Maze Solving Using Deep Q-Network

Anushtup Nandy
f20201981@hyderabad.bits-pilani.ac.in
Department of Mechanical Engineering
Birla Institute of Technology & Science (BITS) Pilani, Hyderabad Campus
Hyderabad, Telangana, India

S Subash
f20200454@hyderabad.bits-pilani.ac.in
Department of Mechanical Engineering
Birla Institute of Technology & Science (BITS) Pilani, Hyderabad Campus
Hyderabad, Telangana, India

Abhishek Sarkar
abhisheks@hyderabad.bits-pilani.ac.in
Department of Mechanical Engineering
Birla Institute of Technology & Science (BITS) Pilani, Hyderabad Campus
Hyderabad, Telangana, India

ABSTRACT

Path planning and obstacle avoidance are crucial for enabling the autonomy of mobile robots to operate in real-world environments. Conventional algorithms are known to be computationally expensive, and they require prior knowledge of the environment. In this paper, instead of using conventional algorithms, we present the usage of DQN (Deep Q-Network), a reinforcement learning algorithm, to solve the path planning problem. The goal was to observe and report the feasibility of using DQN as a path-planning algorithm for mobile robots in maze environments with walls leading to dead-ends. We have also showed the possibility of improvements to the algorithm, which can be used in more challenging maze environments.

CCS CONCEPTS

• **Computing methodologies** → **Motion path planning**; **Planning for deterministic actions**; *Continuous space search*; *Randomized search*; **Q-learning**; *Batch learning*; • **Computer systems organization** → *Neural networks*; • **Theory of computation** → **Markov decision processes**; **Sequential decision making**.

KEYWORDS

path planning, obstacle avoidance, DQN

ACM Reference Format:
Anushtup Nandy, S Subash, and Abhishek Sarkar. 2023. Maze Solving Using Deep Q-Network. In *Advances In Robotics - 6th International Conference of The Robotics Society (AIR 2023), July 05–08, 2023, Ropar, India.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3610419.3610458

1 INTRODUCTION

In the modern world, solving mazes becomes necessary when there is either little or no knowledge of the map. Using the available information about the maps in a search and rescue operations is one example of a scenario where the optimal route is determined. Finding the most direct, hassle-free, and efficient route to the destination point is the major goal in this situation.

In these maze-solving scenarios, the agent has no idea how its environment will be, which is also the case with search and rescue operations. Also, this can be helpful for exploration of sites to uncover new pathways and result in discovering unknown terrain, i.e. space exploration.

Planning a route becomes essential when there are numerous pathways, obstacles, and dead ends in the path chosen by the agent. The reinforcement learning algorithm makes the agent attain knowledge of the environment through multiple attempts made to reach the goal point although may have resulted in failures. This involves modeling the problem into Markov's decision model. This maps the environment function with the state action reward function for that problem. We have our start and goal positions defined, agent's responsibility now is to find its way from the available pathways and avoid any sort of obstructions on its path. One more major difficulty for mobile robots is in a dynamic environment when the environment characteristics are not constant.

Starting with classical algorithms some of which are Dijkstra, A*, D*, Depth First Search(DFS), and Breadth-first Search (BFS), etc. They all function based on the cost function which gets updated after each iteration of the search. Limitations of these algorithms are that we have some information about the map beforehand which is not always the case in real-world problems and that they are computationally expensive [13]. Some algorithms like BFS do come with memory constraints and are time-consuming as the goal distances increases. In DFS we may not even know if we will get the solution. A few of the above shortcomings are taken care of when we use learning-based algorithms [2, 4, 5, 7–9, 11, 12]. Learning-based algorithms make it easier to understand the environment, which parts are smooth for the agent to pass and which are challenging for the agent. Reinforcement learning (RL) is the process of learning based on state reward action function [7, 9, 12]. When RL is combined with neural networks for input and output is what is known today as DQN (Deep Q-network) algorithm. The DQN algorithm [10] focuses to train the agent by assigning weights to its decisions made. In comparison with traditional algorithms like Dijkstra, A*, D*, etc, which are based on approximations from real-life conditions. This traditional algorithm considers all trials made by the agent to be of equal importance, while the DQN algorithm assigns weights based on the result of the preceding trial. Neural networks here help us in

AIR 2023, July 05–08, 2023, Ropar, India

ACM ISBN 978-1-4503-9980-7/23/07…$15.00
https://doi.org/10.1145/3610419.3610458

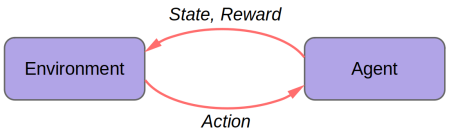


Figure 1: Interaction between elements in reinforcement learning

updating the state action reward function in a gradient-based manner rather that a linear model. Ultimately all these steps become critical in collision avoidance and the primary focus of the agent is to reach its destination (known), from the start [4]. The updating of weights can be done more finely through the use of multiple layers of neural networks. Here the agent can be trained to get sensor data which will give us about the presence of walls or ends of the path, and the application of DQN based on the attempts of the agent. DQN can be summarized as a combination of Q-learning and neural networks [3]. This process of solving to find the path to reach the goal point for the agent is purely an iterative approach and uses the sensor data from the agent as its observations for the decision-making step.

2 METHODOLOGY

In this section, we showcase, in short, the RL framework. Then we move on to the DQN algorithm which we have used to solve RL problems. Then we give a brief description of the simulation environment, the proposed Deep Q-network, and the constraints.

2.1 Theory

In this paper, we have used reinforcement learning-based optimization to solve the motion planning problem. In Reinforcement Learning or RL, the optimal control policy is designed as a Markov Decision Process (state(S), action(A), policy(P), Reward(R), discount factor(γ)), where the main objective is to maximize some rewards at each step. The main elements of a reinforcement learning process [1], as demonstrated in figure 1, are the Agent (the robot in our case), the Reward(R), the environment, the policy(P), and the value function. The Reward helps the Agent differentiate between good and bad actions. The environment is the space where the Agent performs its decisions. The policy defines the Agent's behavior at a particular time and the value function, which is the total reward an agent can expect to get in the future, starting from that state. In the searching problem we solve, the Agent or the robot is described by its state (s_t) for a given time. The Agent can take different actions ($a_t \in A$), which allows the Agent to move from s_t to $s_{(t+1)}$. Here, the `control policy` or the way the Agent will move, is decided by the $\pi(a_t|s_t)$, which is the conditional probability of taking action at provided the Agent is in s_t. The main objective in motion planning problems like the one we have presented here is to find the optimal policy π^* [2, 3, 8, 9, 12]. We solve this problem using reinforcement learning where the Agent gets rewards $R(s_t, a_t)$ on moving from s_t to $s_(t+1)$, and it needs to maximize this Reward. The total Reward is

$$R(s_{t+1} = s|a_t = a) = \sum_{t=0}^{N} \gamma^t R(s_t, a_t) \quad (1)$$

where N is the number of steps taken along a particular trajectory (some policy π_t). Since there can exist a number of such policies, the overall performance of a trajectory is given by the value function:

$$V^{\pi}(s) = E[R^{\pi}|s_0 = s] \quad (2)$$

and the agent needs to find:

$$V^*(s) = max_{\pi}(V^{\pi}(s)) \ \forall \ s \in S \quad (3)$$

2.2 DQN Algorithm

The most commonly used RL algorithm for path planning problems is the Q-learning algorithm [5]. This method suffers from the problems of slow convergence, dimensionality disaster as well as low efficiency. For this purpose, the DQN was produced. In this paper, we use the DQN Algorithm [10] to learn the control policies. DQN is a variant of the Q-learning algorithm [5], which is a model-free, off-policy reinforcement learning that finds the best course of action, given the agent's current state. Depending on the location of the agent, it decides the next action to be taken. Model-free means that the agent uses predictions of the environment's expected response to move forward rather than the reward system to learn, basically trial and error. The core Q-learning algorithm [5]:

$$Q^{new}(s_t, at) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma * maxQ(s_{t+1}, a_t) - Q(s_t, a_t)) \quad (4)$$

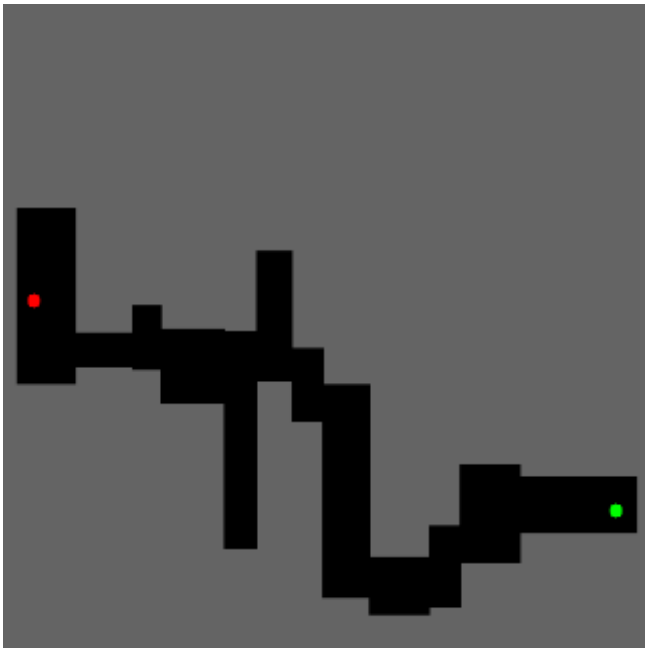
Where α, r_t,γ are the learning rate, reward, and discount factor, respectively. $Q(s_t, a_t)$ is the current Q-value,$maxQ(s_{t+1}, a_t)$ is the maximum expected future reward given some new state s_{t+1} and all possible actions. Q-learning is a tabular method that uses a "look-up table" to keep track of and estimate the state and action space values. In contrast, the DQN algorithm combines the massive prediction and estimation capabilities of an ANN with the Q-learning algorithm. This is because many realistic environments have a large state and action space, whereas the Q-learning table can only be used for a finite number of state and action space values. A Q-learning algorithm's computational and memory requirements in a relatively large environment lead to unstable learning. The DQN algorithm overcomes this by using these techniques: Target network ($Q(s_{t+1}, a_t)$), reward clipping, and experience replay (to avoid over-fitting and catastrophic forgetting).

In DQN, input for the neural network depends on these factors: robot environment, robot behaviour, and sensing of the robot. The output provides the action the agent should take.

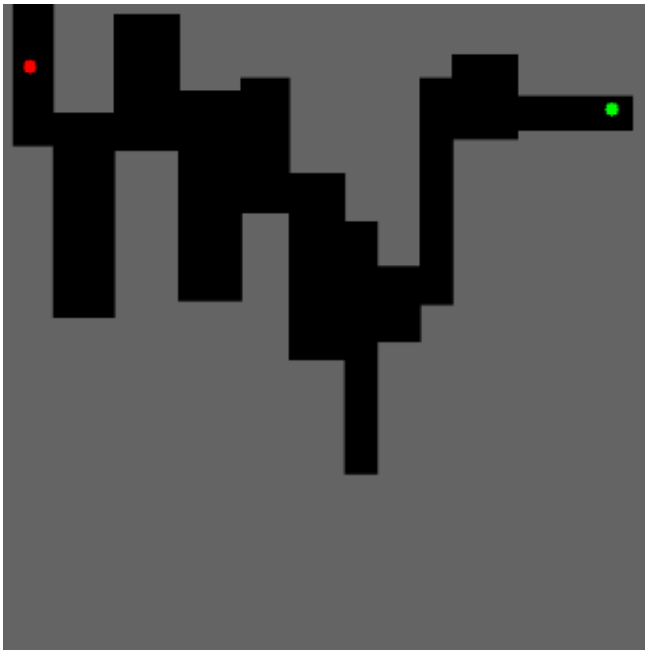
2.3 Simulation

The environment was created with the objective of having obstacles or walls that lead to paths that do not end at the goal point. The maps where we train and test our path planning agent are shown in Fig 2(a) and (b) are designed slightly differently so that Map 1 shown in Fig 2(a) has only 2 misleading paths and is relatively simpler than Map 2 shown in Fig 2(b) which has 4 paths which do not end in the goal point. The agent takes longer to explore Map 2 as compared to Map 1. As shown in Fig 2 and 3, the maps have been designed using OpenCV and Numpy. The black region represents the free space for the robot to move, and the grey region describes the walls. The agent's starting "x" (horizontal) coordinate is fixed, and the starting "y" (vertical) coordinate is taken randomly between

the starting x and the limit of the simulation window, which we have set to be 1.



(a) Map 1



(b) Map 2

Figure 2: The Environments designed using OpenCV and Numpy

The DQN algorithm has been trained using a *Fully connected neural network*, with two hidden layers with 128 nodes (features), each activated using the relu function, and two inputs and optimized using the ADAM optimizer which is regarded to be robust to the choice of hyper parameters [6]. The learning rate was set to 0.005 with a discount factor of 0.9 and epsilon equal to 0.25. This algorithm has been implemented in PyTorch, with a linearly reducing epsilon and a linearly decaying learning rate to stabilize the learning process. It takes input into the agent's actions, the learning rate, and the discount factor.

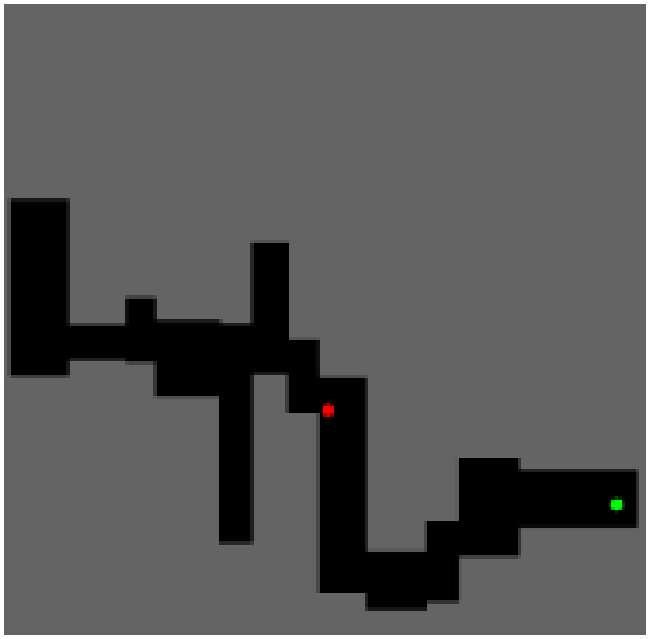


Figure 3: Agent exploring Map-1

For our particular environment, the state space is represented as 2D coordinates (x, y), and it represents the position of the agent within the environment. Similarly, the action space is 2D represented as (d_x, d_y), which is the change in x and y coordinates, respectively. The agent can take these actions to move in any direction within the environment. The reward function is based on the change in the distance to the goal. The reward is designed to encourage the agent to minimize the distance to the goal while providing a higher reward for moving to a better state than staying in the same state.

The process of data collection is done by the agent through interaction with the environment. After each interaction, the agent creates a tuple containing the current state, action, reward and the next state, which is then added to the Replay buffer. The Q-Network is trained using the tuple added to the replay buffer; this is performed when the number of steps taken is greater than the steps taken.

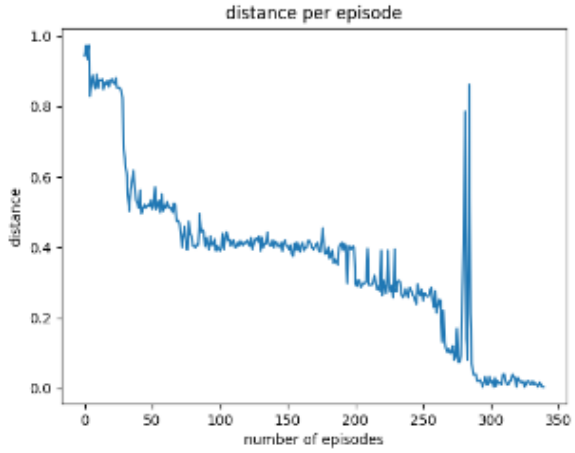
The entire simulation runs for a maximum of 600 seconds and 800 seconds, which is set as the terminating criteria. Each episode has a maximum length of 140. It begins with a random initialization of the agent's y-coordinate and that of the goal. The episodes have a terminal state of the agent hitting the wall, or if the remainder of the number of steps taken by the agent and the episode's length

is equal to 0. When this happens, the simulator is automatically restarted with a new episode. The agent keeps exploring till it hits a wall or the episode finishes.

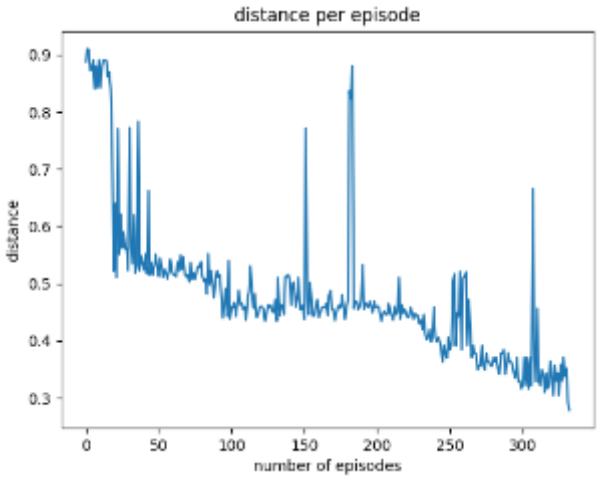
The agent receives a custom reward based on its next state and the distance to its goal. The agent can move by 0.01units in up, down, left or right directions.

In this work, we present our findings after comparing the effects of four cases in 2 different maps as well as 2 different times as the terminating criteria. The cases we explored for Map 1 are: linearly decaying learning rate with a linearly decremented epsilon, linearly decaying learning rate and exponentially decaying epsilon, exponentially decaying learning rate and linearly decaying epsilon, and finally where both the learning rate and the epsilon are exponentially decaying. The findings are presented in the experimentation section.

3 EXPERIMENTATION AND RESULTS



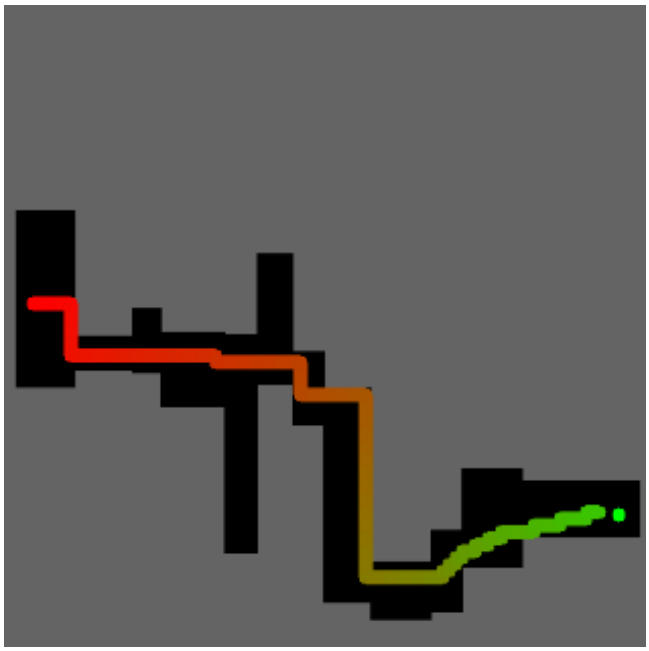
(a) Graph 1



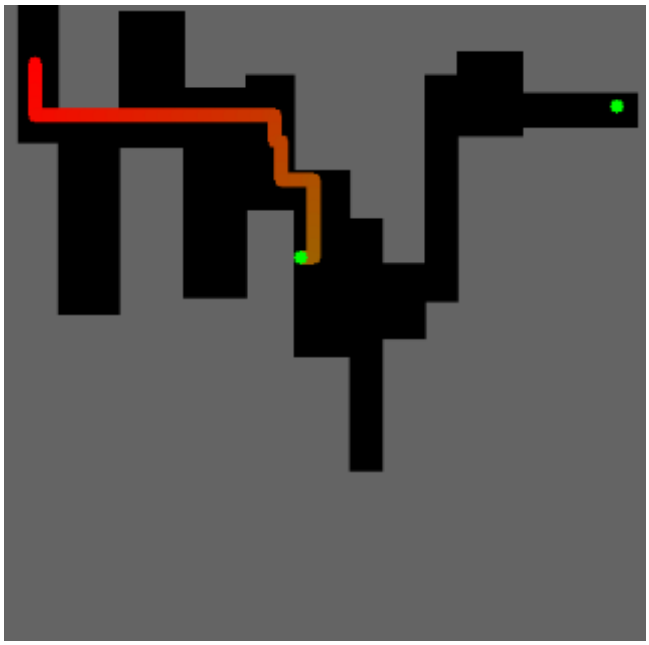
(b) Graph 2

Figure 4: The distance left to travel to reach goal as the episodes progress

Table 1 showcases the four different cases performances, and lays the grounds for our conclusion.



(a) Map 1



(b) Map 2

Figure 5: The path taken by the agent in 800s for Map-1 and Map-2

The entire simulations were run for periods of 600s and 800s. The running times were decided after lots of testing to see how long it takes for the agent to converge to the goal point. The results

Map	Time to goal during training	steps to goal during testing	final distance to goal
Map 1	620s	77	0
Map 2	did not reach	did not reach	0.458

Table 1: Comparison of performance on the Maps

table quantifies our findings, that the agent was not able to train itself to reach the goal under 600s, but 800s was more than enough for the agent to reach the goal point in the simpler environment of Map-1. The above results show the quantitative observations for the maze solver. Fig 4(a), shows us the distance to the goal that the agent has left to travel in each iteration for making the decision. It is high in the beginning and decreases slowly. We were able to observe wherever there are junctions, it covers more distance in that iteration before arriving at its decision. The same quantities are observed for map 2 as displayed in Fig 4(b). We can observe that as map 2 has more branches in comparison to map 1, it has got more distance covered per episode for its decision-making due to more complexity.

4 CONCLUSION

We have developed a DQN-based maze solver for solving the objective of goal-reaching. We generated the maps in Python for the simulation using OpenCV and Numpy. We have used a linearly decreasing learning rate and found it to converge faster in the simpler maze. As the maze becomes complicated, more time is required for convergence. Q-Learning is computationally less efficient and takes more memory compared to the DQN. Hence we choose to implement the DQN algorithm for our maze solver. As we have shown, the DQN converges in more straightforward mazes, and that fails to do so in complicated ones. Improvements such as using an exponentially decaying learning rate, which prevents divergence as the training proceeds, or replacing the DQN with a more advanced algorithm as the DDQN (Double DQN), will help increase the chances of reaching an optimal solution.

REFERENCES

[1] Leemon Baird. 1995. Residual Algorithms: Reinforcement Learning with Function Approximation. In *Machine Learning Proceedings 1995*. Elsevier, 30–37. https://doi.org/10.1016/B978-1-55860-377-6.50013-X

[2] Matej Dobrevski and Danijel Skočaj. 2021. Deep Reinforcement Learning for Map-Less Goal-Driven Robot Navigation. *International Journal of Advanced Robotic Systems* 18, 1 (Jan. 2021), 1729881421992621. https://doi.org/10.1177/1729881421992621

[3] Meng Guan, Fu Xing Yang, Ji Chao Jiao, and Xin Ping Chen. 2021. Research on Path Planning of Mobile Robot Based on Improved Deep Q Network. *Journal of Physics: Conference Series* 1820, 1 (March 2021), 012024. https://doi.org/10.1088/1742-6596/1820/1/012024

[4] Siyu Guo, Xiuguo Zhang, Yiquan Du, Yisong Zheng, and Zhiying Cao. 2021. Path Planning of Coastal Ships Based on Optimized DQN Reward Function. *Journal of Marine Science and Engineering* 9, 2 (Feb. 2021), 210. https://doi.org/10.3390/jmse9020210

[5] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. 2019. Q-Learning Algorithms: A Comprehensive Classification and Applications. *IEEE Access* 7 (2019), 133653–133667. https://doi.org/10.1109/ACCESS.2019.2941229

[6] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs]

[7] Geesara Kulathunga. 2022. A Reinforcement Learning Based Path Planning Approach in 3D Environment. arXiv:2105.10342 [cs]

[8] Xiaoyun Lei, Zhian Zhang, and Peifang Dong. 2018. Dynamic Path Planning of Unknown Environment Based on Deep Reinforcement Learning. *Journal of Robotics* 2018 (Sept. 2018), e5781591. https://doi.org/10.1155/2018/5781591

[9] Feiqiang Lin, Ze Ji, Changyun Wei, and Hanlin Niu. 2021. Reinforcement Learning-Based Mapless Navigation with Fail-Safe Localisation. 100–111. https://doi.org/10.1007/978-3-030-89177-0_10

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602 [cs]

[11] Yuhang Ye. 2022. A Review of Path Planning Based on IQL and DQN. In *2022 3rd International Symposium on Artificial Intelligence for Medicine Sciences*. ACM, Amsterdam Netherlands, 190–195. https://doi.org/10.1145/3570773.3570808

[12] Matt Zucker and J. Andrew Bagnell. 2012. Reinforcement Planning: RL for Optimal Planners. In *2012 IEEE International Conference on Robotics and Automation*. IEEE, St Paul, MN, USA, 1850–1855. https://doi.org/10.1109/ICRA.2012.6225036