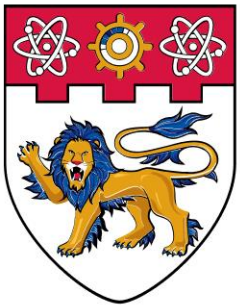

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

AI6101 INTRODUCTION TO AI & AI ETHICS

Reinforcement Learning Assignment

Submitted by

Li Xiaochen

Matric No. G2102142D

School of Computer Science and Engineering

2021/10/17

Introduction

Reinforcement learning (RL) is one of the important branches of Machine Learning, which emphasizes how to act based on the environment to maximize the expected benefits. It is training based on the interaction between the agent and the environment, rewarding the desired actions and punishing the undesired actions to maximize the total rewards. There are several RL algorithms, such as q-learning, Sarsa, Deep Q Network, Policy Gradients, and so on.

In this report, we will discuss about how to implement one of the RL algorithms to solve the Box Pushing game. Q-learning has been chosen as the RL algorithms for this assignment. Q-learning is value-based RL algorithm. Q is Q(S, A) which is the expected profit can obtain when taking action A under state S at a certain moment, and the environment will feedback the corresponding reward R according to the agent's action. Therefore, the main idea of q-learning is to build the State and Action into a Q-table to store Q values, and then select the actions that can obtain the maximum benefits/rewards according to the Q values.

The environment and background of the Box Pushing Game will be introduced below.

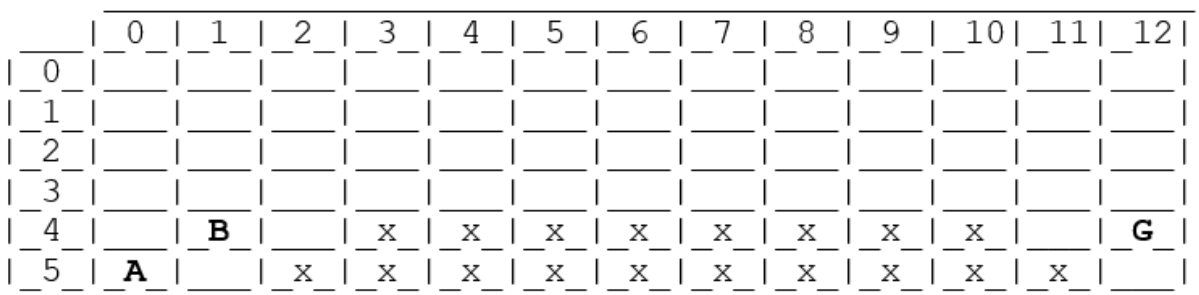

Figure 1. Box Pushing Game Environment

The environment is a 2D grid world as shown in Figure 1, and the size of the environment is 6×14 based on the codes in environment.py file. A indicates the agent and the start position is (5, 0), B stands for the box and the start position is (4, 1), G is the goal position at (4, 12), x means lava or cliff.

Q-Learning Process & Codes Snippets

The code snippets below represent the main steps of the Box Pushing Game implementation with Q-learning:

1. Declare the variables

```
def __init__(self):
    self.action_space = [1,2,3,4]
    self.Q = defaultdict(lambda: np.zeros(len(self.action_space)))
    self.discount_factor=0.99
    self.alpha=0.5
    self.epsilon=0.01
```

- **Action_space** is [1, 2, 3, 4] which is corresponding to [up, down, left, right]. The agent needs to choose one of the actions to navigate in the environment.
- **Q** stands for the Q-table, which will store the q-values for 4 different actions.
- **Discount_factor (γ)** is a number between 0 and 1, it determines how much the agent cares about rewards in the distant future relative to those in the immediate future. If discount factor equals to zero, the agent will only learn about actions that produce an immediate reward. If it equals to one, the agent will take each action based on the total of all the future rewards.
- **Alpha (α)** is the learning rate, it's a number between 0 and 1. The Q-values will be never updates if the learning rate equals to zero, hence nothing is learned. If setting to a high value such as 0.9 means that the learning can occur quickly.
- **Epsilon** will control the action selection to choose best action or randomly choose the action.

2. Choose Actions

```
def take action(self, state):
    if random.random() < self.epsilon:
        action = random.choice(self.action_space)
    else:
        action = self.action_space[np.argmax(self.Q[state])]
    return action
```

In this assignment, the epsilon will be set as 0.01, when random value larger than epsilon, the agent will take the best action, else it will take the random action. When epsilon become smaller, the agent will be greedier to take the best action.

3. Train function

```
def train(self, state, action, next_state, reward):
    '''q-learning algorithm implementation'''
    q_predict = self.Q[state][action-1]
    q_target = reward + self.discount_factor * max(self.Q[next_state])
    self.Q[state][action - 1] += self.alpha * (q_target - q_predict)
```

Q-learning training function implementation to update the Q-table based on the equation below:

$$Q_{new}(S_t, A_t) \leftarrow Q_{old}(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q_{old}(S_{t+1}, a) - Q_{old}(S_t, A_t))$$

4. Build the training loop to run 1000 episodes (some of the printing and chart plot codes were removed here in order to show the training loop clearly.)

```
if __name__ == '__main__':
    env = CliffBoxPushingBase()
    agent = QAgent()
    teminated = False
    rewards = []
    time_step = 0
    num_iterations = 1000
    for i in range(num_iterations):
        env.reset()
        while not teminated:
            state = env.get_state()
            # RL choose actions based on the state
            action = agent.take_action(state)
            # RL take actions and rewards
            reward, teminated, _ = env.step([action])
            # get next state
            next_state = env.get_state()
            rewards.append(reward)
            time_step += 1
            # RL learn and train based on the q-learning formula
            agent.train(state, action, next_state, reward)
        teminated = False
        rewards = []
```

Results Analysis

After 1000 training episodes, the agent A can find the correct and shortest way to push the box B to the goal position G. The best solution is shown as below:

[[4], [1], [3], [1], [4], [4], [4], [4], [4], [4], [4], [4], [4], [4], [4], [1], [4], [2]]

Which can be translated to:

Right → up → left → up → right → right → right → right → right → right → right → right → right → right → up → right → down

Figure 2 below shows the best solution more visually.

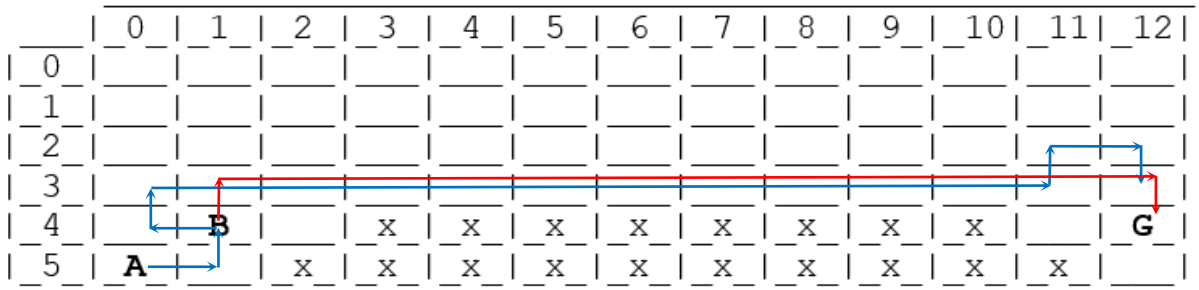

Figure 2. Agent moving route (blue) & box move route (red)

The reward for each action in each episode can be printed as well and it's calculated based on the formula below, the total rewards for each episode can by calculated by sum up the reward for each action, for the best solution, the total rewards will be -153

$$r_t = \begin{cases} -1 - distance\ (box, goal) - distance\ (box, agent) & if\ no\ falling \\ -1 - distance(box, goal) - distance(box, agent) - 200 & if\ falling \end{cases}$$

step: 35023, actions: 4, reward: -13
step: 35024, actions: 1, reward: -14
step: 35025, actions: 3, reward: -15
step: 35026, actions: 1, reward: -14
step: 35027, actions: 4, reward: -13
step: 35028, actions: 4, reward: -12
step: 35029, actions: 4, reward: -11
step: 35030, actions: 4, reward: -10
step: 35031, actions: 4, reward: -9
step: 35032, actions: 4, reward: -8
step: 35033, actions: 4, reward: -7
step: 35034, actions: 4, reward: -6
step: 35035, actions: 4, reward: -5
step: 35036, actions: 4, reward: -4
step: 35037, actions: 4, reward: -3
step: 35038, actions: 1, reward: -4
step: 35039, actions: 4, reward: -3
step: 35040, actions: 2, reward: -2
episode: 999, rewards: -153

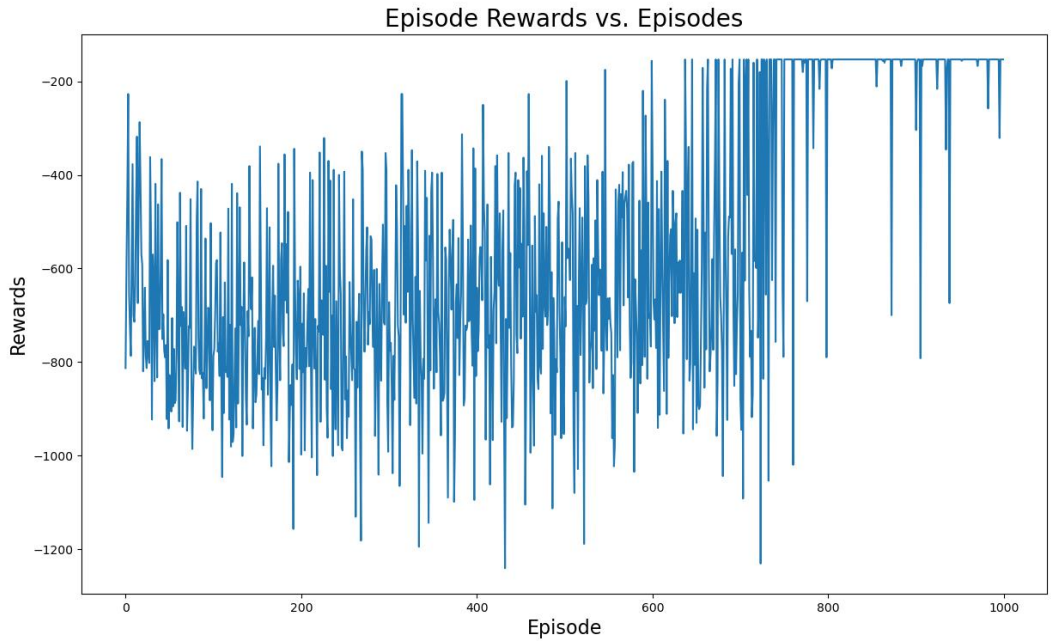

Figure 3. Episode Rewards vs. Episode

The trend chart for Episode Rewards vs. Episodes is shown below in figure 3. Based on the trend chart, we can observe that the agent is able to realize the best solutions after around 750 episodes because the agent can push the box to the goal position more frequently without making any mistakes, which can give the best total rewards -153 per episode. However, the agent will still be making mistakes or detours even after the 750 episodes training.

```
"(Agent), (Box)","[up, down, left, right]"
"((5, 0), (4, 1))",[-144.99297412 -155.71351039 -155.24022917 -144.26653314]
"((4, 0), (4, 1))",[-141.80065399 -136.73242598 -144.15164137 -135.15536069]
"((3, 0), (4, 1))",[-138.08211697 -134.28821901 -132.52840567 -134.39750873]
"((2, 0), (4, 1))",[-133.33981354 -132.86134956 -129.66167544 -125.92542925]
"((1, 0), (4, 1))",[-130.1187017  -129.64671828 -125.03106674 -126.78010903]
"((0, 0), (4, 1))",[-118.61302564 -135.20196076 -118.61302564 -126.11471854]
"((5, 1), (4, 1))",[-132.59245772 -136.71512257 -148.01386501 -192.6       ]
"((4, 1), (3, 1))",[-119.89449576 -122.48622263 -119.79036133 -125.72664184]
"((3, 1), (2, 1))",[-106.26275231 -108.80935135 -106.1239798  -107.9952374 ]
"((2, 1), (1, 1))",[-98.11885856 -91.70034787 -92.38579974 -93.7125571 ]
"((3, 1), (1, 1))",[-98.16654779 -83.34101061 -95.03922001 -97.60547277]
"((2, 0), (1, 1))",[-77.97468723 -82.003479   -89.7591193  -96.59826946]
"((1, 0), (1, 1))",[-70.08693501 -72.33412857 -70.71561161 -63.56141083]
"((0, 0), (1, 1))",[-60.37874605 -57.58656538 -60.37874605 -66.94327489]
"((3, 0), (1, 1))",[-87.82196996 -83.41263211 -95.03906749 -84.9288353 ]
"((2, 2), (1, 1))",[-81.02846393 -92.62185297 -82.93449579 -94.34902107]
"((1, 2), (1, 1))",[-72.4625572  -76.64847521 -84.4791711  -78.67373531]
"((0, 2), (1, 1))",[-75.13533733 -77.56587674 -71.00712923 -77.35373815]
"((3, 2), (1, 1))",[-85.35579616 -85.34673067 -93.60981132 -84.15784114]
"((4, 1), (1, 1))",[-93.50285031 -83.99012371 -85.52616292 -80.35935224]
"((4, 0), (1, 1))",[-94.05553544 -83.70539258 -83.97478878 -82.72127681]
"((4, 2), (1, 1))",[ -80.36153481 -198.          -88.13668482 -198.        ]
"((4, 1), (4, 2))",[-126.15515473 -130.79612093 -127.67327725 -189.9       ]
"((3, 1), (4, 2))",[-129.09344416 -124.56861071 -128.93398425 -115.48149075]
"((2, 1), (4, 2))",[-121.0254266  -121.81708285 -124.90989985 -118.31781842]
"((1, 1), (4, 2))",[-123.83969827 -122.40424548 -111.39791151 -113.58388397]
"((0, 1), (4, 2))",[-111.63578884 -120.16992112 -118.38555521 -113.14281503]
```

Figure 4. Final Q-table example (partial)

Figure 4 shows a portion of the final Q-table. Due to the large size and length of the q-table, it will be saved as a .csv file in the same directory as the code files.

Improvements

We've briefly discussed the results and performances on last chapter based on the Rewards vs. Episode chart. There are some problems that can be observed from the chart. 1) the agent needs to take around 750 episodes find out the best solution. 2) agent will still be making mistakes or detours frequently even after the 750 episodes training.

To solve the first problem, one method can be considered is to increase the alpha value which is also known as the learning rate, higher learning rate means the learning can occur more quickly. By increasing alpha to 0.9, it can take around 450 episodes to find the shortest way. (Shown as figure 5 below)

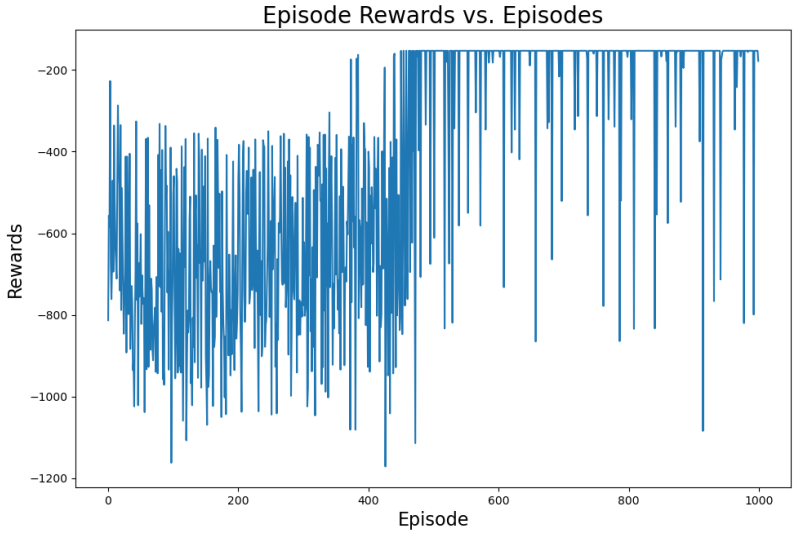

Figure 5. Episode Rewards vs. Episode (alpha = 0.9)

To solve the second problem, we can try to decay the epsilon after certain episodes to make the agent become greedier while the episode increases. The epsilon will be set to 0.01, 0.005, 0.001, 0.0005, 0.0001 at the beginning, and after 200th episode, 400th episode, 600 episode and 800th episode. Based on the Rewards vs. Episode chart we can observe that agent will barely make mistakes or detours after 500 episodes. (Shown as figure 5, combine with the alpha = 0.9 change)

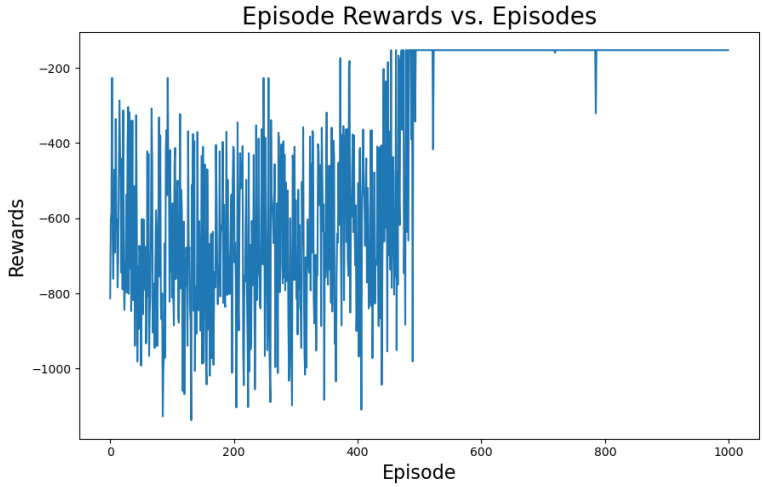

Figure 5. Episode Rewards vs. Episode (Epsilon decaying & alpha = 0.9)

Appendices

The code snippets for plotting Rewards vs. Episodes, Q-table and improvement (Q_learning_improvement.py).

```
class QAgent(object):
    def __init__(self):
        self.action_space = [1,2,3,4]
        self.Q = defaultdict(lambda: np.zeros(len(self.action_space)))
        self.discount_factor=0.99       # ori = 0.99
        self.alpha=0.9      # ori = 0.5
        self.epsilon=0.01   # ori = 0.01

    def take_action(self, state):
        if epsilon_decay1 == True:
            self.epsilon = 0.005
        if epsilon_decay2 == True:
            self.epsilon = 0.001
        if epsilon_decay3 == True:
            self.epsilon = 0.0005
        if epsilon_decay4 == True:
            self.epsilon = 0.00001
        if random.random() < self.epsilon:
            action = random.choice(self.action_space)
        else:
            action = self.action_space[np.argmax(self.Q[state])]
        return action

    def train(self, state, action, next_state, reward):
        q_predict = self.Q[state][action-1]
        q_target = reward + self.discount_factor * max(self.Q[next_state])
        self.Q[state][action - 1] += self.alpha * (q_target - q_predict)

if __name__ == '__main__':
    env = CliffBoxPushingBase()
    agent = QAgent()
    teminated = False
    rewards = []
    time_step = 0
    num_iterations = 1000
    all_episodes = []
    all_rewards = []
    epsilon_decay1 = False
    epsilon_decay2 = False
    epsilon_decay3 = False
    epsilon_decay4 = False
    for i in range(num_iterations):
        if i > 200:
            epsilon_decay1 = True
        if i > 400:
            epsilon_decay2 = True
        if i > 600:
            epsilon_decay3 = True
        if i > 800:
            epsilon_decay4 = True
        env.reset()
        while not teminated:
            state = env.get_state()
            action = agent.take_action(state)
            reward, teminated, _ = env.step([action])
            next_state = env.get_state()
            rewards.append(reward)
            print(f'step: {time_step}, actions: {action}, reward: {reward}')
            time_step += 1
            agent.train(state, action, next_state, reward)

        print(f'episode: {i}, rewards: {sum(rewards)}')
        print(f'print the historical actions: {env.episode_actions}')
        all_episodes.append(i)
        all_rewards.append(sum(rewards))
        teminated = False
        rewards = []

    print("all_episode: ", all_episodes)
    print("all_rewards: ", all_rewards)
    print('\r\nQ-Table:\n', agent.Q)

    df_col = ['(Agent), (Box)', '[up, down, left, right]']
    df = pd.DataFrame(agent.Q.items(), columns = df_col).to_csv("final_q_table.csv", index=None)

    plt.figure()
    plt.plot(all_episodes, all_rewards)
    plt.xlabel('Episode', fontdict={'size' : 16})
    plt.ylabel('Rewards', fontdict={'size' : 16})
    plt.title('Episode Rewards vs. Episodes', fontdict={'size' : 20})
    plt.show()
```