Comparison of Deep Q-Learning, Q-Learning and SARSA Reinforced Learning for Robot Local Navigation

Hafiq Anas[1], Ong Wee Hong[2], and Owais Ahmed Malik[3]

[1] School of Digital Science, Universiti Brunei Darussalam, Jalan Tungku Link, Brunei, hafiq.anas@gmail.com
[2] School of Digital Science, Universiti Brunei Darussalam, Jalan Tungku Link, Brunei,
weehong.ong@ubd.edu.bn
[3] School of Digital Science, Universiti Brunei Darussalam, Jalan Tungku Link, Brunei,
owais.malik@ubd.edu.bn

Abstract. This paper presents a performance comparison of mobile robot obstacle avoidance between using Deep Reinforcement Learning (DRL) and two classical Reinforcement Learning (RL). For the DRL-based method, Deep Q-Learning (DQN) algorithm was used whereas for the RL-based method, Q-Learning and Sarsa algorithms were used. In our experiments, we have used the extended OpenAI Gym ToolKit to compare the performances of DQN, Q-Learning, and Sarsa algorithms in both simulated and real-world environments. Turtlebot3 Burger was used as the mobile robot hardware to evaluate the performance of the RL models in the real-world environment. The average rewards, episode steps, and rate of successful navigation were used to compare the performance of the navigation ability of the RL agents. Based on the simulated and real-world results, DQN has performed significantly better than both Q-Learning and Sarsa. It has achieved 100% success rates during the simulated and real-world tests.

Keywords: Deep Reinforcement Learning · Mobile Robot Navigation · Obstacle Avoidance.

1 Introduction

Obstacle avoidance is one of the most important robot technologies in the field of mobile robot navigation. In any navigation task, a mobile robot should be able to reliably traverse from one position to another with little to no obstacle collisions. Robot navigation is mainly divided into two parts which are global pathfinding and local obstacle avoidance [1]. This paper mainly focuses on the local navigation aspect of mobile robot navigation using Reinforcement Learning (RL) techniques.

Local navigation techniques make use of information from the sensors of a mobile robot to perform obstacle avoidance and these include fuzzy logic methods,

artificial potential field methods, and genetic algorithms [2, 3, 4]. The aforementioned techniques are classical methods that have been proven to perform well to solve the local navigation problem. However, these methods are not robust enough to adapt to changes in the environment. Additionally, programmers have to carefully design a good control strategy in these classical methods to avoid collision in unpredictable situations. Ideally, a mobile robot should be smart enough to adapt to changes, making self-learning mobile robots a prominent research area. In the real world, the environments can be complex and unpredictable, which brings the need for a mobile robot to be capable of self-learning to be able to adapt to different environments for collision-free navigation. Various approaches were explored and Reinforcement Learning (RL) techniques seem to be the ideal approach to enable self-learning. The DQN model was deployed in the physical environment to validate its performance in the real world.

This paper presents a performance comparison between the Deep Q-Learning (DQN) algorithm and two classical RL algorithms of Sarsa and Q-Learning that were used as a self-learning local obstacle avoidance technique. The learning process was carried out using Turtlebot3 burger mobile robot in a simulated environment using the Gazebo software and Robot Operating System (ROS) framework. The evaluation of the RL algorithms was done by comparing the average rewards, steps, and success rates. Based on the training and testing results, the DQN algorithm performed the best which shows that the addition of Deep Learning has the potential to improve the navigation capabilities of a self-learning robot.

The rest of the paper is organized as follows: Section 2 and 3 briefly describe Q-Learning and Sarsa and section 4 describes the DQN algorithm and the network architecture. Then, Section 5 describes the experimental setup and Section 6 discusses the results obtained in the simulated and real-world experiments. Finally, Section 7 concludes the works done in the paper.

2 Q-Learning

Q-Learning [5] is an Off-Policy Temporal Difference (TD) algorithm where an optimal policy can be learned in both exploratory and random based approaches [6]. Eqn. 2.1 shows the equation used to obtain the updated Q-value $Q^{\text{New}}(s_t, a_t)$ for Q-Learning.

$$Q^{\text{New}}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.1}$$

where $Q^{\text{New}}(s_t, a_t)$ is the new Q-value, $Q(s_t, a_t)$ is the old Q-value, α is the learning rate, r_{t+1} is the reward obtained after completing an action, γ is the discount factor, and $Max_a Q(s_{t+1}, a)$ is the max estimated future Q-value associated with all actions.

During $Q(s_t, a_t)$ update, ϵ-greedy [7] is used to choose action a_{t+1} which can be either a random action or the best current action based on the estimates of

Q-value for different actions. The probability equation used in ϵ-greedy to choose action a_{t+1} is shown in Eqn. 2.2.

$$Probability = (1 - \epsilon) + \frac{\epsilon}{k} \tag{2.2}$$

where:

ϵ = exploration constant
k = the number of different actions to select

In comparison to Sarsa, the action with the highest Q-value is chosen as action a_{t+1} which is shown as $Max_a Q(s_{t+1}, a)$. Additionally, the $Q(s_t, a_t)$ update behaves the same way as a greedy policy where ϵ is 0 because it immediately uses the largest Q-value estimate for Q-value updates without further exploration. It is called Off-Policy because it evaluates the action chosen by the ϵ-greedy method while using a different greedy policy to update $Q(s_t, a_t)$. At the end of every iteration of the Q-Learning process loop, only the state s_t is updated with state s_{t+1}.

3 Sarsa

Sarsa [8] is an On-Policy TD algorithm. In comparison to Q-Learning, the largest estimate for the successor state is not necessarily selected to update the Q-values. Rather, the policy that decides the prior action is used to choose the new action [9]. Eqn. 3.1 shows the equation used to obtain $Q^{\text{New}}(s_t, a_t)$ for Sarsa.

$$Q^{\text{New}}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{3.1}$$

where $Q^{\text{New}}(s_t, a_t)$ is the new Q-value, $Q(s_t, a_t)$ is the old Q-value, α is the learning rate, r_{t+1} is the reward obtained after completing an action, γ is the discount factor, and $Q(s_{t+1}, a_{t+1})$ is the estimated future Q-value.

Similar to Q-Learning, ϵ-greedy method [7] is used to choose action a_{t+1}. However, the main difference between Sarsa and Q-Learning is in how the future Q-value estimate is obtained. During $Q(s_t, a_t)$ update, the estimate of future Q-value is a result of directly using ϵ-greedy method as seen by the action a_{t+1} in $Q(s_{t+1}, a_{t+1})$. It is called On-Policy because it evaluates the action chosen by the ϵ-greedy method and the same policy is also used to update $Q(s_t, a_t)$. At the end of every iteration of the Sarsa process loop, the state s_t and action a_t is updated with state s_{t+1} and action a_{t+1} respectively.

4 Deep Q-Learning (DQN)

Deep Q-Learning (DQN) [10] is a TD algorithm that is based on the Q-Learning algorithm that makes use of a deep learning architecture such as the Artificial

Neural Networks (ANN) as a function approximator for the Q-value. The input of CNN are states of the agent and the output is the Q-values of all possible actions. On its own, learning values using a neural network is prone to instability and divergence. So, a separate Target Network and Experience Replay have been used to stabilize the learning. Eqn. 4.1 is the same equation as Eqn. 2.1 because the method to update $Q(s_t, a_t)$ in DQN is based on Q-Learning.

$$Q^{\text{New}}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[\underbrace{r_{t+1} + \gamma Max_a Q(s_{t+1}, a)}_{Target\ Q-value function} - Q(s_t, a_t)] \tag{4.1}$$

4.1 Separate Target Network

As shown in Fig. 4.1, in addition to the DQN model (Prediction Network) itself, a copy of the same model called the Target Network is used for target Q-value estimation. While the Target Network has the same architecture as the Prediction Network, its parameters do not get updated until a certain amount of iterations is reached. For every user-defined iteration, the parameters of the Target Network are overwritten by the Prediction Network parameters, which leads to a more stable training because the target Q-value function stays the same until the user-defined amount of iteration is reached.

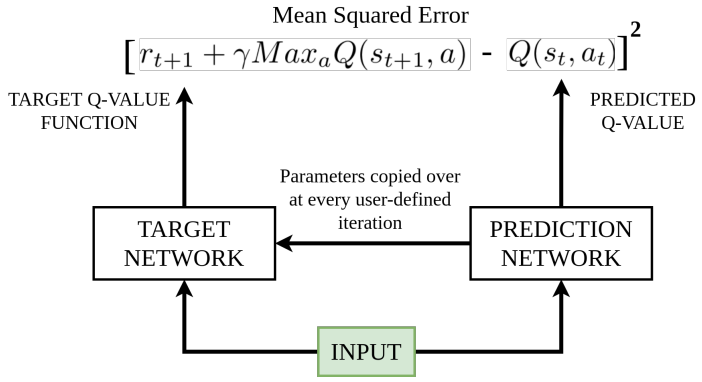

Fig. 4.1. Target Network and Prediction Network for model training.

4.2 Experience Replay

The concept of Experience Replay [11] uses memory to store tuples of (action a_t, state s_t, reward r_{t+1}, the following state s_{t+1}) after an RL agent has completed action a_t in state s_t and receives a feedback in the form of reward r_{t+1} and

following state s_{t+1}. After the size of the memory reaches the user-defined size, the tuples are sampled randomly and the target Q-value function is computed using the Target Network. Then, the Q-values from the target Q-value function and states are used to train the Target Network model for 1 epoch and the output action with the largest Q-value has a probability to be selected based on the ϵ-greedy method [7].

4.3 DQN Process Workflow

The overall steps involved in DQN are as follows. Firstly, the agent will take a random action because the ϵ (Exploration constant) is set to 1.0, and actions are determined according to ϵ-greedy method [7]. Similar to Q-Learning [5], the action with the highest Q-value is chosen. Over time, actions will be chosen based on the highest Q-value output from the Target and Prediction Network because a ϵ decay function is applied for every next episode. Then, the reward r_{t+1} and following state s_{t+1} is obtained as a feedback after taking action a_t. The action a_t, state s_t, reward r_{t+1}, and following state s_{t+1} information is stored in an Experience Replay memory tuple. After that, the Prediction Network is used to approximate the estimated future Q-value based on the randomly sampled mini-batch of the Experience Replay, and the current state s_t is overwritten by the next state s_{t+1}. If a certain number of episode steps is reached, the Target Network will be used for estimated future Q-value approximation and the network weights of the Prediction Network will be copied over to the Target Network. Finally, mean squared error method is used as a loss function to compute the variation between the obtained estimated future Q-value and the target Q-value [12] ($[r_{t+1}+\gamma Max_a Q(s_{t+1}, a)-Q(s_t, a_t)]^2$) as seen in Fig. 4.1. The entire process loops until an agent have reached the maximum episode or steps or a stopping criterion is met.

4.4 Network Architecture

The network architecture of the Prediction Network and Target Network are identical and illustrated in Fig. 4.2. The architecture consists of two hidden dense layers of size 300 nodes each which take an input of size 54 and produces an output of size 3. For these two hidden layers, the kernel initializer of 'LecunUniform', kernel regularizer of 'L2' with a regularization factor of 0.01 were used and the bias is set to True. The activation function used for both hidden layers was Rectified Linear Unit (ReLU). For the output layer, a dense layer of size 3 with a kernel initializer of 'LecunUniform' and the bias was set to true. A linear activation function was used for the output layer. Finally, Mean Squared Error (MSE) was used as a loss function, and RMSprop with a learning rate of 0.00025, a discount factor of 0.9, and an epsilon of 0.00000001 were used for the optimizer.

The network input size of 54 nodes corresponds to the 52 uniform laser scan distance (LSD) readings taken from a 360-degree horizontal field of view (FOV) sensor, 1 distance to goal (DTG) reading, and 1 heading angle to goal (HTG)

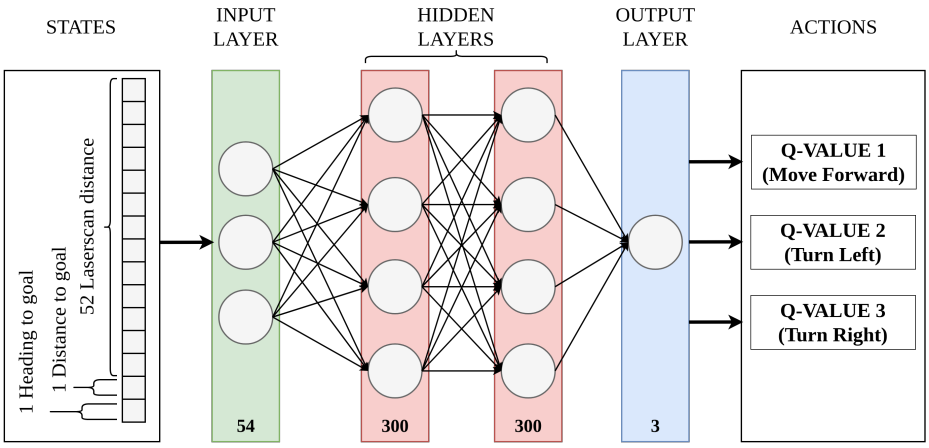

Fig. 4.2. Deep Q Network model architecture.

reading. On the other hand, the network output size is 3 nodes which are the three Q-values for the three available actions that can be taken (i.e move forward, turn left and turn right). The action with the highest Q-value will be selected by the RL agent.

5 Experimental setup

In our experiments, we have used an open-source ToolKit called extended OpenAI Gym ToolKit from erlerobot [13] to compare the performances of DQN, Q-Learning, and Sarsa algorithms in both simulated and real-world environments. The mobile robot hardware used was the Turtlebot3 Burger. The robot is equipped with an LDS-01 LIDAR sensor which is a 360 Degree Laser Distance Sensor that can register 360 distance to obstacle values. A motor encoder and the IMU sensor are used to accurately compute the odometry of the robot. Fig. 5.1 shows the simulated environment that is used to perform RL training and testing on the robot. Meanwhile, Fig. 5.2 shows the real-world environment that is used to deploy and test the trained RL models.

5.1 Q-Learning and Sarsa Parameters

The hyperparameter settings for both Q-Learning and Sarsa followed the same settings used in the OpenAI Gym Toolkit paper [13]. The learning rate α was set to 0.2, the discount factor γ was set to 0.9, and finally, the exploration constant ϵ was set to 0.9. An ϵ decay function was used and set at 0.9986 to discount the ϵ value for every episode until it has reached 0.05.

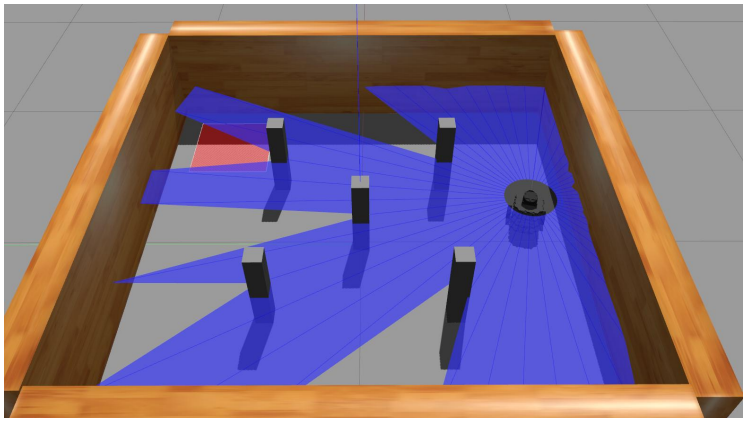

Fig. 5.1. The simulated environment is a square-spaced environment with 2m length walls. Within these walls, five obstacles were arranged close to each other and in the way of the goal position shown by the red box.

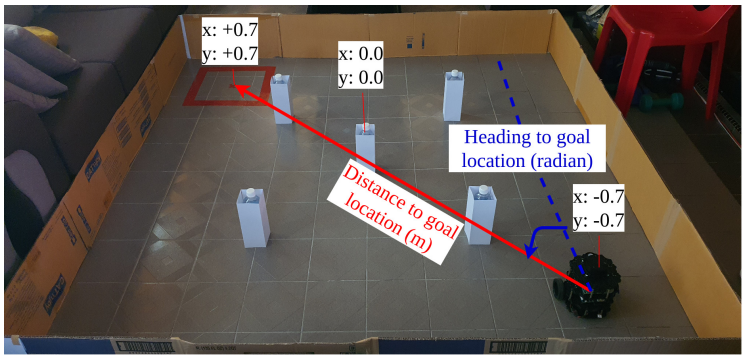

Fig. 5.2. The real world environment used in the experiments.

5.2 DQN Parameters

The hyperparameter settings for DQN were based on the work of the authors who proposed the Deep Q-Network algorithm [10]. The learning rate α was set to 0.00025, the discount factor γ was set to 0.99, and finally, the exploration constant ϵ was set to 1.0. An ϵ decay function was also used and set at 0.995 to discount the ϵ value for every episode until it has reached 0.05.

5.3 Training Setup

The number of maximum episodes and step counts per episode are 1500 and 200 respectively. For the first 1000 episodes, the minimum distance to obstacles was set at 0.18m to create a more strict training setup to allow the robot to learn to navigate to the goal position while maintaining a safe distance to nearby

obstacles. For the last 500 episodes, the training has been made less strict by changing the minimum distance to 0.12m which is close to the physical robot's base radius dimension. This allows the robot to learn and navigate more because of the less strict obstacle collision criterion. As a result, the robot is more likely to reach its goal position.

An episode is considered successful if the robot has managed to reach within 0.2m of the set goal position of x=0.7m and y=0.7m from the starting position of x=-0.7m and y=-0.7m. Meanwhile, an episode is considered a failure if any of its LIDAR sensor's 52 distance information is within 0.18m or 0.12m of any obstacle (i.e obstacle collision). An episode will terminate when the robot has successfully navigated to the goal position or when it has experienced obstacle collision. One episode step here refers to the robot performing one discrete action for 0.2s. If the episode step count reaches its maximum value, the episode will terminate. The entire RL training process will end after it has reached the maximum episode.

The agent can take three discrete actions namely move forward, turn left and turn right and successful execution of any of these three actions will yield a numerical reward. A move forward action can yield a reward of 5 while a turn left or right action yields a reward of 1. The agent can also earn a negative reward of -200 if any of its LIDAR sensor's 52 distance information is within 0.18m or 0.12m of any obstacle (i.e obstacle collision). Additionally, a decrease in distance and heading to goal position will yield a reward of 5 and 1 respectively.

For all the algorithms, The agent's state spaces consisted of 52 LSDs, 1 DTG, and 1 HTG reading. In DQN, the agent uses continuous values for all of its state spaces. Meanwhile, state space discretization was applied for Q-Learning and Sarsa by bucketing a range of continuous values into discrete values. The LSDs and DTG readings were discretized to steps of 0.1m over the range of 0 to 3.0 while the 1 HTG reading was discretized to steps of 0.19625 radians over the range of -3.14 to 3.14.

6 Results and Discussion

In this section, the navigation performances of DQN, Q-Learning, and Sarsa for both the training and testing phase are compared. The following metrics have been used for analytical comparison: 1. cumulated rewards in an episode, 2. cumulated steps in an episode, and 3. rate of successful navigation which is the percentage of successful episodes.

Fig. 6.1 and Fig. 6.2 show the visual comparison of the rewards earned and steps achieved by the agent for every episode during the training in a simulated environment. Meanwhile, Table 6.1 shows the numerical comparison for the average of each metric for every 100th episode during training in the simulated environment. Finally, Table 6.2 shows the result of simulated and real-world environment testing averaged over 10 runs.

During the training phase, the robot was trained in the simulated world shown in Fig. 5.1. For the DQN algorithm training, there was a general increase in the success rate. From episode 1400 to 1500, the agent successfully navigated

Table 6.1. A numerical comparison between the average rewards (R), steps (S), and success rates (SR) of DQN, Q-Learning, and Sarsa for every 100th episode during the training in simulated world.

Episode Intervals	**DQN**			**Q-Learning**			**Sarsa**		
	R	S	SR	R	S	SR	R	S	SR
0-100	74	38	0%	22	30	0%	34	32	0%
100-200	222	59	1%	15	30	0%	26	31	0%
200-300	398	84	4%	26	31	0%	36	33	0%
300-400	638	110	20%	18	29	1%	4	28	1%
400-500	563	101	20%	27	30	2%	9	29	0%
500-600	472	91	13%	31	32	0%	23	30	1%
600-700	606	105	23%	37	32	0%	-3	27	0%
700-800	671	110	32%	21	30	0%	14	30	0%
800-900	907	125	63%	26	31	0%	18	30	0%
900-1000	915	124	66%	13	29	0%	9	29	0%
1000-1100	1059	139	75%	75	38	1%	39	33	1%
1100-1200	1089	139	86%	44	35	0%	49	33	3%
1200-1300	1046	134	84%	63	36	2%	53	35	1%
1300-1400	1075	138	86%	37	33	0%	46	34	2%
1400-1500	963	115	87%	74	38	1%	45	32	1%

Table 6.2. An average of 10 single episodes numerical comparison of all metrics for DQN, Q-Learning, and Sarsa during the simulation (ST) and real-world testing (RT).

Performance Metrics	**DQN**		**Q-Learning**		**Sarsa**	
	RT	ST	RT	ST	RT	ST
Average Rewards	938	947	6	-18	25	22
Average Steps	106	103	31	27	29	30
Average Success Rates	100%	100%	0%	0%	0%	0%

to the goal position in 87 out of 100 episodes. In Table 6.1, it can be seen that the average rewards from episode 1400 to 1500 is much lower (963) compared to the rewards from episode 1000 to 1400 (1059, 1089, 1046, 1075) despite the 1% increase in success rate. This is attributed to the reduced number of steps taken by the robot to reach the goal position as a result of the more lenient obstacle collision criterion in later episodes. A learning strategy improvement can be

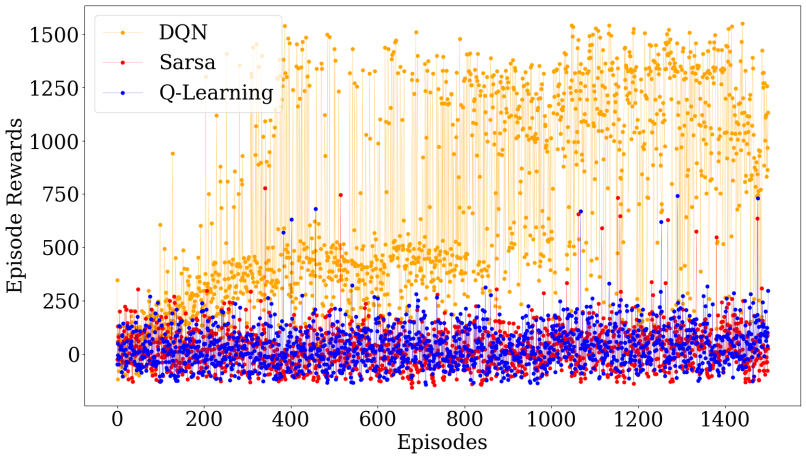

Fig. 6.1. A plotted graph of all episode rewards obtained by the robot for DQN, Q-Learning, and Sarsa during the training in simulated world.

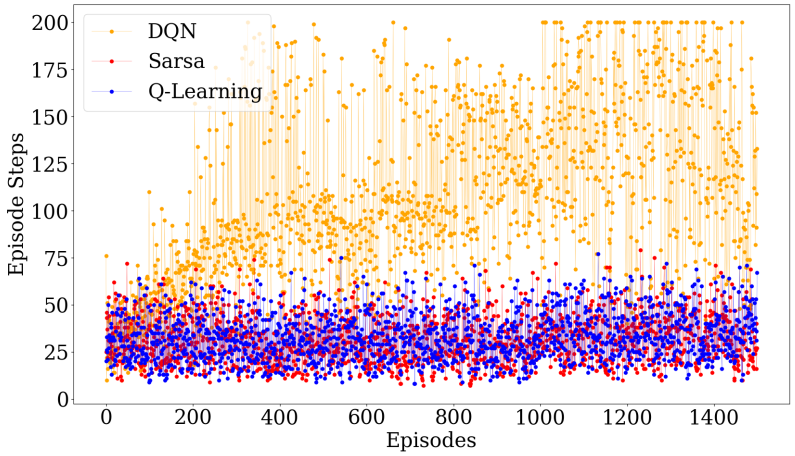

Fig. 6.2. A plotted graph of all episode steps made by the robot for DQN, Q-Learning, and Sarsa during the training in simulated world.

made by making some changes to the reward system where we can reward the robot for taking fewer steps to navigate to the goal position. The robot will be rewarded more for taking less amount of action to complete an episode and this can encourage the robot to learn a more efficient navigation policy.

For both Q-Learning and Sarsa algorithm training, the result in Table 6.1 shows that the robot was not capable of learning any useful navigation policies. Throughout every 100th episode, the robot had a success rate of 0% with a few occasions of 1% and 2%. An explanation as to why there are successful outlier episodes from episodes 0 to 1000 is that ϵ-greedy technique [7] had managed to randomly generate a set of actions that led it to the goal position. Beyond episode 1000, the ϵ hyperparameter had been discounted from 0.22 to 0.11 due to the ϵ decay function so the probability of a random action being generated by ϵ-greedy was low. So, the main factor for the cause of these successful outlier episodes was from the nature of the two algorithms that is to take a random action for it to populate the Q-table for learning if a state observation value is not found in the Q-table.

During the testing phase, we deployed the trained model from the DQN algorithm in both simulated and real environments shown in Fig. 5.1 and Fig. 5.2 respectively and averaged out the results from a total of 10 runs. It can be seen that the DQN model has managed to obtain a 100% success rate and obtained more or less the same amount of average rewards and steps in both the simulated and real-world testing.

On the other hand, we did the same test for Q-Learning and Sarsa and found that the robot failed to navigate to the goal position because the discretized state space was too large. These algorithms are more suitable to be used for RL tasks with smaller state spaces such as cliff walk (60 discrete states), mountain car (121 discretized continuous states), and cart pole balancing (162 discretized continuous states) [14]. In our environment, the discretized state space size was about 6.2 x 10^{79} states (30^{52} x 30^{1} x 32^{1}). Reducing the 52 LSD readings to just one reading could exponentially reduce the state space size to 28,800 states (30^{1} x 30^{1} x 32^{1}). However, the agent would not be able to learn collision-free navigation because it would not detect most of the obstacles during its navigation.

7 Conclusion

This study compared a deep reinforcement learning approach with two classical RL approaches for obstacle avoidance in local navigation. Our results show that the DQN could successfully navigate from its start location to goal location with a 100% success rate in 10 runs in both the simulated and real-world environments. On the other hand, Q-Learning and Sarsa could not complete the course in all 10 runs. Based on the empirical results from simulated and real-world testing, it can be seen that Q-Learning and Sarsa did not work well in an environment with a large state space. This is true in many real-world environments where the continuous state space would discretize to large discrete state spaces. On the other hand, DQN can work directly and perform well in continuous state space.

The source code for this work is available at `https://gitlab.com/ailab.space/comparison-of-dqn-with-q-learning-and-sarsa-for-robot-local-navigation`.

References

[1] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[2] SM Raguraman, D Tamilselvi, and N Shivakumar. "Mobile robot navigation using fuzzy logic controller". In: *2009 International Conference on Control, Automation, Communication and Energy Conservation*. IEEE. 2009, pp. 1–5.

[3] Shuzhi Sam Ge and Yun J Cui. "Dynamic motion planning for mobile robots using potential field method". In: *Autonomous robots* 13.3 (2002), pp. 207–222.

[4] Yanrong Hu and Simon X Yang. "A knowledge based genetic algorithm for path planning of a mobile robot". In: *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*. Vol. 5. IEEE. 2004, pp. 4350–4355.

[5] JCH Christopher. "Watkins and peter dayan". In: *Q-Learning. Machine Learning* 8.3 (1992), pp. 279–292.

[6] *Reinforcement Learning: Q-Learning vs Sarsa*. `http://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html`. Accessed: 2021-06-14.

[7] Chris Watkins. *Learning from Delayed Rewards*. Cambridge, England, 1989.

[8] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[9] *Reinforcement Learning: Sarsa vs Qlearn*. `https://studywolf.wordpress.com/2013/07/01/reinforcement-learning-sarsa-vs-q-learning`. Accessed: 2021-06-14.

[10] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[11] Mohit Sewak. *Deep Reinforcement Learning Frontier of Artificial Intelligence*. 2019, p. 215. ISBN: 9789811382840.

[12] *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python*. `https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/`. Accessed: 2021-06-14.

[13] Iker Zamora et al. *Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo*. 2017. arXiv: `1608.05742 [cs.RO]`.

[14] Yin-Hao Wang, Tzuu-Hseng S Li, and Chih-Jui Lin. "Backward Q-learning: The combination of Sarsa algorithm and Q-learning". In: *Engineering Applications of Artificial Intelligence* 26.9 (2013), pp. 2184–2193.