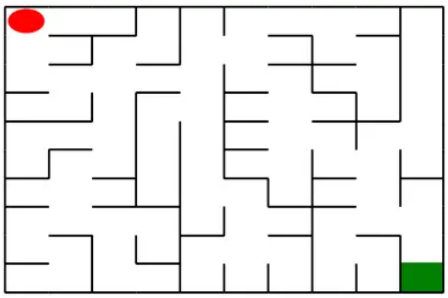
机器人自动走迷宫

3210104591 秦雨扬

实验内容

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
- 需要您分别实现**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。

实验原理

强化学习

强化学习作为机器学习算法的一种，其模式也是让智能体在“训练”中学到“经验”，以实现给定的任务。但不同于监督学习与非监督学习，在强化学习的框架中，我们更侧重通过智能体与环境的**交互**来学习。通常在监督学习和非监督学习任务中，智能体往往需要通过给定的训练集，辅之以既定的训练目标（如最小化损失函数），通过给定的学习算法来实现这一目标。
然而在强化学习中，智能体则是通过其与环境交互得到的奖励进行学习。
这个环境可以是虚拟的（如虚拟的迷宫），也可以是真实的（自动驾驶汽车在真实道路上收集数据）。

在强化学习中有五个核心组成部分，它们分别是：**环境（Environment）**、**智能体（Agent）**、**状态（State）**、**动作（Action）**和**奖励（Reward）**。

在某一时间节点 t：

- 智能体在从环境中感知其所处的状态 s_t
- 智能体根据某些准则选择动作 a_t
- 环境根据智能体选择的动作，向智能体反馈奖励 r_{t+1}

通过合理的学习算法，智能体将在这样的问题设置下，成功学到一个在状态 s_t 选择动作 a_t 的策略 $\pi(s_t) = a_t$。

QLearning 算法

Q-Learning 是一个值迭代（Value Iteration）算法。
与策略迭代（Policy Iteration）算法不同，值迭代算法会计算每个”状态“或是”状态-动作“的值（Value）或是效用（Utility），然后在执行动作的时候，会设法最大化这个值。
因此，对每个状态值的准确估计，是值迭代算法的核心。
通常会考虑**最大化动作的长期奖励**，即不仅考虑当前动作带来的奖励，还会考虑动作长远的奖励。

Q 值的计算与迭代

Q-learning 算法将状态（state）和动作（action）构建成一张 Q_table 表来存储 Q 值，Q 表的行代表状态（state），列代表动作（action）：

Q-Table	a_1	a_2
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$
s_3	$Q(s_3, a_1)$	$Q(s_3, a_2)$

在 Q-Learning 算法中，将这个长期奖励记为 Q 值，其中会考虑每个 ”状态-动作“ 的 Q 值，具体而言，它的计算公式为：

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

也就是对于当前的“状态-动作” (s_t, a)，考虑执行动作 a 后环境奖励 R_{t+1}，以及执行动作 a 到达 s_{t+1} 后，执行任意动作能够获得的最大的Q值 $\max_a Q(a, s_{t+1})$，γ 为折扣因子。

计算得到新的 Q 值之后，一般会使用更为保守地更新 Q 表的方法，即引入松弛变量 $alpha$ ，按如下的公式进行更新，使得 Q 表的迭代变化更为平缓。

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

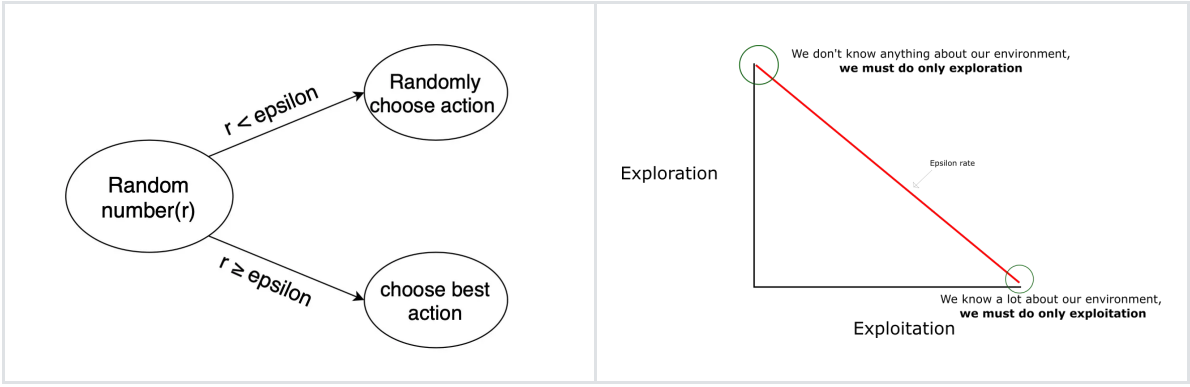
机器人动作的选择

在强化学习中，**探索-利用** 问题是非常重要的问题。
具体来说，根据上面的定义，会尽可能地让机器人在每次选择最优的决策，来最大化长期奖励。
但是这样做有如下的弊端：

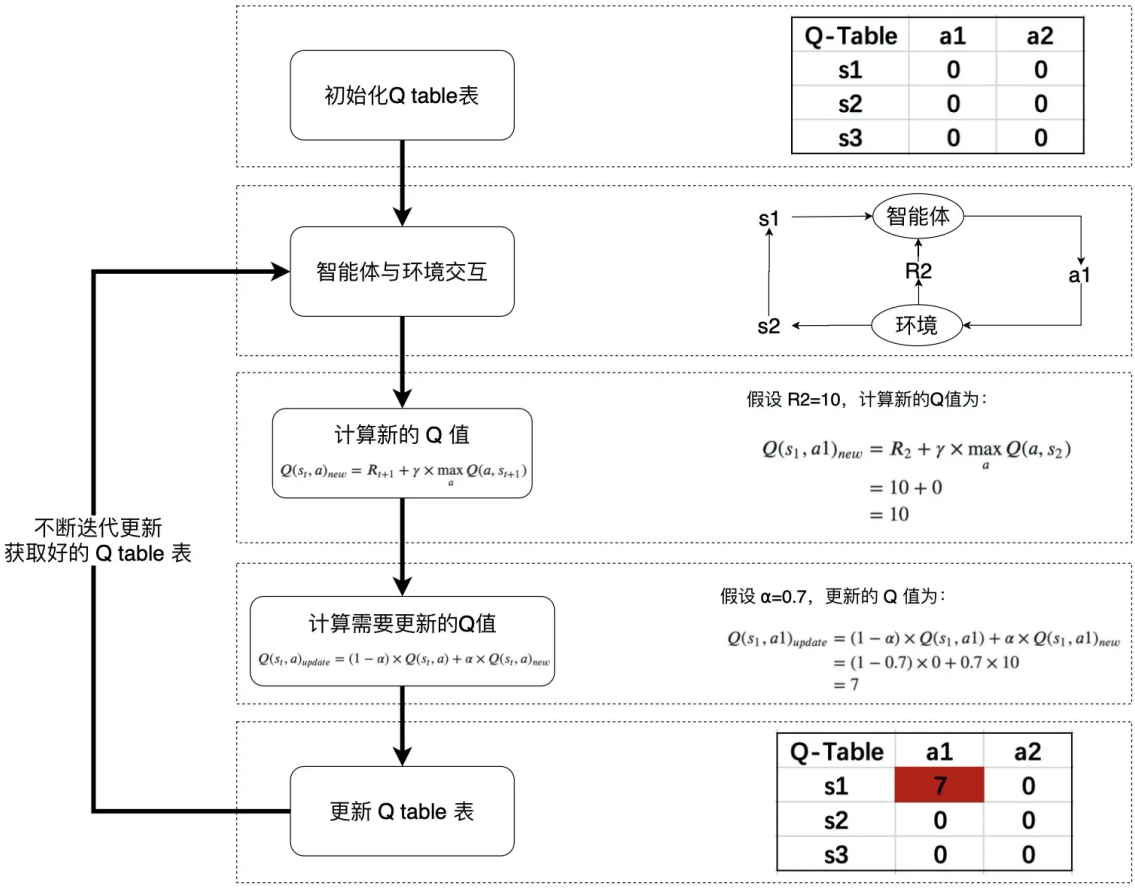
1. 在初步的学习中，Q 值是不准确的，如果在这个时候都按照 Q 值来选择，那么会造成错误。
2. 学习一段时间后，机器人的路线会相对固定，则机器人无法对环境进行有效的探索。

因此需要一种办法，来解决如上的问题，增加机器人的探索。
通常会使用 **epsilon-greedy** 算法：

1. 在机器人选择动作的时候，以一部分的概率随机选择动作，以一部分的概率按照最优的 Q 值选择动作。
2. 同时，这个选择随机动作的概率应当随着训练的过程逐步减小。



Q-Learning 算法的学习过程



DQN 算法介绍

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。(DQN 论文地址)

- **DQN 算法流程**

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-}=\theta$

For episode = 1, M **do**

  Initialize sequence $s_1=\{x_1\}$ and preprocessed sequence $\phi_1=\phi(s_1)$

  For t = 1,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t=\text{argmax}_a Q(\phi(s_t),a;\theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1}=s_t,a_t,x_{t+1}$ and preprocess $\phi_{t+1}=\phi(s_{t+1})$

    Store transition $(\phi_t,a_t,r_t,\phi_{t+1})$ in D

    Sample random minibatch of transitions $(\phi_j,a_j,r_j,\phi_{j+1})$ from D

    Set $y_j=\begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j+\gamma \max_{a'} \hat{Q}(\phi_{j+1},a';\theta^{-}) & \text{otherwise} \end{cases}$

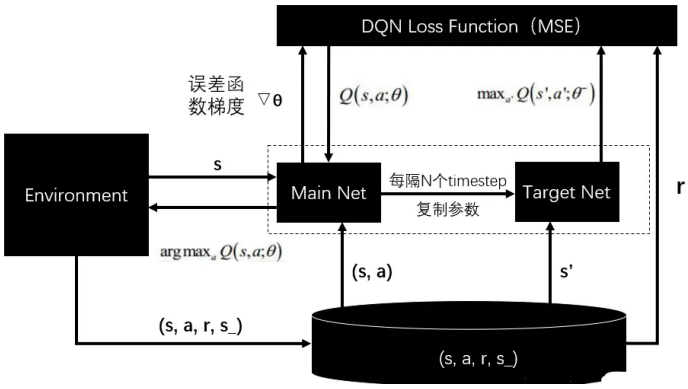
    Perform a gradient descent step on $\left(y_j-Q(\phi_j,a_j;\theta)\right)^2$ with respect to the network parameters θ

    Every C steps reset $\hat{Q}=Q$

  End For

End For

- **DQN 算法框架图**



题目一: 实现基础搜索算法(总分40分)

- 题目要求： 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
- 输入：迷宫
- 输出：到达目标点的路径

可以选用深度优先搜索算法或者最佳优先搜索 A* 算法。由于深度优先搜索算法比A*简单，所以我自然选择了深度优先搜索算法。

深度优先算法简而言之就是先遍历完某一个儿子，再遍历其他儿子。而这个时候递归就派上用场了，既然出入池的逻辑是先入后出，不如直接使用系统的栈。

但是这里要注意的一点是需要判断是否is_visit。因为加入迷宫中有环的话，深度优先搜索会在里面不断地绕，需要这个标志来保证不会重复走过同一个点。

```
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

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
}


is_visit_m=None
def deep_search(maze,current_node):
#     print(current_node,maze.destination)
    is_visit_m[current_node] = 1
    if current_node == maze.destination:  # 到达目标点
        return [-1]
    can_move = maze.can_move_actions(current_node)
    for a in can_move:
        new_loc = tuple(current_node[i] + move_map[a][i] for i in range(2))
        if not is_visit_m[new_loc]:
            path=deep_search(maze,new_loc)
            if(path is not None and len(path)>0):
                path.append(a)
#                 print(path)
                return path
    return []
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--------------------------------------
    start = maze.sense_robot()

    h, w, _ = maze.maze_data.shape
    global is_visit_m
    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=deep_search(maze,start)
    print(path)
    path.reverse()
    path.pop(-1)
    # ----------------------------------------------------------------------
    return path
```

题目二 DQNRobot

- **题目要求:** 编程实现 DQN 算法在机器人自动走迷宫中的应用
- **输入:** 由 Maze 类实例化的对象 maze
- **要求不可更改的成员方法：**train_update()、test_update() **注：不能修改该方法的输入输出及方法名称，测试评分会调用这两个方法。**
- **补充1:**若要自定义的参数变量，在 __init__() 中以 `self.xxx = xxx` 创建即可
- **补充2:**实现你自己的DQNRobot时，要求继承 QRobot 类，QRobot 类包含了某些固定的方法如reset(重置机器人位置),sense_state(获取机器人当前位置)..

本实验中的DQNRobot是基于TorchRobot进行修改的，我检查了其中的代码，发现其中除了EveryUpdate = 1 没有将main net和target net分开之外，没有任何的问题。但是在测试中，会看到机器人会卡在某一个点，或者走过去又走回来，亦或者一直走进死胡同。就算是有一次走通了再update一次又不行了。对此我非常的困惑，在网上查看了一些资料说是reward函数构造的不太好，但是我感觉也没什么问题。此外，我尝试在一开始就开启金钥匙，但是还是无济于事。

后来我参考了学长的代码，发现他其实是在作弊。他先开了金钥匙，在每次训练整个memory,同时每次尝试能不能走通，一直train到能走通，再开始正常的train_update，但是不更新参数，而test_update也是，所以一直可以走通。但是我在实际测试他的代码，对于高阶仍然是走不通。

我参考了这个离线的做法，也开启金钥匙，然后train整个memory，以此来看看我神经网络和Q-learning的效果。我调整多了非常多的参数，光reward都采用了非常多的办法，甚至试图将走一步的reward从+1变成-1,但是效果仍然不好。测试了n=11的情况，大概需要1000次backward可以找到一条通路，但是loss一直非常大，有数十万，也就是说用神经网络拟合的Q表与target相差很大。后来我将main net和target net分开，将EveryUpdate 设为了10，有一定的改善。然后也试了将Batch_size设为16/32，但是这个效果明显变差。同时也试着不开启金钥匙，使用普通的探索模式，但是几乎找不出来。最终还是采用了较为稳妥的离线方案。

后来我也根据原理做了一定的思考，我想是因为Q-table随着每次loss改变后，targe Q-table 也在改变，所以就处于一个一直在用神经网络拟合一个在变的函数，这几乎是不可能的。所以会有将main net和target net分开。但是我们的神经网络又无法再马上拟合，于是乎就效果不佳。于是提出来将main net和target net分开，实则就是每次Q-table的更新都使用main net基于原始的target net的新Q-table。但是神经网络的训练消耗大量算力，所以这个方法虽然可行但是耗时非常长。

于是基于这个原来我恍然大悟，原来main net到target net都是一次全新的拟合，那我就干脆一次次的训练网络当做独立的训练，进行EveryUpdate（此处设为maze_size** 2*5）次拟合，在这个过程中将learning rate进行手动调整。

```
loss_list = []
for params in self.optimizer.param_groups:
    params['lr'] =self.learning_rate
for i in range(self.EveryUpdate):
    # loss = self._learn(batch=64)
    loss = self._learn(batch=batch_size)
    # print("Training on idx: ",idx," loss: ",loss,"
lr",self.optimizer.state_dict()['param_groups'][0]['lr'])
    loss_list.append(loss)
print("Training on idx: ",idx," initial loss: ",loss_list[0]," loss:
",loss_list[-1]," lr",self.optimizer.state_dict()['param_groups'][0]['lr'])
```

```
        """ Minimize the loss"""
        loss.backward()
        self.optimizer.step()
        if self.step %(self.EveryUpdate//6)==0:
```

```
            for params in self.optimizer.param_groups:
                params['lr']*=0.5
        # self.schedule.step()
        # print(self.optimizer.state_dict()['param_groups'][0]['lr'])
        # self.schedule.step()

        """copy the weights of eval_model to the target_model"""
        if self.step % self.EveryUpdate == 0:
            self.target_replace_op()
            self.step=0
```

同时，我也对QNetwork进行了修改，尝试了不同的hidden_size，但是发现当网络流的size先变大后变小或者直接一路变小时的梯度计算较慢，最终仍然采取了线性的大小.也尝试了ELU,RELU,Softmax等非线性层，发现ELU与RELU效果接近，Softmax显著变慢。同时也尝试了每次main net给到target net后清空main net，但是效果不佳。最后的网络如下

```
class QNetwork(nn.Module, ABC):
    """Actor (Policy) Model."""

    def __init__(self, state_size: int, action_size: int, seed: int,
hidden_size=512):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
        """

        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.input_hidden = nn.Sequential(
            nn.Linear(state_size, hidden_size),
            nn.ReLU(True),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(True),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(True),
        )

        self.final_fc = nn.Linear(hidden_size, action_size)
    def _init_weights(self, m):
        if isinstance(m, nn.Linear):
            torch.nn.init.xavier_uniform_(m.weight)
            m.bias.data.fill_(0.0)
    def reset_parameters(self):
        self.input_hidden.apply(self._init_weights)
        self.final_fc.apply(self._init_weights)
    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = self.input_hidden(state)
        return self.final_fc(x)
```

在n=11时的一组测试结果如下，Q-table更新了32次后找到了一条通路，可以看到这里模型下效果还是不错的（"destination": -maze.maze_size ** 2 * 50. 约为5000，rmse约为60）。

```
Training on idx:  1  initial loss:  152563.140625  loss:  15.428552627563477  lr
0.00015625
Training on idx:  2  initial loss:  61703.7890625  loss:  66.4521713256836  lr
0.00015625
Training on idx:  3  initial loss:  150386.46875  loss:  34113.72265625  lr
0.00015625
Training on idx:  4  initial loss:  73682.09375  loss:  22852.05078125  lr
0.00015625
Training on idx:  5  initial loss:  47287.87890625  loss:  11195.234375  lr
0.00015625
Training on idx:  6  initial loss:  29035.505859375  loss:  3323.614501953125  lr
0.00015625
Training on idx:  7  initial loss:  23218.3046875  loss:  3822.65185546875  lr
0.00015625
Training on idx:  8  initial loss:  19481.439453125  loss:  3636.564697265625  lr
0.00015625
Training on idx:  9  initial loss:  19468.076171875  loss:  3550.540283203125  lr
0.00015625
Training on idx:  10  initial loss:  19571.025390625  loss:  5503.8408203125  lr
0.00015625
Training on idx:  11  initial loss:  18715.67578125  loss:  3098.815185546875  lr
0.00015625
Training on idx:  12  initial loss:  13336.138671875  loss:  1942.6514892578125
lr 0.00015625
Training on idx:  13  initial loss:  11377.828125  loss:  1068.9271240234375  lr
0.00015625
Training on idx:  14  initial loss:  9545.2900390625  loss:  2334.322998046875
lr 0.00015625
Training on idx:  15  initial loss:  10009.8876953125  loss:  1969.28759765625
lr 0.00015625
Training on idx:  16  initial loss:  9221.1689453125  loss:  2809.047607421875
lr 0.00015625
Training on idx:  17  initial loss:  9730.2548828125  loss:  3711.917724609375
lr 0.00015625
Training on idx:  18  initial loss:  9579.322265625  loss:  5461.5771484375  lr
0.00015625
Training on idx:  19  initial loss:  10097.33984375  loss:  5450.05126953125  lr
0.00015625
Training on idx:  20  initial loss:  8280.296875  loss:  4762.1806640625  lr
0.00015625
Training on idx:  21  initial loss:  7335.23388671875  loss:  3421.949951171875
lr 0.00015625
Training on idx:  22  initial loss:  6993.84326171875  loss:  4984.00732421875
lr 0.00015625
Training on idx:  23  initial loss:  7199.42724609375  loss:  3349.56884765625
lr 0.00015625
Training on idx:  24  initial loss:  7568.76513671875  loss:  3053.533447265625
lr 0.00015625
Training on idx:  25  initial loss:  7729.90966796875  loss:  4568.7294921875  lr
0.00015625
Training on idx:  26  initial loss:  7921.37744140625  loss:  3330.459228515625
lr 0.00015625
Training on idx:  27  initial loss:  6467.435546875  loss:  3462.94921875  lr
0.00015625
Training on idx:  28  initial loss:  6042.0517578125  loss:  3132.3740234375  lr
0.00015625
Training on idx:  29  initial loss:  5891.67431640625  loss:  3016.17822265625
lr 0.00015625
```

```
Training on idx:  30  initial loss:  5655.40771484375  loss:  3732.8447265625  lr
0.00015625
Training on idx:  31  initial loss:  6151.95068359375  loss:  3224.003173828125
lr 0.00015625
Training on idx:  32  initial loss:  5465.564453125  loss:  3807.381103515625  lr
0.00015625
arrive destination
```

源码

Robot.py

```
import numpy as np
import random

import torch
import torch.nn.functional as F
from torch import optim

from QRobot import QRobot
from Maze import Maze
from ReplayDataSet import ReplayDataSet
from QNetwork import QNetwork
from Runner import Runner


class Robot(QRobot):
    valid_action = ['u', 'r', 'd', 'l']

    ''' QLearning parameters'''
    epsilon0 = 0.5  # 初始贪心算法探索概率
    gamma = 0.94  # 公式中的 γ

    EveryUpdate = 300  # the interval of target model's updating

    """some parameters of neural network"""
    target_model = None
    eval_model = None
    batch_size = 32
    learning_rate = 0.01
    TAU = 1e-3
    step = 1  # 记录训练的步数

    """setting the device to train network"""
    device = torch.device("cuda:0") if torch.cuda.is_available() else
torch.device("cpu")
    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 * 50.,
            "default": 1.,
        })
```

```
        self.maze = maze
        self.maze_size = maze.maze_size
        print(self.device)
        """build network"""
        self.target_model = None
        self.eval_model = None
        self._build_network()
        self.EveryUpdate=maze.maze_size**2*5
        """create the memory to store data"""
        max_size = max(self.maze_size ** 2 * 3, 1e4)
        self.memory = ReplayDataSet(max_size=max_size)
        self.memory.build_full_view(maze=maze)
        self.loss_list = self.init_train()
        self.is_ok(1)
    def is_ok(self,is_print=0):
        self.reset()
        for _ in range(self.maze.maze_size ** 2 - 1):
            a, r = self.test_update()
            if(is_print):
                print(a,r)
            if r == self.maze.reward["destination"]:
                print("arrive destination")
                return True
        return False
    def init_train(self):
        batch_size = len(self.memory)

        # for i in range(int(self.maze_size **2 * 1.5)):
        #     loss = self._learn(batch=batch_size)
        #     loss_list.append(loss)
        #     # if(self.is_ok(0)):
        #     #     break
        idx=0
        while True:

            loss_list = []
            for params in self.optimizer.param_groups:
                params['lr'] =self.learning_rate
            for i in range(self.EveryUpdate):
                # loss = self._learn(batch=64)
                loss = self._learn(batch=batch_size)
                # print("Training on idx: ",idx," loss: ",loss,"
lr",self.optimizer.state_dict()['param_groups'][0]['lr'])
                loss_list.append(loss)
            idx+=1
            print("Training on idx: ",idx," initial loss: ",loss_list[0]," loss:
",loss_list[-1]," lr",self.optimizer.state_dict()['param_groups'][0]['lr'])
            if(self.is_ok(0)):
                break
        return loss_list
    def _build_network(self):
        seed = 0
        random.seed(seed)

        """build target model"""
        self.target_model = QNetwork(state_size=2, action_size=4,
seed=seed,hidden_size=self.maze_size**2*8).to(self.device)
```

```
        """build eval model"""
        self.eval_model = QNetwork(state_size=2, action_size=4,
seed=seed,hidden_size=self.maze_size**2*8).to(self.device)

        """build the optimizer"""
        self.optimizer = optim.Adam(self.eval_model.parameters(),
lr=self.learning_rate)

        # self.schedule=torch.optim.lr_scheduler.StepLR(self.optimizer,
self.EveryUpdate/6, gamma=0.5, last_epoch=-1)

    def target_replace_op(self):
        """
            Soft update the target model parameters.
            θ_target = τ*θ_local + (1 - τ)*θ_target
        """

        # for target_param, eval_param in zip(self.target_model.parameters(),
self.eval_model.parameters()):
        #     target_param.data.copy_(self.TAU * eval_param.data + (1.0 -
self.TAU) * target_param.data)

        """ replace the whole parameters"""
        self.target_model.load_state_dict(self.eval_model.state_dict())

    def _choose_action(self, state):
        state = np.array(state)
        state = torch.from_numpy(state).float().to(self.device)
        if random.random() < self.epsilon:
            action = random.choice(self.valid_action)
        else:
            self.eval_model.eval()
            with torch.no_grad():
                q_next = self.eval_model(state).cpu().data.numpy()  # use target
model choose action
            self.eval_model.train()

            action = self.valid_action[np.argmin(q_next).item()]
        return action

    def _learn(self, batch: int = 16):
        if len(self.memory) < batch:
            print("the memory data is not enough")
            return
        state, action_index, reward, next_state, is_terminal =
self.memory.random_sample(batch)

        """ convert the data to tensor type"""
        state = torch.from_numpy(state).float().to(self.device)
        action_index = torch.from_numpy(action_index).long().to(self.device)
        reward = torch.from_numpy(reward).float().to(self.device)
        next_state = torch.from_numpy(next_state).float().to(self.device)
        is_terminal = torch.from_numpy(is_terminal).int().to(self.device)

        self.eval_model.train()
        self.target_model.eval()

        """Get max predicted Q values (for next states) from target model"""
```

```
        Q_targets_next = self.target_model(next_state).detach().min(1)
[0].unsqueeze(1)

        """Compute Q targets for current states"""
        Q_targets = reward + self.gamma * Q_targets_next *
(torch.ones_like(is_terminal) - is_terminal)

        """Get expected Q values from local model"""
        self.optimizer.zero_grad()
        Q_expected = self.eval_model(state).gather(dim=1, index=action_index)

        """Compute loss"""
        loss = F.mse_loss(Q_expected, Q_targets)
        loss_item = loss.item()

        """ Minimize the loss"""
        loss.backward()
        self.optimizer.step()
        if self.step %(self.EveryUpdate//6)==0:
            for params in self.optimizer.param_groups:
                params['lr']*=0.5
        # self.schedule.step()
        # print(self.optimizer.state_dict()['param_groups'][0]['lr'])
        # self.schedule.step()

        """copy the weights of eval_model to the target_model"""
        if self.step % self.EveryUpdate == 0:
            self.target_replace_op()
            self.step=0

        """---update the step and epsilon---"""
        self.step += 1
        # self.target_replace_op()
        return loss_item

    def train_update(self):
        state = self.sense_state()
        action = self._choose_action(state)
        reward = self.maze.move_robot(action)
        # next_state = self.sense_state()
        # is_terminal = 1 if next_state == self.maze.destination or next_state ==
state else 0

        # self.memory.add(state, self.valid_action.index(action), reward,
next_state, is_terminal)

        """--间隔一段时间更新target network权重--"""

        # 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


if __name__ == "__main__":
    from QRobot import QRobot
    from Maze import Maze
    from Runner import Runner

    """  Deep Qlearning 算法相关参数： """

    epoch = 10  # 训练轮数
    maze_size = 11  # 迷宫size
    training_per_epoch = int(maze_size * maze_size * 1.5)

    """ 使用 DQN 算法训练 """

    g = Maze(maze_size=maze_size)
    r = Robot(g)
    runner = Runner(r)
    runner.run_training(epoch, training_per_epoch)

    # 生成训练过程的gif图, 建议下载到本地查看；也可以注释该行代码，加快运行速度。
    # runner.generate_gif(filename="results/dqn_size10.gif")
    """ create maze"""
    # epoch = 10
    # maze1 = Maze(maze_size=11)
    # maze_size = maze1.maze_size
    # maze1.reward = {
    #     "hit_wall": 10.0,
    #     "destination": -2 * maze_size ** 2,
    #     "default": 0.1,
    # }
    # r=Robot(maze1)
    # runner = Runner(r)
    # runner.run_training(epoch, training_per_epoch=int(maze_size * maze_size *
1.5))
    #
    # # 生成训练过程的gif图, 建议下载到本地查看；也可以注释该行代码，加快运行速度。
    # runner.generate_gif(filename="results/size5.gif")
```