程序报告

学号：2111460　　姓名：张洋

一、 问题重述

从起点开始，通过错综复杂的迷宫，到达目标点。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

本实验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。

二、 设计思想

2.1 基础算法

广度优先搜索

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。

深度优先搜索

1. 建立根节点：以机器人的起始位置作为根节点，并标记为已访问。
2. 深度优先搜索：从根节点开始，不断向某一个方向移动，直到无法再继续移动或到达目标位置。如果到达目标位置，则终止搜索，并记录回溯路径。
3. 回溯：如果无法继续移动或到达目标位置，退回到上一个节点，选择另一未访问过的方向进行搜索，直到找到路径或遍历完所有可能的路径。
4. 标记已访问节点：在搜索过程中，需要标记已访问的节点，以避免重复访问。

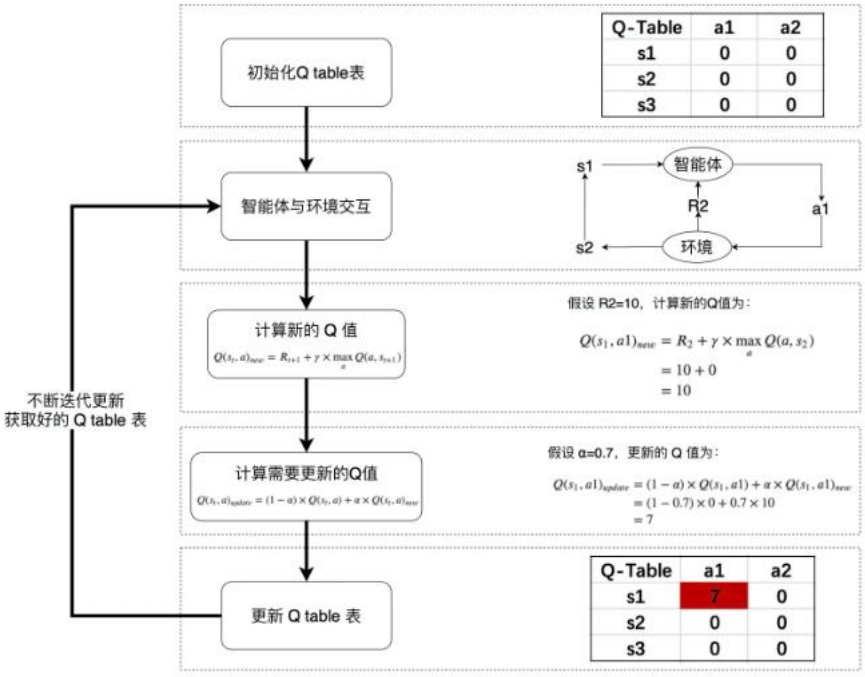
最佳优先搜索（A*算法）

1. 定义启发函数：为了确定节点的搜索优先级，需要定义一个启发函数，用于估计从当前节点到目标节点的距离或代价。
2. 建立根节点：以机器人的起始位置作为根节点，并计算启发函数的值。
3. A*算法搜索：从根节点开始，每次选择优先级最高的节点进行搜索，直到找到目标节点或搜索完所有可能的节点。

4. 扩展节点：对于选定的节点，生成其子节点，并计算子节点的启发函数值。
5. 更新优先级队列：将新生成的子节点加入优先级队列，并根据启发函数值进行排序，以确保优先处理具有更低代价的节点。
6. 回溯和路径记录：如果找到目标节点，终止搜索，并记录回溯路径。

2.2 Deep Qlearning 算法

Qlearning 算法



1. 获取机器人所处迷宫位置
2. 对当前状态，检索 Q 表，如果不存在则添加进入 Q 表

在 Q-Learning 算法中，将这个长期奖励记为 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:

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

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。

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

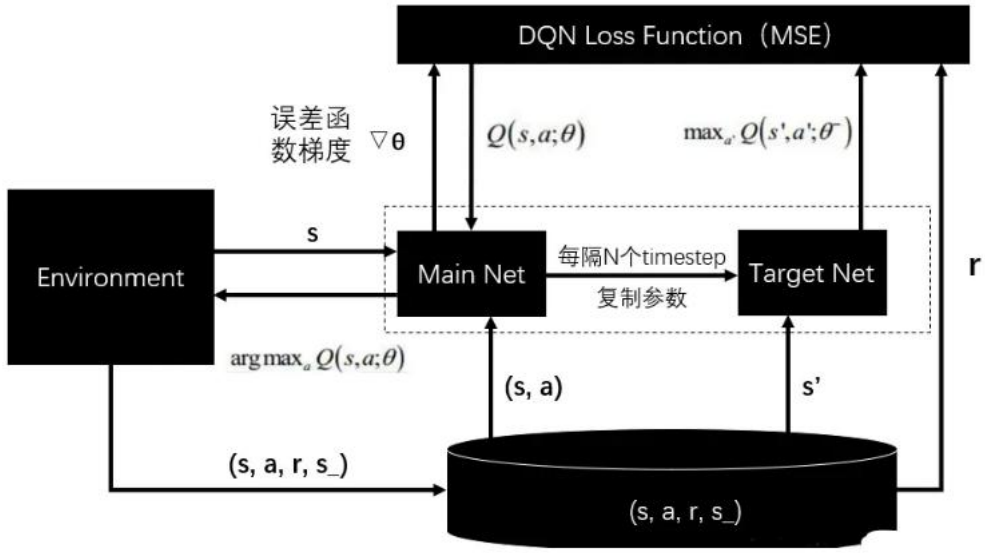
3. 选择动作

为了防止出现因机器人每次都选择它认为最优的路线而导致的路线固定的现象，通常采用 epsilon-greedy 算法:

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。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. 以给定的动作（移动方向）移动机器人
5. 获取机器人执行动作后所处的位置
6. 对当前 next_state ，检索 Q 表，如果不存在则添加进入 Q 表
7. 更新 Q 表中 Q 值以及其他参数

Deep Qlearning 算法

- **DQN 算法框架图**



1. `train_update`函数：
   - 初始化深度神经网络 Q 网络，并定义超参数（学习率、折扣因子、探索率、经验回放缓冲区大小等）。
   - 初始化目标网络 Q_target，并将其与 Q 网络参数同步。
   - 初始化经验回放缓冲区。
   - 迭代进行以下步骤：
     - 在当前状态 s 输入 Q 网络，得到每个动作的 Q 值。
     - 使用ε-greedy 策略选择动作 a。
     - 执行动作 a 并观察奖励 r 和下一个状态 s'。
     - 将(s, a, r, s')存储到经验回放缓冲区。
     - 从经验回放缓冲区中随机采样一批数据。
     - 对于每个样本，计算目标 Q 值：
       - 在下一个状态 s'输入 Q_target 网络，得到每个动作的 Q 值。
       - 根据 Bellman 方程计算目标 Q 值：targetQ(s, a) = r + γ * max(Q_target(s', a'))
     - 将当前状态 s 输入 Q 网络，得到每个动作的 Q 值。
     - 计算 Q 值与目标 Q 值之间的均方差损失函数。
     - 使用梯度下降方法更新 Q 网络的权重。
     - 定期（每隔一定步数）将 Q 网络参数复制到 Q_target 网络。

2. `test_update`函数：
   - 使用已训练好的 Q 网络进行测试。
   - 从起始位置开始，重复以下步骤直到到达迷宫的出口或达到最大步数：
     - 在当前状态 s 输入 Q 网络，得到每个动作的 Q 值。
     - 选择具有最大 Q 值的动作 a。
     - 执行动作 a 并观察奖励 r 和下一个状态 s'。
     - 更新当前状态 s 为下一个状态 s'。
     - 检查是否到达迷宫的出口，如果是则终止测试。
   - 输出测试的路径或到达迷宫出口的结果。

三、 代码内容

3.1 基础算法

```
# 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
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

# 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
}


# 迷宫路径搜索树
class SearchTree(object):


    def __init__(self, loc=(), action='', parent=None):
        """
        初始化搜索树节点对象
        :param loc: 新节点的机器人所处位置
        :param action: 新节点的对应的移动方向
        :param parent: 新节点的父辈节点
```

```
        """

        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

    def add_child(self, child):
        """
        添加子节点
        :param child:待添加的子节点
        """
        self.children.append(child)

    def is_leaf(self):
        """
        判断当前节点是否是叶子节点
        """
        return len(self.children) == 0


def expand(maze, is_visit_m, node):
    """
    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
    """
    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]:
            child = SearchTree(loc=new_loc, action=a, parent=node)
            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 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",...]
    """
    start = maze.sense_robot()
    root = SearchTree(loc=start)
    queue = [root]  # 节点队列，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
    while True:
        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
```

3.2 DQN 算法

```
import random
from QRobot import QRobot

class Robot(QRobot):

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

```
    def __init__(self, maze, alpha=0.5, gamma=0.9, epsilon=0.5):
        """
        初始化 Robot 类
        :param maze:迷宫对象
        """
        self.maze = maze
        self.state = None
        self.action = None
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon  # 动作随机选择概率
        self.q_table = {}

        self.maze.reset_robot()  # 重置机器人状态
        self.state = self.maze.sense_robot()  # state 为机器人当前状态

        if self.state not in self.q_table:  # 如果当前状态不存在，则为 Q 表
添加新列
            self.q_table[self.state] = {a: 0.0 for a in self.valid_action}

    def train_update(self):
        """
        以训练状态选择动作并更新 Deep Q network 的相关参数
        :return :action, reward 如："u", -1
        """
        self.state = self.maze.sense_robot()  # 获取机器人当初所处迷宫位置

         # 检索 Q 表，如果当前状态不存在则添加进入 Q 表
        if self.state not in self.q_table:
            self.q_table[self.state] = {a: 0.0 for a in self.valid_action}
        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)  # action 为机器人选择的动作
        reward = self.maze.move_robot(action)  # 以给定的方向移动机器
人,reward 为迷宫返回的奖励值
        next_state = self.maze.sense_robot()  # 获取机器人执行指令后所处的
位置

        # 检索 Q 表，如果当前的 next_state 不存在则添加进入 Q 表
        if next_state not in self.q_table:
            self.q_table[next_state] = {a: 0.0 for a in self.valid_action}

        # 更新 Q 值表
```

```
        current_r = self.q_table[self.state][action]
        update_r = reward + self.gamma *
float(max(self.q_table[next_state].values()))
        self.q_table[self.state][action] = self.alpha *
self.q_table[self.state][action] +(1 - self.alpha) * (update_r -
current_r)

        self.epsilon *= 0.5  # 衰减随机选择动作的可能性

        return action, reward

    def test_update(self):
        """
        以测试状态选择动作并更新 Deep Q network 的相关参数
        :return : action, reward 如："u", -1
        """
        self.state = self.maze.sense_robot()  # 获取机器人现在所处迷宫位置

        # 检索 Q 表，如果当前状态不存在则添加进入 Q 表
        if self.state not in self.q_table:
            self.q_table[self.state] = {a: 0.0 for a in self.valid_action}

        action =
max(self.q_table[self.state],key=self.q_table[self.state].get)  # 选择
动作
        reward = self.maze.move_robot(action)  # 以给定的方向移动机器人

        return action, reward
```

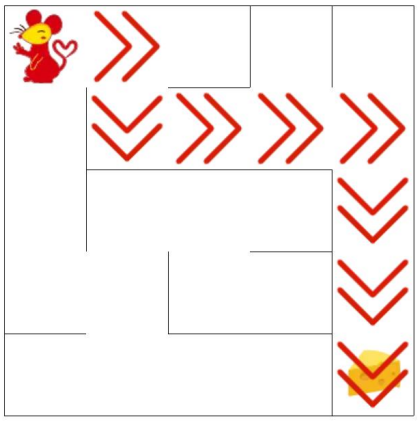
四、实验结果

测试详情 展示迷宫

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

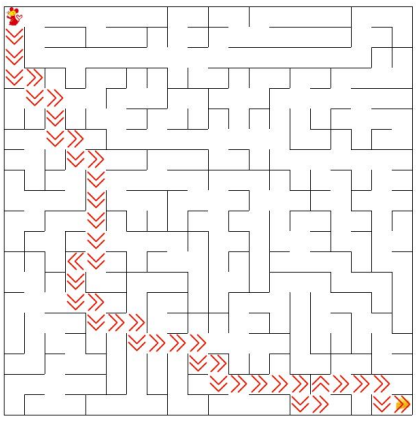
确定

强化学习level5 (Victory)



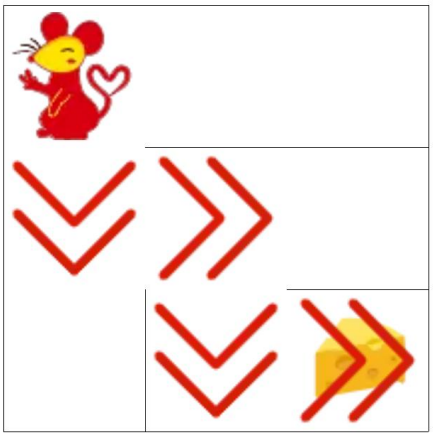
1 / 4

基础搜索算法 (Victory)



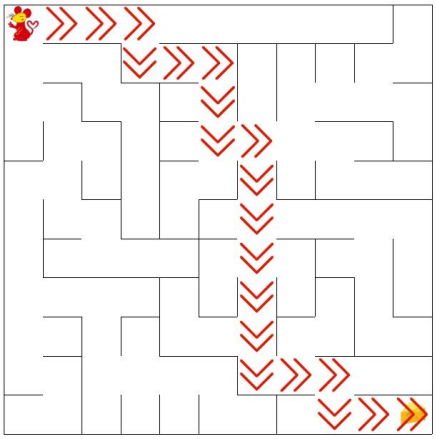
2 / 4

强化学习level3 (Victory)



3 / 4

强化学习level11 (Victory)



4 / 4

五、总结

1. 达到目标预期：

- 在实验中，我使用了广度优先搜索和 DQN 算法来解决机器人走迷宫问题。广度优先搜索算法可以找到最短路径，而 DQN 算法可以学习如何在迷宫中找到最佳路径。
- 通过实验，我发现广度优先搜索算法可以找到迷宫的最短路径，并且在小规模迷宫上表现良好。DQN 算法可以学习到在大规模迷宫中的最佳路径，并且具有一定的泛化能力。

2. 可能改进的方向：

- 调整 DQN 算法的超参数：例如学习率、折扣因子、探索率等，通过调整这些参数可以改善算法的学习性能。
- 使用更复杂的神经网络结构：例如深度卷积神经网络（CNN）或循环神经网络（RNN），这些网络结构可以更好地处理迷宫问题中的空间关系和序列信息。
- 引入经验回放机制：经验回放可以提高数据的利用效率，通过随机采样以及去除数据之间的相关性，可以改善 DQN 算法的学习稳定性。
- 探索策略的改进：可以尝试使用更高级的探索策略，如ε-greedy 策略的改进版本，或者采用其他的探索方法，如 UCB 探索等。

3. 实现过程中遇到的困难：

- 在实现过程中，算法的调参、代码的编写都具有一定难度。

4. 提升性能的方面：

- 提升性能的方面包括：
  - 增加训练时间和迭代次数：更长的训练时间和更多的迭代次数可以使模型更好地学习到迷宫的最佳路径，尤其是对于复杂的迷宫。
- 增加模型的复杂度：使用更复杂的神经网络结构，如深层网络、卷积神经网络（CNN）、循环神经网络（RNN）等，可以提升模型对迷宫问题的建模能力。
- 使用更大的经验回放缓冲区：增加经验回放缓冲区的容量可以提高数据的利用效率，使模型能够更好地学习到经验。
- 融合其他强化学习算法：考虑将其他强化学习算法与 DQN 结合使用，如 Double DQN、Dueling DQN、A3C 等，这些算法可能有助于提升性能。

5. 模型的超参数和框架搜索是否合理：

- 在实现过程中，超参数的选择通过试验来找到最佳的组合。

总的来说，通过使用广度优先搜索和 DQN 算法，我能够在机器人走迷宫问题中得到了正确的结果。通过对超参数和模型结构的调整，可以进一步提升算法的性能。然而，还有许多改进的空间，包括调整超参数、改进探索策略、引入更复杂的网络结构等。同时，模型的训练时间和训练迭代次数也是影响性能的重要因素。进一步的实验和调优可以得到更好的结果。