基于 AI 的游戏应用综合实践

By JingThree

一、 题目背景介绍

在现代游戏开发中，人工智能技术是提升游戏体验的关键组成部分。本项目基于 pygame 库实现了一款小型迷宫游戏，主要使用了 A*寻路算法、Q-learning、Deep Q network 三种算法来实现 player 在砖块环境的迷宫中自动寻路。

二、 算法原理介绍

1. A*寻路算法

   A*寻路算法是一种适用于砖块环境的寻路算法，首先介绍 A*寻路算法中的几个关键变量：

   1) OpenList：存储下一步寻路的坐标节点，每次从开发列表获取 F 值最小的节点坐标
   2) CloseList：存储已经走过的坐标节点，用来判断某个坐标节点是否已经走过。
   3) Parent 节点：父亲节点，记录是由哪个节点搜索过来。
   4) G：表示从起点移动到某一方格的移动耗费，在本项目中规定只能上下左右移动，移动一次耗费+1
   5) H：估算当前坐标到终点的距离。估算方法（启发函数）经常使用曼哈顿距离和欧几里得距离两种方法。本项目中使用的是欧几里得距离，即两点之间直线连接的距离。公式为：h(n) =sqrt((n.x-goal.x)^2 + (n.y-goal.y)^2)
   6) F：等于 G+H。数值越小，表明离终点越近。OpenList 里 F 值最小的节点，有可能是到目标节点的最短路径节点。

   A*寻路算法的主要流程为：

   1) 初始化：将起始节点加入 OpenList 中
   2) 循环：只要 OpenList 不为空，执行：
      从 OpenList 中选取 F 值最小的节点坐标作为当前节点，加入 CloseList 中
   3) 遍历当前节点的邻居节点：① 排除在 CloseList 的节点；② 如果邻居节点不在 OpenList 中，计算节点的 G\F\H 值，并加入 OpenList；③ 如果邻居节点在 OpenList 中，更新节点的 G\F\H 值和父节点
   4) 路径回溯：当终点加入 CloseList 中，或者 OpenList 为空时算法结束。如果终点在 CloseList 通过回溯终点的父节点来重构出整个路径。

2. Q-learning

首先使用马尔可夫决策来描述本项目中的相关信息：

定义 player 为智能体，处于迷宫环境中。定义智能体的状态为玩家在迷宫中的 xy 坐标，智能体的行为包括：向上走、向下走、向左走、向右走四种。奖励包括：

Action	Reward
移动到新区域	1
撞墙	-50

重复走过的路	-0.1
长时间静止在原地	-10
根据到终点的距离的动态奖励	MAX(0,30-distance)

Q 学习（Q-Learning）算法是一种基于值迭代的强化学习算法，用于学习一个具有值函数的最优策略，即在交互式任务中学习一组决策规则，以最大化累积奖励的预期值。
首先定义一个 Q(s,a)，表示在状态 s 下，采取行为 a 的价值。对于每个状态 s 和行为 a 使用贝尔曼方程来更新 Q 值：

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

这里 α 表示学习率，γ 表示折扣因子，r 是执行行为 a 后获得的奖励值，s'表示转移的下一个状态，所以 maxQ(s',a')就是下一个状态的最优行为价值。根据当前状态和行为以及获得的奖励，转移到下一个状态，根据公式计算新的 Q 值，这些 Q 值被存储在 Qtable 中。
而每次选择执行行为的策略是 ϵ− 贪心策略，即执行随机行为的概率是 ϵ，执行 Q 值最大的行为的概率是 1-ϵ。

Q-Learning 算法的主要流程为：

1) 初始化 Q 值为 0
2) 获取玩家当前坐标，即智能体当前状态，根据状态执行 ϵ− 贪心策略来选择执行的行为。
3) 执行选择的行为，计算奖励值、Q 值，然后更新 Qtable
4) 更新到下一个状态，重复步骤 2-3 直至结束

3. Deep Q network

Deep Q Network（DQN）是一种将深度学习与 Q-Learning 结合的强化学习算法。与 Q-Learning 算法不同的是，DQN 使用神经网络来预测 Q 值，通过维护一个经验回放池来存储智能体的经历（即状态、动作、奖励、新状态的组合），在学习过程中，从这个池中随机抽取样本来训练网络。
经验回放：
DQN 通过维护一个经验回放池来存储智能体的经历（状态、动作、奖励、新状态）。这种机制可以提高数据的利用率并减少样本之间的相关性，提高学习的稳定性。
网络结构：
DQN 算法采用了 2 个神经网络，分别是主网络和目标网络，主网络用来计算策略选择的 Q 值和 Q 值迭代更新。目标网络用来计算下一状态的 Q 值。也就是说，当前 Q 值是神经网络基于当前状态和行为给出的预测值，表示网络目前的理解程度，目标 Q 值是在当前状态下采取特定行为的理想（或目标）预期回报。DQN 网络的训练目标是调整神经网络的权重，让主网络计算的当前 Q 值接近目标网络计算的目标 Q 值，这也是强化学习和深度学习结合的关键部分。
网络结构主要使用了三层结构，输入层、隐藏层、输出层。输出层和输出层为全连接层，主要进行线性变换，即 y = Wx + b，其中 x 是输入向量，W 是权重矩阵，b 是偏置向量，y 是输出向量。激励函数则是将数据转为非线性关系。优化网络时使用均方误差损失函数和 Adam 自适应优化算法。

DQN 算法的主要流程为：

1) 初始化两个神经网络：主网络和目标网络，它们具有相同的架构但参数独立。

2) 初始化经验回放池，用于存储智能体的经验（包括状态、动作、奖励和下一个状态）。
3) 获取玩家当前坐标，即智能体当前状态 s，根据状态执行 ϵ− 贪心策略来选择执行的行为 a
4) 计算奖励 r 和新的状态 s'。将经验 (s,a,r,s') 存储到经验回放池中。
5) 从经验回放池中随机抽取一批样本。使用目标网络计算下一状态的最大预期 Q 值。公式为 r+ γ maxQ(s',a')。使用抽取的样本和计算的目标 Q 值来更新主网络。
6) 每隔一定的时间，更新目标网络的参数，使其与主网络的参数相同。
7) 重复步骤 3-6，直到达到终止条件

三、 算法实现

1. A*寻路算法

使用 Node 类来表示迷宫的每个格子，属性包含父节点 parent、坐标 position、g、f、h 三个关键值。

```
class Node:
    def __init__(self, parent=None, position=None):
        self.parent = parent   #父节点
        self.position = position
        self.g = 0
        self.h = 0
        self.f = 0
```

astar 函数实现了 A*寻路算法的核心部分。首先定义起始节点和终点，创建开放列表和关闭列表。开放列表使用优先队列来实现，方便快速获取 F 最小的节点。关闭列表使用集合。然后将起始节点加入开放列表中。

```
def astar(maze, start, end):
    # 创建起始和结束节点
    start_node = Node( parent: None, tuple(start))
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node( parent: None, tuple(end))
    end_node.g = end_node.h = end_node.f = 0
    open_list = []
    closed_list = set()

    # 将起始节点添加到open列表
    heapq.heappush(open_list, start_node)
```

之后进行循环，从开放列表中取出 f 值最小的列表作为当前节点。在 node 节点中定义比较基准为 f 值的大小。先判断当前节点是否为终点，如果是则开始路径回溯。

```
while open_list:
    current_node = heapq.heappop(open_list)
    closed_list.add(current_node)

    # 如果找到目标，构建路径
    if current_node == end_node:
        print('Find end !')
        path = []
        current = current_node
        while current is not None:
            path.append(current.position)
            current = current.parent
        return path[::-1]  #
```

然后定义当前节点的邻居节点队列，去除超出迷宫范围的节点和墙壁节点。

```
children = []
for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]:  # 相邻方格
    node_position = (current_node.position[0] + new_position[0], current
    # 确保在迷宫范围内
    if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or noc
        continue
    # 确保可行走
    if maze[node_position[0]][node_position[1]] ==1:
        continue
    new_node = Node(current_node, node_position)
    children.append(new_node)
```

遍历邻居节点队列，进行三种情况的判断

```
 for child in children:
     if child in closed_list:
         continue
     child.g = current_node.g + 1
     #欧几里得距离
     child.h = math.sqrt(
         (child.position[0] - end_node.position[0]) ** 2 + (chil
     child.f = child.g + child.h
     # 检查邻居节点是否已经在 OpenList 中
     found_in_open_list = False
     for open_node in open_list:
         if child == open_node:
             found_in_open_list = True
             # 检查是否有更好的路径
             if child.g < open_node.g:
                 # 更新 OpenList 中节点的值
                 open_node.g = child.g
                 open_node.h = child.h
                 open_node.f = child.f
                 open_node.parent = current_node
             break

     if not found_in_open_list:
         heapq.heappush(open_list, child)

turn None  # 如果没有找到路径
```

2. Q-learning

首先定义 ReinforcementLearningEnvironment 类。定义了智能体当前状态、计算奖励、以及执行动作的方法。

```python
class ReinforcementLearningEnvironment:
    def __init__(self, maze, player):
        self.maze = maze
        self.player = player
        self.visited_cells = set()  # 用于记录访问过的格子
        self.staypunish=0

    3 usages
    def get_state(self):
        # 返回当前状态，玩家的位置
        return self.player.x, self.player.y
```

```python
def get_reward(self, new_state):
    x, y = new_state  # 分解 new_state 为 x 和 y
    goal_x, goal_y = self.maze.get_end_pos()
    #print('x,y,gx,gy',x//self.maze.cell_size,y//self.maze.cell_size,goal_x,goal_y)
    distance = abs(goal_x - x//self.maze.cell_size) + abs(goal_y - y//self.maze.cell_size)
    #print('distance',goal_x - x//self.maze.cell_size,goal_y - y//self.maze.cell_size)
    # 根据距离计算奖励
    proximity_reward = max(0, 30- distance)
    print('Distance Reward',proximity_reward)
    if self.maze.is_goal(x, y):
        return 100 + proximity_reward+self.staypunish  # 到达终点
    elif self.maze.is_wall(x, y):
        return -50+self.staypunish    # 撞墙
    else:
        if (x, y) in self.visited_cells:
            return -0.1 + proximity_reward+self.staypunish
        else:
            self.visited_cells.add((x, y))  # 添加到访问记录中
            return 1 + proximity_reward + self.staypunish
            #print('Get A Walk')
```

```python
def step(self, action):
    dx, dy = action_to_dxdy(action)
    new_x = self.player.x + dx * self.maze.cell_size
    new_y = self.player.y + dy * self.maze.cell_size

    # 检查是否撞墙
    if self.player.collide_with_maze(new_x, new_y):
        # 如果撞墙，分配撞墙奖励并保持位置不变
        reward = -200+self.staypunish
        print('walalll')
        done = False
        new_state = (self.player.x, self.player.y)  # 保持在原地，未移动
    else:
        # 如果没有撞墙，正常移动玩家
        self.player.move(dx, dy)
        new_state = (self.player.x, self.player.y)  # 更新为新位置
        reward = self.get_reward(new_state)
        done = self.is_done(new_state)

    return new_state, reward, done
```

在 action_to_dxdy 函数中定义了智能体的四种行为：

```python
def action_to_dxdy(action):
    # 行为定义如下：
    # 0 - 上, 1 - 下, 2 - 左, 3 - 右
    if action == 0:  # 上
        return 0, -1
    elif action == 1:  # 下
        return 0, 1
    elif action == 2:  # 左
        return -1, 0
    elif action == 3:  # 右
        return 1, 0
```

Q 学习的核心部分定义在 QlearningController 类中，包含 Q 表更新函数、选择行为函数和 update 函数。

初始化中的关键变量包括：学习率、折扣因子、epsilon（ϵ− 贪心策略的概率阈值，动态衰减）、成功次数和失败次数以及相关阈值

```python
class QLearningController:
    def __init__(self, maze, player):
        self.completed = False
        self.maze = maze
        self.player = player
        self.rl_environment = ReinforcementLearningEnvironment(maze, player)

        self.endState = 0;  # -1 fail   1 suc
        self.stay_count = 0  # 记录在同一格子停留的次数
        self.stay_threshold = 30  # 设定停留阈值


        self.q_table = defaultdict(lambda: defaultdict(lambda: 0))

        self.alpha = 0.1  # 学习率
        self.gamma = 0.9  # 折扣因子

        self.success_count = 0  # 成功次数
        self.success_threshold = 2  # 成功阈值
        self.fail_count = 0  # 游戏失败次数
        self.fail_threshold = 5  # 失败阈值

        self.epsilon = 1.0  # 初始 epsilon 值
        self.epsilon_decay = 0.99  # epsilon 衰减率
        self.epsilon_min = 0.01  # epsilon 的最小值

        self.last_position = None
```

choose_action 函数用于实现根据 ϵ− 贪心策略选择行为。

```python
def choose_action(self, state):
    q_values = self.q_table[state]
    if random.random() < self.epsilon:
        return random.choice([0, 1, 2, 3])  # 随机选择
    else:
        # 如果 q_values 为空，则随机选择动作，否则选择最大 Q 值的动作
        if not q_values:
            return random.choice([0, 1, 2, 3])
        max_q = max(q_values.values())
        actions_with_max_q = [action for action, value in q_values.items() if value == max_q]
        return random.choice(actions_with_max_q)
```

update_q_table 函数实现计算 Q 值，核心部分是使用贝尔曼方程计算的 new_value

```python
    def update_q_table(self, state, action, reward, next_state, done):
        old_value = self.q_table[state][action]
        # 检查 next_state 是否存在于 Q 表中，如果不存在，则自动创建
        next_max = max(self.q_table[next_state].values(), default=0) if not done else 0
        new_value = old_value + self.alpha * (reward + self.gamma * next_max - old_value)
        self.q_table[state][action] = new_value
```

接下来是 Q 学习算法的主要流程函数，选择动作然后执行，更新 Q 表，并且动态修改 epsilon 值，然后转移到下一个状态。并且进行是否到达结束阈值的判断。红框部分为主要代码。

```
def update(self):
    current_state = self.rl_environment.get_state()
    if self.last_position == current_state:...
    else:
        self.stay_count = 0
        self.rl_environment.staypunish =0

    action = self.choose_action(current_state)
    new_state, reward, done = self.rl_environment.step(action)
    self.update_q_table(current_state, action, reward, new_state, done)
    self.epsilon = max(self.epsilon_min, self.epsilon_decay * self.epsilon)
    self.last_position = current_state

    if done:
        if self.rl_environment.is_success():
            self.success_count += 1
            if self.success_count >= self.success_threshold:
                self.completed = True
                print('QLearning Success!')
                self.endState = 1
                return True
        else:
            self.success_count = 0

        self.rl_environment.reset()

    return False
```

3. Deep Q network

首先实现经验回放机制。经验采用命名元组实现，包含当前状态、行为、下一个状态、奖励四种属性，创建 ReplayMemory 类实现经验的抽取和存放。

```
Experience = namedtuple( typename: 'Experience', field_names: ('state', 'action', 'next_state', 'rewa

2 usages
class ReplayMemory:
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    1 usage
    def push(self, *args):
        #存储经验
        self.memory.append(Experience(*args))


    1 usage
    def sample(self, batch_size):
        # 随机抽取一批经验用于训练
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

然后来实现 DQN 模型网络结构。设输入层大小为 2，即玩家的当前位置 x，y。输出层大小为 4，即上下左右四种行为。设隐藏层大小为 64。激活函数采用 ReLU 函数

```
# 初始化 DQN 模型的参数
input_size = 2    # 玩家位置
hidden_size = 64   # 隐藏层的大小
output_size = 4   # 上下左右
4 usages
class DQN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(DQN, self).__init__()
        self.gamma = 0.9
        # 定义第一个全连接层 接受输入玩家状态
        self.fc1 = nn.Linear(input_size, hidden_size)
        # 定义激活函数ReLU
        self.relu = nn.ReLU()    #f(x) = max(0, x)
        # 定义第二个全连接层 输出四个动作的预期 Q 值
        self.fc2 = nn.Linear(hidden_size, output_size)
```

定义网络的前向传播过程：首先将数据输入第一个全连接层，在这一次对数据进行线性变换，然后输出通过激活函数。最后将处理过的数据通过第二个全连接层

```
    def forward(self, x):
        # 前向传播过程：将输入数据通过网络层和激活函数
        x = self.relu(self.fc1(x))  # 通过第一个全连接层然后应用 ReLU 激活函数
        x = self.fc2(x)  # 最后通过第二个全连接层
        return x
```

接下来实现网络中根据 ϵ− 贪心策略选择行为，原理和 Q 学习中的选择行为一样，q 值是使用网络来计算每个可能行为的预期 Q 值，并返回最大 Q 值的行为

```
def predict_action(self, state, epsilon):
    state_tensor = torch.tensor( data: [state], dtype=torch.float).to(device)  # 转换为张量
    if random.random() > epsilon:
        with torch.no_grad():
            q_values = self(state_tensor)
            return q_values.max(1)[1].item()  # 返回具有最大 Q 值的动作
    else:
        return random.randrange(output_size)  # 随机选择一个动作
```

模型学习阶段通过学习经验来优化策略。从经验回放池中提取经验，使用主网络计算当前 Q 值，使用目标网络计算下一个状态的最大 Q 值，通过公式计算出目标 Q 值。

```python
def learn(self, optimizer, criterion, experiences, target_model):
    # 分解经验
    states, actions, rewards, next_states = zip(*experiences)

    # 转换为张量
    states = torch.tensor(states, dtype=torch.float)
    actions = torch.tensor(actions, dtype=torch.long)
    rewards = torch.tensor(rewards, dtype=torch.float)
    next_states = torch.tensor(next_states, dtype=torch.float)

    # 获取当前状态的预测 Q 值
    current_q_values = self(states).gather(1, actions.unsqueeze(-1))

    # 使用目标网络计算下一个状态的最大 Q 值
    max_next_q_values = target_model(next_states).detach().max(1)[0]
    expected_q_values = rewards + (self.gamma * max_next_q_values)
```

计算损失，即当前 Q 值和目标 Q 值之间的差异，通过反向传播更新网络权重，优化模型。

```python
# 计算损失
loss = criterion(current_q_values, expected_q_values.unsqueeze(1))

# 优化模型
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

在 DeepQNetworkController 类中实现了 DQN 的主要流程，首先初始化经验池、两个网络

```python
self.replay_memory = ReplayMemory(10000)
# 两个网络
self.model = DQN(input_size, hidden_size, output_size)
self.target_model = DQN(input_size, hidden_size, output_size)

# 使用 Adam 优化器
self.optimizer = optim.Adam(self.model.parameters(), lr=0.001)
# 使用均方误差作为损失函数
self.criterion = nn.MSELoss()
```

Update 函数中 获取当前状态，选择行为，然后计算经验存储到经验池中

```python
# 从模型中获取动作
action = self.model.predict_action(current_state, self.epsilon)
# 执行动作并获取反馈
new_state, reward, done = self.rl_environment.step(action)
# 将经验存储到经验回放池中
self.replay_memory.push( *args: current_state, action, reward, new_state)
```

然后从经验回放池中随机抽取一批样本来更新主网络。（使用目标网络计算目标 Q 值的

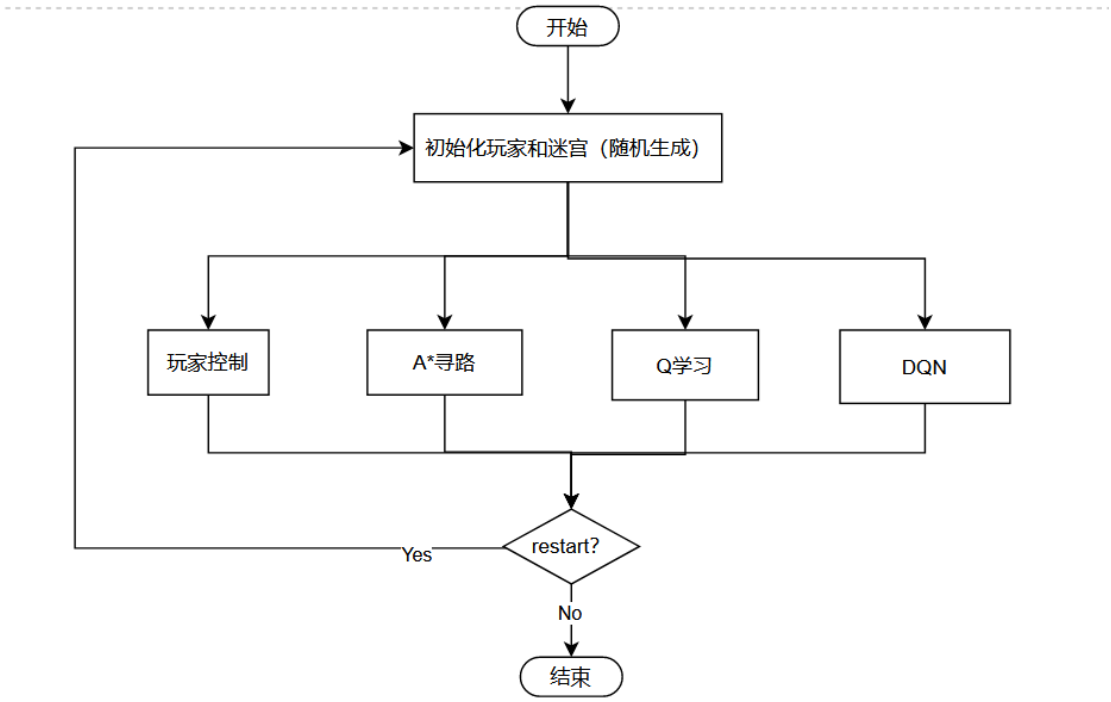
部分写在 learn 函数中）

```
BATCH_SIZE=5
#经验池的经验积累到一定程度开始训练主网络
if len(self.replay_memory) > BATCH_SIZE:
    experiences = self.replay_memory.sample(BATCH_SIZE)
    self.model.learn(self.optimizer, self.criterion, experiences, self.target_model)
```

定期更新目标网络的参数，使其与主网络的参数相同

```
#更新目标网络参数
self.update_count += 1
if self.update_count % 10 == 0:
    self.update_target_model()
    self.update_count = 0  # 重置计数器
```

四、 程序流程



本项目主要包含了三种算法，加上玩家自主控制一共四种控制模式，开始游戏之前可以选择模式。

```
pygame.init()
screen_width, screen_height = 800, 600
screen = pygame.display.set_mode((screen_width, screen_height))
pygame.display.set_caption('Maze Game')

selected_mode = show_menu(screen)
if selected_mode is None:
    pygame.quit()
    exit()


# 创建迷宫和玩家
maze = Maze(screen_width, screen_height)
player = Player(maze)
start_pos = maze.get_start_pos()
end_pos = maze.get_end_pos()

# 根据所选模式设置控制器
if selected_mode == 'Player Control':
    controller = PlayerController(player)
elif selected_mode == 'A* Algorithm':
    controller = AStarController(maze, player, start_pos, end_pos
elif selected_mode == 'Q Learning':
    controller = QLearningController(maze, player)
elif selected_mode == 'Deep Q Network':
    controller = DeepQNetworkController(maze, player)
```

根据选择的模式调用不同的控制器，开始游戏循环。在每一次游戏中迷宫是随机生成的。并设置了 restart 按钮，在玩家控制模式和 A*寻路模式时只有寻路结束才可以 restart。
为了应对 Qlearning 和 DQN 陷入局部搜索时的情况，这两种模式下 restart 按钮随时可以触发，并且设置了成功阈值和失败阈值，当满足任一阈值时游戏结束。判断寻路失败的规则是当 player 在同一个方格中停留超过十次则游戏失败，重新开始寻路。

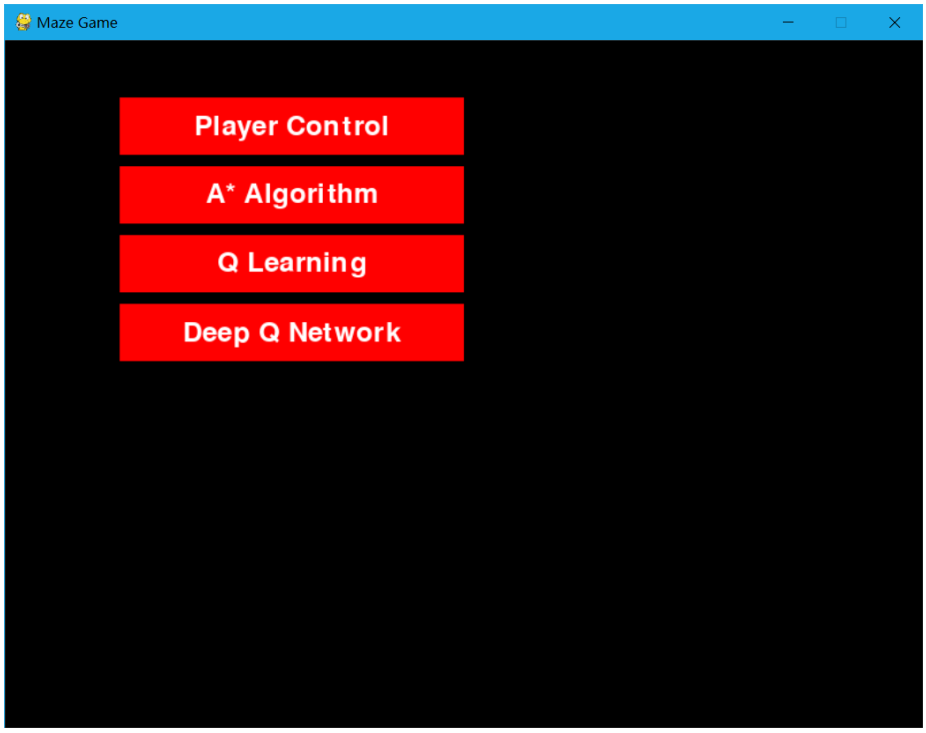
```
while running:
    events = pygame.event.get()
    for event in events:
        if event.type == pygame.QUIT:
            running = False
        if event.type == pygame.MOUSEBUTTONDOWN:
            #if selected_mode !='Reinforcement Learning':
                if restart_button.collidepoint(event.pos) :
                    if selected_mode =='Player Control' or selected_mode =='A* Algorithm
                        if controller.completed:
                            restart_game(maze, player, controller, selected_mode)
                            game_won = False  # 重置游戏通关状态
                            game_lost = False  # 重置失败状态
                    else:
                        restart_game(maze, player, controller, selected_mode)
                        game_won = False  # 重置游戏通关状态
                        game_lost = False  # 重置失败状态


    # 玩家控制时重置移动状态
    if selected_mode == 'Player Control':
        player.moving = False
        controller.update(events)
```

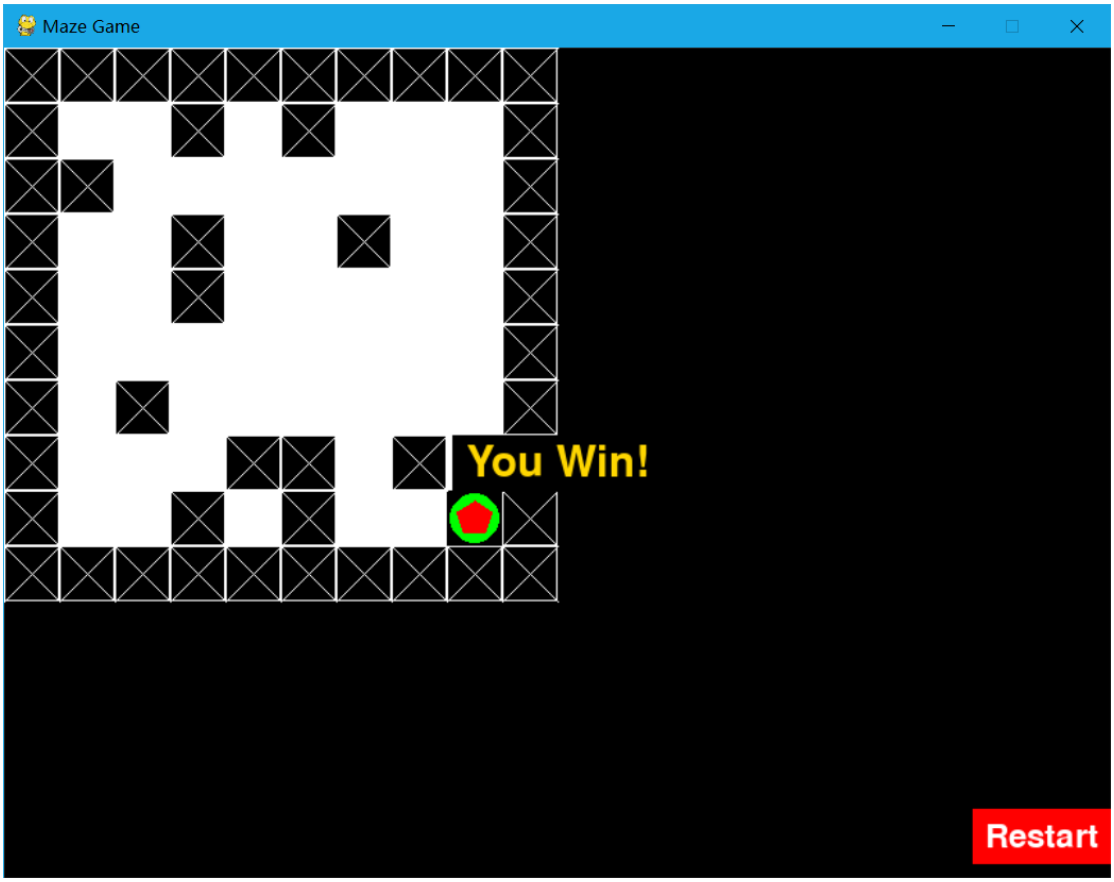
```
        pass
    elif selected_mode == 'Q Learning':
        if not controller.completed:
            if not game_lost and not game_won:
                if controller.update():  # 如果返回 True，则训练结束
                    if controller.completed:
                        if controller.endState == 1:
                            print("Training finished successfully.")
                            game_won = True
                        elif controller.endState == -1:
                            print("Training finished with too many failures.")
                            game_lost = True
    elif selected_mode == 'Deep Q Network':
        if not controller.completed:
            if not game_lost and not game_won:
                if controller.update():  # 如果返回 True，则训练结束
                    if controller.completed:
                        if controller.endState == 1:
                            print("Training finished successfully.")
                            game_won = True
                        elif controller.endState ==-1:
                            print("Training finished with too many failures.")
                            game_lost = True
```

五、 程序运行结果

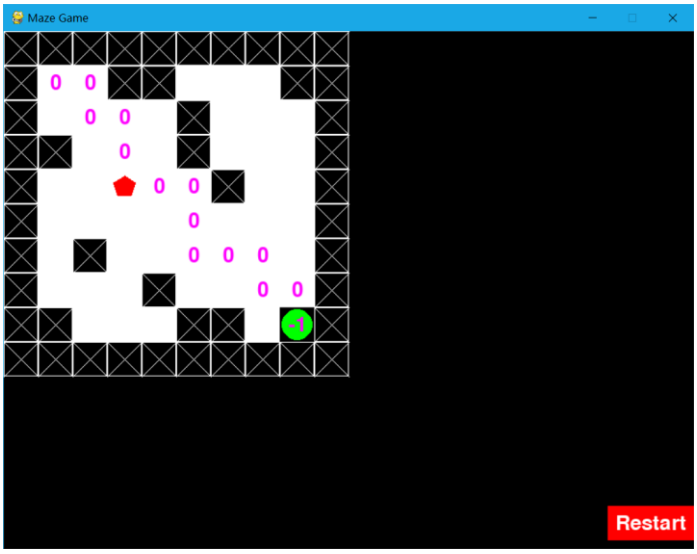
选择模式界面：

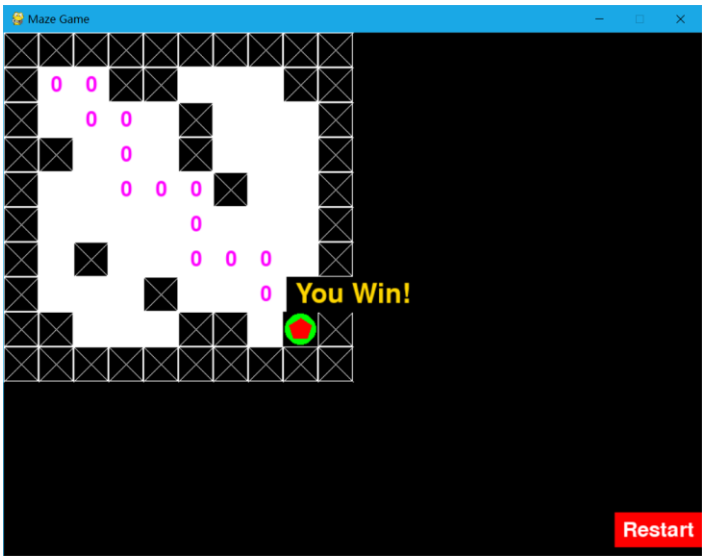


1. 玩家控制

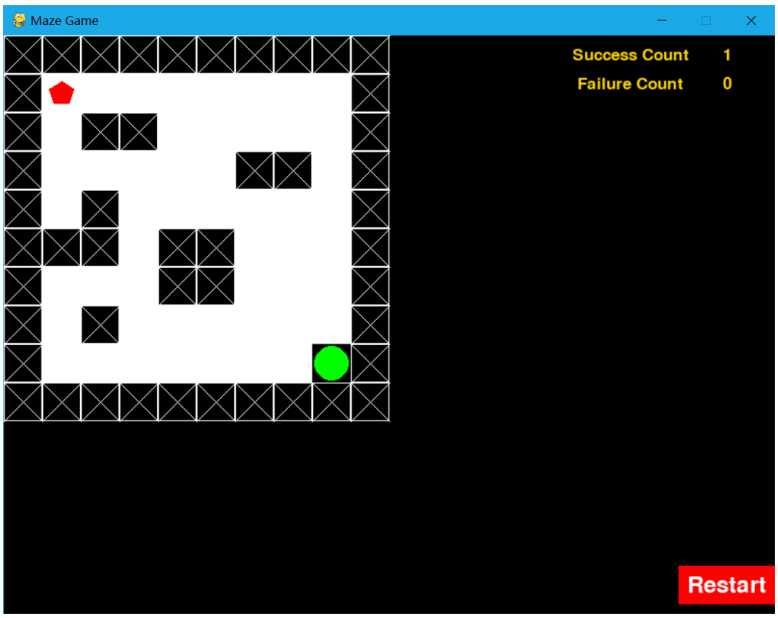


2. A*寻路

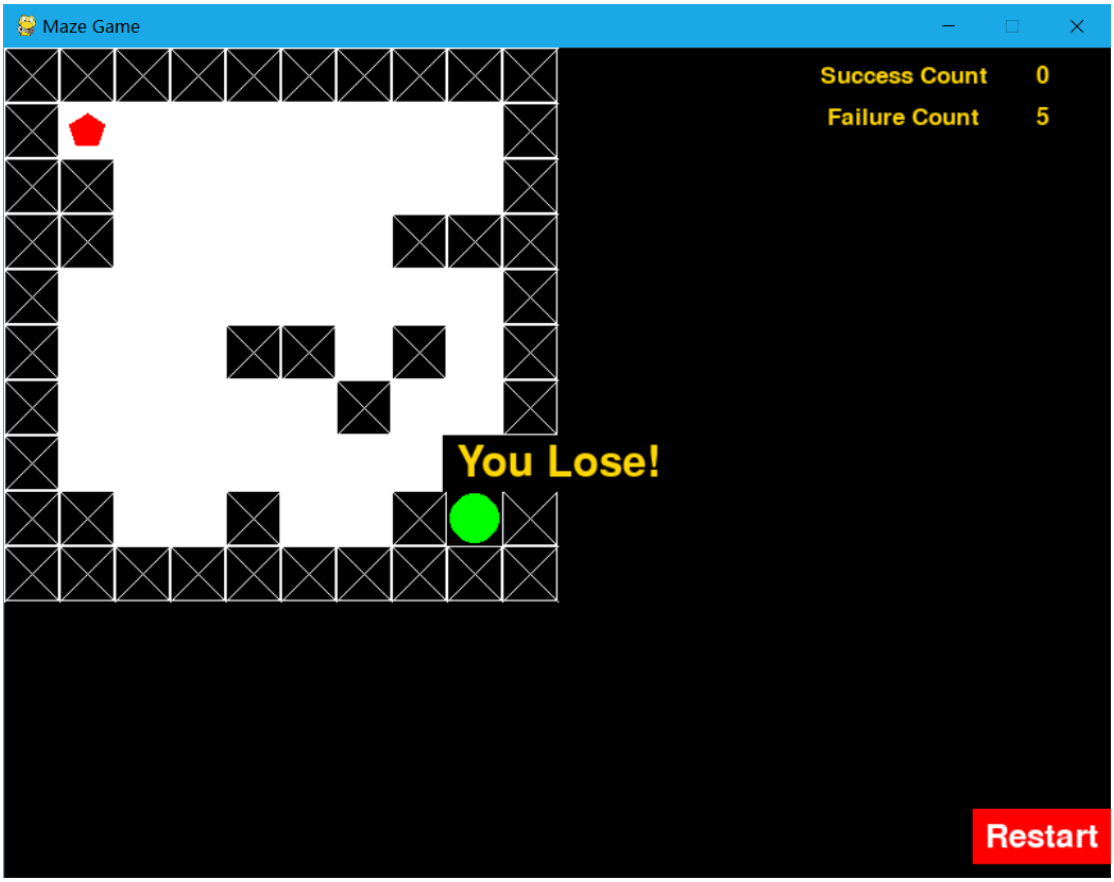




3. Q-learning



4. DQN



六、 心得体会

第一次接触深度学习部分的知识，能够完全理解其实还是有点困难的，但是在写 DQN 部分的时候逐步对深度学习有了整体的认识，并且掌握了 Qlearning 和 DQN 的区别和共同点。可能是由于强化学习环境的奖励机制设置的有些不合理，导致 Qlearning 和 DQN 模式下智能体很容易陷入局部搜索。并且在判定游戏失败时只判断了长时间停留在原地，智能体多次在两个网格中徘徊也可以加入失败判定中，这部分后续还可以再加以改进。