## 机器人路径规划

### 1 实验任务

·设计一个有障碍物的地图，用户可以修改障碍物布局，可以指定起点和终点

·编程实现Q-learning算法，用于机器人规划最短路径，学习算法参数可以由用户设置

·使用可视化界面演示Q值变化过程及最短路径探测过程

### 2 实验原理

#### 2.1 强化学习

**2.1.1 基本概念**

强化学习（Reinforcement learning，RL）讨论的问题是一个智能体(agent) 怎么在一个复杂不确定的环境(environment) 里面去极大化它能获得的奖励。通过感知所处环境的状态(state)对动作(action)的反应(reward)，来指导更好的动作，从而获得最大的收益(return)，这被称为在交互中学习，这样的学习方法就被称作强化学习。强化学习的几个基本要素如图9所示

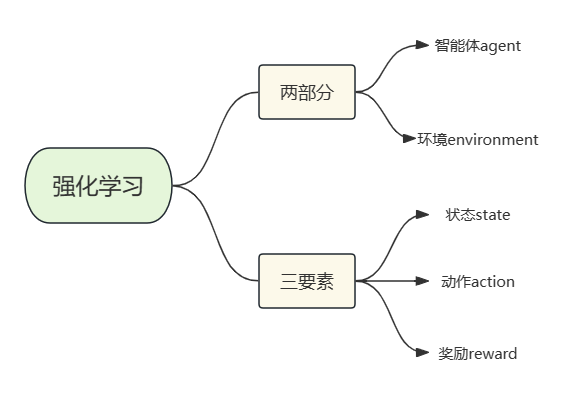


图9

如图10所示，在强化学习过程中，智能体跟环境一直在交互。智能体在环境里面获取到状态，智能体会利用这个状态输出一个动作，一个决策。然后这个决策会放到环境之中去，环境会根据智能体采取的决策，输出下一个状态以及当前的这个决策得到的奖励。智能体的目的就是为了尽可能多地从环境中获取奖励。

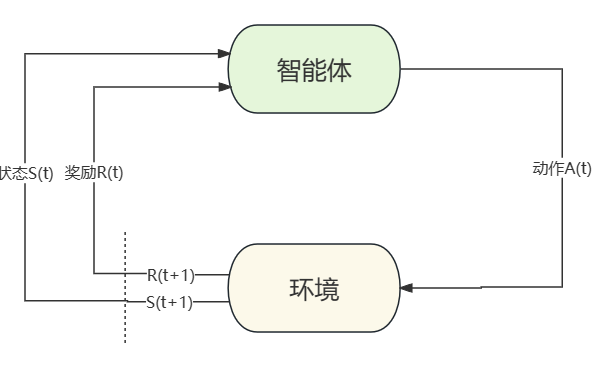
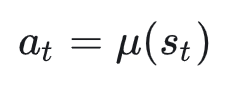


图10

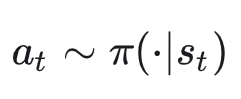
**2.1.2 相关术语**

**策略(Policy)**

策略是智能体用于决定下一步执行什么行动的规则。可以是确定性的，一般表示为µ：

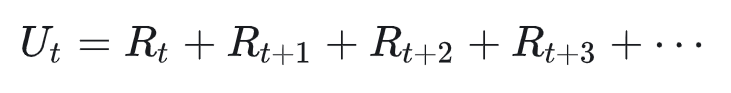


也可以是随机的，一般表示为π：



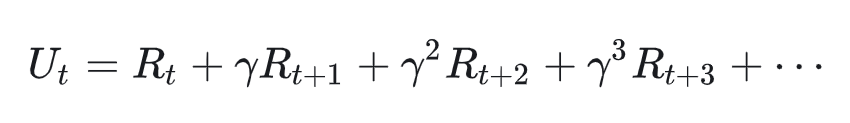
**回报(Return)**

回报又称未来累计奖励，一般表示为U：



未来的奖励不如现在等值的奖励那么好，的权重应该小于，因此强化学习通常用discounted return（折扣回报），取γ为discount rate（折扣率），

0 < γ ≤ 1 ，则有



**价值函数(Value Function)**

价值函数使用期望对未来的收益进行预测，一方面不必等待未来的收益实际发生就可以获知当前状态的好坏，另一方面通过期望汇总了未来各种可能的收益情况。使用价值函数可以很方便地评价不同策略的好坏。

*状态价值函数*：用来度量给定策略的情况下，当前状态的好坏程度

动作价值函数：用来度量给定状态和策略的情况下，采用动作的好坏程度

#### 2.2 Q-learning算法

**2.2.1 算法思想**

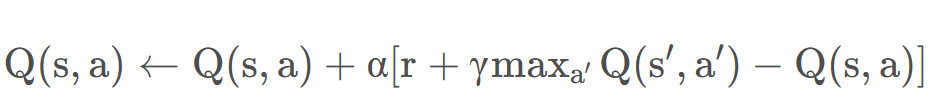
强化学习要解决的一个关键问题便是，在当前状态下，该采取怎样的动作？这个问题的答案是简单的，就是选择所有动作中选择使Q(, \*)值最大的动作。那么又怎样求Q(, )呢？而这就是Q-learning算法要解决的问题。

算法的主要思想就是将State与Action构建成一张如表1所示的Q-Table来存储Q值，然后根据Q值来选取能够获得最大的收益的动作。

|  |  |  |  |
| --- | --- | --- | --- |
| **Q-Table** |  |  | ... |
|  | Q(, ) | Q(, ) | ... |
|  | Q(, ) | Q(, ) | ... |
| ... | ... | ... | ... |

表1 Q-Table

构建并初始化Q-Table后，我们就需要对表进行迭代更新直至满足所设置的条件为止，而迭代的依据便是如下著名的贝尔曼公式：



**2.2.2 算法步骤**

1）初始化Q值表

2）在当前状态s下，使用ε-greedy策略选择动作a

3）执行动作a，得到奖励r和下一状态,

4)更新Q(s, a)，状态更新为,

5)重复步骤2-4至满足条件（如达到最大迭代次数）

### 3 程序设计

#### 3.1 文件描述

**maze.py**：迷宫类

**draw.py**：迷宫可视化

·draw\_maze(ax, my\_maze, label)：绘制迷宫

·draw\_track(ax, my\_maze, step)：绘制智能体运动轨迹

·draw\_qtable(ax, my\_maze, Q\_table, step)：绘制Q-Table变化

**main.py**：主程序，求解路径规划

·q\_value(self, state)：求Q(s, \*)

·predict(self, state)：贪心策略求a = maxQ(s, \*)

·train(self, ax1, ax2)：迭代更新Q-Table

#### 3.2 关键代码

***迭代更新Q-Table***

def train(self, ax1, ax2):

win\_history = []

win\_rate = 0.0

for epoch in range(epoch\_num):

self.my\_maze.reset((0, 0))

game\_over = False

state = (0, 0)

step = 0

while not game\_over:

valid\_actions = self.my\_maze.valid\_actions()

if not valid\_actions: break

# epsilon-greedy

if np.random.rand() < self.epsilon:

action = random.choice(valid\_actions)

else:

action = self.predict(state)

# 实施action

state\_next, reward, game\_status = self.my\_maze.act(action)

# 更新Q\_table

if (state, action) not in self.Q\_table.keys():

self.Q\_table[(state, action)] = 0.0

max\_next\_Q = max(self.q\_value(state\_next))

self.Q\_table[(state, action)] += self.lr \* (reward + self.gamma \* max\_next\_Q - self.Q\_table[(state, action)])

if game\_status == 'win':

win\_history.append(1)

game\_over = True

elif game\_status == 'lose':

win\_history.append(0)

game\_over = True

else:

game\_over = False

state = state\_next

step += 1

# 数据可视化

if (epoch + 1) % 20 == 0:

if action == 0:

a = 'RIGHT'

elif action == 1:

a = 'UP'

elif action == 2:

a = 'LEFT'

else:

a = 'DOWN'

print("epoch: %3d step: %3d action: %7s score: %8.3f status: %6s" % (epoch+1, step, a, self.my\_maze.score, game\_status))

draw\_qtable(ax1, self.my\_maze, self.Q\_table, step)

draw\_track(ax2, self.my\_maze, step)

plt.pause(0.01)

### 

### 4 实验结果与分析

#### 4.1 运行示例

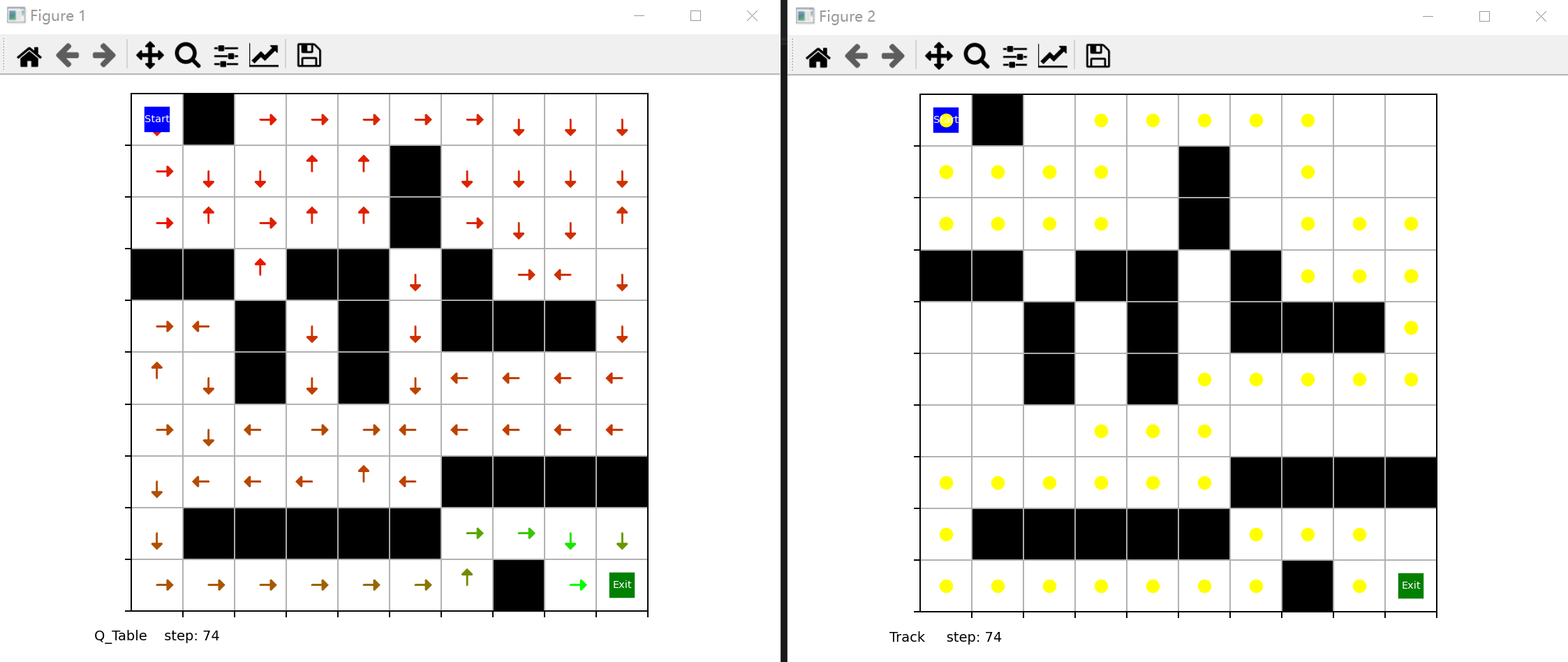
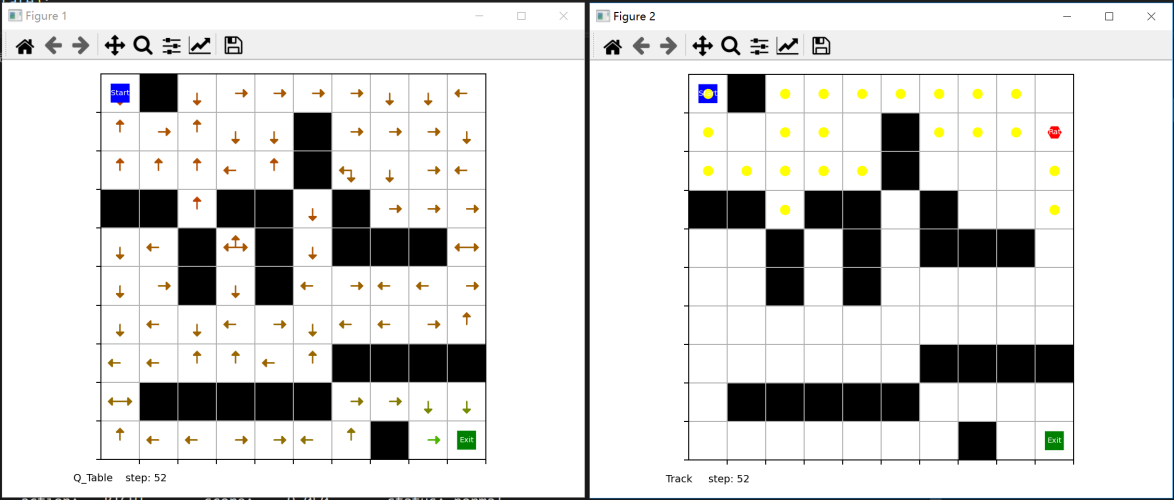
如图11所示，Figure1显示的是Q-Table的变化过程，箭头指向Q值最大的运动方向，Figure2显示的是智能体的运动轨迹。 

图11

#### 4.2 结果分析

本次实验中，我实现了强化学习的Q-Learning算法，并将其应用到迷宫的路径规划当中，效果还算理想。在这个过程中，我对强化学习的概念有了更清晰的认识，对其理解也更加深入。

实验的唯一不足之处在于，虽然最后智能体能在迷宫中找到终点，但走的却不是最短路径，中途仍有错误的步伐，我想这与奖励与惩罚分数的设计，以及算法选择的策略和迭代次数有关，这些都是我后续研究的重点。