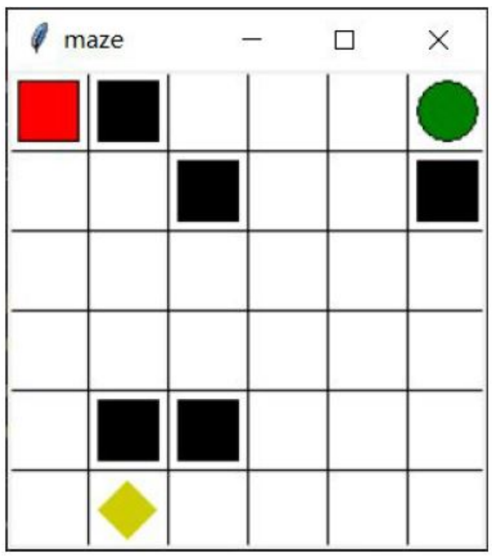
I. INTRODUCTION

A. Purpose

The goal of this homework is to implement a Dyna-Q learning agent to search for the treasure and exit in a grid-shaped maze. The agent will learn by trail and error from interactions with the environment and finally acquire a policy to get as high as possible scores in the game.

Suppose a 6×6 grid-shaped maze in Figure 1. The red rectangle represents the start point and the green circle represents the exit point. You can move upward, downward, leftward and rightward and you should avoid falling into the traps, which are represented by the black rectangles. Finding the exit will give a reward +1 and falling into traps will cause a reward -1, and both of the two cases will terminate current iteration.You will get a bonus reward +3 if you find the treasure, which shown as golden diamond.



(a) The board of Game

B. Environment

There is a minimal amount of equipment to be used in this homework. A few requirements are listed below:

- Python 3.7
- Library: matplotlib,numpy,pandas
- Complier: Vscode

C. Procedure

The key theories of the algorithm are consist of three parts as following:

1. Dyna-Q learning
2. Epsilon Greedy

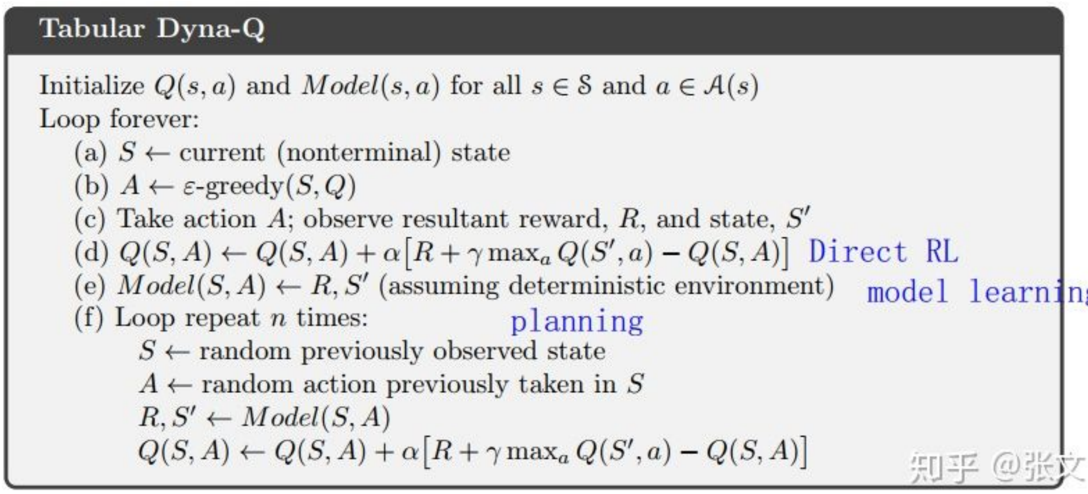
II. IMPLEMENTATION

A. Dyna-Q learning

Compared with normal Q learning,Dyna-Q learning in each exploring step that it uses an experience model to generate sub-optimal policies, which speeds up the learning process.

Dyna-Q Learning is composed of model-free Q-Learning and model-based Dyna-Q Learning. Model-free part updates Q-value simply based on its expensive experience, while model-based part updates Q-value when backpropagating the experienced fragment of episode.

And below is Dyna-Q algorithm's pseudo-code.


Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy(S,Q)
(c) Take action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$ Direct RL
(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment) model learning
(f) Loop repeat n times: planning
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

(b) pseudo-code of Dyna-Q

In this algorithm, I use the normal Dyna-Q algorithm framework, store the data in the tuple, and constantly refresh the state to update the Q value to achieve better score.

B. Epsilon Greedy

The epsilon-greedy algorithm, as a strategy for decision-making, is widely used in many fields of machine learning. For example, when epsilon = 0.6, it means that 60% of the time, the behavior will be selected according to the optimal value of the Q table, and the behavior will be randomly selected 40% of the time.

Using epsilon-greedy algorithm can prevent Dyna-Q algorithm from falling into local optimum to a certain extent, and can help the program to make early decisions.

The most important thing for the epsilon-greedy algorithm is the epsilon attenuation strategy. After trying a variety of different methods and strategies, I finally chose the e-exponential decay strategy, which decreases rapidly at the beginning and slowly decreases afterwards to obtain better results.

C. Parameter Adjustment

In the field of Artificial Intelligence, the selection of suitable parameters can play a great role in a certain sense. In the algorithm used in this experiment, the main adjustable parameters are learning rate alpha, discount factor gamma, and random coefficient episilon.

1. learning rate alpha

   The learning rate affects how much the code adjusts the weights of the network and adjusts the loss gradient. The lower the learning rate, the slower the downward slope. A lower learning rate can ensure that no local minimum is missed, but it also means that it will take a long time to converge.

2. discount factor gamma

   The value of the discount factor determines how we view future rewards. A gamma close to 0 means that future rewards are not valued, and a gamma close to 1 means that future rewards are valued and a long-term vision.

3. episilon

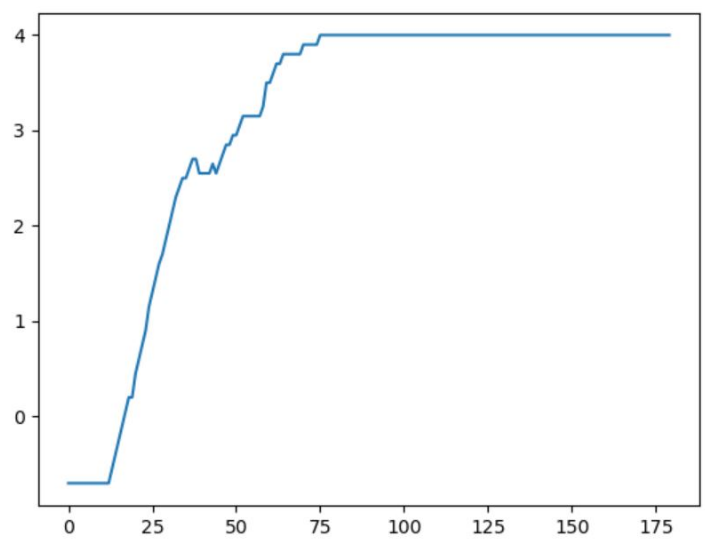
   epsilon and its attenuation strategy determine how long the code can find the optimal solution and the ability to jump out of the local optimal solution.

   Finally I choose the e exponential decay strategy:

$$\varepsilon = \frac{\varepsilon_0}{e^{\frac{n}{35}}}$$

D. Result

The final experimental results are shown in the figure below, the ordinate is the average of ten consecutive scores.



(c) result score

In the figure, we can see that the score basically continued to rise before 75 times, and converged to full marks after 75 times.

III. CONCLUSION

I designed a agent which's named as My_Agent to participate this tournament. From the test in our PC, I found my agent could complete the mission around 75 times.

Now is the ending time, first I want to thank for the help from teaching assistants and our professor Yue Gao. Through this homework, we have attained lots of skills and experiences. We have almost mastered the Dyna-Q learning. We gain a lot.

I have added an adjunct of source codes with this test report, Thanks again!

IV. EXPERIMENT CODE

This section contains my Agent code block of this homework.

My_Agent:

```
import numpy as np
import pandas as pd
import random
import math

class My_Agent:
    def __init__(self, actions):
        self.Qalue_table = {}
        self.actions = actions
    def epsilon_greedy(self, state,epsilon,episode):
        for action in self.actions:
            if (tuple(state), action) not in self.Qalue_table:
                self.Qalue_table[(tuple(state), action)] = 0

        epsilon /= math.exp(episode/35)   #change with the episode

        if  random.random()<epsilon:   #case random occur
            action = np.random.choice(self.actions)
        else:
            # get the action corresponding to max q
            maxval_list=[]
            max_val=-float('inf')
            for action in self.actions:
                val=self.Qalue_table[(tuple(state), action)]
                if(max_val<val):
                    max_val=val
            for action in self.actions:
                if(self.Qalue_table[(tuple(state), action)]==max_val):
                    maxval_list.append(action)
            action = np.random.choice(maxval_list)
        return action

    def Qvalue_refresh(self, new_action, state, new_state, reward,gamma,alpha):
        max_tmp = -float('inf')
        for action in self.actions:
            step_Qvalue = (tuple(new_state), action)
            if step_Qvalue not in self.Qalue_table:
                self.Qalue_table[step_Qvalue] = 0
            if max_tmp < self.Qalue_table[step_Qvalue]:
                max_tmp = self.Qalue_table[step_Qvalue]

        self.Qalue_table[(tuple(state), new_action)] = (1 - alpha) * self.Qalue_table
            [(tuple(state), new_action)] + alpha * ( reward + gamma * max_tmp)
```

Main:

```
from maze_env import Maze
from RL_brain import My_Agent
import random
from time import sleep
import numpy as np
import matplotlib.pyplot as plt
```

```
if __name__ == "__main__":
    ### START CODE HERE ###
    env = Maze()
    agent = My_Agent(list(range(env.n_actions)))

    episilon=0.6
    gamma=0.85
    alpha=0.3
    training_count=200

    training_reward=np.arange(0,training_count)
    reward_plot=np.zeros(training_count)
    for episode in range(training_count):
        state = env.reset()
        episode_reward = 0
        while True:
            #env.render()
            action = agent.epsilon_greedy(state,episilon,episode)
            new_state, reward, done = env.step(action)
            episode_reward += reward
            agent.Qvalue_refresh(action, state, new_state, reward,gamma,alpha)
            state = new_state
            if done:
                #env.render()
                sleep(0.001)
                break

        print('episode:', episode, 'episode_reward:', episode_reward)
        training_reward[episode] = episode_reward

    for i in range(training_count):
        for j in range(0,20):
            reward_plot[i]+=training_reward[i-j]


    plt.plot(reward_plot[20:]/20)
    plt.show()
    ### END CODE HERE ###
    print('\ntraining over\n')
```

maze_env:

```
import numpy as np
np.random.seed(1)
import tkinter as tk
import time


UNIT = 40
MAZE_H = 6
MAZE_W = 6


class Maze(tk.Tk, object):
    def __init__(self):
        super(Maze, self).__init__()
```

```
        self.action_space = ['u', 'd', 'l', 'r']
        self.n_actions = len(self.action_space)
        self.title('maze')
        self.geometry('{0}x{1}'.format(MAZE_H * UNIT, MAZE_H * UNIT))
        self._build_maze()
        self.bonusFlag = False
        self.eating = True

    def _build_maze(self):
        self.canvas = tk.Canvas(self, bg='white',
                           height=MAZE_H * UNIT,
                           width=MAZE_W * UNIT)

        for c in range(0, MAZE_W * UNIT, UNIT):
            x0, y0, x1, y1 = c, 0, c, MAZE_H * UNIT
            self.canvas.create_line(x0, y0, x1, y1)
        for r in range(0, MAZE_H * UNIT, UNIT):
            x0, y0, x1, y1 = 0, r, MAZE_H * UNIT, r
            self.canvas.create_line(x0, y0, x1, y1)

        origin = np.array([20, 20])

        # hell1
        hell1_center = origin + np.array([UNIT * 1, UNIT * 0])
        self.hell1 = self.canvas.create_rectangle(
            hell1_center[0] - 15, hell1_center[1] - 15,
            hell1_center[0] + 15, hell1_center[1] + 15,
            fill='black')
        # hell2
        hell2_center = origin + np.array([UNIT * 2, UNIT * 1])
        self.hell2 = self.canvas.create_rectangle(
            hell2_center[0] - 15, hell2_center[1] - 15,
            hell2_center[0] + 15, hell2_center[1] + 15,
            fill='black')
        # hell3
        hell3_center = origin + np.array([UNIT * 5, UNIT * 1])
        self.hell3 = self.canvas.create_rectangle(
            hell3_center[0] - 15, hell3_center[1] - 15,
            hell3_center[0] + 15, hell3_center[1] + 15,
            fill='black')
        # hell4
        hell4_center = origin + np.array([UNIT * 1, UNIT * 4])
        self.hell4 = self.canvas.create_rectangle(
            hell4_center[0] - 15, hell4_center[1] - 15,
            hell4_center[0] + 15, hell4_center[1] + 15,
            fill='black')
        # hell5
        hell5_center = origin + np.array([UNIT * 2, UNIT * 4])
        self.hell5 = self.canvas.create_rectangle(
            hell5_center[0] - 15, hell5_center[1] - 15,
            hell5_center[0] + 15, hell5_center[1] + 15,
            fill='black')

        oval_center = origin + np.array([UNIT * 5, UNIT * 0])
        self.oval = self.canvas.create_oval(
            oval_center[0] - 15, oval_center[1] - 15,
            oval_center[0] + 15, oval_center[1] + 15,
            fill='green')

```

```
        self.rect = self.canvas.create_rectangle(
            origin[0] - 15, origin[1] - 15,
            origin[0] + 15, origin[1] + 15,
            fill='red')

        # create bonus
        bonus_center = origin + np.array([UNIT * 1, UNIT * 5])
        self.bonus = self.canvas.create_polygon(
            [bonus_center[0]+15, bonus_center[1],
            bonus_center[0], bonus_center[1]-15,
            bonus_center[0]-15, bonus_center[1],
            bonus_center[0], bonus_center[1]+15],
            fill='#CDCD00')
        self.bonus_location = self.canvas.create_polygon(
            bonus_center[0] - 15, bonus_center[1] - 15,
            bonus_center[0] + 15, bonus_center[1] + 15,
            fill='white')

        self.canvas.pack()

    def reset(self):
        self.canvas.delete(self.rect)
        self.canvas.delete(self.bonus)
        origin = np.array([20, 20])
        self.rect = self.canvas.create_rectangle(
            origin[0] - 15, origin[1] - 15,
            origin[0] + 15, origin[1] + 15,
            fill='red')
        bonus_center = origin + np.array([UNIT * 1, UNIT * 5])
        self.bonus = self.canvas.create_polygon(
            [bonus_center[0]+15, bonus_center[1],
            bonus_center[0], bonus_center[1]-15,
            bonus_center[0]-15, bonus_center[1],
            bonus_center[0], bonus_center[1]+15],
            fill='#CDCD00')
        self.bonusFlag = False
        self.eating = True
        return self.canvas.coords(self.rect) + [self.bonusFlag]

    def step(self, action):
        s = self.canvas.coords(self.rect)
        base_action = np.array([0, 0])
        if action == 0:   # up
            if s[1] > UNIT:
                base_action[1] -= UNIT
        elif action == 1:   # down
            if s[1] < (MAZE_H - 1) * UNIT:
                base_action[1] += UNIT
        elif action == 2:   # right
            if s[0] < (MAZE_W - 1) * UNIT:
                base_action[0] += UNIT
        elif action == 3:   # left
            if s[0] > UNIT:
                base_action[0] -= UNIT

        self.canvas.move(self.rect, base_action[0], base_action[1])  # move agent

        s_ = self.canvas.coords(self.rect)  # next state
```

```
        if s_ == self.canvas.coords(self.oval):
            reward = 1
            done = True
        elif (s_ == self.canvas.coords(self.bonus_location)) and (self.bonusFlag ==
            False):
            self.bonusFlag = True
            reward = 3
            done = False
        elif s_ in [self.canvas.coords(self.hell1), self.canvas.coords(self.hell2),
            self.canvas.coords(self.hell3), self.canvas.coords(self.hell4),
        self.canvas.coords(self.hell5)]:#, self.canvas.coords(self.hell6), self.canvas
            .coords(self.hell7)]:
            reward = -1
            done = True
        else:
            reward = 0
            done = False
        s_.append(self.bonusFlag)

        return s_, reward, done

    def render(self):
        self.update()
        if self.bonusFlag and self.eating:
            self.eating = False
            time.sleep(0.5)
            self.canvas.delete(self.bonus)
```