Dynamical maze solver problem

Han Zhang hz5g21@soton.ac.uk

Abstract

Reinforcement learning has been a hot trend in recent years. In this report, a reinforcement learning algorithm is used to solve the dynamic maze problem. The program has achieved good results. The training phase is accelerated by adding the agent's past path to the state that the agent can observe, and a solution to the dynamic maze is given.The Code is available here. (https://github.com/Hansan-clouds/COMP6247-Reinforcemnt-Online-Learning.git).

1 Introduction

The requirement for this assignment is to use reinforcement learning algorithms to solve a sizeable dynamic maze problem. The agent goes from the starting point (1,1) to the maze's exit at (199,199), as shown in Figure 1. There will be fire randomly generated around the agent. The agent can only see the surrounding nearly nine squares, and the overall structure of the maze is not accessible to our agent.

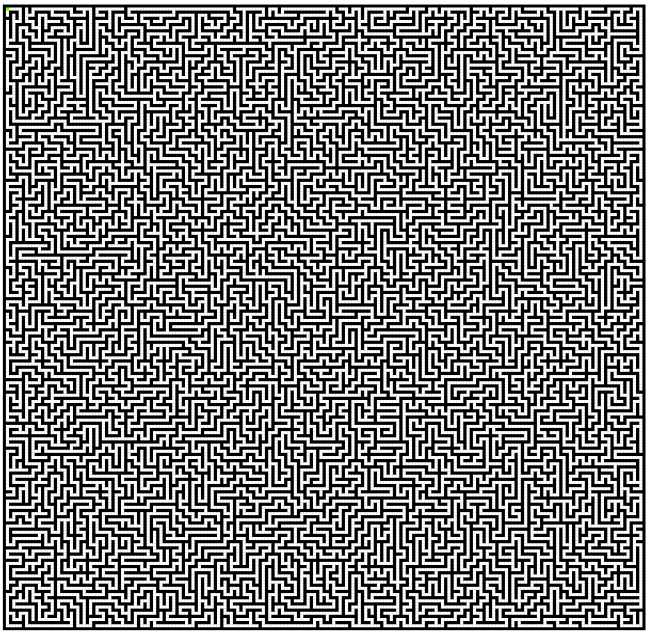

Figure 1: The structure of DQN algorithm

In this report, the Deep Q-learning Network(DQN) is used to solve this problem and achieves good results. Next, I will introduce the method used, algorithm steps, and structure of DQN in section 2. In section 3, the specific implementation details will be explained. Section 4 is to show the results and discusses the program performance and the last section is to summarize this report.

2 Methodology

DQN is an algorithm that combines deep learning and reinforcement learning. Its structure is shown in Figure 2. It consists of 5 agents, namely environment, agent, Experience Reply buffer, main network, and the target network.

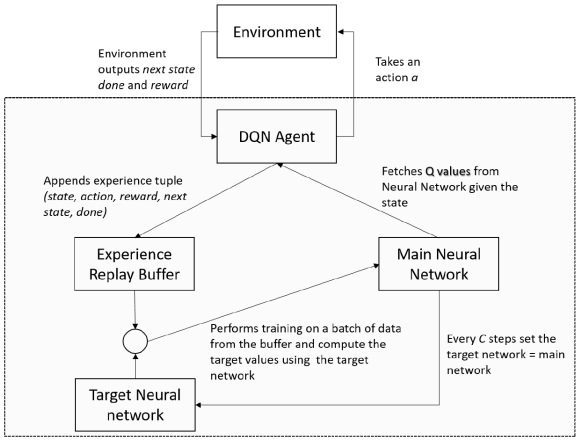

Figure 2: The structure of DQN algorithm [1]

In those parts, the environment is the external environment to be interacted with and the agent is what we train. The main network is used to predict the Q value, and the target network is to keep a copy of the neural network and use it for the Q(s', a') value in the Bellman equation. That is, the predicted Q-values of the second Q-network are used to error backpropagate and train the main Q-network. Experience Reply buffers are large buffers that allow the agent to have past experience and sample training data from it. The buffer contains a set of experience tuples (S, A, R, S'). Tuples are gradually added to the buffer as we interact with the environment.

The procedure of the DQN algorithm[1] is shown in Figure 3. The γ is the discount factor. C is the steps to update the target network using the main network.

Initialize network Q
Initialize target network $\hat{Q}$
Initialize experience replay memory D
Initialize the *Agent* to interact with the Environment
while *not converged* **do**
  /* Sample phase
  $\epsilon \leftarrow$ setting new epsilon with ϵ-decay
  Choose an action a from state s using policy ϵ-greedy(Q)
  Agent takes action a, observe reward r, and next state s'
  Store transition $(s, a, r, s', done)$ in the experience replay memory D
  if *enough experiences in* D **then**
    /* Learn phase
    Sample a random *minibatch* of N transitions from D
    for *every transition* $(s_i, a_i, r_i, s'_i, done_i)$ ***in*** *minibatch* **do**
      if $done_i$ **then**
        $y_i = r_i$
      else
        $y_i = r_i + \gamma \max_{a' \in A} \hat{Q}(s'_i, a')$
      end
    end
    Calculate the loss $\mathcal{L} = 1/\mathrm{N} \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
    Update Q using the SGD algorithm by minimizing the loss $\mathcal{L}$
    Every C steps, copy weights from Q to $\hat{Q}$
  end
end

Figure 3: The structure of DQN algorithm [**?**]

[1]Reference: DQN Algorithm

3 Implementation Details

My program is mainly divided into five parts, including Qmaze, network, Replay Buffer, agent, and main. I will explain how they interact and some details part in the code. Figure 4 shows their direct interaction diagram and the algorithm flow used.

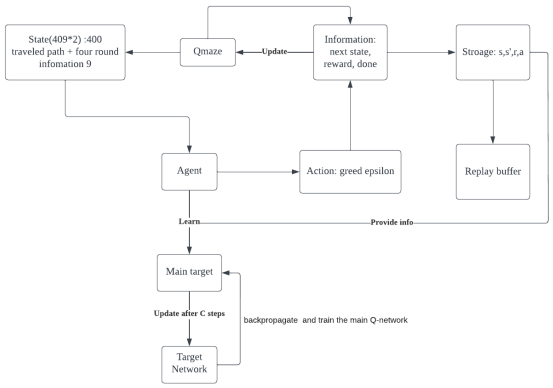

Figure 4: The structure of DQN algorithm [**?**]

1. **Qmaze:** it defines the properties of the maze, and the surrounding environment, defines the information states, rewards, and action dictionaries that the agent can see, and finally prints the maze and outputs the output file.
2. **ExperienceReplayBuffer:** it is used to store states, next states, actions,s and reward data, where states contain the paths the agent has traveled before and the information around the environment he observes.
3. **Network:** it builds 2 fully-connected networks with two hidden layers, each with 100 units. The input is state information, which is a two-dimensional input (dimensions of a state), and the outputs are 5 actions. The SGD optimization function is used and MSE as an error function
4. **Agent** it is used to set the agent and various hyperparameters. The agent takes an epsilon-greedy action and the preferred action is found from DQN. Store the information in the Replay Buffer, and finally define and optimize the loss function.
5. **Main:** It is used to load the maze, and start training, there are many attempts in one training until Qmaze returns done information, then learn the network, update states, store information, print scores, and maze paths.

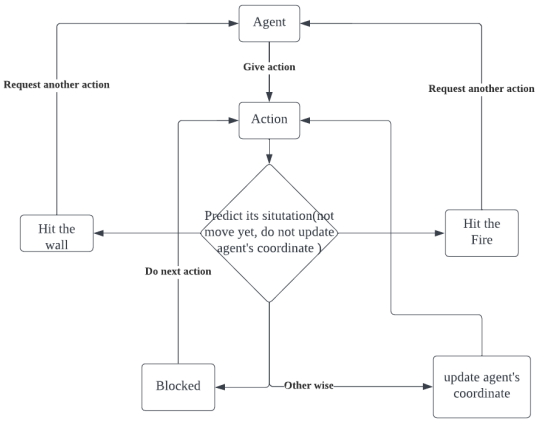

Figure 5: The structure of DQN algorithm [2]

Figure 5 shows the behavioral pattern of the agent. In each step, the next step is predicted in advance, which ensures that we will not encounter fire, hit a wall, and the agent's coordinates will run out of the maze.

Code 1: Reward and action function

```
action = {"0": 'stay',"1": 'up',"2": 'left',"3": 'down',"4": 'right',
rewards = {"move_forward": 0.7,"move_backward":-0.2,"re-visited":-0.5,
"blocked":-0.8,"fire":-1.0,"wall":-0.9,"stay":-0.3,"success":100.0,}
```

The moveforward will be given when the Manhattan Distance between the agent and the exit decrease, and the movebackward is opposite.

Code 2: How to act when meet fire.

```
#get action
act_key = str(action)
# relative move
x_relative, y_relative = action_dir[act_key]['step']
x, y = self.rat
x_expected, y_expected = (1 + x_relative, 1 + y_relative)
# get local info
local_matrix = self.get_around_info
# check first before move
if local_matrix[y_expected][x_expected][0] == 0: # wall
    return self.get_state, rewards_dir['wall'], False
if local_matrix[y_expected][x_expected][1] > 0: # fire
    return self.get_state, rewards_dir['fire'], True
```

Whenever the agent returns an action, we will predict whether it will hit the wall and whether it will encounter fire. If it hits a wall, it will not update the agent coordinates and start the next attempt. If it encounters fire, it will not update coordinates, and end this iteration.

Code 3: How to act when blocked by fire and stay.

```
# whether block
observation = self.observe_environment
is_blocked = False
```

```
if observation[1] !=1 and observation[3] != 1 and observation[5] != 1 and
    observation[7] != 1:
    is_blocked == True

# blocked, need to wait the fire to dispear
if action_dir[act_key]['id'] == 'stay' and is_blocked:
    return self.get_state, rewards_dir['blocked'], False
elif action_dir[act_key]['id'] == 'stay':
    # lazy rat and give punishment
    return self.get_state, rewards_dir['stay'], False
```

When the agent can't move in 4 directions, let him stay in place and give him a penalty. If it is estimated to stay in place, give punishment.

Code 4: State setting.

```
def get_state(self):
    col, rol = self.get_rat_pos
    location = get_local_maze_information(rol,col)
    temp = []

    # states include observation 9
    for i in range(3):
        for j in range(3):
            temp.append((rol-1+i,col-1+j))

    #add latest 400 moves into states
    if len(self.rat_path) <=400:
        for i in range(400):
            temp.append((199,199))
    else:
        for j in range(1,401):
            temp.append(self.rat_path[-j])

    self.state = temp
    return self.state
```

States that agent can observe is the path traveled and surrounding information.For the parameters set, the algorithm setting is completely referring to the Instructions book[1].

Environment used: This project uses jupyter notebook as the compiler, and the library uses pytorch, matplotlib, etc.

4 Results and Discussion

The results of the score and the average score are shown in Figure 6 and the traveling path shows in figure 7. Due to the time limit, the network has only been trained for 5000 epochs, and the agent has walked 100,000 steps. It can be seen from the left picture that the evaluation distribution fluctuates violently, and there are many outliers. For example, at 62000, the reward is -120. Looking at the left picture, in this attempt, the agent took 200 steps, which means that the agent falls into a trap, that is, it predicts to go to point B at point A, and predicts to go to point A at point B, and falls into an infinite loop. In the left image, each attempt took

about 50 steps, proving that the agent is still in the early stages of exploration.

All in all, It can be seen that the training speed of the network is slow, and the reward is oscillating up and down, which proves that it is still in the early stage of learning. The algorithm has not converged yet.

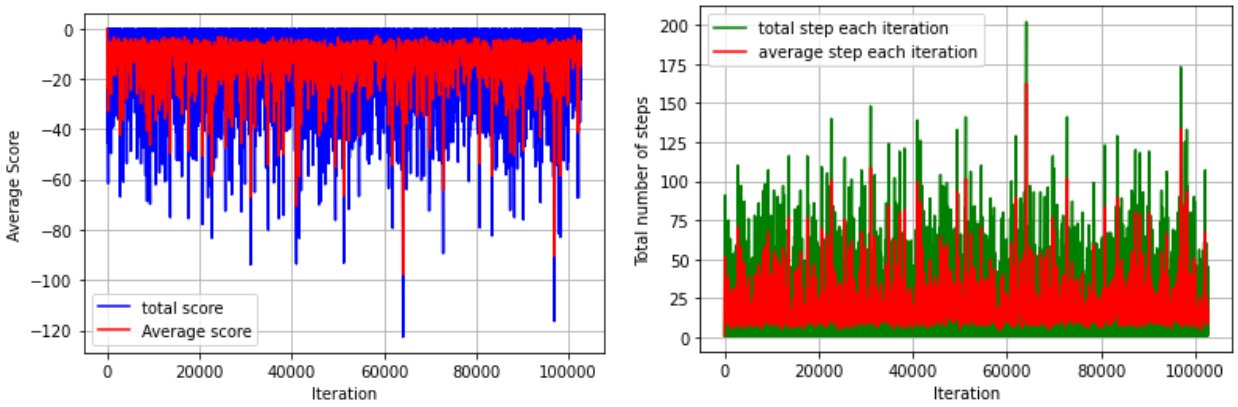

Figure 6: Score with each epochs

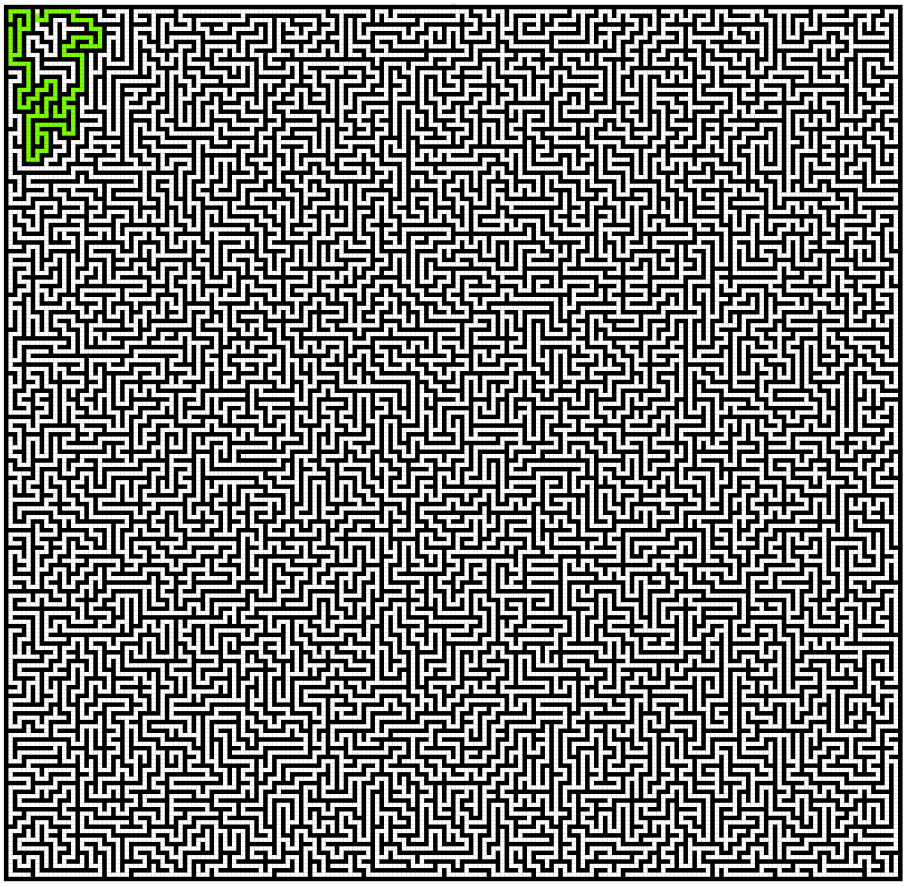

Figure 7: Path tracing

There are many reasons for the slow convergence of this program, I think there are four main reasons.

1. The agent falls into a trap, it is an infinite loop once got into it, It will cost a lot of time.

2. The first is that the maze is too large, and it is difficult for the agent to find the exit path in a short time, resulting in slow program convergence.
3. The second is that the agent only knows the surrounding information but not the information of the entire maze. An information barrier appears. At this time, the agent turns around like a housefly, and there is no other prompt except the reward based on action. It is the same as a person only knowing the surrounding information, then he must need a lot of time for trial and error.
4. The third is the setting of action and reward. Without a good reward function, it takes a lot of time for the agent to find the law of the maze.
5. The last one may be a parameter setting problem. Setting a good parameter will also speed up the convergence.

5 Conclusion

This report discusses and implements a deep q-learning network to solve the maze problem of dynamic mazes, and DQN can solve dynamic mazes. This problem can be effectively addressed through a prediction step and a framework that utilizes past information gathered while traversing the maze. The problem that the maze converges too slowly is further discussed.

References

[1] Luis Santos. Dqn parameter instruction. https://github.com/luispsantos/EL2805-Reinforcement-Learning/blob/main/lab2/lab2$_i$*nstructions.pdf*, 2020.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. CoRR, abs/1312.5602, 2013.