Robot Navigation using Deep Reinforcement Learning

Ahmed Mohsen Ali
Robotics and Computer Vision
Innopolis University
Innopolis, Russia
a.ali@innopolis.university

Sohaila Ahmed Hussein
Robotics and Computer Vision
Innopolis University
Innopolis, Russia
s.hussein@innopolis.university

Felix Sihitshuwam Dalang
Robotics and Computer Vision
Innopolis University
Innopolis, Russia
f.dalang@innopolis.university

Abstract—**Mobile robots have a critical role in many industrial fields and perform their tasks with a higher precision and safety considerations. However, mobile robot navigation is still one of the challenging tasks, since it attempts to move a robot from an initial pose to a final pose in the shortest possible time without hitting any obstacles. Recent techniques use path planning and computer algorithms to perform this task and to navigate the robot around an environment whilst avoiding obstacles. A limitation to most of these algorithms is the need to know the full kinematic structure and motion model of the robot, which is challenging to pursue. As a result, using reinforcement learning, which is a sub-field of artificial intelligence, in which an agent learns by interaction with environment, serves as a possible solution for robot navigation. In this project, we use deep reinforcement learning algorithms to solve the robot navigation problem and we build an environment from scratch and specify its related properties. The agent (a mobile robot) learns how to safely reach a target location. Our proposed solution utilizes discrete and continuous action spaces to navigate the robot in a continuous world. We utilize Deep Q-Network (DQN), Deep Deterministic Policy Gradient (DDPG) and Temporal Difference 3 (TD3) to this solve this problem. We outline the parameters used and finally demonstrate, by graphs and analysis, the performance of our proposed solution.**

Index Terms—**Mobile Robots, Robot Navigation, Reinforcement Learning, Deep Q Network, Deep Deterministic Policy Gradient, Temporal Difference 3.**

I. Introduction

Mobile robots refer to autonomous vehicles which are able to navigate an environment on their own. The ability of a robot to navigate around solves one of the major limitations of industrial robots, i.e, the problem of locomotion [1]. Mobile robots set out to solve this problem by navigating a robot from a one point to another. An obvious matter that immediately comes up is how to navigate and do that safely while avoiding obstacles.

Several techniques exist in the industry to solving this problem. Generally, robot navigation problems involves minimizing the difference between the current robot pose and the desired pose. The techniques mostly utilize the knowledge of the robot's kinematics to generate control laws and path planning algorithms for navigation. Computer vision techniques are also inculcated into this task for improving accuracy and obstacle avoidance.

Reinforcement Learning refers to a class of algorithms which are used to train an agent to act intelligently (make optimal choices and decisions), by utilizing experience and examples, rather than a labeled set of data where the agent learns to classify good and bad actions [2]. In reinforcement learning, actions are not classified but rather an agent learns proper actions by interacting with the environment, and noting how its action affects long term rewards.

Applied to the problem of robot navigation, reinforcement learning offers us a way to teach a robot how to get to a specified location from an initial position. This offers a great benefit that we don't need to know the internal kinematics of the robot (i.e, wheel radius, axle length, etc). The robot learns the correct linear and angular velocities that are needed to get it to the desired state.

In this research, we set out to perform robot navigation by reinforcement learning. We first build the environment and then use both discrete and continuous actions to attempt solving this problem.

This report is consequently structured as follows. Section II presents briefly a literature review of using Reinforcement Learning (RL) approaches in robot navigation. Section III provides the problem formulation along with environment characteristics. Section IV explains the the RL models used in this paper. Section V shows and discusses the results and section VI draws the paper conclusions.

II. Related Work

Several approaches and methods of RL have been implemented in the literature to solve robot navigation problem. There are multiple factors that determine the problem solutions such as sensors used, 2D camera or 3D laser scanner. Also, the map type, whether known or unknown, and the initial robot pose and goal, whether fixed or varied, affect the overall complexity of RL model. Zhang *et al* [3] proposed a Deep Q learning (DQN) to obtain the optimal value function whereby robot can get the actions required to each the goal. In addition, they used successor-feature-based algorithms to facilitate knowledge transfer from solved problems so robot can utilize these information to navigate in new environment setups. Although the robot could navigate successfully in both known and unknown environments with only RGB camera , the action space has to be discrete, which is a limitation to most robot controllers.

Mitrowski *et al* [4] proposed to use A3C for navigation, along with depth estimation and loop closure models to acquire localization skills for the robot. Their model managed to reach the target in complex mazes with different start and end points. However, A3C requires parallel training which is not available in some robotic simulators and no experimental implementation was provided to validate the approach in real time. Tai *et al* [5] suggested using Adaptive Deep Deterministic Policy Gradient (ADDPG) to navigate in unknown environment. They used a different thread for sampling, thus the asynchronous part, generating more samples and maximizing the rewards much faster. The model worked effectively to generate continuous action space as long as robot motion is restricted in the area scanned by the 2D laser scanner. This limitation indicates the robot can not detect obstacles of certain heights or shapes.

Surmann *et al* [6] used an Asynchronous Advantage Actor-Critic (GA3C) with sensor fusion between 2D laser and depth camera to navigate in unknown environments. They created their own simulation to match the required high speed of training. Furthermore, Cimurs *(*et al) [7] divide the navigation into local and global modules. Global modules generates points that robot will follow and local module obtains the motion policy using TD3 network.

One can conclude that there is a versatility of RL models used for robot navigation. In this paper, three of the aforementioned models are implemented: DDPG and TD3 for continuous action space and DQN for discrete case.

III. Problem Formulation

A. Robot Model

We have a non-holonomic wheeled mobile robot, with a specified dimension, which is able to move in 2D space based on control inputs specified as linear and angular velocities, v and ω respectively. The radius (or diameter) of the robot in particular is of importance to our subsequent formulations. The state of the robot is specified by its pose which is given by x, y, θ. That is, its location in the world and the orientation it makes with reference to the x-axis. This orientation (and indeed all the orientation values we used) is normalized to values between $-\pi$ to π (rads).

B. Environment

Given a map of an area, with locations specified by x, y, we desire our robot to learn how to navigate from any part of the environment to another part. The map we use for this problem is a walled area, with obstacles within the wall as shown in Fig. 5. To begin with, we use the same map without obstacles in it to train the robot.

We convert the coloured map into a 2D gray-scale version. The robot only interacts directly with this gray-scale map. The white regions of the map indicates movable areas while dark regions indicates the walls and obstacles.

Hence, more concretely, our problem involves the robot moving from it's current pose (x, y, θ) to a goal location (x_g, y_g), while avoiding obstacles along its path.

In our implementation, we will randomly initialize the robot pose and the target location to any movable area in the map. The initial robot pose and the target location are also constrained never to coincide. We then use these states to train the robot. An episode ends when the robot hits an obstacle, gets to the target location, or reaches 800 steps.

C. Control Inputs

Control Inputs are specified by linear and angular velocities (v, w). These control inputs update the *state* of the robot according to the equations below which are obtained from numeric trapezoidal integration. At time (t), the next states are given by

$$x_{t+1} = x_t + v_t * t_s * cos(\theta_t + \frac{\omega_t}{2} * t_s) \tag{1}$$

$$y_{t+1} = y_t + v_t * t_s * sin(\theta_t + \frac{\omega_t}{2} * t_s) \tag{2}$$

$$\theta_{t+1} = \theta_t + \omega_t * t_s \tag{3}$$

D. Reinforcement Learning Reward

Since our task involves moving to a target location, we formulate the reward system given to the robot as an inverse function of its current location and target location. Getting to the target location gives a reward of 1 whilst, away from the target location yields rewards of progressively lesser magnitude. The reward function is evaluated using the equation below.

$$distance = \sqrt{(x_t - x_g)^2 + (y_t - y_g)^2} \tag{4}$$

$$reward_{dist} = \frac{1}{(1 + distance)} \tag{5}$$

Additionally, walls and obstacles, are given a reward of -10, a negative reward with a *relatively* high magnitude to indicate a high penalty.

And finally, we factored in the orientation of the robot into the reward. As stated previously in III-C, the orientation of the robot at each given time step (θ) is calculated. We now calculate the angle the target makes with the horizontal axis (β) and subtract this angle from the robot orientation to get the orientation error (α). This error is normalized and factored into the reward according to the formulation below.

Given robot pose x, y, θ, and target location (x_g, y_g),

$$\beta = atan2(y_g - y, x_g - x) \tag{6}$$

$$\alpha = |\beta - \theta| \tag{7}$$

$$\alpha = \alpha/\pi \tag{8}$$

$$reward_{orien} = (1 - \alpha) \tag{9}$$

With this system, a robot gets *0* orientation reward when it is aligned directly opposite the target, and *1* when it is aligned directly facing the target. Whilst the reward ranges between 0 and 1 when it is facing elsewhere, according to the magnitude of the orientation error.

This orientation reward is added to the distance reward calculated above to give the reward for a step.

We will also try penalizing the robot for taking too much steps, by subtracting some discount $(0 - 0.1)$ from the final reward.

IV. Methodology

A. Deep Q-Network (DQN)

The DQN algorithm [8] can be thought of as a continuation to the Q-learning algorithm. In which formerly the state-action relationship is defined as a Q-table for storing the data for each state to guarantee reaching an optimal action-value function $Q(s, a)$. Although, it becomes undeniably handy to easily choose the best action at a certain state, it can only work with specific types of environments in which the state-space is finitely small. Hence, a solution was developed to approximate the Q-values in a large continuous state space, in which deep neural networks are incorporated as a tool to approximate the targeted function.

In our model, the Deep Q-Network consists of two convolution layers with *ReLU* activation function and a linearly activated fully connected layer as in Fig. 3. The initial layer has the same dimension as the state-space, while the last layer has the same dimension as the action space. Consequently, the output of the final layer is obtained through the neural network approximation for the distinct Q-values for each possible action at the current input state.

The action is chosen based on the *epsilon greedy* algorithm, where a trade off between exploration and exploitation is achieved. Exploration is carried out epsilon (ϵ) times, while exploitation is carried out $(1 - \epsilon)$ times. It can be expressed through the following mathematical representation:

$$a = \left\{ \begin{array}{ll} \underset{a \in A}{\arg\max} Q_t(a), & \text{if } 1 - \epsilon \\ random(Q_t(a)), & \text{if } \epsilon \end{array} \right\} \tag{10}$$

The ultimate goal is to choose an action corresponding to the highest Q-value, while guaranteeing a sufficient exploration of different states existing in the environment.

The DQN algorithm has an additional feature called the **Replayed Memory**, which is particularly a memory buffer to store all the experiences carried out by the agent. The experience at time (t) is composed of a tuple of four parameters: the current state, the current action, the reward, and the next state (observation). An experience can be represented as:

$$e_t = (s_t + a_t + r_{t+1}, s_{t+1}) \tag{11}$$

In the training phase, two main acts are done: sampling and training. Sampling is concerned about storing the tuple of experiences in *replay memory* after actions are performed. Training, on the other hand, involves the random selection of a mini batch of experience tuple from the *replay memory*. Both acts are incorporated in the pseudo-code of the algorithm represented in alg.(1):

Algorithm 1 Deep Q-Network

```
Initialize replay memory D with size N
Initialize action-value function Q with random weights
for episode ← 1 to $M$ do
    for $t \leftarrow 1$ to $T$ do
        if $\varepsilon$ then
            $a_t \leftarrow$ select random action with probability$\varepsilon$
        else if $(1 - \varepsilon)$ then
            $a_t \leftarrow \arg\max_a Q(s, a)$
        end if
        Execute action $a_t$
        Observe reward $r_t$ and next state $s_{t+1}$
        $D \leftarrow s_t, a_t, r_t, s_{t+1}$
        Sample a random mini batch from D
        Set $y_j = \left\{ \begin{array}{ll} r_j, & \text{if episode terminates} \\ r_j + \gamma \max_{a'} Q(s', a'), & \text{otherwise} \end{array} \right\}$
        Perform gradient descent on $(y_j - Q(s_j, a_j))^2$
    end for
end for
```

Fig. 1. DQN Algorithm

Thus, the network is trained after each episode through allowing the agent for the random sampling of a batch of experiences from the replayed memory to avoid over-fitting and bias. A loss function is calculated as the difference between the predicted Q-value and the Q-target, which is called the *Temporal Difference* error (TD-error). It is updated through carrying out a gradient descent optimizer. The Bellman optimality equation in eq.(12) is used to set the Q-target.

$$q_*(s, a) = E[R_{t+1} + \gamma \underset{a'}{\max} q_*(s', a')] \tag{12}$$

Hence, the final loss function can be formulated as a mean squared error: $(y_j - Q(s_j, a_j))^2$

B. Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is an off-policy approach that consists of Actor and Critic models. This algorithm was first introduced to handle continuous action space and to enhance some of DQN limitations [9]. For example, the Actor takes as an input the state s and outputs an action a, which is a discrete value instead of a normal distribution. Moreover, the Critic model is responsible for generating Q values given a pair of action and state denoted as (s, a). To make the model more stable, additional target models for Actor and Critic are also implemented which are updated less frequently than the main networks. In our Actor and Critic networks in our model, we used three dense Actor layers and five dense layer respectively. The complete architecture is presented in Fig. 3.

Algorithm 2 Deep Deterministic Policy Gradient

```
Initialize main critic Q and main actor networks $\mu$ with weights $\theta^Q, \theta^\mu$
Initialize target critic $Q_{targ}$ and actor $\mu_{targ}$ with weights $\theta^Q_{targ}, \theta^\mu_{targ}$
Initialize replay memory D with size N
for episode $\leftarrow 1$ to $M$ do
    for $t \leftarrow 1$ to $T$ do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}$ from main policy $\mu$ with noise
        Execute action $a_t$ and observe reward $r_t$ and new state $s_{t+1}$
        $D \leftarrow s_t, a_t, r_t, s_{t+1}$
        Select mini batch  b  from  D
        Update main Critic and Actor losses
        Update target network with a smaller rate:
            $\theta^\mu_{targ} \Longleftarrow \tau\theta^\mu_{targ} + (1-\tau)\theta^\mu$
            $\theta^Q_{targ} \Longleftarrow \tau\theta^Q_{targ} + (1-\tau)\theta^Q$
    end for
end for
```

Fig. 2. DDPG Algorithm

Since two distinct models are deployed, two different cost functions are needed. The cost of Actor model (μ) is to maximize the returns at each state by choosing the action that will result in a higher reward. For the Critic model (Q), the cost function is simply a TD error. For more stability, the Q value for the next state and action is calculated by target Actor (μ_{targ})and target Critic (Q_{targ}). Both loss functions are expressed in the following mathematical expressions:

$$J_Q = \frac{1}{N}\sum_{t=1}^{N}(r_t + \gamma(1-d)Q_{targ}(s_{t+1}, \mu_{targ}(s_{t+1})) - Q(s_t, \mu_t))^2 \quad (13)$$

$$J_\mu = \frac{1}{N}\sum_{t=1}^{N} Q(s_t, \mu(s_t)) \quad (14)$$

Since the model wants to maximize the policy cost function J_μ, the optimizer uses $-J_\mu$ as the cost and applies gradient decent to minimize it. In addition, the weights of both target networks are upgraded using soft update. These target network updates are expressed as follows:

$$\theta^\mu_{targ} \Longleftarrow \tau\theta^\mu_{targ} + (1-\tau)\theta^\mu \quad (15)$$

$$\theta^Q_{targ} \Longleftarrow \tau\theta^Q_{targ} + (1-\tau)\theta^Q \quad (16)$$

Where τ is a value near 1 and it indicates how much the target update is soft. The hyper-parameters for the model and and the complete algorithm are shown in table II and alg. 2 respectively.

C. *Temporal-Difference3 (TD3)*

TD3 was introduced to enhance the problems faced by DDPG [10]. For example, DDPG sometimes tend to overestimate the Q-values which leads to poor performance of the agent policy. In general, the architecture for TD3 is similar to DDPG which was explained previously. However, the main difference here is that TD3 uses two networks for the target instead of one and then chooses the smallest value of them. In addition, it updates the main actor less frequently than the main critic and utilize noisy target action so that the model becomes more robust.

In our TD3 model, the network architecture is inspired from [5] and [7]. As shown in Fig. 3, the actor model consists of three dense layer with ReLU activation functions followed by two dense layer with Sigmoid and Tanh as activation functions to generate linear and angular velocities respectively. The choice of these activation functions is to account for the different control range. The hyper-parameters are depicted in table II.

Algorithm 3 Temporal-Difference 3

```
Initialize main critic $Q$ and main actor networks $\mu$ with weights $\theta^Q, \theta^\mu$
Initialize two target critic $Q_{targ1,2}$ and actor $\mu_{targ}$ with weights $\theta^Q_{targ}, \theta^\mu_{targ}$
Initialize replay memory D with size N
for episode $\leftarrow 1$ to $M$ do
    for $t \leftarrow 1$ to $T$ do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}$ from main policy $\mu$ with noise
        Execute action $a_t$ and observe reward $r_t$ and new state $s_{t+1}$
        $D \leftarrow s_t, a_t, r_t, s_{t+1}$
        Select mini batch  b  from  D
        Select action $a_{targ} = \mu(s_t|\theta_{targ})$ for TD target
        if $TargetUpdate$ then
            Update Policy network $\mu$
            Update target networks $Q_{\theta_1}, Q_{\theta_2}, \mu_{targ}$
        end if
    end for
end for
```

Fig. 4. TD3 Algorithm

The cost function for TD3 is similar to those of DDPG as in Eq.14 and Eq.13. However, the target actions is calculated with noise and then clipped to be withing action range as follows:

$$a_{t+1}(s_{t+1}) = clip(\mu_{targ}(s_{t+1}) + clip(\epsilon, -c, c), a_{low}, a_{high}) \quad (17)$$

Where ϵ is noise generated from specified normal distribution $\mathcal{N}(0, \sigma)$ and c is noise maximum value. Moreover, the TD target is calculated by using minimum output of two critic network not one. This TD target for a single step is described in the following formula:

$$y(r_t, s_{t+1}, d) = r_t + \gamma(1-d)\min_{i=1,2} Q_{\phi_i,targ}(s_{t+1}, a_{t+1}(s_{t+1})) \quad (18)$$

V. Results and Discussion

A. *Environment*

The aim of this section is to present and discuss the outcomes of our work. First of all, we successfully managed to build a customized environment from scratch using *gym* as shown in fig.(5). The environment and robot design are inspired from the turtle-bot3[*]. In this environment, we set the initial pose, goal pose, rewards, states, actions, and control inputs. The models (DQN, DDPG, TD3) were tested in an obstacle-free world that included only wall boundaries. The reward function is used to reinforce the robot to reach the goal and put penalty on hitting the wall or exceeding the maximum number of steps.

[*]https://www.turtlebot.com/

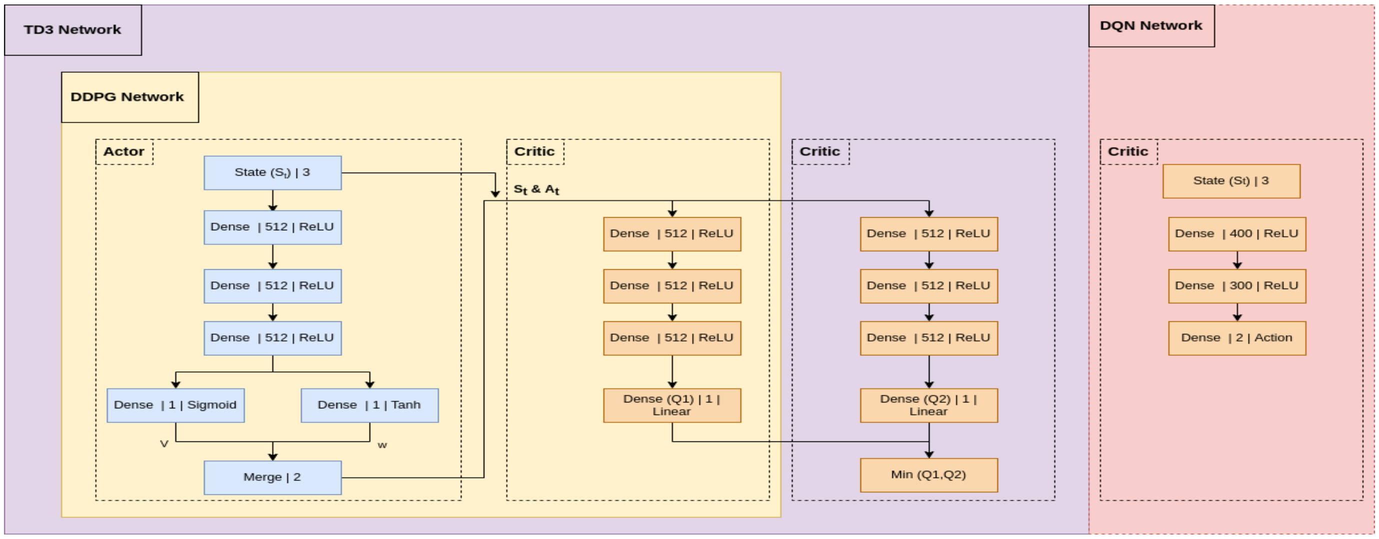


Fig. 3. The network architecture for the three models implemented in paper: TD3, DDPG, and DQN. The Actor and Critic layer have different colors for better visualization. Each layer contains the type, size, and activation function. Due to space limits, the DDPG is drawn here under TD3 network. However, each network has a separate file in the code implementation.

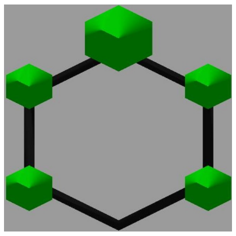
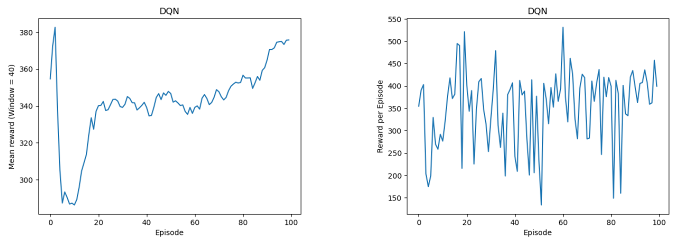


Fig. 5. Map design



(a) Average reward (100 episodes)

(b) Reward per episode (100 episodes)

Fig. 6. (a) shows the average rewards for the DQN for 100 episodes while (b) show the cumulative reward per episode (return).

It is worth noting that for fair evaluation all three models were trained for 100 episodes and maximum iteration steps of 800 in a continuous state space with dimensions (763*654*360), which corresponds to image width, image height, and orientation range respectively. A unified evaluation criteria were implemented for evaluating the models, which is calculating the cumulative reward per episode (return) and the mean average reward by means of a moving window with a size of 40 episodes to smooth the graphs. The full code implementation can be found on this GitHub repo.

B. Discrete Actions

The DQN algorithm in section IV was implemented with a discrete action space of dimension 15 in the aforementioned environment. The range of linear velocities if from 0 to 4, while the angular velocities ranged from -1 to 1. The results of this model is represented in figures 6a and 6b. In figure 6a, the average reward of 100 full episodes is plotted. It is noted that as the number of episodes increase the model converges to the optimal reward. To closely asses the behavior of the agent with the model, the reward per episode (return) is shown in figure 6b. It is perceived that as the number of episodes increase, the agent learns to avoid hitting the walls and move towards the goal.

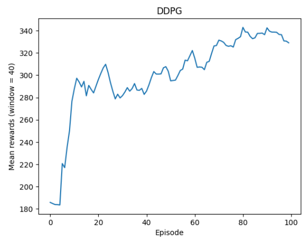
Utilizing the fusion of parameters recorded in table I, the model was able to successfully explore the environment freely. Additionally, the agent succeeded in exploiting the best actions to try to reach the goal more often.

TABLE I
DQN HYPER-PARAMETERS

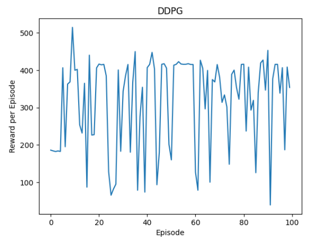
Parameter	Value
Learning rate	0.001
Gamma	0.99
Epsilon	1.0
Decay rate	0.005
Batch size	32

C. Continuous Actions

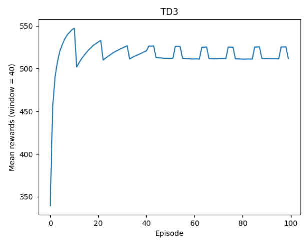
As stated before, DDPG and TD3 were applied for the continuous action space. Their average reward graphs (Fig. 7a, 7c) and reward per episode graphs (Fig. 7b, 7d) showed promising convergence to a high rewards, with TD3 reaching higher reward than DDPG. However, the robot did not reach its optimal case and the action was saturated in a sub-optimal level in both models. This performance was seen in the simulation as moving in circles around the goal but not reaching the goal itself. This phenomena occurred due to high dimension of both state and action spaces, indicating the robot needs more training time.



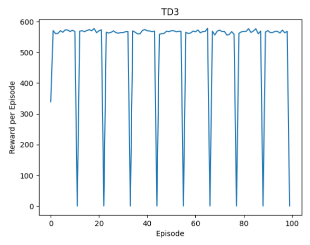
(a) Average reward (100 episodes)



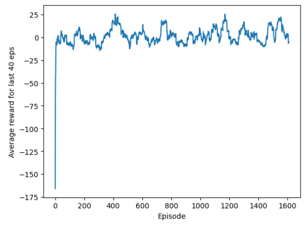
(b) Reward per episode (100 episodes)



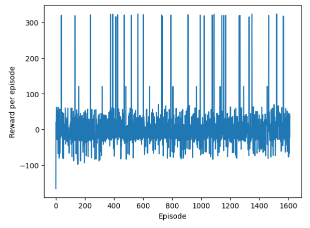
(c) Average reward (100 episodes)



(d) Reward per episode (100 episodes)



(e) Average reward (1600 episodes)



(f) Reward per episode (1600 episodes)

Fig. 7. (a), (b) shows the returns for the DDPG after 100 episodes and (c),(d) show the returns for TD3. (e),(f) show the returns for the TD3 after 1600 episodes.

Thus, multiple variations of the reward functions, action ranges, hyper-parameters and state space were applied. For example, Fig. 7e, 7f show the reward after 1600 episodes. The best combination of variables are those which are mentioned in this paper and the final hyper-parameters are presented in table II.

TABLE II
DDPG & TD3 HYPER-PARAMETERS

Parameter	**Value**
Learning rate	0.001
Gamma	0.99
Decay rate	0.005
Batch size	128
Noise std	0.2

VI. CONCLUSION

In this research, we evaluate the performance of three different deep reinforcement algorithms: DQN, DDPG, and TD3 in solving the task of robot navigation. The robot is expected to reach a fixed goal pose from a specific predefined initial position. The whole environment was created from scratch using the OpenAI gym environment to set the robot kinematics, control inputs, and rewards. Two main versions of the world were created. A world that serves a discrete action space along with a continuous state space for incorporating DQN model. Another world offers both continuous action and state spaces for implementing DDPG and TD3. The DQN model showed promising results in terms of convergence. However, the DDPG and TD3 models reach sub-optimal convergence. It is concluded that this happens due to the challenges of including high dimensional continuous state and action spaces at the same time. It is intended in the future to train the models for larger number of episodes and iteration steps to achieve better outputs.

REFERENCES

[1] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to autonomous mobile robots*. MIT press, 2011.

[2] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[3] J. Zhang, J. T. Springenberg, J. Boedecker, and W. Burgard, “Deep reinforcement learning with successor features for navigation across similar environments,” *CoRR*, vol. abs/1612.05533, 2016. [Online]. Available: http://arxiv.org/abs/1612.05533

[4] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, D. Kumaran, and R. Hadsell, “Learning to navigate in complex environments,” *CoRR*, vol. abs/1611.03673, 2016. [Online]. Available: http://arxiv.org/abs/1611.03673

[5] L. Tai, G. Paolo, and M. Liu, “Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation,” *CoRR*, vol. abs/1703.00420, 2017. [Online]. Available: http://arxiv.org/abs/1703.00420

[6] H. Surmann, C. Jestel, R. Marchel, F. Musberg, H. Elhadj, and M. Ardani, “Deep reinforcement learning for real autonomous mobile robot navigation in indoor environments,” *CoRR*, vol. abs/2005.13857, 2020. [Online]. Available: https://arxiv.org/abs/2005.13857

[7] R. Cimurs, I. H. Suh, and J. H. Lee, “Goal-driven autonomous mapping through deep reinforcement learning and planning-based navigation,” *CoRR*, vol. abs/2103.07119, 2021. [Online]. Available: https://arxiv.org/abs/2103.07119

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, “Playing atari with deep reinforcement learning,” *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

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

[10] S. Fujimoto, H. van Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” *CoRR*, vol. abs/1802.09477, 2018. [Online]. Available: http://arxiv.org/abs/1802.09477