National Taiwan University
Department of Electrical Engineering
Name: Chen-Che Huang
ID: r10946005

112-1 (Fall 2023)
Reinforcement Learning

Assignment 3 Report

Q1. Discuss how you speed up your grid world environment. (10 pts.)

1. In the method *step*(), the conditions that determine all terminal states are put in the beginning so that it can avoid some work if the current state is a terminal state.

2. Comment out the code that calls the method *render_maze*() in the methods *close_door*(), *open_door*(), *bite*(), and *place_bite*().

3. The private method *_get_next_state*() is called once when the method *step*() is called. Improving this private method makes the environment faster a lot. There are two keys that can improve the efficiency:

   - Avoid Numpy array
     In the original implementation provided in advance, a numpy array is initialized from a tuple *state_coord* and then transform it according to the action, as shown below:

     ```
     next_state_coord = np.array(state_coord)
     if action == 0:
         next_state_coord[0] -= 1
     elif action == 1:
         next_state_coord[0] += 1
     elif action == 2:
         next_state_coord[1] -= 1
     elif action == 3:
         next_state_coord[1] += 1
     ```

     Numpy array is not necessary in the subsequent code, and hence we can just use tuple instead to avoid the initialization, as shown in the code below:

     ```
     if action == 0:
         next_state_coord = state_coord[0] - 1, state_coord[1]
     elif action == 1:
         next_state_coord = state_coord[0] + 1, state_coord[1]
     elif action == 2:
         next_state_coord = state_coord[0], state_coord[1] - 1
     elif action == 3:
         next_state_coord = state_coord[0], state_coord[1] + 1
     ```

   - Remove redundant conditions
     To get the next state, it is necessary to check the next state is valid or change the next state if the current state is a portal state and the action makes the agent hit wall. The original conditions are shown below:

     ```
     if not self._is_valid_state(next_state_coord) and self._is_portal_state(
         state_coord):
         next_state_coord = self.portal_next_state[state_coord]
     if not self._is_valid_state(next_state_coord):
         next_state_coord = state_coord
     ```

     We can see that there are some redundancy in these two conditions. It can be improved by change it into a nested condition like below:

     ```
     if not self._is_valid_state(next_state_coord):
         next_state_coord = state_coord
         if self._is_portal_state(state_coord):
             next_state_coord = self.portal_next_state[state_coord]
     ```

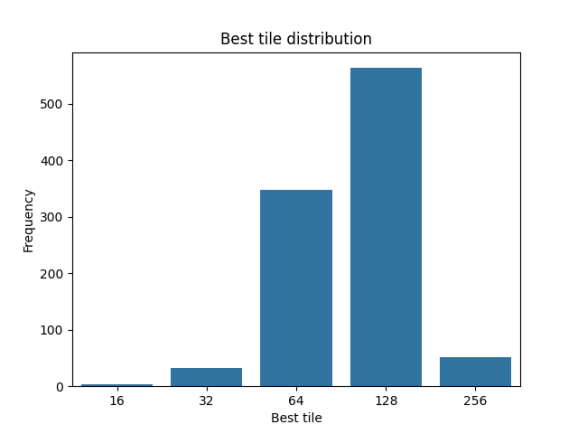
The table below shows the time costs of each task in my MacBook Pro 2022 under the several improvements mentioned above:

Improvements			Time Cost					
No rendering	Avoid Numpy array	Remove redundant conditions	Lava	Exit	Bait	Door	Portal	Maze
			2.4232	2.2226	2.4656	3.7447	2.2325	2.2359
✔			2.2281	2.2081	**2.2948**	**2.3881**	2.1997	2.1996
✔	✔		**1.9979**	**1.9254**	**1.9216**	**2.0912**	**1.9102**	**2.0047**
✔	✔	✔	**1.7803**	**1.7918**	**1.6704**	**1.7983**	**1.6988**	**1.7079**

Specifically, the time required for the task *Door* has been halved.

Q2. What's your best result in 2048? (5 pts.)

My best score is 256, as shown in the figure below:

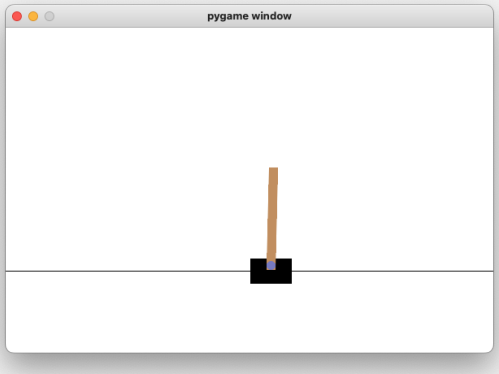


Q3. Describe what you have done to train your best model. (15 pts.)

- The algorithm I chose is PPO, which is the fastest in all available algorithms.
- Set epochs to 400 so the model can converge sufficiently.
- The illegal move reward is set to -256, as it is forced to terminate the game simultaneously if the current move is illegal. A severe punishment can help an agent learn how to avoid making an invalid moves.

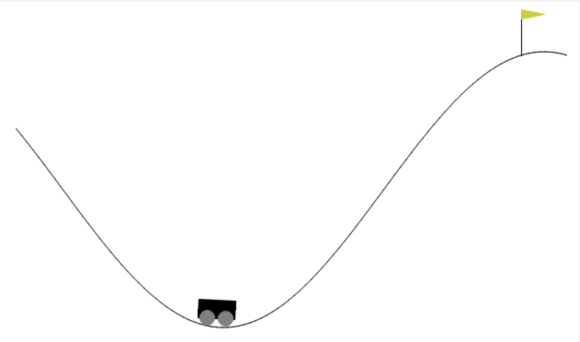
Q4. Choose an environment from the Gymnasium library and train an agent. Show your results or share anything you like. (10 pts.)

CartPole is chosen as my toy environment, and the algorithm I selected is A2C. The reward of $+1$ is given for every step taken, so this environment encourages an agent to keep the pole upright as long as possible. The figure below shows the result.



The following figure shows the average reward during training, and it indicates that the agent really learnt how to hold the pole as long as possible.

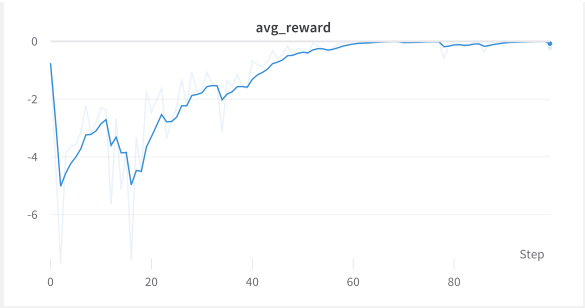
I also tried to train A2C in the environment Mountain Car Continuous and I found that the agent learnt to stay in the valley forever and never tried to reach the goal state on top of the right hill, as shown in the figure below:



It is because that the reward function is defined as

$$r = -0.1 \cdot a^2$$

where a is the force applied on the car. Only when the car reaches the goal, a positive reward $r = 100$ is added. As it is very difficult to apply a sequence of correct force to push the car to the goal, the agent can not learn that there is a large reward when it reaches the goal. So the agent learns to "lie down" in the valley... The figure below shows that the average reward can not breakthrough 0.0.


avg_reward
0
-2
-4
-6
Step
0
20
40
60
80