Proceedings of the ASME 2019
International Design Engineering Technical Conferences
and Computers and Information in Engineering Conference
IDETC/CIE2019
August 18-21, 2019, Anaheim, CA, USA

DETC2019-97536

MOBILE ROBOT OBSTACLE AVOIDANCE BASE ON DEEP REINFORCEMENT LEARNING

Shumin Feng*
Robotics and Mechatronics Lab
Mechanical Engineering Dept.
Virginia Tech
Blacksburg, VA, USA
shumin@vt.edu

Hailin Ren*
Robotics and Mechatronics Lab
Mechanical Engineering Dept.
Virginia Tech
Blacksburg, VA, USA
hailin@vt.edu

Xinran Wang
Robotics and Mechatronics Lab
Mechanical Engineering Dept.
Virginia Tech
Blacksburg, VA, USA
wxinran6@vt.edu

Pinhas Ben-Tzvi**
Robotics and Mechatronics Lab
Mechanical Engineering Dept.
Virginia Tech
Blacksburg, VA, USA
bentzvi@vt.edu

ABSTRACT

Obstacle avoidance is one of the core problems in the field of mobile robot autonomous navigation. This paper aims to solve the obstacle avoidance problem using Deep Reinforcement Learning. In previous work, various mathematical models have been developed to plan collision-free paths for such robots. In contrast, our method enables the robot to learn by itself from its experiences, and then fit a mathematical model by updating the parameters of a neural network. The derived mathematical model is capable of choosing an action directly according to the input sensor data for the mobile robot. In this paper, we develop an obstacle avoidance framework based on deep reinforcement learning. A 3D simulator is designed as well to provide the training and testing environments. In addition, we developed and compared obstacle avoidance methods based on different Deep Reinforcement Learning strategies, such as Deep Q-Network (DQN), Double Deep Q-Network (DDQN) and DDQN with Prioritized Experience Replay (DDQN-PER) using our simulator.

Keywords: Obstacle Avoidance; Deep Reinforcement Learning; Mobile Robots

1. INTRODUCTION

Collision avoidance is one of the major research topics in the field of mobile robotics. Many robotic applications, such as rescue, surveillance, and mining require mobile robots to explore an unknown environment without collision. For fully controlled robots, the collision avoidance task [1] can be achieved by a human operator who controls the robot by sending commands to the mobile robot via cable or wireless communication. However, this mode of operation is of limited use in hazardous environments where cable and wireless communication are unable to be set up. Thus, it is necessary for a mobile robot to navigate autonomously in some situations.

Generally, a global collision-free path from the current location of the robot to the goal position can be planned if an accurate map of the environment is provided. For example, many solutions to global path planning are based on the A* searching algorithm [2] or the visibility graph method [3], which rely on prior information of the environment. However, in the real world, an environment is not always static, and in a dynamic environment, robots need to take into account the unforeseen obstacles and moving objects. Local path planning or obstacle avoidance algorithms, such as the potential field method [4] and the dynamic window approach [5] can be utilized to avoid collisions in a dynamic environment. Under some circumstances, there is no map available in advance for a robot to implement the navigation task, which is known as mapless navigation. Optical flow based methods [6, 7] and appearance-based methods [8] are popular to solve these mapless navigation problems.

In this paper, we use Deep Reinforcement Learning (DRL) to solve the collision avoidance problem and aim to enable the mobile robot to avoid the obstacles without prior knowledge of the environment. DRL is a combination of Reinforcement Learning (RL) and Deep Learning (DL). In general, RL is a type of machine learning method that allows an agent to learn to act in an environment based on feedback rewards or punishments. To solve a simple RL problem, such as the FrozenLake environment [9] that contains sixteen possible states, the values for all the possible actions in each state are listed in a Q-table and can be updated according to the Bellmen equation. However, the Q-table is not suitable for the RL problems in many complex environments due to the nearly infinite number of states and possible actions. In these circumstances, neural networks were introduced as the nonlinear approximators to estimate the values for all the possible actions that extends the RL to DRL. A well-known recent application of DRL is AlphaGo Zero [10], a program

*Authors contributed equally; **Corresponding author – bentzvi@vt.edu

learned to play the game of Go by itself without any human knowledge and successfully won 100-0 against AlphaGo [11], which was the first program to defeat a world champion in the game of Go. Researchers are also exploring and applying RL in robotics [11, 12], and believe that RL has the potential to train a robot to discover an optimal behavior surpassing that taught by human beings in the future.

This work attempts to show the feasibility and practicality of solving collision avoidance problems with DRL. Our main contributions are listed as follows: 1) Successful application of DRL to solve the obstacle avoidance problem for mobile robots. 2) Development of a simulator in Gazebo as the 3D training environment for the obstacle avoidance approach to collect large-scale data to train the neural networks. The simulator can also be used as the testbed for the robot. 3) Implementation and comparison of the performances of the collision avoidance algorithms based on different DRL methods.

The rest of this paper is organized as follows: An overview of the background and related work is presented in Section 2. The implementation of the obstacle avoidance framework and simulator setup are described in Section 3. In Section 4, we show and discuss the results of the training and testing. Finally, concluding remarks are presented in Section 5.

2. BACKGROUND AND RELATED WORK

RL aims to enable an agent to learn the optimal behavior when interacting with the environment by means of trial-and-error search as well as delayed reward [14]. The main elements of an RL problem include an agent, environment states, a policy, a reward function, a value function, and an optional model of the environment. Typically, the interaction between the agent and the environment can be simply described as: at each time step t, the agent determines the current environment state s from all possible states S ($s \in S$), then it chooses an action a out of A ($a \in A$) according to the current policy $\pi(s,a)$. The policy can be considered as a map which shows the probabilities of taking each action a when in each state s. After taking the chosen action a, the agent is in a new state s' and receives a reward signal r. The whole process is depicted in Figure 1.

RL is highly influenced by the theory of Markov Decision Processes (MDPs). An RL task can be described as an MDP if it has a fully observable environment whose response depends only on the state and action at the last time step. A tuple $\langle S, A, P, R, \gamma \rangle$ can be utilized to represent an MDP, where S is a state space, A is an action space, P is a state transition model, R is a reward function and γ is a discount factor. Some extensions and generalizations of MDPs are capable of describing partially observable RL problems [15] or dealing with continuous-time problems [9, 10]. RL algorithms need to be capable of updating the policy due to the training experience and finally determine the policy that fixes one action for each state with the maximum reward. Several solutions to RL problems are presented in the following subsections.

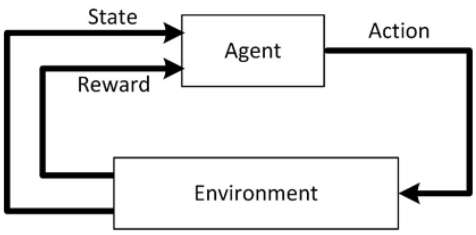

Figure 1. The interaction between the agent and the environment

2.1 Q-learning

Q-learning [18] is a basic and popular RL algorithm developed by Watkins in 1989. It is capable of finding an optimal policy for an MDP by updating the Q-values table $Q(s,a)$ based on Bellman Equation:

$$Q(s,a) = r + \gamma(\max_{a'}(Q(s',a'))). \quad (1)$$

Equation (1) can be decomposed into two parts: the immediate reward r and the discounted maximum Q-value of the successor state $\gamma(\max_{a'}(Q(s',a')))$, where γ is a discount factor, $\gamma \in [0,1]$, s' is the successor state and a' is the action to be taken to get the maximum Q-value in the state s'.

2.2 Deep Q-networks

Solving an RL problem using linear methods, such as the Bellman equation, can be simple and efficient, but not all situations are suitable for updating the action-value iteratively. DeepMind proposed a deep learning model known as DQN [19], which is a convolutional neural network trained with a variant of Q-learning algorithm with experience replay memory [20], represented as a data set D. The agent's experience at each time step can be stored in D, and a mini-batch of experiences are sampled randomly when performing updates.

The DQN was tested on several Atari games. In these cases, the game agent needs to interact with the Atari emulator whose states are represented as $s_t = x_1, a_1, x_2, a_2, ..., a_{t-1}, x_t$, where x_t is the image describing the current screen at each time step t and a_t is the chosen action. The states are preprocessed and converted to an $84 \times 84 \times 4$ image as the input of the neural network, defined as the Q-network, which has three hidden layers and a fully-connected linear output layer. The outputs correspond to the predicted Q-values for each valid action. A sequence of loss functions $L_i(\theta_i)$ are adopted to train the Q-network:

$$L_i(\theta_i) = (r + \gamma(\max_{a'}(Q(s',a';\theta_i^-)) - Q(s,a;\theta_i))^2 \quad (2)$$

where $r + \gamma(\max_{a'}\left(Q\left(s',a';\theta_i^-\right)\right)$ is the target for iteration i, and θ_i^- are the weights from the previous iteration which are fixed when optimizing the loss function by stochastic gradient descent. The gradient can be calculated by differentiating the loss function with respect to the weights as follows:

$$\nabla_{\theta_i} L_i(\theta_i) = (r + \gamma(\max_{a'}(Q(s',a';\theta_{i-1})) - Q(s,a;\theta_i))\nabla_{\theta_i} Q(s,a;\theta_i) \quad (3)$$

2.3 Double Deep Q-networks

One critical shortcoming of DQN is that it sometimes learns unrealistically high action values, known as overestimation. DDQN [21] is introduced to reduce overestimation by separating the action selection and action evaluation in the target. DDQN was also developed by DeepMind in 2015. The way that DDQN updates the target networks is the same as DQN but the target of DDQN differs, and can be expressed as follows,

$$y_i = r + \gamma Q(s', \max_{a'}(Q(s',a';\theta_i);\theta_i^-)). \quad (4)$$

It has two sets of weights, θ_i and θ_i^-. The action is chosen greedily from the network with weights θ_i at each step, but the Q-values assigned to that action is from the target network with weights θ_i^-. This method provides a more stable and reliable learning process which decreases the overestimation error.

2.4 Double Deep Q-networks Prioritized Experience Replay

Prioritized experience replay of Double Deep Q-networks changes the method from randomly sampling experience to selecting experience based on the priority of each experience stored in the replay memory. The priority value of each experience in the replay memory is calculated using temporal-difference (TD) error [22]:

$$\delta_i = r + \gamma Q(s', \max_{a'}(Q(s',a';\theta_i);\theta_i^-)) - Q(s,a) \quad (5)$$

where δ_i is the difference between the target value $y_i = r + \gamma Q(s', \max_{a'}(Q(s',a';\theta_i);\theta_i^-))$ and the Q-value $Q(s,a)$. The transitions from the current state to the next state with the largest δ_i are replayed from the memory, which is called pure greedy prioritization. In this case, the transitions with initially high error are replayed frequently. In order to avoid a lack of diversity, stochastic prioritization [22], which interpolates between pure greedy prioritization and uniform random sampling, is developed to sample the transitions. The probability of sampling transition i can be defined as:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \quad (6)$$

where p_i is the priority of transition i. α is a value within $[0,1]$, which determines the prioritization percentage used in stochastic prioritization. k indicates the size of the minibatch. Several approaches are available for evaluating the priority, for example, $p_i = |\delta_i| + \varepsilon$, where ε is a positive constant to make sure that p_i is not equal to zero in case of $|\delta_i| = 0$. This method is known as proportional prioritization, which converts the error to priority. Another method is known as rank-based prioritization. In this case, the priority of transition i is defined as: $p_i = \frac{1}{rank(i)}$, where $rank(i)$ is the rank of transition i based on the value of $|\delta_i|$.

The stochastic prioritization also leads to the bias of the estimated solution. To correct the bias, prioritized experience replay adds importance-sampling (IS) weights at each transition, which is expressed as [22],

$$\omega_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta \quad (7)$$

where N is the current size of the memory, $P(i)$ is the possibility of sampling transition i in equation (6), and β lies within the range of $[0,1]$.

2.5 Related Work

The work described in [23] applies the DQN to solve the obstacle avoidance problem for a mobile robot. It shows that the well-trained neural network is able to help a robot avoid obstacles in a simulation environment. However, the raw inputs from the Kinect sensor mounted on the robot platform need to be preprocessed by a supervised learning model to create feature maps for the neural network as the inputs, which requires more computing power and reduces the algorithmic efficiency. In addition, only three possible actions are available for the robot to choose according to the outputs of the neural network that was trained using the experiences gained from a simple corridor-like simulation environments. This limits the mobility and performance of the mobile robot and may lead to collisions when the robot is navigating in a complex environment.

Similarly, in [24], the obstacle avoidance method based on Asynchronous Deep Deterministic Policy Gradients (ADDPG) was successfully developed for mobile robots with a laser sensor. Their neural network can take raw sensor data as inputs without much preprocessing, but the simulation environments are still too simple to provide sufficient experiences for training the neural network.

3. COLLISION AVOIDANCE WITH DRL

This section describes the obstacle avoidance problem and its formulation. We introduce the implementation details of three different obstacle avoidance strategies based on DQN, DDQN, and DDQN-PER. The setup of the simulator is presented in this section as well.

3.1 Problem Formulation

This paper focuses on enabling differential drive robots to navigate safely in an environment with narrow paths, corners, and other types of obstacles. In general, the problem can be formulated as,

$$(v_t,\ \omega_t) = f(\mathbf{O}_t)\,. \tag{8}$$

At each time step t, the planner of the mobile robot needs to generate the required linear velocity v_t and angular velocity ω_t from the observation $\mathbf{O}_t$ of the environment. In our approach, no mathematical model is defined to map the input data that contains the environment information to the proper commands for the mobile robot. Instead, we consider the problem as an MDP and use DRL to solve it. All possible observations from the sensor data form the state space S, which is the input of the neural network that works as the nonlinear function approximator $Q(s,a;\theta)$. This estimates the values of all the possible actions a in each state s. When the robot is in state s_t at the current time step t, it is supposed to select the proper action a_t according to the action values provided by the neural network $Q(s,a;\theta_t)$ and an ε-greedy policy. Then the robot will receive a reward r_t after taking that action. These transitions $(s_t, a_t, s_{t+1}, r_{t+1})$ are recorded and will be used to update the parameters of the neural network. A large-scale interaction dataset is required to train the neural network. Thus, we developed a 3D simulator to provide the training and testing environment. A virtual Turtlebot with a laser sensor was put in that environment as the robot platform. To avoid a lack of generalization capability to different scenarios in the real world, the training map contains different features for the robot to gain as many different transitions as possible.

3.3 Simulator

A virtual world was built in Gazebo as the simulation environment, where a Turtlebot with a 2D laser scanner was trained to avoid the collisions. Figure 2 shows the complex

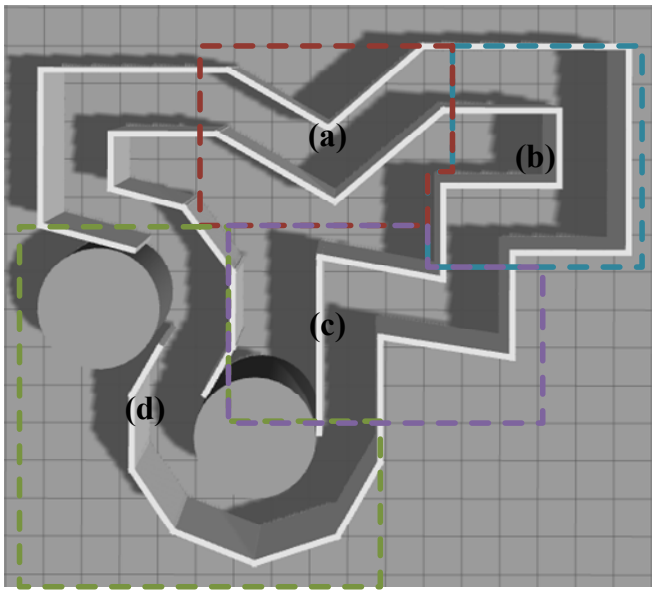

Figure 2. The training map with different wall shapes: (a) obtuse angle turn, (b) right angle turn, (c) acute angle turn, (d) and curve turn.

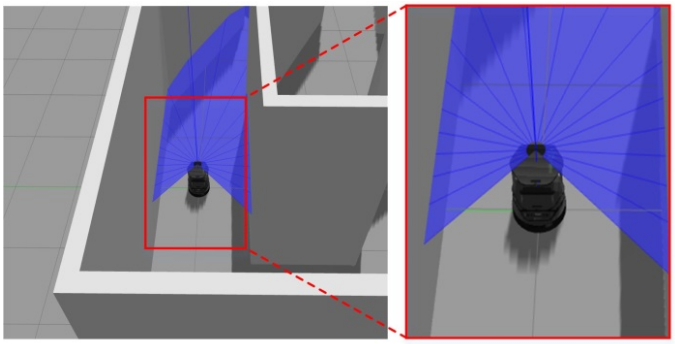

Figure 3. The virtual robot in the simulation environment.

circuit used with different environmental features such as straight tracks, 90-degree turns, and acute- as well as obtuse-angle corners in order to enrich the experiences of the robot during learning, and enable the model to generalize to more complicated situations after training. The Turtlebot is a differential wheeled mobile robot as shown in Figure 3. The blue area indicates the measurements of the 2D laser sensor used. No map is provided in advance for the Turtlebot to locate any obstacles and plan a collision-free path to travel. It needs to learn from scratch about how to behave in such an unknown environment.

3.4 Implementation Details

To implement the obstacle avoidance framework, we developed a software architecture in the publish-subscribe

pattern utilized the Robot Operating System (ROS). The neural network was embedded in a motion planner and implemented using Keras with Tensorflow as the backend. The data from the sensor were acquired and published in 100 Hz for the motion planner to subscribe. Then the commands, such as the linear and angular velocities from the motion planner can be sent to a low-level controller to control the speed.

The OpenAI Gym was used to implement our RL task. The gym environment needs to be set up in advance to define the components of RL tasks such as states, actions, and rewards. The data acquired from the sensor contains the distance values measured by 100 laser rays in a 270 degrees span. Fifty distance measurements are picked up evenly from the raw sensor data and converted into values, with a precision of up to two decimal digits in the range of 0.00m to 6.00m, to represent the current state s_t. Our action set consist of 11 possible actions and each action has two commands: one is to go forward with constant linear velocity at 0.6 m/s, the other is an angular velocity $\omega_m = -0.8 + 0.16 \times m \quad (m = 0,1,2,\cdots 10)$. The constant speed ensures the robot could only move forward instead of pure rotation. Backward motion is not taken into consideration in this work.

The immediate reward at each time step is assigned based upon the following equation,

$$r_t = \begin{cases} 5 & \text{(without collision)} \\ -1000 & \text{(collision)} \end{cases} . \qquad (9)$$

The mobile robot receives 5 points for each time step if it does not bump into the wall after taking the chosen action, otherwise it will receive a punishment of -1000 points.

After the setup of the gym environment, we implemented the neural network using Keras, a Python deep learning library. The inputs of the networks are the observations from the laser data which represent the states of the environment. Two hidden layers, each with 300 neurons are added to the network with ReLU as the activation function. The outputs should be the Q-values of the 11 actions. Different methods were used to update the parameters of the neural network and the implementation details are presented in Algorithm 1,2, and 3. Algorithm 1 and 2 use the same sampling method, a minibatch of transitions sampled randomly from the memory. The difference between these two algorithms is the target y_i to calculate the loss $L(\theta)$. As mentioned in Section 2.3, DDQN needs two sets of parameters to determine the Q-values of the actions to avoid overestimation. In Algorithm 3, a rank-based prioritization is utilized to sample the transitions to improve the efficiency of the learning process.

Each algorithm was ran for 3000 epochs with each containing at most 4000 time steps. Once the time step reaches the maximum value or the Turtlebot crashes into the wall, the simulation resets with the Turtlebot at a random initial position and a new epoch starts.

Algorithm 1: DQN

1. Initialize the Gazebo simulator;
   Initialize the memory D and the Q-network with random weight θ ;
2: **for** episode =1, k **do**
3: Put the visual robot at a random position in the 3D world;
   Get the first state s_1
4: **for** t = 1,T **do**
5: With probability ε select a random action a_t
6: Otherwise select $a_t = \max_a \left(Q(s,a;\theta_i)\right)$
7: Take action a_t ; get reward r_{t+1} and state s_{t+1}
8: Store transition $(s_t, a_t, s_{t+1}, r_{t+1})$ in D
9: Sample random mini-batch of transitions $(s_i, a_i, s_{i+1}, r_{i+1})$
10: **if** r_{i+1} = -1000 **then**
11: $y_i = r_{i+1}$
12: **else**
13: $y_i = r_{i+1} + \gamma \max(Q(s_{i+1}, a; \theta)$
14: **end if**
15: Calculate θ by perform mini-batch gradient descent on the Mini-batch of loss $L(\theta) = (y_i - Q(s,a;\theta))^2$
16: **end for**
17: **end for**

Algorithm 2: DDQN

1. Initialize the Gazebo simulator;
   Initialize the memory D and the Q-network with random weight θ ;
   Initialize the target Q-network with random weights θ^-
2: **for** episode =1, k **do**
3: Put the visual robot at a random position in the 3D world;
   Get the first state s_1
4: **for** t = 1,T **do**
5: With probability ε select a random action a_t
6: Otherwise select $a_t = \max_a \left(Q(s,a;\theta_i)\right)$
7: Take action a_t ; get reward r_{t+1} and state s_{t+1}
8: Store transition $(s_t, a_t, s_{t+1}, r_{t+1})$ in D
9: Sample random mini-batch of transitions $(s_i, a_i, s_{i+1}, r_{i+1})$
10: **if** r_{i+1} = -1000 **then**
11: $y_i = r_{i+1}$
12: **else**
13: $y_i = r_{i+1} + \gamma Q(s_{i+1}, \max(Q(s_{i+1}, a; \theta); \theta^-))$
14: **end if**
15: Calculate θ by perform mini-batch gradient descent on the Mini-batch of loss $L(\theta) = (y_i - Q(s,a;\theta))^2$
16: Replace the target network parameters $\theta^- \leftarrow \theta$ every N step
17: **end for**
18: **end for**

Algorithm 3: DDQN with prioritized replay

1. Initialize the Gazebo simulator;
   Initialize the memory D and the Q-network with random weight θ ;
   Initialize the target Q-network with random weights θ^{-}
2: **for** episode =1, k **do**
3: Put the visual robot at a random position in the 3D world;
   Get the first state s_1
4: **for** t = 1,T **do**
5: With probability ε select a random action a_t
6: Otherwise select $a_t = \max_a\left(Q(s,a;\theta_i)\right)$
7: Take action a_t ; get reward r_{t+1} and state s_{t+1}
8: Store transition $(s_t,a_t,s_{t+1},r_{t+1})$ in D
9: Sample mini-batch of transitions $(s_i,a_i,s_{i+1},r_{i+1})$ using rank-based prioritization
10: **if** r_{i+1} = -1000 **then**
11: $y_i = r_{i+1}$
12: **else**
13: $y_i = r_{i+1} + \gamma Q(s_{i+1}, \max(Q(s_{i+1},a;\theta);\theta^{-}))$
14: **end if**
15: Calculate θ by perform mini-batch gradient descent on the Mini-batch of loss $L(\theta_i) = (y_i - Q(s,a;\theta_i))^2$
16: Replace the target network parameters $\theta^{-} \leftarrow \theta$ every N step
17: **end for**
18: **end for**

3.4 Validation in Simulator

Two tests were set up to evaluate the performances of the virtual robot with the well-trained neural networks. Test 1 was set up as follows:

1. Run the program for three times with different training methods: DQN, DDQN, DDQN-PER respectively and obtain three different trained neural networks.

2. Apply the three neural networks to the Turtlebot successively and let the robot navigate in the training map for five minutes with each of the networks. The metric for evaluation of the performances of the neural networks is chosen as the number of collisions undergone within the five minutes of simulation.

To prove that the proposed approach can help the robot navigate an unknown environment, we introduced Test 2:

1. Place the virtual robot with each neural network in two test maps different from the training map. The detailed information about the maps is shown in Figure 4.

2. Test the robot with each neural network for five minutes with the different test maps and record the number of collisions. The virtual robot will be put in a random location with different orientations on the map. After the robot travels for 1000 time steps without collisions, the robot will be spawned to another random location. This is to demonstrate that the proposed obstacle avoidance approach is robust and capable of navigating an unknown environment regardless of the starting position and the heading. Once the robot is too close to the wall, the algorithm will stop the virtual Turtlebot and reset it at another random location.

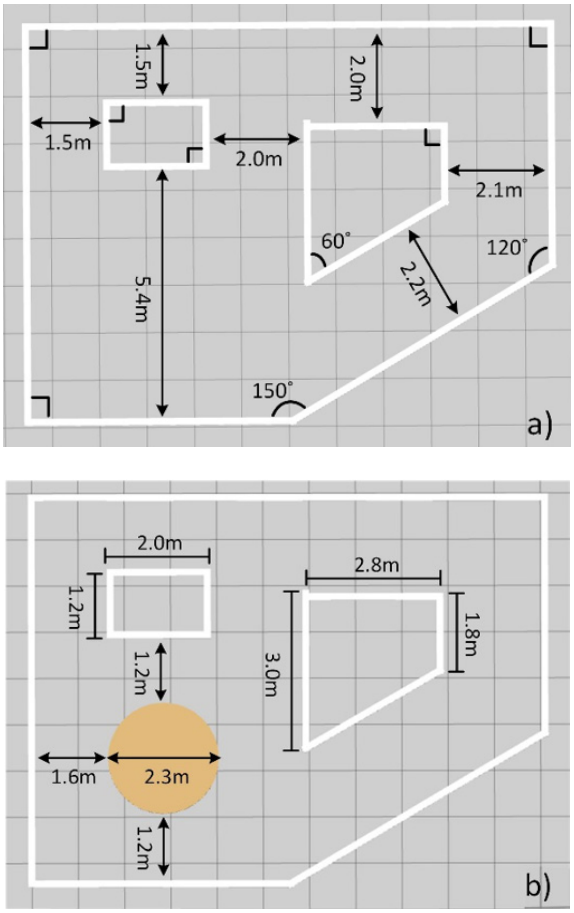

Figure 4. Diagrams of (a) test map 1, and (b) test map 2.

4. RESULTS

4.1 Training Results

At the beginning of the training, the Turtlebot has no previous knowledge about the environment and insufficient experiences are available for it to learn from, and as a result, it keeps rushing to the walls. After training, the Turtlebot is able to avoid the collisions and accidents rarely happened as before, which shows that the Turtlebot managed to learn how to avoid obstacles gradually. Figure 5 shows an example in which the well-trained policy guided the Turtlebot to navigate in the training map successfully without any collision. At each time step, the observation from the sensor data was fed into the well-trained neural network. The robot then took the proper action according to the output of the neural network. This proved that DRL has the ability to solve the obstacle avoidance problem.

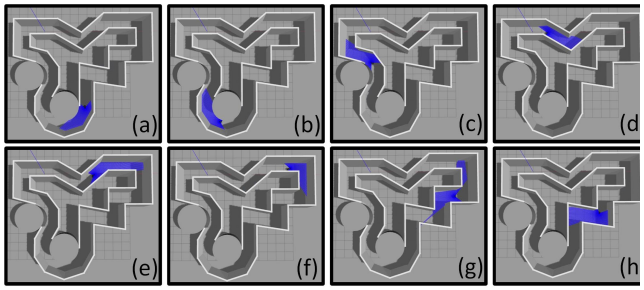

Figure 5. Navigation performance of the well-trained policy in the training map. The blue area indicates the laser rays.

To evaluate the three algorithms, we recorded the average Q-values and accumulated rewards from the training. A moving average filter with a window size of 500 is utilized to process the data in order to identify the trends clearly in the data. The plots in Figure 6 show that the learning curves tend to converge in all three algorithms, which means that these algorithms can achieve the goal of determining an optimal policy. We first compared the average Q-values and cumulated rewards from Algorithm 1 and 2 as shown in the first two rows of Figure 4. The average Q-values estimated by DQN were higher but the rewards dropped due to the overestimation. In addition, the learning curve of DDQN is smoother than that of DQN, which means that learning with DDQN is more stable. The last plot in

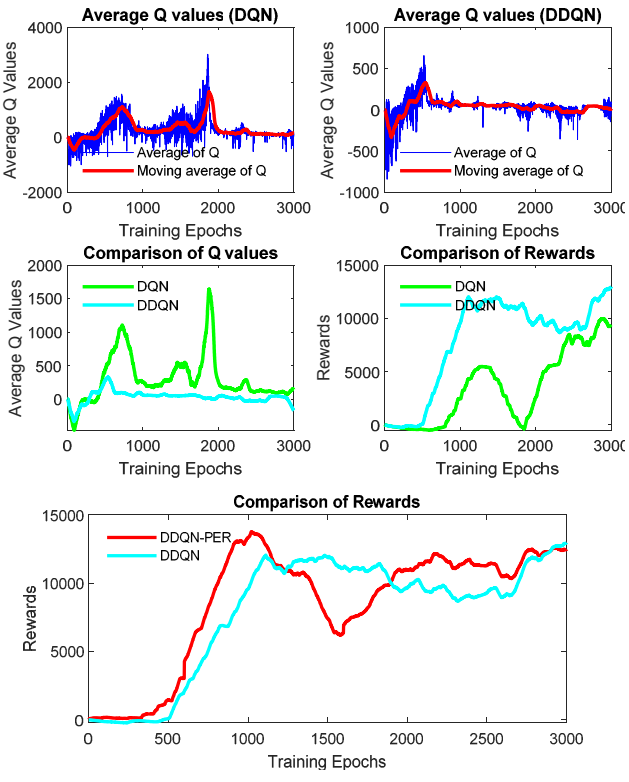

Figure 6. Average q-values estimated by DQN, DDQN and the comparison of rewards.

Fig. 6 shows a comparison between DDQN-PER and DDQN. It is evident that the reward curve of DDQN-PER converge faster than DDQN. This proved that training a neural network with prioritized replay in our obstacle avoidance problem can be more efficient.

In other words, Turtlebot is able to learn to improve its behavior by interacting with the environment using DRL, which indicates that unsupervised training of a model has the potential to solve the obstacle avoidance problems in a complicated environment.

4.2 Validation of Results

Figure 7 shows an instance of the navigation in the test environment using the well-trained policy and the Turtlebot could succeed in navigating around the obstacles. However, the Turtlebot could be trapped into rotating around in the large open area instead of exploring the whole environment. This is because we investigated the possibility and performance of DRL based on pure obstacle avoidance method and no target position is considered in this work. Table 1 shows the performance results from Test 1, where the robot ran with different neural networks over a duration of five minutes. The results demonstrate that the neural networks trained with DDQN and DDQN-PER provide better performance as they lead the robot to less collisions.

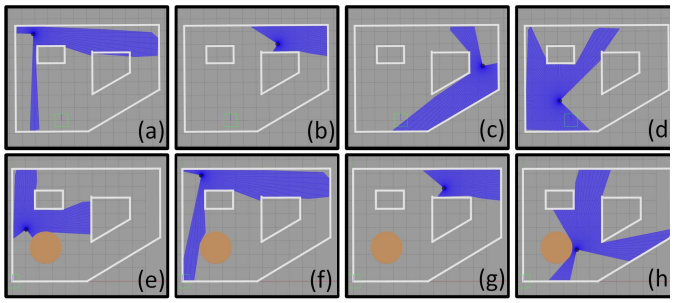

Figure 7. Navigation performance of the well-trained policy in the testing map. The blue area shows the measurements of the laser sensor.

TABLE 1
TEST 1: RESULTS

Algorithms	NUMBER OF COLLISIONS IN 5 MIN
DQN	3
DDQN	1
DDQN-PER	1

The results from Test 2 are shown in Table 2. The robot performance is recorded as unsatisfactory in Test Map 1 although the map is significantly simpler in comparison to Test Map 2. This is because the robot lacks experience in large open areas. This underlines the significance of the experiences a robot acquires during training and demonstrates how critical they are for the superior performance of the robot in an

TABLE 2
TEST 2: RESULTS

Maps	NUMBER OF COLLISIONS IN 5 MIN		
	DQN	DDQN	DDQN-PER
Test Map 1	8	6	6
Test Map 2	2	0	1

unknown environment. As with all DRL approaches, the proposed obstacle avoidance framework faces the limitation of real-world transference. The robot may fail to navigate on a simple map in the real world, if the features in that map are entirely different from that used in the training environment. It is expected that this problem can be solved and compensated for by training the robot in more environments on top of the experiences it has already learned. We will discuss the feasibility of this approach in our future work.

5. CONCLUSION

In this work, a virtual Turtlebot is trained to avoid obstacles in a Gazebo simulator with different optimal training methods. Three different obstacle avoidance algorithms based on DRL are implemented and evaluated in this paper. Through simulations, we demonstrated that the robot has the ability to avoid collisions after training for 3000 epochs with all three algorithms. We plan to verify the feasibility and reliability of the policies through experimentation with a physical robot in the future. Furthermore, while only one robot is considered in this paper and no specific goal position is set for the robot for navigation, we are currently exploring goal-oriented navigation and multi-robot cooperation in our future work.

REFERENCES

[1] P. Ben-Tzvi, "Experimental validation and field performance metrics of a hybrid mobile robot mechanism," *J. F. Robot.*, vol. 27, no. 3, pp. 250–267, 2010.

[2] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, 1968.

[3] N. J. Nilsson, "A Mobile Automation: an Application of Artificial Intelligence Techniques," *Int. Jt. Conf. Artif. Intell.*, vol. 1969, no. January, pp. 509–520, 1969.

[4] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," in *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, 1985, vol. 2, pp. 500–505.

[5] O. Brock and O. Khatib, "High-speed navigation using the global dynamic window approach," *Proc. 1999 IEEE Int. Conf. Robot. Autom. (Cat. No.99CH36288C)*, vol. 1, no. May, pp. 341–346, 1999.

[6] A. Giachetti, M. Campani, and V. Torre, "The use of optical flow for road navigation," *IEEE Trans. Robot. Autom.*, vol. 14, no. 1, pp. 34–48, 1998.

[7] K. Souhila and a Karim, "Optical flow based robot obstacle avoidance," *Int. J. Adv. Robot. Syst.*, vol. 4, no. 1, pp. 13–16, 2007.

[8] N. Ohnishi and A. Imiya, "Appearance-based navigation and homing for autonomous mobile robot," *Image Vis. Comput.*, vol. 31, no. 6–7, pp. 511–532, 2013.

[9] "FrozenLake Environment." [Online]. Available: https://gym.openai.com/envs/FrozenLake-v0/. [Accessed: 11-Mar-2018].

[10] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[11] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[12] K. Mülling, J. Kober, O. Kroemer, and J. Peters, "Learning to select and generalize striking movements in robot table tennis," *Int. J. Rob. Res.*, vol. 32, no. 3, pp. 263–279, 2013.

[13] A. Y. Ng *et al.*, "Autonomous inverted helicopter flight via reinforcement earning," *Springer Tracts Adv. Robot.*, vol. 21, pp. 363–372, 2006.

[14] R. S. Sutton and A. G. Barto, "Chapter 1 Introduction," *Reinf. Learn. An Introd.*, 1988.

[15] T. Jaakkola, S. P. Singh, and M. I. Jordan, "Reinforcement learning algorithm for partially observable Markov decision problems," *Adv. Neural Inf. Process. Syst.*, vol. 7, p. 345, 1994.

[16] K. Doya, "Reinforcement Learning in Continuous Time and Space," *Neural Comput.*, vol. 12, no. 1, pp. 219–245, Jan. 2000.

[17] S. J. Bradtke and M. O. Duff, "Reinforcement Learning Methods for Continuous-Time Markov Decision Problems," *Adv. Neural Inf. Process. Syst.*, pp. 393--400, 1994.

[18] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, no. 3–4, pp. 279–292, 1992.

[19] Y. Zhan, H. B. Ammar, and M. E. Taylor, "Theoretically-grounded policy advice from multiple teachers in reinforcement learning settings with applications to negative transfer," in *IJCAI International Joint Conference on Artificial Intelligence*, 2016, vol. 2016–Janua, pp. 2315–2321.

[20] L. Lin, "Reinforcement learning for robots using neural networks," 1993.

[21] W. Chen, M. Zhang, Y. Zhang, and X. Duan, "Exploiting meta features for dependency parsing and part-of-speech tagging," *Artif. Intell.*, vol. 230, pp. 173–191, Jan. 2016.

[22] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," pp. 1–21, 2015.

[23] T. Lei and L. Ming, "A robot exploration strategy based on Q-learning network," *2016 IEEE Int. Conf. Real-Time Comput. Robot. RCAR 2016*, pp. 57–62, 2016.

[24] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," *IEEE Int. Conf. Intell. Robot. Syst.*, vol. 2017–September, pp. 31–36, 2017.