

## Abstract

RL algorithms have kept making progress in recent decades: AlphaGo, developed by DeepMind in 2016, used RL algorithms and defeated the world's Go masters. At this point, RL has renewed the attention of scholars and is more widely used in the field of robot control. Major technology companies now are giving the highest priority to the development of RL. It can be said that RL is affecting and changing the world.

The shortest path problem has been studied for a long time. It leads the agent to find the shortest path from the starting point to end point in a known or unknown environment, while the traditional shortest path algorithms are powerless to solve the dynamic and unknown environment. Therefore, this paper is based on maze environment to explore the artificial intelligence path-finding method of RL.

RL algorithms implemented in this paper are divided into tabular RL (Q-Learning algorithm, SARSA algorithm) and deep RL (Deep Q Network algorithm). The experimental environment is based on OpenAI gym's reinforcement learning framework to perform maze simulation and test RL algorithm.

Through experiments, the paper optimizes and evaluates the parameters of the tabular RL algorithm. It also gives its own thoughts on the application scope of the maze deep RL algorithm, and proposes a new idea of RL pathfinding in maze. Finally, a brief comparison with the traditional pathfinding algorithm is made, affirming the feasibility of the intensive learning maze pathfinding.

**Keywords:** Reinforcement Learning; Shortest Path; Path Planning; Neural Network

# 目 录

摘 要.....	I
Abstract.....	II
目 录.....	III
第一章 绪论.....	1
1.1 引言.....	1
1.2 研究背景.....	1
1.3 研究现状.....	2
1.4 论文结构.....	3
第二章 基础知识及相关知识介绍.....	4
2.1 强化学习.....	4
2.1.1 求解强化学习问题.....	4
2.1.2 马尔科夫性质（Markov Property）与马尔科夫过程（MP）.....	5
2.1.3 马尔科夫奖赏过程（MRP）与马尔科夫决策过程（MDP）.....	6
2.1.4 贝尔曼方程与最优贝尔曼方程.....	10
2.1.5 蒙特卡罗强化学习和时间差分强化学习.....	12
2.1.6 $\epsilon$ -贪婪策略 .....	14
2.2 OpenAI Gym.....	14
2.2.1 Gym 开源环境库.....	15
2.2.2 Gym 环境工作机制.....	15
2.2.3 自定义 Gym 环境.....	17
2.3 神经网络与强化学习.....	18
2.3.1 近似价值函数的建立和求解.....	19
2.3.2 目标函数和梯度下降.....	20
2.4 本章小结.....	21
第三章 自动寻路系统设计.....	22
3.1 设计思路.....	22
3.2 自动寻路算法设计.....	22
3.2.1 SARSA 算法.....	23
3.2.2 Q-Learning 算法.....	25
3.2.3 Deep Q Network 算法 .....	28
3.3 迷宫环境设计.....	33
3.3.1 迷宫生成算法.....	35
3.3.2 迷宫环境绘制.....	36
3.3.3 重写 gym 方法 .....	37
3.4 本章小结.....	40
第四章 实验结果思考.....	41
4.1 参数对 Q-Learning 算法和 SARSA 算法收敛速度的影响 .....	41
4.1.1 学习率 $\alpha$ .....	42
4.1.2 折扣因子 $\gamma$ .....	43

4.1.3	奖赏函数设置.....	45
4.1.4	Q 表初始值.....	48
4.2	超参数对 DQN 算法收敛速度的影响.....	51
4.2.1	学习率 learning_rate .....	51
4.2.2	折扣因子 $\gamma$ .....	53
4.2.3	奖赏函数设置.....	54
4.2.4	TargetNet 更新步数 .....	56
4.3	迷宫强化学习迷宫寻路的思考.....	57
4.3.1	强化学习迷宫寻路算法的表现.....	57
4.3.2	强化学习迷宫寻路与其他迷宫寻路算法.....	60
4.4	本章小结.....	60
结论.....		62
论文工作总结.....		62
参考文献.....		63
致谢.....		64

## 1.4 论文结构

本文共有四章。第一章叙述强化学习、最短路径问题的研究背景、研究现状和意义；第二章从最底层的马尔科夫决策开始，介绍强化学习的背景知识，为算法理解打下基础；第三章首先介绍基于 OpenAI gym 搭建的迷宫寻路仿真系统的架构设计，其次是三种强化学习寻路算法的原理和实现；第四章是仿真实验流程和结果分析，对三种算法的参数进行介绍、调优、验证。最后，总结研究内容并指出不足之处。

## 第二章 基础知识及相关知识介绍

### 2.1 强化学习

强化学习与其他机器学习方式不同，不是聚类或分类，它的目标就是找最优策略，最大化累积奖赏。在序列状态中采取恰当的多步决策达到这一目标，所以强化学习是一种序列多步决策问题。

用种花例子<sup>[11]</sup>（周志华，机器学习）：种花要经过选购花苗、松土浇水、施肥除草、防鸟杀虫许多操作之后才能得到一朵健康美丽的花。但是，种出来的花品相究竟如何，只有最后花开了才知道。也就是说，种花过程中执行某个操作时，并不是立即知道这个操作能不能培育出一朵美丽的花，只有一个延时反馈：如果浇水的第二天花更健壮就是好的，花蔫了就是坏的。只有多次种花，不断摸索才能总结到种好花的策略，摸索策略的过程就是强化学习的过程。强化学习学习目标是找最优策略，学习过程要和环境交互，从环境的反馈学习。

强化学习过程和人类与环境的交互方式类似，是一套解决各种各样人工智能问题的通用框架。图 2-1 示意强化学习的基本框架：智能体观察当前状态，依策略选择操作，获得环境的奖赏进入新状态。从起始状态到目标状态的每次操作都可以得到奖赏，奖赏的累积叫做回报，学习的最终的目标就是找到能够最大化回报的策略。

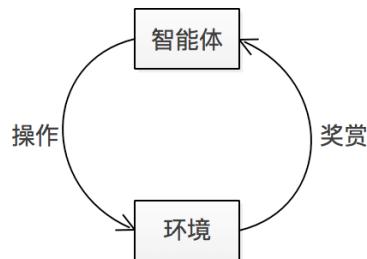


图 2-1 智能体与环境交互框架

#### 2.1.1 求解强化学习问题

求解强化学习问题有两种思路：

- a. 直接寻找最优策略：找到一个最优策略，操作选择时依照这个策略选择操作，就能最大化积累奖赏；
- b. 寻找最优价值函数：价值函数可以每一个状态、操作进行评估，最优价值函数计算出到达任一状态、采取任一操作可以能得到的最大化积累奖赏。操作选择时采取价值函数值最大的操作，也能得最大积累奖赏。

因为直接求最优策略有难度，于是依照思路 b，科学家们提出了解决大部分强化学习问题的框架——马尔科夫决策过程（MDP, Markov Decision Process）。后文遵循马尔

科夫性质（Markov Property）与马尔科夫过程（MP），马尔科夫奖赏过程（MRP）与马尔科夫决策过程（MDP），贝尔曼方程（Bellman Equation）与最优贝尔曼方程（Optimal Bellman Equation）的发展顺序，渐进地介绍强化学习问题的求解理论基础。

### 2.1.2 马尔科夫性质（Markov Property）与马尔科夫过程（MP）

**马尔科夫性质：**当一个随机过程在给定现在状态及所有过去状态情况下，其未来状态的条件概率分布仅依赖于当前状态，那么此随机过程即具有马尔科夫性质。即随机过程中，所有状态的下一状态只与当前状态有关，而与过去状态无关。

**【定义一】**随机过程具有马尔科夫性质，那么对随机过程中的每一个状态  $s_t$  有：

$$P[s_{t+1}|s_t] = P[s_{t+1}|s_1, \dots, s_t] \quad (2-1)$$

**马尔科夫过程：**具有马尔科夫性质的随机过程为马尔科夫过程。马尔科夫链、布朗运动都是有名的马尔科夫过程。如果严格按照状态和时间参数是否连续或者离散，马尔科夫过程分为三种，但通常意义上的马尔科夫过程均指马尔科夫链（Markov Chain）：

- a. 时间、状态都离散：为马尔科夫链；
- b. 时间、状态都连续：为马尔科夫过程；
- c. 时间连续、状态离散：连续时间的马尔科夫链。

**【定义二】**马尔科夫过程可表示为元组  $\langle S, P \rangle$

- $S$ ：有限状态集。 $S = \{S_1, S_2, \dots, S_n\}$ ；
- $P$ ：状态转移概率矩阵。矩阵中  $P_{ij}$  代表从状态  $S_i$  转移至状态  $S_j$  的概率。

$$P = \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{bmatrix}$$

图 2-2 状态转移概率矩阵  $P$

对马尔科夫性质、马尔科夫过程做说明：图 2-3 为一个马尔科夫过程，空心圆框代表状态，方框代表终止状态，状态集  $S=\{\text{Class1}, \text{Class2}, \text{Class3}, \text{Homework}, \text{Paper}, \text{Pass}, \text{Sleep}\}$ 。状态间的箭头上标注状态转移概率，根据图 2-3 可以写出状态转移概率矩阵  $P$ 。

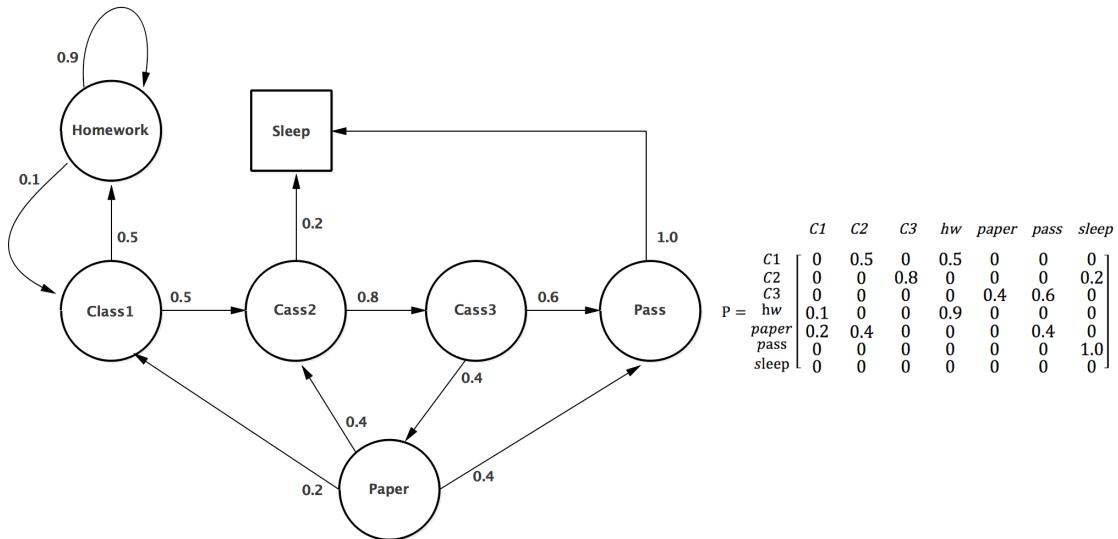


图 2-3 马尔科夫过程与对应的状态转移概率矩阵

当给定状态转移概率矩阵，从某个状态出发存在多个可能的状态序列，马尔科夫过程规定所有状态序列有穷，最后必须到达终止状态。例如从 Class1 开始，可能产生的状态序列有：

- a. Class1 → Class2 → Class3 → Pass → Sleep
- b. Class1 → Homework → Class1 → Class2 → Sleep
- c. .....

马尔科夫过程不足以描述强化学习问题的特点，在强化学习问题中智能体还要通过操作与环境进行交互，获得奖赏。为马尔科夫过程加入奖赏和操作的概念，有了只考虑奖赏的马尔科夫奖赏过程(MRP)、同时考虑奖赏和操作的为马尔科夫决策过程(MDP)。

### 2.1.3 马尔科夫奖赏过程 (MRP) 与马尔科夫决策过程 (MDP)

马尔科夫奖赏过程：在 MP 的基础加入奖赏，得马尔科夫奖赏过程。引入奖赏的概念后，就可以比较不同状态序列的回报。

【定义三】马尔科夫奖赏过程可表示为元组  $\langle S, P, R, \gamma \rangle$

- $S$ : 有限状态集。 $S = \{S_1, S_2, \dots, S_n\}$ ；
- $P$ : 状态转移概率。 $P_{ij}$  代表从状态  $S_i$  转移到状态  $S_j$  的概率；
- $R$ : 奖赏函数。在时间步  $t$ ，从状态  $S_t = s$  进行转移所能得到的即时奖赏的期望；

$$R_s = E [ R_{t+1} | S_t = s ] \quad (2-2)$$

- $G_t$ : 回报。从状态  $S_t$  开始，每次状态转移获得的所有即时奖赏的总和，也是强化学习问题关心的积累奖赏；

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (2-3)$$

- $\gamma$ : 折扣因子。 $\gamma \in [0, 1]$ ，表示对即时奖赏的关注程度；

若  $\gamma=0$ ，只考虑本次状态转移即时奖赏，不考虑未来状态转移奖赏；若  $\gamma=1$ ，同时考虑本次状态转移即时奖赏和未来状态转移即时奖赏，考虑的程度一样重要。 $\gamma$  越接近 0，未来状态转移即时奖赏贬值越快，反之亦然。

折扣因子的存在，首先可在数学证明上保证  $G_t$  存在、收敛；其次隐含了人类对世界的认知规律：对即时奖赏的偏好，选择确定的即时奖赏而不是等待将来可能获得的更大奖赏更保守；最后， $\gamma$  的存在可人为地为未来的即时奖赏增加“贬值”属性，增强模型的表现力。

- $v(s)$ : 状态价值函数。定义状态价值为时间步  $t$  从状态  $S_t=s$  开始，回报的期望。状态价值函数直观地衡量状态  $s$  的价值。

$$v(s) = E [ G_t | S_t = s ] \quad (2-4)$$

对回报  $G_t$ 、状态价值函数  $v(s)$  的计算做说明：图 2-4 为一个马尔科夫奖赏过程，在图 2-3 的基础上增加了使用红色标注的奖赏  $R$ 。

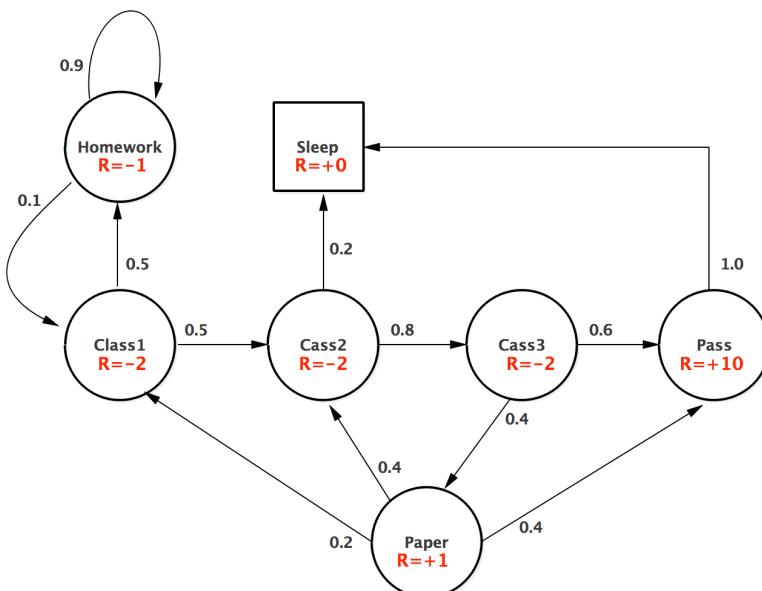


图 2-4 标注奖赏值的马尔科夫奖赏过程

条件： $S_1 = \text{Class1}$ ,  $\gamma = 0.5$ 。

解答：从 Class1 开始，对所有可能的状态序列 a, b, c, .....求回报 Gt 的值，不同的状态序列对应不同的回报，v(s)为随机变量 Gt 的期望：

a. Class1→Class2→Class3→Pass→Sleep

$$Gt1 = -2 - 2 * 0.5 - 2 * 0.25 + 10 * 0.125 + 0 = -2.25$$

b. Class1→Homework→Class1→Class2→Pass→Sleep

$$Gt2 = -2 - 1 * 0.5 - 2 * 0.25 - 2 * 0.125 + 10 * 0.0625 + 0 = -2.625$$

c. .....

因为无法穷举所有的序列状态，所以计算出精确的 Gt 的唯一方法是用过采样近似。

$$V(S_1) = E[G_t | S_t = \text{Class1}] = \frac{G_{t1} + G_{t2} + \dots + G_{tn}}{n} \quad (2-5)$$

马尔科夫决策过程：在 MP 的基础加入奖赏和操作的概念得马尔科夫决策过程。策略的概念使用状态到操作的映射表示。

**【定义四】** 马尔科夫奖赏过程表示为元组  $\langle S, A, P, R, \gamma \rangle$

- S: 有限状态集。  $S = \{S_1, S_2, \dots, S_n\}$  ;
- A: 有限操作集。  $A = \{A_1, A_2, \dots, A_m\}$  ;
- P: 状态转移概率。  $P_{ss'}^a$  代表从在状态  $S_t = s$  采取操作  $A_t = a$  转移到状态  $S_{t+1} = s'$  的概率；

$$P_{ss'}^a = P[S_{t+1} = s' | S_t = s, A_t = a] \quad (2-6)$$

- R: 奖赏函数。在时间步 t, 从状态  $S_t = s$  进行转移所能得到的即时奖赏的期望；

$$R_s^a = E[R_{t+1} | S_t = s, A_t = a] \quad (2-7)$$

- Gt: 回报。从状态  $S_t$  开始，每次状态转移获得的所有即时奖赏的总和，也是强化学习问题关心的积累奖赏；

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (2-8)$$

- $\gamma$  : 折扣因子。  $\gamma \in [0, 1]$ , 表示对即时奖赏的关注程度；
- $\pi(a|s)$ : 策略。从状态集到操作集的映射，代表给定状态  $S_t = s$  时，操作集 A 的分布函数（采取各个操作的概率）；

$$\pi(a|s) = P[A_t = a | S_t = s] \quad (2-9)$$

- $V^\pi(s)$ : 状态价值函数。定义状态价值为在时间步  $t$  从状态  $S_t = s$  开始，依照策略  $\pi(a|s)$ ，采样时考虑所有操作，获得回报的期望。利用  $V^\pi(s)$  可以直观衡量每个状态的价值；

$$V_\pi(s) = E_\pi[G_t | S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s] \quad (2-10)$$

- $Q^\pi(s, a)$ : 状态-操作价值函数。定义操作价值为在时间步  $t$  状态  $S_t = s$  开始，依照策略  $\pi(a|s)$ ，采样时只考虑特定操作  $A_t = a$ ，获得回报的期望。利用  $Q^\pi(s, a)$  可以直观衡量给定状态下每个操作的价值。

$$Q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a] \quad (2-11)$$

状态价值函数  $V^\pi(s)$  和状态-操作价值函数  $Q^\pi(s, a)$  存在关系：

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) * Q_\pi(s, a) \quad (2-12)$$

MRP 与 MDP 对比做说明：图 2-5 中在图 2-4 基础上，MRP 中的状态变为 MDP 中的操作，MDP 中的状态直接用空心圆圈代替。图 2-5 的黑色实心点代表采取操作 *Quit*，产生状态转移，转移概率分别是 0.2, 0.4, 0.4。

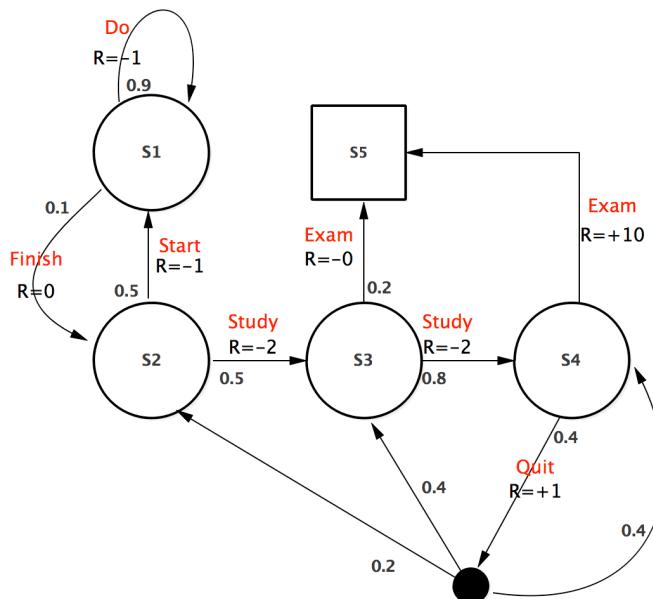


图 2-5 马尔科夫决策过程示例

### 2.1.4 贝尔曼方程与最优贝尔曼方程

贝尔曼方程：或称动态规划方程，方程表示相邻状态的关系。通过贝尔曼方程，上一状态的最优决策问题可转化为下状态的最优决策子问题，使初始状态的最优决策问题逐步迭代转化为终点状态的最优决策子问题求解（通常是易解的）。绝大多数最优决策问题，都可以通过构造贝尔曼方程求解<sup>[12]</sup>。

利用 MDP 对回报  $G_t$  的定义，可以推导出状态价值函数、状态-操作价值函数的贝尔曼方程形式。

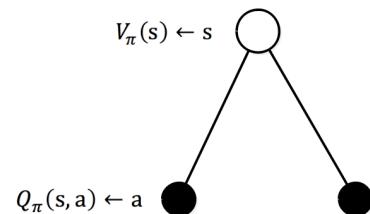
对状态价值函数做变形：

$$\begin{aligned}
 V_\pi(s) &= E_\pi[G_t | S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s] \\
 &= E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s] \\
 &= E_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) | S_t = s] \\
 &= E_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\
 &= E_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s]
 \end{aligned} \tag{2-13}$$

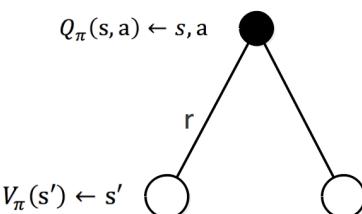
对状态-操作价值函数做变形：

$$Q_\pi(s, a) = E_\pi[R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \tag{2-14}$$

利用递推树与公式 (2-12)，去掉价值函数的期望符号，获得  $V^\pi(s)$ 、 $Q^\pi(s, a)$  表达式 (2-15) (2-16)：

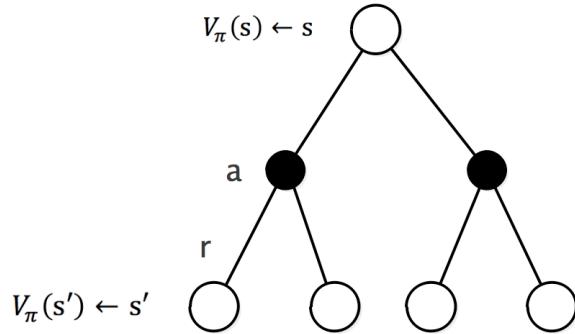


$$V_\pi(s) \text{表达式: } V_\pi(s) = \sum_{a \in A} \pi(a|s) * Q_\pi(s, a) \tag{2-15}$$



$$Q_\pi(s, a) \text{表达式: } Q_\pi(s, a) = R_s^a + \gamma * \sum_{s' \in S} P_{ss'}^a * V_\pi(s') \tag{2-16}$$

合并实心黑色节点，得到完整的状态转移过程，获得只含  $V^\pi(s)$  递推式，即状态价值函数的贝尔曼方程（公式 2-17），递推式表现了当前状态  $s$  的价值与下一状态  $s'$  的价值之间的关系：



$$\text{不含 } Q_\pi(s, a) \text{ 的 } V_\pi(s) \text{ 递推式: } V_\pi(s) = \sum_{a \in A} \pi(a|s) * \left( R_s^a + \gamma * \sum_{s' \in S} P_{ss'}^a * V_\pi(s') \right) \quad (2-17)$$

当已知策略  $\pi(s|a)$ , 按照公式 2-17 就可以计算出任一状态的价值  $V_\pi(s)$ 。若能求解贝尔曼方程 (即找到最优价值函数), 就能找到最优策略, 求解强化学习问题。

**【定义五】** 定义最优状态价值函数:

$$V_*(s) = \max_{\pi} V_\pi(s) \quad (2-18)$$

最优状态价值函数  $V_*(s)$  指从所有策略产生的状态价值函数中, 选取使得所有状态价值最大的函数。

**【定义六】** 定义最优状态-操作价值函数:

$$Q_*(s, a) = \max_{\pi} Q_\pi(s, a) \quad (2-19)$$

最优操作价值函数  $Q_*(s, a)$  指从所有策略产生的状态-操作价值函数中, 选取使得状态-操作对  $(s, a)$  价值最大的函数。

最优价值函数代表 MDP 可能达到的最优表现, 有了最优价值函数, 就有任一状态-操作对的最优价值, 最优策略为选择状态-操作函数值最大操作, MDP 得解。

**【定义七】** 最优策略:

规定策略优劣的比较方式:

$$\text{对所有 } s \in S, \text{ 若 } V_\pi(s) \geq V_{\pi'}(s), \text{ 则 } \pi' \geq \pi \quad (2-20)$$

对所有策略按公式 (2-20) 排序, 获得最优策略  $\pi'$ ,  $\pi'$  有性质:

- a. 任何马尔科夫决策问题，一定存在不少于一个最优策略  $\pi'$ ，优于（至少不劣于）其他策略；
- b. 所有的最优策略拥有相同的状态价值函数和状态-操作价值函数：

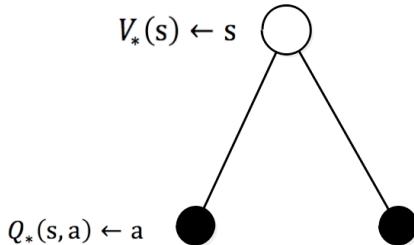
$$V_{\pi^*}(s) = V_*(s) = \max_{\pi} V_{\pi}(s)$$

$$Q_{\pi^*}(s, a) = Q_*(s, a) = \max_{\pi} Q_{\pi}(s, a) \quad (2-21)$$

如何得到最优策略  $\pi^*$ : 若当前处于状态  $s$ , 最优策略就是选择对应的操作  $a$ 。

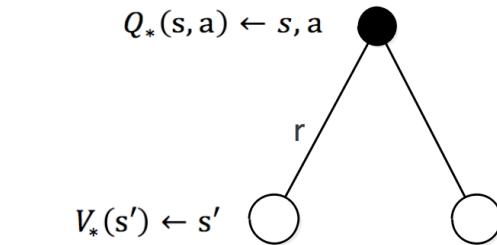
$$\pi_*(a|s) = \begin{cases} 1 & \text{若 } a = \underset{a \in A}{\operatorname{argmax}} Q_*(s, a) \\ 0 & \text{否则} \end{cases} \quad (2-22)$$

最优贝尔曼方程：整合公式 (2-15) (2-16) (2-21)，得最优状态、状态-操作价值函数表达式 (2-23) (2-24) :



$$\text{最优状态值函数: } V_*(s) = \max_a Q_*(s, a)$$

(2-23)



$$\text{最优动作值函数: } Q_*(s, a) = R_s^a + \gamma * \sum_{s' \in S} P_{ss'}^a * V_*(s')$$

(2-24)

将公式 (2-24) (2-23) 结合，得最优状态、状态-操作价值函数：

$$V_*(s) = \max_a R_s^a + \gamma * \sum_{s' \in S} P_{ss'}^a * V_*(s') \quad (2-25)$$

$$Q_*(s, a) = R_s^a + \gamma * \sum_{s' \in S} P_{ss'}^a * \max_{a'} Q_*(s', a') \quad (2-26)$$

有了最优状态、状态-操作价值函数后即可求每一状态、状态-操作对价值，按照定义五找到最优策略，强化学习问题得解。

## 2.1.5 蒙特卡罗强化学习和时间差分强化学习

2.1.3 定义三提到，如果 MDP 的状态转移概率函数未知，就要用到蒙特卡罗强化学习

习法 (Monte-Carlo reinforcement learning, 后称 MC 法) 估计状态的价值。MC 法对马尔科夫决策过程随机采样，模拟多个完整状态转移序列。完整状态转移序列指智能体从任一个状态开始直到终止状态的整个状态转移序列。对任一状态，计算所有完整状态转移序列中包含该状态的序列的回报  $G_t$  的平均值作为该状态价值的估计。

MC 法不依赖状态转移概率函数，直接从经历过的完整状态转移序列中学习，用这些序列回报的平均值近似状态价值。理论上采样获得的完整状态转移序列越多，样本越丰富，估值结果越准确。

我们可以使用 MC 法评估策略优劣：比较回报的大小。计算状态价值的平均值采用了累进更新平均值 (incremental mean)：

$$\begin{aligned}
 \mu_k &= \frac{1}{k} \sum_{j=1}^k x_j \\
 &= \frac{1}{k} \left( x_k + \sum_{j=1}^{k-1} x_j \right) \\
 &= \frac{1}{k} (x_k + (k-1)\mu_{k-1}) \\
 &= \mu_{k-1} + \frac{1}{k} (x_k - \mu_{k-1})
 \end{aligned} \tag{2-27}$$

采样得到新数据后，累进更新平均值不需要历史回报数据，只利用旧平均值、采样所得新数据、采样总个数即可修正平均值。当第  $k$  次采样产生新数据  $X_k$ ，计算  $X_k$  与旧平均值  $\mu_{k-1}$  的差，将差值乘以采样次数的倒数  $1/K$  作为误差，对旧平均值进行修正。

用状态价值、回报值替代公式 (2-27) 中的平均值  $\mu_k$ 、新数据  $X_k$ ，得状态价值更新迭代公式，其中  $K$  为状态被访问的次数。实时计算或状态被访问的次数无法统计准确时，用系数  $\alpha$  (即学习率，表示对误差的吸收程度) 代替系数  $1/K$ ，得到公式 (2-28)。

$$\begin{aligned}
 K &\leftarrow K + 1 \\
 V(S_t) &\leftarrow V(S_t) + \frac{1}{K} (G_t - V(S_t)) \\
 V(S_t) &\leftarrow V(S_t) + \alpha (G_t - V(S_t))
 \end{aligned} \tag{2-28}$$

MC 法需要完整的状态-操作-状态转移序列，因此必须等到采样回合结束后才能更新状态估值（依赖回合）。后来又有人提出来一种新方法，解决了依赖回合的问题：时间差分强化学习方法 (Temporal-Difference Reinforcement Learning, 后称 TD 法)。

TD 法只对一个回合的局部采样，学习采样获得的不完整的状态转移序列，在每个时间步 (time step) 而不是每回合 (episode) 结束后更新状态估值。TD 法是自扩展的

(bootstrapping) , 当前价值函数值的更新依赖于其他已知的价值函数值:

$$V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \quad (2-29)$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \quad (2-30)$$

公式 (2-19) 中 TD 法用离开状态  $S_t$  的即时奖赏  $R_{t+1}$  与下一状态  $S_{t+1}$  的价值估值乘以衰减系数  $\gamma$  二者之和代替 MC 法 (2-28) 中的回报, 计算状态  $S_t$  的价值估值。

TD 法在回合结束前就可开始学习, 或没有结果、回合持续的情况下学习, MC 法则要等回合结束后才可开始学习。更新状态价值时, TD 法使用的 TD 目标值是当前状态价值的有偏估计, 而 MC 法使用回报实际值更新状态估值, 是某一策略下状态价值的无偏估计。尽管 TD 法得到的状态价值估计有偏, 但 TD 法的方差低于 MC 法、对初始值敏感、同时得益于 TD 学习价值更新的灵活性, 通常比 MC 法更高效<sup>[13]</sup>。

### 2.1.6 $\epsilon$ -贪婪策略

MC 法和 TD 法采样需要用到贪婪策略 (Greedy policy) : 操作选择策略为永远选择状态-操作函数值最大的操作, 贪婪策略在动态规划中可以明显加快找到最优策略的速度。完全贪婪策略对实际存在但采样时并没有经历到的状态欠缺考虑, 对经历的次数不多的状态价值估计不够准确, 导致很可能存在一些价值更高而未被探索的状态, 所以完全贪婪一种“片面”的策略。

$\epsilon$  -贪婪策略 ( $\epsilon$  - Greedy) 对贪婪策略优化: 探索过程中有  $1 - \epsilon$  的概率选择状态-操作价值最大的操作, 另有  $\epsilon$  的概率从其他操作中随机地选择一个, 不完全贪婪在采样过程中保持了探索未知状态的概率:

$$\pi_*(a|s) = \begin{cases} 1 - \epsilon & \text{若 } a = \underset{a \in A}{\operatorname{argmax}} Q_*(s, a) \\ \epsilon & \text{其他} \end{cases} \quad (2-31)$$

一方面,  $\epsilon$  -贪婪策略不希望丢掉价值更高的状态和操作, 另一方面, 随着采样增加减小  $\epsilon$  值使算法得以终止于最优策略。

## 2.2 OpenAI Gym

OpenAI 是一家人工智能研究公司, gym 是该公司提供的一款用于研究、对比强化学习算法的环境工具包, gym 环境库为智能体 (agent) 训练提供仿真环境支持, 同时兼

容机器学习的数值计算库，如 tensorflow、theano。由于仿真环境的实现依赖 Python 语言的库 pygame，所以目前 gym 只支持用 Python 语言进行开发。

### 2.2.1 Gym 开源环境库

OpenAI gym 维护了开源的强化学习环境库（<https://github.com/openai/gym>），环境库是测试问题的集合，其中一个问题被称为一个环境（environment），环境是训练模型、测试算法、调节参数的仿真平台。gym 环境库提供了算法设计的接口，在接口中可以进行强化学习算法开发。这种标准化的测试环境有利于比较算法性能，很大程度地减少了编写环境的痛苦。

几乎所有的强化学习经典问题都被囊括在 gym 环境库中，许多学者都利用 gym 环境库进行算法研究。例如：倒立摆小车平衡问题<sup>[14]</sup>、双连杆机械臂运动控制<sup>[15]</sup>、小车爬山问题<sup>[10]</sup>、Atari 游戏<sup>[9]</sup>、实体机器人训练仿真环境<sup>[16]</sup>。除了丰富的内置环境，Gym 还支持用户利用 gym 的强化学习问题框架编写和使用自定义环境。

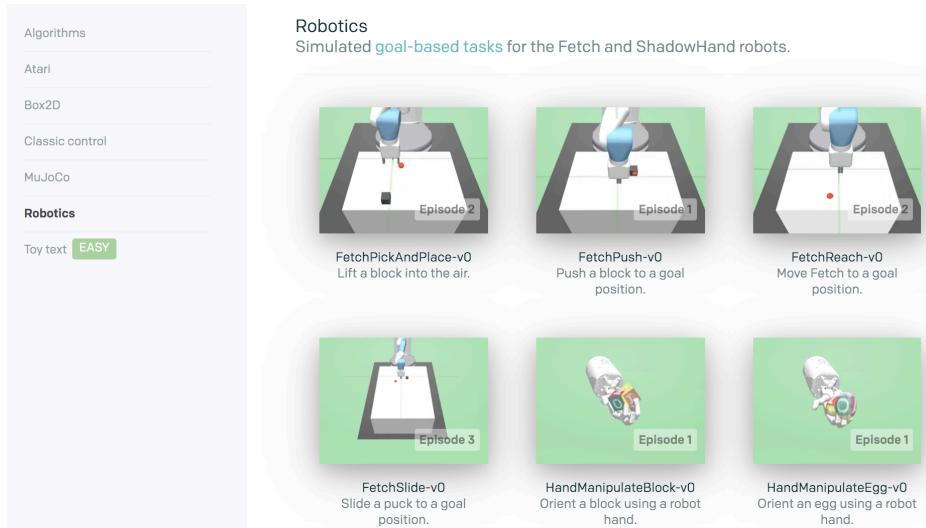


图 2-6 gym 环境库丰富的仿真环境

### 2.2.2 Gym 环境工作机制

Gym 的强化学习问题框架将强化学习问题拆分为两部分：环境（environment）与智能体（agent），一个环境对应一个强化学习问题，智能体被解决问题的算法所控制。环境对外提供统一接口 env，智能体通过这个接口和环境进行交互（采取操作、获得观测值）。



图 2-7 gym 的强化学习框架示意图

Gym 仿真中，最简单的的智能体、环境交互代码示例与运行结果：

```

1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
3 import gym
4
5 env = gym.make('CartPole-v0')
6 env.reset()
7 for _ in range(1000):
8     env.render()
9     action = env.action_space.sample()
10    obv, reward, done, _ = env.step(action)
  
```

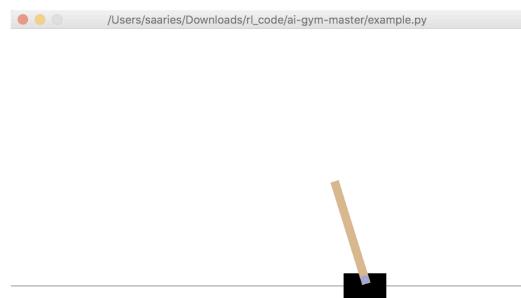


图 2-8 gym 仿真示例

gym 提供的接口 env 包括函数：

a. `env = gym.make('Environment_name')`

创建仿真环境函数。参数 ‘Environment\_name’ 为仿真环境名称。

b. `env.reset()`

环境重置函数。训练过程中，智能体的一次尝试称为一个回合，尝试失败则回合结束，调用该函数重置训练环境，进入下一回合。函数无参数，返回值为环境的初始状态。

c. `env.render(self, mode='human', close=False)`

仿真环境的图像引擎，渲染、绘制、显示环境中的物体图像。图像引擎可视化算法的运行过程。

d. `env.action_space.sample()`

`action_space` 为智能体的操作集合，函数从操作集合里返回一个随机的操作。

e. `env.step(self, action)`

仿真环境的物理引擎，模拟环境中物体的运动规律。该函数描述了智能体与环境交互的所有信息，是环境文件中最重要的函数。函数参数为智能体准备执行

的操作，返回值有四项：object observation: action 执行后，环境的观测值。例如：从相机获取的像素点、机器人关节的角度、棋盘游戏的状态；float reward: action 执行后，智能体获得的奖赏；boolen done: action 执行后，当前回合是否终止。如：当机器人摔倒、棋盘游戏结束、任务失败，返回 `done = True`，则终止当前回合，重置环境；dict Info: 用于调试的诊断信息。

### 2.2.3 自定义 Gym 环境

自定义 Gym 环境需要注意的几点：

- a. 自定义 Gym 环境将被封装为一个库的形式进行调用；
- b. 自定义 Gym 环境将继承 `gym.Env`，需要对 `env` 接口中函数进行重载以实现自定义功能；
- c. 自定义 Gym 环境需要在 Gym 安装目录 “`gym/envs/_init_.py`” 的注册表中进行注册才可以使用 `gym.make()` 调用；

自定义 Gym 环境其实就是自定义一个强化学习问题，它的逻辑封装方式和 Gym 的内置环境是一样的，应该从马尔科夫决策过程入手考虑需要表示的元素：

- a. 问题表达：状态空间 `observation_space`、操作空间 `action_space`、回报函数 `reward`、状态转移概率（可选）；
- b. 环境表达：在子类中，使用 `_render()`、`_step()`、`_reset()` 重写父类 `gym.Env` 的方法 `render()`、`step()`、`reset()`，为自定义环境编写图画引擎与物理引擎；
- c. 问题求解：在算法主函数中，从 `env` 接口的返回值中获取对环境的观测值，计算决策后，再通过 `env` 接口将控制智能体的操作输入至环境中，形成一个交互的过程。

`gym` 构建的自定义迷宫环境描述：

- a.  $m \times n$  的方型栅格；
- b. 具有起点（左上栅格）与终点（右下栅格）；
- c. 栅格间的黑线为墙，无法通过；灰线为通路，可通过；墙壁不占用栅格；
- d. 图中蓝色点为智能体，智能体的移动方向有：`up, down, west, east`；

`gym` 构建的迷宫环境问题描述：

智能体的目标是寻找一条路径，该路径为从迷宫的起点到达终点的最短通路

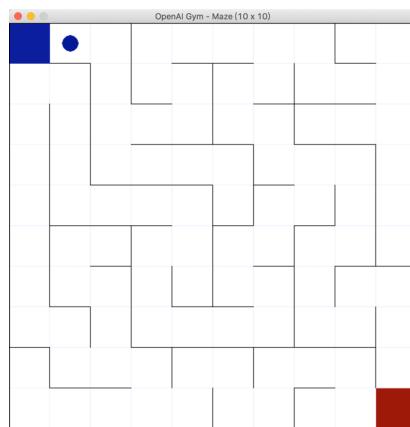


图 2-9 10×10 大小的样例迷宫环境

### 2.3 神经网络与强化学习

文 2.1.3 MDP 中的状态和操作都是离散的，表达的强化学习问题规模较小，而实际问题的状态数量、操作数量通常会庞大或连续。如果继续用 MDP 中的圆圈表示状态、操作，那么问题表示空间会变得巨大，计算效率降低，甚至会无法求得较好的解。

经文 2.1.4 知识铺垫，已确定强化学习问题求解需要找到最优状态、状态-操作价值函数，前文的价值函数是离散的，因为它使用字典之类的表格式方法存储状态和操作，并不能直接用于求连续状态空间上状态价值。因为寻找最优价值函数的思路是一致的，可以将 MDP 模型一般化，用连续的近似价值函数代替了离散的价值函数，对连续状态进行价值评估。

通过引入适当的参数，选取恰当的状态描述特征，拟合出近似价值函数近似计算状态、操作的价值。对带参数的近似价值函数，只要确定参数，就可以近似计算得到状态、操作的价值。这种设计不需要再存储每一个状态或操作的价值，只存储近似函数结构和参数就行了。这样一来，强化学习问题中寻找最优价值函数的问题，就转变成设计近似最优函数和求解近似函数参数。

函数近似方法分为线性近似、非线性近似，非线性近似的表达力比线性近似强，因此多采取非线性近似。非线性近似可利用目标函数梯度的下降训练深度神经网络，求解函数参数。神经网络的加入就是为了求解状态、操作数量巨大或连续的强化学习问题，实践也证明，它取得了很好的效果。

### 2.3.1 近似价值函数的建立和求解

建立状态价值近似函数  $V^*$ ，函数由参数  $\theta$  描述，输入为表示状态的连续变量  $s$ ，输出为状态  $s$  的近似价值。通过调整参数  $\theta$ ，使近似函数  $V^*$  逐渐接近基于某一策略  $\pi$  的最终状态价值，这个函数就是状态价值函数  $V_\pi(s)$  的近似表示。

$$V^*(s, \theta) \approx V_\pi(s) \quad (2-32)$$

类似建立状态-操作价值近似函数  $Q^*$ ，函数由参数  $\theta$  描述，输入为表示状态的连续变量  $s$  和表示操作的连续变量  $a$ ，输出为状态  $s$  对应的所有操作  $a$  的价值。通过调整参数  $\theta$ ，使  $Q^*$  的函数值逐渐接近基于某一策略  $\pi$  的最终状态-操作价值，这个函数就是行为价值函数  $Q_\pi(s, a)$  的近似表示。

$$Q^*(s, a, \theta) \approx Q_\pi(s, a) \quad (2-33)$$

价值近似函数中，表示状态的变量  $s$  是由一组数据构成的特征向量，特征向量的每一项为状态  $s$  的一个特征，该项的数据值为特征值；参数  $\theta$  也是一个向量，通过建立目标函数、训练神经网络、使用梯度下降方法求解。

理论上任何函数都可以被用作近似函数，神经网络是一种常用的非线性近似函数。神经网络的基本单位是进行非线性变换神经元（图 2-9 右），神经元同层排列、异层互连，实现复杂的非线性近似结构。非线性变换神经元在线性近似的基础上增加了偏置项  $b$  和非线性激活函数  $\sigma$ ，因此单个神经元最终输出为拟合内容更丰富的非线性结构：

$$y = \sigma(w^T x + b) \quad (2-34)$$

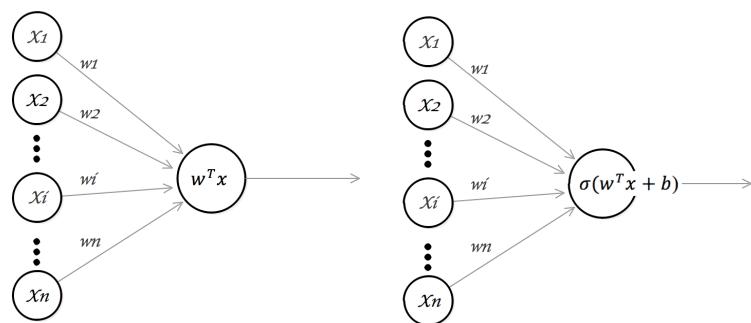


图 2-10 左为线性近似；右为非线性近似

### 2.3.2 目标函数和梯度下降

文 2.1.5 有价值函数更新公式 (2-29) (2-30) :

$$V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \quad (2-29)$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \quad (2-30)$$

因为学习过程中价值估计有偏差，当每个时间步结束后，需要沿目标价值方向以学习率  $\alpha$  做幅度更新。经过多次迭代更新后，价值函数趋于定值，则意味着近似函数估计出的目标值与实际值相同。用  $\hat{Q}$  代替公式 (2-30) 中的  $Q$ ，得近似价值函数更新公式：

$$\hat{Q}(S_t, A_t, \theta) \leftarrow \hat{Q}(S_t, A_t, \theta) + \alpha (R_{t+1} + \gamma \hat{Q}(S_{t+1}, A_{t+1}, \theta) - \hat{Q}(S_t, A_t, \theta)) \quad (2-35)$$

假设找到参数  $\theta$  使得价值函数收敛不再更新，那么意味着找到了基于某策略的最终价值函数或最优价值函数，但事实上，很难找到完美参数  $\theta$  使公式 (2-36) 完全成立。同时由于训练近似函数依靠采样得到的数据，即使上式采样得到的状态转换成立，也并不能保证对所有的状态转换成立。

$$R_{t+1} + \gamma \hat{Q}(S_{t+1}, A_{t+1}, \theta) = \hat{Q}(S_t, A_t, \theta) \quad (2-36)$$

利用采样产生的  $M$  个状态转换中，当前价值  $\hat{Q}$  与目标价值  $R + \gamma \hat{Q}$  差距的方差，可以构建目标函数  $J(\theta)$ ，评判近似价值函数的收敛情况，近似价值函数收敛意味着  $J(\theta)$  值逐渐减小，从公式 (2-37) 知  $J(\theta)$  不会为负，同时存在一个极小值使  $J(\theta)=0$ 。

$$J(\theta) = \frac{1}{2M} \sum_{k=1}^M [R_k + \gamma \hat{Q}(S_{k+1}, A_{k+1}, \theta) - \hat{Q}(S_k, A_k, \theta)]^2 \quad (2-37)$$

对多元函数的所有参数求偏导，各个参数偏导的向量形式就是梯度。函数图像上的点沿该点梯度向量方向函数值增加速度最快；梯度向量反方向函数值减少速度最快；当某点梯度值为 0，函数在该点取得一个极大值或极小值。也就是说，沿着梯度向量，更加容易找到函数的极值。

目标函数  $J(\theta)$  表示当前价值  $\hat{Q}$  与目标价值  $R + \gamma \hat{Q}$  差距的方差，梯度下降的目的是尽量取得  $J(\theta)$  的极小值，加速近似价值函数的收敛。

## 2.4 本章小结

首先，本章简单介绍了强化学习算法理论，从最底层的马尔科夫过程架构，讲述强化学习问题的知识表示，并指出了迷宫问题使用强化学习方法求解的可行性。

其次，OpenAI Gym 为本文用作强化学习算法训练的仿真环境，简要提及。

最后，为深度强化学习做铺垫，介绍了一些神经网络应用在强化学习算法中需要的基本知识。由于神经网络内容庞杂，因此只描述了它与下文强化学习算法有关的部分，其余知识不再赘述。

## 第三章 自动寻路系统设计

### 3.1 设计思路

迷宫自动寻路系统的设计思路如图 3-1。首先需要实现迷宫环境，借助了 OpenAI gym 的强化学习问题框架自定义迷宫环境。其次需要实现智能体寻路的强化学习算法，算法分为两种，借助表格的强化学习算法与借助深度神经网络的学习算法。表格型强化学习算法实现了 Q-Learning 算法与 SARSA 算法，深度强化学习算法实现了 Deep Q Network 算法。

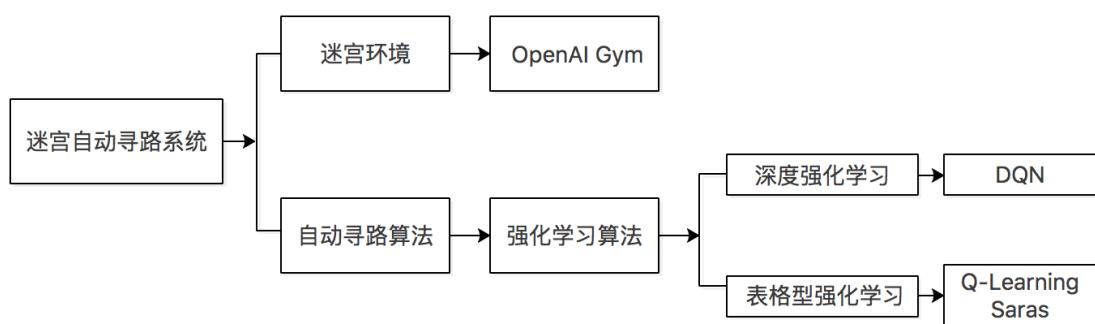


图 3-1 自动寻路系统设计思路

本章将内容将分为两部分，第一部分介绍强化学习自动寻路算法设计，第二部分介绍迷宫环境的编写。

### 3.2 自动寻路算法设计

实现的算法包括表格型强化学习算法：S 算法、Q 算法，深度强化学习 DQN 算法。这些算法对强化学习问题的求解思路都是利用价值迭代逼近状态、状态-操作价值函数，当算法找到最优价值函数即解得到最优策略（文 2.1.4）。

S 算法、Q 算法维护一张状态-操作价值函数值表，在迭代过程中不断更新该表，由文 2.1.6 知，算法可以收敛至最优状态-操作价值函数值表，近似最优的状态-操作价值函数。在迷宫问题中的每一个栅格为一个状态 S，操作  $A=\{up, down, west, east\}$ 。从 S 算法、Q 算法求得的状态-操作价值表中，可以读取出迷宫每一个栅格对应的所有操作的最优状态-操作价值函数值  $Q(S, A)$ ，智能体在每个栅格选择最大  $Q(S, A)$  值对应操作 A，连续决策后的操作 A 序列就是迷宫强化学习问题的最优解。

### 3.2.1 SARSA 算法

SARSA 算法（后称 S 算法）名称来源于对图 3-2 序列：state→action→reward→state →action 的描述：在状态 S 时，智能体依策略（操作选择策略）判断产生操作 A 并执行，产生状态操作对(S, A)并获得环境的即时奖赏 R，进入下一状态 S'；在状态 S'时，智能体依策略（价值更新策略）判断产生操作 A'，但并不执行，而是利用价值函数计算 A' 对应的状态-操作对(S', A')的价值，利用该价值 Q(S', A')与所获得的奖赏 R 之和更新上一个状态-操作对(S, A)的价值，之后才正式执行操作 A'。

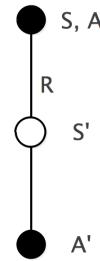


图 3-2 S 算法拆解示意

S 算法属于时间差分强化学习方法(文 2.1.5)，S 算法在采样得到状态序列{ S<sub>1</sub>, S<sub>2</sub>, ..., S<sub>n</sub> }过程中的每一次状态转换：从状态 S 采取操作 A 到达状态 S'后，都要更新状态-操作对(S, A)的价值 Q(S, A)，这一过程使用  $\epsilon$ -贪婪策略进行策略迭代，S 算法的状态-操作函数 Q(S, A)的值更新迭代公式 (3-1)：

$$Q(S, A) \leftarrow Q(S, A) + \alpha (R + \gamma Q(S', A') - Q(S, A)) \quad (3-1)$$

---

#### 算法 1: Sarsa 算法

```

输入: episodes, α, γ
输出: Q
initialize: set  $Q(s, a)$  arbitrarily, for each  $s$  in  $\mathbb{S}$  and  $a$  in  $A(s)$ ; set  $Q(\text{terminal state}, \cdot) = 0$ 
repeat for each episode in episodes
    initialize:  $S \leftarrow$  first state of episode
     $A = \text{policy}(Q, S)$  (e.g.  $\epsilon$ -greedy policy)
    repeat for each step of episode
         $R, S' = \text{perform\_action}(S, A)$ 
         $A' = \text{policy}(Q, S')$  (e.g.  $\epsilon$ -greedy policy)
         $Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$ 
         $S \leftarrow S'; A \leftarrow A'$ ;
    until S is terminal state;
until all episodes are visited;
  
```

---

图 3-3 S 算法流程

由于迷宫的状态和操作都是离散的，所以 S 算法中的状态-操作价值函数 Q(S, A)值也是离散的，函数在内存中的存储形式是一张行为状态 S，列为操作 A 的 Q 表，也就是

S 算法的输出结果。

S 算法输入：episodes 是总训练回合数； $\alpha$  是控制学习的速度学习率； $\gamma$  是控制对未来奖赏的关注度的奖赏折扣因子。算法输出：大小为状态数目×操作数目的 Q 表。每个训练回合 episode 中，step 代表一个时间步，每个时间步都要用采样产生的状态转移序列{S→A→R→S'→A'}更新状态-操作对 Q(S, A) 的价值。

```

if __name__ == '__main__':
    # 初始化环境
    env = gym.make("maze-sample-10x10-v0")
    state_list = []

    # 初始化 Q 表
    for i in range(maze_size):
        for j in range(maze_size):
            state_list.append((i, j))

    # 记录每次训练所用的时间步
    step_used = []

    # SARSA 算法实现: agent
    agent = Agent(state_list, action_num)

    # 尝试 500 回合
    for episode in range(EPISODES):
        if episode>10:
            agent.epsilon = 0

        # 重置迷宫环境, state = (0,0)
        state = env.reset()
        state = (state[0], state[1])
        # 依策略选择动作
        action = agent.choose_act(state)

        for step in range(500):
            # env.render()
            state_, reward, done, _ = env.step(action)
            state_ = (state_[0], state_[1])
            action_ = agent.choose_act(state_)
            # 每个时间步更新一次 Q 表
            agent.update_q(state, action, reward, state_, action_)
            # S->S', A->A'
            state = state_
            action = action_
            # 当使用步数>499, 重新开始回合
            if done or step == 499:
                print("episode: {}/{}, Took = {} steps, Epsilon = {}"
                      .format(episode, EPISODES, step, agent.epsilon))
                step_used.append(step)

```

图 3-4 S 主流程代码

```

class Agent():
    def __init__(self, state_list, action_size):
        self.state_list = state_list
        self.action_size = action_size
        self.gamma = 0.95
        # e-贪婪
        self.epsilon = 1
        self.epsilon_min = 0.1
        self.epsilon_decay = 0.9
        self.alpha = 0.5
        self.q = self.init_q()

    # 初始化 Q 表
    def init_q(self):...

    # 动作选择策略
    def choose_act(self, state):
        # e-贪婪：不断减小 epsilon 值，加速收敛
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

        # e-贪婪：有 epsilon 的概率选择随机操作
        if random.random() < self.epsilon:
            return np.random.randint(4)

        # e-贪婪：有 1-epsilon 的概率选择Q(s, a)值最大的操作
        val = -10000000
        action = 0
        for i in range(self.action_size):
            if self.q[state][i] > val:
                action = i
                val = self.q[state][i]

        return action

    # 更新 Q 表
    def update_q(self, state, action, reward, next_state, next_action):
        self.q[state][action] = self.q[state][action] + \
            self.alpha * (reward + self.gamma * self.q[next_state][next_action] - self.q[state][action])

```

图 3-5 S 实现代码

### 3.2.2 Q-Learning 算法

S 算法和 Q-Learning 算法（后简称 Q 算法）最本质的区别是状态-操作价值函数 Q 的更新方式。如拆解图 3-6：在状态 S 时，智能体依策略（操作选择策略）判断产生操作 A 并执行，产生状态-操作对(S, A)并获得环境的即时奖赏 R，进入下一状态 S'；在状态 S' 时，智能体找到所有可能的操作 A'，但并不执行，而是利用价值函数计算所有可能的下一状态-操作对(S', A')的价值，选择其中  $Q(S', A')$  最大值所对应的操作 A' 作为下一步采取的操作（价值更新策略）；同时，利用该价值  $\max[Q(S', A')]$  与所获得的奖赏 R 之和更新上一个状态-操作对(S, A)的价值。

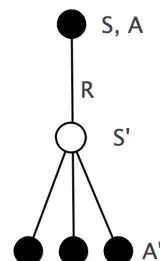


图 3-6 Q 算法拆解示意

S 算法的操作选择策略和价值更新策略是相同的，都是  $\epsilon$ -贪婪；Q 算法的操作选择策略是  $\epsilon$ -贪婪，但价值更新策略却是完全贪婪的（选择  $Q(S', A')$  最大值所对应的操作  $A'$ ）。这种价值更新的方式，引导状态 S 的前进方向永远朝着最大化行为价值的方向更新，比起 SARSA 算法，Q 算法不但能够使个体尝试多方面的探索（操作选择策略为  $\epsilon$ -贪婪），同时加大了算法收敛至最优策略的速度（价值更新策略为完全贪婪）。Q 算法的状态-操作函数  $Q(S, A)$  的值更新迭代式：

$$Q(S, A) \leftarrow Q(S, A) + \alpha (R + \gamma \max [Q(S', A')] - Q(S, A)) \quad (3-2)$$

---

### 算法 3: Q 学习算法

---

```

输入: episodes,  $\alpha$ ,  $\gamma$ 
输出:  $Q$ 
initialize: set  $Q(s, a)$  arbitrarily, for each  $s$  in  $\mathbb{S}$  and  $a$  in  $\mathbb{A}(s)$ ; set  $Q(\text{terminal state}, \cdot) = 0$ 
repeat for each episode in episodes
    initialize:  $S \leftarrow$  first state of episode
    repeat for each step of episode
         $A = \text{policy}(Q, S)$  (e.g.  $\epsilon$ -greedy policy)
         $R, S' = \text{perform\_action}(S, A)$ 
         $Q(S, A) \leftarrow Q(S, A) + \alpha (R + \gamma \max_a Q(S', a) - Q(S, A))$ 
         $S \leftarrow S'$ 
    until  $S$  is terminal state;
until all episodes are visited;

```

---

图 3-7 Q 算法流程

由于 Q 算法的贪婪，它在学习过程中的表现比 S 算法更大胆，在下面的例子中：智能体需要找到沼泽（图 3-8 灰色栅格）一端到另一端的最短路径，规定智能体每移动一步奖赏-1，到达沼泽另一端奖赏+100，跌入陷阱奖赏-100。可以直接看出智能体的最优路径是沿着沼泽周围走到另一端，在学习过程中的差异在于，Q 算法倾向靠近沼泽，而 S 算法倾向于和沼泽保持一定距离。两种学习方式最后都可以收敛到最优策略，但 Q 算法会更快。在现实中，如果采样尝试需要付出代价，如机器人有可能跌入沼泽摔坏了，选择保守的算法会更好；如果更重视算法的收敛速度，选择大胆的 Q 算法会更好。



图 3-8 Q 算法、S 算法的差异

```

if __name__ == '__main__':
    # 初始化环境
    env = gym.make("maze-sample-10x10-v0")
    state_list = []

    # 初始化 Q 表
    for i in range(maze_size):
        for j in range(maze_size):
            state_list.append((i, j))

    # 记录每次训练所用的时间步
    step_used = []

    # Q-Learning 算法实现: agent
    agent = Agent(state_list, action_num)

    # 尝试 500 回合
    for episode in range(EPISODES):
        if episode > 10:
            agent.epsilon = 0

        # 重置迷宫环境, state = (0,0)
        state = env.reset()
        state = (state[0], state[1])

        for step in range(500):
            # env.render()
            # 依策略选择动作
            action = agent.choose_act(state)
            state_, reward, done, _ = env.step(action)
            state_ = (state_[0], state_[1])
            # 每个时间步更新一次 Q 表
            agent.update_q(state, action, reward, state_)
            # S-S'
            state = state_
            # 当使用步数>499, 重新开始回合
            if done or step == 499:
                print("episode: {}/{}, Took = {} steps, Epsilon = {}"
                      .format(episode, EPISODES, step, agent.epsilon))
                step_used.append(step)
                break

```

图 3-9 Q 算法主流程代码

```

class Agent():
    def __init__(self, state_list, action_size):
        self.state_list = state_list
        self.action_size = action_size
        self.gamma = 0.95
        # e-贪婪
        self.epsilon = 1
        self.epsilon_min = 0.1
        self.epsilon_decay = 0.995
        self.alpha = 0.5
        self.q = self.init_q()

    def init_q(self):...

    # 动作选择策略
    def choose_act(self, state):
        # e-贪婪: 不断减小 epsilon 值, 加速收敛
        if agent.epsilon > agent.epsilon_min:
            agent.epsilon *= agent.epsilon_decay

        # e-贪婪: 有 epsilon 的概率选择随机操作
        if random.random() < self.epsilon:
            return np.random.randint(action_num)

        # e-贪婪: 有 1-epsilon 的概率选择Q(s, a)值最大的操作
        val = -10000000
        action = 0
        for i in range(self.action_size):
            if self.q[state][i] > val:
                action = i
                val = self.q[state][i]
        return action

    # 更新 Q 表
    def update_q(self, state, action, reward, state_):
        val = -10000000
        for i in range(self.action_size):
            if self.q[state_][i] > val:
                val = self.q[state_][i]
        self.q[state][action] = self.q[state][action] + \
            self.alpha * (reward + self.gamma * val - self.q[state][action])

```

图 3-10 Q 算法实现代码

### 3.2.3 Deep Q Network 算法

Deep Q Network（后称 DQN 算法）：基于深度神经网络的 Q 算法。同样是求解最优价值函数，Q 算法和 S 算法是通过迭代更新 Q 表格，处理过程中的输入数据是计算所得  $Q(S, A)$ ；但 DQN 算法对最优价值函数的求解是通过神经网络的梯度下降近似价值函数，处理过程中的输入数据是状态转移序列数据( $S, A, R, S'$ )。

为了能更好的结合神经网络进行强化学习，DQN 算法比通常的神经网络加入了两点特别的地方：经验池（Experience Replay）和备份网络<sup>[17]</sup>（Fixed Q Target Network）。经验池存放采样得到的状态转移的序列数据( $S, A, R, S'$ )，训练神经网络时，从经验池随机抽取 batch 大小的序列作为训练样本。这样做的目的是减少一段时间内的数据相关性，

为随机梯度下降模拟一个样本独立同分布的条件；不但如此，还能调节经验池中经验的优先级，有选择性地给予出现次数少但更重要的经验在神经网络中获得更多被学习的机会。

深度强化学习有两套结构相同的神经网络：TargetNet 和 MainNet。MainNet 和 TargetNet 的初始化参数相同，但 MainNet 参数随着训练不断更新，而 TargetNet 参数保持不变。一段时间后，MainNet 的参数趋于稳定，则把 MainNet 的参数复制到 TargetNet，循环往复。这一策略使得 TargetNet 的输出在一段时间内是稳定的，可以为 MainNet 的训练提供稳定的监督数据。Q 和 S 算法中，Q 值更新用到了 Q 目标和 Q 当前，Q 目标为  $\max[Q(S')]$  与奖赏 R 之和，Q 当前来自当前 Q 表（图 3-11）。DQN 中，Q 目标来自具奖赏 R 与 TargetNet 输出之和，Q 当前来自 MainNet 输出（图 3-12）。

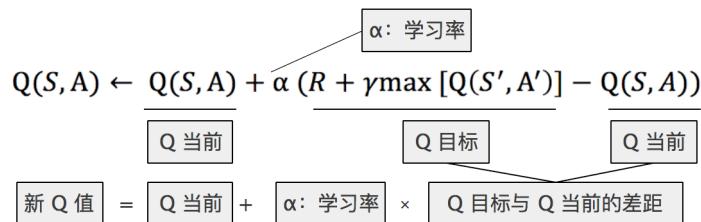


图 3-11 Q 算法 Q 值更新示意

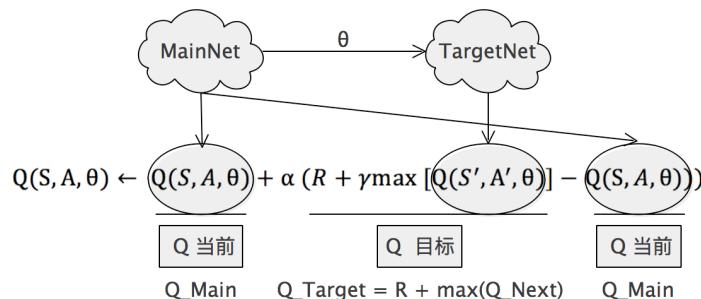


图 3-12 DQN 算法的 Q 目标和 Q 当前

在 DQN 算法中，假设迷宫大小为  $n \times n$ ，为了得到栅格信息，则需要使用大小为  $[2n+1, 2n+1]$  的数组记录栅格：0 代表栅格空，1 代表栅格为墙壁，2 代表智能体处于该栅格，3 代表终点栅格（图 3-13）。

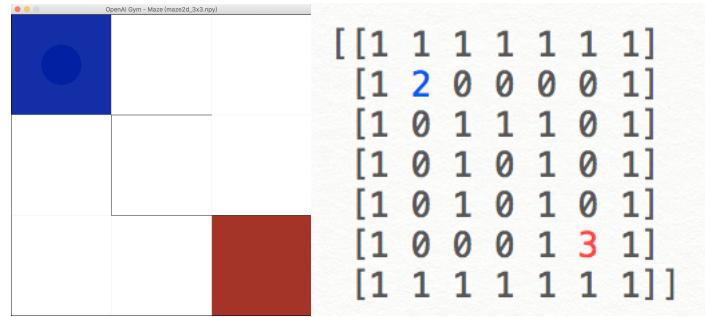


图 3-13 迷宫环境（左）的机器表示（右）

经验池保存的转换序列形式( $S, A, R, S'$ ):  $S$  为智能体发生状态转移前的迷宫状态 ( $s.shape=(2*maze\_size+1,2*maze\_size+1)$ ) ,  $A \in \{\text{north, south, east, west}\}$  为智能体当前采取的操作,  $R$  为采取操作  $A$  后获得的即时奖赏,  $S'$  为智能体发生状态转移后的迷宫状态。

每次训练, 从经验池从中随机抽取大小为  $batch\_size$  的批数据作为训练数据集, 每条数据格式为( $S, A, R, S'$ ), 将  $batch$  的每条数据拆分:  $S$  输入 MainNet, 网络输出为状态  $S$  下所有状态-操作的  $Q$  值:  $Q_{\text{Main}}=\{Q(S, \text{North}), Q(S, \text{South}), Q(S, \text{East}), Q(S, \text{West})\}$ ;  $S'$  输入 TargetNet, 网络输出为状态  $S$  下所有状态-操作的  $Q$  值:  $Q_{\text{Next}}=\{Q(S', \text{North}), Q(S', \text{South}), Q(S', \text{East}), Q(S', \text{West})\}$ 。 $Q_{\text{Eval}}$  为  $Q$  当前,  $Q_{\text{Target}}$  为每个转换序列( $S, A, R, S'$ )对应奖赏  $R$  与  $\max(Q_{\text{Next}})$  之和:  $Q_{\text{Target}}=R+\gamma * \max(Q_{\text{Next}})$ , 作为  $Q$  目标。

有了  $Q$  现实 ( $Q_{\text{Eval}}$ ) 和  $Q$  目标 ( $Q_{\text{Target}}$ ), 加上神经网络的参数  $\theta$ , 就可以写出目标函数公式 (3-3), 训练 TargetNet 和 MainNet, 对目标函数  $J(\theta)$  进行梯度下降, 以求得最优价值函数  $Q$ :

$$\begin{aligned}
 Gradient\ Descent(J(\theta)) &= Optimizer.\minizine(J(\theta)) \\
 &= Optimizer.\minizine(loss) \\
 loss = J(\theta) &= \frac{1}{2M} \sum_{k=1}^M (Q_{\text{Target}} - Q_{\text{Main}})^2
 \end{aligned} \tag{3-3}$$

本次实验中 DQN 算法结如图 3-8, MainNet 和 TargetNet 为两套神经网络;  $S$  与  $S_+$  分别为 MainNet 和 TargetNet 的输入;  $Q_{\text{Main}}(S, A)$  和  $Q_{\text{Next}}(S, A)$  为神经网络的输出  $Q$  值,  $Q_{\text{Next}}$  与奖赏  $R$  的和作为  $Q_{\text{Target}}$ , 用于构造损失函数;  $loss$  操作进行式 3.3 的计算得到  $loss$ ,  $train$  操作为梯度下降操作, 即式 3.4:  $\text{Optimizer}.\minizine(loss)$ 。梯度下降操作将更新网络参数  $\theta$ , 一段时间后, MainNet 的参数  $\theta$  被复制到 TargetNet。

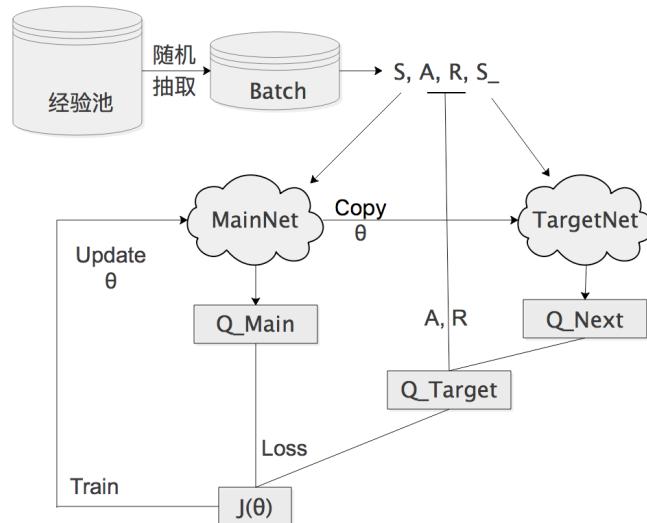


图 3-14 DQN 参数更新的过程

TargetNet 和 MainNet 结构相同（图 3-8）：输入层 $\times 1 \rightarrow$ 隐藏层 $\times 7 \rightarrow$ 输出层 $\times 1$ 。

输入层的特征图尺寸为 $[2n+1, 2n+1, 1]$ （ $n$  为迷宫大小），隐藏层为卷积层 $\times 3$ ，池化层 $\times 3$ ，全连接层 $\times 1$ ，输出层维度为 $[1, 4]$ 对应四种操作的 Q 值。

由于迷宫大小会影响输入向量的维度，而神经网络的结构设计需要参照不同的输入维度进行改进。综合训练时间的考虑，在多个神经网络中，我在多次试验后，选择了效果较好模型的作为实验对象，设计的网络结构适用于  $17 \times 17$ 、 $19 \times 19$  的输入。图 3-10 为结构图。

设计原则遵循：

- 输入层神经元个数等于输入特征图维度，输出层神经元个数等于输出 Q 值维度；
- 池化层位于卷积层之后；
- 隐藏层的神经元个数保持相等；
- 每经过一次池化层，过滤器深度加倍
- 小的卷积核尺寸、步长利于保留特征，增大过滤器深度可以考虑更深的特征维度。

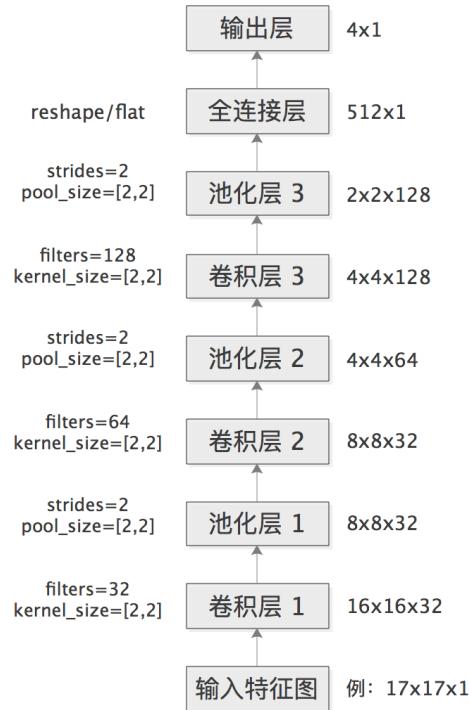


图 3-15 TargetNet 和 EvalNet 的结构

```

def network(self, learning_rate = 1e-3):
    with tf.variable_scope(self.net_name):
        self.X = tf.placeholder(tf.float32, [None, self.data_x, self.data_y, 1])
        observation = self.X
        self.Y = tf.placeholder(tf.float32, [None, self.output_size])

        # 17x17x1
        conv1 = tf.layers.conv2d(
            inputs = observation, filters = 32, kernel_size = [2,2],
            padding = "valid", activation = tf.nn.relu)
        # 16x16x32

        pool1 = tf.layers.max_pooling2d(
            inputs = conv1, pool_size = [2,2], strides = 2)
        # 8x8x32

        conv2 = tf.layers.conv2d(
            inputs = pool1, filters = 64, kernel_size = [2,2],
            padding = "same", activation = tf.nn.relu)
        # 8x8x64

        pool2 = tf.layers.max_pooling2d(
            inputs = conv2, pool_size = [2,2], strides = 2)
        # 4x4x64

        conv3 = tf.layers.conv2d(
            inputs = pool2, filters = 128, kernel_size = [2,2],
            padding = "same", activation = tf.nn.relu)
        # 4x4x128

        pool3 = tf.layers.max_pooling2d(
            inputs = conv3, pool_size = [2,2], strides = 2)
        # 2x2x128
        flat = tf.reshape(pool3, [-1, 2*2*128])
        # reshape to 1x512

        fc = tf.layers.dense(
            inputs = flat, units = self.output_size)
        # 1x4

        self.Qpred = fc
        self.loss = tf.losses.mean_squared_error(self.Y, self.Qpred)
        train = tf.train.AdamOptimizer(learning_rate)
        self.train_op = train.minimize(self.loss)
    
```

图 3-16 DQN 实现代码 (MainNet 与 TargetNet 结构相同)

### 3.3 迷宫环境设计

为了保证测试环境的多样化，迷宫环境有三种模式：sample、random、plus，分别对应标准迷宫环境、随机的标准迷宫环境、随机的转移迷宫环境。

标准迷宫环境于 2.2.3 给出描述，转移迷宫环境在标准迷宫环境的基础上增加了两两成对的转移栅格（图 3-17 中的有色栅格），当智能体进入一转移格，将被强制转移到另一同色转移格。

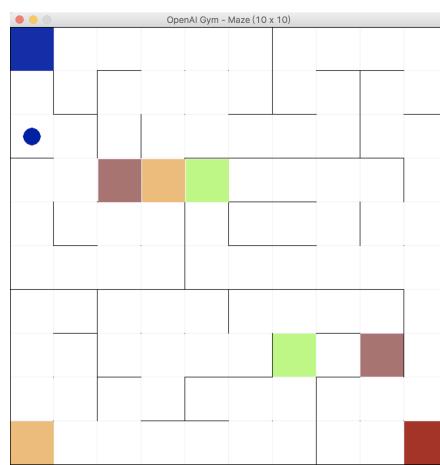
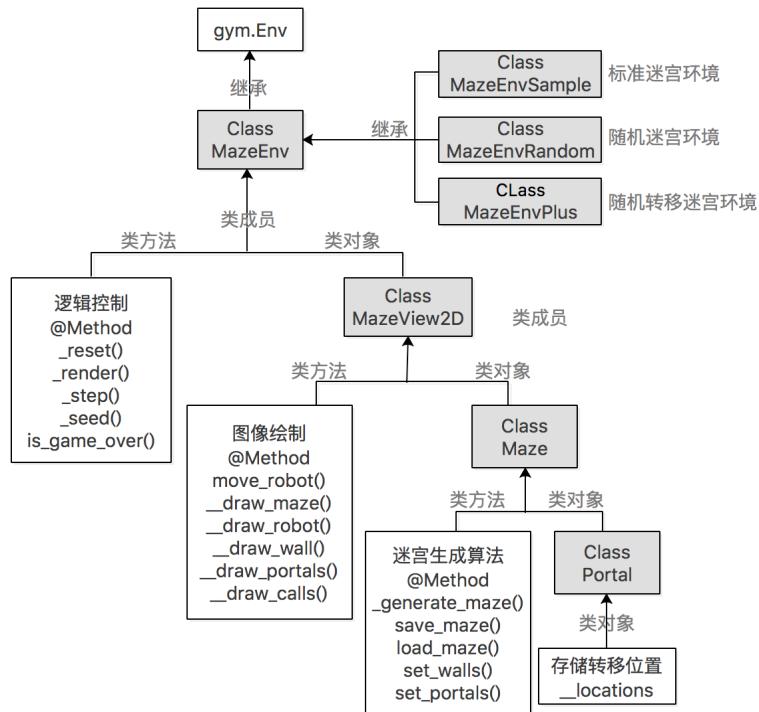


图 3-17 转移栅格迷宫环境

OpenAI gym 环境中，自定义环境继承父类 `gym.Env`，遵循强化学习框架编写基础的类方法。可视化环境的实现采用了 python 的轻量级库 `pygame`，功能囊括图像绘制（形状、线、点、颜色）、动画渲染、文本、自带计时器，满足本次实验交互操作需求。

自定义的迷宫环境将被封装为库 `gym_maze`，结构如下：



实现顺序从底层的类 Maze 与 Portal 开始，生成迷宫、转移格、转移迷宫，之后是类 MazeView2D，将物理内存里的迷宫地图按照逻辑用库 pygame 的方法绘制到屏幕，接着构造类 MazeEnv，类 MazeEnv 继承自父类 gym.Env，在 MazeEnv 实现了 gym.Env 提供的接口。最后是类 MazeEnvSmple、MazeEnvRandom、MazeEnvPlus，为三种模式的迷宫建立各自的类，减少调用的参数，令程序更加整洁。

库文件的存储结构与 gym.Env 文档的要求一致，存放在 gym\_maze/envs 文件夹下。文件夹中第一个 \_\_init\_\_.py 的作用是标识当前文件夹 gym\_maze 是一个自定义库，第二个 \_\_init\_\_.py 是自定义库 gym\_maze 在 gym.Env 中的注册文件。maze\_view\_2d.py 中定义和实现了 Class MazeView2D、Maze、Portal，maze\_generator.py 中的代码用于生成样例迷宫并保存在文件夹 maze\_sample 中，maze\_env.py 中定义和实现了 Class MazeEnv、MazeEnvSample、MazeEnvRandom、MazeEnvPlus。

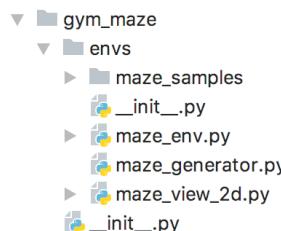


图 3-19 gym\_maze 环境库文件结构

### 3.3.1 迷宫生成算法

最常见的迷宫生成算法有递归回溯法（Recursive backtracker）、随机克鲁斯卡尔法（Randomized Kruskal's algorithm）、随机 Prim 法（Randomized Prim's algorithm）、递归分治法（Recursive division method）、威尔森法<sup>[18]</sup>（Wilson's algorithm）。

我选择的是递归回溯法，它实现形式简单、经典，可以生成每个区域都连通、不存在环的完美迷宫。迷宫生成模块作为成员函数，在类 Maze 中实现。算法流程如下：

- a. 初始化迷宫，所有栅格都被墙隔开，互不相通；
- b. 随机选择一个单元格作为当前栅格，标记该栅格为已访问；
- c. 重复步骤 d, e 直到所有栅格都被访问（即栈空），则算法结束；
- d. 选择当前栅格的一个随机方向的相邻栅格，若该相邻栅格不曾被访问，当前栅格入栈，打通当前栅格与选择的相邻栅格间的墙壁，标记当前栅格与选择的相邻栅格已访问；
- e. 若栈非空，将栈顶栅格弹出，作为当前栅格。

```
def _generate_maze(self):
    # 初始化迷宫
    self.maze_cells = np.zeros(self.maze_size, dtype=int)
    # 随机选择一个单元格作为起始点，标记该栅格为已访问
    current_cell = (random.randint(0, self.MAZE_W-1), random.randint(0, self.MAZE_H-1))
    num_cells_visited = 1
    cell_stack = [current_cell]

    # 直到所有栅格都被访问（即栈空），则算法结束
    while cell_stack:
        # 将栈顶栅格弹出，作为当前栅格
        current_cell = cell_stack.pop()
        # x0, y0 为当前栅格的坐标
        x0, y0 = current_cell

        # 寻找当前栅格的所有相邻栅格
        neighbours = dict()
        for dir_key, dir_val in self.COMPASS.items():
            x1 = x0 + dir_val[0]
            y1 = y0 + dir_val[1]
            # 检测相邻栅格是否超出边界
            if 0 <= x1 < self.MAZE_W and 0 <= y1 < self.MAZE_H:
                # 检测栅格墙壁是否已被打通（是否已被访问）
                if self.all_walls_intact(self.maze_cells[x1, y1]):
                    neighbours[dir_key] = (x1, y1)

        # 若存在相邻栅格
        if neighbours:
            # 选择当前栅格的一个随机方向的相邻栅格
            dir = random.choice(tuple(neighbours.keys()))
            # x0, y0 为所选相邻栅格的坐标
            x1, y1 = neighbours[dir]
            # 打通当前栅格与选择的相邻栅格间的墙壁
            self.maze_cells[x1, y1] = self._break_walls(self.maze_cells[x1, y1], self._get_opposite_wall(dir))
            cell_stack.append(current_cell)
            cell_stack.append((x1, y1))
            num_cells_visited += 1
```

图 3-20 gym 环境的迷宫生成实现代码

对于转移栅格的处理：

- a. 检查转移栅格组数目是否合法，确定每组为一对转移栅格
- b. 在起点（下标记为 0）、终点（下标记为 n\*n-1，n 为迷宫规格）以外的栅格中随机选择转移栅格，记录这些栅格下标
- c. 将记录的栅格下标随机分组，一组表示一对转移栅格
- d. 类 Portal 示例化转移栅格对象，记录每组转移栅格的坐标，并将对象存储到类 Maze 成员字典中，用于下一步迷宫的绘制

```

def __set_random_portals(self, num_portal_sets, set_size=2):
    # num_portal_sets:转移栅格组数, set_size:一组转移栅格为两个栅格
    num_portal_sets = int(num_portal_sets)
    set_size = int(set_size)
    # 保证转移栅格数目不大于所有栅格数
    max_portal_sets = int(self.MAZE_W * self.MAZE_H / set_size)
    num_portal_sets = min(max_portal_sets, num_portal_sets)
    # 随机选择 num_portal_sets*set_size 个栅格作为转移格, cell_ids 记录所选栅格下标
    cell_ids = random.sample(range(1, self.MAZE_W * self.MAZE_H - 1), num_portal_sets*set_size)

    for i in range(num_portal_sets):
        # 将 cell_ids 中的栅格随机分成 set_size 组
        portal_cell_ids = random.sample(cell_ids, set_size)
        portal_locations = []

        for id in portal_cell_ids:
            # 从 cell_ids 中取出已分好组的栅格下标, 表示为栅格坐标
            cell_ids.pop(cell_ids.index(id))
            x = id % self.MAZE_H
            y = int(id / self.MAZE_H)
            portal_locations.append((x,y))

        # 将转移栅格存储到类成员中, 用于下一步的绘制
        # __portals 存储Portal实例, __portals_dict 存储的是<Portal实例, 坐标>的字典
        portal = Portal(*portal_locations)
        self.__portals.append(portal)
        for location in portal_locations:
            self.__portals_dict[location] = portal

```

图 3-21 gym 环境的转移栅格处理代码

### 3.3.2 迷宫环境绘制

环境的绘制借助了库 pygame，用到了 pygame 的方法：绘制点 pygame.draw.circle()、绘制线 pygame.draw.line()、绘制栅格 pygame.draw.rect()、绘制背景 pygame.Surface.fill()、初始化 pygame.init()、结束 pygame.quit()等。

```

class MazeView2D:

    def __init__(self, maze_name="Maze2D", maze_file_path=None,
                 maze_size=(30, 30), screen_size=(600, 600),
                 has_loops=False, num_portals=0):
        # 游戏窗口名称、使用 pygame 自带的时钟
        pygame.init()
        pygame.display.set_caption(maze_name)
        self.__game_over = False

        # 加载迷宫环境
        if maze_file_path is None:
            self.__maze = Maze(maze_size=maze_size, has_loops=has_loops, num_portals=num_portals)
        else:
            ...

        self.maze_size = self.__maze.maze_size

        # 起点、终点、智能体初始位置
        self.__entrance = np.zeros(2, dtype=int)
        self.__goal = np.array(self.maze_size) - np.array((1, 1))
        self.__robot = self.entrance

        # 游戏窗口大小
        self.screen = pygame.display.set_mode(screen_size)
        self.__screen_size = tuple(map(sum, zip(screen_size, (-1, -1)))))

        # 窗口层。更新时只更新表层，不直接更新背景层，减少开销
        self.background = pygame.Surface(self.screen.get_size()).convert()
        self.background.fill((255, 255, 255))
        self.maze_layer = pygame.Surface(self.screen.get_size()).convert_alpha()
        self.maze_layer.fill((0, 0, 0, 0))

        # 迷宫绘制
        self.__draw_maze()

        # 转移栅格上色
        self.__draw_portals()

        # 智能体绘制
        self.__draw_robot()

        # 起点上色
        self.__draw_entrance()

        # 终点上色
        self.__draw_goal()
    
```

图 3-22 gym 环境的迷宫绘制代码

### 3.3.3 重写 gym 方法

利用 gym 环境模拟强化学习问题，使用了强化学习问题的通用框架，类 MazeEnv 实现了迷宫寻路这一强化学习问题的逻辑控制。

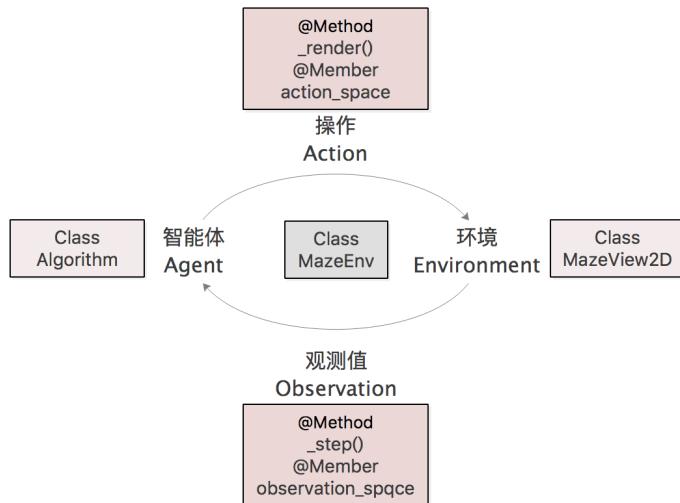


图 3-23 借助 gym 环境打造的强化学习框架逻辑

类 `MazeEnv` 作为强化学习框架的逻辑如图 3-y。迷宫环境下强化学习框架的搭建需要定义的元素：操作空间、状态空间、奖赏函数、智能体与环境的交互函数：

#### a. 操作空间、状态空间的表示

操作空间 `action_space = {"N", "S", "E", "W"}`，规模为 4；状态空间 `observation_space = {(0, 0) ~ (n, n)}`，以  $(0, 0)$  为左上角起点栅格坐标， $(n, n)$  为右下角终点栅格坐标建立迷宫栅格坐标系，用  $(x, y)$  代表当前智能体所在栅格的坐标（图 3-25）。

```
# 操作空间规模为4: ["N", "S", "E", "W"]
self.action_space = spaces.Discrete(4)
# 状态空间规模为(0, 0)至(n, n)，代表当前智能体所在栅格的坐标(x, y)
low = np.zeros(len(self.maze_size), dtype=int)
high = np.array(self.maze_size, dtype=int) - np.ones(len(self.maze_size), dtype=int)
self.observation_space = spaces.Box(low, high)
```

图 3-24 操作空间与状态空间的表示

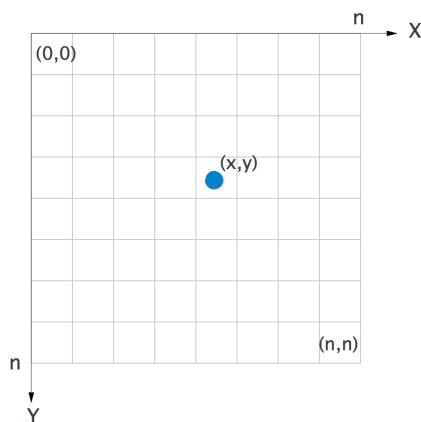


图 3-25 迷宫栅格坐标系示意图

#### b. 奖赏函数

奖赏的设置是算法的核心，如何去设置奖赏函数，将直接决定智能体如何选择下一

步的操作。如果能够做到最优解时大幅度增加奖赏值，则可以帮助模型快速收敛。如何针对求解一个实际的强化学习问题给出优秀的回报函数是一个值得研究的问题，加速模型收敛比起通用的回报函数来的更为有效，可惜在这方面的研究结果并不多。有研究证明了奖赏的选取对于简单的强化学习算法在运行时间上具有相当强的影响力<sup>[19]</sup>。我将在第四章详细探究奖赏函数的选取对算法收敛速度的影响。

#### c. 智能体与环境的交互函数

包括 Class MazeEnv 重写父类 Class gym.Env 的接口函数:\_step()、\_render()、\_reset()，以及制造随机数字的时间种子初始化函数\_seed()。

```
def _step(self, action):
    # 智能体依照所选的操作移动
    if isinstance(action, int):
        self.maze_view.move_robot(self.ACTION[action])
    else:
        self.maze_view.move_robot(action)

    # 到达终点，奖赏为1，回合结束，否则奖赏为负
    if np.array_equal(self.maze_view.robot, self.maze_view.goal):
        reward = 1
        done = True
    else:
        reward = -0.1/(self.maze_size[0]*self.maze_size[1])
        done = False

    self.state = self.maze_view.robot
    info = {}
    # 返回值：下一状态，奖赏，回合结束判断标志，调试信息（未启用）
    return self.state, reward, done, info
```

图 3-26 \_step()函数实现

```
def _render(self, mode="human", close=False):
    # 如果训练结束，关闭 pygame
    if close:
        self.maze_view.quit_game()
    # 如果训练未结束，返回更新后的画布
    return self.maze_view.update(mode)
```

图 3-27 \_render()函数实现

```
def _reset(self):
    self.maze_view.reset_robot()
    # 当前状态回到初始状态 (0, 0)
    self.state = np.zeros(2)
    self.done = False
    return self.state
```

图 3-28 \_reset()函数实现

```
def _seed(self, seed=None):
    self.np_random, seed = seeding.np_random(seed)
    return [seed]
```

图 3-29 \_seed()函数实现

#### d. 注册自定义环境为 gym 环境

Class MazeEnvSample、MazeEnvRandom、MazeEnvPlus 都继承自类 MazeEnv，目的是需要在自定义的环境库 gym-maze 的注册文件 `__init__.py` 中注册。注册时只需要提供两个参数（图 3-30）：id：环境名‘class\_name’；entry\_point：该环境指向的环境类所在位置。

注册后的环境类可以在代码中使用 `gym.make('class_name')` 调用。Sample、Random、Plus 标识了该迷宫环境的模式，nxn 标识了迷宫尺寸，这种调用方式采用了“低内聚高耦合”的思想，减少了调用环境需要的参数，不需要重复修改代码，同时可以随时增减所需要的类环境，保证了 gym 环境的接口统一和完整。

```

class MazeEnv(gym.Env):
    ...
class MazeEnvSample5x5(MazeEnv):
    ...
class MazeEnvRandom5x5(MazeEnv):
    ...
class MazeEnvSample10x10(MazeEnv):
    ...
class MazeEnvRandom10x10(MazeEnv):
    ...
class MazeEnvSample100x100(MazeEnv):
    ...
class MazeEnvRandom100x100(MazeEnv):
    ...
class MazeEnvRandom10x10Plus(MazeEnv):
    ...
class MazeEnvRandom20x20Plus(MazeEnv):
    ...
class MazeEnvRandom30x30Plus(MazeEnv):
    ...

```

图 3-30 gym\_maze 需要注册的类（部分）

```

register(
    id='maze-sample-5x5-v0',
    entry_point='gym_maze.envs:MazeEnvSample5x5',
)
register(
    id='maze-random-5x5-v0',
    entry_point='gym_maze.envs:MazeEnvRandom5x5',
)

```

图 3-31 gym\_maze 的类注册代码（部分）

## 3.4 本章小结

本章介绍了迷宫寻路实验环境的搭建过程，包括自动寻路算法和迷宫环境，详细描述了三种强化学习算法的实现和实验环境的功能与实现。强化学习算法包括基于表格的 Q 算法和 S 算法，以及基于神经网络的 DQN 算法；迷宫环境基于 OpenAI Gym，为实验提供一个直观的可视化环境。

## 第四章 实验结果思考

本章将对实现的三种强化学习算法进行参数优化调节、性能对比。4.1 小节探究学习率  $\alpha$ 、折扣因子  $\gamma$ 、奖赏函数设置、Q 表初始值如何影响 Q 算法和 S 算法的收敛速度；4.2 小节探究 DQN 算法中学习率  $\alpha$ 、折扣因子  $\gamma$ 、奖赏函数设置、TargetNet 更新步数、batch\_size 对算法表现存在何种影响；4.3 小节是实验总结与思考。

### 4.1 参数对 Q-Learning 算法和 SARSA 算法收敛速度的影响

本小结研究的参数包括：学习率  $\alpha$ 、折扣因子  $\gamma$ 、奖赏函数设置、Q 表初始值设置对两种强化学习算法收敛速度的影响。收敛的评判标准为每次探索到达终点使用的步数趋于定值，速度的的评判标准为算法达到收敛花费的回合数。实验结果图像的绘制使用 python 的 matplotlib 库。

算法参数设置：

- a. 学习率： $\alpha \in [0, 1]$ ;
- b. 折扣因子： $\gamma \in [0, 1]$ ;
- c. 奖赏函数设置：终点奖赏、移动奖赏
- d. Q 表初始值：初始化为非零随机值或统一初始化为零
- e. 贪婪因子设置方式不变： $\epsilon$  初始值为 1，贪婪折扣  $\text{epsilon\_decay} = 0.9995$ ；每个回合结束后， $\epsilon$  乘以  $\text{epsilon\_decay}$ ，期间若发现探索到终点，令  $\epsilon = 0$ （实验中发现这是一种确保收敛的  $\epsilon$  调参方式）。

算法的迷宫环境设置：

- a.  $\text{maze\_size} = 10 \times 10$ ：实验结果如图 4-1。总训练回合数为 500，每回合接受最大耗时步数为 1000，若耗时步数达 1000 步，强制结束本回合。
- b.  $\text{maze\_size} = 30 \times 30$ ：实验结果如图 4-2。最大回合数设为 1000，每回合接受最大耗时步数为 1000，若耗时步数达 1000 步，强制结束本回合。
- c.  $\text{maze\_size} = 50 \times 50$ ：实验结果如图 4-3。最大回合数设为 2000，每回合接受最大耗时步数为 1000，若耗时步数达 1000 步，强制结束本回合。

### 4.1.1 学习率 $\alpha$

$Q$  值每次的更新值为学习率  $\alpha$  乘以  $Q$  差距 ( $Q$  差距 =  $Q$  目标 -  $Q$  当前)，学习率  $\alpha$  决定了本次学习对  $Q$  差距的吸收程度。测试环境为地图相同的样例迷宫，大小  $maze\_size = 10 \times 10, 30 \times 30, 50 \times 50$ 。

学习率： $\alpha$  分别取  $0.1, 0.3, 0.5, 0.8$  (原计划的  $\alpha$  分别取  $0.001, 0.1, 0.5$ ，后续实验中发现对尺寸稍大的迷宫 ( $50 \times 50$ )， $\alpha=0.1$  很大程度地减小了收敛速度)；

其余参数相同： $\gamma=0.5$ 、终点奖赏  $reward=1$ 、移动奖赏  $reward=-1/maze\_size^2$ 、 $Q$  表初始值统一初始化为零、贪婪因子  $\epsilon$  初始值为 1，折扣  $epsilon\_decay=0.9995$ 。

图中横轴为训练回合数，纵轴为每回合花费的步数， $Q$  算法的数据使用红色曲线标注， $S$  算法的数据使用蓝色数据标注。图 4-1 a) 的六张结果的横纵坐标的比例尺相同，图 4-1 b)、图 4-2 中  $\alpha=0.1$  情况下的横纵坐标的比例尺与其余五张结果稍有不同（横坐标的最大回合数分别为 2000、4000，而其余分别为 1000、2000）。

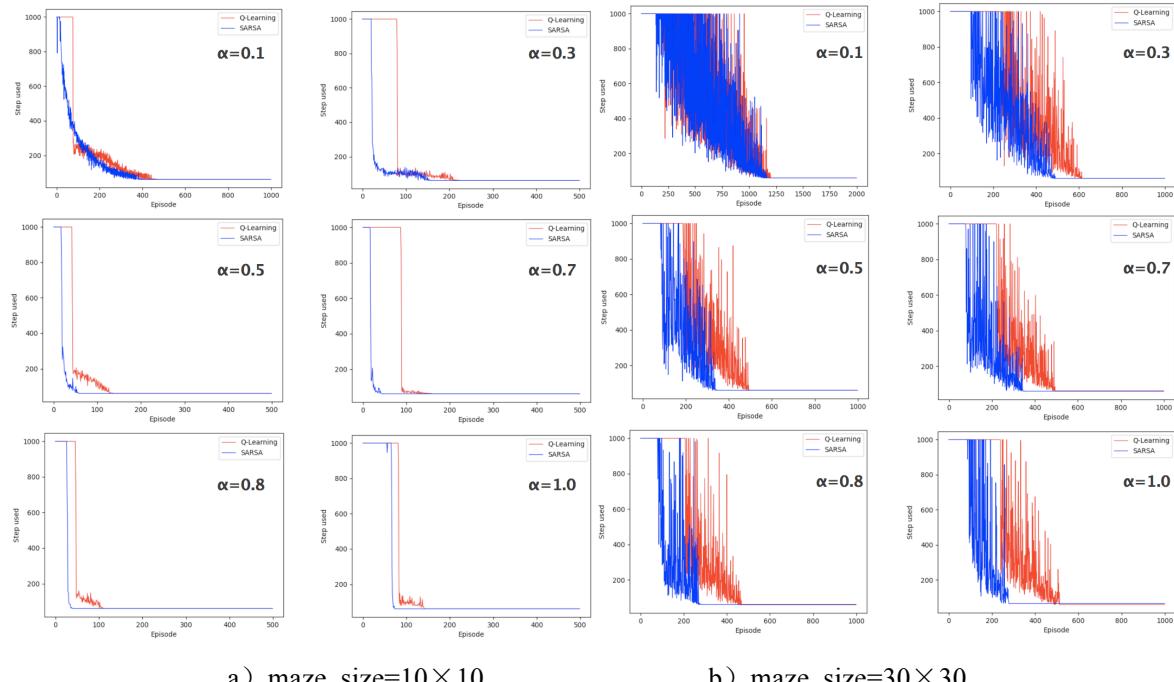
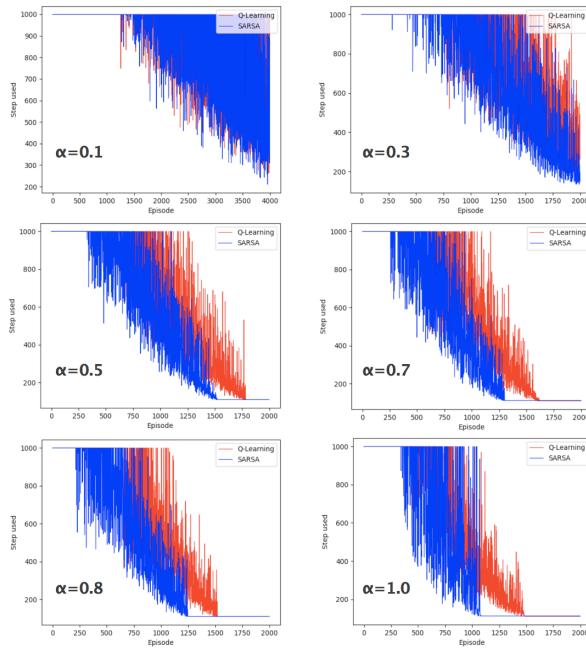


图 4-1  $\alpha$  实验结果

图 4-2  $\alpha$  实验结果 ( $maze\_size=50 \times 50$ )

整体上，实验显现出几个相同的趋势：

- 显然地，迷宫尺寸越大，算法的收敛发生得越迟；
- 随着  $\alpha$  增大，每回合花费步数的震动幅度减小，体现为曲线宽度减小；
- $\alpha$  对收敛速度的影响是非线性的，当  $\alpha$  越小对收敛速度的延缓效果越大；
- S 算法收敛的发生早于 Q 算法，且它们的图像弧度相似，Q 算法像是滞后的 S 算法。

适当增大  $\alpha$  可以减小训练过程中的波动情况，在迷宫环境中，局部最小值的问题似乎并没有那么重要（因为最优解相差的步数不大）。实验环境中，S 算法的收敛速度大于 Q 算法，但优势 Q 算法却可能找到一个优于 S 算法的最优解，尽管最优解之间的差距十分小。最后，本小节的结论是  $\alpha$  在 0.5~0.8 的区间内取值是一个比较靠谱的选择。

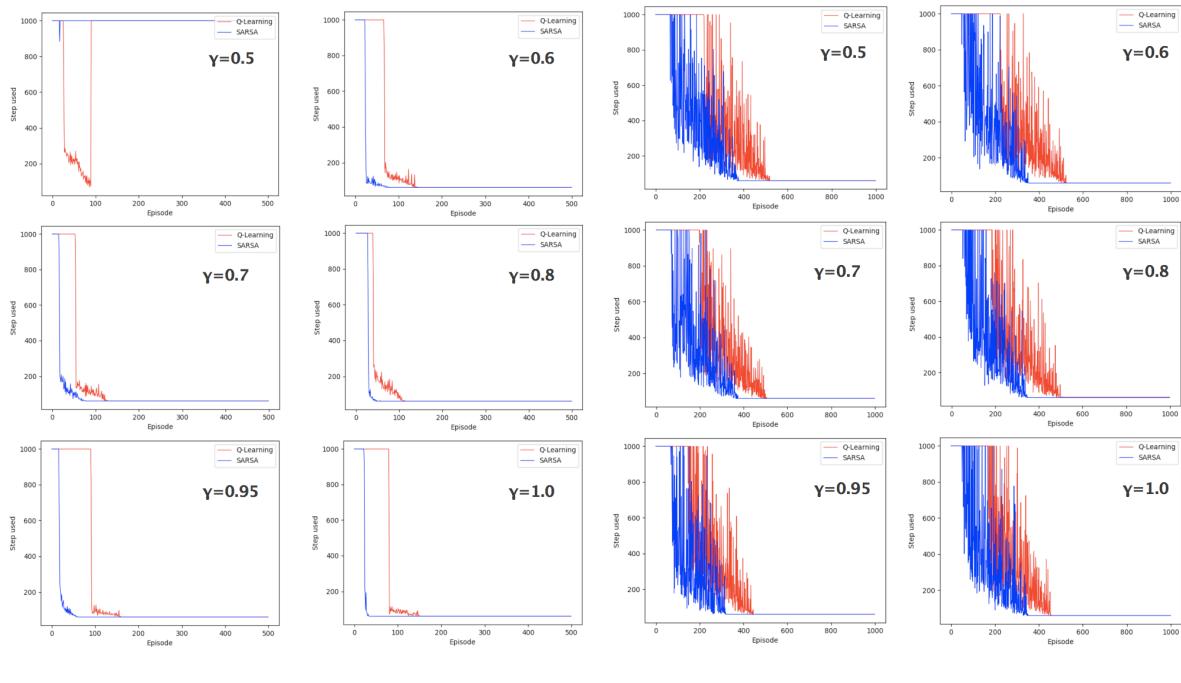
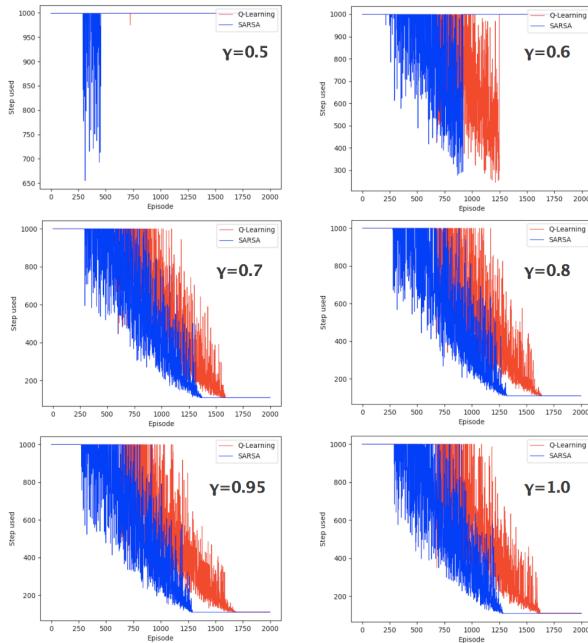
### 4.1.2 折扣因子 $\gamma$

在 Q 值更新式中，折扣因子  $\gamma$  乘在  $Q_{\text{Next}}$  之前， $\gamma$  乘以  $Q_{\text{Next}}$  与奖赏  $R$  之和一起构成 Q 目标， $\gamma$  值越小，则对  $Q_{\text{Next}}$  的关注度越小，奖赏  $R$  在 Q 目标中的占比越大。所以  $\gamma$  决定了对下一状态 Q 值的关心程度。测试环境为地图相同的样例迷宫，大小  $maze\_size = 10 \times 10, 30 \times 30, 50 \times 50$ 。

折扣因子： $\gamma$  分别取 0.1, 0.3, 0.5, 0.8；其余参数相同：由 5.1.1 实验结果，本处取

$\alpha = 0.7$ 、终点奖赏 reward = 1、移动奖赏 reward =  $-1/maze\_size^2$ 、Q 表初始值统一初始化为零、贪婪因子  $\epsilon$  初始值为 1，折扣 epsilon\_decay=0.9995。

图中横轴为训练回合数，纵轴为每回合花费的步数，横纵坐标的比例尺相同。Q 算法的数据使用红色曲线标注，S 算法的数据使用蓝色数据标注。

a)  $maze\_size=10 \times 10$ b)  $maze\_size=30 \times 30$ 图 4-3  $\gamma$  实验结果图 4-4  $\gamma$  实验结果 ( $maze\_size=50 \times 50$ )

整体上，实验显现出几个相同的趋势：

- a.  $\gamma = 0.5$  时， $10 \times 10$  和  $50 \times 50$  的迷宫找不到最优解； $\gamma = 0.6$  时， $50 \times 50$  的迷宫找不到最优解，因此  $\gamma$  取值不能过小，适当增大  $\gamma$  可以确保算法收敛到最优解；
- b.  $\gamma$  取值增大，算法波动的振幅减小，但不明显；
- c. 当  $\gamma$  取值在范围  $0.8 \sim 1.0$  内，对算法收敛速度的影响区别不明显；

适当增大  $\gamma$  可以减小训练过程中的波动情况。给出对  $\gamma$  过小时算法无法收敛到最优解的推测：由于折扣因子过小，随着步数增加，后续状态的 Q 值不断被  $\gamma$  打折，直接探索到终点过程中步数花费代价比终点的 Q 值还要小，算法认为与其费力不讨好接近终点，不如自我了断（进入迷宫的“死胡同”结束回合）。

$\gamma$  实验中，S 算法的收敛速度仍然大于 Q 算法。最后，本小节的结论是对于迷宫环境， $\gamma$  在  $0.8 \sim 1.0$  的区间内取值是一个比较靠谱的选择。

### 4.1.3 奖赏函数设置

奖赏是人为设定的，为了引导智能体做出我们期待它做出的行为，在迷宫问题中，终点的奖赏是唯一的正奖赏，因为算法希望智能体向终点移动；每次移动的奖赏为负奖赏，因为算法希望智能体花费更少的步数，多余的移动被视作是惩罚。

奖赏函数可为常量函数，例如实验 5.1.1、5.1.2 设置终点奖赏  $reward = 1$ 、移动奖赏  $reward = -1/maze\_size^2$ ；同时，也可以是一个带有变量的函数。

对于上面的两种奖赏函数设置方式：

- a. 终点奖赏始终为 1，移动奖赏与迷宫尺寸相关；
- b. 终点奖赏始终为 1，移动奖赏与迷宫尺寸无关；
- c. 终点奖赏为 1，移动奖赏设为智能体当前坐标  $(x, y)$  乘积的负倒数（或智能体与终点坐标的曼哈顿距离的负倒数），目的是增加终点附近栅格的奖赏，让智能体有向终点附近移动的倾向；

实验中奖赏设置为变量，设置情况如表 4-1；其余参数相同：由 5.1.1、5.1.2 实验结果，本处取  $\alpha = 0.7$ 、 $\gamma = 0.95$ 、Q 表初始值统一初始化为零、贪婪因子  $\epsilon$  初始值为 1，折扣  $epsilon\_decay = 0.9995$ 。

表 4-1 实验奖赏设置

	终点奖赏	移动奖赏
<b>a</b>	R = 1	R = - 0.1/ maze_size
	R = 1	R = - 1/ maze_size
	R = 1	R = - 1/ maze_size <sup>2</sup>
	R = 1	R = - 1/ maze_size <sup>3</sup>
<b>b</b>	R = 1	R = - 0.001
	R = 1	R = - 0.01
	R = 1	R = - 0.1
<b>c</b>	R = 1	R = - 1/ robot.x*robot.y or - 1/dis(robot, goal)

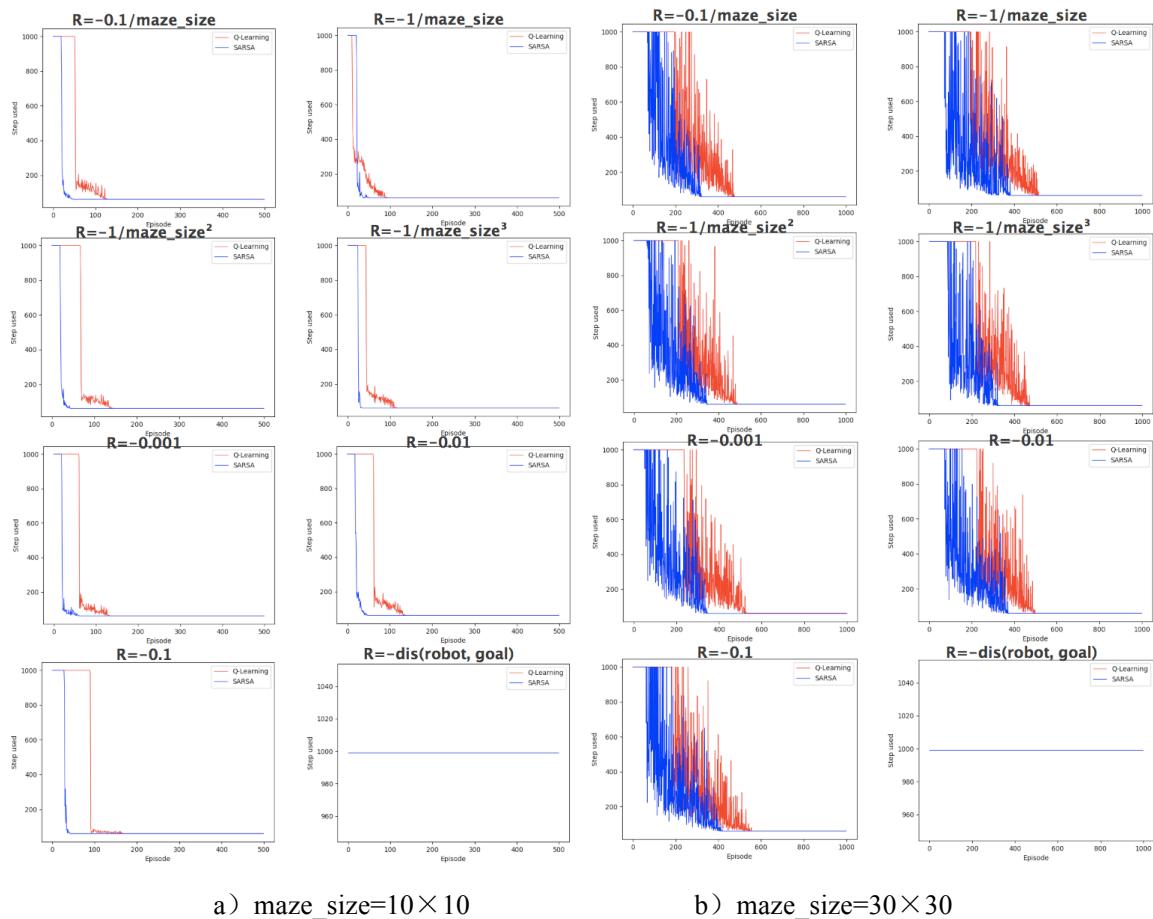
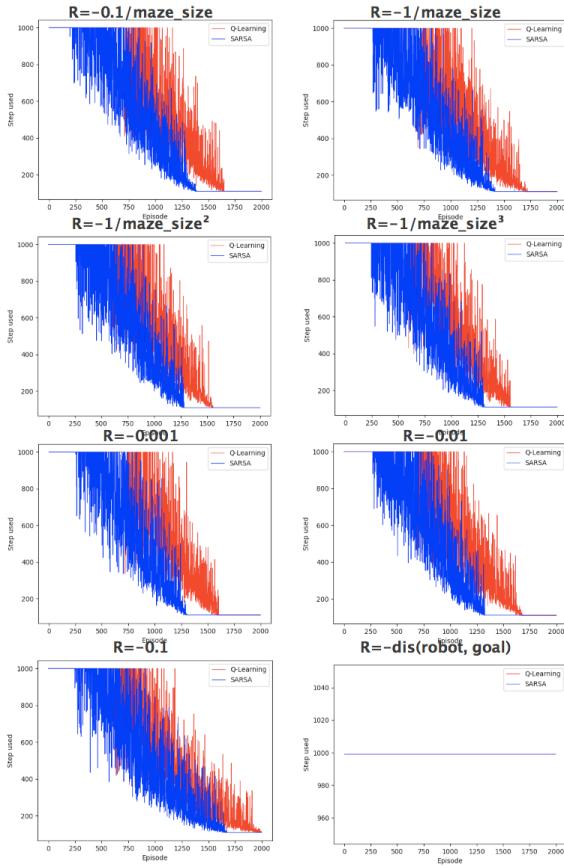


图 4-5 奖赏函数实验结果


 图 4-6 奖赏函数实验结果 ( $\text{maze\_size}=50 \times 50$ )

对奖赏函数设置的总结：

- 奖赏函数与迷宫尺寸相关的设置方式略微优于与迷宫尺寸无关的设置方式；
- 增大终点奖赏与移动奖赏比值的绝对值有利于加速收敛；
- 取  $R=-0.001$  或  $R=-1/\text{maze\_size}^3$  效果较好；
- 将移动奖赏设置为与智能体当前坐标相关的函数，实验结果并不理想。

实验结论符合文献<sup>[20]</sup>，即离散状态空间的奖赏函数设置方式遵循公式(4-1)时表现较好：

$$\text{若奖赏函数为形式: } R(s, a, s') = \begin{cases} r_g & \text{若 } s' \text{ 为终点} \\ r_\infty & \text{否则} \end{cases}$$

$$\text{则对终点奖赏 } r_g \text{ 和移动奖赏 } r_\infty \text{ 的选取符合: } r_g \geq \frac{r_\infty}{1-\gamma} \quad (4-1)$$

对实验 c 糟糕表现的原因做出推想：在迷宫环境中，栅格属于离散状态，起点与终点之间的墙壁障碍太多，四个前进方向并无好坏之分，如果使用智能体与终点曼哈顿距离的负倒数或智能体当前点坐标( $x, y$ )乘积的负倒数，的确可以让智能体有着向终点方向移动的倾向，但是这种倾向基于更大概率地去选择向右和向下的操作，这在复杂的迷宫

环境里是很不利的：刚开始的探索时，所有状态 Q 值初值都是 0，智能体因为向右和向下的操作策略，迭代会导致向右和向下方向栅格的 Q 值更大，若走进“死胡同”，智能体将不愿意选择向上或向左此类 Q 值较小的“回头路”，因此总是掉入陷阱，拿不到正奖励。所以在迷宫问题的奖赏函数设置时，不应该人为地为智能体做出操作的预期打算，因为这种行为实质上已经干扰了 Q 表的迭代方向；反之，给予每个操作相同的奖赏，赋之以均等的探索机会，效果会更好。

#### 4.1.4 Q 表初始值

实验发现，Q 表初始值的设定对算法是否收敛起到极其关键的作用。回顾 2.1.3 回报  $G_t$ 、状态价值函数  $V^\pi(s)$ 、状态-操作价值函数  $Q^\pi(s, a)$  有定义与关系：

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \\ V_\pi(s) &= E_\pi[G_t | S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s] \\ Q_\pi(s, a) &= E_\pi[G_t | S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a] \\ V_\pi(s) &= \sum_{a \in A} \pi(a|s) * Q_\pi(s, a) \end{aligned}$$

由极限理论有：

$$\sum_{k=0}^{\infty} \gamma^k = 1 + \gamma + \gamma^2 + \dots = \frac{1}{1-\gamma} \quad (4-2)$$

如果在探索开始阶段，智能体没有到达过终点，不曾得到终点奖赏  $r_g$ ，那么有公式（4-3），将算法收敛得到的 Q 表值记为  $Q^\infty$ ，那么根据公式（4-1）（4-2）（4-3）可做推理（4-4）：

$$R_{t+k+1} = r_\infty \quad (4-3)$$

$$\begin{aligned} G_t &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \leq \sum_{k=0}^{\infty} \gamma^k r_\infty = r_\infty \sum_{k=0}^{\infty} \gamma^k = \frac{r_\infty}{1-\gamma} \\ E[G_t] &= V_*(s) = \max(Q_*(s, a)) = Q_\infty = \frac{r_\infty}{1-\gamma} \end{aligned} \quad (4-4)$$

同时将 Q 表初值记为  $Q_i$ ，算法探索开始阶段所有栅格 Q 值相同为  $Q_i$ ，如公式（4-5）：

$$\forall S \forall A, Q(S, A) = Q_i \quad (4-5)$$

结合 Q 算法迭代式（3-2）与公式（4-4）（4-5）做变形：

$$\begin{aligned}
 Q(S, A) &\leftarrow Q_i + \alpha(r_\infty + \gamma Q_i - Q_i) \\
 &= Q_i + \alpha(r_\infty + (\gamma-1)Q_i) \\
 &= Q_i + \alpha(1-\gamma)(Q_\infty - Q_i)
 \end{aligned} \tag{4-6}$$

已知  $\alpha > 0$ ,  $1-\gamma > 0$ , 当  $Q_i > Q_\infty$  有  $\alpha(1-\gamma)(Q_\infty - Q_i) < 0$ , 一个状态一旦被探索过, 则该状态  $Q(S, A)$  值减小, 那么在选择下一探索状态时, 因为 Q 算法会选择 Q 值大的状态 (此处为没有被探索过的状态), 所以这种初始化方式会鼓励探索未知状态, 算法可以接近终点, 得到收敛; 当  $Q_i < Q_\infty$  有  $\alpha(1-\gamma)(Q_\infty - Q_i) > 0$ , 一个状态一旦被探索过, 则该状态  $Q(S, A)$  值增大, 那么在选择下一探索状态时, 因为 Q 算法会选择 Q 值大的状态 (此处为已经被探索过的状态), 这种初始化方式很少考虑探索未知状态, 一直在探索过的状态中打转, 算法永远到达不了终点, 无法收敛; 当  $Q_i = Q_\infty$  有  $\alpha(1-\gamma)(Q_\infty - Q_i) = 0$ , 一个状态一旦被探索过, 则该状态  $Q(S, A)$  值增大, 那么选择未被探索状态和已被探索状态的概率均等, 算法拥有到达终点的机会, 未必收敛。

实验中按表 4-2 控制 Q 表初始值为变量; 其余参数相同: 由 5.1.1、5.1.2、5.1.3 实验结果, 本处取  $\alpha = 0.7$ 、 $\gamma = 0.95$ 、终点奖赏  $r_g = 1$ 、移动奖赏  $r_\infty = -1/maze\_size^3$ 、贪婪因子  $\epsilon$  初始值为 1, 折扣  $epsilon\_decay = 0.9995$ 。

表 4-2 Q 表初始值设置

	收敛 Q 值 / $Q_\infty$	初始 Q 值 / $Q_i$
<b>a.</b> 初始化 Q 表为均一值	$Q_\infty = \frac{r_\infty}{1-\gamma}$	$Q_i > Q_\infty: Q_i = 0$
		$Q_i > Q_\infty: Q_i = 1$
		$Q_i = Q_\infty$
		$Q_i < Q_\infty: Q_i = -1$
		$Q_i = random(0, 1)$
		$Q_i = -random(0, 1)$
<b>b.</b> 初始化 Q 表为随机值		

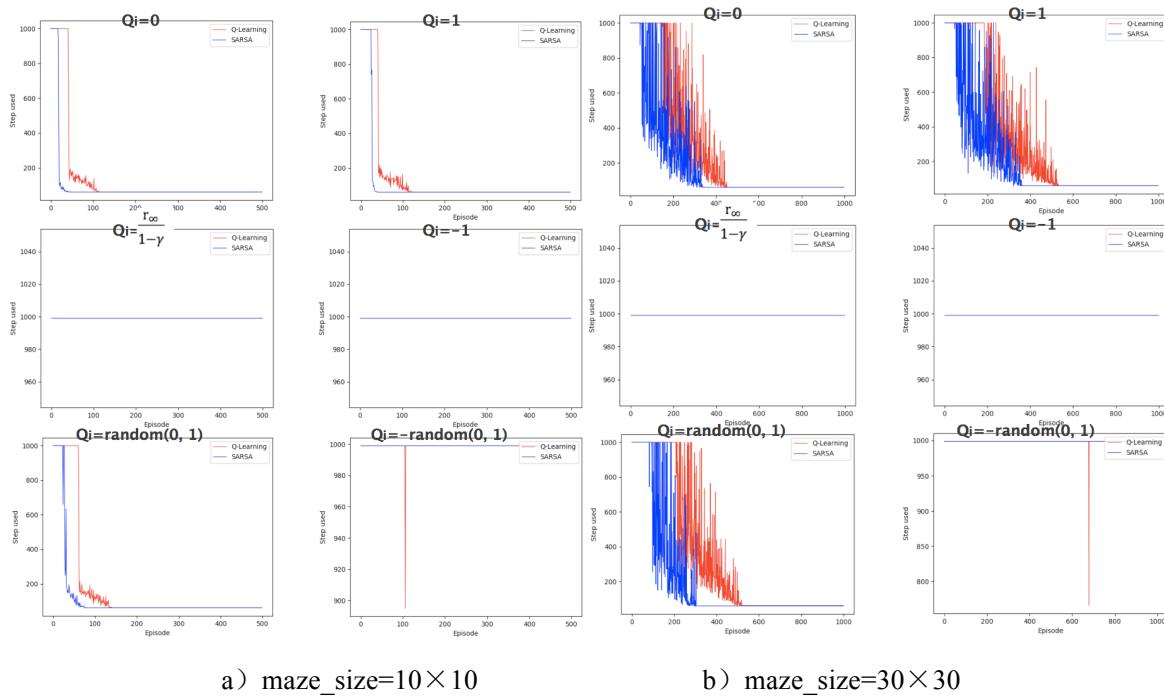
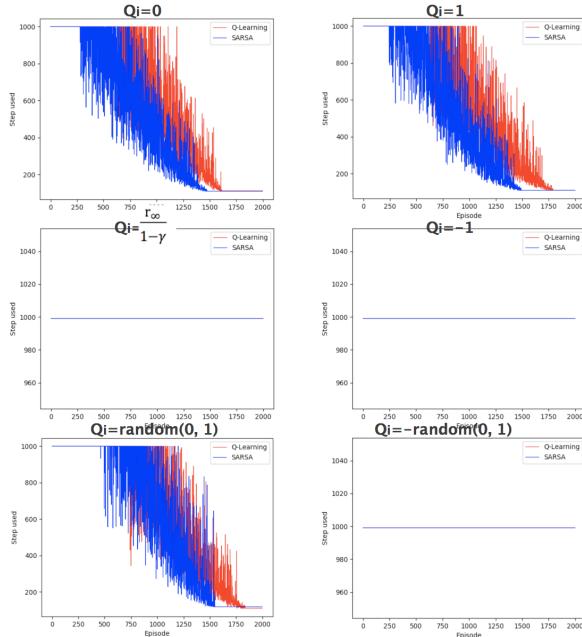


图 4-7 Q 表初始值设置实验结果

图 4-8 Q 表初始值设置实验结果 ( $\text{maze\_size} = 50 \times 50$ )

对 Q 表初值设置的总结:

- 保证  $Q^\infty < Q_i$  情况下, Q 表初值设置对 S 算法几乎无影响, 对 Q 算法影响较大;
- 随机初始化 Q 表值有机会让两种算法的收敛速度都提升, 并且就算降低收敛速度, 程度也不大;
- $Q^\infty < Q_i$  时, Q 表可以收敛到最优解,  $Q^\infty \geq Q_i$  时, Q 表无法收敛到最优解;

- d.  $Q^\infty < Q_i$  时，如果迷宫尺寸较小，探索过程中有可能到达过终点，但实验结果并不理想，难以收敛。

实验总结符合证明 (4-6)，若希望算法可收敛到最优解，对  $Q$  表初始值  $Q_i$  的设置须符合关系 (4-7)，同时，初始化  $Q$  表为随机值将利于模型的表现，这是一种推荐的初始化方式。

$$Q_i > Q_\infty = \frac{r_\infty}{1 - \gamma} \quad (4-7)$$

## 4.2 超参数对 DQN 算法收敛速度的影响

本小结思考超参数对迷宫问题的 DQN 算法收敛速度影响，包括：神经网络学习率 `learning_rate`（神经网络学习率的概念不同于  $Q$  算法）、折扣因子  $\gamma$ 、奖赏函数设置、TargetNet 更新步数 `update_frequency`。由于设计的的神经网络适用的输入向量尺寸介于 15~19，因此将实验环境设定为  $8 \times 8$  与  $9 \times 9$  大小的样例迷宫，转化后尺寸为  $17 \times 17$  和  $19 \times 19$ 。

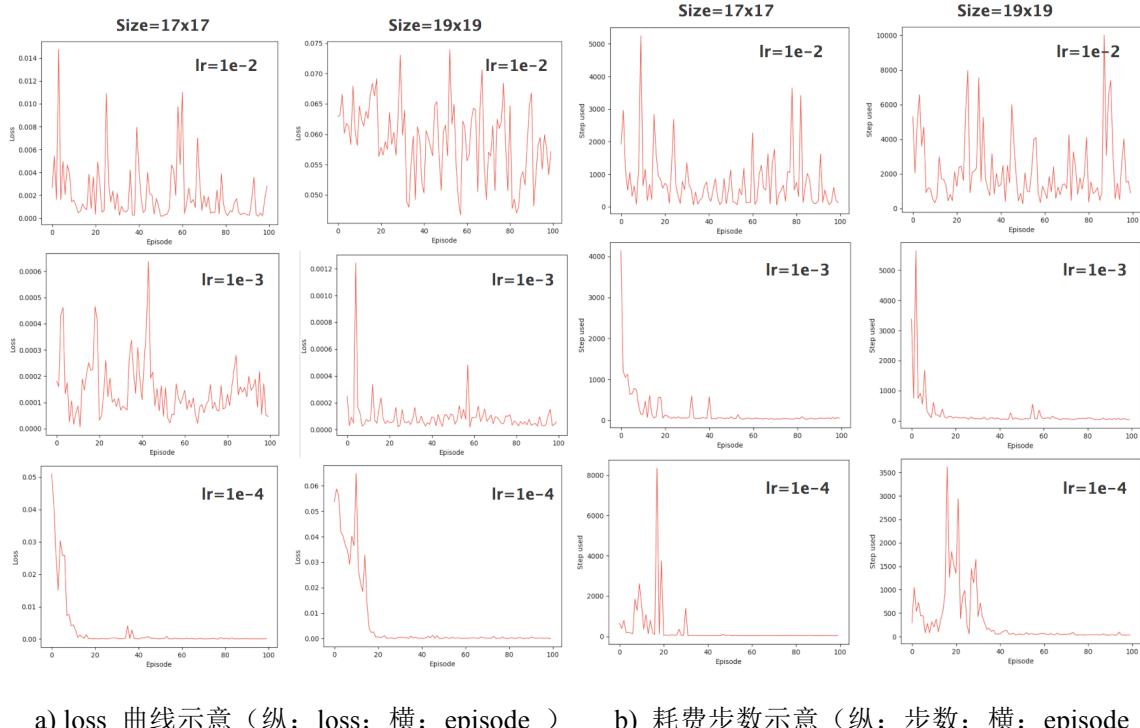
收敛的评判标准、速度的的评判标准与 5.1 相同，还考虑了训练过程中 loss 的变化。结果图像的绘制使用 python 的 `matplotlib` 库和 `tensorboard` 提供的可视化功能。

算法参数设置：学习率： $learning\_rate \in [0, 1]$ ；折扣因子： $\gamma \in [0, 1]$ ；奖赏函数设置：终点奖赏、移动奖赏； TargetNet 更新步数：每 `update_frequency` 步，将 MainNet 参数  $\theta$  更新到 TargetNet；经验池大小不变：`replay memory=10000`，保存 10000 个状态转换过程；批训练数据大小不变：`batch_size=32`；贪婪因子设置方式不变： $\epsilon$  初始值为 1， $\epsilon$  变化过程与训练回合数有关，每个训练回合结束后更新  $\epsilon = 1/(episode / 10 + 1)$ ，当训练回合数增加  $\epsilon$  会逐渐减小。

### 4.2.1 学习率 `learning_rate`

`maze_size = 8 × 8、9 × 9 的训练 loss 曲线于图 4-9 a)，步数耗费曲线于图 4-9 b)，分别为图左图右。总训练回合数为 100，每回合接受最大耗费步数为 1000，若耗费步数达 1000 步，强制结束本回合。若强制结束的回合连续总数大于 5，则默认耗时太久，结束训练。`

折扣因子  $\gamma = 0.95$ ; 终点奖赏  $R=+1$ , 移动奖赏  $R=-0.01$ ; 更新步数  $update\_frequency = 500$ ,  $batch\_size = 32$ 。实验考虑学习率  $learning\_rate$  分别等于  $0.01, 0.001, 0.0001$  的情况, 表 4-4 列出了训练所用时间与最优步数。



a) loss 曲线示意 (纵: loss; 横: episode )      b) 耗费步数示意 (纵: 步数; 横: episode )

图 4-9 DQN-learning\_rate 示意结果

表 4-3 训练所用时间与最优步数 (灰色代表效果不理想)

MazeSize	$learning\_rate=1e-2$	$learning\_rate=1e-3$	$learning\_rate=1e-4$
$17 \times 17$	final best step: 45	final best step: 28	final best step: 28
	total time used: 425	total time used: 340	total time used: 351
$19 \times 19$	best step: 275	best step: 37	best step: 36
	total time used: 2893	total time used: 506	total time used: 1839

对  $learning\_rate$  设置的结果总结:

- 实验中发现, 每次训练耗时的差距大, 究其原因, 猜测与前期的动作随机选择有关: 在探索过程中更早地选择一条正确的路, 耗费小于更早地选择了一条错误的路并在后期不断训练网络来修正它;
- 在不同的学习率下,  $19 \times 19$  的环境有时不会收敛, 原因与 b 相似, 前期的动作随机选择如果积累的抵达终点经验不够多, 则训练样本会被稀释, 此时再降低学习率很可能增大了与正确路线的偏离;
- 该神经网络在  $17 \times 17$  大小的迷宫环境下表现更好,  $learning\_rate=1e-3$  时, 模型

在两个迷宫环境的表现都比较好；

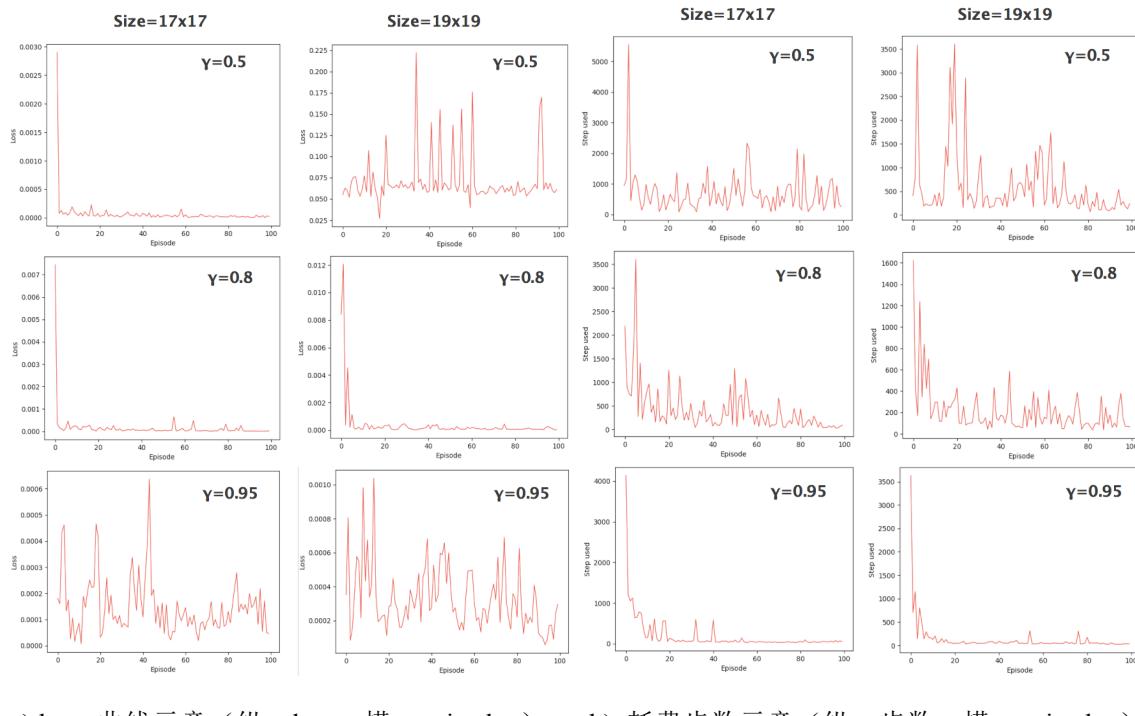
- d. 即使所有参数相同，每次训练耗费的时间差异也大，少数情况，在 $19 \times 19$  迷宫中可能出现不收敛，猜测是因为神经网络权重的随机初始化，Q 值初始值带来的差异。

神经网络在设计时经过一次很大的改动，首先采取的设计思路是仿照论文(Atari)，将输入状态 S 简单理解为智能体当前的状态（例如栅格坐标）而非迷宫环境的状态（整个迷宫作为输入），而输出为每个动作对应的 Q 值。但我忽略了非常重要的一点，Atari 实验中的状态都是连续的。迷宫实验过程中，我发现按照这种方式得到的模型就算在非常简单的环境下也无法收敛。造成这种结果可能是因为仅仅输入栅格坐标  $(x, y)$ ，那么神经网络对迷宫的认知就是局部的、离散的、不连贯的，后一状态的奖赏无法直观地作用在前一状态，提升整条路线的 Q 值。后面阅读了关于 MNIST 手写数字识别的论文，得到启发，将整个迷宫环境视作一张栅格图片作为输入，加入卷积神经网络，目的就是让神经网络对整个迷宫环境有全局的认识。

#### 4.2.2 折扣因子 $\gamma$

实验考虑折扣因子  $\gamma = 0.5, 0.8, 0.95$ 。 $\text{maze\_size} = 8 \times 8, 9 \times 9$  的训练 loss 曲线与耗步数结果如图 4-10，表 4-4 列出了训练所用时间与最优步数。

总训练回合数、每回合接受最大耗步数、结束回合、结束训练的条件与前面相同。遵循 5.1.1 的结果，在 $17 \times 17$  和 $19 \times 19$  的环境中学习率取  $\text{learning\_rate}=1\text{e-}3$ ，终点奖赏、移动奖赏、更新步数、batch\_size 与 5.1.1 相同，保持不变。



a) loss 曲线示意 (纵: loss; 横: episode )      b) 耗费步数示意 (纵: 步数; 横: episode )

图 4-10 DQN- $\gamma$  示意结果

表 4-4 训练所用时间与最优步数 (灰色代表效果不理想)

MazeSize	gamma=0.5	gamma=0.8	gamme=0.95
17×17	final best step: 88	final best step: 31	final best step: 28
	total time used: 1148	total time used: 607	total time used: 340
19×19	best step: 188	best step: 37	best step: 37
	total time used: 3093	total time used: 230	total time used: 506

对  $\gamma$  设置的结果总结：

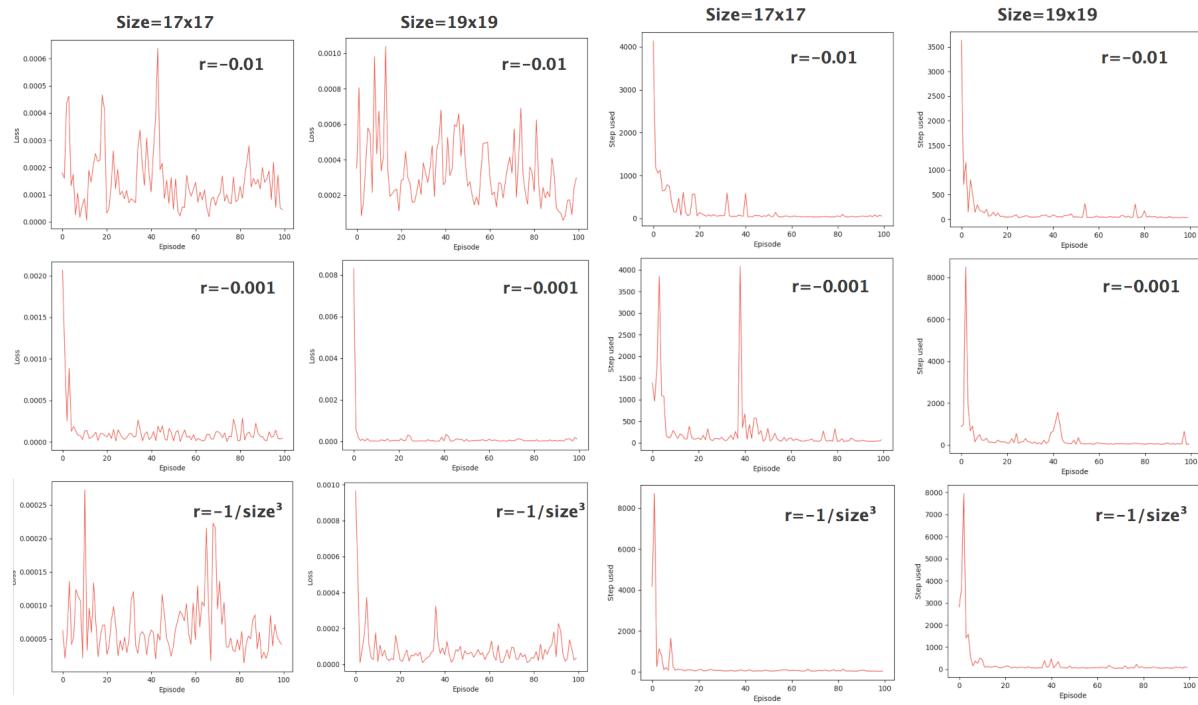
- 增大  $\gamma$  值可以加快整体的训练速度（因为加大了贪婪因子减小的速度），这与 5.1 中 Q 算法、S 算法的表现结论相同；
- 增大  $\gamma$  值，训练震荡振幅减小，原因在于大  $\gamma$  使得对终点奖赏的吸收程度更大，模型向终点奖赏靠近的趋势更大，减少了不必要的探索造成的步数耗费；
- 增大  $\gamma$  值，对  $19 \times 19$  的输入更有益，表现为  $19 \times 19$  输入的表现优于  $17 \times 17$  的表现，从表 4-2 的耗费步数与训练时间可以看出；

### 4.2.3 奖赏函数设置

实验考虑奖赏函数设置，与 5.1.3 思路相同，设置终点奖赏  $R$  始终为 +1，改变移动奖赏  $R = -0.01, -0.001, -1/maze\_size^3$ 。 $maze\_size = 8 \times 8, 9 \times 9$  的实验结果训练如图 4-11

a) b), 分别为 loss 曲线与耗费步数结果, 表 4-5 列出了训练所用时间与最优步数。

总训练回合数、每回合接受最大耗费步数、结束回合、结束训练的条件与前面相同。遵循 5.1.1 学习率取 learning\_rate=1e-3, 参照 5.2.2 折扣因子  $\gamma=0.95$ , 在  $17 \times 17$  和  $19 \times 19$  的环境中、更新步数、batch\_size 与 5.2.1 设置相同。



a) loss 曲线示意 (纵: loss; 横: episode )

b) 耗费步数示意 (纵: 步数; 横: episode )

图 4-11 DQN-奖赏函数示意结果

表 4-5 训练所用时间与最优步数 (黄色代表效果理想)

MazeSize	reward=-0.01	reward=-0.001	reward=-1/maze_size <sup>3</sup>
$17 \times 17$	final best step: 28	final best step: 28	final best step: 29
	total time used: 340	total time used: 425	total time used: 381
$19 \times 19$	best step: 36	best step: 37	best step: 36
	total time used: 167	total time used: 506	total time used: 487

对移动奖赏设置的结果总结:

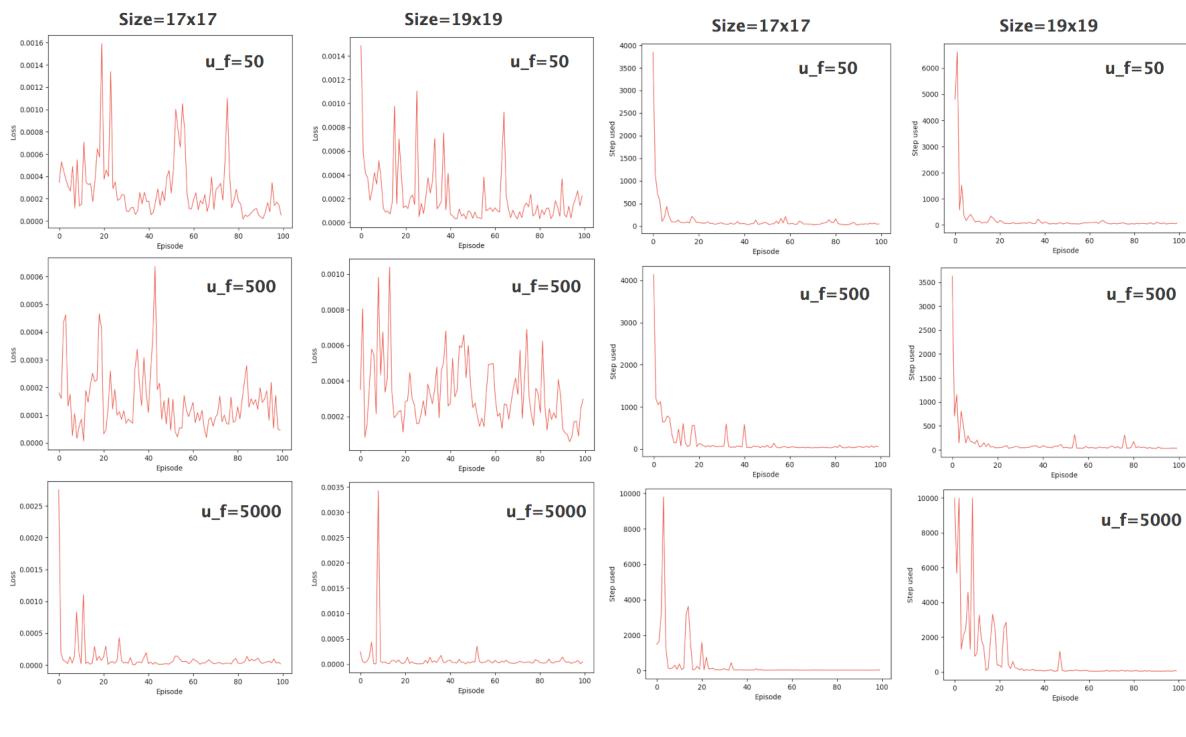
- 与 5.1.3 中 Q 算法、S 算法得到的结论不同, 在 DQN 法中增大终点奖赏与移动奖赏之比的绝对值, 并没有让算法表现得更好;
- 当移动奖赏  $R=-1/maze\_size^3$  时,  $17 \times 17$  和  $19 \times 19$  迷宫环境对应移动奖赏约等于 0.0001, 恰好约为  $R=-0.001$  的十分之一, 即当移动奖赏的绝对值已经很小时, 继续增大终点奖赏与移动奖赏之比的绝对值, 对算法性能的影响不大;

c. 移动奖赏  $R=-0.01$  的表现在  $17 \times 17$  和  $19 \times 19$  迷宫环境都较好，对  $19 \times 19$  迷宫环境训练时间的提升更大。

#### 4.2.4 TargetNet 更新步数

实验 TargetNet 更新步数设置，分别设置为  $update\_frequency=50, 500, 5000$ 。该参数决定经过多长时间更新一次 TargetNet 的参数  $\theta$ ，

$maze\_size = 8 \times 8, 9 \times 9$  的实验 loss 和耗费步数变化曲线如图 4-12，表 4-6 列出了迷宫训练所用时间与最优步数。总训练回合数、每回合接受最大耗费步数、结束回合、结束训练的条件与前面相同。折扣因子  $\gamma = 0.95$  参照 5.2.2，学习率取  $learning\_rate=1e-3$  遵循 5.1.1，在  $17 \times 17$  和  $19 \times 19$  的环境中、更新步数、batch\_size 与前相同。



a) loss 曲线示意（纵：loss；横：episode） b) 耗费步数示意（纵：步数；横：episode）

图 4-12 DQN-TargetNet 更新步数示意结果

表 4-6 训练所用时间与最优步数（黄色代表效果理想）

MazeSize	frequency=50	frequency=500	frequency=5000
$17 \times 17$	final best step: 28	final best step: 28	final best step: 28
	total time used: 210	total time used: 425	total time used: 427
$19 \times 19$	best step: 37	best step: 37	best step: 36
	total time used: 362	total time used: 506	total time used: 1042

对更新步数 `update_frequency` 设置的总结:

- a. 增大更新 TargetNet 参数  $\theta$  的频率（即减小 `update_frequency`）可以明显减少训练时间；
- b. 增大 `update_frequency` 减小了训练期间 loss 的均值，却增大了耗费步数的均值；
- c. 减小 `update_frequency` 可以同时提升模型在  $17 \times 17$  和  $19 \times 19$  迷宫的表现。

设置 TargetNet 本是为了破坏状态之间的相关性,让  $Q_{Target}$  的更新存在延时，作为监督数据，但在迷宫问题里，在不同的栅格，智能体的目标是不同的，变为找到从该栅格开始到终点的最短通路，此时上一状态的选择变得更为重要，因此需要一个与当前网络相近的结果训练局部的最优。不同于 Atari 游戏，迷宫问题对上一状态的依赖更为严重，因此作为监督数据的 TargetNet 更新不宜更久，要“紧跟时代”。

## 4.3 迷宫强化学习迷宫寻路的思考

经过了前文对 Q 算法、S 算法、DQN 算法在迷宫问题应用的研究，在本小节将对强化学习算法在迷宫环境的表现做出评价，论述迷宫强化学习算法的优势、劣势，并与其他的迷宫寻路算法做出比较。

### 4.3.1 强化学习迷宫寻路算法的表现

首先，在迷宫问题中，不管是何种大小的测试尺寸或测试参数，S 算法表现均优于 Q 算法：S 算法的收敛速度大于 Q 算法并且收敛过程中震荡小。在迷宫问题中，Q 算法的优势不明显，这是迷宫的结构决定的，因为迷宫能到达终点的通路只有一条，如果已经找到一条这样的路线，就没有必要探索岔路。而“勇敢好奇”的 Q 算法找到通路后还有很大地概率探索岔路，S 算法则“老老实实”地沿着已有的通路迭代，同时减少这条通路中的回头路，尽快趋向终点。

S 算法、Q 算法在 `maze_size=10×10、30×30、50×50` 达到收敛状态需要的训练时间分别约为 2s、17s、70s（注释掉所有输入输出语句后得到的平均值），可以保证找到最优路线，见图 4-13。

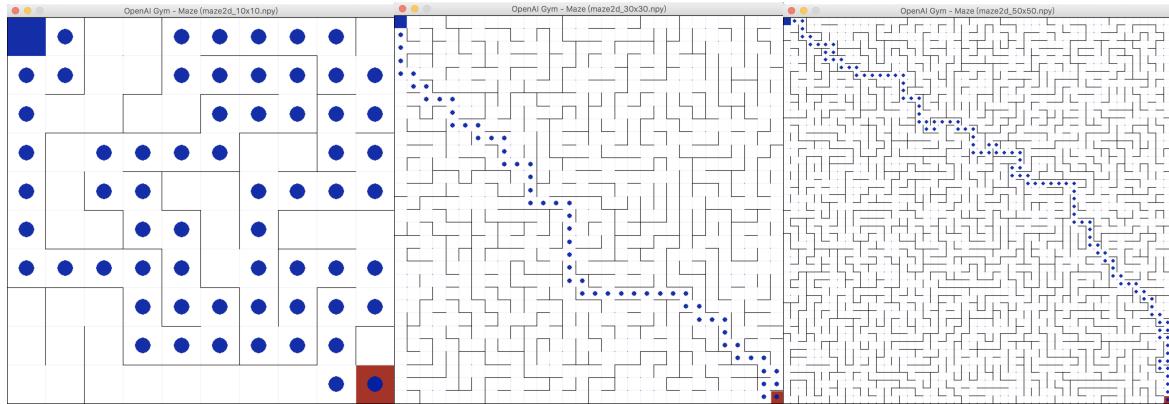


图 4-13 样例迷宫寻路结果（左至右依次为 10、30、50）

本文对 Q 算法、S 算法研究的突破在于对奖赏函数、Q 表初始值的思考，以往的强化学习只关注参数对学习效果的影响，先入为主地将奖赏函数、Q 表初始值作为算法的输入而非参数，缺乏对其设置原理的思考。在 4.1.3 解释了为什么在迷宫环境中均等的移动奖赏比依状态改变的移动奖赏更好，在 4.1.4 还发现在给定范围内随机地初始化 Q 表可以为算法带来惊喜的效果。奖赏函数和 Q 表初始值的研究资料不多，但随着对强化学习性能提升的需求，在未来一定是一个热点话题。

为了验证 Q 算法、S 算法的灵活性，在编写 gym 迷宫环境时还加入了转移栅格模式，不需要改动任何代码，Q 算法、S 算法可以直接应用到转移栅格迷宫里，随机采样的原理让它们在这类地图中的表现比在样例迷宫里的表现还要好。

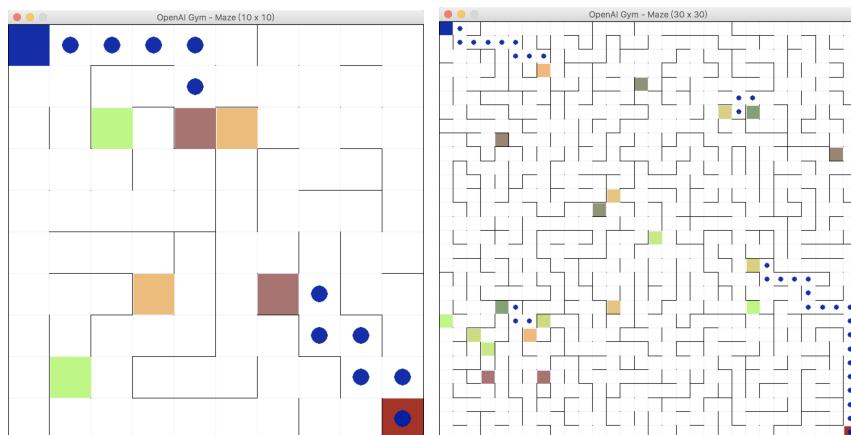


图 4-14 转移栅格迷宫寻路结果（左：10；右：30）

值得一提的是 Deep Q Network 算法在迷宫问题中使用的尝试，尽管效果没有想象中的那么好。最初设计的神经网络将智能体当前所在栅格坐标作为输入，结构如图 4-x，但这种结构效果并不好，就算是尺寸很小的迷宫也要花费很长的时间训练才能找到正确解。在这一设想失败后，我参照 MNIST 手写数字识别，将迷宫环境想象成为一张图，

把整个迷宫作为神经网络的输入，最终得解。

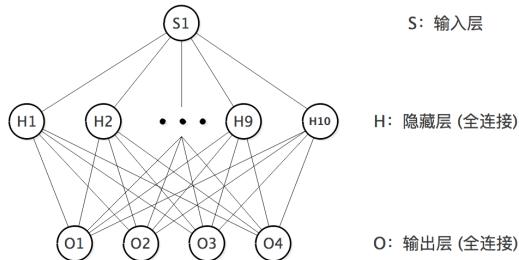


图 4-15 失败的神经网络结构

虽然问题得到解决，但网络表现却不尽人意：

- 网络输入需要整个迷宫状态，包括智能体当前位置、终点栅格、墙壁信息。那么，训练神经网络时已经将地图作为输入，就不适用并动态未知环境了；
- 此网络不能解决任意尺寸的迷宫，因为网络是针对特定的输入尺寸设计的，所有只能解决两种尺寸的迷宫问题， $17 \times 17$  和  $19 \times 19$ ；
- 网络的框架还是卷积神经网络，但在决策过程中用了 Q 算法的思想，且有强化学习与环境交互、奖赏的概念。

这让我不禁思考：DQN 能否成功应用到未知环境迷宫问题中？文章（Atari）提出的这些 DQN 使用场景都是些状态连续的单机游戏，当前状态与下一状态变化的差异并不大（图 4-16），除去背景、音效，剩余只是智能体与动态目标，环境对智能体的影响不大。但是在迷宫问题里，智能体进入下一栅格，四周的环境已经发生了彻底改变：墙壁、在全局的位置、与终点最近路线的偏移。如果说环境对 Atari 游戏的影响像是前进，那对迷宫问题的影响更像是闯关。迷宫太大了，神经网络不断地“遗忘”曾经经历，到达终点的经验太少了，但是 Atari 类的游戏不同，任意时刻的随机操作都有可能带来即时的正奖赏和成功经验。

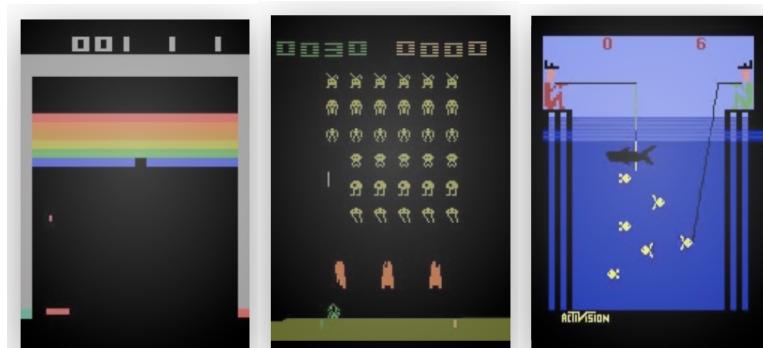


图 4-16 Atari 类游戏

本文迷宫问题里实现的 DQN 算法，虽然已经不是文章<sup>[9]</sup>中提出的形式，但是采取了经验池、TargetNet 结构，创新地结合卷积神经网络和 Q 算法决策方式，让 TargetNet 成为训练监督数据的来源，也能解决一些迷宫问题，这是一种新颖的解决思路，证明卷积神经网络应用的新一种可能性。

### 4.3.2 强化学习迷宫寻路与其他迷宫寻路算法

可以用于与 Q 算法、S 算法做比较的迷宫环境的最短路径算法可以分为两类：一是基本的深度优先搜索（DFS）、宽度优先搜索（BFS）、A\*搜索算法；二是近年的人工智能仿生算法如遗传算法、蚁群算法，模拟了自然界的规律。严格地说，DFS、BFS 只能算是图遍历算法，它们的工作机理是盲目地把地图的所有节点都搜一遍，直到找到终点，A\*搜索算法比 DFS、BFS 多了一个启发函数，让搜索过程更有目的性。启发函数的加入使 A\*搜索算法不断调整自己的目标，算法适用于动态未知环境。A\*搜索算法的启发函数是固定的，但 Q 算法、S 算法的价值函数却在与环境交互中迭代更新。

此外，本文实验中搭建的迷宫环境墙壁不占栅格（区别参见见图 3-8），如果想要在这种环境中使用 DFS、BFS、A\*，就要额外开辟一组空间记录每个栅格四面墙壁是否存在或者把迷宫二次转换为图 3-8 的形式。在求解迷宫最短路径问题上，Q 算法、S 算法相比上述三种算法在时间上并没有优势，但是它们不需要变动就可以解决转移栅格迷宫问题，而上述三种算法却不可以。这在一定程度上说明了强化学习算法更灵活、普适性更好，能拟合更多变的环境，这点品质是难能可贵的。

和 Q 算法、S 算法一样，遗传算法、蚁群算法也能用在动态未知的环境。这类人工智能算法都有蒙特卡洛性质，用随机方法探索环境，在模拟、尝试中学习和提升自己的表现，因为计算量大且算法有仿生性质，流行的研究趋势是加入神经网络结构。

## 4.4 本章小结

本章依托自定义的 gym 迷宫环境 gym\_maze 对三种强化学习算法进行实验。实验遵循控制变量原则，但是仍然有一下不足的地方：三种算法随机采样环节做不到完全相同，实验时多个程序并行也许会分享系统资源使各自性能不能充分展示，这些因素都可能会给算法性能带来影响。

总的来说，实验结果符合我对强化学习算法在迷宫寻路的心理预期。

## 结论

### 论文工作总结

本文采用 OpenAI gym 的强化学习框架，编写迷宫仿真环境进行强化学习寻路算法的研究，实现了 Q-Learning、SARSA、Deep Q Network 三种强化学习算法。经过 gym 仿真实验，进行了参数调优并讨论参数对算法性能的影响。迷宫的 DQN 算法结合了卷积神经网络和 Q 算法决策方式，为迷宫的深度强化学习贡献了一种解决思路。

Q 算法和 S 算法最终可以适用于地图未知迷宫环境，在转移栅格迷宫中的表现也优于传统的迷宫寻路算法。此外，综合实验中的发现与基础知识的验证，本文对 Q 算法和 S 算法奖赏函数与 Q 表初始值的设置提出了独到的见解，指出当今学术界在这方面的研究还不够多。

文中的实现的 DQN 结构需要整个迷宫环境作为网络输入，无法用在未知的迷宫环境里，具体原因在第四章中已做出了解释，第四章还讨论了强化学习迷宫寻路与传统寻路算法的区别。但该结构做到了 CNN 与强化学习的成功结合，证明了卷积神经网络的神通广大。

三种强化学习算法在仿真环境的成功应用证明强化学习作为一种寻路算法的可行性、可靠性。

## 参考文献

- [1] Sutton R S. Temporal Credit Assignment in Reinforcement Learning[J]. 1985.
- [2] Koenig S, Simmons R G. Complexity analysis of real-time reinforcement learning[C]//AAAI. 1993: 99-107.
- [3] Schrijver A. Combinatorial optimization: polyhedra and efficiency[J]. Discrete Applied Mathematics, 2005, 146: 120-122.
- [4] Stefán P, Monostori L, Erdélyi F. Reinforcement learning for solving shortest-path and dynamic scheduling problems[C]//Proceedings of the 3 rd International Workshop on Emergent Synthesis, IWES. 2001, 1: 83-88.
- [5] 刘建伟, 刘媛, 罗雄麟. 深度学习研究进展[J]. 计算机应用研究, 2014, 31(7): 1921-1930.
- [6] 高阳, 陈世福, 陆鑫. 强化学习研究综述[D]., 2004.
- [7] Stanley K O, Miikkulainen R. Evolving neural networks through augmenting topologies[J]. Evolutionary computation, 2002, 10(2): 99-127.
- [8] Silver D, Huang A, Maddison C J, et al. Mastering the game of Go with deep neural networks and tree search[J]. nature, 2016, 529(7587): 484.
- [9] Mnih V, Kavukcuoglu K, Silver D, et al. Playing atari with deep reinforcement learning[J]. arXiv preprint arXiv:1312.5602, 2013.
- [10] 陈学松. 强化学习及其在机器人系统中的应用研究[D]. 广东工业大学, 2011.
- [11] Zhou Z H. Machine learning[M]. Beijing: Tsinghua University Press, 2016.
- [12] Bertsekas D P. Dynamic programming and stochastic control[J]. Mathematics in science and engineering, 1976, 125: 222-293.
- [13] 陈圣磊, 谷瑞军, 陈耿, 等. 基于  $TD(\lambda)$  的自然梯度强化学习算法[J]. 计算机科学, 2010, 37(12): 186-189.
- [14] 张涛, 吴汉生. 基于神经网络的强化学习算法实现倒立摆控制[D]., 2006.
- [15] 朱圆恒, 赵冬斌. 概率近似正确的强化学习算法解决连续状态空间控制问题[J]. 2016.
- [16] Zhongming Z, Linong L, Xiaona Y, et al. 201612 动态资讯[J]. 2016.
- [17] Schaul T, Quan J, Antonoglou I, et al. Prioritized experience replay[J]. arXiv preprint arXiv:1511.05952, 2015.
- [18] 黄恩铭. 隐藏图形信息迷宫自动生成研究[D]. 南京师范大学, 2014.
- [19] Dewey D. Reinforcement learning and the reward engineering principle[C]//2014 AAAI Spring Symposium Series. 2014.
- [20] Matignon L, Laurent G J, Le Fort-Piat N. Reward function and initial values: better choices for accelerated goal-directed reinforcement learning[C]//International Conference on Artificial Neural Networks. Springer, Berlin, Heidelberg, 2006: 840-849.