RRT*-guided robot learning for planning

Vidyaa Krishnan Nivash, Shivam Bhat

***Abstract*—Imitation learning is a type of learning policy where in an agent learns to perform a task by observing demonstrations of the task performed by an expert. In imitation learning policy, the agent learns a policy by imitating the expert's behavior. The goal is to learn a policy that can generalize to new situations beyond the demonstration set. It is useful when it is difficult or costly to design reward functions or when the expert's behavior is optimal or near-optimal. In this work we implement an imitation learning policy for a turtlebot in a maze environment. Data gathered by running the RRT*, including the local lidar observations are used to guide the behaviour cloning policy. RRT* is a probabilistically complete algorithm used for path planning in robotics that aims to find an optimal path from the start to the goal while avoiding obstacles. Furthermore, an imitation learning policy is implemented to generate the next waypoint toward the goal using only the local lidar observations. Broadly the work consists of following three contributions:**

- **Setup maze environment with Turtlebot in Pybullet**
- **Run RRT* to gather data, including local lidar observations**
- **Implement an imitation learning policy to generate the next waypoint toward the goal using only the local lidar observations**

I. Introduction

ROBOT navigation in an unknown dynamic environment remains a challenging problem in the field of robotics. Many approaches have been proposed to address this challenge, including the use of Probabilistic Roadmap (PRM) and Rapidly-Exploring Random Tree (RRT*) algorithms. In this paper, we present a project that integrates RRT*-guided robot learning and Bevaiour Learning for control in a maze environment using Pybullet simulation.

The project involves setting up a maze environment using the Turtlebot robot in Pybullet based 'pybulletgym' environment, generating a RRT Star graph to find the optimal path from one waypoint to another, thereby allowing collection of data which is then used to implementing an Imitation Learning policy to navigate the robot through the maze environment. Our proposed approach involves using local observations from the robot's lidar sensors to learn a policy that enables it to navigate from one waypoint to another while avoiding obstacles in the environment.

The contributions of this work include testing out our approach on two different 3d maze environments Wherein we first run path planning in 2 dimensional setup since in the given scenario of navigating a Turtlebot robot through the maze, the 3d environment is easily reducible to 2d. The RRT Star model then acts as an expert to implement an imitation learning policy by feeding it the transition lidar data. We demonstrate that the proposed approach performs well methods in terms of navigation efficiency and obstacle avoidance.

The rest of the paper is organized as follows: Section 2 presents the related work on RRT*-guided robot learning for control. Section 3 describes the methodology used in this project, including the setup of the maze environment, the generation of RRT* tree, and the implementation of RL policy. Section 4 presents the experimental results and the evaluation of the proposed approach. Finally, Section 5 concludes the paper and discusses future research directions.

II. Frameworks

A. RRT Star

At the heart of this work lies the RRT Star algorithm which acts like an expert and provides data corresponding to valid paths to help train the imitation learning policy.RR* are a improvement on the classic Rapidly-exploring Random Trees (RRT) which have been shown to be an effective algorithm for motion planning in high-dimensional spaces. The original RRT algorithm is designed to explore the entire search space uniformly, which can often result in suboptimal solutions. RRT* is an extension of the original RRT algorithm that addresses this limitation by iteratively rewiring the tree structure to improve the quality of the search. RRT* uses a cost-to-come metric to bias the search towards regions of the space that are more likely to yield optimal paths. Additionally, RRT* employs a dual-tree search approach to further improve the efficiency of the algorithm. Empirical studies have shown that RRT* outperforms the original RRT algorithm in terms of solution quality and computation time. Therefore, RRT* is a promising algorithm for motion planning in complex and high-dimensional spaces like ours.

B. Imitation Learning

Imitation learning, also called learning from demonstrations, is a machine learning technique that has attracted a lot of attention in recent years due to the potential for robots and autonomous agents to learn from human demonstrations. In imitation learning, an agent is trained to imitate the behavior of a human expert by observing a series of expert demonstrations. This approach is widely used in various applications such as robotics, autonomous driving, and gaming. In our apporach the data gathered while traversing the RRT* guided path acts as an expert.

Imitation learning has been shown to be a powerful technique for achieving high performance on complex tasks. The main advantage of imitation learning is that it allows agents to learn from the experience of human experts, avoiding the need for extensive trial-and-error training, which is time consuming and expensive. Furthermore, imitation learning allows agents to learn to perform complex tasks that are difficult to define analytically.

To overcome these challenges multiple approaches have been proposed such as Inverse Reinforcement Learning, where the goal is to learn a reward function that effectively explains the expert's behavior, and Adversarial Imitation Learning, where the goal is to learn a policy that is indistinguishable from the expert's policy. Other approaches include Interactive Imitation Learning, where the agent interacts with the expert to learn from their feedback.

Despite the challenges, Imitation Learning has shown promise in various applications and is being extensively used to train robots to perform tasks such as grasping, manipulation, and locomotion. In our proposed work we implement two variant of the above mentioned Leanrning policy - Behavioral Cloning and DAgger Algorithm.

III. Implementation

A. Environment and Setup

We construct a 3d Environment comprising of maze and obstacles. To test the resilience of this proposed approach we construct two mazes -

- U Shape Maze
- Random Forest Maze

Random Forest Maze introduces more complexity by allowing robot multiple choices for path planning at a junction. This is further made complex by placing 3d obstacles along the path to goal.

We have extended the original PyBullet environments provided in pybullet_envs. Pybullet has a physics engine which can be connected to using graphical or non-graphical mode. URDF are taken from [1] and modified according to our environment. The steps to create the environment in PybulletGym as listed below:

Steps for creating the walls:

- Create ground using a plane urdf by calling loadURDF() function
- each wall (left, right, top, bottom) consist of a block in itself, which can be created using a box geometry with length, width and height specified.
- Parameters of loadURDF()[2]:
  - (x,y,z) - position in which the model has to be loaded
  - useFixedBase set to True, since the walls need not move according to physical interactions
  - URDF_USE_SELF_COLLISION - set to True in order to check the collision of the wall with itself
- changeVisualShape() is used to generate walls of different colors and textures
- createBlock() uses the 'p.createMultiBody()' function to create obstacle
- Everytime the robot is reset to original position, walls are created and updated

B. Turtlebot Robot

PyBullet is a physics simulation engine that can be used to simulate various robots, including the TurtleBot we use herein. Through PyBullet, developers can simulate the TurtleBot's movement, sensor data, and environment in a virtual setting. This allows for testing and development of the TurtleBot's algorithms and behaviors without the need for a physical robot. Additionally, PyBullet also allows for the creation of complex scenarios that may be otherwise difficult to recreate in the real world.

The TurtleBot robot can be represented in PyBullet as a set of rigid bodies and joints joined together by constraints.The robot's primary body can be visualized as a box-shaped structure with two wheels hinged to it.The motors that power the wheels can be modelled in PyBullet by applying rotational forces or torques to the wheels.The depth camera and 2D laser scanner sensors on the TurtleBot can be modelled using PyBullet's collision detection and raycasting features. Another way to simulate the gripper arm is to use a number of rigid bodies and joints. Furthermore, PyBullet has a variety of environmental models and objects that can be utilized to construct intricate scenarios for

The TurtleBot is equipped with various sensors, including a 2D laser scanner, lidar sensors, depth camera amongst other, which enable it to perform autonomous navigation and mapping tasks. In this implementation we will be using the lidar data for the purpose of path planning.

PyBullet provides the "p.rayTest()" function which allows us to perform a ray cast between two points in a physics simulation. This is very useful for a variety of tasks, such as detecting collisions or determining the distance between two objects. The function takes three parameters: "rayFromPosition", "rayToPosition", and an optional "physicsClientId". The start and end points of the ray cast are specified, respectively, by the variables "rayFromPosition" and "rayToPosition". Each tuple in the list of tuples returned by the function denotes a collision that occurred along the ray cast path. Each tuple includes details on the collision point, the colliding surface's normal vector, and the collision object ID.It is possible to define which physics simulation instance will be utilized for the ray cast by using the "physicsClientId" argument. This was helpful during development when several simulations were running at once. Output readings are shown in result section

C. Imitation Learning Frameworks

Imitation learning is a subfield of machine learning that aims to allow an agent to learn a task by emulating the actions of an expert. It is a potent paradigm for instructing intelligent agents like robots to carry out difficult tasks, and it has a wide range of uses in fields including robotics, self-driving cars, and video games. In our work we leverage the 'imitation' package[3] to build code for our use case. The 'imitation' framework provides a PyTorch implementations of imitation and reward learning algorithms. We implement its two popular flavours.

Behavioral Cloning (BC), a popular method of imitation learning, teaches an agent to imitate expert behavior through training on a collection of expert demonstrations. The basic concept is to approximate this mapping by first training a

neural network to map the expert's actions to the associated states or observations. Once trained, the agent can use the network to decide what to do in new circumstances by choosing the course of action that the network thinks the expert in that situation would take.

However, BC has a few drawbacks, such as the propensity to amplify errors and the challenge of handling novel situations not seen in the expert demonstrations. A more sophisticated technique called DAgger (Dataset Aggregation) was suggested as a solution to these problems. DAgger iteratively collects new training data from both the agent and the expert, and aggregates it into a larger dataset for training a new policy. This immensely helps in addressing the issue of distributional shift, wherein the agent's actions may differ from those of the expert in unfamiliar circumstances. Additionally, DAgger can help the agent perform better by actively identifying scenarios in which it is likely to make mistakes and requesting advice from the expert on how to behave better in those circumstances.

Overall, imitation learning is a promising approach to enabling robots and other agents to learn complex tasks from human experts.Although BC is a straightforward and understandable approach, it has drawbacks that more sophisticated algorithms like DAgger can address. It is conceivable to develop agents that can learn from people in a robust and efficient way by combining the advantages of both methodologies.

D. Running File-Maze.py

Main code where the episodes of robot and maze are created using gym.make() and run using env.reset(), step() and render()

E. Versions

- gym==0.17.2
- pybullet==2.6.4
- opencv-python==4.1.2.30
- imitation=0.3.2

F. Code structure and methodology

The main driving code is present in the file 'maze.py'. After registering the various global parameters using the 'bullet_envs' init file the environment 'TurtlebotMazeEnv-v0' is initialized using gym.make(). The scene is then finally rendered using a random policy. '/src/bullet_envs/turtlebot/' contains all the assets required to render the robot and maze environments. '/src/bullet envs/utils.py' is used to write various utillity functions like those for seeding and rnadom noise generation.

We construct the maze and obstacle by using the various API provided by the native pybullet environment. Once we have the environment we map it to a 2d environement. In the given scene, the robot in consideration - turtlebot, only moves in the XY plane. Further, since there are no protruding links, any collision in Z plane is equivalent to a collision to in XY plane. This unique arrangement allows us to perform most of the path planning in 2 dimension space.

Although we run RRT star algorithm in a 2d space. As an additional check, we have coded collision checks in the 3d gym environment space too where we look for collisions between the robot and environment. While running the RRT we capture data which is then used for developing an imitation learning policy. The data we capture includes its location in observation space and lidar meta data. Since each RRT path has different number of waypoints, we were required to pad the observations to be consistent with the batch size requirement of the BC class provided by imitation package.

The sensor we were able to obtain was limited in its range. It showed collision only if the object was within a certain distance of the robot. In case of a U shape maze we noticed that it was only when very near to the turns, that it showed any difference in motion. This resulted in obstacle detection only near the turns. This also often resulted in robot navigating too close to the boundary and being momentary stationary before turning. We first tested on the required U maze and once we were able to get satisfactory performance we explored and tested on more complex environments as suggested by professor during our discussion.This allowed us to test the robustness of our proposed implementation.

We tried and tested various combination of data as an input to our Behaviour and Daggers trainers. We first used observation space data which mainly consisted of location data. We were able to get satisfactory results however this made our model biased towards the environment structure,path and location of obstacle instead of nature of task-which is obstacle detection. We then augmented that data with some parts of the lidar meta data. This resulted in better performamce and prevented our model from overfitting on the location as opposed to the environment.

We first implemented Behaviour cloning, followed by DAgger. We notices that the losses are converging as the number of epochs increase. Further in our implementation we are feeding our rewards into the next epoch of expert training. Through our experiments we were able to get better performance using Dagger, which is inline with our expectations. In the DAgger algorithm, one key challenge is tuning the reward function in a way that accurately reflects the expert's behavior and incentivizes the agent to mimic that behavior. This is important because the quality of the reward function can have a significant impact on the performance of the learned policy. We spent considerable time in fine tuning the rewards and trying out various API provided by the framework. The results and the video in different maze environments in presented in the result section and the project video.

For the controller, the action consists of (force, theta) pair. These forces along with the simulated resistive forces which include drag and mechanical resistance, allow us to calculate the required velocity for the robot wheels. The force is restricted between [0,1]. We further wrap the angle in theta ranges from $-\pi$ to π, which is then passed to the $\arctan()$ function to obtain required orientation. Our Implementation at a higher lever implements a PD position controller for actuating the steering joints of the wheels and a velocity controller for torque as torque controller requires

IV. Result

A. Environment and Execution Images

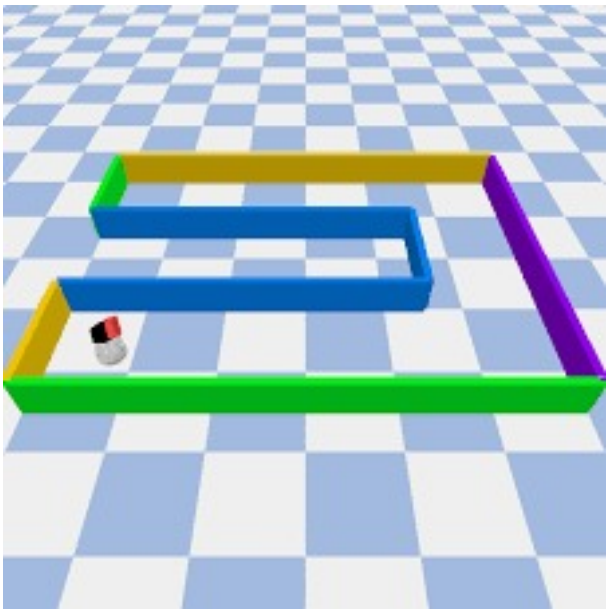

Fig. 1. U Maze environment with turtlebot

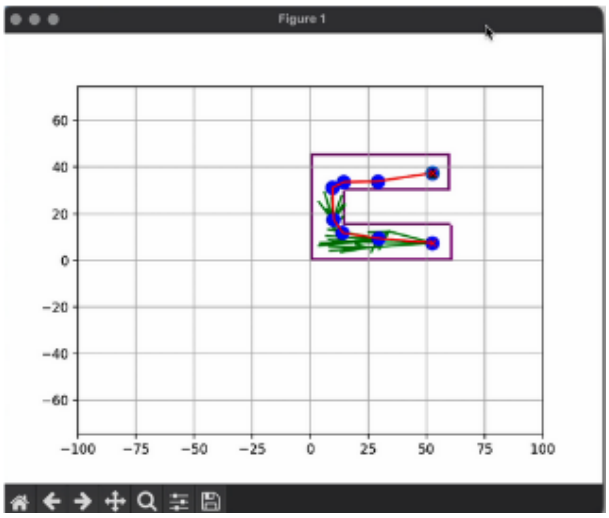

Fig. 2. RRT* in U Maze environment with turtlebot

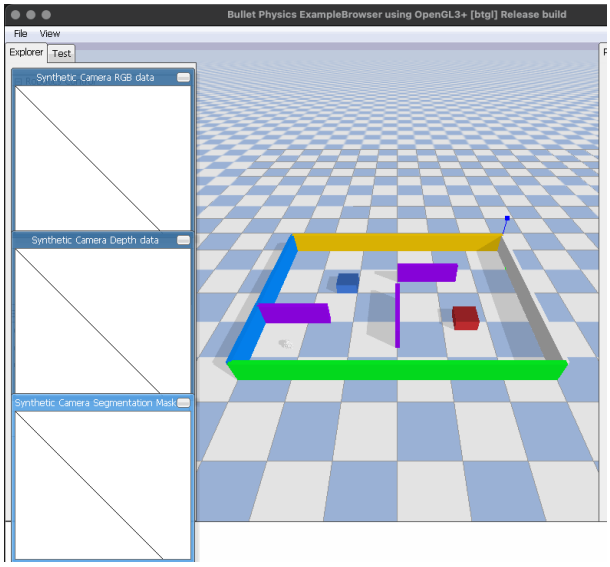

Fig. 3. Random Forest Maze environment with turtlebot

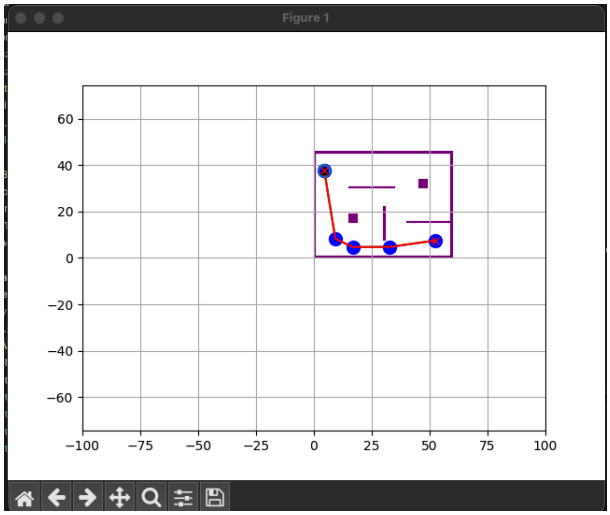

Fig. 4. RRT* in U Maze environment with turtlebot

References

[1] https://github.com/astrid-merckling/bullet_envs
[2] https://medium.com/@chand.shelvin/pybullet-getting-started-a068a0e3d492
[3] https://imitation.readthedocs.io/

_p.rayTest(rayFromPosition, rayToPosition, physicsClientId*)

objectUniqueId	int	object unique id of the hit object
linkIndex	int	link index of the hit object, or -1 if none/parent.
hit fraction	float	hit fraction along the ray in range [0,1] along the ray.
hit position	vec3, list of 3 floats	hit position in Cartesian world coordinates
hit normal	vec3, list of 3 floats	hit normal in Cartesian world coordinates

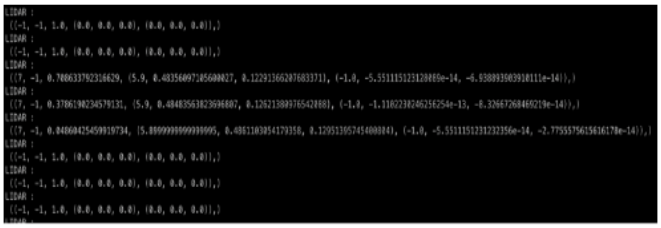

Fig. 5. Lidar metadata

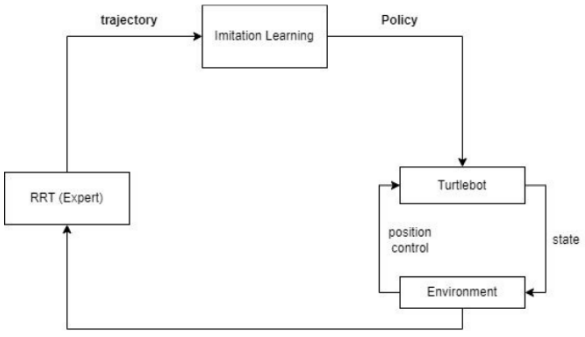

Fig. 6. Lidar metadata