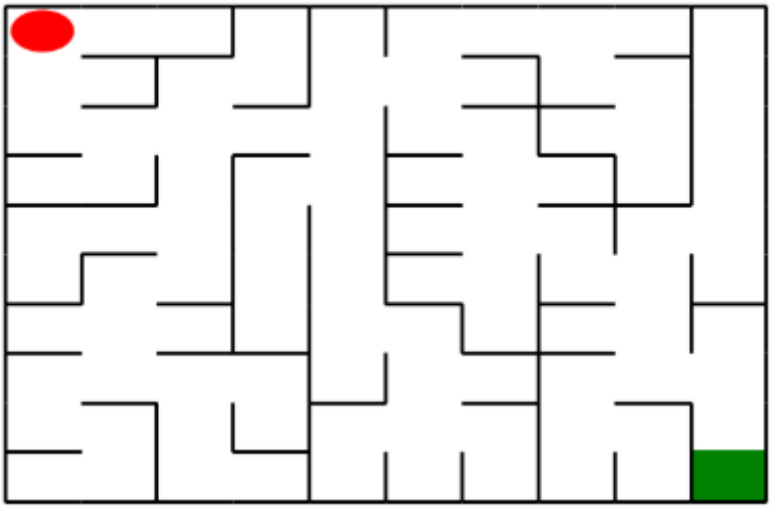
# 程序报告

**学号：2313211　　姓名：王众**

# 一、问题重述

## 1.1 实验背景

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，具体而言，有以下几种情况。
  - 撞墙
  - 走到出口
  - 其余情况
- 你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。

## 1.2 实验要求

- 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。

## 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。

### 1.4 注意事项

- $Python$与$Python\ Package$的使用方式，可在右侧$API$文档中查阅。
- 当右上角的『$Python\ 3$』长时间指示为运行中的时候，造成代码无法执行时，可以重新启动$Kernel$解决（左上角『$Kernel$』-『$Restart\ Kernel$』）。

### 1.5 参考资料

- 强化学习入门$MDP$：https://zhuanlan.zhihu.com/p/25498081
- $QLearning$简单例子（英文）：http://mnemstudio.org/path-finding-q-learning-tutorial.htm
- $QLearning$简单解释（知乎）：https://www.zhihu.com/question/26408259
- $DeepQLearning$论文：https://files.momodel.cn/Playing%20Atari%20with%20Deep%20Reinforcement%20Learning.pdf

# 二、算法设计

## 2.1 基础搜索算法

我们采用的方法是$DFS$算法来找到我们的出口，我们利用的是堆栈的方式来进行一层一层的迭代。最终搜索出我们的路径，具体的代码分析如下所示：

首先，我们定义一个自己的基础搜索的函数`my_search`，首先在最前面写入一个前面的路径地图函数，表示我们的机器人的行走路径。

然后，我们写入了前面已经帮我们完成的一个类`SearchTree`，其中包含了添加孩子，判断是否是叶子结点等功能。后面的两个函数`expand`、`back_propagation`都是前面已经帮我们实现好了的。

该部分的重点算法部分就是后面我们自己设计的一个$DFS$函数：

```
def DFS(maze):
        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 = 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
        temp = 0
        while True:
            current_node = queue[temp]  # 栈顶元素作为当前节点
            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[current_node.loc] == 0:  # 如果该点存在叶子节点且未拓展
                is_visit[current_node.loc] = 1  # 标记该点已拓展
                child_number = expand(maze, is_visit, current_node)
                temp+=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(temp)  # 如果无路可走则出栈
                temp-=1

        return path
```

我们来分析一下这段代码。首先，我们定义一个根节点$root$，然后我们的`is_visit`函变量用于存储我们的$bool$值，用于标记我们的迷宫的各个位置是否被访问过。然后我们开始对于栈的遍历，做一个循环遍历运算，首先将我们的栈顶元素作为当前节点，如果我们到达了我们的目标点，那么就给我们的路径$path$赋值，跳出我们的$while$循环。如果我们的节点是叶子节点的话，而且还没有扩展，那么我们首先对其进行标注已扩展，然后将其进行入栈操作，我们只需要做这样的操作，直到我们的算法跳出循环就可以了。

# 2.2Deep-Qlearning

我们实现的方法如下：

## 2.2.1继承 TorchRobot

`TorchRobot` 是一个基于 PyTorch 的深度强化学习基类，它封装了神经网络的初始化、训练循环、参数管理等底层功能，包含了DQN算法的核心组件：

`self.eval_model`：行为网络（用于选择动作）

`self.target_model`：目标网络（用于计算目标Q值）

`self.optimizer`：优化器（如Adam）

`self.device`：计算设备（CPU/GPU）

```
class Robot(TorchRobot):
    def __init__(self, maze):
        super(Robot, self).__init__(maze)
```

## 2.2.2使用神经网络 - Q函数近似

函数近似：神经网络学习映射 `Q(s,a) = Network(s)[a]`,输入：状态向量（如位置坐标、环境特征）

,输出：所有动作的Q值向量

工作流程：

1. 状态输入 → 神经网络前向传播
2. 输出所有动作的Q值
3. 选择Q值最大的动作（贪婪策略）

```
self.eval_model.eval()
with torch.no_grad():
    q_value = self.eval_model(state).cpu().data.numpy()
```

## 2.2.3PyTorch 张量操作

**并行计算**：GPU加速

**自动微分**：反向传播

**批处理**：同时处理多个样本

**内存管理**：高效的内存分配

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

## 2.2.4经验回放机制

**工作机制：**

1. **存储经验**：每个时间步存储 `(s, a, r, s', done)`
2. **随机采样**：从缓冲区随机抽取批量数据
3. **打破相关性**：避免连续样本的时序相关性
4. **提高效率**：重复利用历史经验

```
self.memory.build_full_view(maze=maze)
loss = self._learn(batch=batch_size)
```

## 2.2.5损失函数优化

**学习过程：**

1. **前向传播**：计算当前Q值
2. **目标计算**：使用Bellman方程计算目标Q值
3. **损失计算**：MSE损失函数
4. **反向传播**：计算梯度
5. **参数更新**：优化器更新网络权重

```
loss_list.append(loss)

def _learn(self, batch):
    states, actions, rewards, next_states, dones = batch

    # 1. 当前Q值
    current_q = self.eval_model(states).gather(1, actions)

    # 2. 目标Q值（使用目标网络）
    next_q = self.target_model(next_states).max(1)[0].detach()
    target_q = rewards + self.gamma * next_q * (1 - dones)

    # 3. 计算损失
    loss = F.mse_loss(current_q, target_q)

    # 4. 反向传播
    self.optimizer.zero_grad()
    loss.backward()
```

```
    self.optimizer.step()

    return loss.item()
```

## 2.2.6与传统Q-Learning对比：

```
# 传统Q-Learning：直接更新
self.q_table[state][action] += alpha * (target - current)

# DQN：梯度下降优化
loss = mse_loss(predicted_q, target_q)
loss.backward()
optimizer.step()
```

### 2.2.7QNetwork.py

修改了原有的 `torch_py/QNetwork.py` 中的 QNetwork 类的网络结构，将原有的 state_size -> 512 -> 512 -> action_size 改成了 state_size -> 128 -> 64 -> action_size。

# 三、最终代码

### 3.1main.py

```
# 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


def my_search(maze):

    move_map = {
        'u': (-1, 0), # 表示往上走
        'r': (0, +1), # 表示往右走
        'd': (+1, 0), # 表示往下走
        'l': (0, -1), # 表示往左走
    }

    class SearchTree(object):


        def __init__(self, loc=(), action='', parent=None):
            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):
            self.children.append(child)

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

    def expand(maze, is_visit, node):
        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[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):
        path = []
        while node.parent is not None:
            path.insert(0, node.to_this_action)
            node = node.parent
        return path

    def DFS(maze):
        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 = 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
        temp = 0
        while True:
            current_node = queue[temp]  # 栈顶元素作为当前节点
            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[current_node.loc] == 0:  # 如果
该点存在叶子节点且未拓展
                is_visit[current_node.loc] = 1  # 标记该点已拓展
                child_number = expand(maze, is_visit, current_node)
                temp+=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(temp)  # 如果无路可走则出栈
                temp-=1

        return path

    path = DFS(maze)
    return path
```

```
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):
        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):

        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为机器人选择的动作
        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)

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

        return action, reward

    def test_update(self):
        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

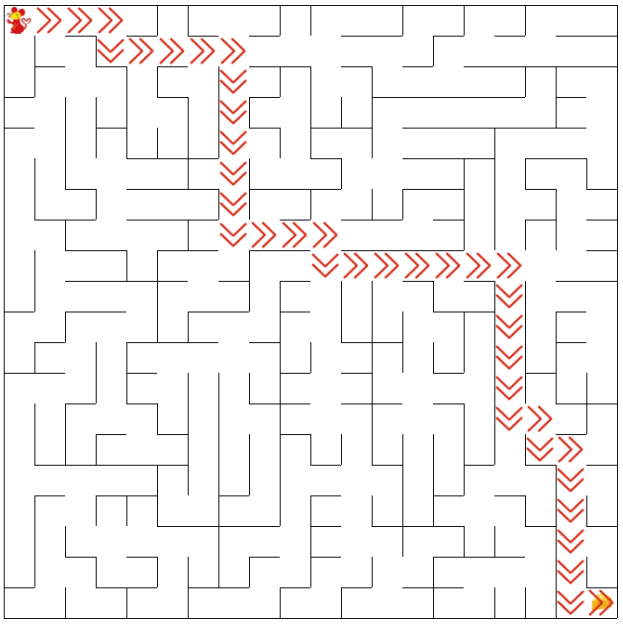
我对我的代码进行测试，系统帮我测试了四种样例，我都很好地完成了我们的强化学习的目标。下面展示一下我的实验结果的截图：

## 4.1总结果

## 4.2.1DFS

### 测试详情 隐藏迷宫

基础搜索算法 (Victory)



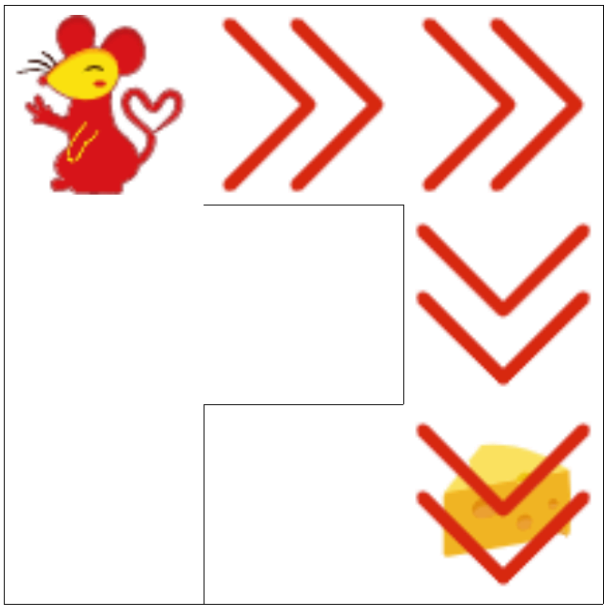
1 / 4

确定

## 4.2.2深度学习样例一

测试详情 隐藏迷宫

强化学习level3 (Victory)



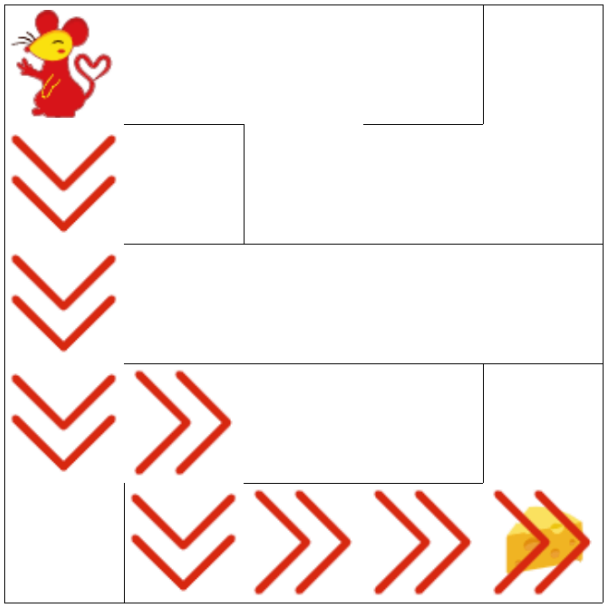
2 / 4

确定

## 4.2.3深度学习样例二

测试详情 隐藏迷宫

强化学习level5 (Victory)



3 / 4

确定

### 4.2.4深度学习样例三


从上面的四个样例可以看出，我们成功地完成了我们的测试，正确率高，该实验就 圆满完成了，也就是说《人工智能导论》的实验就全部结束了！完结撒花~

# 五、总结

本次实验的各项指标都达到了我们的预期水平，从基础搜索算法到我们的$QLearning$，都很好地实现了我们的目标，都通过了系统的测试。

当然，我们也可以对其进行一定的改进，比如说，对于基础搜索算法部分，我们可以使用其他的一些搜索算法，比如说$A^*$算法、$BFS$算法等等来进一步优化我们的性能。对于第二部分的$DQN$，我们可以使用我们的双向$DQN$算法等来进行进一步的优化。当然，我们还可以通过调整我们的参数来实现性能的优化，但是可能会有一些不确定因素，使得我们的训练结果不太相同。

遇到的最大的问题就是，我们的$DFS$算法在找到路径后就会停止我们的算法，这可能会导致我们寻找不出我们的最优路径，在面对复杂的迷宫的时候可能会绕远路甚至失败。当然，对于大规模的迷宫的解决，可能会有很大的计算量。