程序报告

学号：2112514　　姓名：辛浩然

1　问题重述

机器人走迷宫问题，即在给定起点、终点和迷宫结构的情况下，通过搜索算法或强化学习来找到从起点到终点的有效路径。

通过简单的搜索算法，机器人可以使用广度优先搜索、深度优先搜索或启发式搜索等方法来遍历迷宫的各个位置，逐步扩展探索的范围，直到找到连接起点和终点的路径。这些算法在迷宫规模较小、路径复杂度较低的情况下，能够有效地找到解决方案。

然而，在更复杂的迷宫结构中，强化学习可以发挥重要作用。通过建立一个智能体（机器人）和环境（迷宫）的交互模型，强化学习可以通过试错和学习的方式，自主地发现有效的路径。在训练过程中，机器人通过尝试不同的行动，并根据环境的奖励值进行反馈，逐渐优化其行为策略，直到能够准确地导航到终点。

强化学习中的奖励值在迷宫问题中起到关键作用。通常情况下，机器人会受到正向奖励，如到达终点或走近终点的路径，以及负向奖励，如碰到障碍物或走回头路等。通过合理设计奖励函数，可以引导机器人快速而准确地学习到最佳路径。

2　设计思想

深度优先搜索是通过递归地探索迷宫中的路径，从当前位置开始，沿着一个方向一直走到无法继续为止，然后回溯到上一个位置，选择下一个未探索的方向继续前进。这样，直到找到迷宫的出口或者所有路径都被探索完为止。DFS 的优点是简单易实现，但可能会陷入无限循环或者遍历冗余的路径。

Q-learning 是一种强化学习算法，用于训练机器人学习迷宫问题的最优策略。其设计思想是通过不断地与环境交互，机器人根据当前状态选择动作，并根据动作的奖励来更新策略。在迷宫问题中，机器人通过选择向上、向下、向左、向右等动作来移动，并且在每个位置根据奖励函数得到一个反馈。Q-learning 通过维护一个 Q 值表，记录每个状态-动作对的累计奖励，根据 Q 值表来进行策略选择和更新。通过不断的训练和学习，机器人能够找到最优的路径来解决迷宫问题。

3 代码内容

3.1 基础搜素算法

使用深度优先搜索方法找到有效路径。从迷宫的起始位置开始，递归地探索可能的路径寻找到达目标位置的路径。在搜索过程中，使用一个搜索树来记录已访问的节点，并通过回溯操作来回到父节点继续探索其他可能的路径。

具体来说，首先，根据迷宫的起始位置创建一个搜索树的根节点，并初始化一个标记数组，用于记录迷宫中的位置是否被访问过。接下来，对于根节点调用 dfs 函数。

dfs 函数首先将当前节点的位置标记为已访问，并检查该位置是否为目标位置。如果是目标位置，则表示已经找到了从起始位置到目标位置的路径，执行回溯操作来记录节点路径。在回溯操作中，从当前节点开始向上追溯到根节点，记录路径上的所有节点。这样，就得到了从起始位置到目标位置的路径。如果当前节点是叶子节点（没有子节点），则扩展该叶子节点。扩展操作意味着从当前位置可以继续探索新的路径。在扩展过程中，创建当前节点的子节点，并更新搜索树。对于当前节点的每个子节点，递归调用深度优先搜索函数，继续向下探索。当递归返回到父节点时，将当前节点的位置标记为未访问状态，以便在其他路径中重新访问该位置。

```
def dfs(maze,current_node,is_visit_m, path):
    is_visit_m[current_node.loc] = 1
    if current_node.loc == maze.destination:
        # 回溯并记录节点路径
        res = back_propagation(current_node)
        for items in res:
            path.append(items)
        return
    if current_node.is_leaf():
        # 拓展叶子节点
        expand(maze, is_visit_m, current_node)
    for child in current_node.children:
        dfs(maze,child,is_visit_m, path)
    is_visit_m[current_node.loc] = 0

def depth_first_search(maze):
    # 对迷宫进行深度优先搜索
    start = maze.sense_robot()
    root = SearchTree(loc=start)
    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
    dfs(maze,root,is_visit_m,path)
    return path

def my_search(maze):
    path = depth_first_search(maze)
    return path
```

3.2 Deep QLearning 算法

Robot 类的初始化方法：传入一个迷宫对象 maze，调用父类 TorchRobot 的初始化方法。然后，通过 maze.set_reward() 方法设置迷宫的奖励值，包括撞墙的奖励、到达目的地的奖励和默认奖励值。接着，初始化 epsilon 值，用于 epsilon-greedy 策略中的随机探索概率。

train() 方法用于训练机器人。调用 _learn() 方法进行批量训练，并返回损失值，将损失值添加到 loss_list 列表中。然后，调用 reset() 方法重置机器人的状态和环境。接下来，使用循环遍历迷宫中的每个步骤，调用 test_update() 方法执行测试更新，并返回动作和奖励值。如果奖励值等于目标奖励，即到达目的地，就返回 loss_list 作为训练过程中的损失值记录。

train_update() 方法用于训练更新。它首先通过 sense_state() 方法获取当前状态，然后通过 _choose_action() 方法根据当前状态选择动作，在迷宫中移动机器人并计算奖励，最后返回动作和奖励值。

test_update() 方法用于测试更新。它首先使用评估模型获取当前状态的 Q 值，然后根据 Q 值选择动作，根据动作获取奖励值，最后返回动作和奖励值。

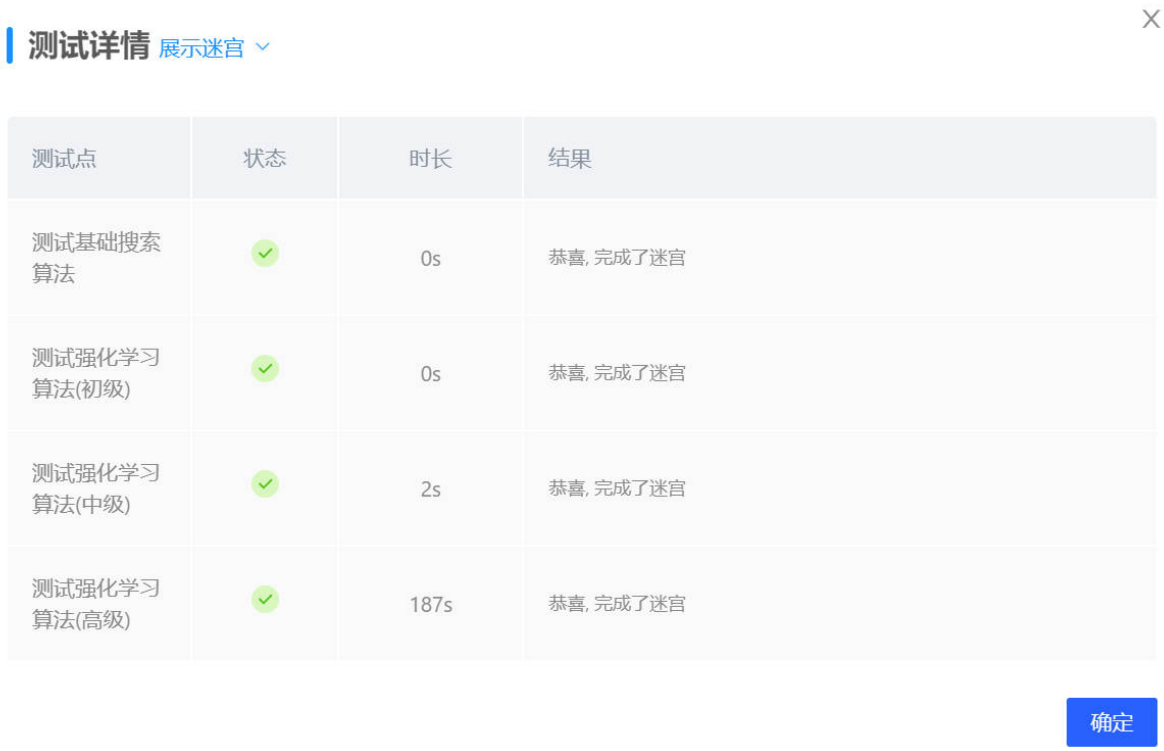
```
class Robot(TorchRobot):
    def __init__(self, maze):
        # 初始化Robot对象，传入迷宫对象maze作为参数
        super(Robot, self).__init__(maze)
        maze.set_reward(reward={
            "hit_wall": 50,
            "destination": -1000,
            "default": 1,
        })
        # 设置迷宫的奖励值
        self.epsilon = 0.1
        # 初始化epsilon值，用于epsilon-greedy策略中的随机探索概率
        self.maze = maze
        # 将迷宫对象赋值给Robot对象的属性maze
        self.memory.build_full_view(maze=maze)
```

```
        # 在内存中建立完整的迷宫视图，即将迷宫的状态存储在内存中，以便智能体进行学习和决策
        self.loss_list = self.train()
        # 调用train()方法进行训练，并将返回的损失值列表赋值给Robot对象的属性loss_list

    def train(self):
        # 训练方法
        loss_list = []  # 存储每次迭代的损失值的列表
        maze_size_squared = self.maze.maze_size ** 2

        while True:
            loss = self._learn(batch=len(self.memory))
            # 调用_learn()方法进行批量训练，并返回损失值
            loss_list.append(loss)
            # 将损失值添加到损失值列表中
            self.reset()
            # 重置智能体的状态和环境

            for step in range(maze_size_squared - 1):
                action, reward = self.test_update()
                # 执行测试更新，获取动作和奖励值

                if reward == self.maze.reward["destination"]:
                    # 如果奖励值等于目标奖励值
                    return loss_list
                    # 返回损失值列表作为训练过程中的损失值记录

    def train_update(self):
        # 训练更新方法
        state = self.sense_state()
        # 获取当前状态
        action = self._choose_action(state)
        # 根据当前状态选择动作
        reward = self.maze.move_robot(action)
        # 根据动作移动机器人并计算奖励
        return action, reward
```

```
        # 返回动作和奖励值

    def test_update(self):
        # 测试更新方法
        state = self.get_state()
        # 获取当前状态
        q_value = self.eval_model(state).cpu().data.numpy()
        # 使用评估模型获取当前状态的Q值，并将结果转换为numpy数组
        action = self.get_action(q_value)
        # 根据Q值选择动作
        reward = self.get_reward(action)
        # 根据动作获取奖励值
        return action, reward
        # 返回动作和奖励值

    def get_state(self):
        # 获取状态方法
        state = self.sense_state()
        # 获取当前状态
        return torch.from_numpy(np.array(state, dtype=np.int16)).float
    ().to(self.device)
        # 将状态转换为torch张量，并将其放置在指定设备上

    def get_action(self, q_value):
        # 获取动作方法
        action = self.valid_action[np.argmin(q_value).item()]
        # 根据Q值选择动作
        return action
        # 返回动作

    def get_reward(self, action):
        # 获取奖励值方法
        reward = self.maze.move_robot(action)
        # 根据动作移动机器人并计算奖励
        return reward
        # 返回奖励值
```

4 实验结果

提交代码进行测试，搜索算法能够正确运行，都能够完成迷宫。



5 总结

根据实验结果，编写的算法代码能够正确运行，基本达到预期结果。以下是一些改进方向：

1. 调整初始探索概率、奖励等值，更好地引导学习过程，提高性能；

2. 考虑使用改进的算法，比如 Double Q-Learning 算法。DQL 算法在训练过程中会产生估计误差，导致对 Q 值的估计不准确。Double Q-Learning 是一种改进的 Q-Learning 算法，可以减小这种估计误差。通过使用两个独立的 Q 网络来估计 Q 值，一个网络用于选择动作，另一个网络用于评估所选择的动作的值，可以减少过高估计的情况。