# AU 332 Artificial Intelligence: Principles and Techniques

By: Yangqian Wu

Instructor: Yue Gao

October 31, 2019

# I. INTRODUCTION

In homework 3, the topic is focus on reinforcement learning.

In part I, I implement a agent based on Dyna-Q learning to find a best path which can select the highest reward from the environment. In the process, I found that it's hard to guide the agent to select the treasure after it find the +1 reward. Finally, I use the exploration function to solve the problem.

In part II, I implement a gent to play the atari breakout game. By using the DQN method, we achieve the 5 points goal within 5000 iterations. But we also found that there still are several improvement we might have.

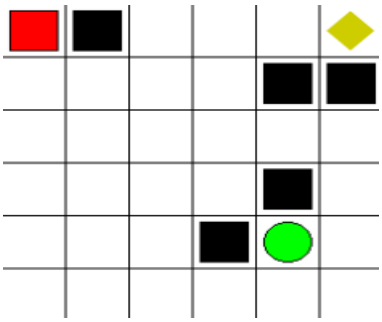
# II. REINFORCEMENT LEARNING IN MAZE ENVIRONMENT

## A. Game Description

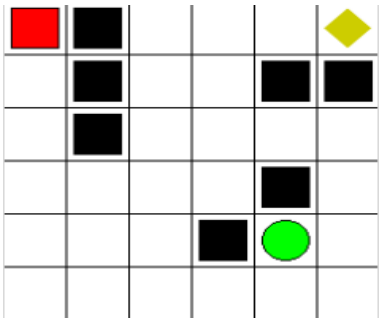
Suppose a 66 grid-shaped maze. The red rectangle represents the start point and the green circle represents the exit point. The agent can move upward, downward, leftward and rightward.We want the agent to avoid falling into the traps, which are represented by the black rectangles. Finding the exit will give a reward +1 and falling into traps will cause a reward -1, and both of the two cases will terminate current iteration.Moreover, the agent will get a bonus reward +2 if it find the treasure, which shown as golden diamond.

The state space and action space are briefly described as follows.

- 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.



(a) The picture of Maze1



(b) The picture of Maze2

I use the Dyna-Q learning to implement my agent, which will be trained by the reward recieved from the environment. After hundreds of iterations, the Q-table could be converged and the Dyna-Q agent have already learned the environment. Therefore, it could choose the best action according to current state by choosing the highest Q-value.

We want our agent to get as high score as possible within least iterations. One difficulty is that how to achieve the global optimum instead of local optimum, I will talk about this in the third part. S

### B. Dyna-Q Learning

In this section, I implement an agent based on Dyna-Q learning, which can find optimal policy both in maze1 and maze2.

I created a Dyna-Q agent who is able to choose the best action in a certain state based on the Q-value it has learned.

In Dyna-Q learning, the difference from Q-learning is that it simulate many times after take a real action. That means we can get a more precise model if we can simulate more times after a real action.

Therefore, In agent.py, I created four functions:choose_action(observation),update(s,a,r,s_),store(s,a,r,s_),simulate(). I'will talk about these one by one below:

1. choose_action(observation):

   There are two versions of strategy to choose the action.

   The first one is using epsilon-greedy strategy: we set a epsilon to let the agent choose action randomly, for the purpose of exploring more states to get more information. In the early game, we set epsilon as a low value, wanting the agent to explore more, as game continuing, we increase the epsilon to make a better performance.

   The second is exploration function strategy which is taken in the final py file I submitted: In this strategy, I created three tables to store the calculated Q-value, real Q-value and the times a state has been reached. If a state is reached a little times, the function Q+k/N will produce a tendency whcih can lead the agent towards the state with little reaching times. That will help the agent to find the treasure and to find the global optimum instead of loacl one.

```
def choose_action(self, observation):
    if observation not in self.q_table.index:
        self.q_table = self.q_table.append(pd.Series([0]*len(self.actions),index=self.q_table.columns,name=observation,))
    if observation not in self.ob_table.keys():
        self.ob_table[observation]=1
    if observation not in self.func_table.index:
        self.func_table = self.func_table.append(pd.Series([1]*len(self.actions),index=self.func_table.columns,name=obse

    if np.random.uniform() < self.epsilon:
        action = self.func_table.ix[observation,:]
        if action[0] == action[1] and action[1] == action[2] and action[2] == action[3]:
            action = np.random.choice(self.actions)
        else:
            action = action.reindex(np.random.permutation(action.index))
            action = action.argmax()

    else:
        action = np.random.choice(self.actions)    #随机选择一个动作

    self.ob_table[observation] += 1
    return action
```

(c) The code for function choose action

2. update(s,a,r,s_)

   In this function, I update the Q-value with the exploration function $Q^* = Q^* + k/N$.

   In this function, firstly check whether the state s_ is on the three tables; secondly calculate the biggest function value and Q-value in four actions which can be taken in state s_; finally update the function table and Q-value table.

```
def update(self, s, a, r, s_):
    if s_ not in self.q_table.index:
        self.q_table = self.q_table.append(pd.Series([0]*len(self.actions),index=self.q_table.columns,name=s_))
    if s_ not in self.ob_table.keys():
        self.ob_table[s_]=1
    if s_ not in self.func_table.index:
        self.func_table = self.func_table.append(pd.Series([1]*len(self.actions),index=self.func_table.columns,name=s_

    coe = 1
    func_update = -100
    for action in self.actions:
        func_tmp = r + self.decay * (self.q_table.ix[s_,action] + coe/self.ob_table[s_])
        if func_update < func_tmp:
            func_update = func_tmp
    self.func_table.ix[s, a] = (1 - self.rate) * self.func_table.ix[s,a] + self.rate * (func_update - self.func_table.

    q_update = -100
    for action in self.actions:
        q_tmp = r + self.decay * self.q_table.ix[s_,action]
        if q_update < q_tmp:
            q_update = q_tmp
    self.q_table.ix[s, a] = (1 - self.rate) * self.q_table.ix[s,a] + self.rate * (q_update - self.q_table.ix[s,a])
```

(d) The code for function update

3. store(s,a,r,s_)

   This function is uesd in simulation. We save the next stable and its reward in the position of (s,a) in action table.

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

(e) The code for function store

4. simulate()

   It's the biggest difference between Q-learning and Dyna-Q learning. After we step a real action, we simulate several times to make the model more precise.

   The process is that we choose a (state,action,reward,next state) object from the action table. Then simulate this step virtually. Finally, we update the three tables we have.

```
def simulation(self):
    while True:
        s_simu = np.random.choice(self.action_table.index)
        a_simu = np.random.choice(self.action_table.loc[s_simu].index)
        if self.action_table.ix[s_simu,a_simu] != None:
            break
    r_simu,ns_simu = self.action_table.ix[s_simu,a_simu]
    return s_simu,a_simu,r_simu,ns_simu
```

(f) The code for function simulate

### C. Exploration-Exploitation Strategy

Considering the explore-exploit dilemma, I complete my Dyna-Q learning agent by implementing an exploration exploitation strategy of exploration function, meaning it chooses actions besed on both the Q-value and the times a state has been visited to explore better policy.

It's obvious that in the early stage of the training, exploration should be encouraged as we hope that the agent can find better policy (e.g. getting the treasure). Then with the advancement of training, exploration may be reduced for the divergence of the policy. Our goal is to get high score with as few as possible interactions.

After a long time of trying.I found the exploration function is quite suitable in this task. I list the procedures of implementing the exploration function below:

1. Created the Q_table,function_table and ob_table to store the Q-value, function value and the times a state has been visited.
2. When steping an action or simulate an action, I update the tables to make the model more precise.
3. When choosing a best action based on a certain state, I choose the action which has the highest function value. Moreover, I update the ob_table with the current state plus one.

### D. The result of homework and some thoughts about it

#### *1. The result of homework*

I set the train iterations as 300 times to make sure the best policy could be get.(Although in real training, I found that only 30 iterations for Maze1 and 80 90 iterations for Maze2 is enough.)

- Maze1:

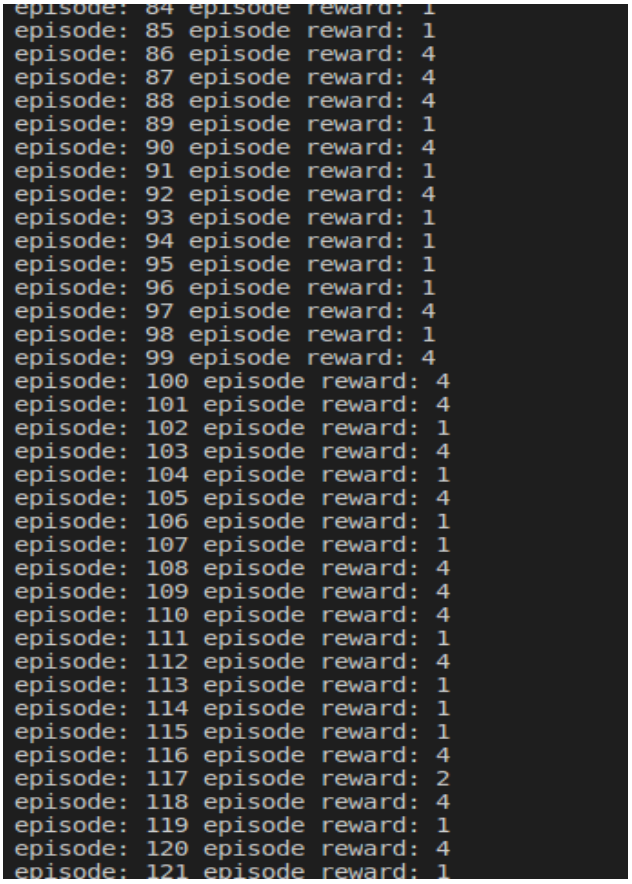
  The best policy converge in 30 iterations. And the test implies it can achieve the best path.


(g) The training for Maze1

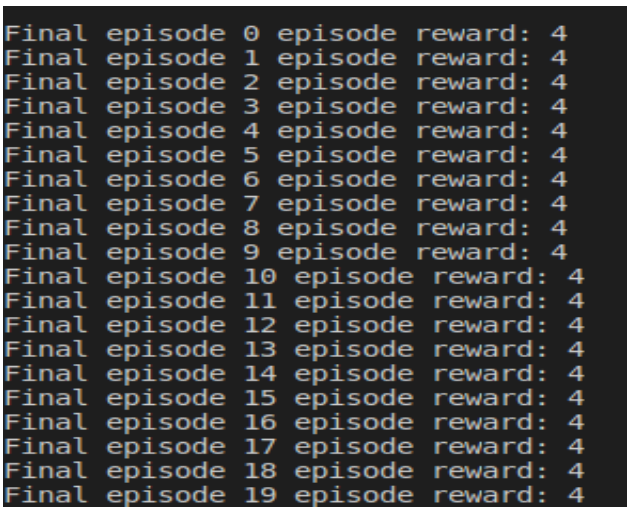

(h) The result for Maze1

- Maze2:

  The best policy converge in 100 iterations. And the test implies it can achieve the best path.




(i) The training for Maze2




(j) The result for Maze1

*2. Some thoughts about homework*

There are several ideas I come up with related to the assignment.

- Besides the exploration function, are there any other functions we can use to help us find the global optimum? After reading the pdf the professer Gao gives, I found a lot of other approachs.
- Considering the state of Maze is not that big, we can use tables to store our Q-values. But if the maze become bigger, perhaps we can try DQN as well.
- In the experiment, I found that Dyna-Q learning didn't play better than Q-learning. That's a question I still think about.

## III. REINFORCEMENT LEARNING ON ATARI GAME

In this part, we implement a DQN agent to play atari game breakout.We accomplish the remaining part which train the agent to perform better and get at least 5 points within 5000 iterations. In order to achieve this, we have to tune the weights and other parts to get higher scores.

### A. Game Description

Atari breakout is one of the classic and leading games. We need to bounce back the ball to crush all of theboxes which are showing upward. In this assignment, we design a DQN agent to learn control policies directly from the visual information of the game.


(k) The game of breakout-ran-v0

### B. Deep Q-Learning

In this part, we implement a DQN agent, and complete the training process in main(). In the training process, we set the iterations limitation of 5000.

We set two neural networks, which is to predict the Q-value if we give the input of a state.The target network is used to predict the Q-value of the current state, which is trained by the reward and the Q-value given by evauation network. The evaluation network is the old version of target network, which updates in a certain frequency.

The whole process is as below:

1. The brief structure is quite like Dyna-Q learning, but in Deep Q Network, we use the neural network to acquire the Q-value by inputing a state.
2. When start training, we choose a action based on current state(In the early game, most of the actions are randomly choose to explore more).Then we apply it, and get reward,next state,done,etc.We store the information for later's replay.
3. We repeat the process above and replay every several steps in order to update the target network.
4. Approximately, we update the evaluation network weights every 20000 steps, which can help us train the target network more efficiently.
5. In thousands of iterations, we may happily observe that our agent perform better than before.

```
#############The frequency of replay and update evalaution network########
T_target = 7000
T_main = 16
########################################################################

episode100_reward = 0
total_reward = 0
update_count = 0
sum_step = 0
result=[]

for episode in range(episodes):
    episode_reward = 0
    s = env.reset().reshape(1,128)
    s = preprocess(s)

    for step in range(trial_len):

        if episode >= 3000:
            env.render()

        update_count += 1
        sum_step += 1
        action = dqn_agent.choose_action(s, sum_step)
        s_, reward, done, info =env.step(action)
        s_ = s_.reshape(1,128)
        s_ = preprocess(s_)

        episode_reward += reward
        episode100_reward += reward
        total_reward += reward
        dqn_agent.remember(s, action, reward, s_,done)
```

(l) The code of training

### C. Tune the Agent

It's a quite hard experience that we run the original DQN code and found that whole night's training results in no progress.

It's undeniable that we have to tune the agent whether in parameters or in neural network structures. In our work, we mainly tuned the agent in following aspects:

1. We preprocess the input state, the (1*128) matrix is divided by 255, which can help the traing process to converge more easily.

2. We tune the parameters related to epsilon-greedy. We hope the randomness of agent to choose action based on state can gradually decrease when training is continuing.That's the banlance between exploration and exploitation.

3. We also tuned the frequency of replay and update the evaluation network. We found that more frequently we replay and update, more scores we will receive when training certain iterations. But it will cost more time if we increase the frequency, so we keep a balance after a range of experiments.

4. We do change the neural network professor provided. Beacause we find it is more useful if we add a layer, and it might cost more time if we add other more layers.

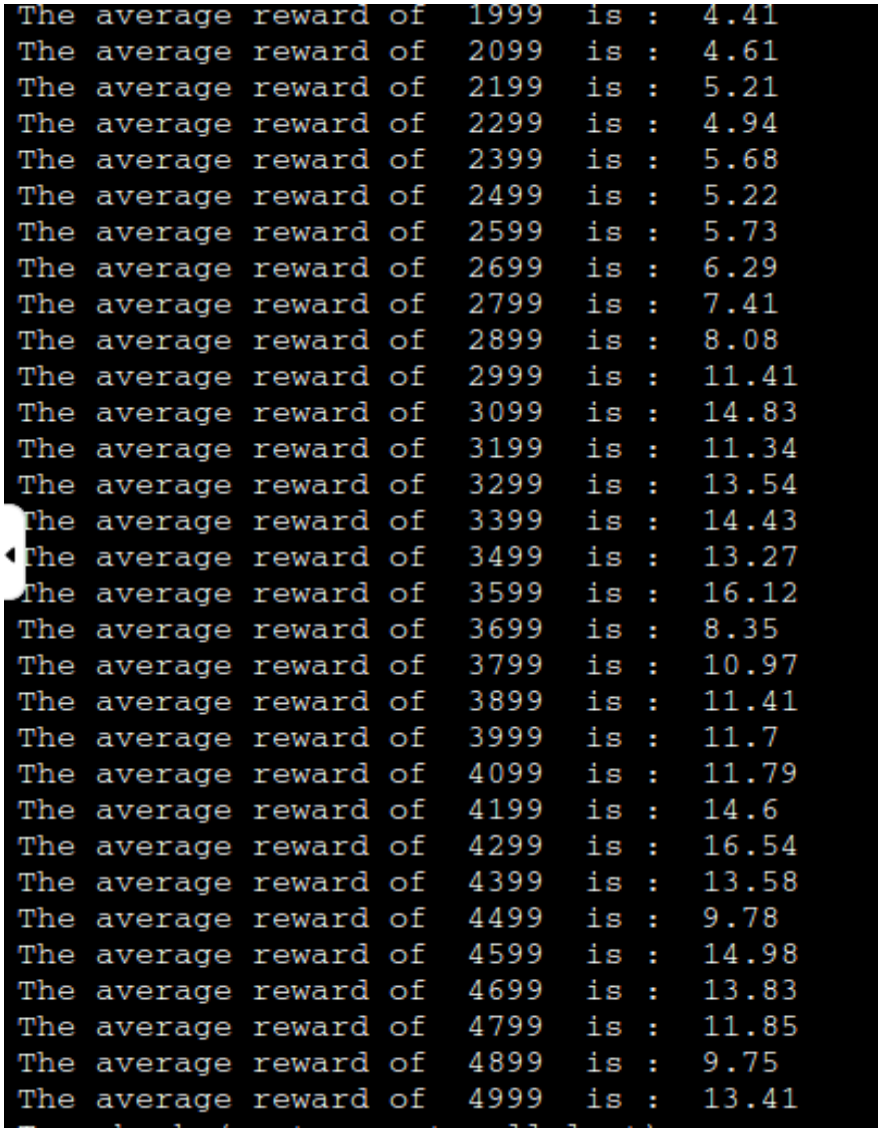
```
def __init__(self, env):
    self.env = env
    self.memory = deque(maxlen=500000)
    self.gamma = 0.99
    self.epsilon = 1.0
    self.epsilon_min =0.01
    self.epsilon_decay = (self.epsilon-self.epsilon_min) / 1000000

    self.batch_size = 32
    self.train_start = 20000
    self.state_size = self.env.observation_space.shape[0]
    self.action_size =self.env.action_space.n
    self.learning_rate = 0.0001

    self.evaluation_model = self.create_model()
    self.target_model = self.create_model()

def create_model(self):
    model = Sequential()
    model.add(Dense(128*2, input_dim=self.state_size,activation='relu', kerne
    model.add(Dense(128*2, activation='relu', kernel_initializer="he_uniform"
    model.add(Dense(128*2, activation='relu', kernel_initializer="he_uniform"
    model.add(Dense(128*2, activation='relu', kernel_initializer="he_uniform"
    model.add(Dense(self.env.action_space.n, activation='linear'))
    optimizer = optimizers.Adam(lr=self.learning_rate)
    model.compile(loss='mean_squared_error', optimizer=optimizer)
    return model
```

(m) The code we tuned the agent

### D. Results

- In our code, Tmain is the frequency to make experience replay. Ttarget is the frequency to update the network. C is the time constance for epsilon decay.

- After normalizing the input and weights of each layer, our training becomes much more efficient and effective.
- We find that the more time to explore, the more promising the agent will be. We have tried C=1000000 and C=2000000. Finally we choose the former one because it's enough for 5000 episodes.
- We find that 4 layer-network leads higher scores more quickly but also suffers from overfitting. While the training of 3-layer network is more stable, which proves that a deeper network brings about stronger fitting ability.
- The highest score we get within 100 episodes is over points. However it drops quickly because of the problem of overfitting. This result is achieved by this set of parameters: Tmain = 6, Ttarget = 5000, C = 1000000.




(n) Training process figure


(o) Training process

### E. Some thoughts in the homework

Altough after days of exploring, we achieve the 5 points after 5000 iterations. We still come up with several ideas which might make some improvement.

- Now, the input we give the neural network is a (1*128) matrix, which might not provide enough information we want. Because a state only provide information within one scene, it's hard to get the direction information from it. Therefore, we assume that combine 4 continual states as a matrix of (1*512) might be helpful.
- By the limitation of equipment and time, we do not increase the frequency of replaying and updating the evaluation network, which might cost more and more time to run. That's also a aspect we can consider.
- In this homework, we do not change the layer of neural network too much, for we think it's enough to accomplish the task. However, if we want to get high score in certain iterations, we think it's necessary to add more layers to get better performance.