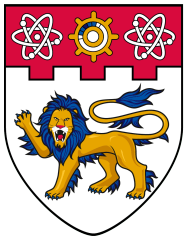

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Nanyang Technological University

College of Computing and Data Science

Assignment I: Reinforcement Learning

Author:
Deng Jie G2403273K

Instructor:
Bo An, Professor

An Assignment Submitted for NTU's Course:

[AI601] Introduction to AI and AI Ethics

October 17, 2024

Contents

1 Requirement

This project requires you to implement and evaluate one of the Reinforcement Learning (RL) algorithms (e.g., Q-learning, SARSA, etc.) to solve the CliffBoxPushing grid-world game. Novel ideas are welcome and will receive bonus credit. In this assignment, you need to implement the code on your own and present a convincing presentation to demonstrate the implemented algorithm.

The environment is a 2D grid world as shown in 1. The size of the environment is 6×14. In 1, A indicates the agent, B stands for the box, G is the goal, and x means cliff. You need to write code to implement one of the RL algorithms and train the agent to push the box to the goal position. The game ends under three conditions:

1. The agent or the box steps into the dangerous region (cliff).
2. The current time step attains the maximum time step of the game.
3. The box arrives at the goal.

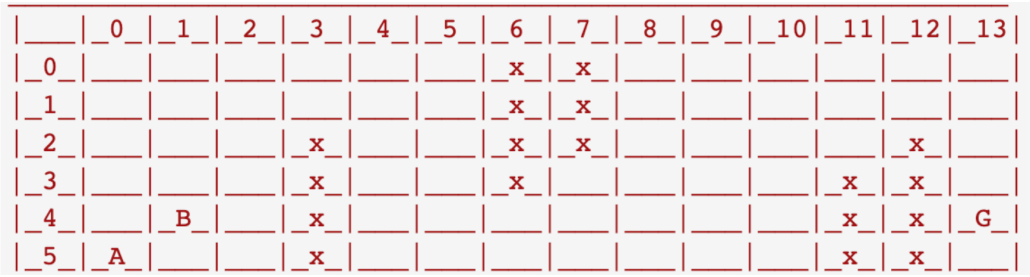

Figure 1: The Cliff Box Pushing Grid World.

The grading criterion are as follows (total 100 marks):

Item	Marks
Bug-free: correctly implement the code of your chosen RL algorithms and visualization	50%
Plot the learning progress: episode rewards vs. episodes	25%
Final V-table (shown in the gird) and the **policy**.	25%

Figure 2: The Marking Criteria

A bonus of 10 mark is awarded for each of the following task:

- Compare different exploration techniques (such as UCB) with the default epsilon-greedy and non-exploration strategy. Come up with your analysis.
- Use only sparse reward (the agent only receives a reward of 1000 if the box reaches the goal position) and implement your ideas to solve this problem (hint: exploration technique, hierarchical RL, etc.).

Nevertheless, your final project mark will still be capped at 100 even if the total mark plus the bonus mark exceeds 100.

2 Task 1: Complete the Code

2.1 RLAgent Learning

Reinforcement learning (RL) is an interdisciplinary area of machine learning and optimal control concerned with how an intelligent agent ought to take actions in a dynamic environment in order to maximize the cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning. [1] There are several RL algorithms, such as Q-learning, Sarsa, Deep Q Network, Policy Gradients and so on. In this assignment, I implemented the **Q-learning** algorithm to solve the Box Pushing problem.

The following code block is the initial function of Q-learning. Q-learning training function implementation to update the Q-table based on the equation below:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(s',a') - Q(s,a) \right) \tag{1}$$

In Equation 1, $Q(s,a)$ means taking action a: up, down, left or right in the state s. R_{t+1} means the obtained rewards after taking action a. The $\max_{a'} Q(s',a')$ means the maximum Q-value that the agent can obtain by taking the best action a' starting from state s'.

And, γ is called discount factor which is used to balance the influence of $\max_{a'} Q(s',a')$. Similarly, α is called learning rate which is used to balance the old and new value of Q.

Firstly, we declare some variables:

Code 1: Initial Function of Q-learning

```
def __init__(self, env, num_episodes, epsilon=0.1, alpha=0.1, gamma
    =0.99):
    self.action_space = env.action_space
    self.q_table = dict() # Store all Q-values in a dictionary
    self.env = env
    self.num_episodes = num_episodes
    self.epsilon = epsilon
    self.alpha = alpha
    self.gamma = gamma
```

- action_space: is the list [1, 2, 3, 4] which means [up, down, left, right]. The every step of moving action will be chosen from this list;
- num_episodes: is the episode number of interaction with environment. In each episode, the agent will start from a initial position until episode ends (reaches the goal, fails);
- gammma: is the γ in Equation 1;
- alpha: is the α in Equation 1;
- epsilon: controls the exploration vs. exploitation trade-off in reinforcement learning. We will discuss it in detail in the later chapter.

2.2 Exploration vs. Exploitation

Exploration allows an agent to improve its current knowledge about each action, hopefully leading to long-term benefit. Improving the accuracy of the estimated action-values, enables an agent to make more informed decisions in the future.

Exploitation on the other hand, chooses the greedy action to get the most reward by exploiting the agents current action-value estimates. But by being greedy with respect to action-value estimates, may not actually get the most reward and lead to sub-optimal behaviour.

When an agent explores, it gets more accurate estimates of action-values. And when it exploits, it might get more reward. It cannot, however, choose to do both simultaneously, which is also called the exploration-exploitation dilemma [2]. After a period of learning, the fixed values in the table will make each route fixed, and the environment cannot be effectively explored, which will easily lead to the problem of local optimality.

Epsilon-greedy is a great strategy to balance exploitation and exploration, where epsilon refers to the probability of choosing to explore, exploits most of the time with a small chance of exploring. The following code block is the action function of Q-learning:

Code 2: Action Function

```
def act(self, state, is_training = True):
    """Returns (epsilon-greedy) optimal action from Q-Value table."""
    if np.random.uniform(0,1) < self.epsilon and is_training:
        action = self.action_space[np.random.randint(0, len(self.
            action_space))]
    else:
        q_values_of_state = self.q_table[state]
        maxValue = max(q_values_of_state.values())
        action = np.random.choice([k for k, v in q_values_of_state.
            items() if v == maxValue])

    return action
```

We randomly generate a number and compare it with epsilon. If it is less than epsilon, we adopt the exploration strategy and randomly select an action. If it is greater than epsilon, we adopt the exploitation strategy and select the best action. The following code block is the learning process of Q-learning algorithm:

Code 3: Learning Process of Q-Learning Algorithm

```
def learn(self):
    """Updates Q-values iteratively."""
    rewards = []
    for _ in range(self.num_episodes):
        cumulative_reward = 0 # Initialise values of each game
        state = self.env.reset()
        done = False
        while not done: # Run until game terminated
            # TODO: Update Q-values
            action = self.act(state)
            next_state, reward, done, info  = self.env.step(action)
```

```
12              current_q_value = self.q_table[state][action]
13              if not done:
14                  max_next_q_value = max(self.q_table[next_state].
                        values())
15                  target_q_value = reward + self.gamma *
                        max_next_q_value
16              else:
17                  target_q_value = reward
18              self.q_table[state][action] = current_q_value + self.
                    alpha * (target_q_value - current_q_value)
19              state = next_state
20              cumulative_reward += reward
21          rewards.append(cumulative_reward)
```

3 Task 2: Episode Rewards vs. Episodes

The following is the comparison of the random agent and reinforcement agent learning process:

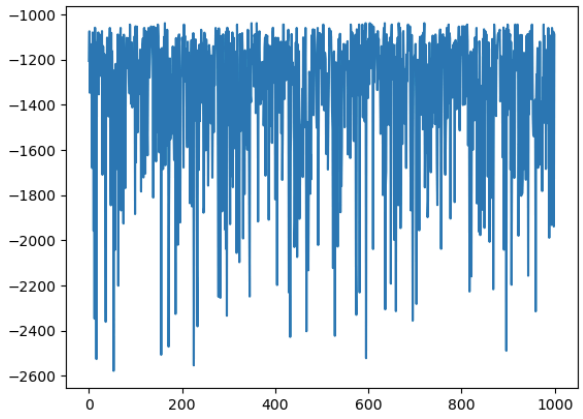

Figure 3: Random Agent

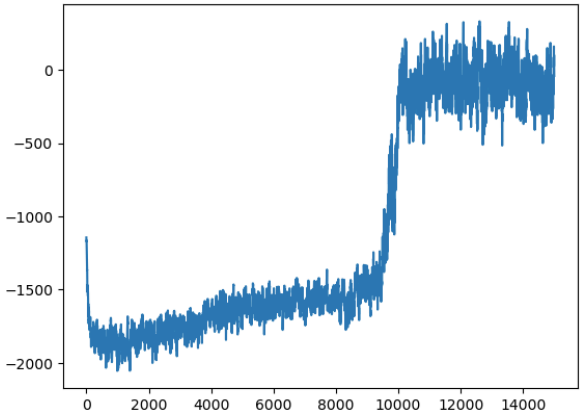

Figure 4: RL Agent with Epsilon = 0.1

As Figure 3 shows, if we use the strategy of random exploration, it will be difficult to win the game, because the movement is random, so the lack of patterns in the chart is consistent with cognition. Compared with Figure 4, we can intuitively see that after around 10000 episodes, the agent gradually learn a optimal path to the position G. Because agent will spent many timesteps to arrive the position G, the accumulated rewards will not be to high. In summary, after 10000 episodes, the agent will choose a relatively stable and optimal path so from the Figure 4 we can see that the reward value stabilizes as the episodes increase.

Next, we will try the different values of alpha and epsilon:

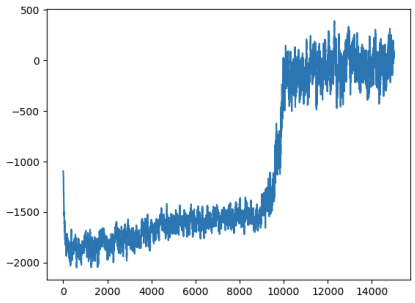

Figure 5: RL with Alpha = 0.1

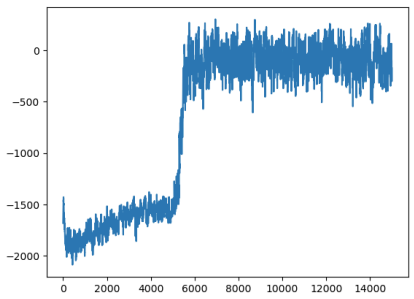

Figure 6: RL with Alpha = 0.2

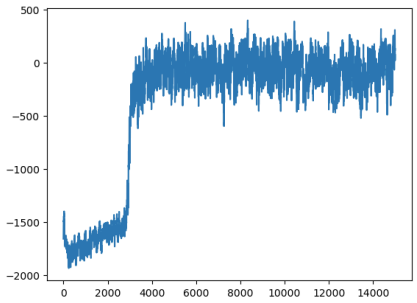

Figure 7: RL with Alpha = 0.5

From the above three figures, we can see that as alpha increases, it means that the impact of new value on old value increases. The agent tends to choose the optimal path, so the episodes required to obtain the optimal path will decrease(around 10000 in Figure 5, 6000 in Figure 6, 4000 in Figure 7).

4 Task 3: Final V_Table and the Policy

The following code block is to visualize v_table and policy. This code will print the largest 20 state values in v_table and the corresponding policy.

Code 4: V_Table and Policy

```
def visualize(q_table):
    v_table = {}
    policy = {}
    for key, v in q_table.items():
        v_table[key] = max(v.values()) # calculate the value function
        policy[key] = max(v, key=v.get) # calculate the policy
    state_num = len(q_table.keys())
    print(f"State space: {state_num}")

    # Print the largest 20 state values in v_table and the
       corresponding policy
    for k, val in sorted(v_table.items(), key=lambda x: x[1], reverse
       =True)[:20]:
      print("v_table", k, val / state_num)
      print("policy", k, policy[k])

visualize(agent.q_table)
```

After reinforcement learning process, the agent will find the optimal path, next code block will visualize the policy to move.

Code 5: Policy Visualization

```
env = CliffBoxGridWorld(render=True)
state = env.reset()
env.print_world()
done = False
rewards = []

while not done: # Run until game terminated
    action = agent.act(state)
    next_state, reward, done, info = env.step(action)
    state = next_state
    rewards.append(reward)
    print(f'step: {env.timesteps}, state: {state}, actions: {action},
        reward: {reward}')
    env.print_world()

print(f'rewards: {sum(rewards)}')
print(f'action history: {env.action_history}')
```

The output of Code 4 and Code 5 are saved in *v_table_and_policy.txt* and *policy.txt* respectively. Readers can open the folder. Readers can find these two text files and review them in the **code** folder which is in the same directory as this report.

References

[1] Wikipedia contributors, "Reinforcement learning," 2024. Accessed: 2024-10-15.

[2] GeeksforGeeks, "Epsilon-greedy algorithm in reinforcement learning," 2024. Accessed: 2024-10-15.