Path Planning Amidst Moving Obstacles

Jong Seo Yoon, Hifza Khalid

1 Abstract

In this project, we explore how to deal with the path planning problem in the presence of moving obstacles using deep reinforcement learning. We use Deep Q-Network to train our turtlebot in a ROS simulation environment. The state space is continuous and the action space is discrete. The reward is positive for all the state action pairs except when the turtlebot collides with an obstacle where it gets a negative reward. To incorporate moving obstacles in our environment, we use a ROS package called pedsim_ros which introduces randomly moving humans in our simulation environment. The results from our experiments show that the turtlebot learns to avoid moving obstacles.

2 Introduction

In this project, we wanted to deal with the path planning 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 [7]. 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 [7]. Therefore, for such scenarios, we would like our unit to have the additional capability to avoid moving obstacles.

We could have used Q-learning which is a very powerful reinforcement learning algorithm to solve this problem, however, its main weakness is lack of generality. Q-learning can be viewed as updating numbers in a two-dimensional array (Action Space * State Space), which, in fact, resembles dynamic programming [2]. This indicates that for states that the Q-learning agent has not seen before, it has no clue which action to take. In other words, Q-learning agent does not have the ability to estimate value for unseen states. To deal with this problem, we will use Deep Q-Network which incorporates Q-learning approximation with deep neural network.

Neural Networks with Q-Learning can be described as an agent that approximates Q-values for the current continuous state and discrete action pairs. Given the approximated Q-value outputs from the neural network, it uses coefficients to approximate the Q-values 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]. Neural networks seem a good fit to solve our problem because our state space is continuous.

The algorithm that we used to combine Reinforcement Learning and Neural Networks in this project is call Deep Q-Network (DQN). The algorithm aims at choosing the best action for given circumstances [9]. Each action has a corresponding Q-value which determines the quality of that action in that particular state. To end up with accurately approximated Q-values, we use deep neural networks. For each state experienced by our agent, we remember it and perform an experience replay. Experience replay is a biologically inspired process that samples experiences from the memory uniformly to reduce correlation between subsequent actions and for each entry updates its Q value [9]. In short, we update our Q values with the cumulative discounted future rewards.

Our aims for this project can be broadly classified as:

1. Aim 1: Identify the state space, action space, rewards and a reinforcement learning algorithm that can be used to solve our problem.
2. Aim 2: Learn how to use and combine neural networks with reinforcement learning.
3. Aim 3: Simulate Turtlebot navigation using deep reinforcement learning in Robot Operating System (ROS).
4. Aim 4: Teach the Turtlebot how to navigate around the environment with moving obstacles from a starting position to the goal using the optimal path.

We began working toward the above-mentioned aims by first figuring out what would be the state space and the action space for our project. Since, we are working with a 3-D simulation environment, our state space is continuous. To represent different states in our environment, we decided to use LIDAR in this project which is a distance technology that measures how long it takes for the emitted light to return back to the sensor. The discretized laser data is used to represent each state. Our action space is discrete where each the actions for each state are limited to 21. The linear velocity in each state is constant, however, the 21 different actions specify the angular velocity. The reward is always positive and maximum for the goal state. The turtlebot gets a negative reward when it collides with any obstacle.

We decided to work with DQN reinforcement learning algorithm to solve our problem because Google DeepMind implemented DQN on 49 Atari-2600 games

and this method outperformed almost all of other state-of-the-art reinforcement learning methods [3].

3 Related Work

Robots that navigate among moving obstacles especially humans need a safe and efficient way to avoid collision. The folks at MIT have recently introduced an algorithm that learns collision avoidance among different types of dynamic agents and outperforms the existing state-of-the-art methods [5]. The paper assumes that the moving obstacles are other agents that do not communicate among each other and the robot does not assume the behavior of other decision-making agents as their behavior deviates from reality as the number of agents in the environment increases. The collision avoidance algorithm described in the paper uses observations of the arbitrary number of moving agents and deep reinforcement learning to solve the multiagent collision avoidance problem [5]. Our problem is quite similar to their problem, however, we are not assuming that the moving obstacles are other agents.

The paper by group of people at Deepmind[6] 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 data from LIDAR scan is the state which is fed to the convolutional neural network to output possible actions for the turtlebot to choose from.

However, we had a problem regarding continuous action spaces because The number of actions possible for the Turtlebot to decide how to avoid 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 decided to discretize the continuous action space of the Turtlebot by limiting the number of actions possible by the Turtlebot in each state. In each state, actions are determined by the maximum Q value for the current state which the robot has from the memory of the past. Our linear velocity is always the same, that is, 1, however, we have a choice of 21 different angular velocities which determine our action in each state.

4 Technical Approach

4.1 Problem Formulation

In order to build the framework to simulate the moving obstacles environment in gazebo and implement DQN model on the turtlebot, we have combined several

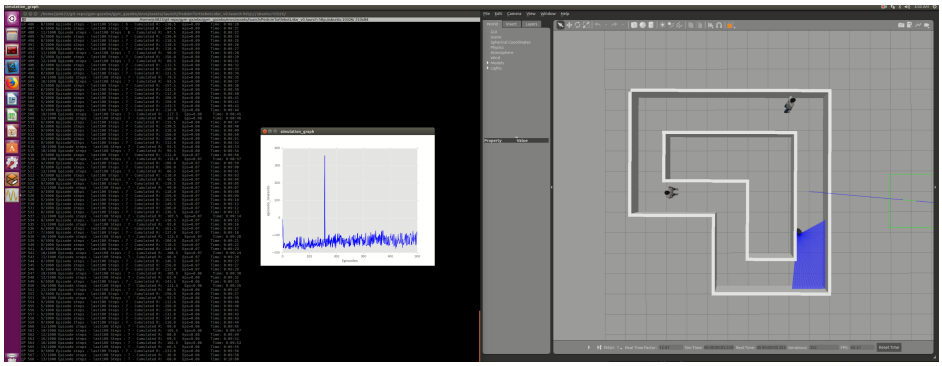

Figure 1: Visual representation of the neural network in DQN. Left window is the terminal where the command has been invoked to run the experiment, and right window is Gazebo showing what the turtlebot is doing.

previous pieces of work done by others in the past to create an environment of our own. Because there is no single solution on ROS which implements moving obstacles and provide libraries to develop DQN model on ROS, we had to combine necessary packages (explained below) and heavily modify the code to make our experiment work. Figure 1 shows the overall look of the package. The left window shows the number of episodes it has taken so far, with the number of steps it took for the episode and the decreasing exploration rate via epsilon decay value. The blue color coming out of the turtlebot on the right window represents the LIDAR and what the turtlebot is currently moving toward.

4.1.1 Reinforcement Learning Framework

We formulate this problem on the simulator called gym-gazebo which is built on ROS[10] environment which provides a wrapper to OpenAI's gym package. OpenAI's gym package is a library for developing reinforcement learning techniques, and it has gained popularity in recent years after Deepmind's work on combining reinforcement learning techniques with deep neural network known as DQN[9]. We will also develop our DQN model using the gym-gazebo package. Gym-gazebo package has been heavily modified to fit our need of creating a new environment, as well as updating the code as necessary in order to make the installation much easier for future work.

4.1.2 Pedestrian Simulator

To simulate the moving obstacle environment, we used pedsim_ros package[8], which was developed on ROS in order to simulate the pedestrian movement using social force model to generate random behaviors. This package allows us to simulate the moving obstacle on the map which the turtlebot navigates on. Unfortunately, pedsim_ros package does not implement the pedestrians on gazebo, but merely simulates walking pedestrians on a non-physical space.

Therefore, this package has also been modified to simulate the pedestrians on gazebo which emulates the physical environment. This work was necessary in order for the turtlebot to detect objects using the LIDAR laser scan.

4.1.3 Deep Q-Network

Reinforcement learning techniques prior to DQN deal with finite state and action spaces. By restricting spaces to discrete, it was much easier for the agent to learn the environment. However, the assumption that states and action spaces are finite largely restricts the agent from learning the actual environment in which we live in. Moreover, most of the environment which we would like the agent to learn in deals with infinite states, infinite actions, or both, which requires immense computational power even if we restrict infinite spaces to a large number of spaces.

DQN was first introduced by Deepmind in a NIPS 2013 workshop paper, and was later published by Nature in 2015 [9]. It has been widely recognized to revolutionize reinforcement learning technique to deal with problems in continuous spaces. While a technique called Deep Deterministic Policy Gradients (DDPG) and double DQN was developed recently, we will focus on using a single DQN model for this paper due to its simplicity on implementation level.
DQN model deals with the environment with continuous state spaces, but with discrete number of actions. By providing a convolutional neural network(CNN) with a variant of Q-learning algorithm, DQN provides us a way to train the agent by using the network to choose an action for a certain state, and adjust the weights using a stochastic gradient descent algorithm. The traditional approach to Q-learning is

$$Q(s,a) = Q(s,a) + \alpha * (r(s,a) + \gamma * max(Q(s') - Q(s,a)) \tag{1}$$

However, we are not able to estimate the Q-value using the formula above, because the traditional approach calculates the exact Q-value at a given state and action space, which is impossible to do on a scale of infinite states with discrete action pairs. Instead, we need to devise a way to approximate the Q-value. DQN approaches the problem by implementing a new formula for the $Q(s,a)$:

$$Q(s,a) = r + \gamma * max(Q(s',a')) \tag{2}$$

The formula (2) states given a state and an action pair, the updated Q-value will be the sum of the reward it gets and the learning rate multiplied by the maximum Q-value for the next state-action pair. Because Q-values are approximated, we assume that given enough Q-values mapped to the reward it gets, neural network's weight can be adjusted to be trained on the continuous state and discrete action pairs.

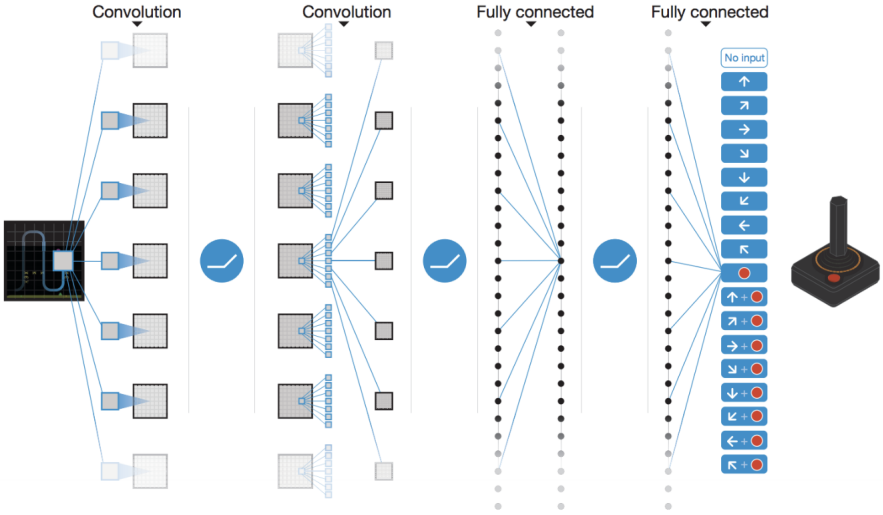

Figure 2: Visual represenatation of the neural network in DQN. [2]

4.1.3.1 Neural Network Structure

The neural network in DQN in our case is a Convolutional neural network, except we are not taking the image portrayed by the turtlebot as an input to the neural network. Instead, we take the range of values given from the LIDAR laser scan mounted on the turtlebot to neural network's input. Turtlebot's LIDAR laser scan gives us 100 range of values, therefore the input to the neural network will be 100 nodes. Then, we create 300 hidden layers combined with 300 hidden nodes with weight regularizers, and we use RELU function as the activation function. Figure 2 is a visual representation of our neural network, where it takes LIDAR's range of distance variables as inputs, and output Q-values which in turn results in action to be taken.

4.1.3.2 Deep Q-learning with Experience Replay

To further improve the neural network in training, we introduce experience replay. On each step during the episode, instead of updating the model based on the last step only, the minibatch is initialized with random numbers which updates the minibatch after certain period of time from the cumulative information of the steps taken before. This allows the neural network to learn not based on the current and previous states, but from the cumulative Q-function it has generated in the past.

The idea behind experience replay is that after a certain point in time, it is highly probable that the neural network has forgotten about the previous experience taken long time ago, and we want to make sure that by returning to the previous

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 3: DQN algorithm from the DQN paper by Deepminds. [6]

experience from time to time, the neural network will be more robustly trained. By introducing previously experienced input spaces and actions, we reduce the risk of the neural network completely forgetting about past experiences. Figure 3 shows DQN algorithm as proposed by the paper written from the people at Deepminds [6].

4.1.3.3 State and Action Space

State space contains the range of values represented in floating point, which calculate the distance from the turtlebot's current position to the physical obstacle in the laser's pointing direction. The state space is continuous because range of values on each step during the episode will differ as the turtlebot takes actions around the map.

Action space, on the other hand, is discrete. We have specified that the linear velocity of the turtlebot is set to 1, and divided the angular velocity into 21 different actions. This gives us 21 outputs from the neural network as the Q-values, and turtlebot makes the decision of choosing the action based on the maximum Q-value generated by the neural network.

The rest of the algorithm functions just like Q-learning algorithm, where there is a discount factor, learning rate, and exploration rate. We set the discount factor as 0.99, learning rate as 0.00025, and and exploration rate as 1. For the exploration rate, we implemented a epsilon decay value of 0.995 so that the exploration rate will gradually decrease as the run goes on. This method was mentioned in the DQN paper[9], and is also implemented in openAI[10].

4.1.3.4 Rewards

We set the reward to be the following.

$$reward = (max\_angular\_velocity - angular\_velocity) * 15 \quad (3)$$

Equation (3) calculates the reward for the angular velocity the turtlebot has decided to take. Since linear velocity is constant throughout, we do not have to consider the reward for moving straight. We have currently set maximum angular velocity to be 0.3 in order to prevent the turtlebot from spinning around. The action taken will determine the angular velocity and thus promote choosing higher angular velocity when avoiding obstacles. This reward equation only applies while the steps are being taken. By adding the angular velocity which is chosen by the maximum Q-value output from the neural network, the turtlebot will learn to choose the correct angular velocity.

The reward given when the turtlebot collides with an obstacle is set to -200 to penalize the agent. The reward for reaching a goal state is 500 to provide the agent with a greater incentive to reach the goal state. Upon colliding with the obstacle or reaching the goal state, the current environment resets position for the next episode.

5 Experiments and Results

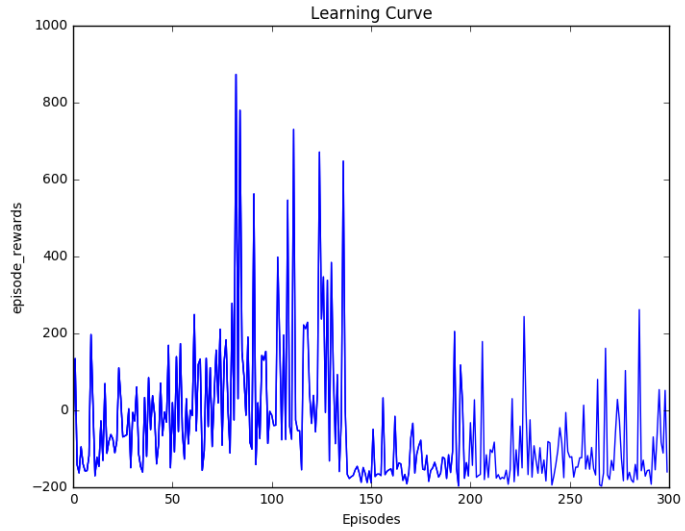

Figure 4: Result from running DQN on our own engineered environment. Moving obstacle may have had an impact after 140 episodes of learning.

Figure 4 shows our current result from the turtlebot navigation around the map. For the first 140 episodes, we can see that the turtlebot is gradually learning to

navigate around the map, and the overall reward is increasing during the first 140 episodes. However, the average rewards received suddenly drop after the first 140 episodes and continue to be around -180. We realized that after many trial runs, the curve looks essentially the same onward.

Until the first 140 episodes, we observed that the turtlebot acts aggressively and intends to explore the environment. It collides with obstacles, including pedestrians, much more frequently and keeps trying. After colliding with the pedestrians many times, it learns to navigate around the pedestrian obstacles. On several episodes, it has also reached the goal state and is awarded with a big reward. However, after 140 episodes, we observed that the turtlebot doesn't deviate much from the starting position. We believe this is due to our error on assigning reward for certain action, as well as the fact that the turtlebot has not learned to reach the goal state to increase the cummulative reward received. Because the area in which the turtlebot must get to in order to achieve the goal reward is hard, the turtlebot hardly chooses to reach the goal state after 140 episodes, and learns to stay on the current position and not collide with any obstacle to get the most rewards.

6 Conclusion and Future Work

In this project, we explored how to use reinforcement learning to allow our agent to avoid moving obstacles. By looking at different research papers [5][6] that were trying to solve a similar problem, we decided to use Deep Q-Network to solve our problem. We used a ROS package called pedsim_ros combined with gazebo to introduce physical moving obstacles (humans) in our environment. Our agent uses output from LIDAR scan that is fed to a neural network which outputs 21 Q-values that are used to determine the next action. Our results show that the agent learns to avoid the moving obstacles until a specific number of episodes because our goal state area is very small and if the agent does not hit the goal state while moving around the environment and hitting the obstacles, it tends to restrict itself to a smaller area to avoid obstacles or negative rewards completely.

As future work, we would like our agent to learn better. Moreover, we would like our agent to have the additional capability to slow down when encountered with a moving obstacle that is hindering its path momentarily. We would also like to see if the agent is using an optimal path while avoiding moving obstacles. If we get successful at training the Turtlebot perfectly 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 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] Yu Fan Chen Michael Everett and Jonathan P. How. Motion planning among dynamic, decision-making agents with deep reinforcement learning. 2018.

[6] 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.

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

[8] University of Freiburg Social Robotics Lab. Pedestrian simulator. https://github.com/srl-freiburg/pedsim$_ros$.

[9] Greg Surma. Cartpole - introduction to reinforcement learning (dqn - deep q-learning). 2018. Available at https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288.

[10] 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.