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

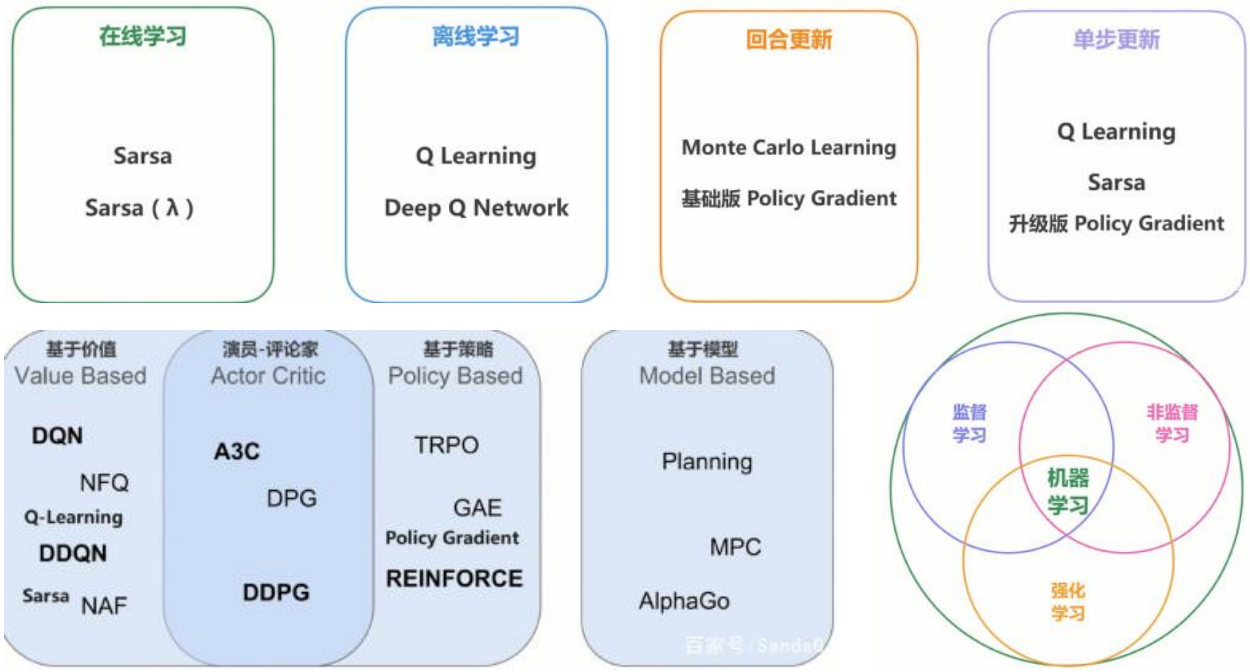
E14 Reinforcement Learning (C++/Python)

18364066 Lu Yanzuo

December 14, 2020

Contents

1 Overview



2 Tutorial

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

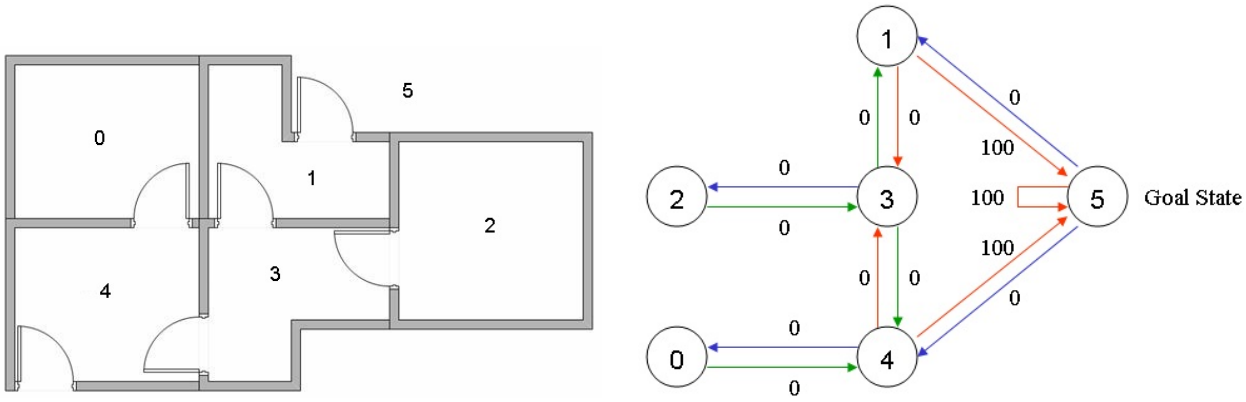
Chinese version: https://blog.csdn.net/itplus/article/details/9361915

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



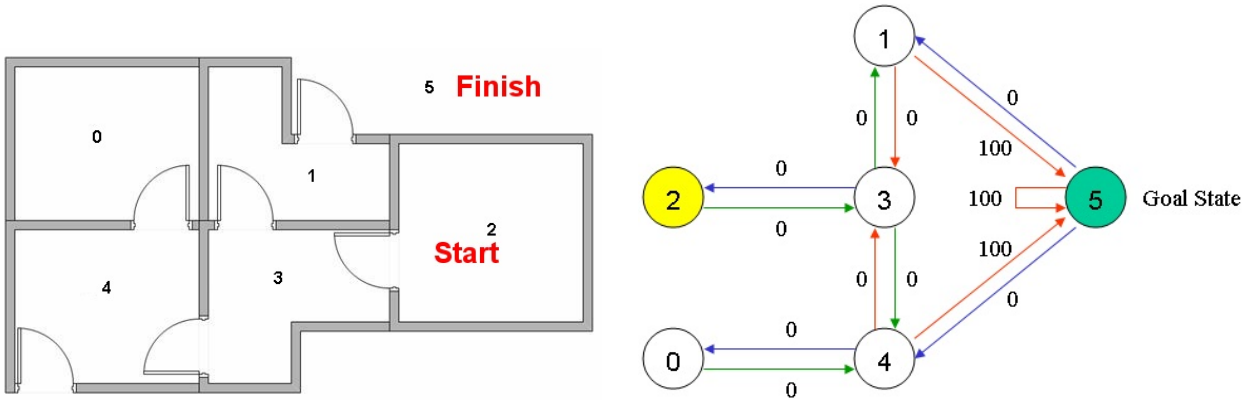
room carry a reward of 100. In Q-learning, the goal is to reach the state with the highest reward, so that if the agent arrives at the goal, it will remain there forever. This type of goal is called an ”absorbing goal”.

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”.

Now we’ll add a similar matrix, ”Q”, to the brain of our agent, representing the memory of what the agent

$$
R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{array}{c}
\begin{array}{cccccc} \multicolumn{6}{c}{\text{Action}} \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{bmatrix}
-1 & -1 & -1 & -1 & 0 & -1 \\
-1 & -1 & -1 & 0 & -1 & 100 \\
-1 & -1 & -1 & 0 & -1 & -1 \\
-1 & 0 & 0 & -1 & 0 & -1 \\
0 & -1 & -1 & 0 & -1 & 100 \\
-1 & 0 & -1 & -1 & 0 & 100
\end{bmatrix}
\end{array}
$$

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

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 γ, multiplied by the maximum value of Q for all possible actions in the next state. Here the α is a hyper-parameter similar to γ, 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:

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.

Set the gamma parameter, and environment rewards in matrix R;
Initialize matrix Q to zero;
foreach *episode* **do**
    Select a random initial state;
    while *the goal state hasn't been reached* **do**
        Select one among all possible actions for the current state;
        Using this possible action, consider going to the next state;
        Get maximum Q value for this next state based on all possible actions;
        Compute:

$$Q(state, action) = (1 - \alpha) * Q(state, action)$$

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

        ;
        Set the next state as the current state;
    end
end

Algorithm 1: The Q-Learning Algorithm

The γ parameter has a range of 0 to 1 ($0 \leq \gamma < 1$). If γ is closer to zero, the agent will tend to consider only immediate rewards. If γ 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 α, because the algorithm can convergence in this simple case without α. 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.

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

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.

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

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

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:

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

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:

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

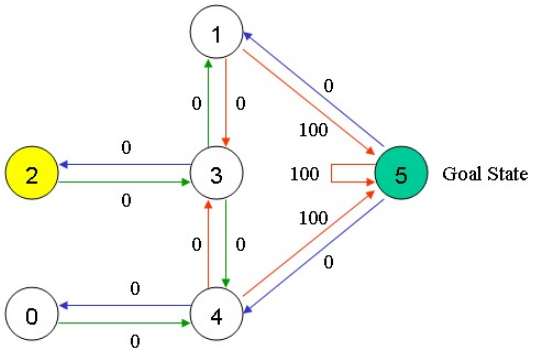
$$Q(3,1) = R(3,1) + 0.8 \times \max[Q(1,2), Q(1,5)] = 0 + 0.8 \times \max(0, 100) = 80$$

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:

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

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.

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:

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

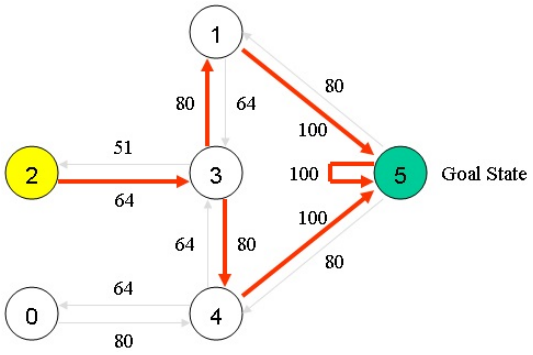
If our agent learns more through further episodes, it will finally reach convergence values in matrix Q like:

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

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):

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

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/.

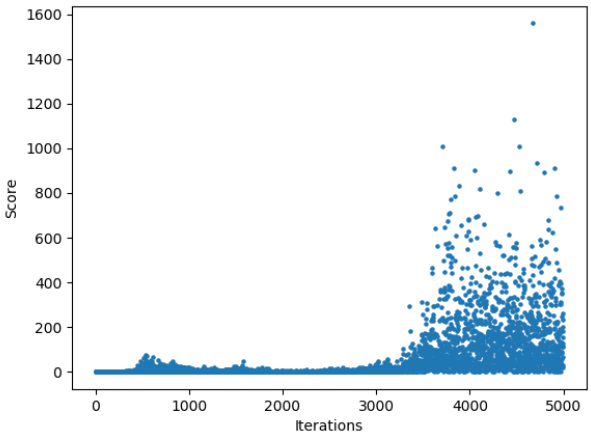
4 Tasks

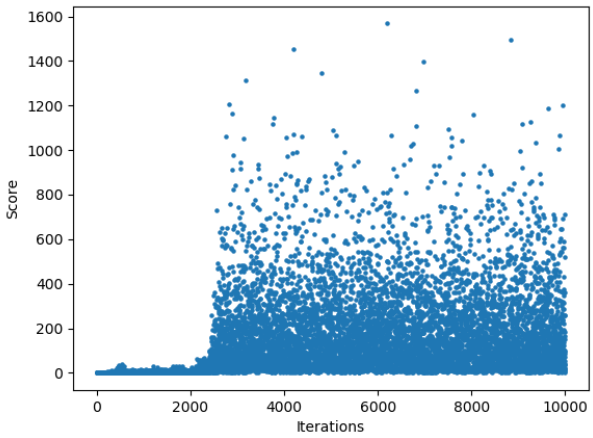
1. **Implement the algorithm in the tutorial example , and output the Q-matrix and the path with the highest values.**
2. Now here is a flappy bird project (Python3) for you, and the file `bot.py` is incomplete. You should implement a flappy bird bot who learns from each game played via Q-learning.

   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

首先是第一个任务中的 QLearning 算法实现，在 Reward 矩阵中我将不可执行的操作的值均设成了 np.nan 作为标识，此外对当前状态是否为终点进行了额外判定，若是则直接跳出循环，所以最后的 Q 矩阵的第六行会全部为零，毕竟到达该状态就会立即停止，以下是运行了 500 个迭代、alpha 和 gamma 值分别设置为 0.5 和 0.9 的 Q 矩阵以及每个状态的最高 Q 值对应的下一个状态，该结果和本文档给出的基本相同，只是我在逻辑上使得无意义的第六行置为全零。

```
State-Action Q Table:
[[  0.    0.    0.    0.   90.    0. ]
 [  0.    0.    0.   81.    0.  100. ]
 [  0.    0.    0.   81.    0.    0. ]
 [  0.   90.   72.9   0.   90.    0. ]
 [ 81.    0.    0.   81.    0.  100. ]
 [  0.    0.    0.    0.    0.    0. ]]
highest value for state 0:
        [4]
highest value for state 1:
        [5]
highest value for state 2:
        [3]
highest value for state 3:
        [1 4]
highest value for state 4:
        [5]
```

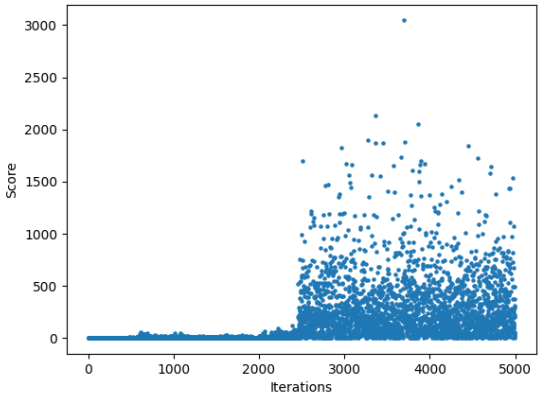
```
import numpy as np

# State-Action R Table
R = np.array([
    [np.nan, np.nan, np.nan, np.nan, 0, np.nan],
    [np.nan, np.nan, np.nan, 0, np.nan, 100],
    [np.nan, np.nan, np.nan, 0, np.nan, np.nan],
    [np.nan, 0, 0, np.nan, 0, np.nan],
    [0, np.nan, np.nan, 0, np.nan, 100],
    [np.nan, 0, np.nan, np.nan, 0, 100]
])

# State-Action Q Table
Q = np.zeros((6, 6))

def getValidAction(position):
    action_list = []
    R_position = R[position]
    for index in range(len(R_position)):
        if not np.isnan(R_position[index]):
            action_list.append(index)
    return np.array(action_list)

def getNextActionWithRandomChoice(position, returnMax=False):
    valid_action = getValidAction(position)
    if len(valid_action) == 0:
        return -1
    random_float = np.random.rand()
    if not returnMax:
        # randomly choose action from valid_action
        return valid_action[np.random.randint(0, len(valid_action))]
    else:
```

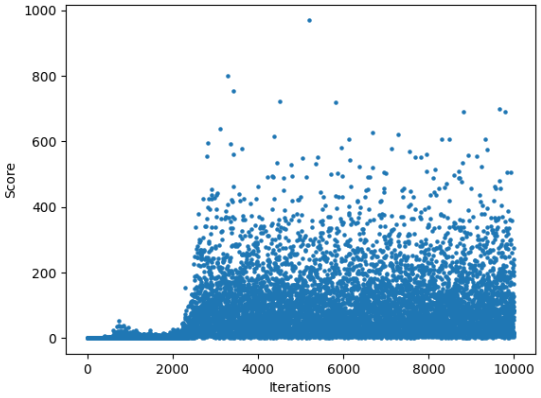
```
        # choose a* action form Q
        return valid_action[np.argmax(Q[position][valid_action])]

def QLearning(maxIteration=500, alpha=0.5, gamma=0.9):
    for iterIndex in range(maxIteration):
        state = np.random.randint(0, 6)
        while state != 5:
            nextState = getNextActionWithRandomChoice(state)
            nextAction = getNextActionWithRandomChoice(nextState,
               returnMax=True)
            Q[state][nextState] += alpha * (R[state][nextState] + gamma
               * Q[nextState][nextAction] - Q[state][nextState])
            state = nextState

if __name__ == '__main__':
    QLearning()

    print("State-Action Q Table:")
    print(Q)

    for i in range(5):
        print("highest value for state {}: ".format(i))
        print("\t", np.where(Q[i] == Q[i][np.argmax(Q[i])])[0])
```

然后是 flappy bird 游戏中的 Q 值更新函数，这里需要注意的点有两个，一个是 history 是 reversed 也即逆转的，我们在迭代过程中遇到的第一个动作对应的是 crash 碰撞也就是游戏结束，因此要将这一步的 reward 设置为-1000，事实上如果我们想要往前再想一步，也可以将倒数第二个动作的 reward 也置为-1000，毕竟我们在玩这个游戏的时候也能切实地感受到，其实当小鸟快要撞上的时候，就算往前一步也无法挽回了。

另一个点是为了防止小鸟频繁的 flap 往上飞，以至于动作过多撞上管道的上方，根据代码中 high_death_flag 变量的提示，我们可以对其进行惩罚，惩罚的方式是若本次 history 确实是撞上了管道的上方，就将第一次 flap 也就是 action=1 的动作 Reward 设置为-1000。

下面是 5000 步和 10000 步的两次分别运行结果，这里的结果在运行前都对文件夹内的 Q 值 json 进行了初始化，可以看到和本文档给出的结果也是出入不大的。






```
# Q-learning score updates
t = 1
#You code here
for state, action, next_state in history:
    # Select reward
    #Your code here
    if t == 1:
        reward = self.r[1]
    elif high_death_flag and action == 1:
        reward = self.r[1]
        high_death_flag = False
    else:
        reward = self.r[0]
    #Your code here
```

```
    self.qvalues[state][action] += self.lr * (reward + self.discount *
        max(self.qvalues[next_state]) - self.qvalues[state][action])
    t += 1
```