Path Planning Amidst Moving Obstacles

Hifza Khalid, Jong Seo Yoon

1 Project Overview

In this project, we aim to deal with the path finding problem in the presence of moving obstacles using deep reinforcement learning. A path finding algorithm computes path around static obstacles, however, what if the obstacles move? What if by the time the unit reaches a particular point, the obstacle that was initially obstructing its path is no more there and this new path is way longer than the previous chosen path [6]. Moreover, depending upon our world and the algorithm used, the agent might have to recalculate the new path which would introduce an additional computation overhead [6]. Therefore, for such scenarios, we would like our unit to have the additional capability to slow down or stop when it encounters an obstacle.

For this project, we plan to use neural networks with reinforcement learning [3]. Neural networks can be described as an agent that learns to map state-action pairs to rewards. It uses coefficients to approximate the function relating inputs to outputs, and its learning consists of finding the right coefficients, or weights, by iteratively adjusting those weights along gradients that promise less error[1]. Convolutional neural networks can be used to recognize an agent's state; e.g. the area or screen that our unit is on[7]. Neural networks seem a good fit to solve our problem because our state space and action space are continuous or high dimensional.

Our aims for this project can be broadly classified as:

1. Aim 1: Learn how to use and combine neural networks with reinforcement learning.
2. Aim 2: Simulate Turtlebot navigation using deep reinforcement learning in Robot Operating System (ROS)[8].
3. Aim 3: Teach the Turtlebot how to navigate around the environment with moving obstacles from a starting position to the goal using the optimal path.

2 Background and Related Work

The paper by group of people at Deepmind[5] proposes a method to combine reinforcement learning methods to neural networks by incorporating preprocessed frames as input and possible actions as outputs, called Deterministic Policy Gradients. Preprocessed frames in terms of reinforcement learning is the state the agent is in, and convolutional neural network calculates possible actions for the agent to choose in the output layer. We can use similar approach for our project, where turtlebot's *costmap_2d*'s frame is the state, and is fed to the convolutional neural network to output possible actions for the turtlebot to choose from.

However, we have a problem regarding continuous action spaces. The number of actions possible for the Turtlebot to decide whether to wait, slow down, or move quickly around the moving obstacle deals with continuous action spaces. Most robotic control problems fall into this category, where the agent is not able to converge to optimal policy due to the continuous action spaces [4]. We could discretize the continuous action space of the Turtlebot by limiting the number of actions possible by the Turtlebot, but that leads to the curse of dimensionality problem, where the dimension space grows exponentially as we add more features[5].

Fortunately, DeepMind has devised an algorithm on top of their Deterministic Policy Gradients in this paper[4] to solve the continuous action spaces problem. The algorithm is called Deep Deterministic Policy Gradients(DDPG), which is an off-policy and model-free algorithm that uses Deep-Q-Networks and operates on the estimation of Q-value. DDPG allows us to train our agent in a continuous action and state space, and we will incorporate this into the Turtlebot to investigate the performance of agent's path planning in the presence of moving obstacles in the environment.

3 Problem Formulation and Technical Approach

3.1 Problem Formulation

We formulate this problem on the simulator built on ROS[8] environment in order to explore different behaviors the agent may take and also to minimize unexpected errors. Moreover, we will use maps provided by ROS which already contains necessary parameters for our agent to move around the map. Moving obstacles with different moving velocities will be used to test our agent performance. Turtlebot, which is our agent, produces frames on each time step which maps to states from the reinforcement learning's perspective. Because DDPG algorithm directly maps the state to the action via Deep Q Network strategy, we can formulate the translation function as the following:

$$v_t = f(x_t, d_t, v_{t-1})$$

where x_t is the current state of the *costmap_2d*, p_t is the destination position which the agent must reach, and v_{t-1} is the velocity of the agent in the last time step.

3.2 State Space

State space will be visualized via ROS's *costmap_2d*, which is a continuous state space as frames are being fed to the network. However, we will set a certain time interval after which the state gets updated, as feeding every frame into the network generates unnecessary bottleneck in performance. We are also thinking about using sensor information provided by the agent, such as lasers, for input layer of the network.

3.3 Action Space

Each action for the Turtlebot is a vector that involves linear and angular velocity at each time step. This vector allows the Turtlebot to move in any direction, slow down and stop when needed. This formulates into continuous action space, which DDPG should be able to handle correctly.

3.4 Reward Function

At each time step, in which the agent has not reached the destination, we will give a negative reward to the agent This will promote the agent to move toward the destination. If the agent collides with any obstacles including moving obstacles, we will again give a negative reward. If the robot arrives at the destination, we will give positive reward. If the agent has reached the destination or collided with an obstacle, the training episode is stopped and starts over again.

3.5 Network Structure

Figure 1 above describes the Actor-Critic architecture[2], where the actor network is used to tune the parameter of the policy function, while the critic network is used for evaluating the policy function estimated by the actor using the temporal difference(TD) error in the value function. We can also interpret this as the actor network producing an action given the current state of the environment, and the critic network producing TD error based on the given state and rewards it received. The output of the critic network allows the actor network to learn.

The input layer of both critic and actor network is provided by the state space of *costmap_2d*, but the actual input is to be determined. Critic and actor

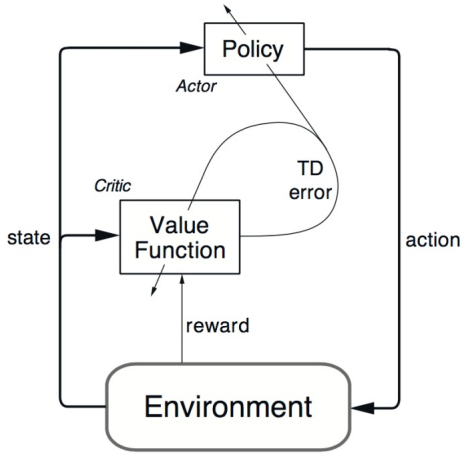

Figure 1: Actor-Critic Architecture of Deep Q Network[2]

networks will both consist of numerous hidden layers in order to satisfy the deep network conditions.

4 Evaluation and Expected Outcomes

The first part of our evaluation is to ensure that our agent is trained successfully. For this, we would first evaluate if our agent is able to avoid all the obstacles in our environment. Second, we would see if the agent prefers to wait or move in case of a moving obstacle that is passing by, and in the third step, we would like to see if the agent is using an optimal path at that particular time step while avoiding static or moving obstacles.

Project's evaluation other than successfully training the agent to reach the destination from some particular starting point will depend upon two factors: normalized reward per iteration, and Q-value per iteration to show how the Q-value is increasing as the agent is trained. We hope to see both normalized reward and Q-value to increase as the iteration increases, as that will show us the agent is learning. At the end of the training, we should see a linear pattern for normalized reward and Q-value as we would see the agent has reached the optimal policy.

If we are successful at training the Turtlebot via simulators provided by ROS, we would like to implement our strategy on the actual Turtlebot in Halligan and see if the Turtlebot can move from some start position to the target destination using the optimal path without hitting any static or moving obstacles.

References

[1] A beginner's guide to deep reinforcement learning. Available at https://skymind.ai/wiki/deep-reinforcement-learning.

[2] Introduction to various reinforcement learning algorithms. part i (q-learning, sarsa, dqn, ddpg). 2018. Available at https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287.

[3] Tai Lei and Liu Ming. A robot exploration strategy based on q-learning network. In *Real-time Computing and Robotics (RCAR), IEEE International Conference on*, pages 57–62. IEEE, 2016.

[4] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[6] Amit Patel. Dealing with moving obstacles. 2018. Available at http://theory.stanford.edu/ amitp/GameProgramming/MovingObstacles.html.

[7] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[8] Iker Zamora, Nestor Gonzalez Lopez, Victor Mayoral Vilches, and Alejandro Hernandez Cordero. Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo. *arXiv preprint arXiv:1608.05742*, 2016.