School of Data and Computer Science, Sun Yat-sen University

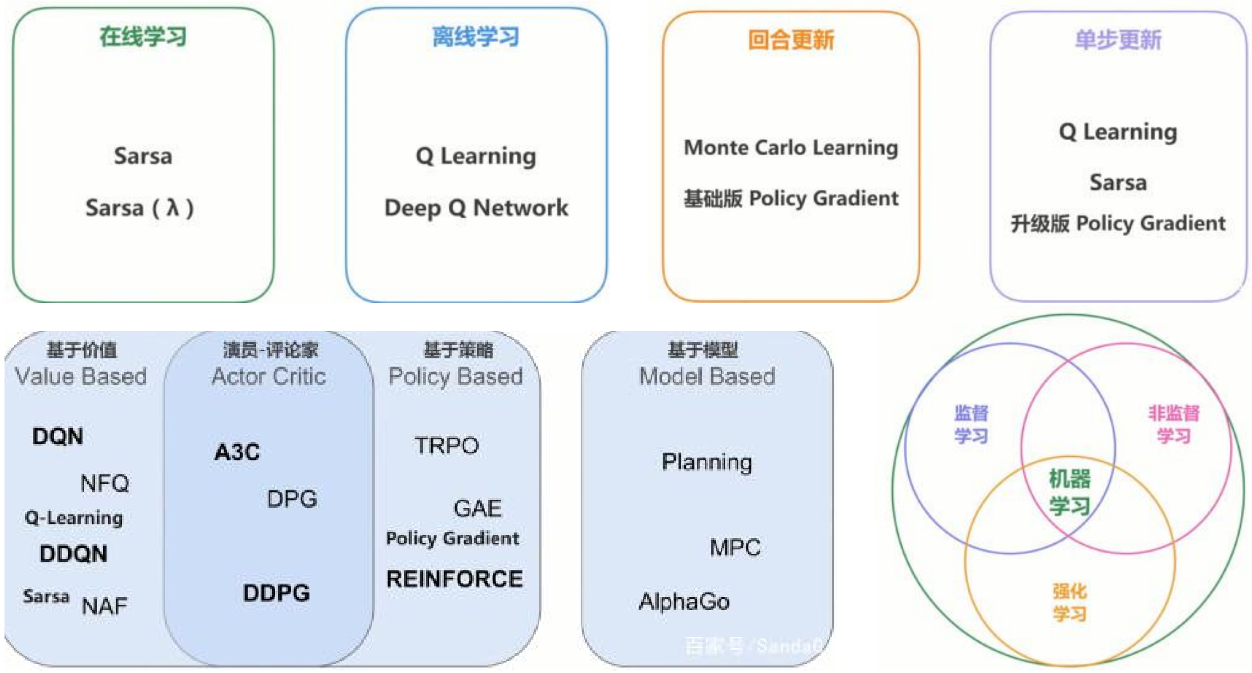
# E15 Reinforcement Learning (C++/Python)

Suixin Ou    Yangkai Lin

December 16, 2020

## Contents

## 1 Overview



## 2 Tutorial

English version: `http://mnemstudio.org/path-finding-q-learning-tutorial.htm`

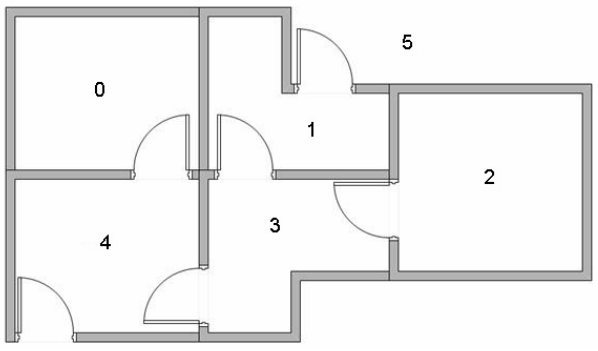
Chinese version: `https://blog.csdn.net/itplus/article/details/9361915`

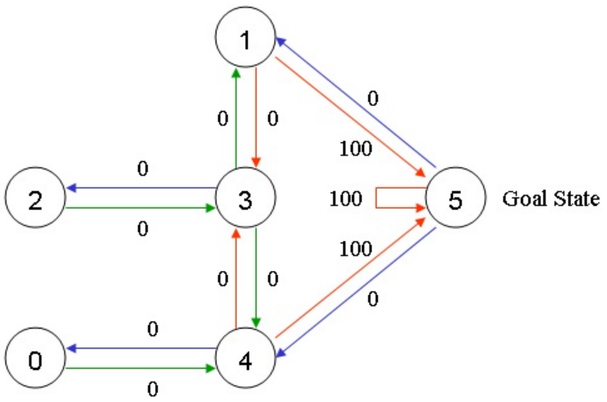
### 2.1 Step-By-Step Tutorial

Suppose we have 5 rooms in a building connected by doors as shown in the figure below. We'll number each room 0 through 4. The outside of the building can be thought of as one big room (5). Notice that doors 1 and 4 lead into the building from room 5 (outside). We can represent the rooms on a graph, each room as a node, and each door as a link.

For this example, we'd like to put an agent in any room, and from that room, go outside the building (this will be our target room). In other words, the goal room is number 5. To set this room as a goal, we'll associate a reward value to each door (i.e. link between nodes). The doors that lead immediately to the goal have an instant reward of 100. Other doors not directly connected to the target room have zero reward. Because doors are two-way ( 0 leads to 4, and 4 leads back to 0 ), two arrows are assigned to each room. Each arrow contains an instant reward value, as shown below:

Of course, Room 5 loops back to itself with a reward of 100, and all other direct connections to





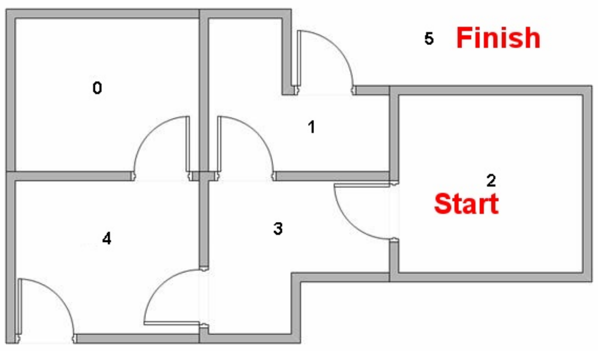
the goal room carry a reward of 100. In Q-learning, the goal is to reach the state with the highest reward, so that if the agent arrives at the goal, it will remain there forever. This type of goal is called an "absorbing goal".

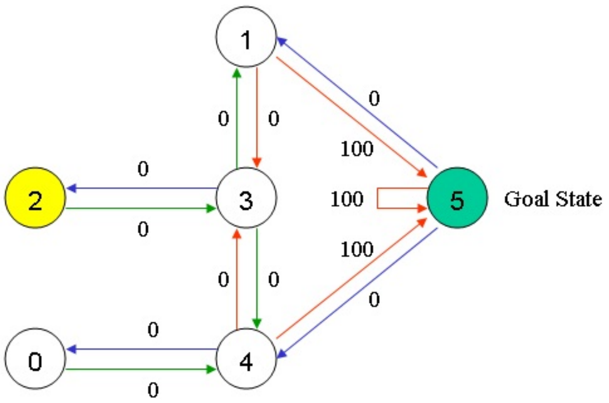
Imagine our agent as a dumb virtual robot that can learn through experience. The agent can pass from one room to another but has no knowledge of the environment, and doesn't know which sequence of doors lead to the outside.

Suppose we want to model some kind of simple evacuation of an agent from any room in the building. Now suppose we have an agent in Room 2 and we want the agent to learn to reach outside the house (5).

The terminology in Q-Learning includes the terms "state" and "action".

We'll call each room, including outside, a "state", and the agent's movement from one room to another will be an "action". In our diagram, a "state" is depicted as a node, while "action" is represented by the arrows.





Suppose the agent is in state 2. From state 2, it can go to state 3 because state 2 is connected to 3. From state 2, however, the agent cannot directly go to state 1 because there is no direct door connecting room 1 and 2 (thus, no arrows). From state 3, it can go either to state 1 or 4 or back to 2 (look at all the arrows about state 3). If the agent is in state 4, then the three possible actions are to go to state 0, 5 or 3. If the agent is in state 1, it can go either to state 5 or 3. From state 0, it can only go back to state 4.

We can put the state diagram and the instant reward values into the following reward table, "matrix R".

$$R = \begin{array}{c|cccccc} & & & \text{Action} & & & \\ \text{State} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & -1 & -1 & -1 & -1 & 0 & -1 \\ 1 & -1 & -1 & -1 & 0 & -1 & 100 \\ 2 & -1 & -1 & -1 & 0 & -1 & -1 \\ 3 & -1 & 0 & 0 & -1 & 0 & -1 \\ 4 & 0 & -1 & -1 & 0 & -1 & 100 \\ 5 & -1 & 0 & -1 & -1 & 0 & 100 \end{array}$$

The -1's in the table represent null values (i.e.; where there isn't a link between nodes).

Now we'll add a similar matrix, "Q", to the brain of our agent, representing the memory of what the agent has learned through experience. The rows of matrix Q represent the current state of the agent, and the columns represent the possible actions leading to the next state (the links between the nodes).

The agent starts out knowing nothing, the matrix Q is initialized to zero. In this example, for the simplicity of explanation, we assume the number of states is known (to be six). If we didn't know how many states were involved, the matrix Q could start out with only one element. It is a simple task to add more columns and rows in matrix Q if a new state is found.

The transition rule of Q learning is a very simple formula:

$$Q(state, action) = (1-\alpha) * Q(state, action) + \alpha * (R(state, action) + \gamma \max[Q(nextstate, allactions)])$$

According to this formula, a value assigned to a specific element of matrix Q, is equal to the sum of the corresponding value in matrix R and the learning parameter $\gamma$, multiplied by the maximum value of Q for all possible actions in the next state. Here the $\alpha$ is a hyper-parameter similar to $\gamma$, which is used to **assure the convergence** of Q-learning.

Our virtual agent will learn through experience, without a teacher (this is called unsupervised learning). The agent will explore from state to state until it reaches the goal. We'll call each exploration an episode. Each episode consists of the agent moving from the initial state to the goal state. Each time the agent arrives at the goal state, the program goes to the next episode.

The Q-Learning algorithm goes as follows:

```
1  Set the gamma parameter, and environment rewards in matrix R;
2  Initialize matrix Q to zero;
3  foreach episode do
4      Select a random initial state;
5      while the goal state hasn' t been reached do
6          Select one among all possible actions for the current state;
7          Using this possible action, consider going to the next state;
8          Get maximum Q value for this next state based on all possible actions;
9          Compute:
               $Q(state, action) = (1 - \alpha) * Q(state, action)$
               $+\alpha * (R(state, action) + \gamma \max[Q(nextstate, allactions)])$
           ;
10         Set the next state as the current state;
11     end
12 end
```
**Algorithm 1:** The Q-Learning Algorithm

The algorithm above is used by the agent to learn from experience. Each episode is equivalent to one training session. In each training session, the agent explores the environment (represented by matrix R ), receives the reward (if any) until it reaches the goal state. The purpose of the training is to enhance the 'brain' of our agent, represented by matrix Q. More training results in a more optimized matrix Q. In this case, if the matrix Q has been enhanced, instead of exploring around, and going back and forth to the same rooms, the agent will find the fastest route to the goal state.

The $\gamma$ parameter has a range of 0 to 1 ($0 \leq \gamma < 1$). If $\gamma$ is closer to zero, the agent will tend to consider only immediate rewards. If $\gamma$ is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

To use the matrix Q, the agent simply traces the sequence of states, from the initial state to goal state. The algorithm finds the actions with the highest reward values recorded in matrix Q for current state:

Algorithm to utilize the Q matrix:

1. Set current state = initial state.
2. From current state, find the action with the highest Q value.
3. Set current state = next state.
4. Repeat Steps 2 and 3 until current state = goal state.

The algorithm above will return the sequence of states from the initial state to the goal state.

## 2.2 Q-learning Example By Hand

To understand how the Q-learning algorithm works, we'll go through a few episodes step by step. The rest of the steps are illustrated in the source code examples. **In this illustration, we ignore the hyper-parameter $\alpha$, because the algorithm can convergence in this simple case without $\alpha$. Maybe you need to add it in our task2 flappy bird.**

We'll start by setting the value of the learning parameter $\gamma = 0.8$, and the initial state as Room 1. Initialize matrix Q as a zero matrix. Look at the second row (state 1) of matrix R. There are two possible actions for the current state 1: go to state 3, or go to state 5. By random selection, we select to go to 5 as our action.

Now let's imagine what would happen if our agent were in state 5. Look at the sixth row of the reward matrix R (i.e. state 5). It has 3 possible actions: go to state 1, 4 or 5.

$$Q(state, action) = R(state, action) + \gamma \max[Q(nextstate, allactions)]$$

$$Q(1,5) = R(1,5) + 0.8 \times \max[Q(5,1), Q(5,4), Q(5,5)] = 100 + 0.8 \times 0 = 100$$

$$Q = \begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \qquad R = \begin{array}{c|cccccc} & & & \text{Action} & & & \\ \text{State} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & -1 & -1 & -1 & -1 & 0 & -1 \\ 1 & -1 & -1 & -1 & 0 & -1 & 100 \\ 2 & -1 & -1 & -1 & 0 & -1 & -1 \\ 3 & -1 & 0 & 0 & -1 & 0 & -1 \\ 4 & 0 & -1 & -1 & 0 & -1 & 100 \\ 5 & -1 & 0 & -1 & -1 & 0 & 100 \end{array}$$

Since matrix Q is still initialized to zero, Q(5, 1), Q(5, 4), Q(5, 5), are all zero. The result of this computation for Q(1, 5) is 100 because of the instant reward from R(5, 1).

The next state, 5, now becomes the current state. Because 5 is the goal state, we've finished one episode. Our agent's brain now contains an updated matrix Q as:

$$Q = \begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 100 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

For the next episode, we start with a randomly chosen initial state. This time, we have state 3 as our initial state.

Look at the fourth row of matrix R; it has 3 possible actions: go to state 1, 2 or 4. By random selection, we select to go to state 1 as our action.

Now we imagine that we are in state 1. Look at the second row of reward matrix R (i.e. state 1). It has 2 possible actions: go to state 3 or state 5. Then, we compute the Q value:

$$Q(state, action) = R(state, action) + \gamma \max[Q(nextstate, allactions)]$$

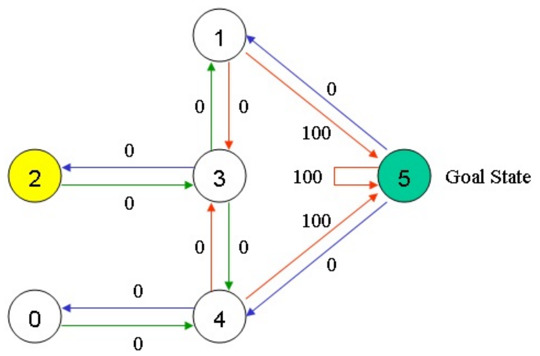
$$Q(3,1) = R(3,1) + 0.8 \times \max[Q(1,2), Q(1,5)] = 0 + 0.8 \times \max(0,100) = 80$$

We use the updated matrix Q from the last episode. Q(1, 3) = 0 and Q(1, 5) = 100. The result of the computation is Q(3, 1) = 80 because the reward is zero. The matrix Q becomes:

The next state, 1, now becomes the current state. We repeat the inner loop of the Q learning algorithm because state 1 is not the goal state.

$$
Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 100 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 80 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

So, starting the new loop with the current state 1, there are two possible actions: go to state 3, or go to state 5. By lucky draw, our action selected is 5.



Now, imaging we're in state 5, there are three possible actions: go to state 1, 4 or 5. We compute the Q value using the maximum value of these possible actions.

$$Q(state, action) = R(state, action) + \gamma \max[Q(nextstate, allactions)]$$

$$Q(1,5) = R(1,5) + 0.8 \times \max[Q(5,1), Q(5,4), Q(5,5)] = 100 + 0.8 \times 0 = 100$$

The updated entries of matrix Q, Q(5, 1), Q(5, 4), Q(5, 5), are all zero. The result of this computation for Q(1, 5) is 100 because of the instant reward from R(5, 1). This result does not change the Q matrix.

Because 5 is the goal state, we finish this episode. Our agent's brain now contain updated matrix Q as:

$$
Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 100 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 80 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

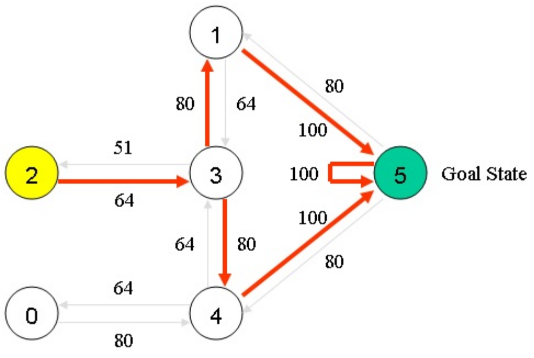
If our agent learns more through further episodes, it will finally reach convergence values in matrix Q like:

$$Q = \begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 400 & 0 \\ 1 & 0 & 0 & 0 & 320 & 0 & 500 \\ 2 & 0 & 0 & 0 & 320 & 0 & 0 \\ 3 & 0 & 400 & 256 & 0 & 400 & 0 \\ 4 & 320 & 0 & 0 & 320 & 0 & 500 \\ 5 & 0 & 400 & 0 & 0 & 400 & 500 \end{array}$$

This matrix Q, can then be normalized (i.e.; converted to percentage) by dividing all non-zero entries by the highest number (500 in this case):

$$Q = \begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 80 & 0 \\ 1 & 0 & 0 & 0 & 64 & 0 & 100 \\ 2 & 0 & 0 & 0 & 64 & 0 & 0 \\ 3 & 0 & 80 & 51 & 0 & 80 & 0 \\ 4 & 64 & 0 & 0 & 64 & 0 & 100 \\ 5 & 0 & 80 & 0 & 0 & 80 & 100 \end{array}$$

Once the matrix Q gets close enough to a state of convergence, we know our agent has learned the most optimal paths to the goal state. Tracing the best sequences of states is as simple as following the links with the highest values at each state.



For example, from initial State 2, the agent can use the matrix Q as a guide:

From State 2 the maximum Q values suggests the action to go to state 3.

From State 3 the maximum Q values suggest two alternatives: go to state 1 or 4. Suppose we arbitrarily choose to go to 1.

From State 1 the maximum Q values suggests the action to go to state 5.

Thus the sequence is 2 - 3 - 1 - 5.

## 3 Flappy Bird

Flappy Bird was a side-scrolling mobile game, the objective was to direct a flying bird, named "Faby", who moves continuously to the right, between sets of Mario-like pipes. Note that the pipes always have the same gap between them and there is no end to the running track. If the player touches the pipes, they lose. Faby briefly flaps upward each time that the player taps the screen; if the screen is not tapped, Faby falls because of gravity; each pair of pipes that he navigates between earns the player a single point, with medals awarded for the score at the end of the game. Android devices enabled the access of world leaderboards, through Google Play. You can also play this game on-line: `http://flappybird.io/`.

## 4 Tasks

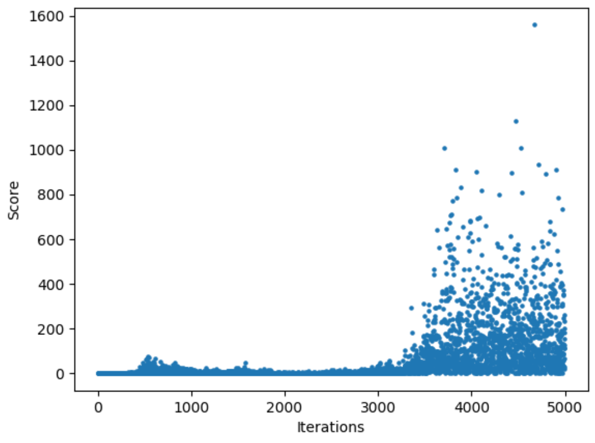
1. **Implement the algorithm in the tutorial example , and output the Q-matrix and the path with the highest values.**
2. Now here is a flappy bird project (Python3) for you, and the file `bot.py` is incomplete. You should implement a flappy bird bot who learns from each game played via Q-learning.

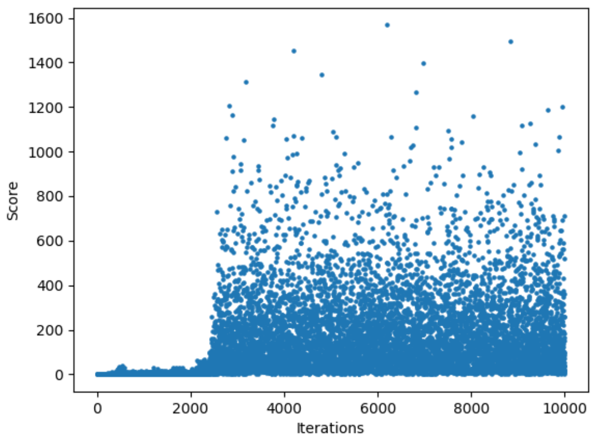
   Please pay attention to the following points:

   - The state of the bird is defined by the horizontal and vertical distances fro the next pip and the velocity of the bird.
   - In order to understand the state space, you have You need to briefly understand the following sizes: `SCREENWIDTH=288,SCREENHEIGHT=512, PIPEGAPSIZE=100, BASEY=SCREENHEIGHT*0.79, PIPE=[52,320],PLAYER=[34,24],BASE=[336,112], BACKGROUND=[288,512],etc.`
   - The Q values are dumped to the local JSON file `qvalues.json`.

- `initialize_qvalues.py` is an independent file, and we can run `python initialize_qvalues.py` to initialize the Q values. Of course, this file has been initialized.
- You can run `python learn.py --verbose 5000` to update the Q values dumped to `qvalues.json` with 5000 iterations, and then run `python flappy.py` to observe the performance the bird.

**Please complete the function `update_scores()` in `bot.py`, and run `python learn.py --verbose 5000` and `python learn.py --verbose 10000` to get the following figures, respectively,:**





3. Please submit a file named `E14_YourNumber.pdf` and send it to `ai_2020@foxmail.com`

## 5 Codes and Results

### 5.1 Codes

#### 5.1.1 Q-Learning

使用邻接表表示图，随机初始化room 0-4，运行500轮。最终展示从room 0-5起始的最优路径。

```
import random
from tqdm import tqdm


class Game(object):
    def __init__(self):
        self.__edges = ((4, ),
                        (3, 5),
                        (3, ),
```

```python
                    (1, 2, 4),
                    (0, 3, 5),
                    (1, 4, 5))
        self.__goal = 5

    def list_states(self):
        return range(len(self.__edges))

    def start(self, state=None):
        if state is None:
            self.__current = random.randint(0, 4)
        elif state >= len(self.__edges):
            return None
        else:
            self.__current = state
        return self.__current

    def current_state(self):
        return self.__current

    def list_actions(self):
        return range(len(self.__edges[self.__current]))

    def next_state(self, action):
        if action >= len(self.__edges[self.__current]):
            return None
        return self.__edges[self.__current][action]

    def reward(self, action):
        next_state = self.next_state(action)
        if next_state is None:
            return None
        if next_state == self.__goal:
            return 100
        return 0

    def run(self, action):
        next_state = self.next_state(action)
        if next_state is None:
            return None
        self.__current = next_state
        if next_state == self.__goal:
            return 100
        return 0

    def reach_goal(self):
        return self.__current == self.__goal


def greedy(actions, Q_state):
    return max(actions, key=lambda k: Q_state[k])

def epsilon_greedy(actions, Q_state, epsilon):
    if epsilon < 0 or epsilon > 1:
        return None
    if random.random() < epsilon:
        return random.choice(actions)
    return greedy(actions, Q_state)

Q = dict()
def q_learning(num_episodes, epsilon, alpha, gamma, game):
    global Q
    for episode in tqdm(range(num_episodes)):
        game.start()
        while not game.reach_goal():
            actions = game.list_actions()
            state = game.current_state()
            if state not in Q:
                Q[state] = {a: 0 for a in actions}
            action = epsilon_greedy(actions, Q[state], epsilon)
            reward = game.run(action)
            actions = game.list_actions()
            next_state = game.current_state()
            if next_state not in Q:
                Q[next_state] = {a: 0 for a in actions}
            Q[state][action] += alpha * (reward + gamma * Q[next_state][greedy(actions, Q[next_state])] - Q[state][action])


if __name__ == '__main__':
    random.seed(0)
    game = Game()
    q_learning(500, 0.01, 0.5, 1, game)
    print(Q)
    for start in game.list_states():
        game.start(start)
        while not game.reach_goal():
            actions = game.list_actions()
            state = game.current_state()
            if state not in Q:
                Q[state] = {a: 0 for a in actions}
            print(state, end='->')
            action = greedy(actions, Q[state])
            reward = game.run(action)
        print(game.current_state())
```

### 5.1.2 Flappy Bird

1. Q-Learning：不同于普通的Q-Learning，它是跑完整个episode后更新一次Q。[1]

   其中有几个细节，首先这里的history是反向的。

   ```
   history = list(reversed(self.moves))
   ```

   其次这里有一个标志表示最终死亡是因为触顶。

   ```
   # Flag if the bird died in the top pipe
   high_death_flag = True if int(history[0][2].split("_")[1]) > 120 else False
   ```

   补充代码，首先需要遍历history。

   ```
   for state, act, Next in history:
   ```

   然后注意这里的0（不动），1（振翅）两个动作的reward要根据是否导致游戏结束来给。

   对于最终导致游戏结束的，即反向history中最初的几个actions给很低的reward。

   ```
           # Select reward
           if t == 1 or t == 2:
               cur_reward = self.r[1]
   ```

   如果是触顶，则还需用很低的reward来惩罚1（振翅）动作。

   ```
           elif high_death_flag and act:
               cur_reward = self.r[1]
               high_death_flag = False
   ```

   其他动作给普通的reward即不惩罚。

```
        else:
                cur_reward = self.r[0]
```

最后更新Q。

```
        self.qvalues[state][act] += self.lr * (cur_reward + self.discount * max(
            self.qvalues[Next]) - self.qvalues[state][act])
```

2. Monte Carlo：实际上这里可以用Monte Carlo的方法。因为Q-Learning引入TD其实是用$R_{t+1}+\gamma Q_{t+1}$来近似替代$G_t = R_{t+1}+\gamma R_{t+2}+\gamma^2 R_{t+3}+\cdots$，这样的好处是不需要跑完整个episode来获得全部的reward来计算$G_t$。而这里的Q-Learning却改成了跑完一个episode再学习，摒弃了Q-Learning边跑episode边学习的优势，所以不妨用回Monte Carlo。

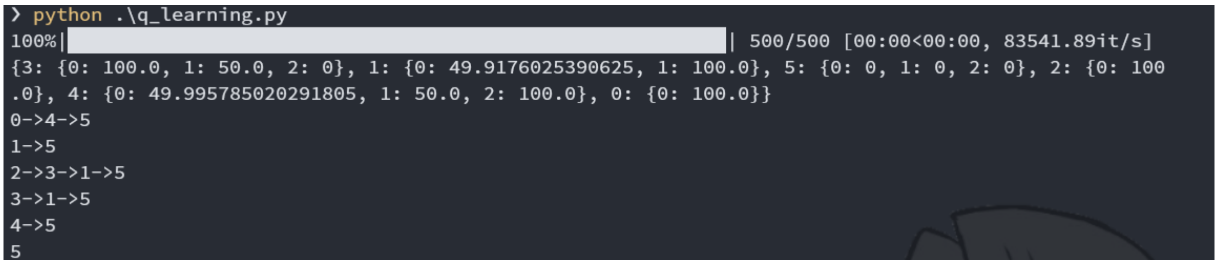
```
Gt = 0
for state, act, Next in history:
        # Select reward
        #Your code here
        if t == 1 or t == 2:
                cur_reward = self.r[1]
        elif high_death_flag and act:
                cur_reward = self.r[1]
                high_death_flag = False
        else:
                cur_reward = self.r[0]
        # Update self.qvalues[state][act]
        #Your code here
        Gt = cur_reward + self.discount * Gt
        self.qvalues[state][act] += self.lr * (Gt - self.qvalues[state][act])
        t += 1
```

## 5.2 Results

以下结果均设置了种子，可以复现。

### 5.2.1 Q-Learning

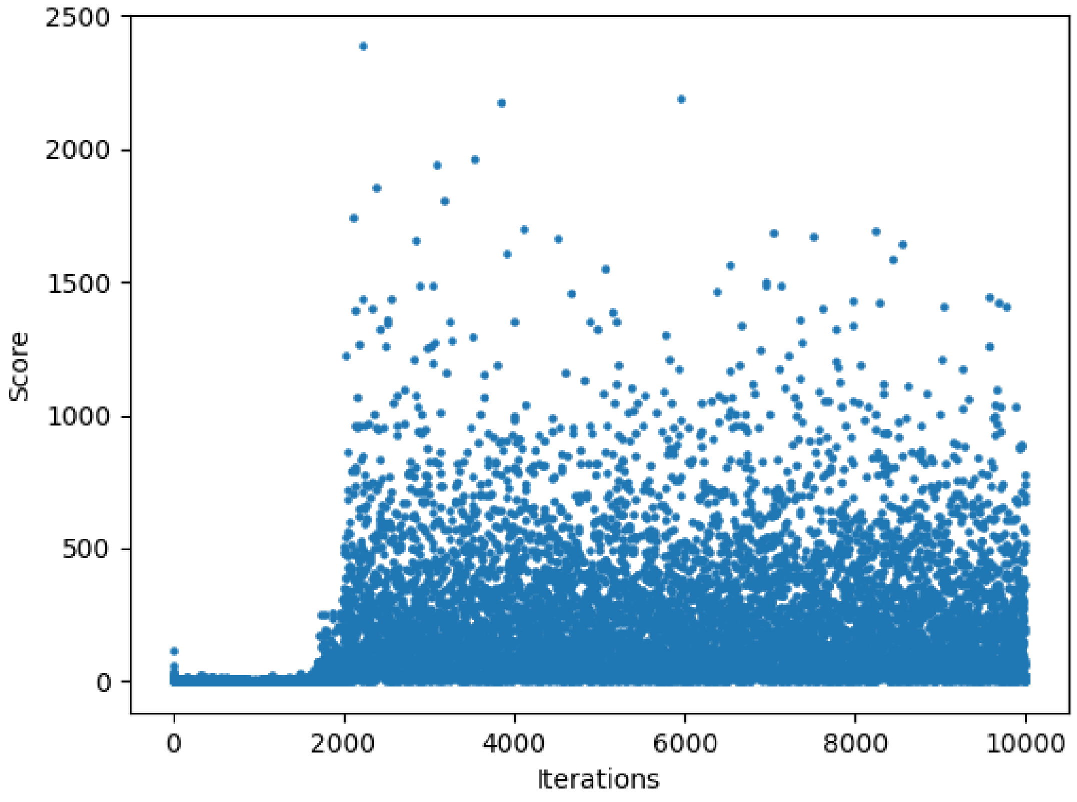
最终获得路径均是最优解。



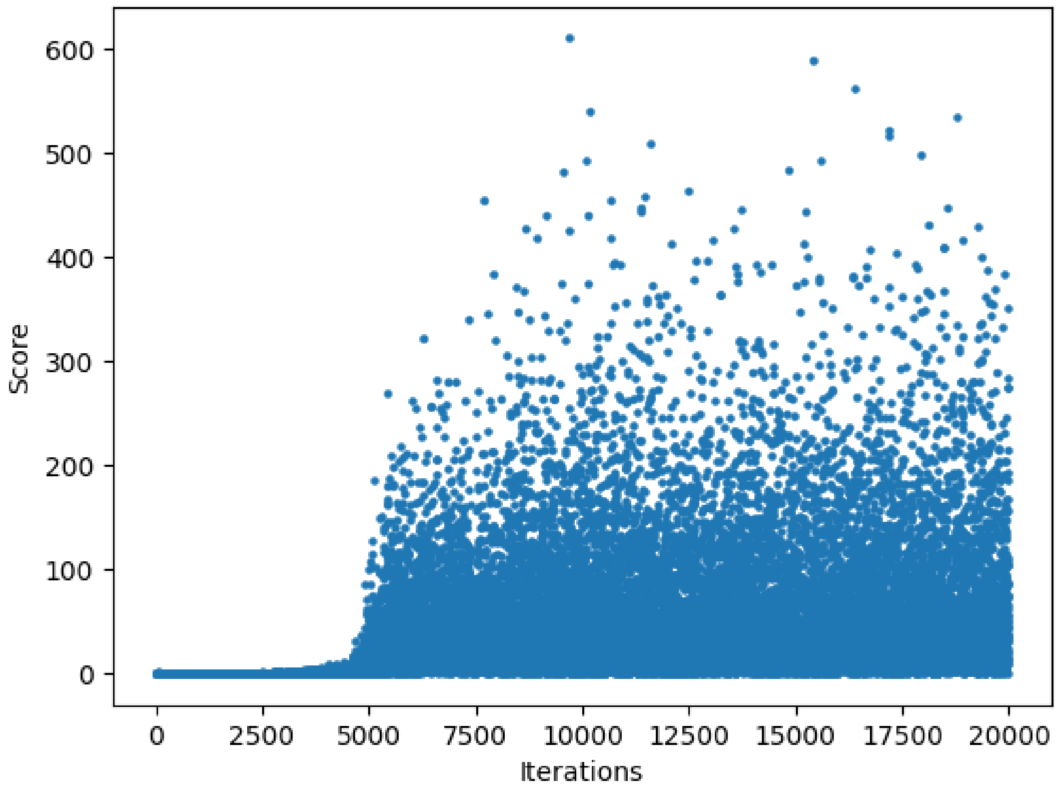
```
❯ python .\q_learning.py
100%|████████████████████████████████████████| 500/500 [00:00<00:00, 83541.89it/s]
{3: {0: 100.0, 1: 50.0, 2: 0}, 1: {0: 49.9176025390625, 1: 100.0}, 5: {0: 0, 1: 0, 2: 0}, 2: {0: 100
.0}, 4: {0: 49.995785020291805, 1: 50.0, 2: 100.0}, 0: {0: 100.0}}
0->4->5
1->5
2->3->1->5
3->1->5
4->5
5
```

### 5.2.2 Flappy Bird

Q-Learning：即使用初始给的超参数（$\gamma = 1, \alpha = 0.7$），效果仍然好。



Monte Carlo：即使调参（$\gamma = 0.7, \alpha = 0.3$），效果仍不如Q-Learning，因为鸟存活时间短，所以多训练了10000轮但仍效果不佳。



可能原因是一方面我认为是reward做的不够好（我没有针对Monte Carlo进行调整），另一方面Monte Carlo本身有高方差的问题需要更大的数据量，而TD的高bias问题可以通过设置固定LR来缓解[2]。

## References

[1] `https://github.com/chncyhn/flappybird-qlearning-bot`

[2] `https://stats.stackexchange.com/questions/336974/when-are-monte-carlo-methods-preferred-o`