I. INTRODUCTION

A. Purpose

The goal of today's lab or homework3 is to render me to be familiar with Reinforcement Learning(RL). There are two environments: the Maze Environment and Atari Game. I have to implement Dyna-Q Learning and Deep-Q Learning, respectively.

B. Environment

The environment for this lab and later labs are listed as follows:

- Pycharm(Python 3.7)
- Ubuntu 16.04 LTS

II. REINFORCEMENT LEARNING IN MAZE ENVIRONMENT

In the maze part of homework3, I will implement an agent based on Dyna-Q Learning, which can find optimal policy in two mazes(a simple one and a hard one), shown 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. Moreover, you will get a bonus reward +2 if you find the treasure, which shown as golden diamond.

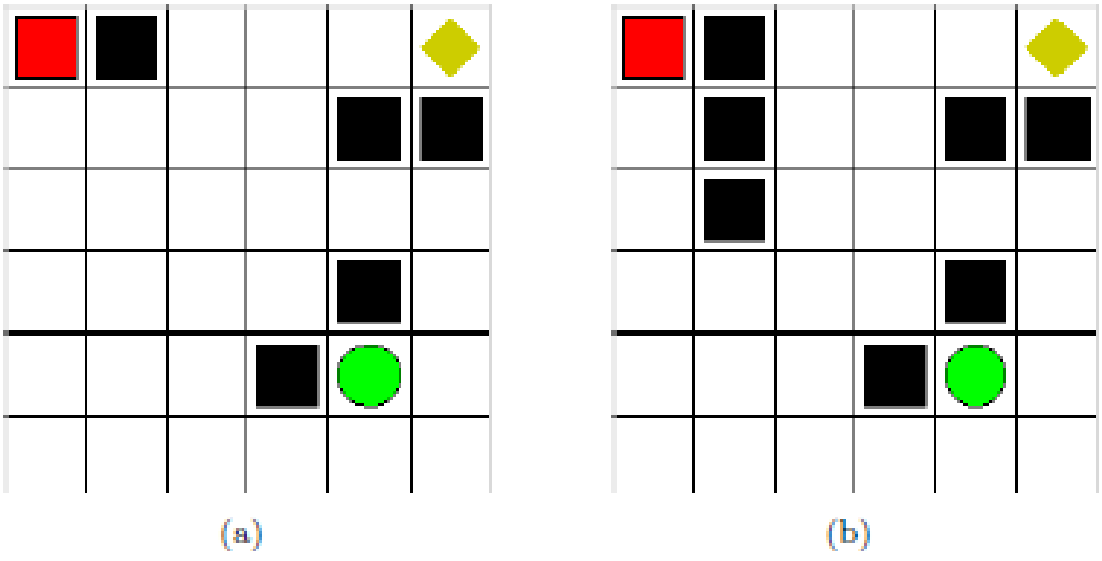

FIG. 1: Maze Environment

In general, besides fully implementing Dyna-Q Learning, I use a combination of undirected Semi-Uniform Distributed Exploration and directed Counter-Based Exploration

A. Dya-Q Learning

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.

1. Exploration Function

Undirected Semi-Uniform Distributed Exploration This is the same as epsilon-greedy strategy. $(1 - P_{best})$ equals to ϵ, where one legal action is randomly choosed, including the optimal action. Besides this, the optimal action has another P_{best} probability to be chosen. The theoretical equation is given below:

$$P(a) = \begin{cases} P_{best} + \dfrac{1 - P_{best}}{\#\text{ of actions}} & \text{, if } a \text{ maximizes } f(a) \\ \dfrac{1 - P_{best}}{\#\text{ of actions}} & \text{, otherwise} \end{cases}$$

Directed Counter-Based Exploration Here, I simplify the counter-based exploration strategy to some extent. The principle is that the location with higher expected Q-value and less visited counts should be preferred. This makes sense, as this kind of locations do have the potential to be the black horse.The exploration term is the quotient of the counter value for the current state and the expected counter value for the state that results from taking an action. In the given equation, α is constant factor weighting exploration versus exploitation. The theoretical equation is also given below:

$$eval_c(a) = \alpha \cdot f(a) + \frac{c(s)}{E[c \mid s, a]}$$
$$E[c \mid s, a] = \sum_{s} P_{s \to s'}(a) \cdot c(s')$$

B. Code Part

The code in main.py and agent.py are appended as follows:
main.py

```
from agent import Agent, Model #import class Agent from agent
import time

maze = '1'

if maze == '1':
```

```
    from maze_env1 import Maze
elif maze == '2':
    from maze_env2 import Maze


if __name__ == "__main__":
    ### START CODE HERE ###
    # This is an agent with random policy. You can learn how to interact with the
                                                environment through the code below.
    # Then you can delete it and write your own code.

    episodes             = 100
    model_based_episodes = 5
    env    = Maze()
    model = Model(actions=list(range(env.n_actions)))
    agent = Agent(actions=list(range(env.n_actions)))       #   range  (4)      0      ,1,2,
                                                              3
    for episode in range(episodes):                         #

        s = env.reset()
        episode_reward = 0
        while True:
            #env.render()                 # You can comment all render() to turn off the
                                                          graphical interface in training
                                                          process to accelerate your code.

            # move one step
            a = agent.choose_action(str(s))

            s_, r, done = env.step(a)

            # update Q model-free
            agent.update(str(s), a, r, str(s_), done)

            model.store_transition(str(s), a, r, s_)

            # update Q model-based
            for n in range(model_based_episodes):
                ss, sa  = model.sample_s_a()
                sr, ss_ = model.get_r_s_(ss, sa)
                agent.update(ss, sa, sr, str(ss_), done)

            episode_reward += r
            s = s_

            if done:
                #env.render()
                time.sleep(0.5)
                break

        # format output
        print('episode: %2s'%episode, 'episode_reward: %2s'%episode_reward)

    ### END CODE HERE ###

    print('\ntraining over\n')
```

agent.py

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

class Agent:
    ### START CODE HERE ###
```

```
#                       actionsepsilon
def __init__(self, actions, alpha=0.1, gamma=0.9, epsilon=1):
    self.actions = actions
    self.alpha   = alpha                                # learning_rate
    self.gamma   = gamma                                # reward_decay
    self.epsilon = epsilon
    self.qTable  = pd.DataFrame(columns=self.actions) # tabular q-value
    self.cTable  = pd.DataFrame(columns=self.actions) # tabular counter value

def choose_action(self, observation):
    self.state_exist(observation)

    # tune epsilon
    # sigmoid to decrease
    global cnt
    epsilon_ = 1 / (1 + math.exp(cnt - 250))
    if epsilon_ > 0.01:
        self.epsilon = epsilon_
        cnt = cnt + 1
    else:
        self.epsilon = 0.01
    #print("self.epsilon is:", self.epsilon)

    # epsilon-greedy method
    if np.random.uniform() < self.epsilon:
        # random
        action = np.random.choice(self.actions)
    else:
        # get all potential actions
        action_to_be = self.qTable.loc[observation]
        # if more than one
        action = np.random.choice(action_to_be[action_to_be == max(action_to_be)].index)

    return action

def update(self, s, a, r, s_, done):
    # weight between Q-value & counter value
    # decrease
    k = 3 / (1 + math.exp(cnt - 220))

    self.state_exist(s_)

    q_predict = self.qTable.loc[s, a]

    if not done:
        q_target = r + self.gamma * (self.qTable.loc[s_].max())
    else:
        q_target = r

    # tune alpha
    # decrease
    '''
    if 0 <= cnt < 100:
        self.alpha = 0.5
    elif 100 <= cnt <= 500:
        self.alpha = 0.2
    else:
        self.alpha = 0.1
    '''

    self.cTable.loc[s, a] += 1
    self.qTable.loc[s, a] += self.alpha * (q_target - q_predict) + k / (self.cTable.loc[
                                                        s].sum())
```

```
    def state_exist(self, state):
    # if state not exist, initialize all 0
        if state not in self.qTable.index:
            self.qTable = self.qTable.append(pd.Series(
                [0]*len(self.actions),
                index=self.qTable.columns,
                name=state))

        if state not in self.cTable.index:
            self.cTable = self.cTable.append(pd.Series(
                [0]*len(self.actions),
                index=self.cTable.columns,
                name=state))

class Model:
    '''
    Model(s,a) is utilized to implement model-based RL:
    1. Learn from interaction
    2. Update Q(s,a) after each learning process
    '''
    def __init__(self, actions):
        self.actions = actions
        self.storage = pd.DataFrame(columns=actions, dtype=np.object)

    def store_transition(self, s, a, r, s_):
        if s not in self.storage.index:
            self.storage = self.storage.append(
                pd.Series([None] * len(self.actions),
                index = self.storage.columns,
                name = s)
            )

        self.storage.set_value(s, a, (r, s_))

    def sample_s_a(self):
        s = np.random.choice(self.storage.index)
        a = np.random.choice(self.storage.loc[s].dropna().index)
        return s, a

    def get_r_s_(self, s, a):
        r, s_ = self.storage.loc[s, a]
        return r, s_

# global variables storing last iteration's information
np.random.seed(100)
cnt = 0
    ### END CODE HERE ###
```