MARL (Multi-Agent Reinforcement Learning) for navigation in grid environment

Junhyuck Woo, Shihan Zhao, Jiefan Yu, and Jyotirmoy Deshmukh

University of Southern California, Los Angeles CA 90007, USA.
Email: {junhyuck, shihanzh, jiefanyu, jdeshmuk}@usc.edu

Abstract— The food delivery problem is one of the hottest research topics in robotics communities. Many recent approaches are to use a single robot equipped with many sensors. The industry standard has been traditional global planning algorithms such as Dijkstra and A* combined with local planning algorithms. We solve this problem by utilizing more than one robot, and each robot will be controlled by a model trained by the reinforcement learning algorithm.

Keywords: Multi-Agent (MA), Reinforcement Learning (RL), Multi-Agent Reinforcement Learning (MARL), Navigation.

1 Introduction to the problem

1.1 Problem in using a single robot

Food industry began to apply robotics technology [1]. Among the application of the food industry, the food delivery robot is actively researched facing the Coronavirus disease (COVID-19) pandemic. In this trend, food delivery robots have been used in many restaurants. Many restaurants use a single robot to serve the food from the kitchen to customers' tables. One robot can handle the food delivery problem if the restaurant is small enough. However, if the scale of the place is increased, the robot will face the bottleneck problem: Too many foods are given to the robot so that the food will be delivered in delayed time. In the end, customers get food, not in the best condition.

1.2 Problem in using the traditional algorithm: Dijkstra, A*

The traditional global path planning algorithm such as Dijkstra and A* are mainly used to solve this problem. These methods work well but have a dimensional problem. It means that when the scale of the world is increased, the computation time increases exponentially.

2 Overview of approach

We believe that the problem can be solved by applying a multi-agent concept and uti-

lizing the reinforcement learning algorithm. Two robots replace a single robot, and reinforcement learning is used instead of the traditional algorithm.

2.1 Assumption

The food delivery process can be divided into two parts in the real world. The first part is to move from its location to the kitchen, and the second part is to move from kitchen to tables. Both movements have the same goal the moves from A to B. It is the same with the navigation problem, so our delivery problem is rephrased as a navigation problem. To simplify the problem and focus on the high decision process, we design our environment, restaurant, as a grid world. Each cell has two characteristics in this grid environment: occupied or vacant. Each cell can be occupied by a single object, such as a robot or obstacle, and if the obstacles or robots are on the cell, it is considered as an occupied state. If not, it is considered a vacant state. The collision is a condition that more than two agents try to move to the same cell. The main problem when using multi-agent is a collision with other agents. The minimum condition for this collision can be satisfied by two agents. Our project is for the proof of concept, so we decided to use two agents. The robots cannot have all information on a map in the real world, which means the robot is not fully observable. We assume that the robot can detect the environment near the two cells, so the observed areas are maximal 25 cells and minimal nine cells. We defined observation as the same size as the map and set the rest to zero except for the observed parts. And we add one more thing to our observation. When we find a path to go someplace, we know its destination. So we add destination information even if it is not in the observation area.

2.2 Approach

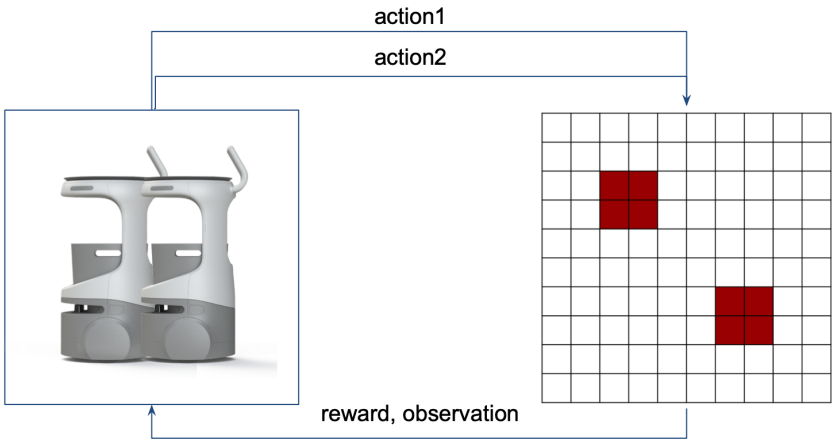

Fig. 1. Shows the relationship between two robots and environment.

We use the RL method as our main approach. RL trains models while interacting with the environment. Agents make actions in the world, and the environment returns the

updated world and a reward value. So, we build an environment that interacts with agents and a neural network making a decision of robots.

3 Summary of demo

We will show two demonstrations that are the result of our simulations. The top line represents the time steps, and the bottom part is a map (Environment and robots). The map is represented as numbers. 1 means the robots, 2 means the obstacles, 3 means the destinations, and 0 means the open space where robots can move. In each time step, robots can move to neighbor cells (up, down, left, right) or stay in the same location. If all robots reach their destinations, the simulation is over.

The first one was trained with the same initial locations, and the episode's length was 1000. It shows the optimal paths, but it turns out that the models did not work in the other environments, which means that the models are overfitted. So we trained the other models by giving various initial positions. We trained the models for 15000 episodes. Although it took 72 steps, all the robots reached destinations.

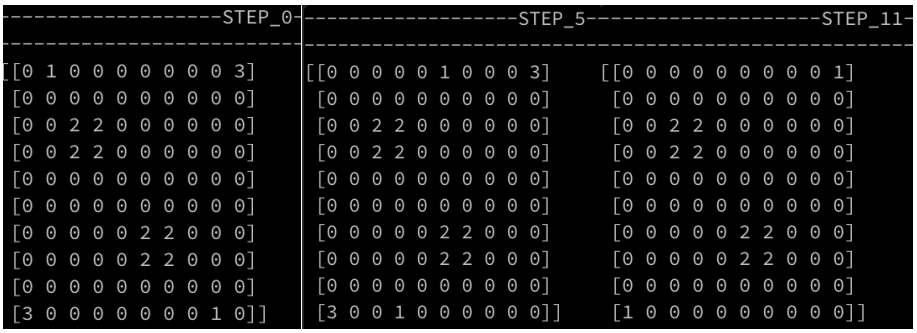

Fig. 2. The first demonstration snapshot: First step, middle step, and last step.

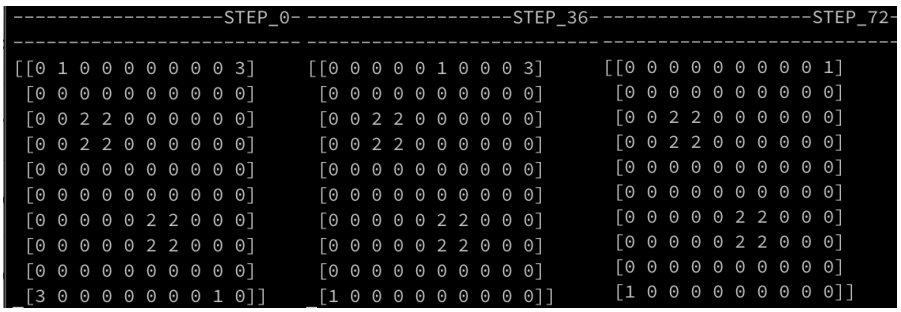

Fig. 3. The second demonstration snapshot: First step, middle step, and last step.

4 Implementation

In reinforcement learning, the interaction between environment and agent is one of the important factors. Aside from the choice of the RL algorithm itself, state, action, and reward design may have a huge impact on the RL performance. We will now talk about each of these in turn.

4.1 Integrated Development Environment

We used Python to implement the simulator because many libraries and packages support the language. We used two main packages. The first one is NumPy, and it helps us handle the matrix and array calculation. The second one is TensorFlow. It is one of the most famous tools for building a neural network. We use this package to build our decision-making model and save it using the Pickle module.

4.2 Reinforcement Learning

For the choice of RL algorithm, our initial intention had been to adapt a variant of an actor-critic method to our problem. We started off building an independent policy-based and another independent value-based solution to our MARL problem, hoping that we would be able to eventually assemble them into a full solution. We did not have time to finish the last step, but we did successfully built the two independent solutions. We will now talk about their implementations.

The policy-based solution we chose was a policy network with two dense layers trained with policy gradient. We trained one such network for each of the robots. Each network took the aforementioned clipped world grid surrounding its corresponding robot as the input and outputted the possibilities for choosing each of the five actions (i.e., up, down, left, right, stay still). Since there was not a critic to evaluate the outputs, the loss was simply the negated cumulative reward, i.e., summing up all the discounted rewards in a complete training episode. This meant the training was off-policy and carried out on a per-episode basis. To summarize, the general flow of the algorithm went as follows:

1. Initialize the grid environment and a set of policy networks.
2. The observation grid of each robot (the same that will be fed into its policy network) is saved.
3. Each network chooses an action for its robot, and all the chosen actions are saved.
4. The chosen actions are executed, with the resulting global reward saved.
5. Check if the episode terminates, if not goes back to step 2.
6. When the episode terminates, sum up all the recorded rewards.
7. The aforementioned reward, the recorded observations and actions will be used to train the network.

The value-based solution we chose was a DQN. Since the reward in our environment design was global, we just trained one global DQN critic. This DQN took the entire world grid as input and outputted the evaluations of all possible action assignments,

which would be 5^number_of_robots. Other than this, our critic was just a generic DQN with experience replay, so we will not linger on it too much.

4.3 Agent

Action. It is relatively straightforward, being the four directions up, down, left, and right on the grid map. When we presented the proposal, we received the suggestion to include an extra action "stay still", which we thought would be a nice addition, since we were specifically targeting scenarios such as two robots competing for a narrow corridor when we started this project, and one robot waiting for the other to pass would have been the reasonable solution to such scenarios. However, when we started training, we found that the action "stay still" discouraged the robots from exploring. There were often episodes where just one of the robots would move to one of the goals and then stay still, while the other robots would just stay still the entire time. We tried to address this by tailoring the reward structure.

Neural Network. We chose the neural network as our decision-making model. Since our map is small, we implemented the neural network in a simple form. It has three layers structure, including the input, output layer, and one hidden layer. The number of nodes in the hidden layer is 256. The model gets environment observation data as input. Since its shape is a 1D array with 100 lengths, the input layer has 100 nodes. Since our agents have five-movement options, the output layer has five nodes.

Class Variables. This class was built for handles all robots' data. The two main variables are networks and memories, and robots use the networks to decide, and the memories store all data during the episodes.

Class Method. Three methods were implemented. For each episode, robots first applied the method choose_action(). The robots chose one of the five actions with corresponding probability. Then the store_transition() method recorded the taken action and the resulting reward. The third method is the learn(). This method trains models after finishing the episode using the stored data.

4.4 Environment

State. Unlike the grid example we saw in class, the MARL problem we are trying to solve cannot simply use agent coordinates as the state. This is because the coordinates cannot capture all the dynamic information by themselves. In the grid example, all obstacles are static, but in our MARL problem, there are other moving robots that may change the utility of a coordinate by moving close to or away from it. Therefore, if we just used coordinates as states, there would not have been a set of "correct" q-values for RL to approximate since they would always be changing. For these considerations, we use a sub-section of the world grid itself as the state for each robot — we simply clip from the world grid an area of fixed dimensions around the robot to represent the limited visibility, and set all other cells to 0, except for the goals which are always

shown on the state regardless of distance from the robot. In hindsight, we could have probably engineered the state representation better since our current approach includes too much null information into the state. We could have, for example, explicitly used goal directions and distances from the robot as representations for the goal locations. We will take note of this lesson when we work on similar projects in the future.

Table 1. Map information: the meaning of each cell.

Cell number	Meaning
0	Open space
1	Robot
2	Obstacle
3	Destination

Reward. We initially set a high reward for reaching all the goals and a negative reward for any collisions, with all other cases, including reaching only some of the goals, giving zero rewards. However, we soon found this all-or-none style of reward to be too steep. To help guide the robots towards goals, we also set a negative reward (-1) when it overs the maximum step number to discourage wandering. At the same time, we assigned -0.1 for each valid movement to discourage wandering, looking forward to the robots finding the shortest path. We also realized that adding a positive reward for moving, combined with our environment design which terminated an episode upon collision, replaced the need to have a negative reward for collision, so we just set the reward for a collision to be -2.

Table 2. Reward function structure.

Action	Reward
Move	-0.1
Collide	-2
Reach to max step	-1
Reach destination	100 * (1 - 0.9 * (step / max_steps))

Class Variables. This class handles three main data, map (grid environment), location of robots, and goal locations. The map is 10 by 10 size 2D array, and it has obstacles, robots, and destination locations. Since the map is a 2D array, the location of a cell has a 1D array with length 2. We consider the two agents, so the location of robots and destination have the same shape: 2 by 2 size 2D array.

Class Method. Three main methods were implemented. The first one is reset(). The map should be reset for every new episode to start a new simulation. The robot location and destination are randomly assigned to face various circumstances in this method. The second one is observe(). This method returns the environment data observed by robots. By following our assumption, the 100 sizes 1D array is returned. The last one

is step(). This method calculates the effect of actions and returns rewards, observation, and terminal conditions. The terminal conditions check whether all robots are located at the goal positions.

5 Conclusion and related work

We simulated within various conditions and reward functions. Without random initialization, we could get a model that works well, but it turns out that it is overfitted, so we could not use it at a different setup. To make a model that works well under various initial conditions, we assigned a new location and destination in each episode. From this setting, we can get models that achieve the goal. However, the path was not optimal. There are papers related to our problem. [3] The Multi-Agent Pickup and Delivery (MAPD) problem is handled by applying the Token Passing method. [4] applies reinforcement learning to solve multi-agent motion planning problems. [5] solves the Multi-Agent Path Finding (MAPF) problem by using reinforcement learning and imitation learning. By comparing those works, we concluded that our limitation could be overcome by improving the neural network structure and increasing the training time. Large and complicated neural networks are used to solve the problems, and the large structure requires long training episodes. So focusing on the structure of neural networks would be future works in this project.

References

1. Iqbal, Jamshed, Zeashan Hameed Khan, and Azfar Khalid: Prospects of Robotics in Food Industry. Ciência e tecnologia de alimentos vol. 37 (2), pp. 159–165. (2017).
2. Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. Advances in Neural Information Processing Systems 12, pp. 1057–1063 (2000).
3. Ma, Hang, Wolfgang Hönig, T. K. Satish Kumar, Nora Ayanian, and Sven Koenig. Lifelong Path Planning with Kinematic Constraints for Multi-Agent Pickup and Delivery. Proceedings of the ...AAAI Conference on Artificial Intelligence vol. 33, pp. 7651–7658. (2019).
4. Arbaaz Khan, Chi Zhang; Shuo Li; Jiayue Wu; Brent Schlotfeldt; Sarah Y. Tang; Alejandro Ribeiro; Osbert Bastani; Vijay Kumar. Learning Safe Unlabeled Multi-Robot Planning with Motion Constraints. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 7558-7565. (2019).
5. Sartoretti, Guillaume, Justin Kerr, Yunfei Shi, Glenn Wagner, T. K. Satish Kumar, Sven Koenig, and Howie Choset. PRIMAL: Pathfinding via Reinforcement and Imitation Multi-Agent Learning. IEEE robotics and automation letters, vol. 4, no. 3, pp. 2378–2385. (2019).