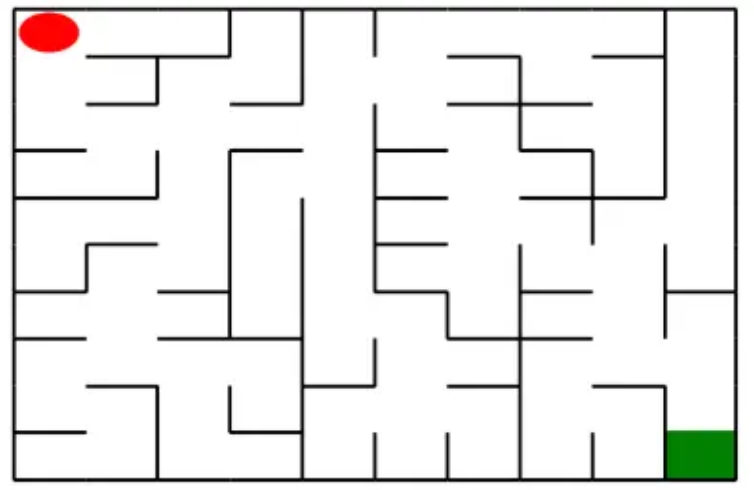
**机器人自动走迷宫问题程序报告**

学号2112213姓名：冯思程

1. **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：

i撞墙

ii走到出口

iii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。

1. **设计思想**

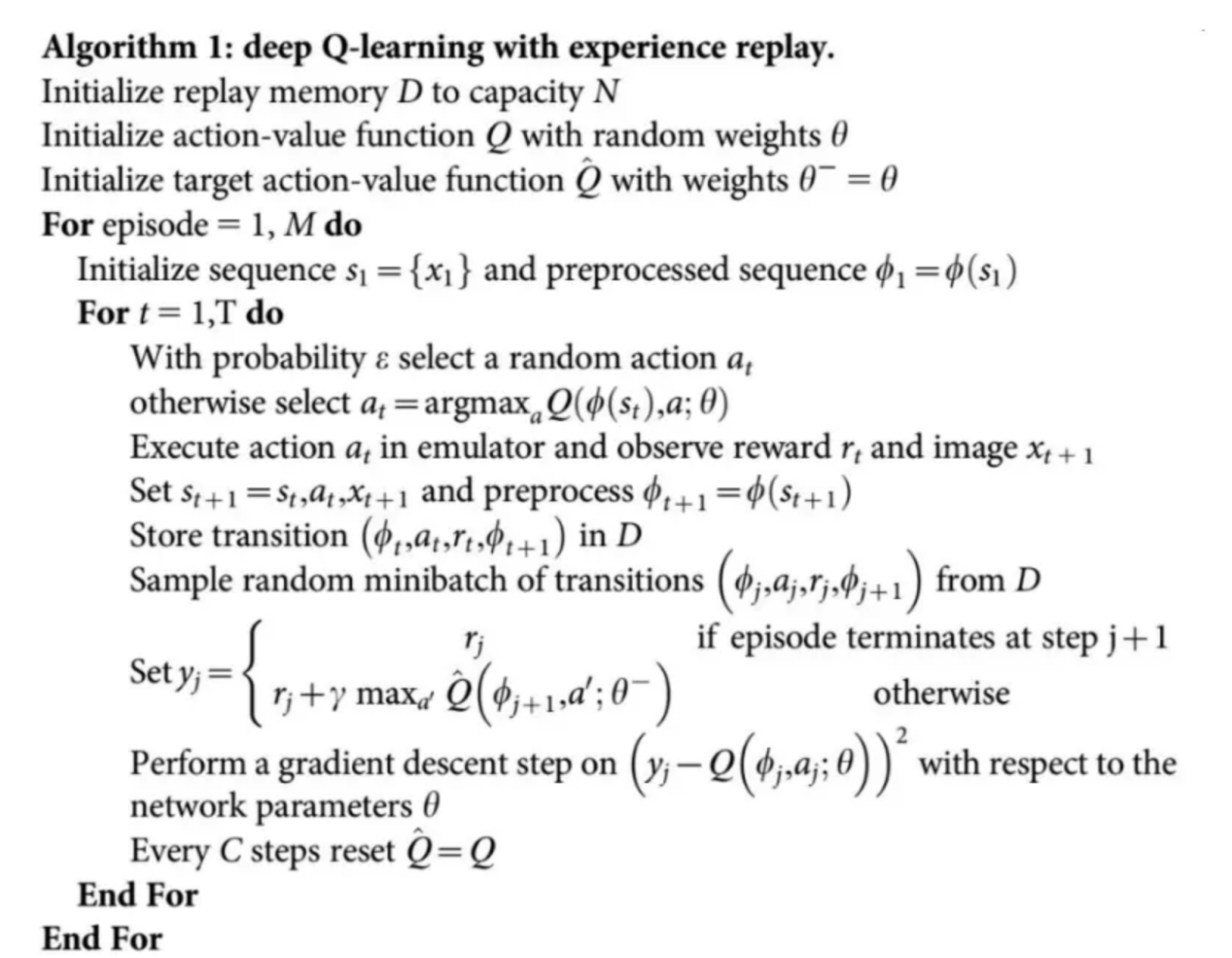
**方法：**

**题目一**：

任选深度优先搜索算法、最佳优先搜索 A\* 算法其中一种实现机器人走迷宫。这里利用的方法原理非常简单，利用深度优先搜索去寻找一条可以走到终点的路径，基本原理就是算法会一直沿着算法前进，直到撞墙或者走到以及访问过的位置，然后会根据reverse函数进行路径回溯，然后继续探索，通过这样的探索机制逐步达到最后的目标位置。

**题目二**：

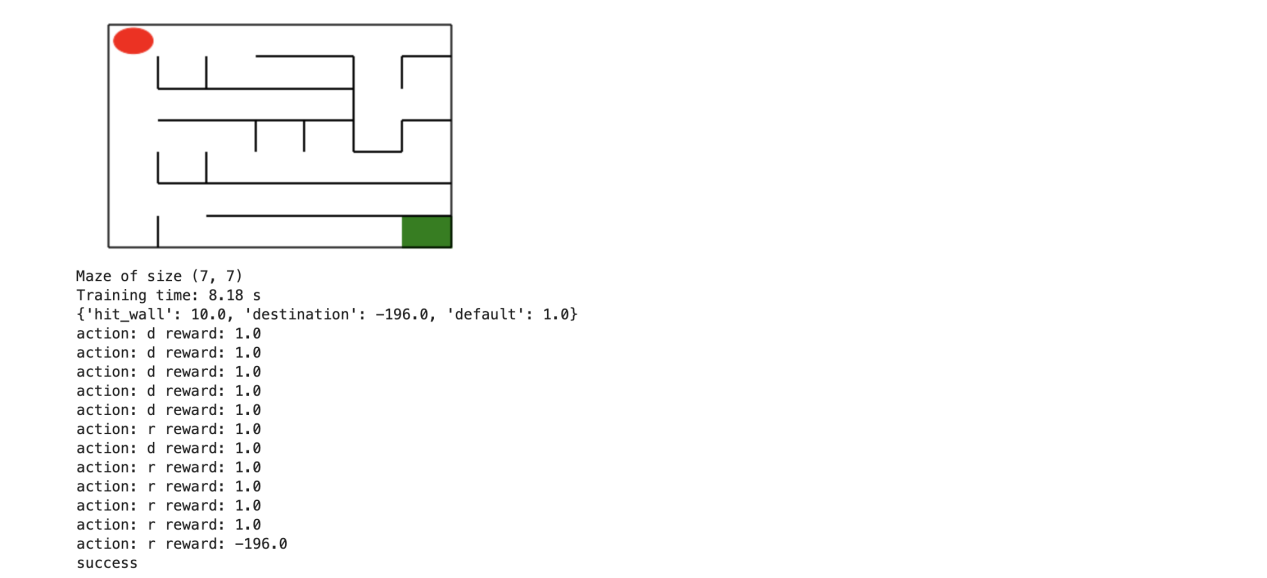
这里主要利用的算法是DQN算法，强化学习是一个反复迭代的过程，每一次迭代要解决两个问题：给定一个策略求值函数，和根据值函数来更新策略。而 DQN 算法使用神经网络来近似值函数。下图是DQN算法的伪代码：



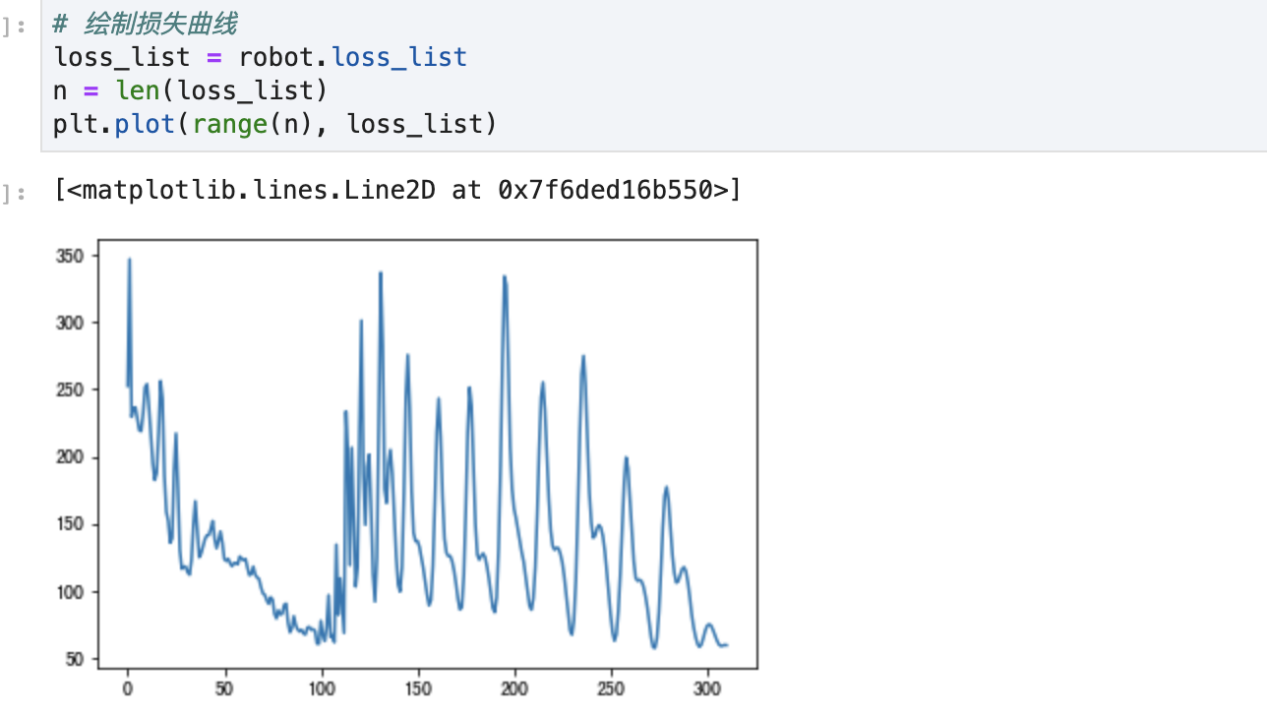
然后这里DQN算法的完成流程一共分为三步：

1. 实现简单的DQNrobot，源代码中提供了简单的两层全连接神经网络决策动作，这里的源代码我在后面进行了修改。
2. 实现我的DQNrobot，根据自己的设计原理进行智能体的代码实现。
3. 进行测试，这里我选择了在mo平台上根据自己实现的机器人智能体创建出ipynb文件，

在mo平台上进行运行测试，下面是一次的测试结果，成功实现了机器人自走迷宫的目标；



这里是对应的损失的曲线：



然后对我具体的实现方法进行介绍：

我采用了全局观察策略。与传统的逐步探索策略（只能看到当前位置附近的环境）不同，全局观察策略能够看到整个迷宫环境。这种策略对于智能体在环境中学习和决策是有帮助的，因为它能够让智能体知道迷宫的全貌，从而能够更好地计划其行动。

这段代码的逻辑基于强化学习的思想，尤其是基于Q学习的算法。Q学习是一个价值迭代算法，它试图找到一个函数Q，该函数能够预测在给定状态下采取给定行动的期望奖励。通过这种方式，智能体能够学习如何通过在迷宫中选择有利的行动来最大化它的总体奖励。

train函数是这段代码的主要驱动部分。这个函数将智能体放入迷宫中，并让其执行一系列的行动，直到到达目标位置。在每次行动后，它都会使用\_learn方法来更新Q函数。这个方法的具体实现没有在这段代码中给出，但它很可能会使用一种方法来最小化预测的Q值和实际收到的奖励之间的差异。这就是为什么我们会在每次迭代后将损失（预测和实际奖励之间的差异）添加到loss\_list的原因。

在每次训练迭代后，train函数都会调用test\_update函数来测试智能体的表现。这个函数将智能体放入同样的迷宫中，并让它根据当前的Q函数来选择行动。如果智能体成功地到达目标位置，那么训练就完成了，函数将返回损失列表。

train\_update函数和test\_update函数都使用了智能体的sense\_state方法来感知当前的状态，然后根据这个状态选择行动。这是强化学习的一个关键步骤，因为智能体需要能够感知其环境，才能做出明智的决策。

这段代码的主要特点是它使用了一个"全局观察"的策略。这意味着智能体在开始训练前就已经知道了整个迷宫的布局。这种策略可能会加速学习过程，因为智能体可以立即看到所有可能的路径，而不是只能看到当前位置附近的环境。然而，这种策略在现实世界的许多环境中可能并不可行，因为智能体通常无法在开始任务前就完全了解其环境。

**优化方向：**

**题目一：**

对于题目一来说，我认为在算法角度来说，已经没有什么优化方向，DFS是一个非常成熟的算法，这里的要求也是利用这个算法，所以在算法上我不进行优化。

从算法的时间复杂度的角度来看，我认为可以利用一个字典来记录已经访问过的节点，防止重复访问浪费资源，这个算法最坏的情况就是O(n)，n是节点个数。所以这里我也只是进行了一个简单防止重复访问的优化。

**题目二：**

这里我考虑到的优化方向有如下：

1. 这里我修改了原有的 `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。代码如下：（红色部分是修改代码）

super(QNetwork, self).\_\_init\_\_()

self.seed = torch.manual\_seed(seed)

self.input\_hidden = nn.Sequential(

nn.Linear(state\_size, 128),

nn.ReLU(False),

nn.Linear(128, 64),

nn.ReLU(False),

)

self.final\_fc = nn.Linear(64, action\_size)

1. 修改奖励策略，将终点奖励设置为和迷宫大小相关，否则在过大的迷宫中，机器人不一定能「看得到」太远的终点。

3）由于本次实验没有调参数的要求，只需要实现，所以这里没有过多的优化策略。

**局限性：**

1全局知识：在这个实现中，机器人一开始就能看到整个迷宫的布局，这在许多真实世界的应用中可能并不现实。在很多情况下，智能体可能只能看到局部的环境，或者只能通过互动逐步了解环境。如果机器人不能获取到全局视野，那么可能需要一个更复杂的方法来探索和了解环境。

2动态环境：如果迷宫环境是动态的，例如墙壁的位置可能会随着时间变化，那么这种一次性的全局观察和固定路径的策略就不再适用。在这种情况下，智能体可能需要有能力重新评估其策略，并适应环境的变化。

3实时决策：在这个实现中，机器人先完成了全部的训练过程，然后才开始执行策略。然而，在一些应用中，智能体可能需要在学习和执行策略之间进行交替，即在执行策略的同时进行学习。这种情况下可能需要一个更复杂的学习策略。

4多智能体：如果迷宫中有多个智能体，并且他们需要协同工作来达到目标，那么就需要一种能够处理多智能体学习的方法。这个实现只考虑了单智能体的情况。

5可解释性：这个实现主要关注了找到有效的路径，但并没有提供关于如何找到这条路径的详细信息。在某些应用中，可能需要能够解释智能体的决策过程。

1. **代码内容**

**main.py**的代码是：

import os

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

def my\_search(maze):

"""

任选深度优先搜索算法、最佳优先搜索（A\*)算法实现其中一种

:param maze: 迷宫对象

:return :到达目标点的路径 如：["u","u","r",...]

"""

path = []

# -----------------请实现你的算法代码--------------------------------------

visited = {}

path\_t = []

direction = ['u', 'r', 'd', 'l']

def reverse(d):

idx = direction.index(d)

return direction[(idx + 2) % 4]

def dfs():

nonlocal path

if path:

return

location = maze.sense\_robot()

if location == maze.destination:

path = [\_ for \_ in path\_t]

visited[location] = True

for d in maze.can\_move\_actions(location):

maze.move\_robot(d)

location = maze.sense\_robot()

if location not in visited:

path\_t.append(d)

dfs()

del path\_t[-1]

maze.move\_robot(reverse(d))

dfs()

# -----------------------------------------------------------------------

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": 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)

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)

success = False

self.reset()

for \_ in range(self.maze.maze\_size \*\* 2 - 1):

a, r = self.test\_update()

# print("action:", a, "reward:", r)

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---"""

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

**Fscrobot.ipynb文件代码**；（测试用）

# 导入相关包

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

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，为了在足够大的迷宫中，机器人也能「看得到」终点。

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)

start = time.time()

# 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"]:

print('Training time: {:.2f} s'.format(time.time() - start))

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

maze = Maze(maze\_size=7) #这里地方可以更改迷宫规格。进行测试。

print(maze)

robot = Robot(maze=maze)

print(robot.maze.reward) # 输出最小值选择策略的reward值

"""Test Robot"""

robot.reset()

for \_ in range(maze.maze\_size \*\* 2 - 1):

a, r = robot.test\_update()

print("action:", a, "reward:", r)

if r == maze.reward["destination"]:

print("success")

Break

# 绘制损失曲线

loss\_list = robot.loss\_list

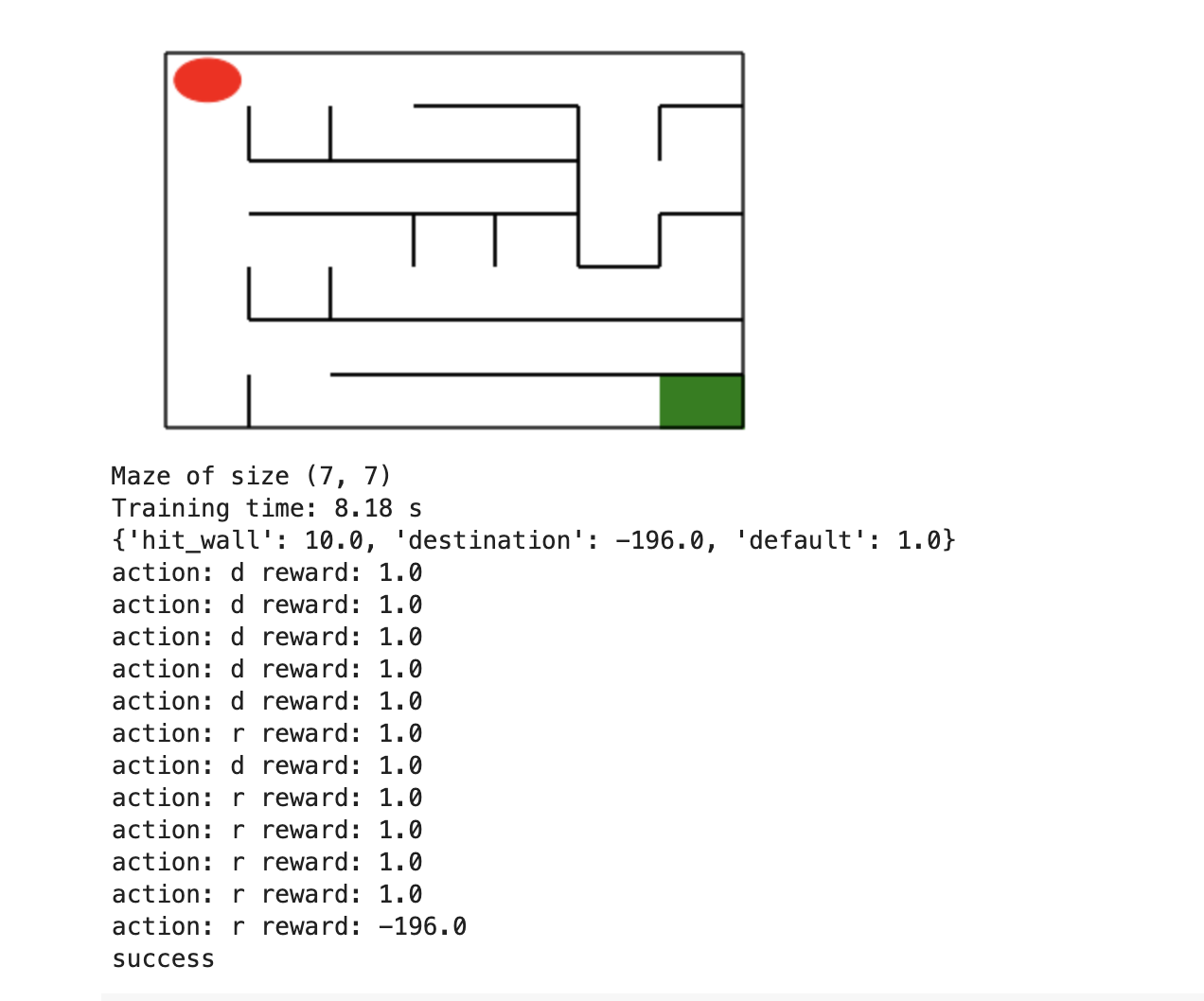
n = len(loss\_list)

plt.plot(range(n), loss\_list)

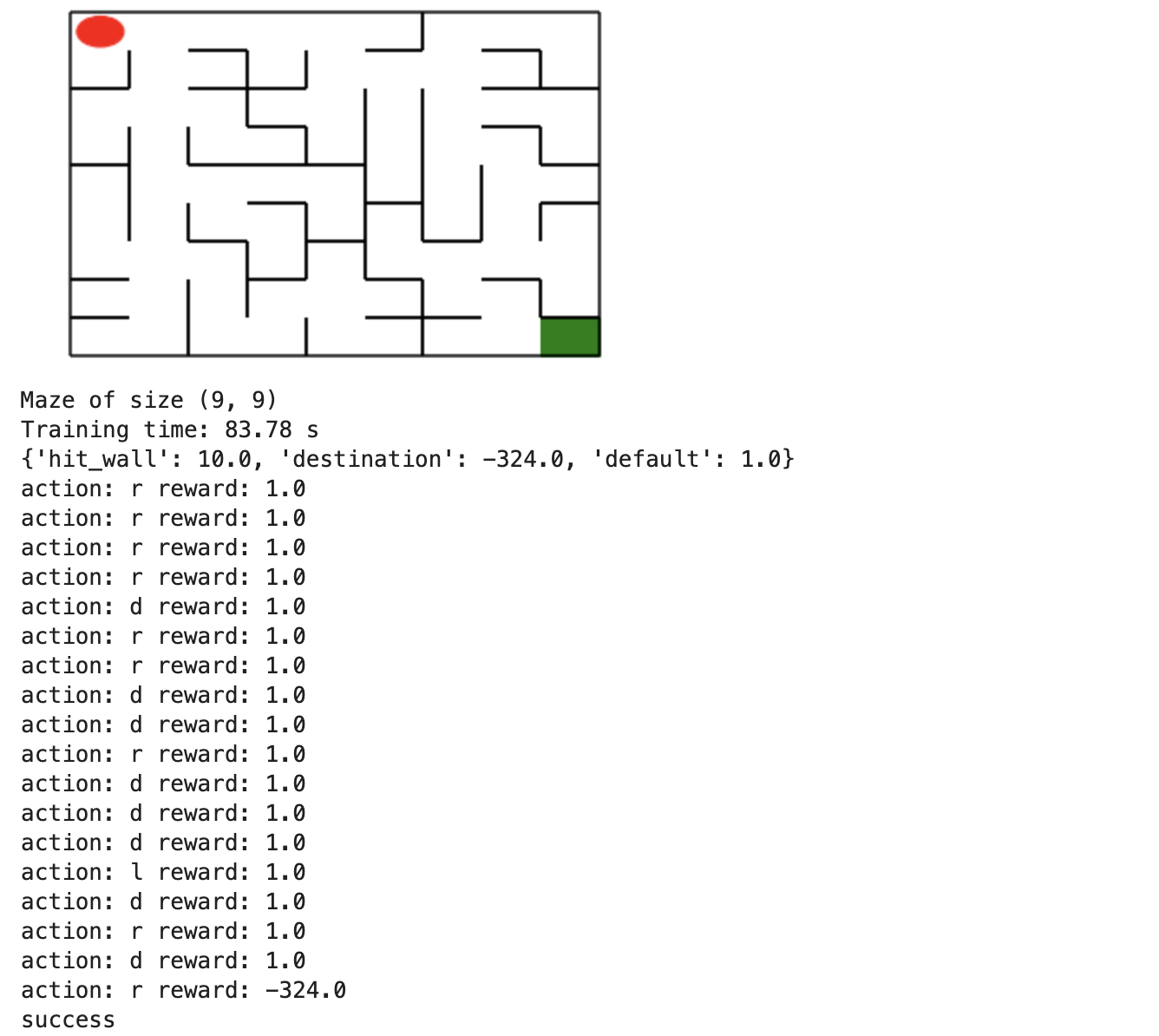
1. **实验结果**

首先展示我的robot在随机maze上测试结果：

迷宫规格=7：

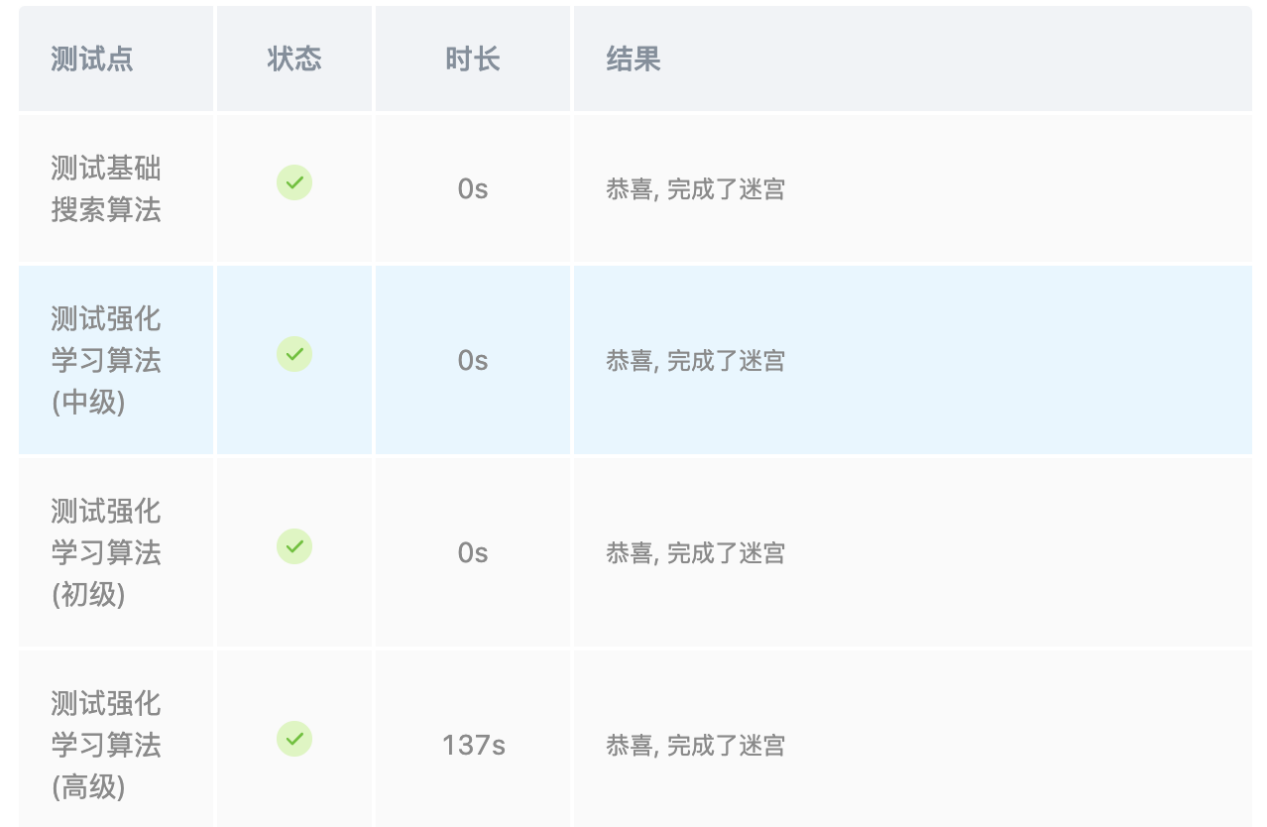


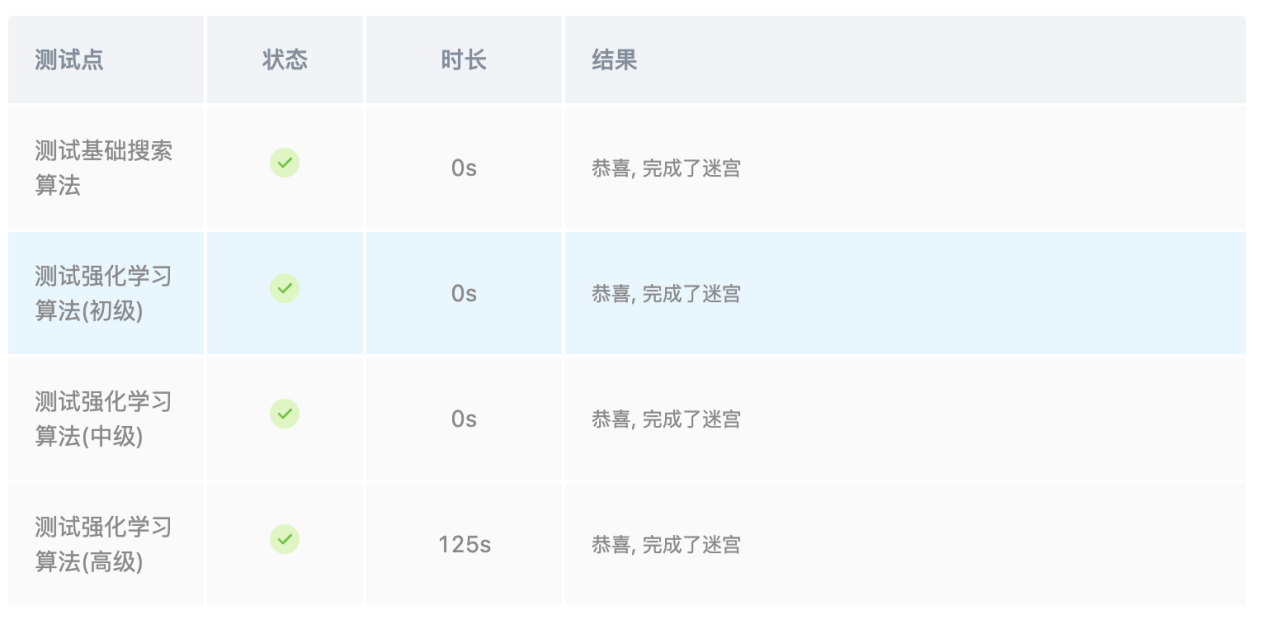
迷宫规格=9：



mo平台提交测试结果：

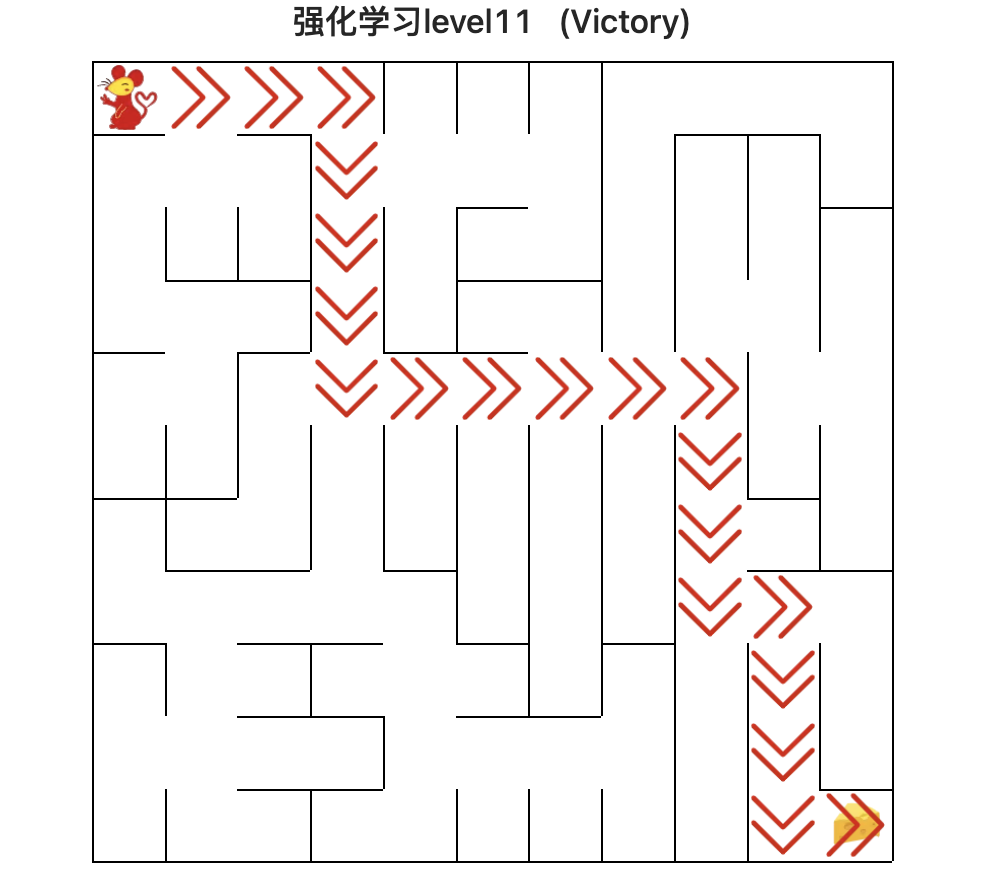
1:



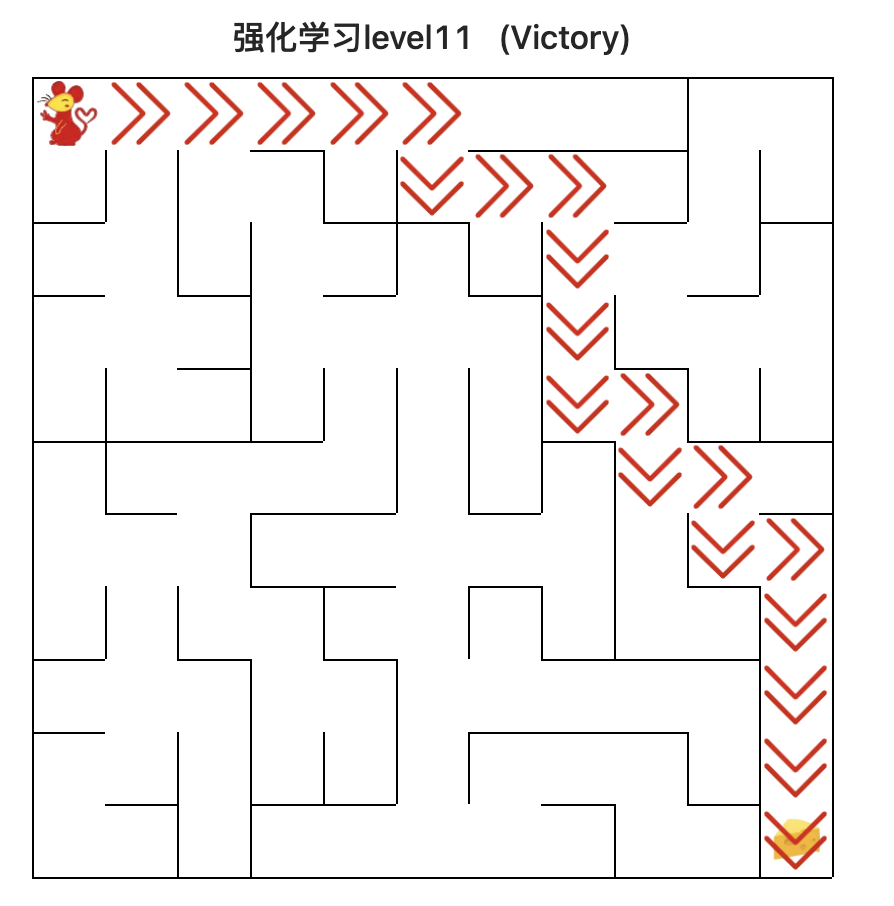
2:

对应的高级迷宫展示：

1:



2:



1. **总结**

额外考虑到的一些**更加深层次的优化方向**：

1. 双DQN（Double DQN）：在标准的DQN中，同一个网络用于选择行动并估计该行动的Q值，这可能导致过度估计Q值的问题。双DQN是一种改进，它使用两个网络：一个选择行动，另一个估计Q值，这样可以减少过度估计的问题。
2. Noisy Nets：在标准的DQN中，探索是通过ε-greedy策略实现的，这种方法可能无法充分地探索环境。Noisy Nets通过向网络中添加噪声，提高了探索能力。
3. Dueling DQN：这种架构将Q值分解为状态值和行动优势值，这样可以更有效地学习每个行动的相对优势。
4. 多步学习（Multi-step Learning）：普通的DQN是一步学习，也就是只考虑了当前状态的即时奖励和下一状态的最大Q值。而多步学习则考虑了n步的奖励，这样能更好地引导智能体学习。

**心得与感想总结**：

完成这个实验后，我认识到了搜索算法和强化学习在解决实际问题中的重要性。通过实现基础的搜索算法，我深入理解了深度优先搜索和广度优先搜索的工作原理，这些基础的搜索算法在复杂性和可解决问题的范围上都有限制，但它们为我提供了一个很好的基础，让我了解如何处理搜索和路径查找问题。

接下来，通过实现Deep Q-Learning算法，我进一步提高了解决此类问题的能力。这种算法结合了深度学习和强化学习，能够处理更复杂的环境，并且可以自我学习和改进。通过使用这种方法，我可以使机器人在更复杂的迷宫中找到出口，并且不断优化其策略，使其能够在尽可能少的步骤内到达目标。

处理强化学习的探索-利用权衡问题是一项挑战。即，我需要找到一个策略，使得智能体既可以尽可能多地探索环境，也可以利用已经学习的知识。

在完成这个实验的过程中，我收获了很多。首先，我学习并实现了基础搜索算法和Deep Q-Learning算法，这提高了我的编程技巧和理解深度。其次，我对强化学习和深度学习有了更深入的理解，尤其是如何应用它们解决实际问题。最后，面对挑战并找到解决办法的过程，提高了我的问题解决能力，让我更好地理解了算法调优和模型优化的重要性。