Reinforcement Learning for Autonomous Driving Obstacle Avoidance using LIDAR

Sahruday Patti
University of Maryland
College Park
sahruday@umd.edu

Denesh Nallur Narasimman
University of Maryland
College Park
deneshn@umd.edu

***Abstract*—This report discusses our project which involves navigating an Autonomous Vehicle in this case, a Turtlebot in a cluttered environment by processing the LIDAR information using Reinforcement Learning. We have simulated a Turtlebot in Gazebo which at the end of 400 episodes with 500 steps per episode, will automatically be able to navigate around autonomously without any human interference and successfully learns a custom Controller. Here, based on each action the autonomous vehicle takes, rewards have been assigned to it which determines what the robot will learn from the environment. The results of the project have also been discussed.**

***Index Terms*—Autonomous Vehicle Navigation, Reinforcement Learning,Q Learning, ROS simulation.**

I. Introduction

The reinforcement learning algorithms have been proven to be extremely accurate in performing a variety of tasks. We propose a reinforcement learning based approach to autonomous driving. The autonomous vehicles must be able to deal with all external situations to ensure safety and to avoid undesired circumstances such as collisions. We propose the use of the Q Learning algorithm for the Autonomous Vehicle to navigate through a cluttered environment. We don't assume a map of the environment a priori but rather rely on our sensor (in this case a LIDAR) to inform control decisions. The standard approach to solving this problem would require building map of the environment using methods like vertical decomposition and using sampling based planning algorithms like RRT*. we also implement simple output feedback controllers such as Braitenberg controllers, which can be quite effective for obstacle avoidance tasks and see how reinforcement learning performs in learning an output feedback controller for obstacle avoidance.

II. Related Work

Reinforcement Learning was starting to attract some attention from late 1970's and has come a long way since. This field has improved and evolved and has become one of the most effective fields in Machine Learning [1].Reinforcement Learning, for parameterizing of feedback controllers helps by giving a general procedure for optimizing the controllers especially for robotics. Most hardware applications with an RL component involve the use of open loop trajectory or control tape. The stabilizing of the controllers that are deisgned can be done with the trajectories using conventional methods like Model Predictive Control [2] and LQR controller [3].

For controlling of dynamic systems such as Autonomous Vehicles, RL uses the past observations instances which are stored [4]. Difficulties like accurate obstacle detection, avoid obstacles and to navigate automatically, the technique policy gradient is also being used which is very effective in training the agent to control the control the car but only in a simulated environment [5].

III. Methodology

A. Car Model

For our car model we consider a simplified Dubin's car model. The location of the vehicle is described by (x, y, θ) where (x, y) denotes Cartesian position and θ denotes the orientation. For our purposes we suppose that we can control the derivative of the orientation, namely our control is $u = \dot{\theta}$. We also suppose a fixed speed v. Then the dynamics of the vehicle are given by

$$\dot{x} = V \cos\theta \tag{1}$$

$$\dot{y} = V \sin\theta \tag{2}$$

$$\dot{\theta} = u \tag{3}$$

B. Sensor Model

For our sensor we use an array of points 75 degrees either side of the car. So, there will be 150 beams spread over the field of view $[\theta_{min}, \theta_{max}]$, where $[\theta_{min}$ = -75 degrees and $\theta_{max}]$ = 75 degrees. Each laser beam has a maximum range d_{max} = 1m. so the n^{Th} laser beam returns x_n which is the distance to the first obstacle it encounters or d_{max} if no obstacle is present. We divide this field of view into 4 regions and take the minimum distance measurement as the reading for each region. Thus each LIDAR measurement can be summarized as X = $(x_1,...x_N)$ where N = 4.

C. Obstacle environment

Obstacles that are too thin along one direction were not a good choice because they could fall between the point Lidar measurements. Thus, circular Obstacles were chosen. But we have seen how the reinforcement learning controller performs in different obstacle environments which will be explained in the results.

D. State representation

To simplify the state representation, we have divided our lidar data to different zones explained below. The Turtlebot Kobuki laser scan is a 360 degree laser scan. But, we considered only 150 degrees (75 degrees either side of the robot), that is the front of the robot to represent our state. Now, we have divided this angle of vision into 4 states.

We name the first state as zero to the left which has a span of 75 degrees. This state has only two representations. If the minimum range of the lidar signal in this region is less than 0.4 ,that means the obstacle is too close and the value of state is 0. If the minimum range of the lidar signal in this region is greater than 0.4 and less than 0.7 , that means the object is in the vicinity and the value of the state is 1. Otherwise, the value of state is 2, which means there is no obstacle nearby. The second state as we call it zero to right, also has the same state value representation.

The third state as we call it, the left zone considers three divisions of the 75 degrees i.e 0 to 25 degrees, 25 to 50 degrees, 50 to 75 degrees. These regions are straight, left and far left respectively. Now, any object is considered as an obstacle if it lies in the range of less than 1 meter in any Lidar reading.

Now, the value of this state will be 0, if an obstacle is present in front of and on the left but not on the far left; the value of state will be 1, if an obstacle is present on the left and far left but not straight and the value of state will be 2, if an obstacle is present in all the regions and the state value will be 3 if there is no obstacle present in any of the regions. The same notation is used for the fourth state as well, but angles are considered right of the robot. By this representation, we have considerably reduced the number of states which was our goal. The x1 and x2 state values range from 0 to 2 and the x3 and x4 ranges from 0 to 3. Thus the total states will be 3 * 3 * 4 * 4 = 144 states. So, at any point of time in our simulation our robot is in one of these states. Here, x1 and x2 are zero to left and zero to right states and x3,x4 are left and right states respectively. To detect and avoid the obstacle, we need only one lidar reading to be minimum in the region and by this representation we can make an informed decision to avoid the obstacle. By this representation, we implement a Valentino Braitenberg style controller, where we look at the state and decide whether to go straight or left or right. Our state represents where the Obstacle is present, so we take action to avoid the obstacle.

E. Braitenberg (Hand-Designed) Controller

We have designed a controller inspired by the controllers of Valentino Braitenberg. The Controller reads the Lidar measurements in the range of 150 degrees and segregates them into 4 regions and reads the minimum measurement from each region and then takes a turn towards the maximum distance measured in this 4 regions. Thus, the action space for this simple controller is $[u, V, -u]$.

```
Braitenberg Hand Designed Controller

Input: States x1, x2, x3, x4
State x1 = $[0, 25]$
State x2 = $[25, 75]$
State x3 = $[-25, 0]$
State x4 = $[-75, -25]$

if x1 and x4 $>$ 1m then
    Go Straight
else
    if x1 $<$ 1m and x4 $>$ 1m then
        Turn Right
    else
        if x1 $>$ 1m and x4 $<$ 1m then
            Turn Left
        else
            if x1 and x4 $<$ 1m then
                if x2 $>$ 1m and x4 $>$ 1m then
                    if x2 $>$ x4 then
                        Turn Left
                    elseTurn Right
                        if x2 $>$ 1m and x4 $<$ 1m then
                            Turn Left
                            if x2 $<$ 1m and x4 $>$ 1m then
                                Turn Right
```

Fig. 1. Braitenberg Hand Designed Controller.

F. Rewards

Reinforcement learning (RL) is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward. So, our reward functions determines what our agent learns to do in the environment. We formulated our Reward for actions as:

- For every crash the reward is -100
- The reward to go straight is +0.2
- The reward to take is -0.1
- The reward for avoiding Obstacle is +0.2
- A reward of -0.8 will be awarded if the decision to change the direction is changed twice i.e when it takes left and then decides to take a right.

G. The Markov Property

The Markov Property states that, the future is independent of the past given the present. S_t denotes the current state of the agent and S_{t+1} denotes the next state. What this equation means is that the transition from state S_t to S[t+1] is entirely independent of the past. So, the RHS of the Equation means the same as LHS if the system has a Markov Property.

Intuitively meaning that our current state already captures the information of the past states.

H. State Transition Probability Matrix

The state transition probability tells us, given we are in state s what the probability the next state s' will occur.

$$P_{ss^{'}} = P[S_{t+1}|S_t] \tag{4}$$

We use Q learning algorithm which is a model free iterative algorithm, hence to solve for State transition Probability Matrix we use Bellaman's Optimality Equation for q*.

I. Markov Decision Process

A Markov Decision Process is a tuple of (**S,A,P,R,**γ).

- S is a finite set of states.
- A is a finite set of actions.
- P is State transition Probability Matrix
- R is a reward function
- γ is discount factor $\gamma \epsilon [0, 1]$

*J. Bellman's Optimality Equation for Q**

Optimal Policy: The goal of reinforcement learning algorithms to find a policy that gives maximum rewards for the agent if the agent indeed follows that policy.In terms of return, a policy π is considered to be better than or the same as policy π^* if the expected return of π^* is greater than or equal to the expected return of π for all states.

Optimal Action-Value Function: The action value function gives us a measure of how good is it to take a particular action under a state s.The optimal policy has an optimal action-value function, or optimal Q-function, which we denote as q_* and can be defined as

$$q_*(s, a) = max_\pi q_\pi(s, a) \forall s \epsilon S, a \epsilon A \tag{5}$$

In other words, q_* gives the largest expected return achievable by any policy π for each possible state-action pair.
Bellman's optimality equation It states that, for any state-action pair (s,a) at time t, the expected return from starting in state s, selecting action a and following the optimal policy thereafter is going to be the Expected reward we get from taking action a in state s, which is R_{t+1}, plus the maximum expected discounted return that can be achieved from any possible next state-action pair($s^{'}$,$a^{'}$).

$$q_*(s, a) = E[R_{t+1} + \gamma max_{a'} q_*(s^{'}, a^{'})] \tag{6}$$

The essence is that this equation can be used to find optimal q* in order to find optimal policy π and thus a reinforcement learning algorithm can find the action a that maximizes q(s, a).That is why this equation has its importance.
The Optimal Value Function is recursively related to the Bellman Optimality Equation.
Bellman's principle of optimality describes how to make the decision problem into smaller subproblems which is the basis for Dynamic Programming.
The Q learning algorithm aim to find the optimal policy of a dynamic programming problem. Thus the first step is to reformulate our problem as a dynamic programming problem.

K. Dynamic Programming formulation

To formulate our problem as a dynamic programming problem we need to define a "state." Let S = (x,y,θ) be the state of the car and let X = (x_1,..x_n) be the sensor readings at a particular state of the car. What the agent can observe is S and X. The true state is the environment state as whole which includes the environments description as a whole. We dont give our learning algorithm a prior map and description of the environment apart from the car and sensor state. hence, the agent is in a partially observable state. To make our problem similar to the limitations of a real car robot with just the described LIDAR sensor model, we only allow our reinforcement learning agent to access X, and so we consider S = X (this is because the dynamics of the car are trivial here). Next we need to define our action set. In principle our model has a continuous action set. However, for the purposes of SARSA and Q-Learning it is much easier to have a discrete action space. Thus we restrict ourselves to a ϵ A. We have already set up the Rewards for these actions. Now, we can formulate our problem as a dynamic program. The problem is to find the policy $\pi : S \rightarrow A$ to solve

$$max_\pi E_\pi [\sum_{t>=0} \gamma^t R(S_{t+1}, a_t)] \tag{7}$$

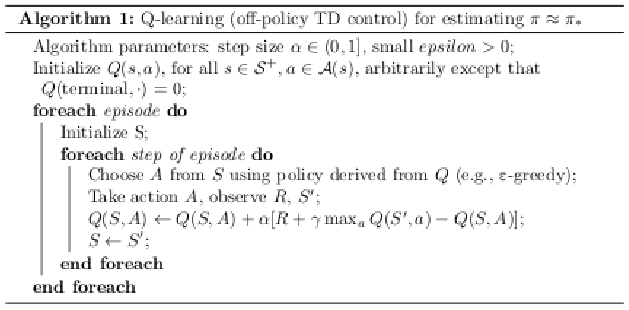
The expectation is over the reward states that result from taking actions according to policy π. In a standard dynamic programming framework the state S_t would be Markov. That is, future states depend only on the current state and future actions. In our case however our reward state is non-Markov since the map of the world is not included as part of our state.

L. Watkins Q learning

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" refers to the function that the algorithm computes – the expected rewards for an action taken in a given state.

IV. Simulation Environment

For the simulation, we make use of Robotic Operating System and Gazebo. For our Dubin's car model we use an existing Turtle bot Package which are equipped with Lidar Sensors. Initially, we decided to use Open-AI gym for ROS to simulate a car in Gazebo which is better adept at respwaning the robot once it crashes.But, Open-AI gym had integration problems with Ubuntu 20.04. Additionally, we encountered



```
Algorithm 1: Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$
Algorithm parameters: step size $\alpha \in (0,1]$, small $epsilon > 0$;
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that
  $Q(\text{terminal}, \cdot) = 0$;
foreach episode do
    Initialize S;
    foreach step of episode do
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., ε-greedy);
        Take action $A$, observe $R$, $S'$;
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$;
        $S \leftarrow S'$;
    end foreach
end foreach
```

Fig. 2. Q learning

many errors specific to Open-AI gym for which no support was available for Ubuntu 20.04. So to Re spawn the robot we use the Lidar sensor to detect the collision and reset the simulation. We achieve this by reading the Lidar ranges in 150 degree range and when the minimum of these ranges is less than 0.14m we consider a collision has occurred and we reset the simulation. Now, in order to recognize that the robot is at a distance of 0.14m, weighted arrays are being used where the extreme left and right point (75Th degree on either sides) is multiplied by 1.2 and the middle point (0Th degree) is multiplied by 1. The values in between the extreme points and the middle points are multiplied by values in between 1.2 and 1. This gives us an equally weighted distribution and an accurate result of whether the robot has collided or not. Once this is determined, "/gazebo/set model state", is used to reset the TurtleBot. We have also experimented with resetting the TurtleBot from random positions and initial positions in the environment.All the code for the working of simulation which includes Lidar scan descretization, control mapping and QLearning algorithm is written in Python.

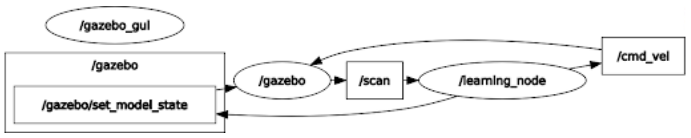


Fig. 3. RQT Graph

The code package can be found in this link. https://github.com/saahu27/Obstacle_Avoidance_QLearning

V. Experimental Results

We initialized our Q table with 0 as action values for each action of every state and started our simulation to update our Q table. So each round of our training involved 400 episodes and each episode involved 500 steps for episode to reach the terminal state. we had to do several rounds of this training in order for our Q table to be updated to optimal values. All the below graphs shown and explained below are with respect to the optimal Q table updated after several iterations of training.So to begin with, our exploration is very high. we start with 0.90 probability of choosing a random action in order to explore all the states and update the action values in those states. After, each episode we exponentially decrease our exploration rate to a minimum exploration probability of 0.05. The exploration is not set to zero because we still want to continue to explore. The below graph is the epsilon value wrt number of episodes.

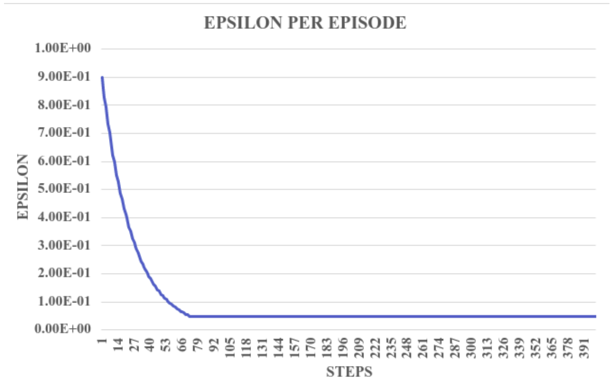


Fig. 4. Epsilon per episode.

As the Exploration rate is high initially,agent chooses random actions with high probability which may not be max action value actions for that particular states.Hence,the steps per episode in each episode initially are low and the agent crashes more frequently.But as the number of episodes increase, the exploration rate decays and we choose more max action value actions with more probability and hence our steps per episode increase.This is shown in the below graph. We end up having some episodes with lesser than max steps for episode because we are still exploring random actions albeit with a low probability. Towards the end our of episodes, we achieve maximum steps for episode and the car doesn't collide with any obstacles through out the entire episode. This shows that our formulation has achieved its optimality and we are able go through the entire episode without crashing more often.

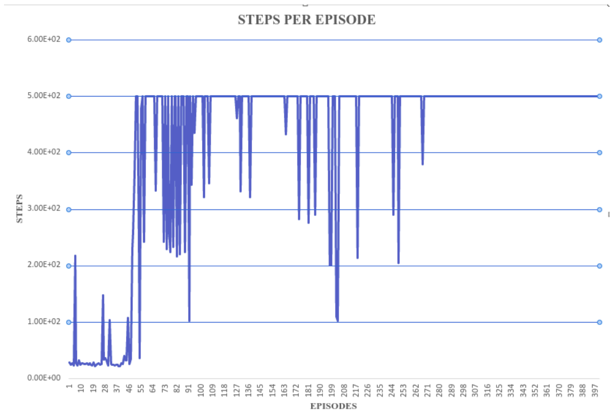


Fig. 5. Steps per episode.

We evaluate the performance of our RL agent in the rewards it gained during each episode which gives us an accurate estimate of how our agent is performing. Below graph shows the rewards gained at end of each episode. It is in accordance with our reward system, exploration rate and steps for episodes. As you can Observe in the graph our reward system is robust and our car doesn't revolve around in circles. Even at the latter episodes there is change in reward and are not constant reward which might have been the case if the car revolved around in circles. This shows that our reward system is good enough and robust to prevent our agent to learn a particular pattern. Ensuring we are not going around in circles or in any particular pattern we also ensured our robot follows different paths in different episodes.

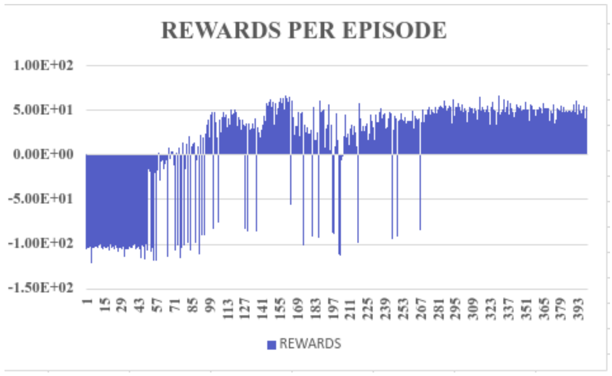


Fig. 6. Reward per episode.

We have also implemented Sarsa for the same obstacle avoidance task which just involves picking action from a random and updating the table. But we were not able to train this algorithm to show the results.

VI. Conclusion

In conclusion, we have investigated an RL method for learning controllers for our obstacle-avoidance task. The Q learning algorithm was able to learn controllers that would improve performance with training time, as expected. While the learned controller was impressive, we have the hand designed controller worked much better with a lot less effort.

The future work would involve trying this method to learn more complex controllers in dynamic environments.

References

[1] Andrew, A.M., 1999. REINFORCEMENT LEARNING: AN INTRODUCTION by Richard S. Sutton and Andrew G. Barto, Adaptive Computation and Machine Learning series, MIT Press (Bradford Book), Cambridge, Mass., 1998, xviii+ 322 pp, ISBN 0-262-19398-1,(hardback,£ 31.95). Robotica, 17(2), pp.229-235.

[2] Camacho, Eduardo Bordons, Carlos. (2004). Model Predictive Control. 10.1007/978-0-85729-398-5.

[3] J. W. Roberts, I. R. Manchester and R. Tedrake, "Feedback controller parameterizations for Reinforcement Learning," 2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), 2011, pp. 310-317, doi: 10.1109/ADPRL.2011.5967370.

[4] Jeffery Roderick Norman Forbes (1993). Reinforcement Learning for Autonomous Vehicles [Unpublished bachelor's thesis]. University of California at Berkeley.

[5] Huynh, A.T., Nguyen, B.T., Nguyen, H.T., Vu, S. and Nguyen, H.D., 2021. A Method of Deep Reinforcement Learning for Simulation of Autonomous Vehicle Control. In ENASE (pp. 372-379).

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.