程序报告

学号：2213924　　姓名：申宗尚

一、问题重述

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。

如下图所示，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'。

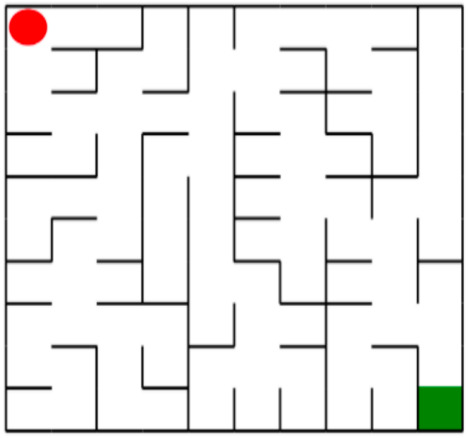
执行不同的动作后，根据不同的情况会获得不同的奖励，具体而言，有以下几种情况：

1.撞墙

2.走到出口

3.其余情况

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。



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。

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。。

二、设计思想

在本次实验中，给出给出 torch 框架和 keras 框架，可以任选一种实现。给出的文件如下：

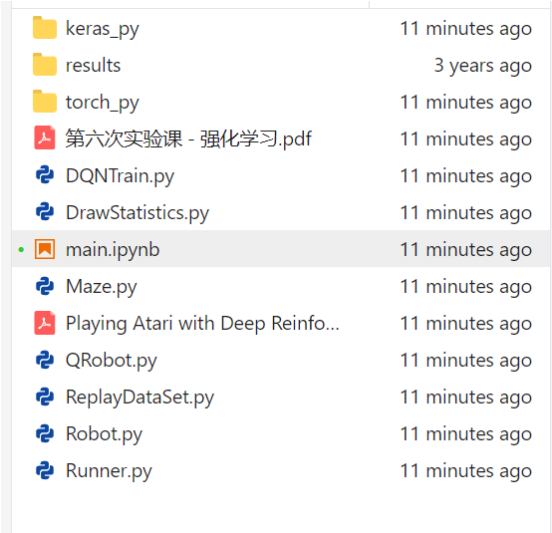
DQNTrain：训练代码，需要根据所选框架修改 robot = KerasRobot(maze=maze) 或者 robot = TorchRobot(maze=maze)

Maze：迷宫类定义

QRobot：Agent 类，定义智能体主要功能函数。

ReplayDataset：经验缓冲区，存储需要回放的经验

Runner: 用于 Agent 的训练和可视化。



Maze 类中重要的成员方法如下：

1. 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。

2. 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

3. 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']

4. 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)

5. draw_maze()：画出当前的迷宫

QRobot 类的核心成员方法

1. sense_state()：获取当前机器人所处位置

   return：机器人所处的位置坐标，如： (0, 0)

2. current_state_valid_actions()：获取当前机器人可以合法移动的动作

   return：由当前合法动作组成的列表，如： ['u','r']

3. train_update()：以训练状态，根据 QLearning 算法策略执行动作

   return：当前选择的动作，以及执行当前动作获得的回报，如： 'u', -1

4. test_update()：以测试状态，根据 QLearning 算法策略执行动作

   return：当前选择的动作，以及执行当前动作获得的回报，如：'u', -1

5. reset()： return：重置机器人在迷宫中的位置

Runner 类的核心成员方法

1. run_training(training_epoch, training_per_epoch=150)：训练机器人，不断更新 Q 表，并把训练结果保存在成员变量 train_robot_record 中

   training_epoch, training_per_epoch：总共的训练次数、每次训练机器人最多移动的步数

2. run_testing()：测试机器人能否走出迷宫

3. generate_gif(filename)：将训练结果输出到指定的 gif 图片中

   filename：合法的文件路径,文件名需以 `.gif` 为后缀

4. plot_results()：以图表展示训练过程中的指标：Success Times、Accumulated Rewards、Runing Times per Epoch

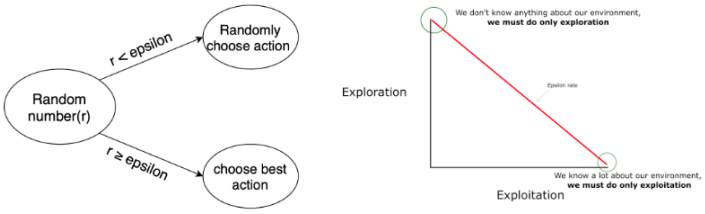
在本次实验中，我们主要采用 Q-Learning 算法：存储 Q 表格，行为状态，列动作，即二维数据 Q[s][a]代表在状态 s 下采取动作 a 可以获得多大的奖励。更新方式为动态规划。意为当前状态下采取动作 a 的价值为获得的即时奖励和进入下一个状态下所获得的长期奖励（最高的回报）。

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

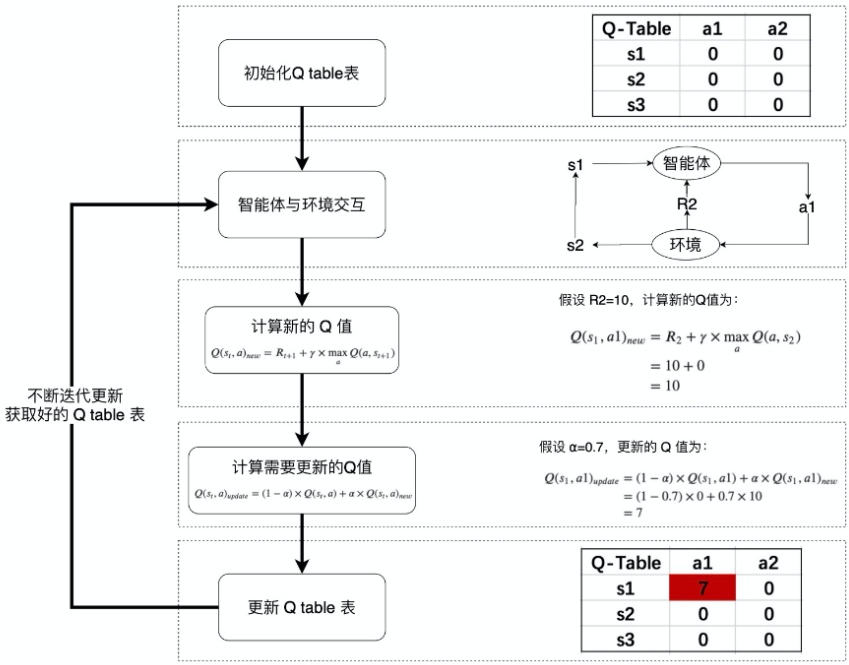
更新太过激进，导致出现震荡的现象，引入松弛变量。

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

以及 epsilon-greedy: 训练过程动作的选择并不是通过简单的贪心找最大实现的，我们通常会使用 epsilon-贪心策略，即在某一个状态下，有 epsilon 的概率选择其余的部分进行探索，有 1-epsilon 的概率选择最高的奖赏进行迭代，是超参数，为了平衡利用和探索，类似于 MCT 中的超参数 c。



综合以上几种算法，我们有本次实验的代码大致流程图如下：



在实验中，对于深度优先搜索，我们只需要用一个栈和一个记录位置的表，然后利用代码已经给出的广度优先搜索树即可完成。

对于 DQN 算法，我们主要实现微调 QRobot 类，并调整相应的超参数。

目前的 DQNRobot 仅使用两层全连接进行拟合 Q-Function，主要功能函数与给定代码完全类似，只需要调节参数和重新设计网络结构即可。

需要调整的参数如下：

1. QNetwork.py 的 QNetwork 类，其网络结构极为简单，可以调节为相对复杂的结构。增加其表征能力。

2. epoch 训练轮数

3. epsilong-greedy 中 epsilon，平衡探索与利用。

4. 设计的 reward，使用 maze.set_reward 方法。考虑到达终点的 reward；在轨迹中的 reward，撞墙的 reward。Reward Penalty：可以考虑将撞墙的 reward 调节相对小很多，给 model 较大的惩罚，使其避免撞墙。

5. reward discount：奖励折扣因子，在 0-1 之间，考虑长期奖励与短期奖励之间的关系。

在调整算法框架的时候，由于初始化与奖励机制、训练过程和深度学习模型都已经给出，我们着重实现以下几个操作：

1、训练更新：采取随机选择动作的方式，在每次执行完动作之后，更新 epsilon 值，然后减少探索率即可。

2、测试更新：编写代码，执行动作并获取奖励，测试是否到达目标点。

三、代码介绍

基础搜索算法部分：

```
def my_search(maze):
    start = maze.sense_robot()#起点
    root = SearchTree(loc=start)
    stack = [root]  # 节点栈，用于深度优先遍历
    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

    while stack:
        current_node = stack.pop()
        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
            return back_propagation(current_node)

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

        # 入栈
        for child in current_node.children:
            stack.append(child)

    return []  # 未找到路径
```

该函数函数接收一个迷宫对象 maze，并使用深度优先搜索（DFS）算法尝试找到从起点到终点的路径。如果找到路径，则返回该路径；如果没有找到路径，则返回一个空列表。

具体来说，首先先获取迷宫起点位置、创造搜索树的根位置为起点，使用栈储存待访问的节点，获取迷宫大小，并创造 is_visit 存储每个位置是否被访问。

然后使用一个循环来遍历栈中的节点，每次弹出一个节点，并且设置为已访问。检查当前节点是否为目标点。如果是，调用 back_propagation 函数从当前节点回溯到起点，生成并返回路径。再检查当前节点是否是叶子节点（即没有子节点）。如果是，则调用 expand 函数扩展节点，即生成当前节点的所有可能的子节点。将当前节点的所有子节点依次入栈，以便在后续的循环中处理它们。如果栈为空且未找到路径，返回一个空列表表示未找到路径。

从而实现一个简单的深度优先搜索框架。

DQN 算法部分：

```python
class Robot(TorchRobot):
    def __init__(self, maze):
        """
        初始化 Robot 类
        :param maze: 迷宫对象
        """
        super(Robot, self).__init__(maze)
        # destination 设置为和迷宫大小相关
        maze.set_reward(reward={
            "hit_wall": 10.,
            "destination": -maze.maze_size ** 2 * 4.,
            "default": 1.,
        })
        self.maze = maze
        self.epsilon = 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.loss_list = self.train()

    def train(self):
        loss_list = []
        batch_size = len(self.memory)
        # 训练，直到能走出这个迷宫
        while True:
            loss = self._learn(batch=batch_size)
            loss_list.append(loss)
            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
```

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

    """---update the step and epsilon---"""
    # self.epsilon = max(0.01, self.epsilon * 0.995)

    return action, reward

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

    self.eval_model.eval()
    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
```

Robot 类继承自 TorchRobot，用于在迷宫中寻找路径并进行训练以提高性能。该类的初始化方法设置了迷宫的奖励机制，并开启了“全图视野”功能，使机器人能够获取整个迷宫的视图。初始化后，机器人立即开始训练，通过调用 train 方法进行学习。

train 方法是训练的核心部分，它不断进行学习，直到机器人能够成功走出迷宫。每次训练迭代中，机器人会通过调用_learn 方法进行学习，计算损失并重置状态，然后根据迷宫大小进行多次更新尝试。如果机器人达到了目标位置，则训练结束并返回损失列表。

train_update 方法用于在训练过程中选择动作并获取奖励。它通过感知当前状态，选择动作并移动机器人，然后返回执行的动作和获得的奖励。

test_update 方法用于在测试过程中选择动作并获取奖励。它首先将当前状态转换为张量，然后使用评估模型计算每个动作的 Q 值，选择最小 Q 值对应的动作，并移动机器人获取奖励。此方法在测试过程中不进行模型训练。

四、代码内容

下面给出本次的训练代码：

```python
# 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
import torch
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

import numpy as np

# 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 breadth_first_search(maze):
    """
    对迷宫进行广度优先搜索
    :param maze: 待搜索的 maze 对象
    """
    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
```

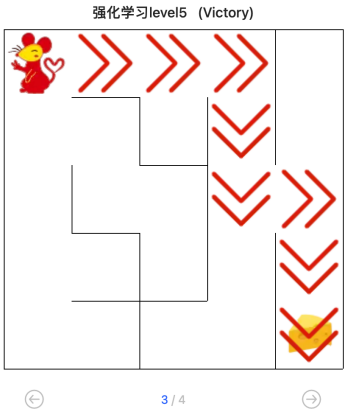
```
    path = []  # 记录路径
    while True:
        current_node = queue[0]
        is_visit_m[current_node.loc] = 1  # 标记当前节点位置已访问

        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

def my_search(maze):
    start = maze.sense_robot()#起点
    root = SearchTree(loc=start)
    stack = [root]  # 节点栈，用于深度优先遍历
    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

    while stack:
        current_node = stack.pop()
        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
            return back_propagation(current_node)

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

        # 入栈
        for child in current_node.children:
            stack.append(child)

    return []  # 未找到路径
```

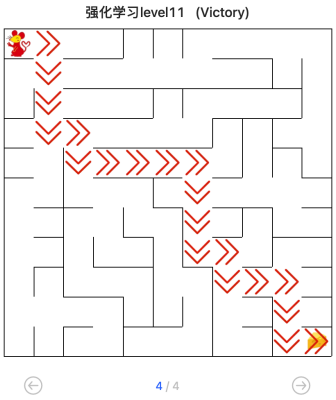
```python
class Robot(TorchRobot):
    def __init__(self, maze):
        """
        初始化 Robot 类
        :param maze: 迷宫对象
        """
        super(Robot, self).__init__(maze)
        # destination 设置为和迷宫大小相关
        maze.set_reward(reward={
            "hit_wall": 10.,
            "destination": -maze.maze_size ** 2 * 4.,
            "default": 1.,
        })
        self.maze = maze
        self.epsilon = 0
        self.memory.build_full_view(maze=maze)
        # 初始化后即开始训练
        self.loss_list = self.train()

    def train(self):
        loss_list = []
        batch_size = len(self.memory)
        # 训练，直到能走出这个迷宫
        while True:
            loss = self._learn(batch=batch_size)
            loss_list.append(loss)
            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):
        state = self.sense_state()
        action = self._choose_action(state)
        reward = self.maze.move_robot(action)

        """---update the step and epsilon---"""
        # self.epsilon = max(0.01, self.epsilon * 0.995)

        return action, reward

    def test_update(self):
        state = np.array(self.sense_state(), dtype=np.int16)
```

```
        state = torch.from_numpy(state).float().to(self.device)

        self.eval_model.eval()
        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
```

五、实验结果

全部通过，通过测试。





六、总结

在本次实验中，通过实施基础搜索算法和 Deep Q-Learning 算法，成功地解决了机器人走迷宫的问题。

首先，采用了深度优先搜索（DFS）算法来寻找从起点到终点的路径。DFS 通过使用栈来存储待访问的节点，并通过递归方式进行遍历，最终成功实现了路径的寻找。实验中设计了一个 SearchTree 类，帮助我们构建了搜索树，并实现了节点的拓展和路径回溯功能。

通过实验发现，DFS 在迷宫较为复杂的情况下，可能会因为过多的回溯而导致效率较低。不过，对于相对简单的迷宫，DFS 仍然是一个高效且易于实现的搜索算法。

在 Deep Q-Learning 算法部分，选择 PyTorch 作为深度学习框架，继承和改进了给定的 TorchRobot 类。通过构建一个简单的两层全连接网络，我们实现了 Q 值的近似计算。此外，我们还实现了训练过程中的 epsilon-greedy 策略，平衡了探索与利用。

在训练过程中，通过不断调整超参数，包括训练轮数、epsilon 值、奖励机制和奖励折扣

因子，最终使得机器人能够成功地走出迷宫。实验中发现，适当的奖励设计对训练效果有显著影响，特别是对撞墙的惩罚和到达终点的奖励设置，能够有效地引导机器人避开障碍物并寻找最优路径。

通过实验，我们成功地使机器人在给定的迷宫中找到了到达终点的路径，并且在多次实验中验证了算法的稳定性和有效性。基础搜索算法在简单迷宫中表现优异，而 Deep Q-Learning 算法则在复杂迷宫中展示了强大的学习能力和适应性。

实验中遇到的挑战主要集中在超参数的调优和奖励机制的设计上，这需要不断地实验和调整。在今后的研究中，可以进一步优化网络结构，尝试更复杂的迷宫和更多的算法改进，提升机器人的智能水平。