Collision Avoidance using Reinforcement Learning

Chethan Mysore Parameshwara

1. Abstract

In the current era of Autonomous cars and Artificial Intelligence, robots are expected to learn to explore and navigate in an unknown environment like Humans. Autonomous navigation problem is very interesting and challenging problem. There are many subproblems like control, planning, and collision avoidance to be solved to achieve autonomous navigation. In this project, the collision avoidance problem is tackled with integration of Reinforcement Learning and Deep Learning algorithms.

2. Introduction

Collision Avoidance is one of the fundamental problems for next-generation autonomous vehicles. The objective of autonomous navigation is to plan and control the motion of a vehicle from its initial position to its goal position, while avoiding obstacles. Avoiding collisions with pedestrians and other vehicles is required for autonomous vehicles, particularly those operating in traffic environments. The problem of collision avoidance for autonomous vehicles has been the focus of tremendous research effort. Most of the algorithms are only suitable for avoiding stationary obstacles and/or moving obstacles whose future position and moving direction are predictable. In other words, algorithms need maps of its environment to avoid collisions. However, generating a map of unknown environment is practically impossible. This project aims at solving the collision avoidance problem in an unknown environment through Reinforcement Learning.

3. Related Work

Pomerleau [1] developed ALVINN (Autonomous Land Vehicle In a Neural Network), a 3-layer back-propagation network designed for the task of road following. ALVINN takes images from a camera and a laser range finder as input and produces as output the direction the vehicle should travel in order to follow the road. Training has been conducted using simulated road images and successfully tested on the Carnegie Mellon autonomous navigation test vehicle. Forbes et. al [2] developed a decision-theoretic architecture to tackle the driving problem through Bayesian Automated Taxi (BAT) project. Forbes [3] also examine the task of learning to control an autonomous vehicle from trial and error experience, which is nothing but Reinforcement Learning.

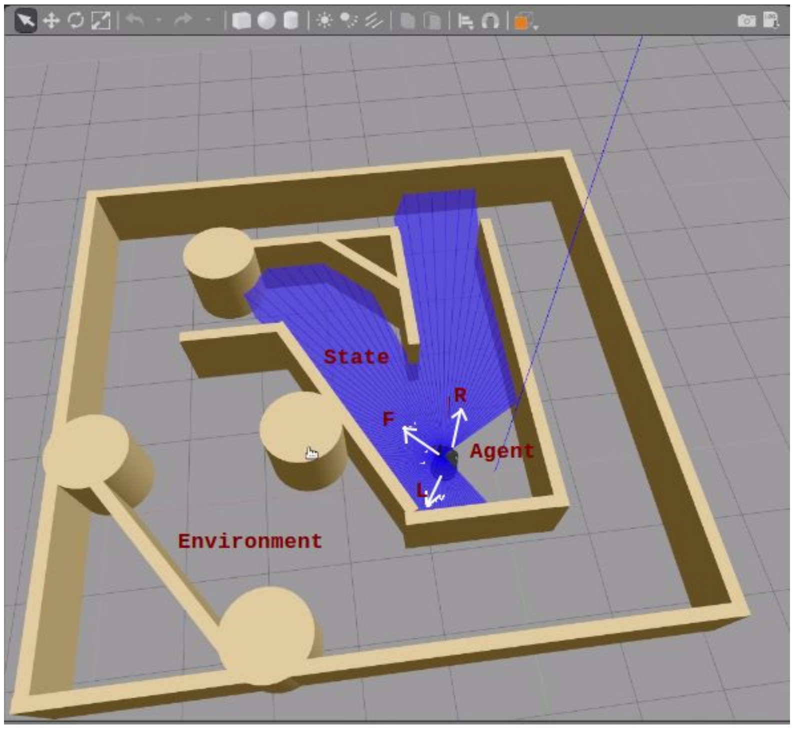

Fig 1. Problem Formulation

4. Problem Formulation

In this project, Reinforcement Learning algorithms are used to solve the collision avoidance problem. The most common method to represent the reinforcement learning problem is through Markov decision process. In our case, Turtlebot is an agent situated in a simulated maze environment. Turtlebot determines its state by analysing the data from LIDAR sensor to perform certain actions like moving forward or colliding with a wall. Each action is rewarded by a predefined reward function. Turtlebot chooses actions based on its current state through policy. The set of states and actions, together with policy for transitioning from one state to another, make up a Markov decision process. A Markov decision process relies on the Markov assumption, that the probability of the next state depends only on current state and action, but not on preceding states or actions

5. Approach

The problem of collision avoidance is solved by implementing Reinforcement Learning (RL) algorithms on a robot in a simulated environment. In this project, LIDAR sensor is used to get data of the environment. Sensor data is fed to Q-learning or Deep - Q Network as a state. Further, Corresponding Q -values for each actions is obtained as the output. The action corresponding to maximum Q-value is carried out on a robot. During learning phase, Q values are continuously updated using reward function. Learning is carried out by using both Q-learning and DQN. Before diving into details of implementation let us review about RL algorithms in brief.

5.1. Q-learning

In Q-learning, an agent tries to learn the optimal policy from its experience of interaction with the environment. A experience of an agent is a sequence of state-action-rewards:

$$s_0,a_0,r_1,s_1,a_1,r_2,s_2,a_2,r_3,s_3,a_3,r_4,s_4... ,$$

We treat this experience of interaction as a sequence of episodes, where an episode is a tuple

$$s,a,r,s' ,$$

These experiences will be the data from which the agent can learn what to do. As in decision-theoretic planning, the aim is for the agent to maximize its value, which is usually the discounted reward. Q-learning uses temporal differences to estimate the value of $Q^*(s,a)$. In Q-learning, the agent maintains a table of $Q[S,A]$, where S is the set of states and A is the set of actions. $Q[s,a]$ represents its current estimate of $Q^*(s,a)$. An episode s,a,r,s' provides one data point for the value of $Q(s,a)$. The target is estimated by adding current reward to discounted estimated future value i.e. $r+ \gamma max_a Q(s',a')$. The agent can use the temporal difference equation to update its estimate for $Q(s,a)$:

$$Q[s,a] \leftarrow Q[s,a] + \alpha(r+ \gamma max_{a'} Q[s',a'] - Q[s,a])$$

Q-learning learns an optimal policy no matter which policy the agent is actually following (i.e., which action a it selects for any state s) as long as there is no bound on the number of times it tries an action in any state (i.e., it does not always do the same subset of actions in a state). Because it learns an optimal policy no matter which policy it is carrying out, it is called an off-policy method.

5.2. Deep Q- Network

Constructing Q - table for complex problems would be very cumbersome. In literature[4], researchers have used Neural Network to approximate the Q - function. Neural networks are exceptionally good at coming up with good features for highly structured data. We could represent our Q-function with a neural network, that takes the state and action as input and outputs the corresponding Q-value. Alternatively we could take LIDAR data as input and output the Q-value for each possible action. This approach has the advantage, that if we want to perform a Q-value update or pick the action with the highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions available immediately. Outputs of the network are Q-values for each possible action. Q-values can be any real values, which makes it a regression task, that can be optimized with simple squared error loss.

$$L = ½ [r+ \gamma max_{a'} Q(s',a') - Q[s,a]]^2$$

5.2.1.Problems with DQN

Reinforcement learning becomes unstable or even diverges when a nonlinear function approximator such as a neural network is used. This instability leads to correlations in the sequence of observations and correlations between the action-values (Q) and the target values (r + max (Q))

These problems can be addressed through use of two key ideas. First, the biologically inspired mechanism termed Experience Replay. In Experience Replay, sequence of observations are randomized thereby removing correlations. Second, target values are periodically updated, thereby reducing correlations with the action-values (Q).

6. Implementation

6.1. Software Architecture

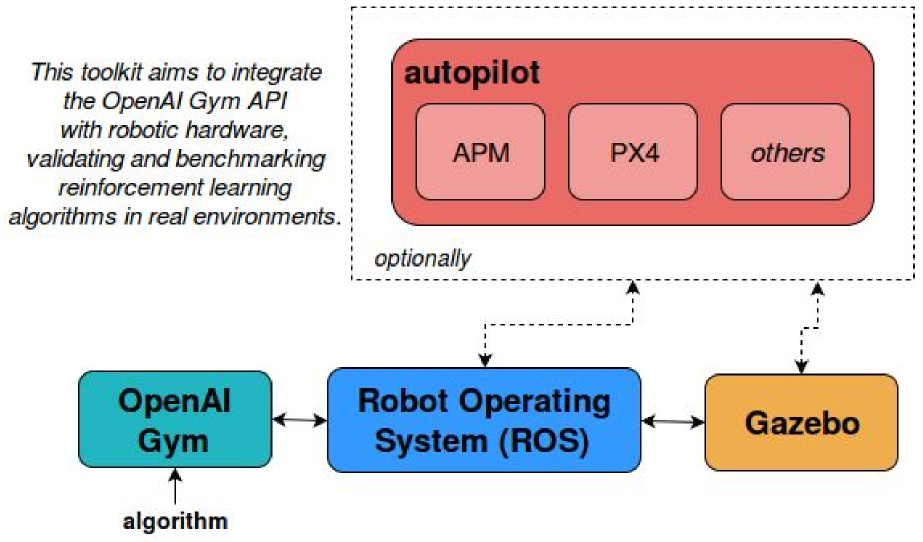
This toolkit aims to integrate the OpenAI Gym API with robotic hardware, validating and benchmarking reinforcement learning algorithms in real environments.
autopilot
APM
PX4
others
optionally
OpenAI Gym
Robot Operating System (ROS)
Gazebo
algorithm

Figure 2. Software Architecture

The architecture [5] consists of three main software blocks: OpenAI Gym, ROS and Gazebo. Environments developed in OpenAI Gym interact with the Robot Operating System, which is the connection between the Gym itself and Gazebo simulator. ROS and Gazebo softwares are used to simulate the Turtlebot robot in a maze environment. The RL algorithms are implemented using libraries of OpenAI Gym and Keras. Python is used for writing scripts throughout the project.

6.2. State and Action Design

The data from simulated LIDAR data is fed as the state information to RL algorithms. The sensor data contains 100 points which are discretized to reduce the size of the state vector before storing it in Q- table.

In the case of DQN, all the 100 points are fed to Neural network in finding the Q-function. The size of action vector in both Q - Learning and DQN remains as three i.e. Forward, Right, and Left.

6.3. Source code and Pseudocode for Q-learning and DQN [6]

The source code has been uploaded to Github website [7]

6.3.1. Q-learning

```
initialize $Q[num\_states, num\_actions]$ arbitrarily
observe initial state $s$
repeat
    select and carry out an action $a$
    observe reward $r$ and new state $s'$
    $Q[s,a] = Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
    $s = s'$
until terminated
```

6.3.2. Deep Q- Network

```
initialize replay memory $D$
initialize action-value function $Q$ with random weights
observe initial state $s$
repeat
    select an action $a$
        with probability $\varepsilon$ select a random action
        otherwise select $a = \mathrm{argmax}_{a'} Q(s,a')$
    carry out action $a$
    observe reward $r$ and new state $s'$
    store experience <$s$, $a$, $r$, $s'$> in replay memory D

    sample random transitions <$ss$, $aa$, $rr$, $ss'$> from replay memory D
    calculate target for each minibatch transition
        if ss' is terminal state then $tt = rr$
        otherwise $tt = rr + \gamma \max_{a'} Q(ss', aa')$
    train the Q network using $(tt - Q(ss, aa))^2$ as loss

    $s = s'$
until terminated
```

7. Results

Turtlebot is started from center of the maze and expected to traverse throughout the environment without colliding any walls. Q - Learning and Deep Q - Network are tested on same experiment setup with different learning parameters.

7.1. Q - Learning

The robot learned to traverse the environment without collision at around 1000 episodes. The video of Q- Learning implementation is uploaded to Youtube [8]. Figure 3 shows the reward collected by Turtlebot in each episode throughout the experiment. Even though the robot had created Q-table for collisionless traverse by 1000 episodes, it collided with wall because the exploration rate was set constant throughout the experiment and also it was traversing for more than one time after 1000 episodes.

Learning Parameters

- Max_Episodes = 3000
- Exploration_Rate = 0.3
- Learning_Rate = 0.9
- Discount_Factor = 0.99

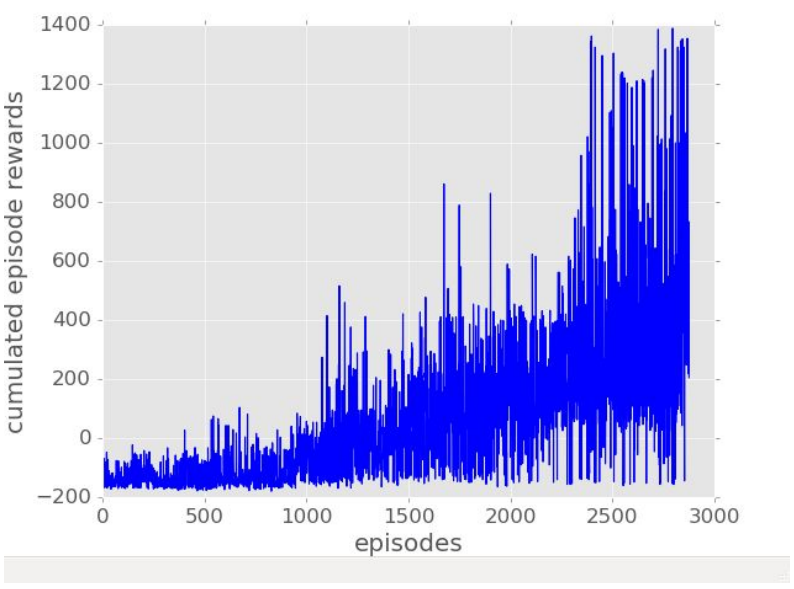

Figure 3 - Cumulated Rewards in Q - Learning case

7.2. Deep Q-Network

Three fully connected layer Neural Network is used to approximate the Q-function. The network parameters and architecture information is provided below. The video of DQN implementation is uploaded to Youtube [9]. Even after running 15,000 episodes, the robot didn't traverse the whole maze completely for at least once. According to literature, the maximum number of episodes should be at least 100,000. Due to hardware constraints in this project, it is not possible to cross more than 15,000 episodes. Figure 4 shows the rewards collected at each episodes. The reward function for DQN has different value for different actions when compared to Q - learning. Hence, the cumulative rewards are very high in the DQN case.

Network Architecture

- Input Size - 100
- Output Size - 3
- Network = 3 Fully Connected Layer
- Activation Function - ReLU
- Optimization Solver- RMSprop

Network Parameters

- Max_Episodes = 15000
- Max_steps_episode = 1000
- Exploration_Rate = 1 (adaptive)
- Minibatch_size = 64
- Learning_Rate = 0.8
- Discount_Factor = 0.99

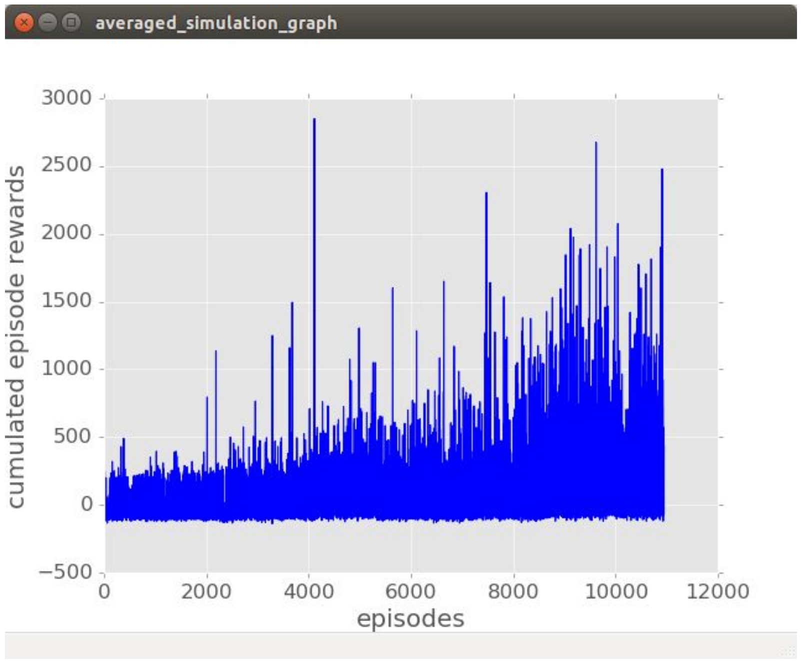

Figure 4 - Cumulated Rewards in DQN case

8. Analysis

Q - learning is easy to train and converges rapidly with less number of episodes. Even though Turtlebot completed the maze for couple of time, actions taken by it are not smooth and very vague. This is because the LIDAR state information is discretized from 100 to 3 points. Although DQN didn't complete the maze traversal, the movement of the robot in the environment was very smooth. In the DQN case, the robot takes all the 100 points to decide which action to take. Furthermore, the incompletion of robot in the DQN case is because of insufficient training. Due to hardware constraint, it is not possible to run the training for more than 15,000 episodes.

9. Conclusion

In this project, collision avoidance problem in autonomous navigation is tackled using Reinforcement Learning approach. The Q - Learning and Deep Q-Network are implemented in ROS using OpenAI libraries. Each of the reinforcement algorithm implemented in this project has its own advantages and disadvantages. Q- learning has fast convergence but non smooth action performance. On the other hand, DQN has slow convergence but very smooth performance. Even though DQN is overkill for solving collision avoidance problem in a simulated environment, it was implemented to understand the training and analyse the performance.

10. Future Work

The future work of this project would be implementation of Reinforcement Learning algorithms on complex maze environment with usage of high performance GPU. The project can be extended to different vehicles like Erle-copter (a quadrotor) and Erle-rover (a rover) whose models are available in the Open AI Gym library.

11. Bibliography

[1] Pomerleau, Dean A. Alvinn: An autonomous land vehicle in a neural network. No. AIP-77. CARNEGIE-MELLON UNIV PITTSBURGH PA ARTIFICIAL INTELLIGENCE AND PSYCHOLOGY PROJECT, 1989.

[2] Forbes, Jeff, et al. "The batmobile: Towards a bayesian automated taxi." IJCAI. Vol. 95. 1995.

[3] Forbes, Jeffrey Roderick Norman. Reinforcement learning for autonomous vehicles. Diss. UNIVERSITY of CALIFORNIA at BERKELEY, 2002.

[4] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.

[5] Zamora, Iker, et al. "Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo." arXiv preprint arXiv:1608.05742 (2016).

[6] https://www.nervanasys.com/demystifying-deep-reinforcement-learning/, Demystifying Deep Reinforcement Learning

[7] Source Code, https://github.com/analogicalnexus/gym-gazebo

[8]Collision Avoidance using Q - Learning, https://www.youtube.com/watch?v=9dVvnyUN0EQ

[9] Collision Avoidance using Deep Q - Network, https://www.youtube.com/watch?v=u8w3MB4vGj8