程序报告

姓名：秦嘉俊　学号：3210106182

一、问题重述

1.1 实验背景

在本实验中，要求分别使用基础搜索算法和 Deep QLearning 算法，完成机器人自动走迷宫。

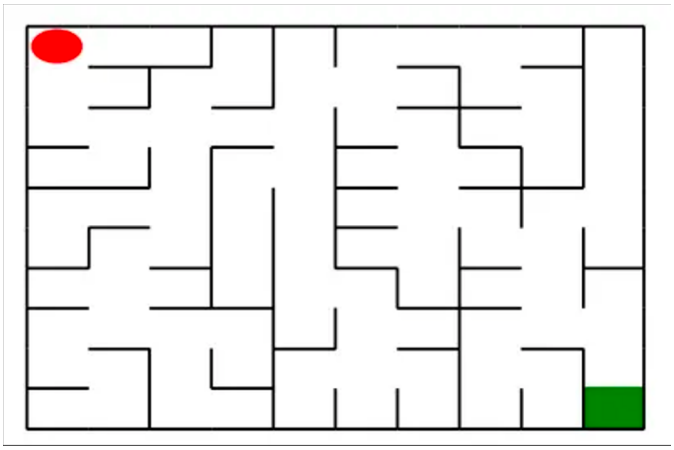


图 1: 机器人自动走迷宫

如上图所示，左上角的红色椭圆既是起点也是机器人的初始位置，右下角的绿色方块是出口。

游戏规则为: 从起点开始，通过错综复杂的迷宫，到达目标点 (出口)。在任一位置可执行动作包括：向上走'u'、向右走'r'、向下走'd'、向左走'l'。执行不同的动作后，根据不同的情况会获得不同的奖励，具体而言，有以下几种情况。

- 撞墙
- 走到出口
- 其余情况

该实验中，需要实现基于**基础搜索算法**和 **Deep QLearning 算法**的机器人，使机器人自动走到迷宫的出口。

1.2 实验要求

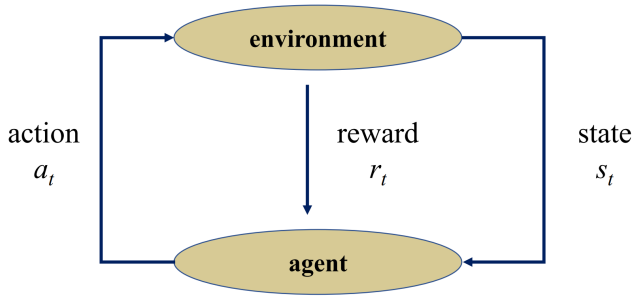
1. 使用 Python 语言。
2. 使用基础搜索算法完成机器人走迷宫。
3. 使用 Deep QLearning 算法完成机器人走迷宫。
4. 算法部分需要自己实现，不能使用现成的包、工具或者接口。

二、设计思想

2.1 Q-learning

Q-Learning 是一个值迭代（Value Iteration）算法。与策略迭代（Policy Iteration）算法不同，值迭代算法会计算每个”状态“或是”状态-动作“的值（Value）或是效用（Utility），然后在执行动作的时候，会设法最大化这个值。因此，对每个状态值的准确估计，是值迭代算法的核心。通常会考虑“最大化动作的长期奖励”，即不仅考虑当前动作带来的奖励，还会考虑动作长远的奖励。

在 Q-learning 中，我们通过维护一张 Q 值表，通过贝尔曼 (Bellman) 方程对其进行不停的迭代尝试直至收敛，然后根据 Q 值表获取 Agent 在每个状态下的最优策略。但 Q 值表在状态和动作空间都是有限且低维的时候适用，当状态-动作空间高维且连续时，维护一张无限庞大的 Q 值表是不现实的。因此，DQN 提出将 Q-Table 的更新问题变成一个函数拟合问题，相近的状态将得到相近的动作输出，即使用神经网络对动作-状态的 Q 值进行建模估计。



使用强化学习算法设计一个可以在一定大小，的迷宫中找到宝藏的 agent(智能体)。假设迷宫可以使用一系列字符表示，字符”o” 表示 agent 的位置，字符”T” 表示宝藏的位置，该位置的收益为 1，且为游戏的终结状态，字符”P” 表示陷阱的位置，该位置的收益为-1，且为游戏的终结状态，字符”_” 表示收益为 0 的中间状态。初始情况下，迷宫如下图表示：

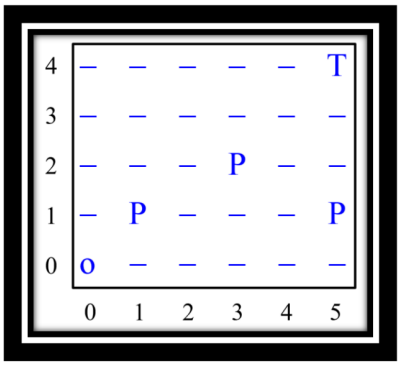


图 2: 迷宫

每次行动 agent 可以选择向不超过迷宫边界的相邻位置移动一格，通过 Q-Learning 算法希望 agent 最终找到一条通往宝藏的路径。

用 States 表示智能体在环境中可能处于的所有状态，Actions 表示智能体在环境中所能采取的所有行动，Rewards 表示智能体所能够获取的奖励，那么：

$$States = \{(0,0),(0,1),\ldots,(4,4),(4,5)\}$$

$$Actions = \{(+1,0),(-1,0),(0,-1),(0,+1)\}$$

$$Rewards = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

算法中使用到的三个参数如表格中所示。

1. 1. 以 0.2 的概率随机选择策略
   以 $(1-0.2)$ 的概率执行贪婪策略 Q-table 中对应状态贪婪策略有多个备选 action 时，从备选 action 中随机选择 action；否则选取最大 Q-value 的 action。

2. 更新 Q-Table

ε	**epsilon**	**0.2**	**贪婪度 greedy**：此实验算法中以epsilon概率进行随机决策，以(1-epsilon)概率进行贪婪决策
α	**alpha**	**0.2**	**学习率**：在强化学习中，学习率α越大,表示采用新的尝试得到的结果比例越大,保持旧的结果的比例越小
γ	**gamma**	**0.8**	**奖励递减值(折现率)**：强化学习中，期望奖励会以奖励乘以奖励递减值的形式体现

图 3: **Q-Table**

Q-learning 算法将状态（state）和动作（action）构建成一张 Q_table 表来存储 Q 值，Q 表的行代表状态（state），列代表动作（action）。

Q-Table	a_1	a_2
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$
s_3	$Q(s_3, a_1)$	$Q(s_3, a_2)$

在 Q-Learning 算法中，这个长期奖励记为 Q 值，其中会考虑每个“状态-动作”的 Q 值，具体而言，它的计算公式为：

$$\mathbf{q}(\mathbf{s}, \mathbf{a}) \leftarrow (\mathbf{1} - \alpha)\mathbf{q}(\mathbf{s}, \mathbf{a}) + \alpha[\mathbf{R} + \gamma \max_{\alpha'} \mathbf{q}(\mathbf{s}^{'}, \mathbf{a}^{'})]$$

计算得到新的 Q 值之后，一般会使用更为保守地更新 Q 表的方法，即引入松弛变量 α, 按如下的公式进行更新，使得 Q 表的迭代变化更为平缓。

$$\mathbf{q}(\mathbf{s}, \mathbf{a}) \leftarrow \mathbf{q}(\mathbf{s}, \mathbf{a}) + \alpha[\mathbf{R} + \gamma \max_{\alpha'} \mathbf{q}(\mathbf{s}^{'}, \mathbf{a}^{'}) - \mathbf{q}(\mathbf{s}, \mathbf{a})]$$

具体来说，根据上面的定义，会尽可能地让机器人在每次选择最优的决策，来最大化长期奖励。但是这样做有如下的弊端：

在初步的学习中，Q 值是不准确的，如果在这个时候都按照 Q 值来选择，那么会造成错误。学习一段时间后，机器人的路线会相对固定，则机器人无法对环境进行有效的探索。因此需要一种办法，来解决如上的问题，增加机器人的探索。通常会使用 epsilon-greedy 算法：

1. 在机器人选择动作的时候，以一部分的概率随机选择动作，以一部分的概率按照最优的 Q 值选择动作。
2. 同时，这个选择随机动作的概率应当随着训练的过程逐步减小。

下面是使用 ϵ 贪心算法进行探索的 Q 学习算法：

Algorithm 1 ϵ 贪心算法进行探索的 Q 学习算法

procedure EPSILONGREEDY

Input: 状态 s, 动作 - 价值函数 q_π, 参数 ϵ

Output: 动作 a

$n \sim Uniform(0, 1)$

if $n < \epsilon$ **then**

$a \leftarrow \ from\ A$

else

$a \leftarrow arg\max_{a'} q_\pi(s, a')$

end if

end procedure

procedure QLEARNING

Input: 马尔可夫决策过程 $MDP = (s, A, P, R, \gamma)$

Output: 策略 π

Initialise q_π arbitrarily

repeat

$s \leftarrow$ Initial State

repeat

$a \leftarrow$ EpsilonGreedy(s, q_π, ϵ)

Perform action a and observe reward R and the next state s'

$q(s, a) \leftarrow q(s, a) + \alpha[R + \gamma \max_{\alpha'} q(s', a') - q(s, a)]$

$s \leftarrow s'$

until s is the abort state

until q_π convergence

$\pi(s) := arg\max_a q(s, a)$

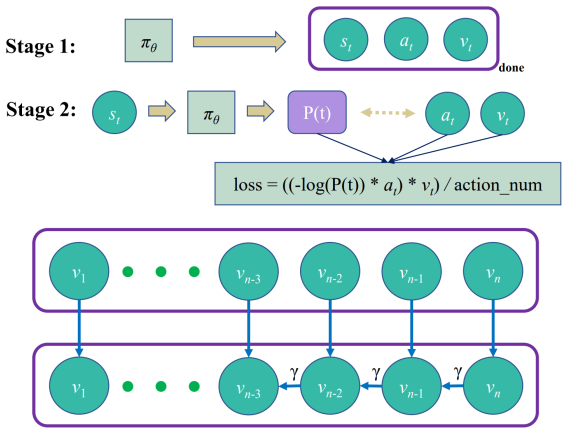
end procedure

2.2 基于策略梯度的强化学习

REINFORCE 算法用来估计策略梯度：

Algorithm 2 基于策略梯度的强化学习

function REINFORCE
  Initiallise θ arbitrarily
  for each episode $\{s_1, a_1, r_1, \ldots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
  end for
  return θ
end function



其中 $v_t = \dfrac{v_t - mean(v)}{std(v)}$

2.3 DQN 算法

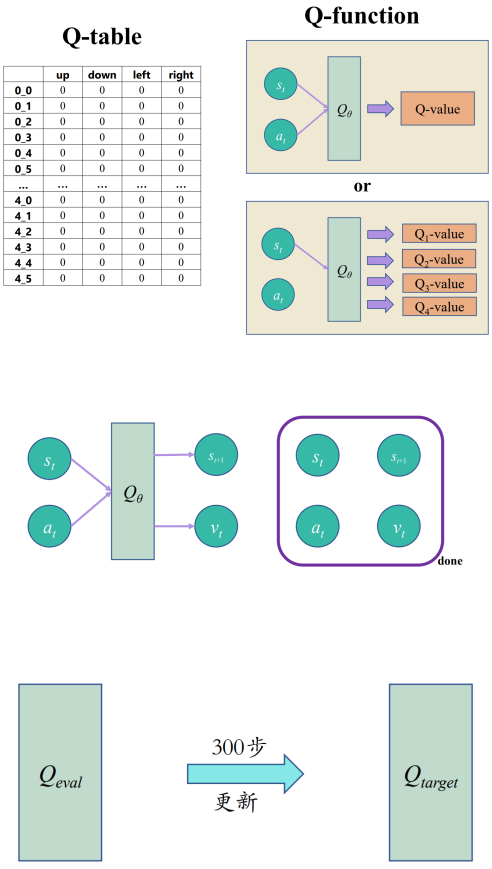
Q 学习算法中，动作-价值函数需要用一个数组来记录。但是在有些实际问题中，状态数量会非常多甚至是无限的，这样会造成如下两个问题：首先是算法很难用一个数组来存储动作-价值函数的值；其次是有些状态的访问次数可能很少甚至根本没有被访问过，这些状态的价值估计是不可靠的。

为了解决这两个问题，一种解决方案是将动作-价值函数参数化 (parametrize)，即用一个回归模型来拟合 q_π 函数。如果回归模型是一个深度神经网络，那么这样的算法就被称为深度强化学习 (deep reinforcement learning) 算法。

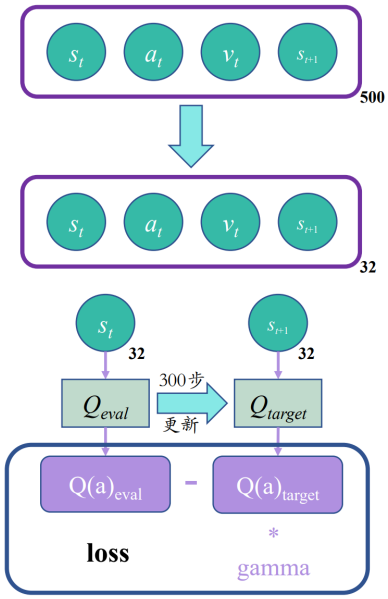
参数化算法会带来心的问题。第一个问题是，集中优化相关性强的样本会导致神经网络处理其他样本无法取得较好效果。另一个问题是，在单个样本优化目标不明确的情况下，由于采样具有概率性，算法的收敛会变得不稳定。

在 Mnih 等人提出的深度 Q 网络 (deep Q-network, DQN) 中，这两个问题得到了针对性的解决。

对于样本相关性太强的问题，DQN 采用经验重现 (experience replay) 的方法来应对，经验重现将算法的探索片段以 (s, a, R, s') 的四元形式存在一张表中，每当算法探索得到一个新的四元组，便将该四元组加入经验重现表；而在算法对参数 θ 进行更新时，则从经验重现表中随机选取一批样本用于计算损失函数。使用经验重现主要有两个好处，一方面是样本之间的相关性显著减弱，因为随机采样得到的样本状态会更均匀地分布在已探索的样本空间中。另一方面过去的经验可以被重复利用，因此提高的信息利用的效率。



算法如下所示：



Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

图 4: 具体算法

三、代码内容

3.1 基础搜索算法

Maze 类中重要的成员方法如下：

- `sense_robot()`：获取机器人在迷宫中目前的位置。
  `return`：机器人在迷宫中目前的位置
- `move_robot(direction)`：根据输入方向移动默认机器人，若方向不合法则返回错误信息。
  `direction`：移动方向，如:"u"，合法值为：`['u', 'r', 'd', 'l']`
  `return`：执行动作的奖励值
- `can_move_actions(position)`：获取当前机器人可以移动的方向
  `position`：迷宫中任一处的坐标点
  `return`：该点可执行的动作，如：`['u','r','d']`
- `is_hit_wall(self, location, direction)`：判断该移动方向是否撞墙
  `location, direction`：当前位置和要移动的方向，如`(0,0) , "u"\verb`
  `return`：`True`(撞墙) / `False`(不撞墙)

- `draw_maze()`：画出当前的迷宫

对于迷宫游戏，常见的三种的搜索算法有广度优先搜索、深度优先搜索和最佳优先搜索 (A*)。在示例中已经给出的广度优先搜索的实现。在本实验中采用的是深度优先搜索算法。

```
def depth_first_search(maze, path):
    start = maze.sense_robot()
    end = maze.destination
    if start == end:
        return path
    h, w, _ = maze.maze_data.shape
    is_visit_m = np.zeros((h, w), dtype=np.int) # 标记迷宫的各个位置是否被访问过
    maze.reset_robot()
    for pos in path:
        is_visit_m[maze.sense_robot()] = 1
        maze.move_robot(pos) # 从初始位置走到当前位置，标记被访问过的位置
    directions = maze.can_move_actions(start) #获取当前可以 yiddong
    for direction in directions:
        maze.move_robot(direction) #尝试每一个可行的方向
        if is_visit_m[maze.sense_robot()] == 1:
            maze.robot["loc"] = start #如果已经访问过则回到原位置
            continue
        path.append(direction) #延长路径
        new_path = depth_first_search(maze, path) #递归搜索
        if maze.sense_robot() == end: #若到终点则结束
            path = new_path
            break
        del(path[-1]) #若没到终点则还原当前路径
        maze.robot["loc"] = start #回到原位置
    return path

def my_search(maze):
    """
    任选深度优先搜索算法、最佳优先搜索(A*)算法实现其中一种
    :param maze: 迷宫对象
    :return :到达目标点的路径 如: ["u","u","r",...]
    """

    path = depth_first_search(maze, [])
    return path
```

3.2 实现 DQN Robot

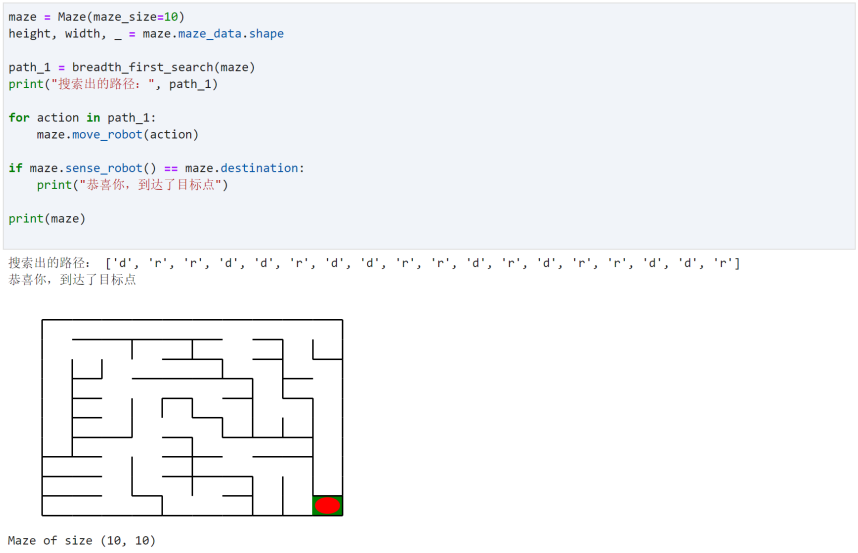
在平台提供的代码中，QRobot 类，实现了 Q 表迭代和机器人动作的选择策略; Runner 用于机器人的训练和可视化。

```
from QRobot import QRobot
class Robot(QRobot):

    valid_action = ['u', 'r', 'd', 'l']

    def __init__(self, maze, alpha=0.8, gamma=0.5, epsilon0=0.8):
        super(Robot, self).__init__(maze, alpha, gamma, epsilon0)
        self.maze = maze

    def train_update(self):
        """
        以训练状态选择动作，并更新相关参数
        :return :action, reward 如："u", -1
        """

        self.state = self.sense_state() # 获取机器人当初所处迷宫位置
        self.create_Qtable_line(self.state) # 对当前状态，检索 Q 表，如果不存在则添加进入 Q 表
        action = random.choice(self.valid_action) if random.random() < self.epsilon else max(
            self.q_table[self.state], key=self.q_table[self.state].get) # 选择动作
        reward = self.maze.move_robot(action) # 以给定的动作（移动方向）移动机器人
        next_state = self.sense_state() # 获取机器人执行动作后所处的位置
        self.create_Qtable_line(next_state) # 对当前 next_state，检索 Q 表，如果不存在则添加进入 Q 表
        self.update_Qtable(reward, action, next_state) # 更新 Q 表 中 Q 值
        self.update_parameter() # 更新其它参数
        return action, reward

    def test_update(self):
        """
        以测试状态选择动作，并更新相关参数
        :return :action, reward 如："u", -1
        """

        self.state = self.sense_state() # 获取机器人当初所处迷宫位置
        self.create_Qtable_line(self.state) # 对当前状态，检索 Q 表，如果不存在则添加
```

```
        进入 Q 表
35      action = max(self.q_table[self.state],
36      key=self.q_table[self.state].get) # 选择动作
37      reward = self.maze.move_robot(action) # 以给定的动作（移动方向）移动机器人
38      return action, reward
```

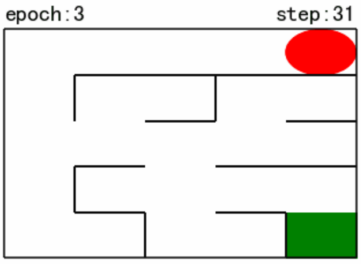
这里我们实现 DQN 算法在机器人自动走迷宫中的应用。我们实现的 `Robot` 类继承了 `QRobot` 类，基本保留了父类的属性和方法，不同的是在初始化 `alpha, gamma, epsilon` 的值时做了一定的调整，以达到更好的效果。

四、实验结果



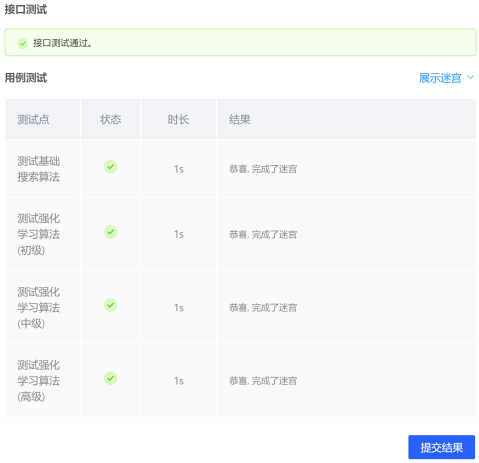
(a) 基础搜索算法

(b) 测试 DQN 算法

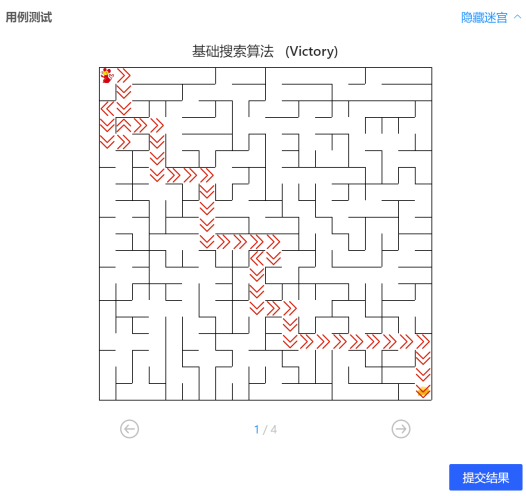


(c) 测试 DQN 算法

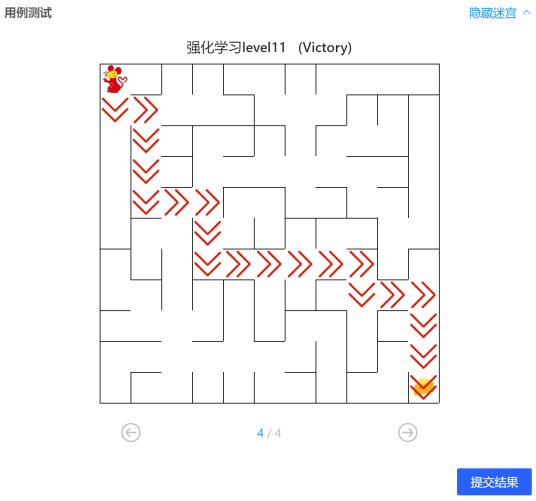
图 5: 自主检验算法效果



(a)



(b) 基础算法



(c) DQN 算法

图 6: 系统测试

五、总结

本次试验对强化学习中的 Q-Learning 和 DNQ 算法进行了学习，并使用其完成了机器人自动走迷宫。其中 DQN 是深度学习最有代表性的工作之一。强化学习能够广泛应用于解决实际生活中常见的决策问题。强化学习作为一种通用的策略学习框架，向人们展示了其强大的能力和应用前景。

人工智能基础的实验到此告一段落，收获颇丰，感慨良多，期待未来还能有机会学习这方面的知识！江湖路远，后会有期！