AU 342 Principles of Artificial Intelligence

By: Shen Zhen (518021910149)
Cheng Xinwen (518021910031)

HW#: 2

November 1, 2020

I. INTRODUCTION AND IMPLEMENTATION

In this assignment, we will implement Reinforcement Learning with Dyna-Q in Maze Environment and with Deep Q Network on Atari Game.

A. Reinforcement Learning in Maze Environment

In this assignment, I will 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. This part is finished individually.

1. Game Description

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.

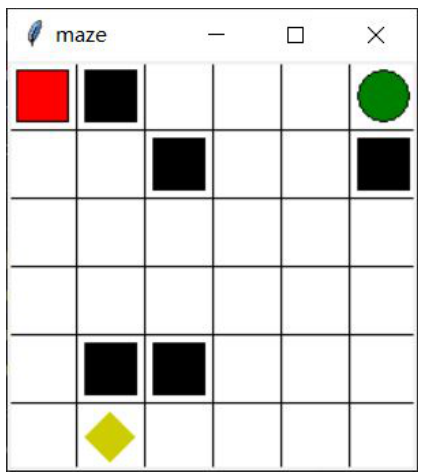


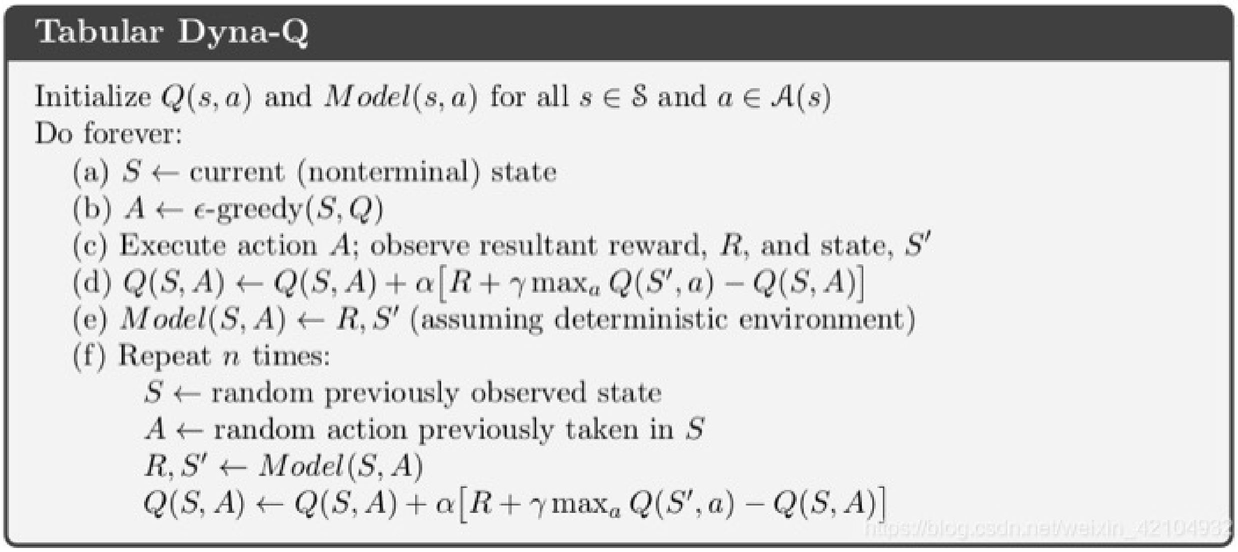
FIG. 1: Maze environment

2. State and Action Space

State(5-dimension): The current position of the agent (4D) and a bool variable (1D) that indicates whether the treasure has been found.

Action(1-dimension): A discrete variable and 0, 1, 2, 3 respectively represent move upward, downward, rightward and leftward.

I implemented an agent based on Dyna-Q learning.It is an algorithm different from Q-Learning in that it tries to build the model based on accessible data, which leads to less interaction with the maze environment and needs less data. In general, it is cheaper and saves space. Here is Dyna-Q algorithm shown with pseudocodes.



Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \epsilon$-greedy(S,Q)
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
(f) Repeat n times:
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S,A)$
$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

FIG. 2: Dyna-Q

In my code, I choose an N for Dyna-Q, which means that after N actions, the Dyna-Q algorithm will be called and update the Q value table according to existing experience. In my way to calculate the Q value, based on the traditional formula, I added a factor k related to the count of times of which the agent has been to the next state. Also, Temporal-Difference Learning is used. The new formula can be illustrate as following.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'}\left(Q\left(s',a'\right) - C\left(s',a'\right)\right) - Q(s,a)\right]$$

Here $C\left(s',a'\right)$ denotes the count of times agent has chosen state s' with action a'. When next state has been rarely been to, the agent tends to choose it. With this formula, the agent can fully explore the whole maze environment.The code is below.

```
for j in range(4):
    self.cnt_of_state[state_][j] = self.cnt_of_state[state_][j] * 0.92

self.cnt_of_state[state][a] += 1
self.update_cnt_of_state(s)
self.has_been_to_this_state[float((state[0] + state[2]) / 2), float((state[1] + state[3]) / 2)] = True
self.q_dict[state][a] += self.alpha * (
            r + self.gamma * max(((self.q_dict[state_][i]) - self.cnt_of_state[state_][i] * self.k) for i in range(4))
            -
            self.q_dict[state][a])
self.model_dict[(state, a)] = [state_, r]

N = self.Dyna_N
for n in range(N):  # iteration for N times
    ms, ma = random.choice(list(self.model_dict))
    ms_, mr = self.model_dict[(ms, ma)]
    self.q_dict[ms][ma] += self.alpha * (mr + self.gamma * max(((self.q_dict[ms_][i])
```

Considering the exploration-exploitation dilemma, I completed my Dyna-Q learning agent by implementing ϵ-greedy action selection, meaning it chooses random actions in an epsilon fraction of the time, and follows its current best Q-values otherwise.

Epsilon-Greedy policy selects a random action with probability ϵ or otherwise follows the greedy policy $a = \underset{a}{\operatorname{argmax}} Q^{\pi}(s, a)$. It can contribute to exploration of the maze environment. However in the real situation, if the probability ϵ is too big, convergence can be a tough problem. Therefore, my plan is to set a major value of ϵ at first. In order to ensure the agent access to every grid of the panel, I estimated whether every grid has been been to. When they are all arrived at, ϵ is set zero and exploitation should take effect.The code is below.

```
def epsilon_decay(self, s):
    '''
    epsilon is large at first ,then gradually decays
    :return:
    '''
    epsilon = self.epsilon
    if self.all_arrived == False:
        epsilon = 0.1
    if self.all_arrived == True:
        epsilon = 0
    return epsilon

self.epsilon = self.epsilon_decay(state)
        # use epsilon greedy
        if random.random() < self.epsilon:
            action = np.random.choice(self.actions) # a random action
        else:
            max_actions=[]
            for i in range(4):
                if self.q_dict[state][i] == max(self.q_dict[state]):
                    max_actions.append(i)
            action = random.choice(max_actions) #choose action of maximum value
```

4. Results and Analysis

In general, my agent is able to finish the game and find the treasure then leave with a total reward of 4. In most cases, my agent can get convergence in 100 steps. However in a few cases, it takes about 200 steps to be converged, I estimated and I think it has something to do with my exploration-exploitation strategy, which can be further optimized.

On the other hand, how to choose the best parameters also matters. The most important parameters in this environment is the ϵ, the discount γ, the learning rate α, and the exploration factor k. ϵ is chosen based on the principal that it should be big at first and then decay to zero. γ is the discount of the contribution of Q values in next steps, which should be a proper value to guarantee convergence. α together with k also should be a proper value. Each parameter is significant in the learning process.

Below is my average learning result, in most cases it can converge in less than 50 episodes.

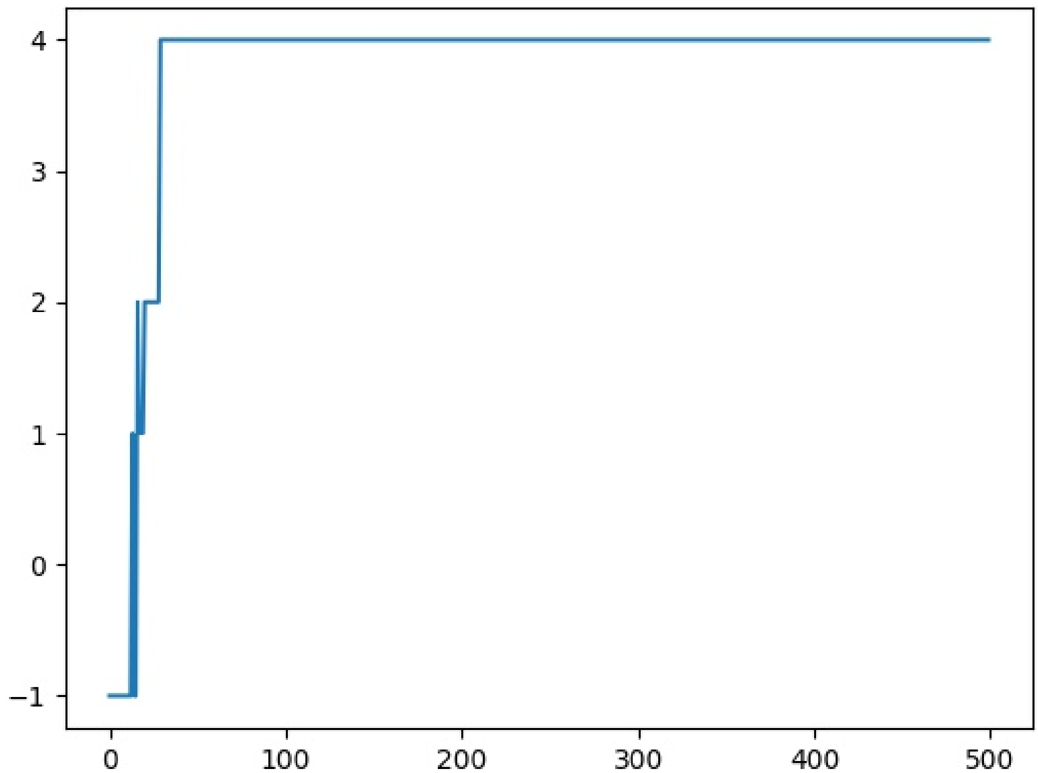


FIG. 3: Q-Learning Result

B. Reinforcement Learning on Atari Game

In this part, we will implement a DQN agent to play the atari game pacman. We first completed the given code and get a simple DQN agent. After tuning the hyper-parameters of the agent using both our experience and suggestions from skillful engineers, we still could not get a satisfying set of parameters. So we decided to improve our DQN algorithm by changing the uniform sample with prioritized experience replay and using Double DQN instead of Nature DQN. Although we still could not reach a satisfying score, in this process, we developed our unique understanding of those hyper-parameters and had a primary knowledge of how their values will influence the convergence and the outcome of learning.

1. Game Description

Pacman is one of the classic and leading games. You need to guide the pac-Man to eat all the dots and avoid the ghosts. In this assignment, you are asked to design a DQN agent to learn control policies directly from the visual information of the game.

As shown in the figure, the 'MsPacman-ram-v0' gym environment is utilized as the training environment.This environment provides the ram(128 bytes) of the atari console as model input. Each time, the agent should choose an action from 9 available actions, corresponding to the 8 buttons on the handle and "do nothing".

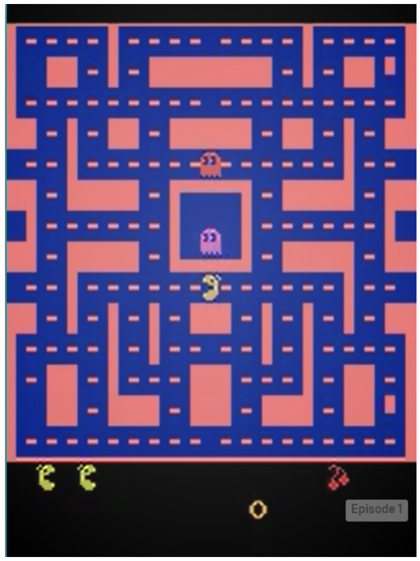


FIG. 4: Atari MsPacman

2. Codes and Explanations

There are already many functions in the given code, so the first step is to understand and use these functions and complete DQN training process. Firstly, we initialized the parameters we would use, and then tuned them according to the training and learning curve. To speed up the learning efficiency and training process, we decided to improve the DQN algorithm first and after that we began to tune the agent. So we will first introduce the primary DQN algorithm frame and next is our improving approach on prioritized experience replay and Double DQN, and finally it comes to our tuning of parameters.

```
# These are hyper parameters for the DQN
self.discount_factor = 0.9
self.learning_rate = 0.01
self.epsilon = 1.0
self.epsilon_min = 0.01
self.epsilon_decay = (self.epsilon - self.epsilon_min) / 50000
self.batch_size = 32
self.train_start = 1000
self.beta = 0.1
self.maxlen = 100000
self.memory = Memory(self.batch_size, self.maxlen, self.beta)
```

Then in each episode, we firstly initialized the score, state and lives.During one game, pacman gets an action using get-action function, which is based on DQN and we will introduce it later. After executing an action, we got next state, reward, done and info as feedback from the environment. Then we judged if the agent was dead,if not the game continues and we save the sample<s, a, r, s'> to replay. To reduce computational overhead, we decided to train the evaluation every 4 steps and updated the target network after 2500 update-times. Update-times here increases only when the agent has taken more than 250 steps under the circumstance of each live. The following is our code of main loop.

```
if __name__ == "__main__":
```

```
    # load the gym env
    env = gym.make('MsPacman-ram-v0')
    # set  random seeds to get reproduceable result(recommended)
    set_random_seed(0)
    # get size of state and action from environment
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    # create the agent
    agent = DQNAgent(state_size, action_size)
    # log the training result
    scores, episodes = [], []
    graph_episodes = []
    graph_score = []
    avg_length = 100
    sum_score = 0
    iteration = 0
    update_times = 0

    # train DQN
    for e in range(EPISODES):
        done = False
        score = 0
        state = env.reset()
        state = np.reshape(state, [1, state_size])
        lives = 3
        while not done:
            dead = False
            step = 0
            while not dead:
                # render the gym env
                if agent.render:
                    env.render()
                step += 1
                # get action for the current state
                action = agent.get_action(state, step)
                # take the action in the gym env, obtain the next state
                next_state, reward, done, info = env.step(action)
                next_state = np.reshape(next_state, [1, state_size])
                # judge if the agent dead
                dead = info['ale.lives'] < lives
                lives = info['ale.lives']
                # update score value
                score += reward
                # save the sample <s, a, r, s'> to the replay memory
                agent.append_sample(state, action, reward, next_state, done)
                # train the evaluation network
                if step % 4 == 0:
                    agent.train_model()
                # go to the next state
                state = next_state

            # update the target network after some iterations.
            if update_times >= 2500:
                update_times = 0
                agent.eval2target()
        # print info and draw the figure.
        if done:
            scores.append(score)
            sum_score += score
            episodes.append(e)
            # plot the reward each episode
            # pylab.plot(episodes, scores, 'b')
            print("episode:", e, "  score:", score, "  memory length:",
                  agent.memory._sum_tree.size, "  epsilon:", agent.epsilon,"learning_rate",agent.learning_rate)
        if e % avg_length == 0:
            graph_episodes.append(e)
            graph_score.append(sum_score / avg_length)
            sum_score = 0
            # plot the reward each avg_length episodes
            pylab.plot(graph_episodes, graph_score, 'r')
            pylab.savefig("./pacman_avg.png")
```

Next we will introduce the prioritized experience replay of the agent. To implement the prioritized experience replay, we first implementd the SumTree data structure. SumTree has four functions: adding a node, updating the priority of the node, getting total priority and sampling according to a given weight. Then we created a class called Memory to implement the prioritized experience replay. In Memory, we can store a transition experienced during the game, get mini batches using prioritized experience replay and update the td-error which is used as the priority of each transition.

In class SumTree, firstly we set the tree capacity. Then we initialized self.tree to a list of 0 and self.data to a list of None. Here we used self.tree to save the priority and self.data to save the transitions. When adding a node, we saved it into self.data and set its priority as the maximum to ensure every node can be sampled at least once. Then we increased the curr-point and self.size. In SumTree, only leaves save the transitions and according priority, other nodes' value is the sum of their children's priorities. When updating the priority of a certain leaf node, we update all the ancestors as well. The following is our code for class SumTree:

```
class SumTree:
```

```python
    def __init__(self, capacity):

        self.capacity = capacity
        self.tree = [0] * (2 * capacity - 1)
        self.data = [None] * capacity
        self.size = 0
        self.curr_point = 0


    def add(self, data):
        self.data[self.curr_point] = data
        self.update(self.curr_point, max(self.tree[self.capacity - 1:self.capacity + self.size]) + 1)

        self.curr_point += 1
        if self.curr_point >= self.capacity:
            self.curr_point = 0

        if self.size < self.capacity:
            self.size += 1


    def update(self, point, weight):
        idx = point + self.capacity - 1
        change = weight - self.tree[idx]
        self.tree[idx] = weight
        parent = (idx - 1) // 2
        while parent >= 0:
            self.tree[parent] += change
            parent = (parent - 1) // 2

    def get_total(self):
        return self.tree[0]


    def get_min(self):
        return min(self.tree[self.capacity - 1:self.capacity + self.size - 1])

    def sample(self, v):
        idx = 0
        while idx < self.capacity - 1:
            l_idx = idx * 2 + 1
            r_idx = l_idx + 1
            if self.tree[l_idx] >= v:
                idx = l_idx
            else:
                idx = r_idx
                v = v - self.tree[l_idx]

        point = idx - (self.capacity - 1)
        return point, self.data[point]
```

In class Memory we used SumTree to implement an experience pool. The main function is get-mini-batches, in which we sampled a mini-batch to train the evaluation model using randomized priority, which combines greedy priority preferential and uniform random sampling. By using this algorithm,we sped up the convergence and usually it will converge in no more than 2000 episodes.

```python
class Memory(object):
    def __init__(self, batch_size, max_size, beta):
        self.batch_size = batch_size
        self.max_size = 2 ** math.floor(math.log2(max_size))
        self.beta = beta
        self._sum_tree = SumTree(max_size)

    def store_transition(self, s, a, r, s_, done):
        self._sum_tree.add((s, a, r, s_, done))

    def get_mini_batches(self):
        n_sample = self.batch_size if self._sum_tree.size >= self.batch_size else self._sum_tree.size
        total = self._sum_tree.get_total()
        points = []
        transitions = []
        step = total // n_sample
        for i in range(n_sample):
            v = np.random.uniform(i * step, (i + 1) * step - 1)
            t = self._sum_tree.sample(v)
            points.append(t[0])
            transitions.append(t[1])
        return points, transitions

    def update(self, points, td_error):
        for i in range(len(points)):
            self._sum_tree.update(points[i], td_error[i])
```

Finally,we will introduce our hyper-parameters setting.

- **The discounting factor:** We tested different discounting factors of 0.9, 0.95 and 0.99 under the same other hyper-parameters. After some basic tests, we got the sense that they contribute to convergence to the same extent. However it showed the tendency that a higher discounting factor may cost more steps to converge. We supposed that a higher value of the discounting factor means the agent shows more interest in the future, which is harder than focusing on the present situation. Thus the training process takes more pain. Finally we chose 0.9 for the discounting factor.

- **The learning rate alpha:**We tested different alpha of 0.01, 0.005 and 0.02 under the same other hyper-parameters. The outcome showed that a higher learning rate performs well in the initial term when the agent does not have enough samples, however it is obviously over-fitting in that it could not converge later with a heavier burden of learning. But if alpha is too low, the learning process takes a long time which is so expensive for us. Finally we chosen the learning rate alpha of 0.01.

- **The batch size:**We tested different alpha of 128, 64 and 32 under the same other hyper-parameters. A larger batch size will contribute to better use of past experience, however it will take too much time to wait for it to take effect. So we chosen the batch size of 32 in order to save time while it performs well as the others.

- **The epsilon:**We used Epsilon-Greedy Strategy to ensure exploration of the environment. We tested different epsilon and its related parameters such as epsilon_decay and epsilon_min under the same other hyper-parameters. We concluded that a higher epsilon is needed in the beginning to explore more possibilities. As time goes on, the epsilon value should decays to force the agent to choose actions from the trained model, otherwise it can't converge. So we chosen epsilon of 1.0, epsilon_min of 0.01 and epsilon_decay equals to (epsilon - epsilon_min) / 50000.

3. Results and Analysis

It's a pity that we didn't get a satisfying result and by the time limit,we can only run about 2500 episodes. Because after that,the speed decrease quickly and we need nearly one hour to run 100 episodes.From the FIG 5,we consider it converges to about 500 scores.

- The first problem we met is that we did not know how to implement prioritized experience replay. The first data structure we designed was using binary priority heap, all the nodes denoting both the transitions and the priority. However we did not know how to rank according to priority when we zipped the transitions and priority in a tuple, or how could we save them separately and maintain their subscript's accordance. Then we learned about SumTree, which is a new data structure for us. By trial and error, we implemented the prioritized experience replay successfully.

- When it comes to tuning the agent, we got lost in how these parameters influence the effect of the network. Our cognition about these parameters was all about a trend, such as high learning rate will cause vibration, and if the learning rate is too low it may not converge.But we did not have specific concepts about high and low for this net, so we spent a long time to try different numbers to find the feeling. Although we thought to have found a certain suitable value for each parameter, when we combined these suitable values it still could not turn out to perform well.

- At first, we didn't tune the batch-size and it ran so slow that it took us nearly 6 hours to run 4000 episodes. After we decreased the batch-size to 32, we thought we would get lower scores and higher speed, but it surprised us that it worked better with the same parameter setting as the 128 batch-size. This sped up my training process greatly.

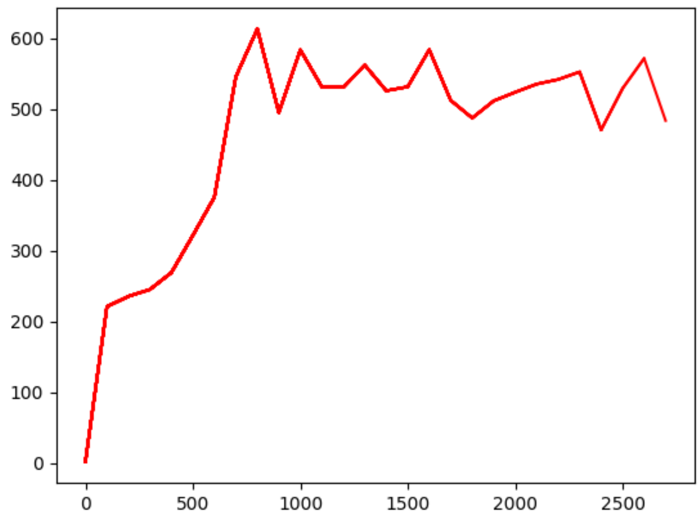


FIG. 5: DQN Result

II. EQUIPMENT

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

- macOS High Sierra (v10.13.4)
- Python 3.6 version
- TensorFlow(v1.13.1), Keras(v2.3.1), Matplotlib, Pandas

III. CONCLUSION

In this assignment, we finally tried to implement Reinforcement Learning with Dyna-Q and DQN, which we have learnt about the theories and the principles behind the concepts in class, into mechanical puzzles. And we attempted to implement Epsilon-Greedy to get rid of the explore-exploit dilemma, aiming to compel the agent to explore more possibilities of state and action.

In the first experiment, I firstly used Dyna-Q Learning to explore a maze environment. To deal with the exploration-exploitation dilemma, I used Epsilon-Greedy Strategy, and set a decay for the probability ϵ. Besides, I connected the Q value of each state and action with the times the agent has been to the next state. In that way, the agent has a tendency to enter a state which it has never been to.

In the second experiment, we completed a DQN network ourselves. To tune the agent, we implemented prioritized experience replay. Thus we can fully exploit the most important experience. Also Epsilon-Greedy Strategy is implemented to ensure exploration. Finally our agent can roughly converge.

In the two experiments, the first tough problem is how to choose the parameters. Sometimes they have close relations. When changing one parameter, the others may be influenced. So I spent a lot of time to choose the proper parameters. The another toughest problem is the time to train DQN network. Because of the limitation of the hardware, it took too much time to wait for the training process to end.

After all, I sincerely appreciate our professor Yue Gao for providing us with such a great chance to implement Reinforcement Learning algorithms into practice. And I sincerely appreciate our teaching assistants for answering our doubts and helping us solve problems. It is such a special experience to do this work.

IV. APPENDIX

A. Codes of Experiment 1

```python
from maze_env import Maze
import numpy as np
import pandas as pd
import random

UNIT = 40
MAZE_H = 6
MAZE_W = 6

class Agent:
    ### START CODE HERE ###
    ### a random agent ###

    def __init__(self, actions):
        self.actions = actions
        self.epsilon = 1

    def choose_action(self, observation):
        action = np.random.choice(self.actions)
        return action

    ### END CODE HERE ###

class myAgent:
    ### and agent of mine using Dyna-Q###

    def __init__(self,actions):
        self.actions = actions #actions
        self.epsilon = 0.1  #initialize for epsilon greedy
        self.decay = 0.99 #a decay factor on epsilon
        self.gamma = 0.8 #choose a key for gamma
        self.alpha = 0.6 #choose a key for alpha
        self.q_dict = {} #remember Q(s,a)
        self.model_dict = {} #remember Model(s,a)
        self.Dyna_N = 10 #repeat N times
        self.epsilon_dict = {}#remember the next step it can take when in this state
        self.has_been_to_this_state = {}#whether has been to this state?
        self.greedy_random_actions = [] #"random" actions for e-greedy
        self.cnt_of_state = {}# remember how many times has come to this state
        self.cnt_of_state_all_actions = {}
        self.new_state = (20.0,20.0)
        self.all_arrived = False
        self.k = 0.3
        self.epoch_num = 0 # same with episode

        for is_treasure_be_found in [False,True]:
            for i in range(MAZE_W):
                for j in range(MAZE_H):
                    '''
                    initialize state and some dictionaries and stuff
                    state(origin for x,origin for y,destination for x,destination for y,a bool)
                    every item in q state is [a,b,c,d],which corresponds to rewards of four directions
                    '''
                    state = (float(i*UNIT+5),float(j*UNIT+5),float((i+1)*UNIT-5),float((j+1)*UNIT-5),
                        is_treasure_be_found)
                    self.q_dict[state] = [0,0,0,0]
                    self.epsilon_dict[state] = [False,False,False,False]
                    self.has_been_to_this_state[float((state[0]+state[2])/2),float((state[1]+state[3])/2)] = False
                    self.cnt_of_state[state] = [0,0,0,0]
                    self.cnt_of_state_all_actions[state] = 1


    def update_random_actions_for_greedy(self,s):
        state = tuple(s)
        self.all_arrived = False
        self.has_been_to_this_state[float((state[0] + state[2]) / 2), float((state[1] + state[3]) / 2)] = True
        expected_states = []
        actions_to_get_close = []
        present_state = ((state[0]+state[2])/2,(state[1]+state[3])/2)

        #has not been to this state before
        for s1 in self.has_been_to_this_state:
            if self.has_been_to_this_state[s1] == False:
                expected_states.append(s1)

        if not expected_states:
            self.all_arrived = True
            return []

        if present_state == self.new_state:
            new_state = random.sample(expected_states, 1)
            new_state = new_state[0]
        else:
            new_state = self.new_state
        if new_state[0] > present_state[0]:
            actions_to_get_close.append(2)
        if new_state[0] < present_state[0]:
            actions_to_get_close.append(3)
        if new_state[1] < present_state[1]:
            actions_to_get_close.append(0)
        if new_state[1] > present_state[1]:
```

```python
                actions_to_get_close.append(1)
        for a2 in actions_to_get_close:
            if self.q_dict[state][a2] == -1:
                actions_to_get_close.remove(a2)
        self.new_state = new_state

        return actions_to_get_close

    def epsilon_decay(self,s):
        '''
        epsilon is large at first,then gradually decays
        :return:
        '''
        epsilon = self.epsilon
        if self.all_arrived == False:
            epsilon = 0.1
        if self.all_arrived == True:
            epsilon = 0
        return epsilon

    def update_cnt_of_state(self,s):
        state = tuple(s)
        total = 0
        for l in range(4):
            total += self.cnt_of_state[state][l]
        self.cnt_of_state_all_actions[state] = total

    def choose_action(self,s):
        '''
        choose an action,use epsilon greedy
        :param s: present state,a list
        :return: an action(x0,y0,x1,y1)
        '''
        actions_to_get_close2 = self.update_random_actions_for_greedy(s)
        state = tuple(s)  # transform list s into a tuple
        self.has_been_to_this_state[float((state[0] + state[2]) / 2), float((state[1] + state[3]) / 2)] = True
        self.epsilon = self.epsilon_decay(state)
        # use epsilon greedy
        if random.random() < self.epsilon:
            action = np.random.choice(self.actions) # a random action
        else:
            max_actions=[]
            for i in range(4):
                if self.q_dict[state][i] == max(self.q_dict[state]):
                    max_actions.append(i)
            action = random.choice(max_actions) #choose action of maximum value

        if (action == 0 and state[1] == 5) or (action == 1 and state[3] == 235) or (action == 2 and state[2] ==
            235) or \
                (action == 3 and state[0] == 5):
            self.q_dict[state][action] = -100
            self.epsilon_dict[state][action] = True
            self.update_cnt_of_state(s)
            return self.choose_action(list(s))
        else:
            return action

    def update(self,s,a,s_,r):
        '''
        update Q value
        :param s: previous state
        :param a: action
        :param s_: new state
        :param r: reward
        :return: q_dict
        '''
        state = tuple(s)
        state_ = tuple(s_)
        if self.all_arrived:
            self.k = 0


        for j in range(4):
           self.cnt_of_state[state_][j] = self.cnt_of_state[state_][j] * 0.92

        self.cnt_of_state[state][a] += 1
        self.update_cnt_of_state(s)
        self.has_been_to_this_state[float((state[0] + state[2]) / 2), float((state[1] + state[3]) / 2)] = True
        self.q_dict[state][a] += self.alpha*(r + self.gamma*max(((self.q_dict[state_][i])-self.cnt_of_state[state_
            ][i]*self.k) for i in range(4))-self.q_dict[state][a])
        self.model_dict[(state,a)] = [state_,r]

        N = self.Dyna_N
        for n in range(N): # iteration for N times
            ms,ma = random.choice(list(self.model_dict))
            ms_,mr = self.model_dict[(ms,ma)]
            self.q_dict[ms][ma] += self.alpha * (mr + self.gamma * max(((self.q_dict[ms_][i])-self.cnt_of_state[
                state_][i]*self.k)for i in range(4)) - self.q_dict[ms][ma])

        return self.q_dict
```

B. Codes of Experiment 2

```
# -*- coding:utf-8 -*-
# DQN homework.
import os
import sys
import gym
import pylab
import random
import numpy as np
from collections import deque
from keras.layers import Dense
from keras.optimizers import Adam
from keras.models import Sequential
from gym import wrappers
from utils import *
import math

# hyper-parameter.
EPISODES = 5000


class SumTree:
    def __init__(self, capacity):

        self.capacity = capacity
        self.tree = [0] * (2 * capacity - 1)
        self.data = [None] * capacity
        self.size = 0
        self.curr_point = 0


    def add(self, data):
        self.data[self.curr_point] = data
        self.update(self.curr_point, max(self.tree[self.capacity - 1:self.capacity + self.size]) + 1)

        self.curr_point += 1
        if self.curr_point >= self.capacity:
            self.curr_point = 0

        if self.size < self.capacity:
            self.size += 1


    def update(self, point, weight):
        idx = point + self.capacity - 1
        change = weight - self.tree[idx]
        self.tree[idx] = weight
        parent = (idx - 1) // 2
        while parent >= 0:
            self.tree[parent] += change
            parent = (parent - 1) // 2

    def get_total(self):
        return self.tree[0]


    def get_min(self):
        return min(self.tree[self.capacity - 1:self.capacity + self.size - 1])

    def sample(self, v):
        idx = 0
        while idx < self.capacity - 1:
            l_idx = idx * 2 + 1
            r_idx = l_idx + 1
            if self.tree[l_idx] >= v:
                idx = l_idx
            else:
                idx = r_idx
                v = v - self.tree[l_idx]

        point = idx - (self.capacity - 1)
        return point, self.data[point]


class Memory(object):
    def __init__(self, batch_size, max_size, beta):
        self.batch_size = batch_size
        self.max_size = 2 ** math.floor(math.log2(max_size))
        self.beta = beta
        self._sum_tree = SumTree(max_size)

    def store_transition(self, s, a, r, s_, done):
        self._sum_tree.add((s, a, r, s_, done))

    def get_mini_batches(self):
        n_sample = self.batch_size if self._sum_tree.size >= self.batch_size else self._sum_tree.size
        total = self._sum_tree.get_total()
        points = []
        transitions = []
        step = total // n_sample
        for i in range(n_sample):
            v = np.random.uniform(i * step, (i + 1) * step - 1)
            t = self._sum_tree.sample(v)
            points.append(t[0])
            transitions.append(t[1])
        return points, transitions
```

```
    def update(self, points, td_error):
        for i in range(len(points)):
            self._sum_tree.update(points[i], td_error[i])


class DQNAgent:
    def __init__(self, state_size, action_size):
        self.render = False
        self.state_size = state_size
        self.action_size = action_size

        # These are hyper parameters for the DQN
        self.discount_factor = 0.9
        self.learning_rate = 0.01
        self.epsilon = 1.0
        self.epsilon_min = 0.01
        self.epsilon_decay = (self.epsilon - self.epsilon_min) / 50000
        self.batch_size = 32
        self.train_start = 1000
        self.beta = 0.1
        self.maxlen = 100000
        self.memory = Memory(self.batch_size, self.maxlen, self.beta)

        # create main model
        self.model_target = self.build_model()
        self.model_eval = self.build_model()

    # approximate Q function using Neural Network
    def build_model(self):
        model = Sequential()
        model.add(Dense(128, input_dim=self.state_size, activation='relu',
                        kernel_initializer='he_uniform'))
        model.add(Dense(32, activation='relu',
                        kernel_initializer='he_uniform'))
        model.add(Dense(self.action_size, activation='linear',
                        kernel_initializer='he_uniform'))
        model.summary()
        model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
        return model

    # get action from model using epsilon-greedy policy
    def get_action(self, state, step):
        if step > 250:
            if self.epsilon > self.epsilon_min:
                self.epsilon -= self.epsilon_decay
            if self.epsilon<self.epsilon_min:
                self.epsilon=self.epsilon_min
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        else:
            q_value = self.model_eval.predict(state)
            return np.argmax(q_value[0])

    # save sample <s,a,r,s'> to the replay memory
    def append_sample(self, state, action, reward, next_state, done):
        self.memory.store_transition(state, action, reward, next_state, done)

    # pick samples randomly from replay memory (with batch_size)
    def train_model(self):
        if self.memory._sum_tree.size < self.train_start:
            return
        batch_size = min(self.batch_size, self.memory._sum_tree.size)
        points, mini_batch = self.memory.get_mini_batches()
        update_input = np.zeros((batch_size, self.state_size))
        update_target = np.zeros((batch_size, self.state_size))
        td_error = np.zeros(batch_size)
        action, reward, done = [], [], []

        for i in range(self.batch_size):
            update_input[i] = mini_batch[i][0]
            action.append(mini_batch[i][1])
            reward.append(mini_batch[i][2])
            update_target[i] = mini_batch[i][3]
            done.append(mini_batch[i][4])

        target = self.model_eval.predict(update_input)
        target_val = self.model_target.predict(update_target)

        for i in range(self.batch_size):
            # Q Learning: get maximum Q value at s' from model
            if done[i]:
                td_error[i] = abs(reward[i] - target[i][action[i]])
                target[i][action[i]] = reward[i]

            else:
                td_error[i] = abs(reward[i] + self.discount_factor * (np.amax(target_val[i])) - target[i][action[i
                    ]])
                target[i][action[i]] = reward[i] + self.discount_factor * (
                    np.amax(target_val[i]))

        # and do the model fit!
        self.model_eval.fit(update_input, target, batch_size=self.batch_size,
                            epochs=1, verbose=0)
        self.memory.update(points, td_error)

    def eval2target(self):
        self.model_target.set_weights(self.model_eval.get_weights())


if __name__ == "__main__":
    # load the gym env
```

```python
    env = gym.make('MsPacman-ram-v0')
    # set  random seeds to get reproduceable result(recommended)
    set_random_seed(0)
    # get size of state and action from environment
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    # create the agent
    agent = DQNAgent(state_size, action_size)
    # log the training result
    scores, episodes = [], []
    graph_episodes = []
    graph_score = []
    avg_length = 100
    sum_score = 0
    iteration = 0
    update_times = 0

    # train DQN
    for e in range(EPISODES):
        done = False
        score = 0
        state = env.reset()
        state = np.reshape(state, [1, state_size])
        lives = 3
        while not done:
            dead = False
            step = 0
            while not dead:
                # render the gym env
                if agent.render:
                    env.render()
                step += 1
                # get action for the current state
                action = agent.get_action(state, step)
                # take the action in the gym env, obtain the next state
                next_state, reward, done, info = env.step(action)
                next_state = np.reshape(next_state, [1, state_size])
                # judge if the agent dead
                dead = info['ale.lives'] < lives
                lives = info['ale.lives']
                # update score value
                score += reward
                # save the sample <s, a, r, s'> to the replay memory
                agent.append_sample(state, action, reward, next_state, done)
                # train the evaluation network
                if step % 4 == 0:
                    agent.train_model()
                # go to the next state
                state = next_state

            # update the target network after some iterations.
            if update_times >= 2500:
                update_times = 0
                agent.eval2target()
        # print info and draw the figure.
        if done:
            scores.append(score)
            sum_score += score
            episodes.append(e)
            # plot the reward each episode
            # pylab.plot(episodes, scores, 'b')
            print("episode:", e, "  score:", score, "  memory length:",
                 agent.memory._sum_tree.size, "  epsilon:", agent.epsilon,"learning_rate",agent.learning_rate)
        if e % avg_length == 0:
            graph_episodes.append(e)
            graph_score.append(sum_score / avg_length)
            sum_score = 0
            # plot the reward each avg_length episodes
            pylab.plot(graph_episodes, graph_score, 'r')
            pylab.savefig("./pacman_avg.png")
```