AU 332 Artificial Intelligence: Principles and Techniques

By: WangChunhui (517021910047) Ming Cheng (517021910750)(part2)

HW#: 3

October 31, 2019

I. INTRODUCTION

A. Purpose

The goal of this week lab 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 acquire a policy to get as high as possible scores in the game. The second part of homework is to build a deep q learning network.

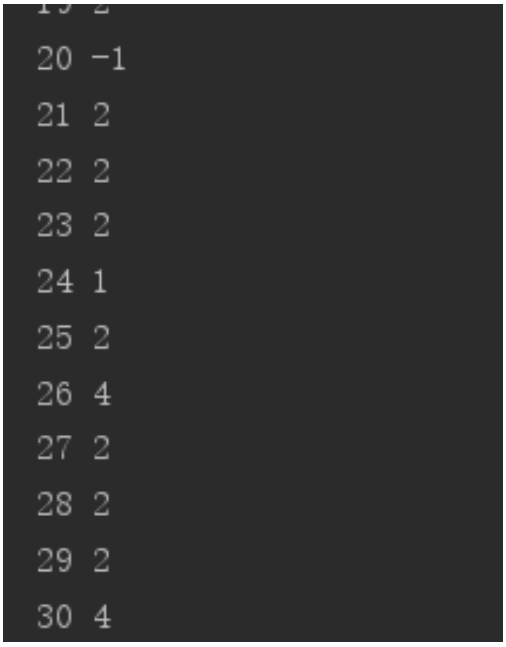
B. Equipment

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

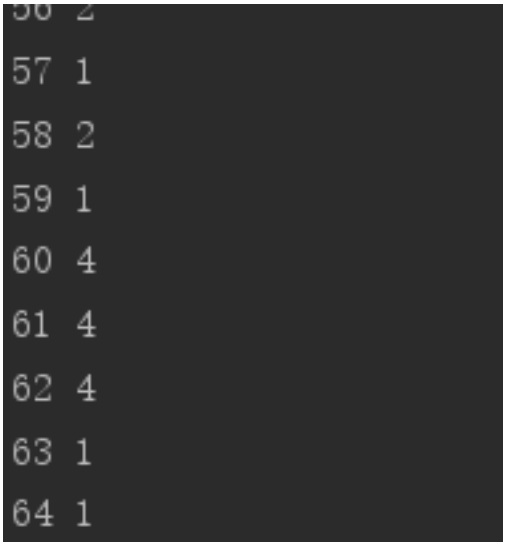
- python
- gym
- keras
- tensorflow
- numpy

II. HOMEWORK3-1

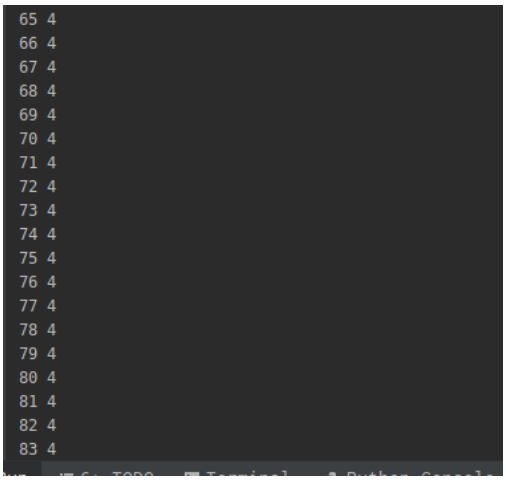
A. Dyna Q



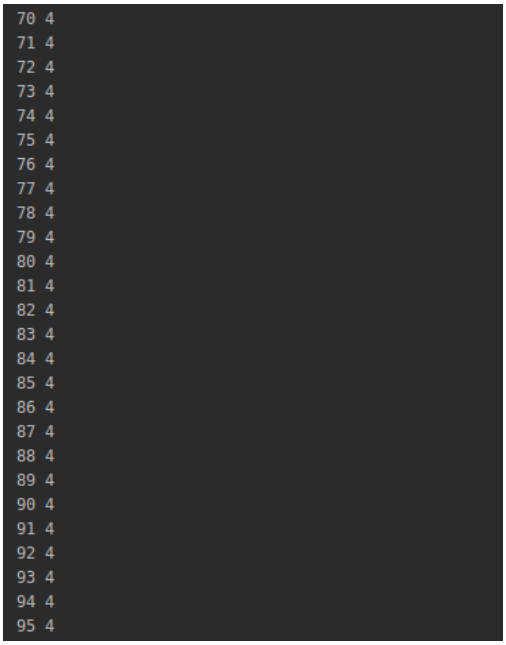
(a) First optimal stratedy appearance of maze1



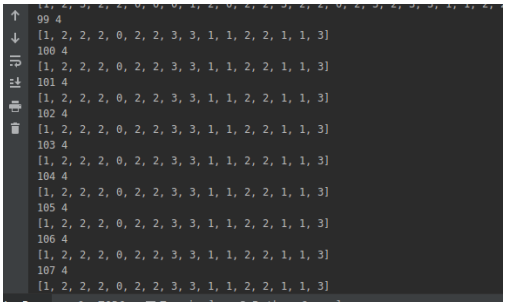
(b) First optimal stratedy appearance of maze1



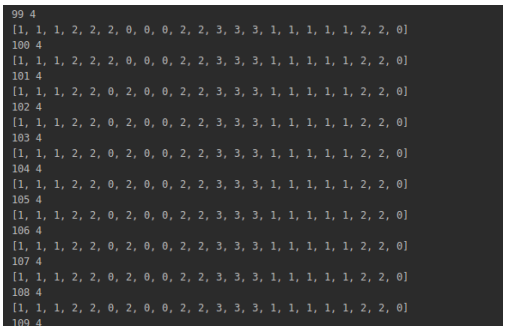
(c) converges to the best strategy in maze1



(d) converges to the best strategy in maze2



(e) maze1 path



(f) maze2 path

Results in maze 1 or maze 2,it both can find the optimal path of the game. Besides that,my agent can let the path converge to the shortest one. After 100 or a little bit more iterations,it will always choose the best solutions which will eat the diamonds and then leave and will follow the best and shortest path.In two

different mazes, the best strategy appears arround the 50th time, converges to the best strategy before the 70th time, and converges to the best path after the 100th time.

B. Exploration-Exploitation Strategy

The main paraments about the Exploration-Exploitation Strategy is gamma,alpha,and Exploration of e-greedy algorithm.

- When I choose Exploration ,which means I choose random actions,i just choose actions from the that is bigger than 0.Because the initial q table is all put 0,if the q in q table smaller than 0, it means its reward is worse than what has not been explored. So I randomly choose actions from whose q is bigger than 0.It gains a very good results.
- The alpha I don't change .I just put it constant.For I think no matter reward is positive or negtive. We should learn them all. And even in the late training,we still need to learn the new path. So I don't declcine th alpha.
- The gamma is 0.7.If the gamma is too big,it can not find the optimal path.It will round and take some unnecessary steps. But if you set it too small,it will be trapped in some high q nearby. 0.7 is a good try.It can find the shortest path.
- The most important thing is the setting of E. in the early stage, I took a high exploration rate in the first 70 rounds, so that I could fully explore the whole map

  If it's less than 70, I tested maze2 10 times, one of which failed to find the best strategy, and the other nine times converged earlier. For insurance, I set the threshold value of the first stage to 70, so that although convergence will occur later, the success rate of finding the best path is 100After making sure that I've explored the space enough, I set E as a function that goes up exponentially with the number of times. It's just to avoid the possibility that the first 70 times have not been explored. The third stage is set very high. At this time, my main strategy is to follow the best choice and explore a path that will be rewarded as soon as possible.

C. Codes

```
if __name__ == "__main__":
    env = Maze()
    RL = Agent(actions=list(range(env.n_actions)))
    env_model = EnvModel(actions=list(range(env.n_actions)))
    r_expolerd = -1


    for episode in range(120):
        s = env.reset()
        allre=0
        ass=[]
        while True:
            #env.render()

            a = RL.choose_action(str(s),episode,r_expolerd)
            if (episode > 70):
                ass.append(a)
            s_, r, done = env.step(a)

            allre=allre+r
            RL.learn(str(s), a, r, str(s_))
            env_model.remember(str(s), a, r, s_)
            for n in range(10):
                ms, ma = env_model.sample()
                mr, ms_ = env_model.get(ms, ma)
                RL.learn(ms, ma, mr, str(ms_))
            s = s_
            if done:
                print(episode,allre)
                break
    env.destroy()
    env.mainloop()
```

(g) Training process

```
    def choose_action(self, observation,n,re):
        self.check(observation)
        if(n<70):
         e=0.1
        elif(100>n>=70):
         e=1-1/(pow(1.001,n)+2)
        elif(n>=100):
         e=0.99
        elif (n >= 200):
         e = 1
        if np.random.uniform() < e:
                state_action = self.q_table.loc[observation, :]
                action = np.random.choice(state_action[state_action == np.max(state_action)].index)
        else:

                    state_action = self.q_table.loc[observation, :]
                    action = np.random.choice(state_action[state_action >=0].index)
```

(h) Dyna Q-Agent class-choose action

```
def learn(self, s, a, r, s_):
    self.check(s_)
    q_predict = self.q_table.loc[s, a]
    if s_ != 'terminal':
        q_target = r + 0.7 * self.q_table.loc[s_, :].max()
    else:
        q_target = r

    self.q_table.loc[s, a] +=0.1 * (q_target - q_predict)
```

(i) Dyna Q-Agent class-learn

```
def learn(self, s, a, r, s_):
    self.check(s_)
    q_predict = self.q_table.loc[s, a]
    if s_ != 'terminal':
        q_target = r + 0.7 * self.q_table.loc[s_, :].max()
    else:
        q_target = r

    self.q_table.loc[s, a] +=0.1 * (q_target - q_predict)
```

(j) Dyna Q-Agent class-check state

```python
class EnvModel:
    def __init__(self, actions):
        self.actions = actions
        self.database = pd.DataFrame(columns=actions, dtype=np.object)

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

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

    def get(self, s, a):
        r, s_ = self.database.loc[s, a]
        return r, s_
```

(k) Dyna Q-env class

III. HOMEWORK3-2

1. result during modifying the code

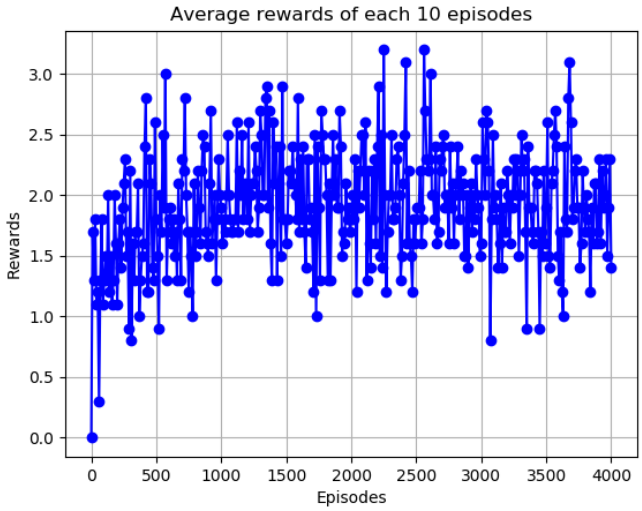


FIG. 1: first result of the question 2

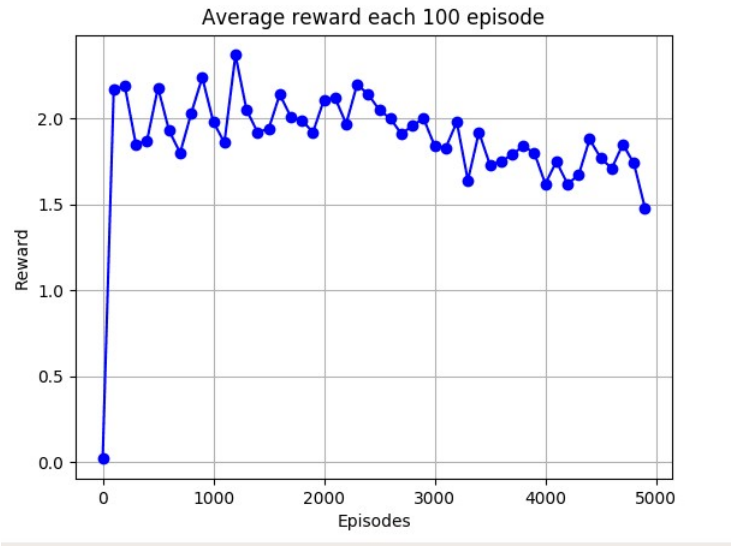


FIG. 2: second result of the question 2

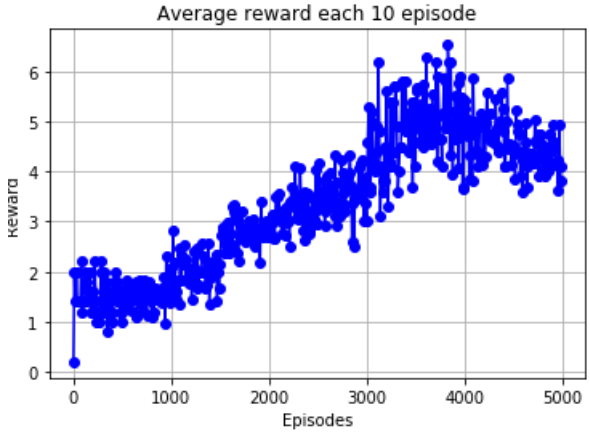


FIG. 3: last successful result of the question 2

According to the first figure, we can see that at the beginning of the training when *episode* is less than 1000, the average rewards increases fast and can reach to around 2. However, as the training process goes, the average rewards do not continue to increase, but stay almost unchanged instead. In other words, the training process does not behave satisfactorily during the middle and late period of the whole procedure.

According to the second figure, we can see that after trying to train for 5000 times, we can see that at the beginning of the procedure, the rewards increase very fast and can reach to more than 2. However, after 2000 times of iteration, the rewards begin to decrease, which is not expected.

Therefore, we can conclude that during the middle and late period of the whole training process, the network begins to behave like the over-fitting phenomenon. So we should modify our model to eliminate the over-fitting phenomenon.

As we can see from the final figure, the average reward each 10 episodes increases to 6 at most, which is pretty satisfatory. Although there exists over-fitting phenomenon at the last 1000 episodes, we have weaken it by adding drop-out layer into the network. As the figure shows, we can get the best value, around 5.2, if

we train for around 3500 episodes, which is pretty fast.

2. code explanation

In the whole process of improving DQN network, we have adopted many methods. Here are some of the improvements we think are useful to our entire training process.

- Add a dropout layer for the previously encountered results with worse performance in the later stage. The k value of dropout is set to 0.3, with a 30 percent chance of dropout.
- Improved the batchsize from 16 to 32. We believe that in the operation of minibatch, increasing batch will have a good improvement.
- We improved the activation function of the last layer before and set it as softmax. After improving the activation function, we found that the activation function of softmax was much better than the previous one.
- . Improve epsilon. We thought epsilon was too small, so we set up a function on epsilon, and in the early days we explored as many new ways as we could, in the middle we basically had a 50/50 chance of running the network, and a 50/50 chance of running it randomly. Later basically adopted the way of network operation.
- The key is we found that the effect is poor because learning a lot of bad data, then we will study the standard set to reward is greater than 0, but found that learning data is too little, so we found more than 270 steps and summing up the general can get higher scores, will study whether step is greater than 270 as the standard.
- Our target-train interval is basically 2000 steps. This data debugging took a lot of time, and it was found that the results were better if the interval was larger. Our replay is set to every five steps. If the Settings are too large, the training will be poor, and if the Settings are too small, the training will be very long.

Below are figures of the code:

```
def create_model(self):
    model = Sequential()
    model.add(Dense(128*2, input_dim=self.state_size,activation='relu'))
    model.add(Dense(128*2, activation='relu'))
    model.add(Dense(128*2, activation='relu'))
    model.add(Dropout(0.3))
    model.add(Dense(self.env.action_space.n, activation='softmax'))
    model.compile(loss='mean_squared_error', optimizer=optimizers.RMSprop(lr=self.learning_rate,decay=0.99,epsilon=1e-
    return model
```

FIG. 4: code of *create_model* function

```
def choose_action(self, state, steps,episode):
    if episode<=1000:
        self.epsilon = 1-episode*0.0001
    elif episode <= 3000:
            self.epsilon = 0.8 - (episode-1000) * 0.0003
    elif episode <= 5000:
        self.epsilon =0.2*pow(0.9985,episode-3000)



    if np.random.random() < self.epsilon:
        return self.env.action_space.sample()
    return np.argmax(self.evaluation_model.predict(state.reshape(128,1).T)[0])
```

FIG. 5: code of *choose_action* function

```
def main():
    env = gym.make('Breakout-ram-v0')
    env = env.unwrapped

    episodes =3000
    trial_len = 10000
    tmp_reward=0
    sum_rewards = 0
    graph_reward = []
    graph_episodes = []
    dqn_agent = DQN(env=env)
    ####### Training ######
    ### START CODE HERE ###
    graph_reward1 = []
    maxreward=[]
    episode_reward10=0
    episode_reward100=0
    step=0
```

FIG. 6: code of *main* function (part 1)

```
for episode in range (episodes):
        print("episode",episode)
        obseervation=env.reset()
        step1=0
        episode_reward=0
        done=False
        for step1 in range (trial_len):
            step+=1
            step1 += 1
            action=dqn_agent.choose_action(obseervation,step,episode)
            obseervation_,r,done,infor=env.step(action)
            episode_reward += r

            obseervation=obseervation_

            if (step1 > 270):
                    dqn_agent.remember(obseervation, action, r, obseervation_, done)
                    if (step % 1000 == 0):
                        dqn_agent.target_train()
```

FIG. 7: code of *main* function (part 2)

```
                    dqn_agent.replay()
            else:
                    if np.random.uniform() < 0.5:
                        dqn_agent.remember(obseervation, action, r, obseervation_, done)
                    if (step % 1000 == 0):
                            dqn_agent.target_train()

                            dqn_agent.replay()




            if done:
                print(episode_reward)
                maxreward.append((episode_reward))
                print(max(maxreward))
                print(step1)
                print(dqn_agent.epsilon)
                break
        episode_reward10+=episode_reward
        episode_reward100 += episode_reward
```

FIG. 8: code of *main* function (part 3)