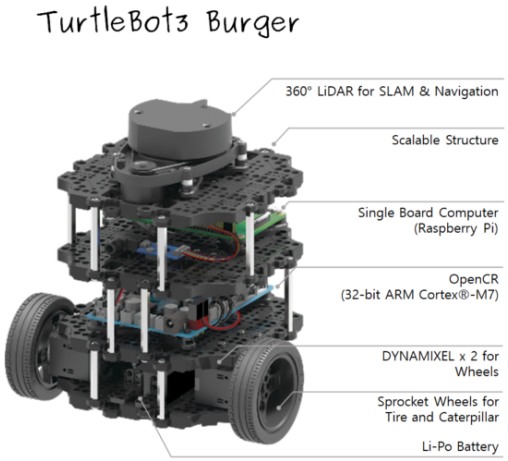

Fig. 4. The structure of Turtlebot3 robot

As shown in figure 1, We conducted our experiments in a complex environment. Environment was simulated within a 5 x 5 square meter area, enclosed by walls. The complex environment was additionally populated with obstacles strategically placed to challenge the navigation capabilities of our robot. Throughout the experiments, we utilized the Turtlebot as the robotic platform for these trials (Figure 4).

In Figure 5, the agent interacts with the Gazebo environment. Following initialization, the agent selects an action a to interact with the environment. As the agent progresses, it checks for collisions or reaching the maximum time step, indicated by the Boolean flag `done`. If `done` is true, the environment resets, providing the current reward r, the next state s', and the `done` flag.

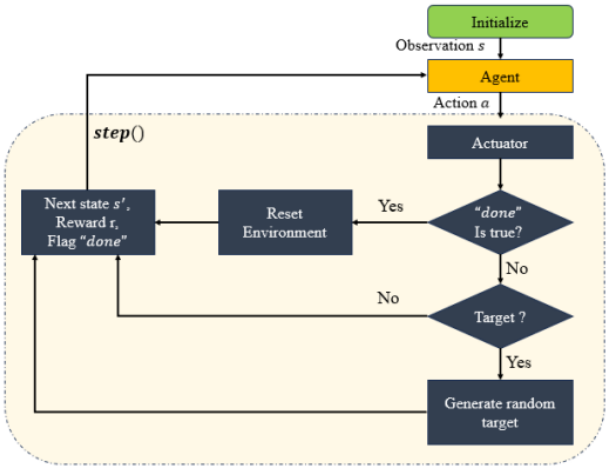

Fig. 5. The flowchart of partial DRL environment

If `done` is false, and the agent has reached the target position, another target position is generated, returning $(r, s', done)$. If the target position has not been reached, it returns $(r, s', done)$ directly. This process continues until the maximum time step is reached, with the collected data $(s, a, r, s', done)$ utilized for DRL algorithm training.

For each training episode, the target's position was randomized within the environment, ensuring it was placed away from obstacles to prevent immediate collisions. This approach of random initialization was vital for training the model to adapt to a wide range of navigation scenarios, enhancing its ability to operate effectively in varied environments.

The training for all the DRL algorithms was conducted on our Laptops equipped with NVIDIA GPUs [13] to expedite computations, particularly for training the neural networks. The implementation of the neural networks leveraged the **PyTorch** [14] library, known for its flexibility and efficiency in deep learning applications. PyTorch's dynamic computational graph facilitated efficient experimentation and debugging throughout the development process.

LiDAR Data Preprocessing: To enhance navigation in complex environments, raw LiDAR sensor data is preprocessed to condense information and prioritize computational efficiency. This optimizes decision-making for the navigation task. The LiDAR data processing approach is given below:

- *Input Data*: The LiDAR sensor provides 360 distance measurements of the robot's surroundings.
- *Batching*: These measurements are divided into 10 batches, with each batch containing 3 consecutive points.
- *Minimum Distance Selection*: Within each batch, the minimum distance is selected, representing the closest obstacle in the robot's field of view.
- *Output*: This process results in 10 streamlined observations that summarize the most relevant information for navigation.

This preprocessing reduces the data dimensionality, prioritizes actionable insights (i.e., nearest obstacles), and minimizes computational overhead. The streamlined LiDAR data improves the efficiency of decision-making by focusing on relevant environmental features critical for safe navigation.

V. RESULTS

*1) **Deep Q-Learning (DQN)**:* The performance of the robot trained using Deep Q-Learning (DQN) was evaluated in static environments to assess its learned behavior under varying conditions. Two scenarios are described below:

- **Reward Improvement:** The figure 6 below shows the evolution of the moving average reward (over 100 episodes) during training. The steady increase in rewards highlights the effectiveness of DDQN in improving policy

performance. The highest reward achieved was approximately 1610 at 1000 episodes, showcasing successful convergence of the training process.

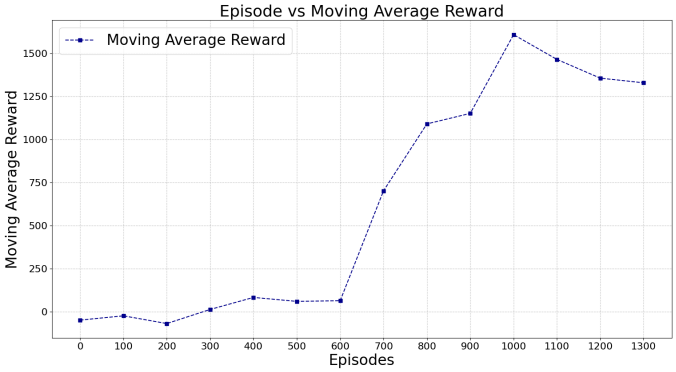

Fig. 6. Reward VS Iterations (DDQN)

- **Path Planning Example:** The figure 7 demonstrates a specific test case where the robot starts at $(-2, -2)$ and navigates to the goal at $(1.8, 0)$. Unlike the PPO planner, which follows a Euclidean trajectory, the DDQN planner follows a more grid-like trajectory, resembling a Manhattan path. This behavior indicates that DDQN prioritizes discrete directional transitions, which is effective in environments with structured obstacles.

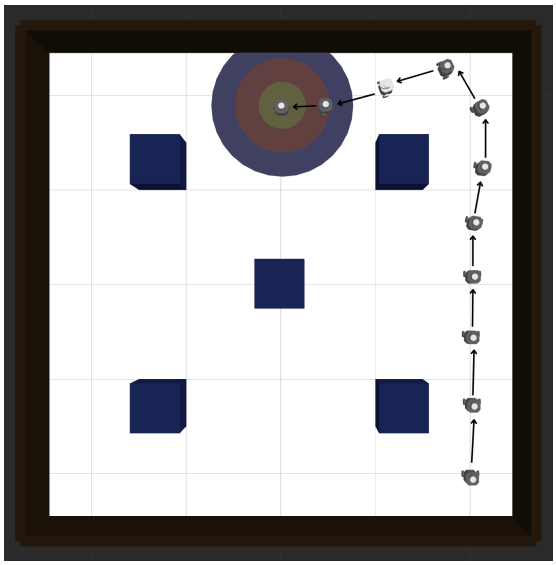

Fig. 7. Example of Autonomous Robot Navigation using DDQN

- **Accuracy Assessment:** The DDQN agent was tested over 100 episodes with random goal placements, achieving a success rate of **85%**, a significant improvement compared to the 50% accuracy achieved with the standard DQN. This demonstrates the effectiveness of DDQN in mitigating overestimation bias and improving decision-making in complex navigation tasks.
- **Discussion:** The high success rate and the convergence of rewards emphasize the strength of DDQN in handling challenging navigation scenarios. The observed trajectory pattern suggests that DDQN may favor structured planning in grid-like spaces, which could be advantageous for certain environments. While the performance surpasses DQN, future work can explore enhancements to the reward structure and adapt the algorithm for multi-agent or dynamic obstacle settings.

*2) **Proximal Policy Optimization (PPO)**:* The results obtained from the Proximal Policy Optimization (PPO) algorithm demonstrate its efficiency in training the TurtleBot to navigate through a cluttered environment while avoiding obstacles and reaching the goal.

- **Reward Improvement:** The figure 8 below illustrates the evolution of cumulative rewards over training iterations. The steady increase in rewards highlights the agent's improved performance as it learns to optimize its policy. The eventual plateau suggests that the agent converged to an optimal behavior for the given environment.

Fig. 8. Reward VS Iterations (PPO)

- **Path Planning Example:** The figure 9 showcases a specific test case where the robot is initially spawned at $(-2, -2)$ and successfully navigates to the goal located at $(1.8, 0)$ while avoiding obstacles. This case visually demonstrates the effectiveness of PPO in generating safe and efficient trajectories in a dynamic environment.
- **Accuracy Assessment:** To evaluate the robustness of the PPO algorithm, the agent was tested over 100 episodes with random goal placements. The agent achieved a success rate of **88%**, demonstrating its ability to generalize across different scenarios and efficiently navigate to its target.
- **Discussion:** The high success rate and the observed convergence of rewards highlight the strength of PPO in handling complex navigation tasks. However, the performance could vary depending on the distribution of obstacles, the initialization of weights, and the reward function parameters. Future improvements may focus on adapting the algorithm for environments with dynamic obstacles or multi-agent settings.

*3) **DDPG**:* The results obtained from Deep Deterministic Policy Gradient (DDPG) algorithm shows its efficiency in

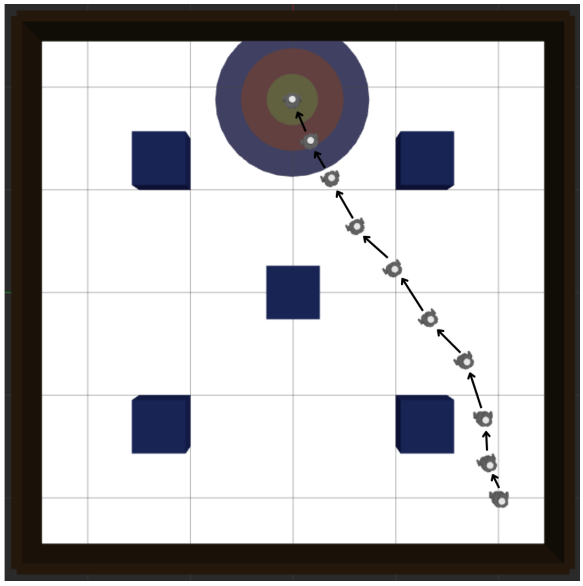

Fig. 9. Example of Autonomous Robot Navigation using PPO

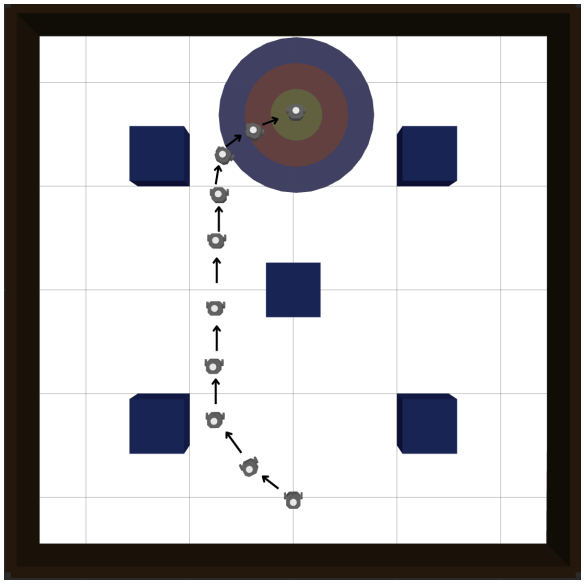

Fig. 11. Example of turtlebot3 Navigation using DDPG

training the TurtleBot to navigate through a environment to reach the goal while avoiding obstacles. DDPG was successfully able to learn a policy that helped the robot to navigate toward the goal.

- **Reward Improvement:** The figure 10 shows the evolution of rewards over episode. The steady increase in rewards highlights the agent's improved performance as it learns to optimize its policy. And the steady graph at the end indicates that the agent performs the optimal actions for the given environment.

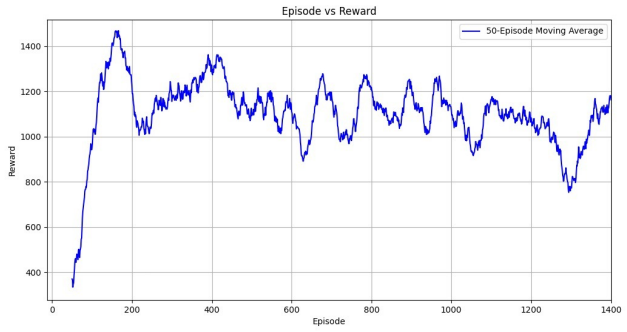

Fig. 10. Reward VS Episode

- **Path Planning Example:** The figure 11 shows a specific test case where the robot is initially spawned at $(-2, 0)$ and successfully navigates to the goal located at $(1.8, 0)$ while avoiding obstacles. This case visually demonstrates the effectiveness of DDPG in generating safe and efficient trajectories in the given custom environment.
- **Accuracy Assessment:** To evaluate the Performance and efficiency of the DDPG algorithm, the agent was tested over 100 episodes with random goal placements. The agent achieved a success rate of **71%**, demonstrating its ability to generalize the behavior across different scenarios and efficiently navigate to the goal.
- **Discussion:** The Performance metric - success rate shows the potential of DDPG in handling navigation tasks under a given environment. However, the performance could be affected depending on the distribution of obstacles, the environment, the initialization of weights, and the reward function. Future improvements may focus on adapting this algorithm for environments with dynamic obstacles.

*4) **TD3**:* We trained the agent for the given simulation configurations and hyperparameters and observed its behavior. At the end of the training cycle, we observed that the TD3 algorithm was successfully able to learn a policy that helped the Turtlebot navigate toward the goal. The following observations and inferences were made after training and testing the model:

- **Reward Improvement:** Figure 12 shown below demonstrates the effectiveness of the TD3 algorithm in learning the desired policy. As we can see from the graph, over the span of the episodes, the agent is able to attain a high cumulative reward, which then eventually stabilizes, which denotes that the agent has converged at the optimal cumulative reward point.
- **Path Planning Example:** For testing the trained model, we utilized the same environment, with the Turtlebot starting the navigation task at coordinates $(-2, -2)$, and moving towards a goal position of $(1.8, 0)$. The Turtlebot was successfully able to navigate towards this goal and avoid the obstacles. The trajectory behavior can be observed in Figure 13. As seen in the figure, the Turtlebot adapts a trajectory directed towards the goal and accordingly avoids obstacles to reach the goal safely.
- **Accuracy Assessment:** To evaluate the overall accuracy

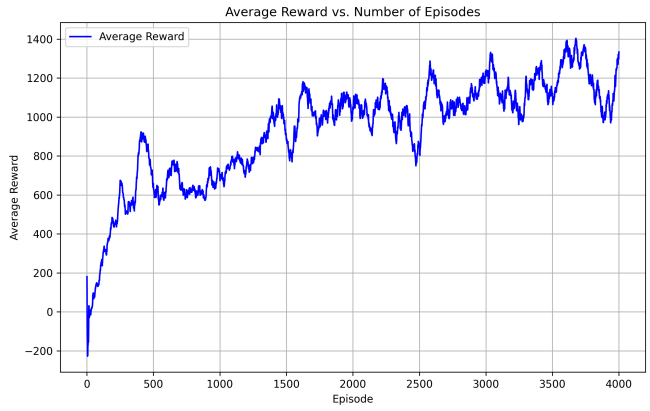

Fig. 12. Average Reward vs Number of Episodes

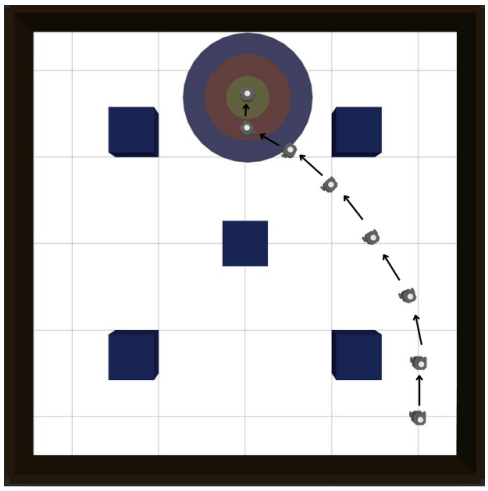

Fig. 13. Example of Autonomous Robot Navigation using TD3

of the algorithm, the agent was tested over 100 episodes with random goal placement. The agent achieved a success rate of **75%**, which shows that the agent was effectively able to learn how to navigate in the environment and reach the goal in various scenarios, hence learning a generalized behavior.

- **Discussion:** The high success rate and consistent reward convergence underscore TD3's effectiveness in tackling complex navigation challenges. Nevertheless, its performance may depend on factors such as obstacle distribution, weight initialization, and reward function parameters. Future enhancements could aim to refine the algorithm for environments with dynamic obstacles or multi-agent scenarios.

VI. CONCLUSION

In this project, we implemented and evaluated four reinforcement learning algorithms—**Proximal Policy Optimization (PPO)**, **Deep Deterministic Policy Gradient (DDPG)**, **Twin Delayed Deep Deterministic Policy Gradient (TD3)**, and **Deep Q-Network (DQN)**—for autonomous navigation in a TurtleBot environment.

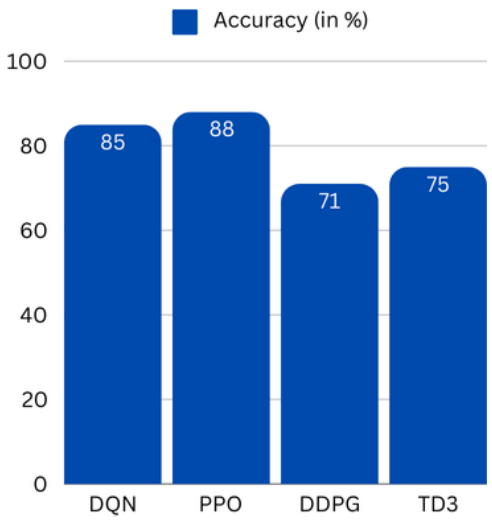

Fig. 14. Accuracy of DRL Algorithms

Each algorithm was assessed based on its ability to navigate the robot to randomly spawned goal locations while avoiding obstacles. The results were measured over 100 test episodes, and the accuracy of each algorithm was recorded as follows:

- **Proximal Policy Optimization (PPO):** 88%
- **Deep Q-Network (DQN):** 84%
- **Deep Deterministic Policy Gradient (DDPG):** 71%
- **Twin Delayed Deep Deterministic Policy Gradient (TD3):** 75%

Among the tested algorithms, PPO achieved the highest accuracy (88%), showcasing its robustness and efficiency in navigating complex environments with continuous action spaces. DQN followed closely with an accuracy of 84%, demonstrating its effectiveness in discrete action spaces. However, DDPG and TD3 showed relatively lower accuracies, which can be attributed to their sensitivity to hyperparameter tuning and challenges in managing exploration in high-dimensional spaces.

The results highlight the trade-offs between policy-based and value-based methods for robotic navigation tasks. While PPO and DQN performed well overall, they have different strengths—PPO excels in continuous control problems, while DQN is suitable for tasks with discrete actions. On the other hand, DDPG and TD3, despite being designed for continuous spaces, require careful tuning and can underperform in environments with significant noise or variability.

These findings suggest that the choice of algorithm depends on the specific requirements of the task, such as the nature of the action space, the complexity of the environment, and the computational resources available.

REFERENCES

[1] S. Luo and L. Schomaker, "Reinforcement learning in robotic motion planning by combined experience-based planning and self-imitation learning," 2023.

[2] T. Bhuiyan, L. Kästner, Y. Hu, B. Kutschank, and J. Lambrecht, "Deep-reinforcement-learning-based path planning for industrial robots using distance sensors as observation," 2023.

[3] H. Taheri, S. R. Hosseini, and M. A. Nekoui, "Deep reinforcement learning with enhanced ppo for safe mobile robot navigation," 2024.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[5] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[7] H. Tan, "Reinforcement learning with deep deterministic policy gradient," in *2021 International Conference on Artificial Intelligence, Big Data and Algorithms (CAIBDA)*, pp. 82–85, 2021.

[8] T. Tiong, I. Saad, K. T. K. Teo, and H. b. Lago, "Deep reinforcement learning with robust deep deterministic policy gradient," in *2020 2nd International Conference on Electrical, Control and Instrumentation Engineering (ICECIE)*, pp. 1–5, 2020.

[9] Y. Dong and X. Zou, "Mobile robot path planning based on improved ddpg reinforcement learning algorithm," in *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 52–56, 2020.

[10] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018.

[11] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[12] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, (Sendai, Japan), pp. 2149–2154, Sep 2004.

[13] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," in *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, (New York, NY, USA), Association for Computing Machinery, 2008.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019.

APPENDIX

1) Path planning video using **DDQN model**. DDQN.mp4
2) Path planning video using **PPO model**. PPO.mp4
3) Path planning video using **DDPG model**. DDPG.mp4
4) Path planning video using **TD3 model**. TD3.mp4
5) Zip file of code can be found here. Code