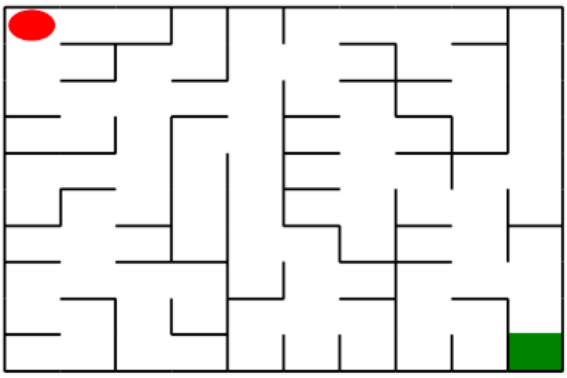
1 实验介绍

1.1 实验内容

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。



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
- 需要分别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。

1.3 实验环境

可以使用 Python 实现基础算法的实现，使用 PyTorch 框架实现 Deep QLearning 算法。

2 基础搜索算法

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*)。这里分别完成了广搜和深搜。

2.1 广度优先搜索

2.1.1 算法思想

广搜主要通过建立一颗搜索树并进行层次遍历实现。算法的实现步骤为:
首先以机器人起始位置建立根节点，并入队；接下来不断重复以下步骤直到判定条件:

(1). 将队首节点的位置标记已访问；判断队首是否为目标位置 (出口)，是则终止循环并记录回溯路径

(2). 判断队首节点是否为叶子节点，是则拓展该叶子节点

(3). 如果队首节点有子节点，则将每个子节点插到队尾

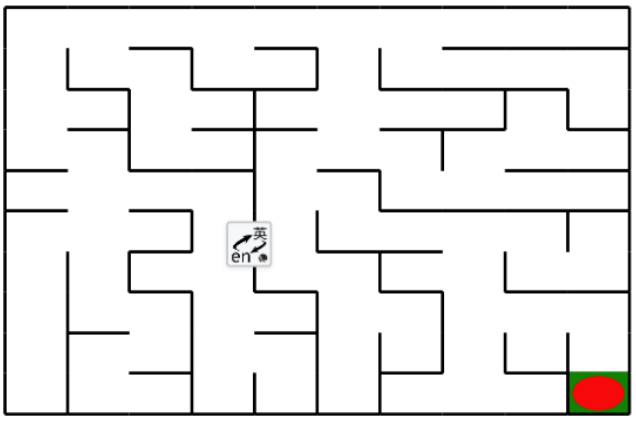
(4). 将队首节点出队

2.1.2 代码实现

```
1  def breadth_first_search(maze):
2      """
3      对迷宫进行广度优先搜索
4      :param maze: 待搜索的maze对象
5      """
6      start = maze.sense_robot()
7      root = SearchTree(loc=start)
8      queue = [root] # 节点队列，用于层次遍历
9      h, w, _ = maze.maze_data.shape
10     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
11     path = [] # 记录路径
12     while True:
13         current_node = queue[0]
```

```
        is_visit_m[current_node.loc] = 1 # 标记当前节点位置已访问

        if current_node.loc == maze.destination: # 到达目标点
            path = back_propagation(current_node)
            break

        if current_node.is_leaf():
            expand(maze, is_visit_m, current_node)

        # 入队
        for child in current_node.children:
            queue.append(child)

        # 出队
        queue.pop(0)

    return path
```

2.1.3 实验结果



对于上图所示迷宫，算法顺利找到了出口：['r', 'r', 'r', 'r', 'r', 'd', 'd', 'd', 'l', 'd', 'r', 'd', 'r', 'r', 'd', 'd', 'r', 'r', 'd', 'd']

2.2 深度优先搜索

2.2.1 算法思想

深度优先搜索的思想是沿着树的分支深入到最深的节点，再回溯到上一个分支点，继续尝试未访问的路径，直到找到目标位置或搜索完所有可能路径。深搜算法的实现步骤如下：

(1). 将起始位置作为根节点入栈。

(2). 重复以下步骤直到栈为空或找到目标位置：

(a) 将栈顶节点出栈，并标记其为已访问。

(b) 判断栈顶节点是否为目标位置（出口），是则终止循环并记录回溯路径。

(c) 如果栈顶节点不是叶子节点，拓展其子节点，并将所有未访问的子节点按某种顺序（如上、右、下、左）依次入栈。

(3). 若遍历完所有节点后仍未找到目标位置，说明目标不可达。

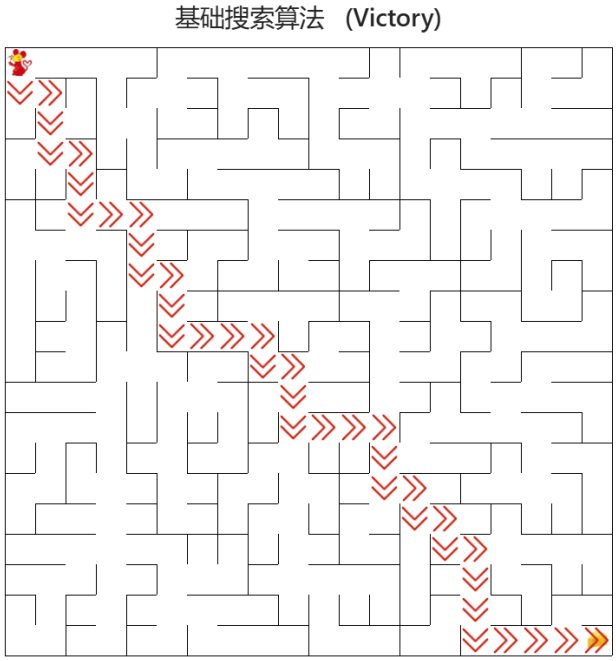
2.2.2 代码实现

```
def DFS(maze):
        """
        对迷宫进行深度
        :param maze: 待搜索的maze对象
        """
        start = maze.sense_robot()
        root = SearchTree(loc=start)
        queue = [root] # 节点堆栈，用于层次遍历
        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
        path = [] # 记录路径
        peek = 0
        while True:
            current_node = queue[peek] # 栈顶元素作为当前节点
            #is_visit_m[current_node.loc] = 1 # 标记当前节点位置已访问

            if current_node.loc == maze.destination: # 到达目标点
                path = back_propagation(current_node)
                break
```

```
20
21          if current_node.is_leaf() and is_visit_m[current_node.loc] ==
                0: # 如果该点存在叶子节点且未拓展
22              is_visit_m[current_node.loc] = 1 # 标记该点已拓展
23              child_number = expand(maze, is_visit_m, current_node)
24              peek+=child_number # 开展一些列入栈操作
25              for child in current_node.children:
26                  queue.append(child) # 叶子节点入栈
27          else:
28              queue.pop(peek) # 如果无路可走则出栈
29              peek-=1
30          # 出队
31          #queue.pop(0)
32
33      return path
```

2.2.3 实验结果

上传 MO 进行验证，可以看到得到了一条正确的路径。



3 强化学习基础

3.1 强化学习简介

强化学习作为机器学习算法的一种，其模式也是让智能体在“训练”中学到“经验”，以实现给定的任务。但不同于监督学习与非监督学习，在强化学习的框架中，我们更侧重通过智能体与环境的交互来学习。

通常在监督学习和非监督学习任务中，智能体往往需要通过给定的训练集，辅之以既定的训练目标（如最小化损失函数），通过给定的学习算法来实现这一目标。然而在强化学习中，智能体则是通过其与环境交互得到的奖励进行学习。

这个环境可以是虚拟的（如虚拟的迷宫），也可以是真实的（自动驾驶汽车在真实道路上收集数据）。

3.2 核心概念

在强化学习中有五个核心组成部分，它们分别是: 环境（Environment）、智能体（Agent）、状态（State）、动作（Action）和奖励（Reward）。

在某一时间节点 t:

(1). 智能体在从环境中感知其所处的状态 s_t

(2). 智能体根据某些准则 (策略) 选择动作 a_t

(3). 环境根据智能体选择的动作，向智能体反馈奖励 r_{t+1}

通过合理的学习算法，智能体将在这样的问题设置下，成功学到一个在状态 s_t 选择动作 a_t 的策略 $\pi(s_t) = a_t$。

4 Q-learning

4.1 算法思想

Q-Learning 是一个值迭代（Value Iteration）算法。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。

而 Q-Learning 维护的 Q 值表中记录的 Q 值，即为对每个”状态“或是”状态-动作“的值（Value）/效用（Utility）的估计。

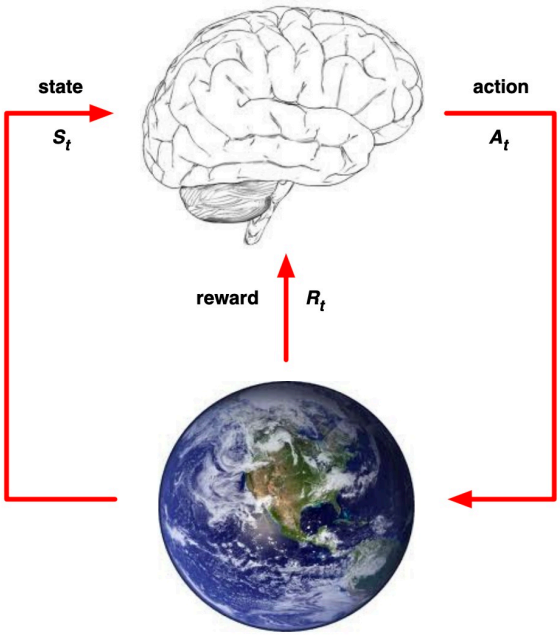


Figure 1: 智能体与环境的交互

4.2 Q 值的计算和迭代

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）：

在 Q-Learning 算法中，将长期奖励记为 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：

$$Q(s_t, a) = R_{t+1} + \gamma \times \max_a Q(a, s_{t+1})$$

也就是对于当前的“状态-动作”(s_t, a)，考虑执行动作 a 后环境奖励 R_{t+1}，以及执行动

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	$Q(s_3, a_1)$	$Q(s_3, a_2)$

作 a 到达 s_{t+1} 后，执行任意动作能够获得的最大的 Q 值 $\max_a Q(a, s_{t+1})$，γ 为折扣因子。

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 $alpha$ ，按如下的公式进行更新，使得 Q 表的迭代变化更为平缓。

$$Q(s_t, a) = (1 - \alpha) \times Q(s_t, a) + \alpha \times (R_{t+1} + \gamma \times \max_a Q(a, s_{t+1}))$$

4.3 动作的选择

在强化学习中，”探索-利用”问题是非常重要的问题。

具体来说，从最大化长期奖励的目标来看，在选择动作时，会尽可能地让机器人在每次选择最优的决策。

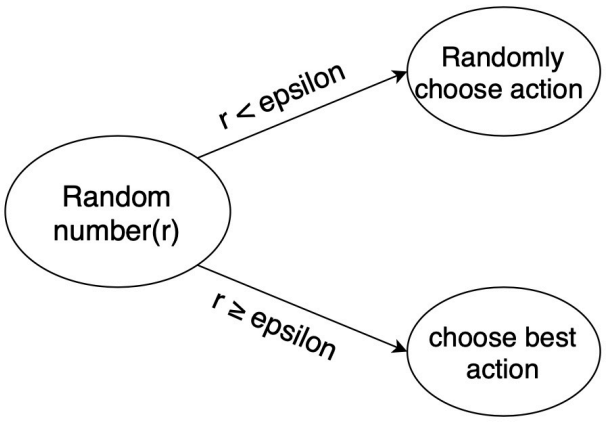
但是这样做有如下的弊端：

- 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。

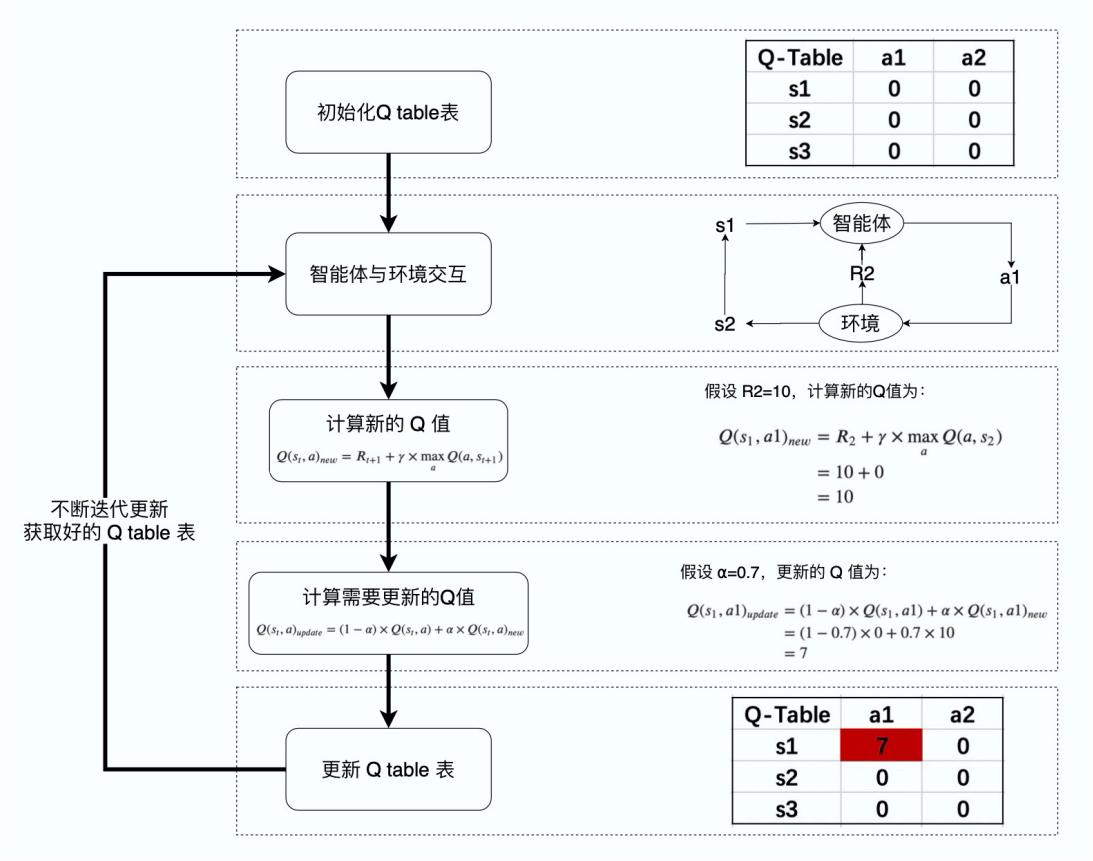
因此需要一种办法，来解决如上的问题，增加机器人的探索。最简单的是使用“epsilon-greedy”算法：

(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。



4.4 算法流程



5 DQN

5.1 从 Qlearning 到 DQN

DQN 是 Q-learning 的扩展，其主要区别在于：

(1). Q 函数的表示:

- 在 Q-learning 中，$Q(s,a)$ 是一个表格或简单的函数，而在 DQN 中，$Q(s,a)$ 被一个深度神经网络用参数 θ 近似表示。
- 使用 $Q(\phi(s),a;\theta)$ 表示状态-动作值。

(2). 目标计算:

- Q-learning 的目标值是：$y_j = r_j + \gamma \max_{a'} Q(s_{j+1},a')$。
- DQN 中的目标值是类似的，但通过目标网络 $\hat{Q}$ 计算：

$$y_j = r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1},a';\theta^-)$$

(3). 经验回放:

- DQN 使用经验回放机制 D，通过采样过去的转移数据打破相关性。
- Q-learning 通常直接使用当前的状态转移进行更新。

(4). 目标网络:

- 为了提高稳定性，DQN 引入了目标网络 $\hat{Q}$，而 Q-learning 则直接使用当前网络更新。

5.2 核心概念

(1). $\phi(s)$:

- 状态 s 的预处理表示 (state representation)。
- 通常指经过神经网络或其他方法对状态进行编码后的特征表示。
- 在 DQN 中，$\phi(s)$ 是输入到深度神经网络的特征。

(2). $Q(\phi(s), a; \theta)$:

- 带参数 θ 的 Q 函数近似。
- 深度神经网络用参数 θ 近似 Q-learning 中的 $Q(s, a)$ 函数。

(3). $\hat{Q}(\phi(s), a; \theta^-)$:

- 目标 Q 函数 (target Q-function)。
- 目标网络使用参数 θ^- 来计算目标值 y_j，通常是 θ 的一个延迟副本。

(4). D:

- 经验回放存储器 (replay memory)。
- 用于存储过去的状态转移 $(\phi_1, a_2, r_1, \phi_{t+1})$，方便后续采样训练。

(5). y_j:

- 目标值 (target value)。
- 用于计算 TD 误差的目标值，具体为:
  - $y_j = r_j$ (如果是终止状态)。
  - $y_j = r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$ (如果非终止状态)。

(6). ε:

- 探索概率 (exploration probability)。

- 用于 ε-贪婪策略，控制随机探索与利用当前策略的平衡。

(7). C:

- 目标网络更新频率 (target network update frequency)。
- 每 C 步，将主网络的参数 θ 复制到目标网络 θ^- 中。

(8). $L(\theta)$:

- 损失函数 (loss function)。
- 用于训练神经网络，表示预测值 $Q(\phi_j, a_j; \theta)$ 与目标值 y_j 之间的平方误差:

$$L(\theta) = (y_j - Q(\phi_j, a_j; \theta))^2$$

5.3 算法流程

1. 初始化

(1). 创建一个经验回放内存 D:

- 类似一个“记事本”，用来存储智能体过去的状态转移（状态、动作、奖励、下一个状态）。
- 我们会从中随机抽取小批量的“记忆”用来训练网络。

(2). 初始化两个深度网络:

- 主网络 $Q(\phi(s), a; \theta)$：用来预测动作价值。
- 目标网络 $\hat{Q}(\phi(s), a; \theta^-)$：用来生成目标值（更新慢）。

(3). 智能体随机探索环境，执行随机动作，填充经验回放池 D 至一定数量。

2. 每一轮训练的主要步骤

以“玩游戏”为例，每一轮是一个“关卡”（episode），从起点到结束。我们以每一步的操作为基础:

Step 1: 行为选择（探索与利用）

(1). 智能体处于状态 s_t。

(2). 按照 ϵ-贪婪策略选择动作 a_t:

- 以 ϵ 的概率随机选择一个动作 a_t（探索）。

- 否则，选择当前 Q 网络预测的最大值对应的动作:

$$a_t = \arg\max_a Q(\phi(s_t), a; \theta)$$

- tip: 这是用来平衡 “尝试新策略”（探索）和 “使用已有经验”（利用）的策略。

Step 2: 执行动作，观察奖励和新状态

(1). 执行动作 a_t，获得即时奖励 r_t。

(2). 观察下一个状态 s_{t+1}，将其通过预处理得到特征表示 $\phi(s_{t+1})$。

Step 3: 存储到经验回放

(1). 将当前转移 $(\phi(s_t), a_t, r_t, \phi(s_{t+1}))$ 存储到回放内存 D 中。

(2). 如果 D 已满，则会移除最早的记忆，保持容量固定。

Step 4: 从经验回放中采样，更新网络

(1). 随机从 D 中抽取一小批转移 $(\phi_j, a_j, r_j, \phi_{j+1})$。

(2). 对每个样本计算目标值 y_j:

- 如果 ϕ_{j+1} 是终止状态（比如游戏结束）:

$$y_j = r_j$$

- 如果非终止状态:

$$y_j = r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$$

- tip：这是 Q-learning 的核心思想：即时奖励 + 未来价值折扣。

(3). 计算损失函数:

- 损失函数衡量网络输出的 Q 值和目标值 y_j 的差距:

$$L(\theta) = (y_j - Q(\phi_j, a_j; \theta))^2$$

- tip：这类似于预测和真实值的 “误差”，通过不断减小误差让网络更精确。

(4). 执行梯度下降，更新主网络的参数 θ。

Step 5: 更新目标网络（定期）

(1). 每 C 步，将主网络的参数 θ 复制到目标网络 θ^- 中:

$$\theta^- = \theta$$

(2). tip: 防止目标值计算时波动过大，提供稳定性。

通过不断重复以上过程，DQN 逐渐学会评估状态和动作的价值。

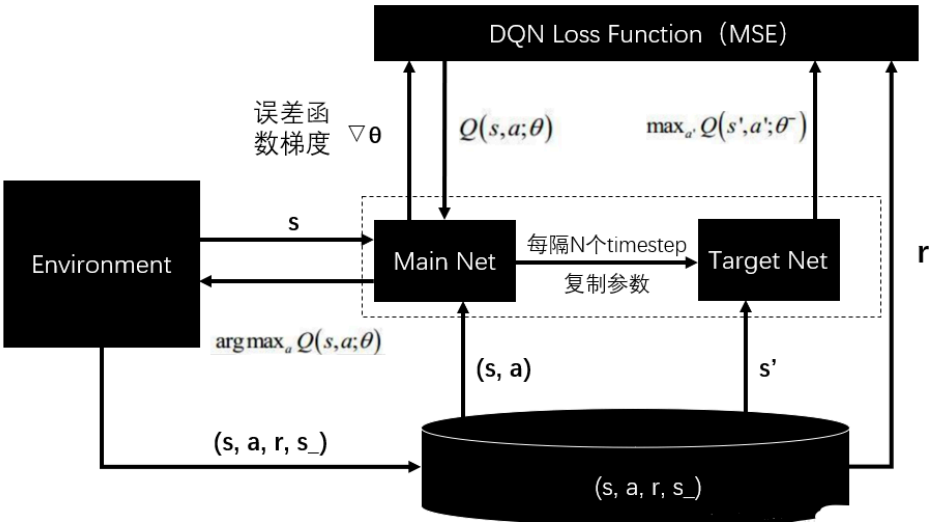


Figure 2: DQN 算法框图

5.4 算法实现

这里继承了已实现的 MinDQNRobot 类，使用三层神经网路（输入：机器人当前的位置坐标，输出：执行四个动作（up、right、down、left）的评估分数)。

实验可以发现，原模型对于维度更高的迷宫，无法在限定时间内给出正确走法。推测有以下原因:

(1). 到达 destination 的奖励是固定的，如果维度过高，则即使按照最优路径走到终点，机器人所获得的奖励仍可能过高 (这里的 reward 取了负值，即目标变成最小化长期奖励)。

(2). 训练回合数较少，且 epoch 不太合适。

(3). 由于训练回合有所限制，神经网络不能太过复杂。

因此针对以上问题，做了如下优化:

(1). 修改到达 destination 的奖励，设置为 $8mazesize^2$

(2). 扩大 batch size 到最大容量 len(self.memory)，同时 epoch 的轮数改为直到模型训练出最优路线为止。

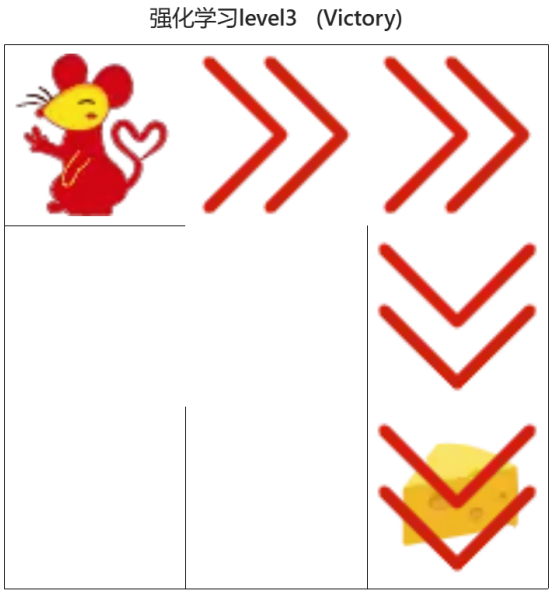
(3). 将神经元数量调整为 2（状态 x，y）→ 128 → 64 → 4（四个动作的评分）

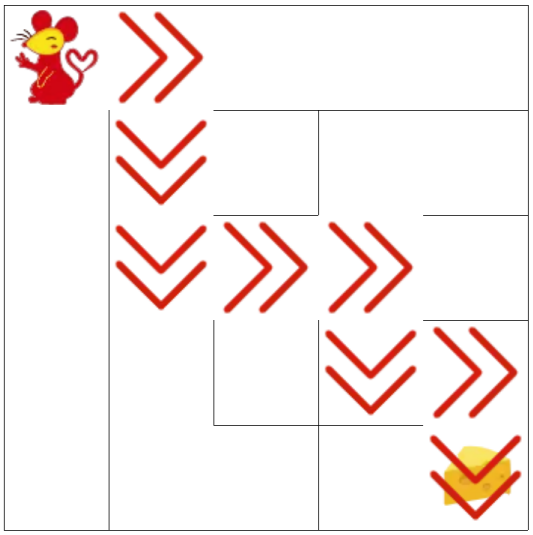
```
class Robot(TorchRobot):

def __init__(self, maze):
    """
    初始化 Robot 类
    :param maze:迷宫对象
    """
    super(Robot, self).__init__(maze)
    maze.set_reward(reward={
        "hit_wall": 10.0,
        "destination": -maze.maze_size ** 2.0*8,
        "default": 1.0,
    })
    """开启金手指，获取全图视野"""
    self.memory.build_full_view(maze=maze)
    # 初始化后即开始训练
    self.train()


def train(self):
    # 训练，直到能走出这个迷宫
    while True:
        self._learn(batch=len(self.memory) )
        self.reset()
        for _ in range(self.maze.maze_size ** 2):
            a, r = self.test_update()
            if r == self.maze.reward["destination"]:
                return

def train_update(self):
    state = self.sense_state()
    action = self._choose_action(state)
    reward = self.maze.move_robot(action)
```

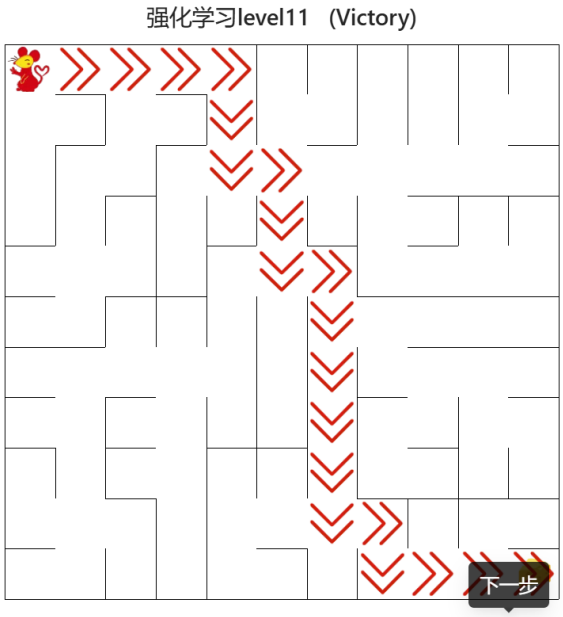
```
34
35          return action, reward
```

5.5 实验结果

上传至 MO 平台进行验证，顺利通过测试：







同时，速度也十分可观：

测试点	状态	时长	结果
测试强化学习算法(初级)	✓	0s	恭喜, 完成了迷宫
测试基础搜索算法	✓	0s	恭喜, 完成了迷宫
测试强化学习算法(中级)	✓	1s	恭喜, 完成了迷宫
测试强化学习算法(高级)	✓	90s	恭喜, 完成了迷宫

6 总结

在本次机器人走迷宫的实验中，我们主要探索了两种算法：传统的搜索算法和基于强化学习的深度 Q 网络（DQN）算法。通过本次实验，我进一步了解了两类算法的优缺点。对于基本搜索算法，总体实现都比较简单：深度优先搜索简单而直观，适用于简单场景；广度优先搜索能够找到最短路径，但可能受内存限制；A* 算法则巧妙地融合了 DFS 和

BFS 的优点，提供了一种更为智能的搜索策略。然而，当面对更高维度的未知复杂迷宫时，这些传统搜索算法往往难以达到理想的效果相比之下，DQN 算法通过强化学习的方式，能够适应复杂且未知的环境。它通过不断的学习来优化策略，展现出对复杂问题的应对能力。但同时，DQN 算法对参数设置和训练过程非常敏感，要获得一个表现良好的模型，往往需要大量的训练数据和计算资源。

在 DQN 算法的实现过程中，我们针对一些关键问题进行了改进：为了解决训练轮数不足和批量大小（batch size）过小的问题，我们将 batch size 调整至经验回放存储器（memory）的最大容量，即 len(self.memory)。同时，我们将训练的轮数（epoch）设置为直至模型能够稳定地找到最优路径为止，而非固定轮数。另外，我们选择了更少的神经元数量。通过这些改进，我们的 DQN 算法在处理复杂迷宫问题时展现出了更好的性能和稳定性。