人工智能与机器学习报告

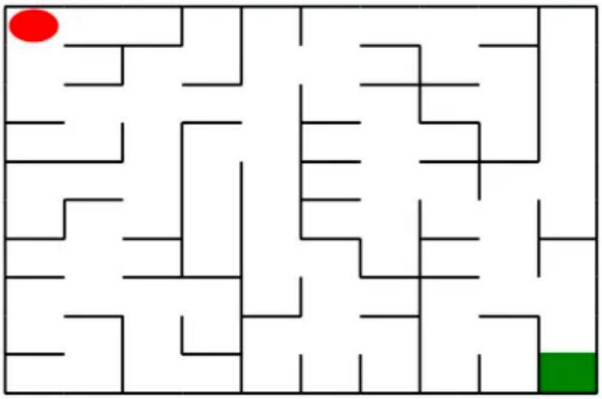
2. 机器人走迷宫

3220101111 洪晨辉

1. 实验介绍

1.1 实验背景

本实验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。

1.2 实验要求

A)使用 Python 语言。

B)使用基础搜索算法完成机器人走迷宫。

C)使用 Deep QLearning 算法完成机器人走迷宫。

D)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 Keras、PyTorch 等框架实现 Deep QLearning 算法。本例中采用 PyTorch。

2. 实验操作

2.1 创建迷宫

通过迷宫类 Maze 可以随机创建一个迷宫。具体实现存在 Maze.py 中，已经写好。使用 Maze(maze_size=size) 来随机生成一个定大小的迷宫。

raw_maze()：画出当前的迷宫

2.2 创建机器人

同理，有一个机器人类，存在 Maze.py 中。

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"

return：True(撞墙) / False(不撞墙)

2.3 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*）。这里采用最佳优先搜索算法。当然，经过测试，这三种算法都可以很好地完成搜索的任务。

```
# 导入相关包
import os
import random
import numpy as np
from Maze import Maze
from Runner import Runner
from QRobot import QRobot
from ReplayDataSet import ReplayDataSet
```

```python
from torch_py.MinDQNRobot import MinDQNRobot as TorchRobot # PyTorch 版本
from keras_py.MinDQNRobot import MinDQNRobot as KerasRobot # Keras 版本
import matplotlib.pyplot as plt
from functools import total_ordering

import numpy as np
import heapq

# 机器人移动方向
move_map = {
    'u': (-1, 0),  # up
    'r': (0, +1),  # right
    'd': (+1, 0),  # down
    'l': (0, -1),  # left
}

@total_ordering
class SearchTree(object):
    def __init__(self, loc=(), action='', parent=None, g=0, h=0):
        self.loc = loc  # 当前节点位置
        self.to_this_action = action  # 到达当前节点的动作
        self.parent = parent  # 当前节点的父节点
        self.children = []  # 当前节点的子节点
        self.g = g  # 实际代价
        self.h = h  # 启发式代价

    def add_child(self, child):
        self.children.append(child)

    def is_leaf(self):
        return len(self.children) == 0

    def f(self):
        return self.g + self.h

    def __eq__(self, other):
        return self.f() == other.f()

    def __lt__(self, other):
        return self.f() < other.f()

def heuristic(loc, goal):
    """
    计算启发式代价函数 (曼哈顿距离)
    :param loc: 当前节点位置
    :param goal: 目标节点位置
    :return: 曼哈顿距离
    """
    return abs(loc[0] - goal[0]) + abs(loc[1] - goal[1])

def expand(maze, is_visit_m, node, goal):
    """
    拓展叶子节点，即为当前的叶子节点添加执行合法动作后到达的子节点
    :param maze: 迷宫对象
    :param is_visit_m: 记录迷宫每个位置是否访问的矩阵
    :param node: 待拓展的叶子节点
    :param goal: 目标位置
    """
    can_move = maze.can_move_actions(node.loc)
    for a in can_move:
        new_loc = tuple(node.loc[i] + move_map[a][i] for i in range(2))
        if not is_visit_m[new_loc]:
            g = node.g + 1  # 每移动一步的代价为1
            h = heuristic(new_loc, goal)
            child = SearchTree(loc=new_loc, action=a, parent=node, g=g, h=h)
            node.add_child(child)

def back_propagation(node):
    """
    回溯并记录节点路径
    :param node: 待回溯节点
    :return: 回溯路径
    """
    path = []
    while node.parent is not None:
```

```
        path.insert(0, node.to_this_action
)
        node = node.parent
    return path

def a_star_search(maze):
    """
    使用 A*算法对迷宫进行路径搜索
    :param maze: 待搜索的 maze 对象
    :return: 从起点到目标点的路径
    """
    start = maze.sense_robot()
    goal = maze.destination
    root = SearchTree(loc=start, g=0, h=he
uristic(start, goal))

    open_list = []  # 优先队列存储待扩展节点
    heapq.heappush(open_list, (root.f(), r
oot))

    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

    while open_list:
        _, current_node = heapq.heappop(op
en_list)
        is_visit_m[current_node.loc] = 1
# 标记当前节点位置已访问

        if current_node.loc == goal:  # 到
达目标点
            return back_propagation(curren
t_node)

        if current_node.is_leaf():
            expand(maze, is_visit_m, curre
nt_node, goal)

        for child in current_node.children
:
            if not is_visit_m[child.loc]:
                heapq.heappush(open_list,
(child.f(), child))

    return []  # 未找到路径
```

这里，其启发式函数 h(n)为普通的曼哈顿距离。算法会从 open_list 中取出 f(n) 最小的节点。如果当前节点位置等于目标位置，回溯输出路径。否则，扩展当前节点的子节点，并将所有未访问的子节点加入 open_list，从而实现 A*算法。

2.4 实现 DQNRobot

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。

相较于传统根据 Q 值表估计 Q 值的方法，本报告采用 Deep-QLearning，以神经网络代替。用 PyTorch 实现：

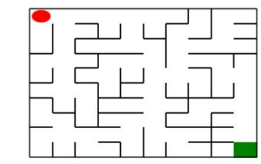
```
class Robot(TorchRobot):
    def __init__(self, maze):
        super().__init__(maze)
        # 设迷宫奖励，负向强化目的地奖励以突出
目标重要性
        maze.set_reward({"hit_wall": 10., "
destination": -maze.maze_size ** 2 * 10, "
default": 1.})
        self.maze = maze
        self.epsilon = 0
        # 建全图视野加速学习决策
        self.memory.build_full_view(maze)
        self.train()

    def train(self):
        # 持续训练直至成功出迷宫
        while True:
            self._learn(batch=len(self.mem
ory))
            self.reset()
            for _ in range(self.maze.maze_
size ** 2):
```

```
                action, reward = self.test_update()
                if reward == self.maze.reward["destination"]:
                    return

    def train_update(self):
        # 依状态选动作获奖励
        state = self.sense_state()
        action = self._choose_action(state)
        reward = self.maze.move_robot(action)
        return action, reward

    def test_update(self):
        # 转状态为张量算 Q 值选动作获奖励
        state = np.array(self.sense_state(), dtype=np.int16)
        state = torch.from_numpy(state).float().to(self.device)
        with torch.no_grad():
            q_value = self.eval_model(state).cpu().data.numpy()
        action = self.valid_action[np.argmin(q_value).item()]
        reward = self.maze.move_robot(action)
        return action, reward
```

3. 结果分析

3.1 基础算法

在自己测试时，本报告采用了经典的 10*10 迷宫，效果良好。

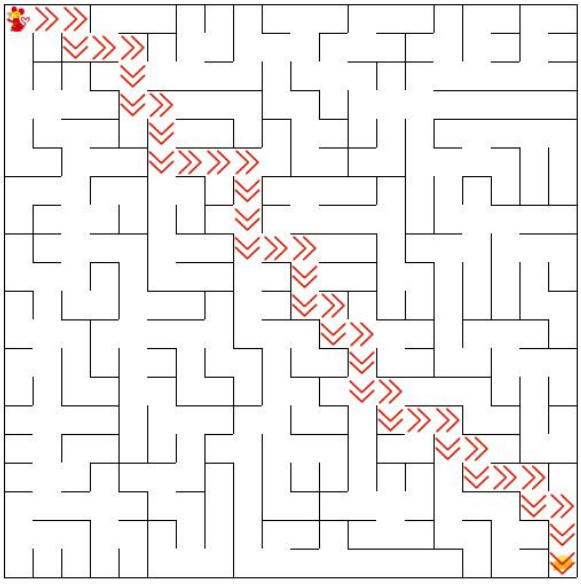


```
Maze of size (10, 10)
搜索出的路径： ['r', 'r', 'r', 'r', 'd', 'd', 'r', 'd', 'd', 'r', 'u', 'r', 'r', 'd', 'r', 'd', 'd', 'd', 'd', 'd']
恭喜你，到达了目标点
```

在实机测试时，其表现也非常优异，使用时长达到了惊人的 0s。

测试基础搜索算法	✓	0s	恭喜, 完成了迷宫

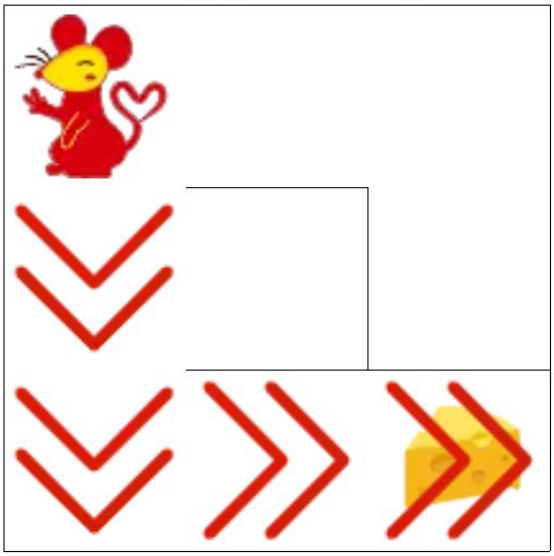
基础搜索算法 (Victory)



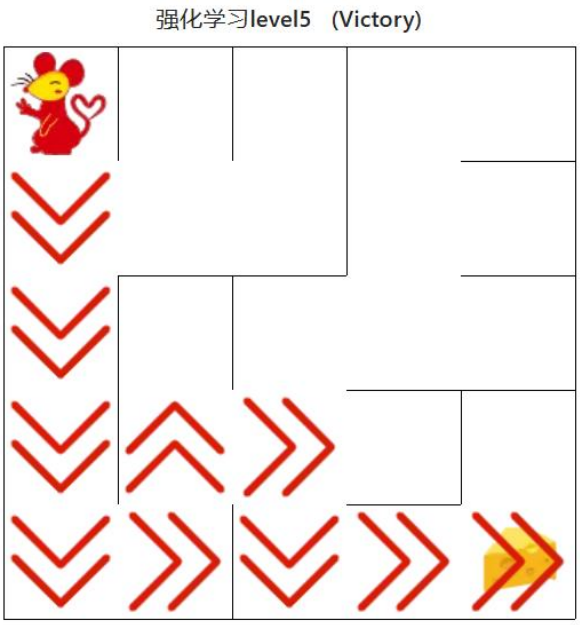
3.2 DQN

DQN 由于需要较长时间的训练过程，相对以上算法要慢很多。但是对于比较简单的迷宫，其仍然能够胜任：

强化学习level3 (Victory)

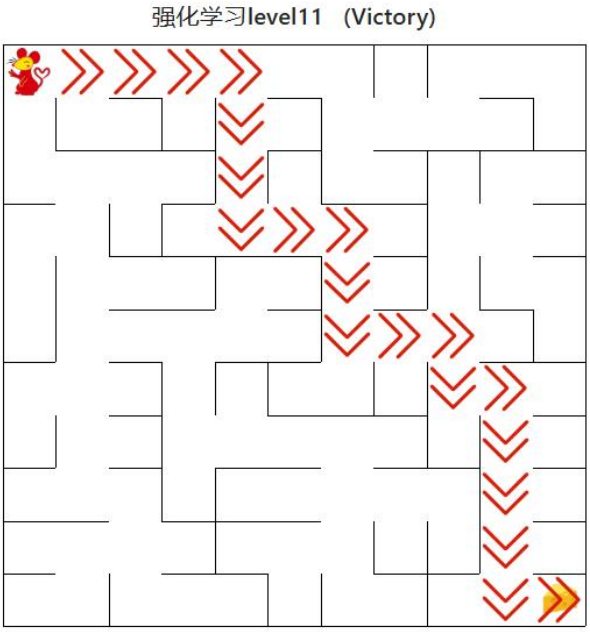


测试强化学习算法(初级)	✓	0s	恭喜, 完成了迷宫



测试强化学习算法(中级)	✓	1s	恭喜, 完成了迷宫

而对于比较复杂的迷宫，其运行时间明显增大，达到了比较惊人的程度，不过最后还是完成了任务：



测试强化学习算法(高级)	✓	273s	恭喜, 完成了迷宫

4. 总结与讨论

针对 DQN 的算法问题，有数种改进方向。传统的 DQN 在评估 Q 值时可能会高估，从而导致不稳定的训练结果。使用 Double DQN 可以有效解决这个问题。方法为增加目标网络 (target_model)，使用目标网络来计算目标 Q 值，避免直接使用评估网络。也可以使用分布式强化学习，让多个代理并行探索迷宫并共享经验，可以大大加速训练。根据迷宫的复杂程度和运行环境，可以逐步实现这些改进方案，优先从最能提升性能的地方（如分布式 DQN）入手。