程序报告

学号：　　2113997　　姓名：齐明杰

一、问题重述

（简单描述对问题的理解，从问题中抓住主干，必填）

==========

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。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。

要求：

使用 Python 语言。

使用基础搜索算法完成机器人走迷宫。

使用 Deep QLearning 算法完成机器人走迷宫。

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

（所采用的方法，有无对方法加以改进，该方法有哪些优化方向（参数调整，框架调整，或者指出方法的局限性和常见问题），伪代码，理论结果验证等… **思考题，非必填**）

==========

本项目的主要目标是通过强化学习让机器人在一个迷宫环境中找到最短路径。在设计实验时，我们主要采用了深度优先搜索（DFS）和深度 Q 学习（Deep Q-Learning）的方法。

深度优先搜索（DFS）：我们使用 DFS 来搜索迷宫并找到出口。DFS 是一种用于遍历或搜索树或图的算法。这种方法是沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点 v 的所有边都已被探寻过，搜索将回溯到发现节点 v 的那条边的起始节点。这个过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。在本实验中，DFS 被用来指导机器人通过迷宫，找到从起点到终点的路径。

深度 Q 学习（Deep Q-Learning）：我们使用了深度 Q 学习来使机器人学习在迷宫环境中做出最优决策。深度 Q 学习是 Q 学习和深度神经网络的结合。在我们的迷宫任务中，深度 Q 网络（DQN）试图学习一个策略，使得机器人可以根据其当前的观察来采取能够最大化未来奖励的行动。在训练过程中，机器人会根据其经验（状态、行动、奖励）来更新 Q 值，以此来学习一个优化策略。

优化方向：对于 DFS，可以结合其他搜索策略如广度优先搜索（BFS）或 A*搜索来提高搜索效率。对于 DQN，可能的优化方向包括使用双 DQN、优先经验回放等改进算法。此外，调整超参数（如学习率、折扣因子等）和网络结构（如增加隐藏层或改变激活函数）也可能会有助于提高性能。

局限性：DFS 在找出一条路径后就会停止，可能并不是最优路径。而 DQN 在面对复杂

的迷宫和大量的状态时可能会遇到难以学习的问题，需要大量的训练周期和计算资源。

三、代码内容

（能体现解题思路的主要代码，有多个文件或模块可用多个"===="隔开，必填）

Main.py

```
==========================================================================
import random
import numpy as np
import torch
from QRobot import QRobot
from ReplayDataSet import ReplayDataSet
from torch_py.MinDQNRobot import MinDQNRobot as TorchRobot # PyTorch 版本
import matplotlib.pyplot as plt
from Maze import Maze
import time
from Runner import Runner

def my_search(maze):
    # 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
        """
        child_number = 0   # 记录叶子节点个数
        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)
                child_number+=1
        return child_number   # 返回叶子节点个数

    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 myDFS(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

            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
                is_visit_m[current_node.loc] = 1  # 标记该点已拓展
                child_number = expand(maze, is_visit_m, current_node)
                peek+=child_number  # 开展一些列入栈操作
                for child in current_node.children:
                    queue.append(child)  # 叶子节点入栈
            else:
                queue.pop(peek)  # 如果无路可走则出栈
                peek-=1
            # 出队
            #queue.pop(0)

        return path
    path = myDFS(maze)
    return path


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": 5.0,
            "destination": -maze.maze_size ** 2.0,
```

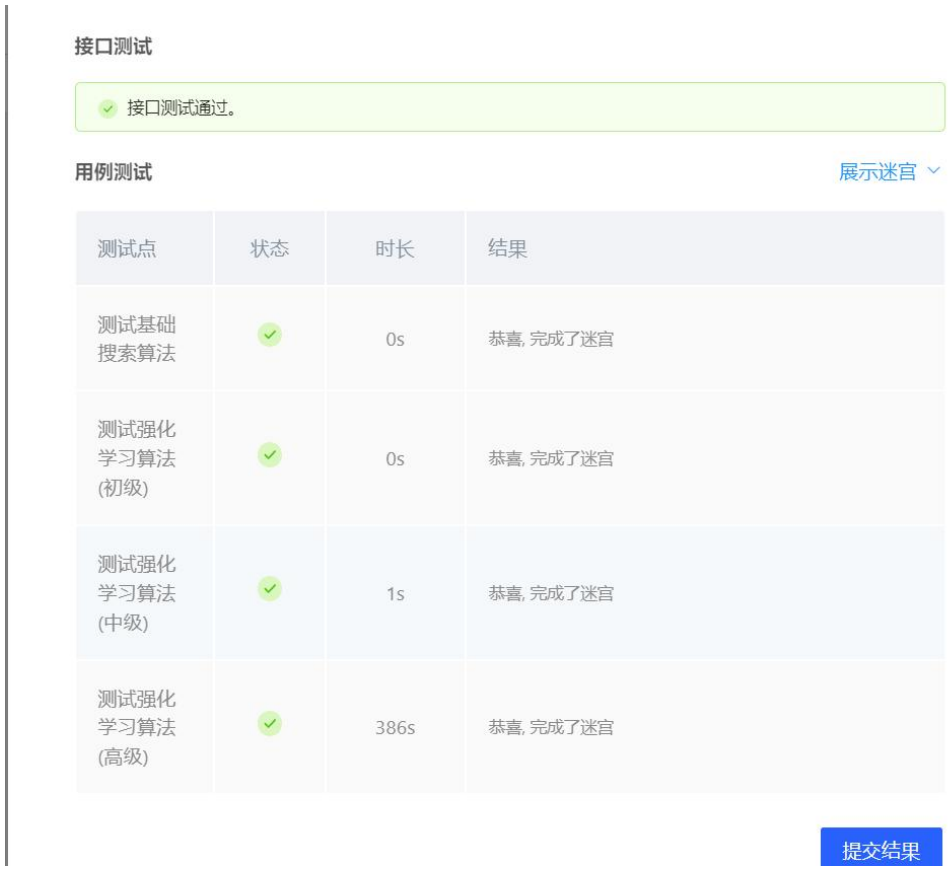
```
                "default": 1.0,
            })
            self.maze = maze
            self.epsilon = 0
            """开启金手指，获取全图视野"""
            self.memory.build_full_view(maze=maze)
            self.loss_list = self.train()

        def train(self):
            loss_list = []
            batch_size = len(self.memory)

            while True:
                loss = self._learn(batch=batch_size)
                loss_list.append(loss)
                success = False
                self.reset()
                for _ in range(self.maze.maze_size ** 2 - 1):
                    a, r = self.test_update()
                    if r == self.maze.reward["destination"]:
                        return loss_list

        def train_update(self):
            def state_train():
                state=self.sense_state()
                return state
            def action_train(state):
                action=self._choose_action(state)
                return action
            def reward_train(action):
                reward=self.maze.move_robot(action)
                return reward
            state = state_train()
            action = action_train(state)
            reward = reward_train(action)
            return action, reward

        def test_update(self):
            def state_test():
                state = torch.from_numpy(np.array(self.sense_state(),
dtype=np.int16)).float().to(self.device)
                return state
            state = state_test()
            self.eval_model.eval()
```

```
        with torch.no_grad():
            q_value = self.eval_model(state).cpu().data.numpy()
        def action_test(q_value):
            action=self.valid_action[np.argmin(q_value).item()]
            return action
        def reward_test(action):
            reward=self.maze.move_robot(action)
            return reward
        action = action_test(q_value)
        reward = reward_test(action)
        return action, reward
```

四、实验结果

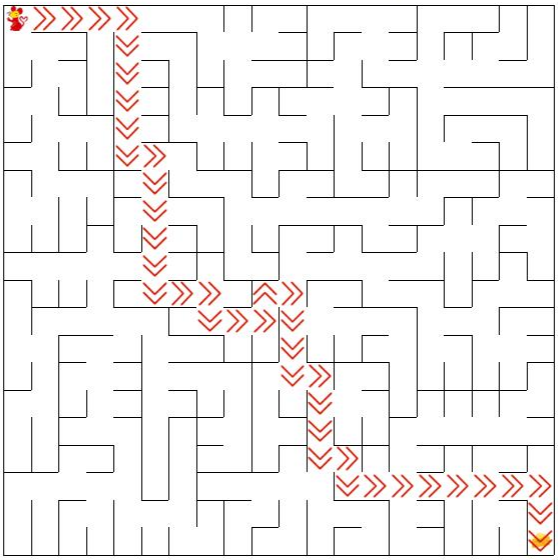
（实验结果，必填）

==========



用例测试 隐藏迷宫

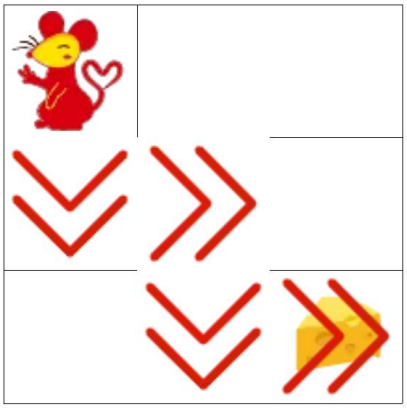
基础搜索算法 (Victory)



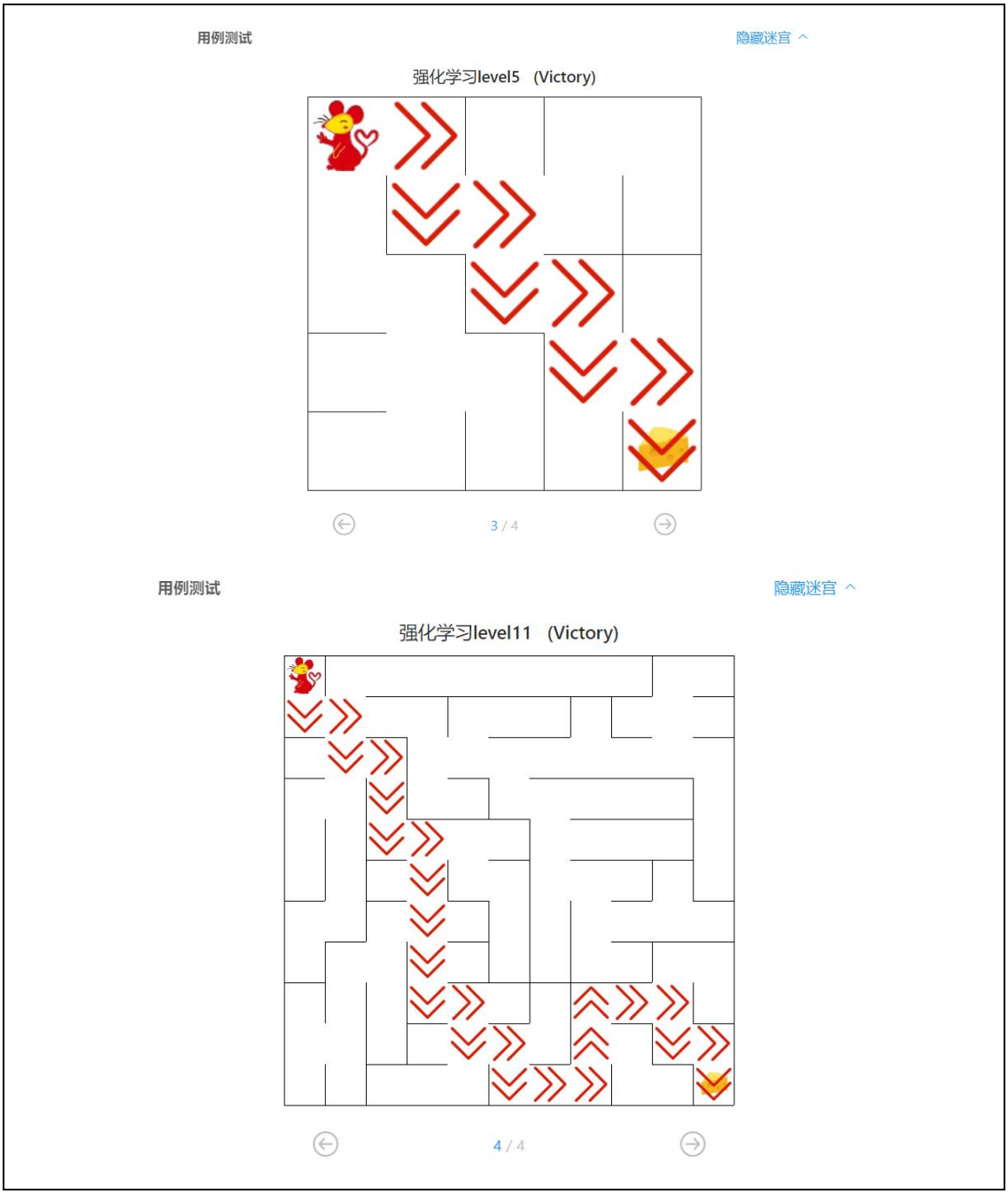
1 / 4

用例测试 隐藏迷宫

强化学习level3 (Victory)



2 / 4



五、总结

（自评分析（是否达到目标预期，可能改进的方向，实现过程中遇到的困难，从哪些方面可以提升性能，模型的超参数和框架搜索是否合理等），**思考题，非必填**）

==========

在本次实验中，我成功地通过深度优先搜索和深度 Q 学习让机器人找到了从迷宫入口到出口的路径。虽然已经取得了一些进展，但我认为还有许多地方可以进行改进。

代码优化：虽然代码已经可以正常运行并且结果正确，但是我认为代码结构和注释可以进一步优化，使得代码更容易阅读和理解。

实验设置：我认为我可以通过改变迷宫的大小和复杂度，以及调整训练周期和超参数，来进一步探索 DFS 和 DQN 的性能。

算法改进：虽然我在本实验中使用了 DFS 和 DQN，但我认为还可以尝试其他的搜索策略和强化学习算法。例如，可以尝试使用蒙特卡洛树搜索（MCTS）或者 Actor-Critic 方法。

训练方法：我认为可以进一步优化训练方法。例如，可以引入更复杂的奖励机制，或者利用先验知识来引导机器人的学习。

参数调整：虽然我已经进行了一些参数调整，但我认为可以进一步尝试不同的参数组合，来找到最优的参数设置。

超参数和框架搜索：我认为可以尝试使用自动化的方法（如贝叶斯优化）来进行超参数调优，也可以尝试使用不同的网络结构和训练策略。