Obstacle Avoidance using Reinforcement Learning

Shravan S Rai
Robotics and Autonomous Systems
Arizona State University

Sivanaga Surya Vamsi Popuri
Electrical and Computer Engineering
Arizona State University

Varadaraya Ganesh Shenoy
Electrical and Computer Engineering
Arizona State University

***Abstract*—Learning to navigate in an unknown environment is a crucial capability of mobile robot. Conventional method for robot navigation consists of three steps, involving localization, map building and path planning. These methods becomes obsolete when the environment is unknown because they require an obstacle map for path planning. Also, classical approaches can get tedious and can get stuck at local optima as the environment gets more complex. In autonomous navigation collision avoidance is also a key area to be addressed. Navigating the robot safely to the target without any collision is extremely significant in both static and dynamic environments. Heuristic approaches are gaining prominence nowadays because of its closeness to human way of behavioral learning. In this project, we propose using the Deep Deterministic Policy Gradient Reinforcement learning method to navigate the robot from the start location to the target location without collisions with static obstacles.**

***Index Terms*—Obstacle Avoidance, Reinforcement Learning,DDPG**

I. Problem Statement

This project studies the application of Reinforcement learning techniques to navigate a mobile robot from a desired spawning location to a desired goal location in an environment which possesses static obstacles. The environment is assumed to a Markov Decison Process and has states and rewards corresponding to every state. The agent or mobile robot tries to navigate from a start position to a goal position by collecting maximum reward and also trying to avoid the static obstacles. The states of the environment that the robot observes are the goal position,its own position, its linear and angular velocities and also distance from the obstacles. For every action, the robot takes it gets a small negative reward so that is compelled to move to next state, it gets a large negative reward if it collides with the obstacles. If it reaches the goal, it gets a large positive reward. For the robot to move faster, a reward component of its velocity can also used.

II. Introduction

Applications of autonomous vehicles stress on safety, while managing navigation in environments with obstacles. As a result of which , there arises a need for trajectory planning and real-time collision avoidance. Several available strategies are computationally heavy. Path planning with model free controllers [1] although light in computation , is conservative. Model based methods, such as Model Predictive Control [3], incorporates slow global planning with fast local avoidance [5], avoidance via search of a motion primitive library [2] is intensive in calculation. According to [7] These methods have limited scope, taking into account dynamics-specific model. Furthermore, estimation of obstacle positions from sensor is necessary for these models. In this project, such issues are looked at with a novel learning algorithm, schematically summarized in Fig.1,[7] that produces control commands from sensor inputs.

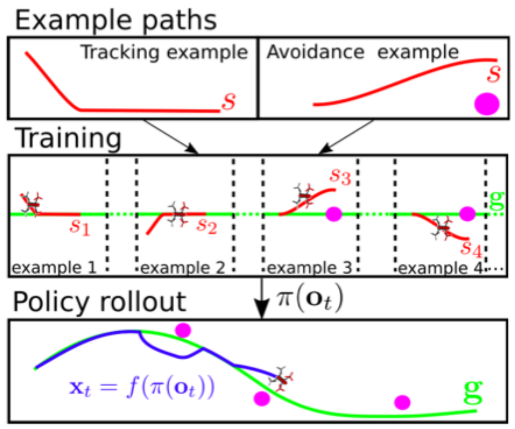

Fig. 1: A policy is learned form few, short local collision avoidance and path following maneuvers (red). The learned policy generalizes to unseen scenes and can track long guidance paths (green) through complex environments while successfully avoiding obstacles (blue).

As the control is produced from sensors directly,the algorithm does not require estimation of obstacle positions.

In addition, neural networks, is more efficient compared to traditional methods. Learning can be combined with motion planning.

The most general approach, to learn a control policy, is model-free reinforcement learning [4], a class of methods that learns the control policy through interaction with the environment.

III. Methods

A. *Q-Learning*

Q-learning is a model-free reinforcement learning algorithm to learn a policy telling an agent what action to take under what circumstances. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.For any finite Markov decision process(FMDP) , Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state.Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy. "Q" names the function that returns the reward used to provide the reinforcement and can be said to stand for the "quality" of an action taken in a given state.[14]
The weight for a step from a state δ_t steps into the future is calculated as $\gamma^{\delta t}$, where the discount factor,γ which lies between 0 and 1 has the effect of valuing rewards received earlier higher than those received later thus, reflecting the value of a "good start").γ may also be interpreted as the probability to succeed at every δt.
Thus, the algorithm can be approximated as shown in eq(1).[14]

$$Q{:}S \times A \rightarrow \mathbb{R} \tag{1}$$

The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information as depicted in the below equation.[14]

$$\begin{aligned} Q^{new}(s_t, a_t) \leftarrow Q^{old}(s_t, a_t) + \alpha(r_t + \\ \gamma \max_a Q^{optimal}(s_{t+1}, a) - Q^{old}(s_t, a_t)) \end{aligned} \tag{2}$$

An episode of the algorithm ends when state s_{t+1} is a final or terminal state. However, Q-learning can also learn in non-episodic tasks.Since Q-learning is an iterative algorithm, it implicitly assumes an initial condition before the first update occurs. High initial values, also known as "optimistic initial conditions", can encourage exploration, no matter what action is selected, the update rule will cause it to have lower values than the other alternative, thus increasing their choice probability. The first reward r can be used to reset the initial conditions. [14]

Although for problems with discrete action space and smaller number of states, the Q learning algorithm works very well. However, for problems like navigation, balancing an inverted pendulum on a cart which have continuous action space, the Q learning algorithm is not feasible. This is because in continuous spaces finding the greedy policy requires an optimization of actions a_t at every time-step t. This optimization is too slow to be practical with large, unconstrained function approximators and nontrivial action spaces.

B. *Deep Q-Learning*

Q-learning is a simple yet quite powerful algorithm to create a cheat sheet for our agent. This helps the agent figure out exactly which action to perform.[11] It is pretty clear that we can't infer the Q-value of new states from already explored states. This presents two problems:

- First, the amount of memory required to save and update that table would increase as the number of states increases
- Second, the amount of time required to explore each state to create the required Q-table would be unrealistic

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. [11] The algorithm includes the following steps in sequence :

1) All the past experience is stored by the user in memory
2) The next action is determined by the maximum output of the Q-network
3) The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem. Going back to the Q-value update equation derived from the Bellman equation. We have:

$$\boxed{(r_{t+1} + \gamma \max_a Q^{optimal}(s_{t+1}, a)} \tag{3}$$

Eq(3) represents the target. We can argue that it is predicting its own value, but since R is the unbiased true reward, the network is going to update its gradient using backpropagation to finally converge. Deep Q-Learning is depicted in the below figure :

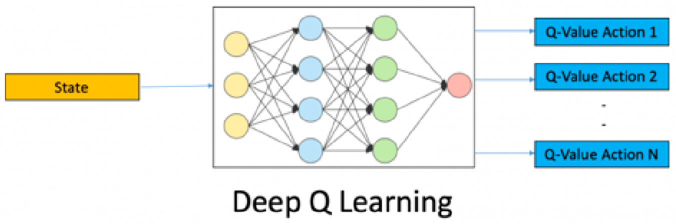

Fig. 2: Pictorial Representation of Deep Q-Learning

C. *Deep Deterministic Policy Gradient*

Out of the numerous reinforcement learning algorithms available, Deep Deterministic Policy Gradient (DDPG) is best suited for continuous action space tasks. DDPG concurrently learns a Q-function and a policy μ. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy. This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $Q^*(s,a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving

$$a^*(s) = \arg\max_a Q^*(s,a). \tag{4}$$

DDPG interleaves learning an approximator to $Q^*(s,a)$ with learning an approximator to $a^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces. When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem would make calculating $\max_a Q^*(s,a)$ a painfully expensive subroutine.

Because the action space is continuous, the function $Q^*(s,a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s,a)$, we can approximate it with $\max_a Q(s,a) \approx Q(s,\mu(s))$.

DDPG consists of two types of Networks called the actor and the critic network. The actor network takes the environment states as its input and predicts the actions the agent will perform in the next time step, whereas the critic network is used for evaluating the policy function estimated by the actor according to the temporal difference (TD) error. This combination of actor network and critic network along with experience replay constitutes Deep Deterministic Policy Gradient algorithm (DDPG).

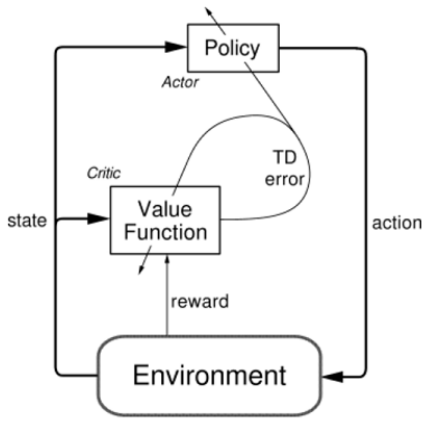

Fig. 3: DDPG Architecture

From [9] , it is clear that the DPG algorithm maintains a parameterized actor function $\mu(s \mid \theta^\mu)$ which specifies the current policy by deterministically mapping states to a specific action. The critic Q(s, a) is learned using the Bellman equation as in Q-learning. The actor is updated by following the applying the chain rule to the expected return from the start distribution J with respect to the actor parameters [9]:

$$\begin{aligned}\nabla_{\theta^\mu} J = E_{st\rho^\beta}[\nabla_a Q(s,a \mid \theta^Q) \mid_{s=s_t,a=\mu_t} \times \\ \nabla_{\theta_\mu}\mu(s \mid \theta^\mu) \mid_{s=s_t}]\end{aligned} \tag{5}$$

D. *Deep Deterministic Policy Gradient with Hindsight Experience Replay*

[6] Experience replay was first introduced by Lin (1992). The key idea of experience replay is to train the agent with the transitions sampled from the buffer of previously experienced transitions. A transition is defined to be a quadruple (s,a,r,s), where s is the state, a is the action, r is the received reward after executing the action a in the state s and s is the next state. At each time step, the current transition is added to the replay buffer and some transitions are sampled from the replay buffer to train the agent.

[10] HER is an experience replay method which can be used to overcome the learning difficulties caused by the use of sparse rewards and avoid complex reward projects. Different from the traditional RL methods, HER is proposed with a new parameter goal which consists

of desired goal and achieved goal. The desired goal represents the task that the agent should accomplish. The achieved goal represents the task that the agent has completed at the current time. The key idea of HER is as follows: at some moment, although the agent has not achieved the desired goal, it has completed achieved goal. At this time, the desired goal can be replaced by the achieved goal so that the method can transform failed experiences into successful experiences and learn from them. In HER, even though the desired goal is not completed at present, if the learning process is repeated, the agent will complete the desired goal in the final so as to complete the task with the sparse rewards.

[12] Suppose our agent performs an episode of trying to reach goal state G from initial state S, but fails to do so and ends up in some state S' at the end of the episode. We cache the trajectory into our replay buffer:

$$\begin{aligned}(S_0, G, a_0, r_0, S_1), (S_1, G, a_1, r_1, S_2), ...,\\ (S_n, G, a_n, r_n, S')\end{aligned} \tag{6}$$

where r with subscript k is the reward received at step k of the episode, and a with subscript k is the action taken at step k of the episode. The idea in HER is to imagine that our goal has actually been S' all along, and that in this alternative reality our agent has reached the goal successfully and got the positive reward for doing so. So, in addition to caching the real trajectory as seen before, we also cache the following trajectory:

$$\begin{aligned}(S_0, S', a_0, r_0, S_1), (S_1, S', a_1, r_1, S_2), ...,\\ (S_n, S', a_n, r_n, S')\end{aligned} \tag{7}$$

This trajectory is the imagined one, and is motivated by the human ability to learn useful things from failed attempts. It should also be noted that in the imagined trajectory, the reward received at the final step of the episode is now a positive reward gained from reaching the imagined goal. By introducing the imagined trajectories to our replay buffer, we ensure that no matter how bad our policy is, it will always have some positive rewards to learn from.

IV. Implementation

In this project, we are using a Simulation tool named CoppeliaSim (previously known as v-rep) by Coppelia Robotics along with OpenAI's gym environment [13]. The Coppelia Sim provides mobile robot and the scene in which the robot moves around. This when coupled with gym environment, we get the states and information about the environment without any additional sensors.The environment looks like the Fig:4.

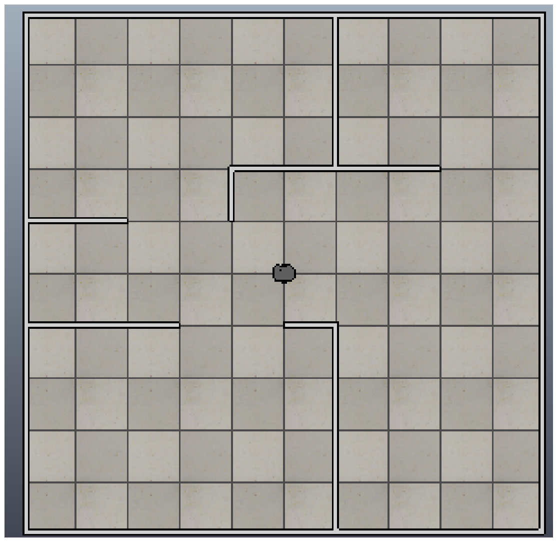

Fig. 4: Simulation Environment

To get the states and information about the environment, we have used 'PyRep'[8] which is a toolkit for robot learning research, built on top of CoppeliaSim and helps in interfacing with OpenAI gym. To build the neural networks for the actor and the critic we are using Tensorflow, because of its ease of use and powerful training methods.We have 4 networks in total, the actor, the critci, the target actor and target critic. For all the four networks we employ shallow neural networks each with 2 hidden layers and each layer have 200 and 100 neurons respectively. The activation function used is the relu function which is most effective in training. Both the actor and the critic networks are trained by the Adam optimizer with the aim of reducing the mean square error of the output between the critic and the target critic. For effective training of the agent, a good reward functions needs to be formulated which takes into account the variables that affect the performance of the robot. We formulated a reward function considering the all the important aspects and the reward equation is given in Eq(8).

$$R = \begin{cases} 1, & d < d_{th} \\ -1, & \exists D < d_{collision} \\ -0.1, & \exists D < d_{proxth} \\ V_L \cos(\theta), & otherwise \end{cases} \tag{8}$$

where, V_L is the linear velocity of the mobile robot, θ is the heading angle, D is the vector od distance read from the ultrasonic sensor, d is the distance between robot and goal position,$d_{collision}$distance when collision occurs, d_{th} threshold distance between robot and goal position, d_{proxth} is the safety distance threshold for regular operation.

V. Results

The Coppelia robotic simulator has been used for implementing the navigation of the robot in a 2-D environment.In order to navigate, the virtual robot uses multiple ultrasonic sensors to detect obstacles. The field of detection of the sensors together is shown in the Fig6. As can be seen, the field of detection is wide enough for the robot to sense the presence of multiple obstacles within its range, and this assists the robot to avoid them. The red arrows in the figure indicate presence of an obstacle in close proximity, while the blue lines indicate a clear path for the robot to take.

We conducted experiments with multiple start points and end points. We used 15 pairs and trained for each pair for 5000 episodes. Initially we tried with simple reward functions which penalized the robot for collision, and gave a reward for reaching the goal. But this was not a good reward formulation and finally we arrived at the above mentioned reward which took into account multiple factors. After training for 75000 episodes, we ran it for 100 episodes for evaluation and the results we got are as shown in fig:8 and fig:9. We achieved success well above 50% during the evaluation.

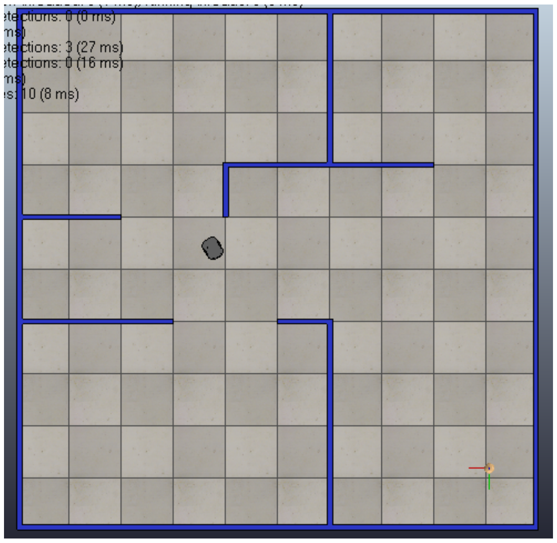

Fig. 5: Bot Origin

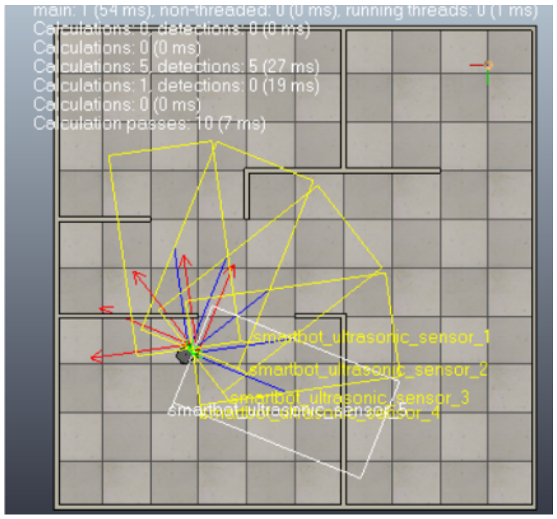
main: 1 (54 ms), non-threaded: 0 (0 ms), running threads: 0 (1 ms)
Calculations: 0 (0 ms)
Calculations: 5, detections: 5 (27 ms)
Calculations: 1, detections: 0 (19 ms)
Calculations: 0 (0 ms)
Calculation passes: 10 (7 ms)
smartbot_ultrasonic_sensor_1
smartbot_ultrasonic_sensor_2
smartbot_ultrasonic_sensor_3
smartbot_ultrasonic_sensor_4
smartbot_ultrasonic_sensor_5

Fig. 6: Gathering Local Information

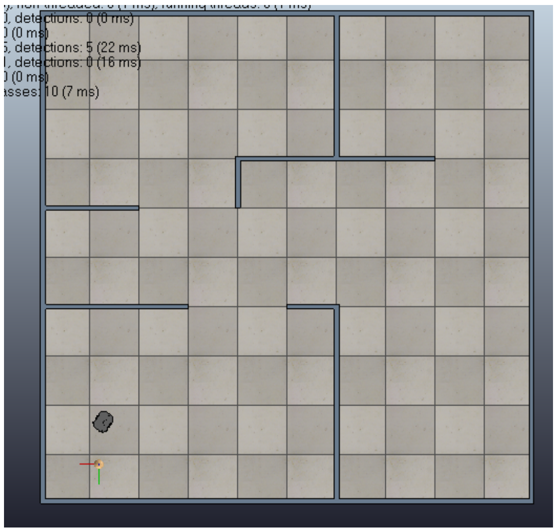

Fig. 7: Final Bot Position

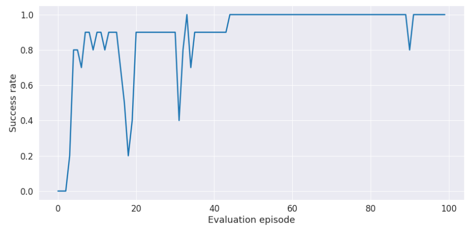

Fig. 8: Success Rate

VI. Conclusions and Future Work

As a result of this project , it can be concluded that Deep Deterministic Policy Gradient works quite well in the simulation environment. Although , it fails in initial evaluation episodes it is quick enough to recover and achieve a higher success rate in the later episodes. As an extension of this project, the next step would be to incorporate Hindsight Experience Replay to learn from the previous transitioning experience to avoid complex reward projects from occurring. Also, to have a better

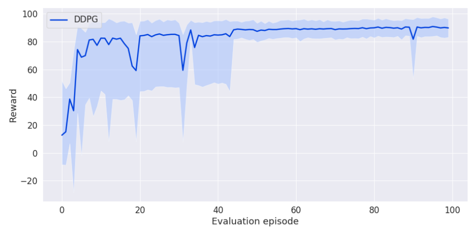

Fig. 9: Reward vs Evaluation Episode

understanding of the environment, we can incorporate vision sensors for better perception by the robot and hence leading to improved navigation strategies.

References

[1] G. Grisetti S. Grzonka and W. Burgard. "A fully autonomous indoor quadrotor," in: *IEEE Transactions on Robotics* 28 (2012), pp. 90–100.

[2] D. Mellinger M. Pivtoraiko and V. Kumar. "Incremental micro-uav motion replanning for exploring unknown environments". In: *IEEE International Conference on Robotics and Automation (ICRA), 2013* (2013), pp. 2452–2458.

[3] M. W. Mueller and R. D'Andrea. "A model predictive controller for quadrocopter state interception". In: *Control Conference (ECC), 2013 European* (2013), pp. 1383–1389.

[4] D. Silver A. Graves I. Antonoglou D. Wierstra V. Mnih K. Kavukcuoglu and M. Riedmiller. "Playing atari with deep reinforcement learning," in: *arXiv preprint arXiv:1312.5602, 2013.* (2013).

[5] Z. Taylor J. Nieto R. Siegwart H. Oleynikova M. Burri and E. Galceran. "Continuous-time trajectory optimization for online uav replanning". In: *International Conference on Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ* (2016), pp. 5332–5339.

[6] Richard S. Sutton Shangtong Zhang. "A Deeper Look at Experience Replay". In: *NIPS 2017 Deep Reinforcement Learning Symposium* (2017).

[7] Javier Alonso-Mora Otmar Hilliges Stefan Stevsic Tobias Nageli. "Sample Efficient Learning of Path Following and Obstacle Avoidance Behavior for Quadrotors". In: *IEEE Robotics and Automation Letters* 3.4 (2018). DOI: 10.1109/LRA.2018.2856922.

[8] Stephen James, Marc Freese, and Andrew J. Davison. "PyRep: Bringing V-REP to Deep Robot Learning". In: *arXiv preprint arXiv:1906.11176* (2019).

[9] Alexander Pritzel Nicolas Heess Tom Erez Yuval Tassa David Silver Daan Wierstra Timothy P. Lillicrap Jonathan J. Hunt. "CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING". In: *Google Deepmind* (2019).

[10] Jiahao Lu Guoyu Zuo Qishen Zhao and Jiangeng Li. "Efficient hindsight reinforcement learning using demonstrations for robotic tasks with sparse rewards". In: *International Journal of Advanced Robotic Systems* (2020).

[11] Ankit Choudhary. *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python*. URL: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

[12] Or Rivlin. *Reinforcement Learning with Hindsight Experience Replay*. URL: https://towardsdatascience.com/reinforcement-learning-with-hindsight-experience-replay-1fee5704f2f8.

[13] Souphis. *Open-AI Gym extension for robotics based on V-REP*. URL: https://github.com/Souphis/gym-vrep.

[14] Wikipedia. *Q-learning*. URL: https://en.wikipedia.org/wiki/Q-learning.