

实 验 报 告

成绩

教 师：_____

年 月 日

班 级：_____ 01 _____

学 号：_____ 19031211355 _____

姓 名：_____ 张志鹏 _____

实验时间：_____ 11 - 16 周 _____

实验报告

1.实验目的

通过学习此次认知计算与决策实验中方敏老师讲解的强化学习部分，我对强化学习基本概念，Q-Learning、DQN等强化学习算法等有了初步的了解，提高了我进一步学习强化学习的兴趣。

但仅仅停留在掌握知识层面上是远远不够的，想要加深理解还需要将知识付诸于实践。因此，通过此次强化学习Q-Learning迷宫算法，将所学的强化学习知识应用到具体问题中，在完成实验的过程中深化对Q-Learning算法的理解。通过此次Q-Learning迷宫算法实验，可以进一步了解到Q-Learning算法的优势和不足。

此外，通过本次强化学习Q-Learning迷宫算法实验，还可以加强自己的编程能力。学会使用Python图形界面库Tkinter。

2.实验软硬件环境

硬件环境：

- CPU: Intel Core i5-6300HQ
- GPU: NVIDIA GTX950M 2G DDR5
- 内存: 12GB DDR4

软件环境：

- 操作系统: Windows 10
- Python: Python 3.6.10
- IDE: PyCharm 2017.1.2
- Numpy: Numpy 1.18.2

3.实验内容

3.1题目要求

随机生成一个迷宫，例如图 1 所示（5X5、10X10...）。红色表示起点，绿色表示目的点，黑色表示障碍物，黄色表示宝藏，碰到障碍扣分，拾到宝藏得分。利用 Q Learning 算法或深度Q学习算法找一条起点到终点的路径，以得分最高且路径最短者胜。

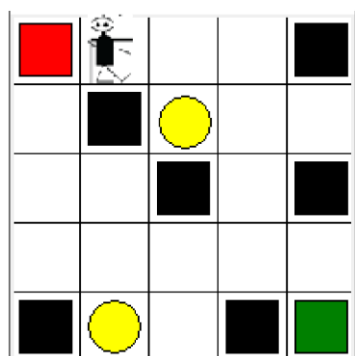


图 1 5X5迷宫示意

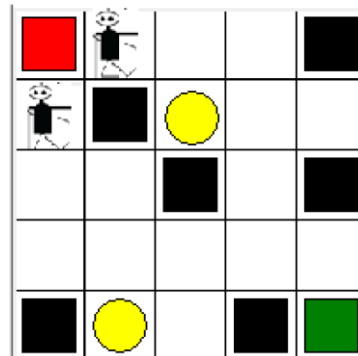


图 2 5X5两机器人迷宫示意

3.2具体实现

随机生成一个迷宫，保证迷宫有从起点到终点的通路，可在程序运行前指定迷宫规格。例如图 3、图4 所示（8X8、16X16...）。红色表示起点和机器人的位置，绿色表示目的点，黑色表示障碍物，黄色表示宝藏，碰到障碍扣分，拾到宝藏得分，走到终点得分，走到空白处扣少量分。利用 Q Learning 算法找一条起点到终点的路径，目标是使得分尽量高且路径尽量短。

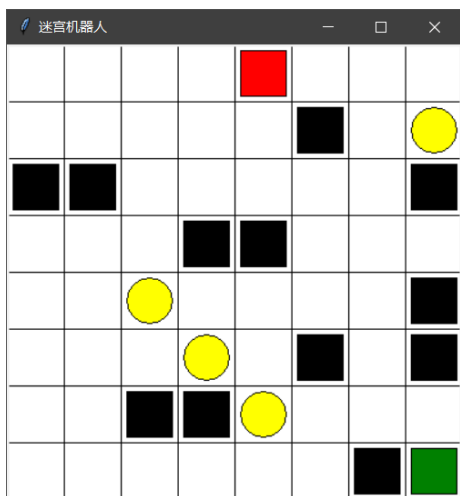


图 3 8X8迷宫示意

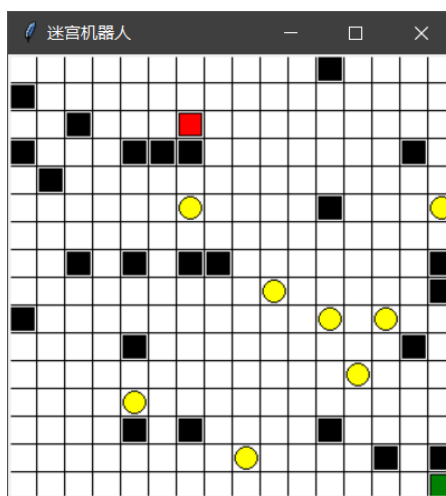


图 4 16X16迷宫示意

程序将迷宫规格的大小、每个迷宫网格的像素大小、宝藏得分、终点得分、撞墙扣分、步数扣分等参数设为全局变量，在程序运行前只需简单修改就能更改迷宫的参数。提高程序的复用性的同时，通过不同参数设置，可以方便的对比不同参数设置对结果的影响，方便实验结果分析，有利于进一步理解Q-Learning算法。

4.算法描述

Q-Learning算法过程如下：

- Q(s,a)初始化为0
- 初始化当前状态s
- 迭代执行：
 - 选择动作a
 - 采取动作a并获得奖励r
 - 获得状态s'
 - 根据公式 $Q(s, a) = Q(s, a) + \eta \cdot (r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a))$ 更新Q值
 - 当前状态更新为s'

5.实验程序及分析

本次Q-Learning迷宫机器人实验采用Python语言编写。项目名称定义为Maze_Robot。项目共包含3个Python文件：Maze.py、Robot.py、Run.py。项目结果如下图所示：

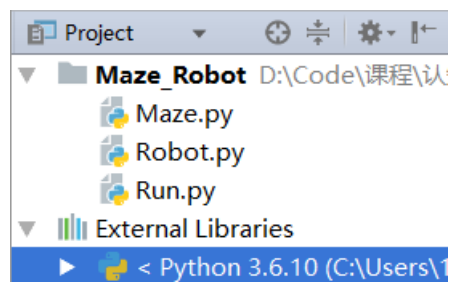


图 5 程序结构

其中Maze.py是迷宫管理程序，负责迷宫生成、图形界面的渲染等。

Maze.py定义了继承tkinter.Tk的类Maze。

Maze.py需要导入以下Python包：

```
import tkinter as tk
import numpy as np
import random
import time
```

其中，Tkinter模块(Tk接口)是Python的标准Tk GUI工具包的接口。Tk、和Tkinter可以在大多数的Unix平台下使用,同样可以应用在Windows和Macintosh系统里。Tk8.0 的后续版本可以实现本地窗口风格，并良好地运行在绝大多数平

台中。

NumPy(Numerical Python) 是 Python 语言的一个扩展程序库，支持大量的维度数组与矩阵运算，此外也针对数组运算提供大量的数学函数库。

Random库可以生成随机数。在随机生成迷宫时使用。

Time库提供的sleep()函数推迟调用线程的运行，可通过参数secs指秒数，表示进程挂起的时间。在程序中用于控制图形界面刷新频率。

Maze.py定义了以下变量：

```
PIXEL = 20  
COIN_SCORE = 50  
END_SCORE = 100  
BLANK_SCORE = -1  
WALL_SCORE = -5
```

其中，PIXEL控制迷宫每个网格的像素数量。PIXEL=20和PIXEL=50的网格大小对比如下图所示，左图为PIXEL =20，大小为10X10的迷宫，右图为PIXEL=50，大小为10X10的迷宫。



图 6 不同 PIXEL 的迷宫对比

COIN_SCORE对应机器人捡到宝藏（图中用黄色圆圈表示宝藏）时获得的分数；

END_SCORE表示机器人到达终点（图中用绿色方块表示终点）时获得的分数；

BLANK_SCORE指机器人走到空白位置时扣除的分数；

WALL_SCORE表示机器人撞到迷宫中障碍物（图中用黑色方块表示障碍物）时扣除的分数。

Maze迷宫类包含以下函数：

- `__init__(self, size)`

该函数是迷宫类Maze的初始化函数。定义了迷宫中所必须的参数。包括，迷宫尺寸size；迷宫支持的行动列表action，action中的‘u’、‘d’、‘l’、‘r’分别表示上下左右四个动作；迷宫中机器人位置，初始化为（0,0）位置等。此外，初始化时还需要调用迷宫生成、画面渲染函数。

```
def __init__(self, size):
    super(Maze, self).__init__()

    self.title("迷宫机器人")
    self.geometry('{0}x{0}'.format(PIXEL*size))

    self.size = size
    self.action_list = ['u', 'd', 'l', 'r']
    self.action_num = len(self.action_list)
    self.robot_location = np.array([0, 0]) # 机器人位置
    self.init_reward()
    self.coin_dict = {} # 硬币图id 列表
    self.coin_list = [] # 地图硬币列表
    self.wall_list = [] # 地图墙列表

    self.init_maze()
```

- `mpath(self, maze, x1, y1, x2, y2)`

mpath函数使用回溯法判断迷宫是否有从起点到终点的通路。其中传入的参数maze是用01表示的迷宫矩阵，x1,y1表示起点的坐标，x2,y2表示终点的坐标。

```

def mpath(self, maze, x1, y1, x2, y2):
    dirs = [lambda x, y: (x + 1, y),
            lambda x, y: (x - 1, y),
            lambda x, y: (x, y - 1),
            lambda x, y: (x, y + 1)]
    stack = []
    stack.append((x1, y1))
    while len(stack) > 0:
        curNode = stack[-1]
        if curNode[0] == x2 and curNode[1] == y2:
            return True
        for dir in dirs:
            nextNode = dir(curNode[0], curNode[1])
            if maze[nextNode[0]][nextNode[1]] == 0:
                stack.append(nextNode)
                maze[nextNode[0]][nextNode[1]] = -1
                break
        else:
            maze[curNode[0]][curNode[1]] = -1
            stack.pop()
    return False

```

- `init_reward(self)`

`init_reward()`函数用于随机生成迷宫。首先根据迷宫尺寸size使用random库取随机数，将随机数分配为宝藏和墙壁。根据宝藏随机数和墙壁随机数填充奖励矩阵。奖励矩阵与迷宫一一对应。将奖励矩阵处理后通过mpath函数判断迷宫是否连通，若不连通则再次随机生成迷宫，直到生成连通的迷宫。

```

def init_reward(self):
    connected = False
    while not connected:
        self.reward = np.zeros((self.size,self.size),dtype=int)
        choosen =
np.array(random.sample(range(1,self.size*self.size-1),self.size*2))
        self.coin_list = choosen[:int(self.size/2)]
        self.wall_list = choosen[int(self.size/2):]
        for x in range(self.size):
            for y in range(self.size):
                if self.coin_list.__contains__(x*self.size + y):
                    self.reward[x][y] = COIN_SCORE
                elif self.wall_list.__contains__(x*self.size + y):
                    self.reward[x][y] = WALL_SCORE
                elif x*self.size + y == self.size*self.size - 1:
                    self.reward[x][y] = END_SCORE
                else:
                    self.reward[x][y] = BLANK_SCORE
        self.rewarding=self.reward.copy()
        check = np.zeros((self.size + 2, self.size + 2))
        for x in range(self.size):
            for y in range(self.size):
                if self.reward[x][y] == BLANK_SCORE:
                    check[x + 1][y + 1] = 0
                elif self.reward[x][y] == COIN_SCORE:
                    check[x + 1][y + 1] = 0
                elif self.reward[x][y] == WALL_SCORE:
                    check[x + 1][y + 1] = 1
                elif self.reward[x][y] == END_SCORE:
                    check[x + 1][y + 1] = 0
        check[0] = 1
        check[self.size + 1] = 1
        check[:, 0] = 1
        check[:, self.size + 1] = 1
        connected = self.mpath(check, 1, 1, self.size, self.size)
    return True

```

- `init_maze(self)`

该函数作用是迷宫画面的渲染。根据init_reward()函数随机生成的迷宫矩阵画出对应的图形。


```

def init_maze(self):
    self.canvas = tk.Canvas(self, bg='white',
height=PIXEL*self.size, width=PIXEL*self.size)
    # 迷宫网格
    for x in range(0,PIXEL*self.size,PIXEL):
        self.canvas.create_line(x,0,x,PIXEL*self.size)
    for y in range(0,PIXEL*self.size,PIXEL):
        self.canvas.create_line(0,y,PIXEL*self.size,y)
    # 填充起点、终点、硬币、墙
    for x in range(self.size):
        for y in range(self.size):
            # 起点
            if x*self.size + y == 0:
                self.robot =
self.canvas.create_rectangle((y+0.1)*PIXEL,(x+0.1)*PIXEL,(y+0
.9)*PIXEL,(x+0.9)*PIXEL,fill='red')
            # 终点
            elif self.reward[x][y] == END_SCORE:
                self.end =
self.canvas.create_rectangle((y+0.1)*PIXEL,(x+0.1)*PIXEL,(y+0
.9)*PIXEL,(x+0.9)*PIXEL,fill='green')
            # 墙
            elif self.reward[x][y] == WALL_SCORE:
self.canvas.create_rectangle((y+0.1)*PIXEL,(x+0.1)*PIXEL,(y+0
.9)*PIXEL,(x+0.9)*PIXEL, fill='black')
    self.canvas.pack()

```

- restart(self)

在Q-Learning迭代过程中，迷宫中机器人的位置（红色方块）会不断移动变化，宝藏再被吃掉后，表示宝藏位置的黄色圆圈会消失，因此每次迭代中机器人位置和宝藏位置都会发生变化，在一次迭代结束完成之后，需要重新绘制迷宫的其实画面中机器人和宝藏的画面。restart()就完成了这一功能。

```

def restart(self):
    self.robot_location=[0,0]
    self.rewarding=self.reward.copy()
    # 删除起点和剩余硬币
    self.canvas.delete(self.robot)
    coin_view_list = list(self.coin_dict.values())
    for coin in coin_view_list:
        self.canvas.delete(coin)
    # 重画起点和硬币
    self.robot =
self.canvas.create_rectangle((0.1)*PIXEL,(0.1)*PIXEL,(0.9)*PI
XEL,(0.9)*PIXEL,fill='red')
    for x in range(self.size):
        for y in range(self.size):
            if self.reward[x][y] == COIN_SCORE:
                oval = self.canvas.create_oval((y + 0.1) * PIXEL, (x
+ 0.1) * PIXEL, (y + 0.9) * PIXEL,
(x + 0.9) * PIXEL, fill='yellow')
                self.coin_dict[(x, y)] = oval
    return self.robot_location

```

- judge(self,action)

Judge()函数的功能是根据当前机器人所在的位置，判断是否可以执行action动作，这里只对迷宫的边缘设置了限制，而不对迷宫中的障碍（黑色方块）设置限制。即机器人是不能突破边界冲出迷宫，但可以撞向迷宫中的障碍物。

```
def judge(self,action):  
    if (action == 'u' ) & (self.robot_location[0]== 0):  
        return False  
    elif (action=='d' ) & (self.robot_location[0]==self.size-1):  
        return False  
    elif (action == 'l' ) & (self.robot_location[1]==0):  
        return False  
    elif (action=='r' ) & (self.robot_location[1]==self.size-1):  
        return False  
    else:  
        return True
```

- step(self,action)

Step()函数是机器人行动处理函数。根据当前机器人所处的位置以及机器人向执行的动作action给以奖励，做出惩罚、画面更新等。首先根据judge函数判断机器人执行的动作是否合法。若合法，当机器人执行的动作会撞到障碍物时，机器人的位置保持不变，并受到惩罚。当机器人捡到宝藏，则给与奖励并删除迷宫中的宝藏。当机器人走到终点时，返回表示完成参数done=True。

```

def step(self, action):
    reward = 0
    done = False
    if self.judge(action):
        s_ = self.robot_location.copy()
        move_pixel=[0,0]
        if action=='u':
            s_[0] -= 1
            move_pixel[1] -= PIXEL
        elif action=='d':
            s_[0] += 1
            move_pixel[1] += PIXEL
        elif action=='l':
            s_[1] -= 1
            move_pixel[0] -= PIXEL
        elif action=='r':
            s_[1] += 1
            move_pixel[0] += PIXEL
        x = s_[0]
        y = s_[1]
        reward = self.rewarding[x][y]
        if reward == COIN_SCORE:
            self.robot_location=s_[0:]
            self.canvas.delete(self.coin_dict[(x,y)])
            self.coin_dict.pop((x,y))
            self.canvas.move(self.robot,move_pixel[0],move_pixel[1])
            done=False
        self.rewarding[self.robot_location[0]][self.robot_location[1]]
    ]=BLANK_SCORE
        elif reward == WALL_SCORE:
            done=False
        elif reward == BLANK_SCORE:
            self.robot_location=s_[0:]
            self.canvas.move(self.robot,move_pixel[0],move_pixel[1])
            done=False
        elif reward == END_SCORE:
            self.robot_location=s_[0:]
            self.canvas.move(self.robot,move_pixel[0],move_pixel[1])
            done=True
        else:
            raise Exception("Reward Error! ")

```

- render(self,progress)

Render函数功能是调控程序运行的速度。通过参数progress可以实现不同的运行效果。

```

def render(self,progress):
    time.sleep(0.1*progress)
    self.update()

```

Robot.py定义了机器人Robot类，该类主要实现了机器人维护Q表，进行学习的相关函数。

Robot包含以下三个函数：

- `__init__(self, env, Eta=1, Gamma=0.9)`

`__init__`函数是Robot类的初始化函数，接收迷宫类对象`env`作为环境知识，Eta和Gamma是Q-Learning中的学习率 η 和折扣因子 γ ，默认值分别设置为1和0.9。Q_table即Q表，初始化成大小为（迷宫宽度，迷宫长度，行动数量）的三维矩阵。

```
def __init__(self, env, Eta=1, Gamma=0.9):
    self.env = env
    self.size = env.size
    self.action_num = 4
    self.Eta = Eta
    self.Gamma = Gamma
    self.Q_table = np.zeros((self.size, self.size,
self.action_num))
```

- `epsilon_greedy_policy(self, state, epsilon):`

`epsilon_greedy_policy()`函数即机器人采用 ϵ -greedy 策略选择动作，探索迷宫。State表示当前机器人的位置。

```
def epsilon_greedy_policy(self, state, epsilon):
    if random.uniform(0,1) < epsilon:
        x=state[0]
        y=state[1]
        action_index = np.argmax(self.Q_table[x][y])
        return self.env.action_list[action_index]
    else:
        rand_index = random.randint(0,3)
        return self.env.action_list[rand_index]
```

- `learn(self, state, action, reward, state_)`

`Learn()`函数的功能是根据传入的参数：位置`state`，动作`action`，奖励`reward`，和下一位置`state_`，通过Q-Learning公式更新Q表。

```
def learn(self, state, action, reward, state_):
    x = state[0]
    y = state[1]
    x_ = state_[0]
    y_ = state_[1]
    action_index = self.env.action_list.index(action)
    Q = self.Q_table[x][y][action_index]
    action_index_ = np.argmax(self.Q_table[x_][y_])
    Q_ = self.Q_table[x_][y_][action_index_]
    self.Q_table[x][y][action_index] = Q + self.Eta * (reward +
self.Gamma * Q_ - Q)
    return True
```

最后，Run.py函数是程序入口。执行Q-Learning的迭代学习过程。

Run.py定义以下参数：

```
SIZE = 10
EPISODE = 500
```

其中，SIZE定义了迷宫的长度和宽度，单位是格子数。EPISODE定义了迭代次数。

主函数定义如下：

```
if __name__ == '__main__':
    env = Maze.Maze(SIZE)
    robot = Robot.Robot(env)
    env.after(100, update)
    env.mainloop()
```

首先通过Maze迷宫类创建环境实体env，然后根据Robot类创建机器人实体robot，然后调用update()函数让机器人迭代学习。

Update函数实现了机器人迭代学习过程。通过for循环控制迭代次数，每次迭代开始时，先调用Maze类的restart函数重置画面，并将当前迭代的奖励清零。然后通过while循环，重复让机器人选择行动action，并获得执行该action后机器人的状态以及奖励信息，然后通过机器人类的learn函数更新Q表，直至机器人走到终点，则while循环结束，进入下一个迭代。

全部迭代完成之后，使用matplotlib库画出奖励曲线。

```

def update():
    reward_list = []
    for episode in range(EPISODE):
        print('episode ',episode)
        s =env.restart()
        all_reward=0
        while True:
            x = episode / EPISODE
            # 延迟刷新界面
            # 慢速模式
            # env.render(1)
            # 匀加速模式
            # env.render(x)
            # 变加速模式
            # env.render(1-(1-x**2)**0.5)
            # 快速模式
            env.render(0)
            action = robot.epsilon_greedy_policy(s,x)
            s_, reward, done = env.step(action)
            all_reward+=reward
            robot.learn(s,action,reward,s_)
            s = s_
            if done:
                print("reward:",all_reward)
                reward_list.append(all_reward)
                break
        plt.title('robot')
        plt.xlabel('episode')
        plt.ylabel('reward')
        plt.plot(range(len(reward_list)), reward_list)
        plt.show()

```

此外，程序设置了多种执行模式。

- 慢速模式：此时机器人始终以0.1s一步的恒定速率执行；该模式适合观察机器人的运动过程，但在迷宫较大，迭代次数较多时，会运行很长时间；
- 匀加速模式：此时机器人从最快速到0.1s/步的速度以 $y=x$ 线性加速执行；
- 变加速模式：该模式参考了圆的方程，由以（0,1）为圆心，以1为半径的圆的方程 $x^2 + (y - 1)^2 = 1$ 推导出 $y = 1 - \sqrt{1 - x^2}$ 。开始时，程序执行的很快，最后几次迭代时程序执行速度大幅减慢。该模式在兼顾执行时间的同时保

证迭代末期清楚地看到运行轨迹。原理如下图所示：

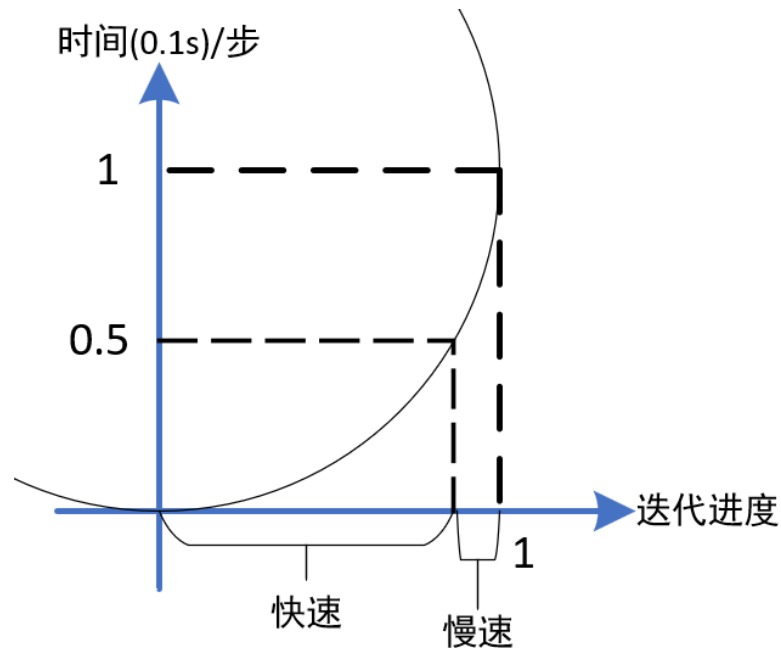


图 7 以圆的方程作为迭代速度控制参数

- 快速模式：该模式以硬件能达到的最快速度执行程序，可以最快获得结果，但不能清晰的看到机器人运行轨迹。

6. 实验结果分析

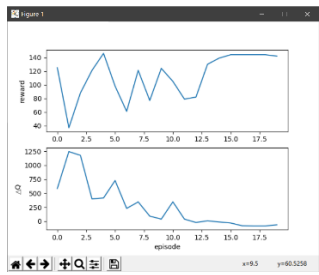
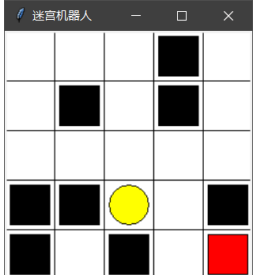
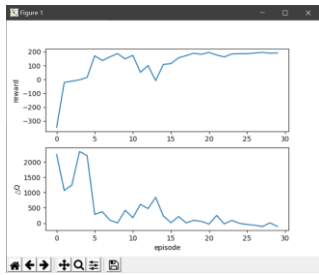
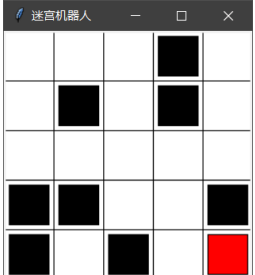
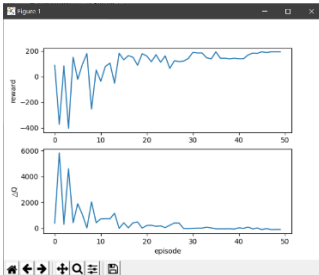
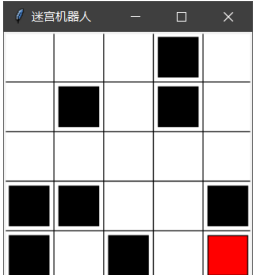
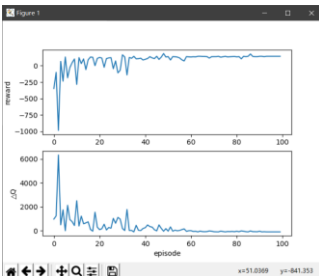
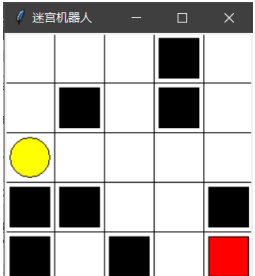
通过设置不同的参数值，分别进行了以下实验

(1) 研究迭代次数对结果的影响。

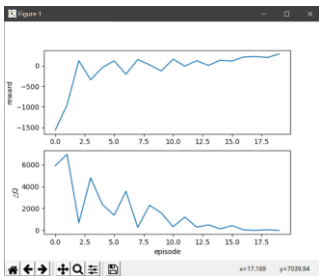
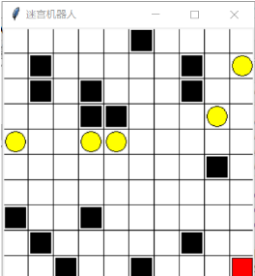
第一次随机生成迷宫时将迷宫奖励矩阵保存到文件reward.npy，在更改迭代次数时，从文件加载迷宫矩阵，从而保证更改迭代次数时迷宫保持不变。此外除迭代次数以外的所有参数均保持不变，设置如下：

```
PIXEL = 50
COIN_SCORE = 50
END_SCORE = 100
BLANK_SCORE = -1
WALL_SCORE = -5
```

5x5迷宫中，不同的迭代次数得到的最终奖励关系如下：

迭代次数	最终奖励	奖励折线图	最终迷宫图
20	133		
30	195		
50	195		
100	144		

10X10迷宫中，不同迭代次数对奖励的影响：

迭代次数	最终奖励	奖励折线图/ ΔQ	最终迷宫图
20	68		

30	130	
50	178	
100	236	
200	134	

通过观察以上两个设置的实验结果，可以发现，随着迭代次数的增加，机器人获得的奖励先增加后减少。在适中的迭代次数时，机器人可以兼顾宝藏和路径长度。当迭代次数过多时，Q-Learning可能会过拟合于一条最短的路径而忽略迷宫中的宝藏。

(2) 研究奖励设置对最终奖励的影响。

导致过拟合于最短路径的原因可能是由于宝藏奖励和终点奖励差异导致的。因此设置实验(2)，研究不同奖励设置对最终奖励的影响，使用8X8迷宫图（包含4个奖励），期间固定迭代次数为100次。迷宫如图8所示。

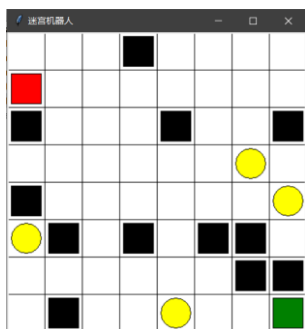


图 8 实验 2 中的迷宫

奖励设置	捡到宝藏	奖励折线图	最终迷宫图
宝藏 50 终点 100 空白 -1 撞墙 -5	1		
宝藏 100 终点 50 空白 -1 撞墙 -5	1		
宝藏 10 终点 50 空白 -1 撞墙 -5	1		
宝藏 50 终点 10 空白 -1 撞墙 -5	1		

通过实验(2)发现更改奖励设置不能有效的改善实验(1)中遇到的问题。宝藏奖励和终点奖励的比值越大，训练初期的离散程度越高，最终都能找到最短路径，但不能尽可能的寻找更多的宝藏。

7. 实验方法优点

通过上述实验可以发现，Q-Learning算法能够快速收敛，经过数秒，几十次迭代就能找到迷宫中的最短路径。

Q-Learning属于无监督学习，不需要训练集就能学到知识。

8. 实验方法缺点及改进

通过实验可以发现，Q-Learning不能很好的兼顾宝藏数和最短路径。在实验中，Q-learning算法倾向于找最短路径，而不是探索着更多的宝藏。

此外，在实验中，当宝藏奖励过大时，机器人可能出现困在一定范围内的问题。

通过分析Q-Learning的原理可以发现，对于Q-Learning而言，智能体是不会两次经过同一地点，如果两次经同一地点，则最终会陷入死循环。由于这个限制的存在，导致Q-Learning不会去捡某些特殊位置的宝藏。对于图9的迷宫而言。如果机器人捡起宝藏1，要想拿到宝藏2则必须从宝藏1往右走，即Q表中宝藏1位置最大值对应的行动是右。捡到宝藏2后机器人必须返回到宝藏1位置，由于宝藏1位置最大的行动概率是右，因此机器人会再次往右，陷入死胡同。

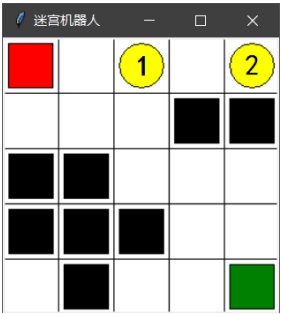


图 9 迷宫

导致这一问题出现的原因是在此程序中只建立了一个Q表，该Q表只能表示迷宫的初始状态。而实际对于有2个宝藏的迷宫，应该存在4种状态：2个宝藏、只有宝藏1、只有宝藏2、没有宝藏。如下图所示：

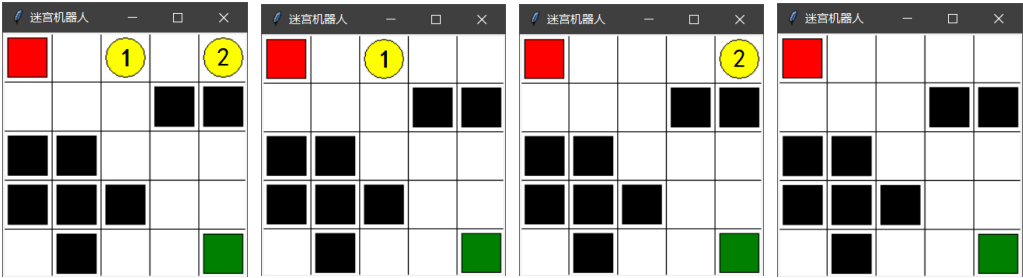


图 10 5x5 迷宫的 4 种状态

改进方法：根据宝藏的数量增加Q表的数量，为每一种状态设置一个Q表，当机器人捡

到宝藏时，跳转到对应的Q表。