NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

AI6101 INTRODUCTION TO AI & AI ETHICS

INDIVIDUAL PROJECT 1

School of Computer Science and Engineering

Programme: MSAI

Date: 24 March 2024

Authored By: Tan Jie Heng Alfred (G2304193L)

Contents

1. Introduction

In this project, we attempt to train an agent to solve the cliff-box-pushing problem, using the reinforcement learning (RL) algorithm, Q-learning.

We have an agent (A) and a box (B) existing in a 2D grid world. The size of the 2D grid world is 6×14. Our goal is to train the agent to push the box to a pre-defined position, i.e., goal (G). However, in this environment, there are cliffs (x) which the agent and box should not fall into. Figure 1 provides an illustration of this environment. The initial position of A and B are at the points $(5,0)$ and $(4,1)$ respectively.

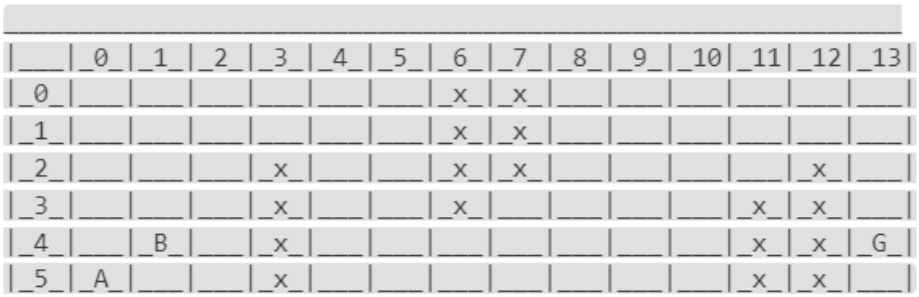

Figure 1: Grid world with agent, A, box, B, cliffs, x, and goal, G.

At every point, point on the grid world, the agent A can either move up, down, left, or right, provided the action does not cause it to move outside of the 6×14 grid. Consequently, the box can be moved by the agent in the same direction as its action, provided it does not fall outside of the grid as well. The game ends when either the box or agent drops into a cliff, or if the box reaches the goal. To train the agent using RL, we have the following reward function:

1. Reward of -1 at ever timestep. This penalizes unnecessary moves.
2. Reward of $-d_{bg}$, where d_{bg} is the distance between the box and goal.
3. Reward of $-d_{ab}$, where d_{ab} is the distance between the agent and the box.
4. Reward of -1000 if the agent of the box falls into the cliff. This heavily penalizes the act of falling into the cliff.
5. Reward of 1000 if the box reaches the goal.

Notice that reward is a function of the agent and box positions. Therefore, the actions chosen by the agent, which aims to maximize the cumulative reward, is dependent on both the agent and box positions. Therefore, the state space of our Markov decision process (MDP) is made up of all possible agent and box positions. The rules and set up of the grid world is handled by the object class `CliffBoxGridWorld`, while our agent is defined by the object class `RLAgent`. With these, we will train our agent using the RL algorithm, Q-learning.

2. Training of Agent

Q-learning is an off-policy algorithm that continuously updates the Q-value of the agent. In an MDP, the Q-value of the agent is defined to be the expected reward starting at state s and taking the action a, and thereafter following the policy π,

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_{\pi}[\textstyle\sum_{i\geq 0} \gamma^i R_{t+i} \,|S_t = s, A_t = a], \end{aligned}$$

where R_{t+i} is the reward i time steps into the future, and $0 < \gamma < 1$ is the discount factor.

In Q-learning, we update the Q-value of some state-action pair at each time step. To do this, we first initialize $Q(s,a)$ for all $s \in S$ and $a \in A$ arbitrarily (in our implementation, we have $Q(s,a) = 0$, for all s and a). Then, given an initial state s_0, we could choose $a_0 \in A$ by using a policy, say the ϵ-greed policy, derived from Q. The ϵ-greedy policy, π_ϵ, is defined, for some small $\epsilon > 0$, as,

$$\pi_\epsilon(s) = \begin{cases} \arg\max_a Q(s,a), & \text{with probability } 1-\epsilon \\ \text{random action from } A, & \text{with probability } \epsilon \end{cases}.$$

The randomness in π_ϵ affords some degree of exploration but is largely greedy in the sense that it seeks the action that maximizes $Q(s,a)$. After sampling $a_0 \sim \pi_\epsilon(\cdot\,|s_0)$, we update $Q(s_0,a_0)$ as follows,

$$Q(s_0,a_0) \leftarrow Q(s_0,a_0) + \alpha\left[R_0 + \gamma \max_a Q(s_1,a) - Q(s_0,a_0)\right],$$

where $\alpha \in (0,1]$ is the learning rate, R_0 is the reward for taking action a_0 at state s_0, and s_1 is the observed state after taking action a_0. We will run this update rule for the entirety of the episode. That is, for each step t in the episode, having observed current state s_t, we sample $a_t \sim \pi_\epsilon(\cdot\,|s_t)$ and update $Q(s_t,a_t)$ as,

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[R_t + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right].$$

Note that the $\max_a Q(s_{t+1},a)$ term finds an action that does not necessarily follow the policy, but maximizes the value at the state s_{t+1}. This is the reason Q-learning is an off-policy algorithm. After sufficiently many episode, our policy will be able to estimate the optimal policy π^*.

In our implementation, we defined the `learn` and `_max_Q_value` methods within the `RLAgent` class (see Appendix (a) and (b)). The former implements the Q-value updates, while the latter exhaustively searches all $a \in A$ that maximizes $Q(s_{t+1},a)$, hence dealing with the $\max_a Q(s_{t+1},a)$ term. Here, our action space $A = \{1,2,3,4\}$, corresponding to the actions 'up', 'down', 'left', and 'right'.

For each episode of learning, we ran 100 timesteps, with $\epsilon = 0.1$ for π_ϵ, $\alpha = 0.1$ for the learning rate, and $\gamma = 0.99$ for the discount factor of the rewards. Finally, we ran for 20,000 episodes and stored the cumulative reward of each episode in an array `rewards`. With this, we plotted a smoother curve (Figure 2) resembling the trend of the cumulative rewards as each episode passes (i.e., learning progress).

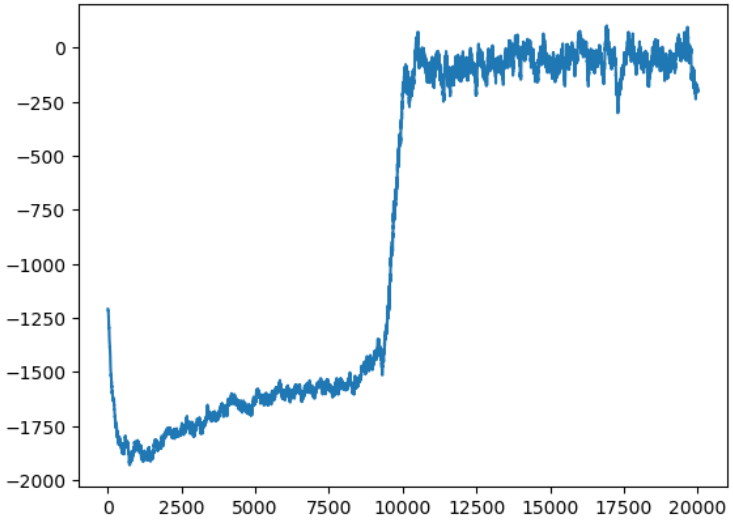

Figure 2: Learning progress as episodes increase.

The figure suggests that, as we ran more episodes, the cumulative rewards in each episode gradually increase, until episode ≈ 10,000, where the cumulative rewards in each episode markedly increased. This suggests that the agent manages to learn the right policy, such that it maximizes the cumulative rewards. Notably, because the only positive reward signal is from reaching the goal, a cumulative reward close to 0 suggests that the agent had most likely reached the goal in that episode.

3. Visualization of V-table and Policy, and Other Observations

In this section, we will visualize the V-value in a table, as well as the learned policy of the agent. The V-value of a state s is defined to be the expected sum of discounted rewards at state s,

$$V^{\pi}(s) = \mathbb{E}_{\pi}(G_t|S_t = s)$$
$$= \sum_{a \in A} Q^{\pi}(s,a)\pi(a|s)$$

Intuitively, if $V^{\pi}(s) > V^{\pi}(s')$ then it is more desirable to be in state s than in s', since the reward signal at s is higher. However, our state space is a 4-tuple, where the first two indices describe the agent position, and the last two indices describe the box position. Hence, to visualize the V-value in a table, we will use the Q-values, averaged across all the possible box positions, for each agent position. The implementation can be seen in Appendix (c). Hence, we produce a V-table, which have values superimposed over the grid world, as a proxy of the V-value of each agent position (see Figure 3).

```
v_table
[-35.1, -35.0, -34.85, -34.39, -31.41, -64.49, 0.0, 0.0, -64.34, -25.87, -23.7, -21.28, -15.61, -8.81]
[-27.21, -25.83, -26.47, -64.09, -30.85, -63.5, 0.0, 0.0, -58.12, -18.62, -17.45, -12.2, -46.07, -10.47]
[-30.53, -26.7, -62.13, 0.0, -68.75, -67.19, 0.0, 0.0, -62.68, -18.46, -17.7, -71.3, 0.0, -46.4]
[-33.38, -28.07, -58.71, 0.0, -66.1, -66.44, 0.0, -88.2, -22.53, -19.56, -59.3, 0.0, 0.0, -48.29]
[-32.67, -29.7, -63.27, 0.0, -64.02, -27.82, -68.34, -23.67, -20.43, -21.87, -55.35, 0.0, 0.0, -45.64]
[-31.87, -28.94, -65.53, 0.0, -66.39, -32.3, -33.76, -28.71, -24.42, -19.67, -56.33, 0.0, 0.0, -44.55]
```

Figure 3: V-table, where the $[i,j]$-value corresponds to the average Q-value, across all possible box positions, of the agent at position (i,j) in the grid world.

Notice that most of the values in the V-table are negative, while some of them are 0. The positions with values of 0 actually correspond to the positions with the cliffs. This is because the game terminates when the agent reaches those positions. Hence, it is not possible for it to choose an action at those positions to update the Q-values. Since we initialized the Q-values to be 0 for all states, the Q-values of these positions are not updated and remain at 0. Notice also that the values near the cliffs are more negative the rest. This means that the average Q-value for the agent to be in those positions are relatively lower than the other positions. This is reasonable, as it is more likely to obtain a huge penalty (i.e., falling into the cliff) when the agent is right beside the cliff than further away. Along the same vein, notice that the values in the last column and right next to a cliff (e.g., position [2,13]) are appreciably higher than the other values right next to a cliff (e.g., [5,3]). This is a result of being much closer to the goal, which would output the highest reward. In general, if we disregard the positions of the cliffs, we can interpret the V-table as, positions with a higher value are more desirable for the agent to be in (assuming no additional information about the box positions).

Next, we want to visualize the policy, which is also a function of the current state s, hence a function of both the agent and box positions. For us to better visualize what the agent's action will be, we will be visualizing the policy table for each possible box position (see Figure 4. Remaining policy tables can be viewed in the codes). The $[i,j]$-entry in a policy table corresponds to the action $a \in A$ when the agent is at position (i,j), for the given box position. The implementation is detailed in Appendix (c).

```
Box_pos (2, 10):
[1 4 1 1 1 1 1 1 1 4 1 1 4 1]
[2 4 3 3 1 1 1 1 1 2 1 2 1 1]
[3 1 2 1 1 2 1 1 4 4 1 3 1 2]
[2 2 3 1 2 1 1 2 4 4 1 1 1 2]
[1 1 2 1 1 1 3 3 2 4 1 1 1 4]
[1 1 1 1 2 3 1 2 2 2 1 1 1 2]
Box_pos (2, 11):
[4 1 2 3 4 3 1 1 4 1 1 2 2 3]
[1 1 3 4 3 1 1 1 1 3 2 3 3 4]
[1 1 2 1 2 1 1 1 2 4 2 1 1 4]
[1 1 1 1 4 2 1 4 3 2 3 1 1 4]
[1 1 1 1 2 1 2 4 2 4 2 1 1 2]
[1 1 1 1 2 2 3 3 3 4 3 1 1 4]
```

Figure 4: Policy table, where the value $[i,j]$-entry in a table corresponds to the action taken by the agent when it is at position (i,j), with box positions $(2,10)$ and $(2,11)$, respectively.

We noticed that many of the tables are just filled with actions 1. For instance, Figure 5 shows the policy table for box positions $(0,0), (0,1), \text{and } (0,2)$.

```
Box_pos (0, 0):
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
Box_pos (0, 1):
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
Box_pos (0, 2):
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1 1 1 1]
```

Figure 5: Policy table for box positions $(0,0), (0,1), \text{and } (0,2)$.

Since the ϵ-greedy policy chooses the action $a' = \arg\max_a Q(s,a)$ for a given state $s = (\text{agent\_pos}, \text{box\_pos})$ most of the time, the implementation using `numpy.argmax` is such that it chooses the first index if all the $Q(s,a)$ are identical, which consistently chooses the action $= 1$. Upon further investigation, we realized that many of the $Q(s,a)$ values in `agent.q_table[state]` are 0, for all $a \in A$ and fixed $s \in S$. This suggests that, across all the runs and episodes, the system never reaches these states, which is reasonable. For example, it is unlikely that the box would be pushed to $(0,0)$, given that it starts at $(1,4)$ and that the agent starts at $(5,0)$. With sufficient runs, these states would be unreachable, given that we use an ϵ-greedy policy, and a small $\epsilon = 0.1$, which does not encourage much exploration and is largely greedy. In fact, we found that 4,830 of the 7,056 possible states were not reached. This may be good if our agent manages to find the optimal path, however, if it is stuck at a local maximum and unable to reach the goal state, then we may have to explore these different states for a potential solution.

With our updated Q-values and the policy, we then let the agent act according to it and it managed to push the box to G by taking the following actions: `[4, 1, 1, 1, 3, 1, 4, 4, 4, 4, 1, 4, 2, 2, 2, 3, 2, 4, 4, 4, 4, 4, 2, 4, 1, 1, 1, 3, 1, 4, 4, 4, 1, 4, 2, 2, 2]`.

Before we conclude, we tried to also initialize the Q-values with values such as -100 and -1000, holding other hyperparameters constant. We realized that the former still converged well and the policy allowed the agent and box to reach the goal. However, the latter failed to converge well, and the policy did not allow the actions to terminate. Although Q-learning can theoretically converge, one of the criteria is that each of the state-action pair (s,a) can be reached infinitely many times. However, again, our ϵ-greedy policy does not allow for such exploration, resulting in only a fraction of the states (hence an even smaller set of state-action pairs) to be reached.

4. Conclusion

In this project, we attempted to train an agent to navigate in a grid world to push a box to a goal state, while not falling into cliffs. We trained our agent using Q-learning, which updates the Q-value of the current state-action pair, with the action determined by the ϵ-greedy policy, at each time step. With an initialization of $Q(s,a) = 0$, for all $s \in S$ and $a \in A$, we managed to find that the learning progress converges well, and the agent managed to push the box to the goal state following the policy and the updated Q-values.

Appendix

(a) Implementation of `RLAgent.learn()`.

```
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

            action = self.act(state)           #Action from policy
            state_new, reward, done, info = self.env.step(action)    #Observed state and reward
            Q_current = self.q_table[state][action]
            Q_new = self._max_Q_value(state_new)
            Q_current = Q_current + self.alpha*(reward + (self.gamma * Q_new) - Q_current) #Update

            self.q_table[state][action] = Q_current #Update table
            cumulative_reward += reward
            state = state_new


        rewards.append(cumulative_reward)

        #print(f"Cumulative reward for this episode: {cumulative_reward}")

    return rewards
```

(b) Implementation of `RLAgent._max_Q_value()`.

```
def _max_Q_value(self, state):
  q_values = self.q_table.get(state, {action for action in self.action_space})
  return max(q_values.values())
```

(c) Implementation to obtain V-table and policy table.

```
def visualize(q_table):
    v_table = {k: [] for k in range(6)}
    policy = {(k, j): [] for k in range(6) for j in range(14)}
    actions = [1, 2, 3, 4]

    for key, value in q_table.items():
      v_table[key[0]].append(np.mean(list(value.values())))
      policy[(key[2],key[3])].append(actions[np.argmax([value[i] for i in range(1, 5)])])

    for key, value in v_table.items():
      v_value = np.average(np.reshape(value, (-1, 6*14)), axis=1)
      v_table[key] = v_value

    for key, value in policy.items():
      policy[key]= np.reshape(value, (6,14))

    return v_table, policy
```