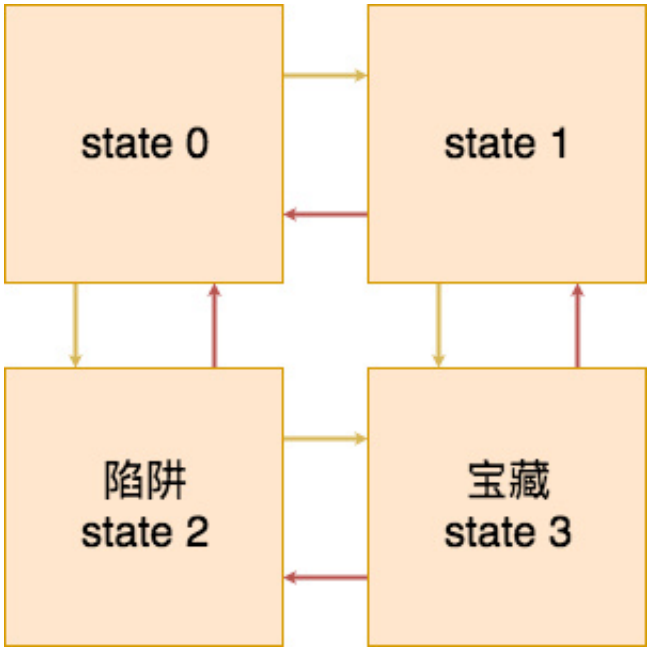
Q Learning 介绍

我们知道了 q-learning 最重要的状态转移公式，这个公式也叫做 Bellman Equation，通过这个公式我们能够不断地进行更新 Q 矩阵，最后得到一个收敛的 Q 矩阵。

下面我们通过代码来实现这个过程

我们定义一个简单的走迷宫过程，也就是



初始位置随机在 state 0， state 1 和 state 2 上，然后希望智能体能够走到 state 3 获得宝藏，上面可行的行动路线已经用箭头标注了

```
import numpy as np
import random
```

下面定义奖励矩阵，一共是 4 行，5 列，每一行分别表示 state 0 到 state 3 这四个状态，每一列分别表示上下左右和静止 5 种状态，奖励矩阵中的 0 表示不可行的路线，比如第一个行，上走和左走都是不可行的路线，都用 0 表示，向下走会走到陷阱，所以使用 -10 表示奖励，向右走和静止都给与 -1 的奖励，因为既没有触发陷阱，也没有到达宝藏，但是过程中浪费了时间。

```
reward = np.array([[0, -10, 0, -1, -1],
                   [0, 10, -1, 0, -1],
                   [-1, 0, 0, 10, -10],
                   [-1, 0, -10, 0, 10]])
```

接下来定义一个初始化为 0 的 q 矩阵

```
q_matrix = np.zeros((4, 5))
```

然后定义一个转移矩阵，也就是从一个状态，采取一个可行的动作之后到达的状态，因为这里的状态和动作都是有限的，所以我们可以将他们存下来，比如第一行表示 state 0，向上和向左都是不可行的路线，所以给 -1 的值表示，向下走到达了 state 2，所以第二个值为 2，向右走到达了 state 1，所以第四个值是 1，保持不同还是在 state 0，所以最后一个标注为 0，另外几行类似。

```
transition_matrix = np.array([[-1, 2, -1, 1, 0],
                              [-1, 3, 0, -1, 1],
                              [0, -1, -1, 3, 2],
                              [1, -1, 2, -1, 3]])
```

最后定义每个状态的有效行动，比如 state 0 的有效行动就是下、右和静止，对应于 1，3 和 4

```
valid_actions = np.array([[1, 3, 4],
                          [1, 2, 4],
                          [0, 3, 4],
                          [0, 2, 4]])
```

```
# 定义 bellman equation 中的 gamma
gamma = 0.8
```

最后开始让智能体与环境交互，不断地使用 bellman 方程来更新 q 矩阵，我们跑 10 个 episode

```
for i in range(10):
    start_state = np.random.choice([0, 1, 2], size=1)[0] # 随机初始起点
    current_state = start_state
    while current_state != 3: # 判断是否到达终点
        action = random.choice(valid_actions[current_state]) # greedy 随机选择当前状态下的有效动作
        next_state = transition_matrix[current_state][action] # 通过选择的动作得到下一个状态
        future_rewards = []
        for action_nxt in valid_actions[next_state]:
            future_rewards.append(q_matrix[next_state][action_nxt]) # 得到下一个状态所有可能动作的奖励
        q_state = reward[current_state][action] + gamma * max(future_rewards) # bellman equation
        q_matrix[current_state][action] = q_state # 更新 q 矩阵
        current_state = next_state # 将下一个状态变成当前状态

    print('episode: {}, q matrix: \n{}'.format(i, q_matrix))
```

```
    print()
```

```
episode: 0, q matrix:
[[  0.   0.   0.  -1.  -1.]
 [  0.  10.  -1.   0.  -1.]
 [  0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.]]

episode: 1, q matrix:
[[  0.   0.   0.  -1.  -1.]
 [  0.  10.  -1.   0.  -1.]
 [  0.   0.   0.  10.   0.]
 [  0.   0.   0.   0.   0.]]

episode: 2, q matrix:
[[  0.   -2.    0.    7.    4.6]
 [  0.   10.    4.6   0.    7. ]
 [ -1.8   0.    0.   10.   -2. ]
 [  0.    0.    0.    0.    0. ]]

episode: 3, q matrix:
[[  0.   -2.    0.    7.    4.6]
 [  0.   10.    4.6   0.    7. ]
 [  4.6   0.    0.   10.   -2. ]
 [  0.    0.    0.    0.    0. ]]

episode: 4, q matrix:
[[  0.   -2.    0.    7.    4.6]
 [  0.   10.    4.6   0.    7. ]
 [  4.6   0.    0.   10.   -2. ]
 [  0.    0.    0.    0.    0. ]]

episode: 5, q matrix:
[[  0.   -2.    0.    7.    4.6]
 [  0.   10.    4.6   0.    7. ]
 [  4.6   0.    0.   10.   -2. ]
 [  0.    0.    0.    0.    0. ]]

episode: 6, q matrix:
[[  0.   -2.    0.    7.    4.6]
 [  0.   10.    4.6   0.    7. ]
 [  4.6   0.    0.   10.   -2. ]
 [  0.    0.    0.    0.    0. ]]

episode: 7, q matrix:
```

```
[[  0.   -2.    0.    7.    4.6]
 [  0.   10.    4.6   0.    7. ]
 [  4.6   0.    0.   10.   -2. ]
 [  0.    0.    0.    0.    0. ]]

episode: 8, q matrix:
[[  0.   -2.    0.    7.    4.6]
 [  0.   10.    4.6   0.    7. ]
 [  4.6   0.    0.   10.   -2. ]
 [  0.    0.    0.    0.    0. ]]

episode: 9, q matrix:
[[  0.   -2.    0.    7.    4.6]
 [  0.   10.    4.6   0.    7. ]
 [  4.6   0.    0.   10.   -2. ]
 [  0.    0.    0.    0.    0. ]]
```

可以看到在第一次 episode 之后，智能体就学会了在 state 2 的时候向下走能够得到奖励，通过不断地学习，在 10 个 episode 之后，智能体知道，在 state 0，向右走能得到奖励，在 state 1 向下走能够得到奖励，在 state 3 向右 走能得到奖励，这样在这个环境中任何一个状态智能体都能够知道如何才能够最快地到达宝藏的位置

从上面的例子我们简单的演示了 q-learning，可以看出自己来构建整个环境是非常麻烦的，所以我们可以通过一些第三方库来帮我们搭建强化学习的环境，其中最有名的就是 open-ai 的 gym 模块，下一章我们将介绍一下 gym。