**程序报告**

学号：2213410 姓名：徐俊智

1. **问题重述**

（简单描述对问题的理解，从问题中抓住主干，必填）

====================================================================

本次实验的目标是使用基础搜索算法和 Deep QLearning 算法，完成机器人自动走迷宫。

1. **设计思想**

（所采用的方法，有无对方法加以改进，该方法有哪些优化方向（参数调整，框架调整，或者指出方法的局限性和常见问题），伪代码，理论结果验证等… **思考题，非必填**）

====================================================================

基础搜索算法：我使用的是广度优先搜索算法，维护了一个队列来存储节点。

Deep QLearning 算法：我使用的是DQN算法

优化方向：

main.py

1.调整参数batch\_size从32改为128，提供更稳定的梯度估计

2.调整目的地奖励destination从-50改为-maze.maze\_size \*\* 2 \* 4，更快地引导机器人学习到达目的地的策略

3.调整Replay Memory 的 max\_size 从 max(self.maze\_size \*\* 2 \* 3, 1e4) 改为 max(self.maze\_size \*\* 2 \* 10, 1e4)，让网络有更多的数据来学习

4.新增 self.memory.build\_full\_view(maze)

5.ε-greedy 探索策略的更新规则从 self.epsilon = max(0.01, self.epsilon \* 0.995) 改为 self.epsilon = max(0.08, self.epsilon \* 0.43)，加快了 ε 的衰减速率

QNetwork.py

把  
self.input\_hidden = nn.Sequential(

nn.Linear(state\_size, 512), # 从状态维度映射到512维的隐藏层

nn.ReLU(False), # 激活函数

nn.Linear(512, 512), # 第二个512维的隐藏层

nn.ReLU(False), # 激活函数

)

self.final\_fc = nn.Linear(512, action\_size) # 从最后一个隐藏层映射到动作值

修改为  
self.input\_hidden = nn.Sequential(

nn.Linear(state\_size, 1024), # 从状态维度映射到1024维的隐藏层

nn.ReLU(False), # 激活函数

nn.Linear(1024, 1024), # 第二个1024维的隐藏层

nn.ReLU(False), # 激活函数

)

self.final\_fc = nn.Linear(1024, action\_size) # 从最后一个隐藏层映射到动作值

原先的网络有两个大小为 512 的全连接层，修改为两个大小为 1024 的全连接层后可以提供更多的容量，有助于特征捕捉。

1. **代码内容**

（能体现解题思路的主要代码，有多个文件或模块可用多个"===="隔开，必填）

====================================================================

**main.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

# 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

}

# 迷宫路径搜索树

class SearchTree(object):

def \_\_init\_\_(self, loc=(), action='', parent=None):

"""

初始化搜索树节点对象

:param loc: 新节点的机器人所处位置

:param action: 新节点的对应的移动方向

:param parent: 新节点的父辈节点

"""

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):

"""

添加子节点

:param child:待添加的子节点

"""

self.children.append(child)

def is\_leaf(self):

"""

判断当前节点是否是叶子节点

"""

return len(self.children) == 0

def expand(maze, is\_visit\_m, node):

"""

拓展叶子节点，即为当前的叶子节点添加执行合法动作后到达的子节点

:param maze: 迷宫对象

:param is\_visit\_m: 记录迷宫每个位置是否访问的矩阵

:param node: 待拓展的叶子节点

"""

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\_m[new\_loc]:

child = SearchTree(loc=new\_loc, action=a, parent=node)

node.add\_child(child)

def back\_propagation(node):

"""

回溯并记录节点路径

:param node: 待回溯节点

:return: 回溯路径

"""

path = []

while node.parent is not None:

path.insert(0, node.to\_this\_action)

node = node.parent

return path

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",...]

"""

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

start = maze.sense\_robot()

root = SearchTree(loc=start)

queue = [root] # 节点队列，用于层次遍历

h, w, \_ = maze.maze\_data.shape

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

peek = 0

while True:

current\_node = queue[peek] # 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\_m[current\_node.loc] == 0: # 如果该点存在叶子节点且未拓展

is\_visit\_m[current\_node.loc] = 1 # 标记该点已拓展

expand(maze, is\_visit\_m, current\_node)

peek+=1 # 开展一些列入队操作

for child in current\_node.children:

queue.append(child) # 叶子节点入队

else:

queue.pop(peek) # 如果无路可走则出队

peek-=1

return path

from QRobot import QRobot

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 torch\_py.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 = 1 # the interval of target model's updating

"""some parameters of neural network"""

target\_model = None

eval\_model = None

batch\_size = 128

learning\_rate = 1e-2

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 \* 4.,

"default": 1.,

})

self.maze = maze

self.maze\_size = maze.maze\_size

"""build network"""

self.target\_model = None

self.eval\_model = None

self.\_build\_network()

"""create the memory to store data"""

max\_size = max(self.maze\_size \*\* 2 \* 10, 1e4)

self.memory = ReplayDataSet(max\_size=max\_size)

self.memory.build\_full\_view(maze)

def \_build\_network(self):

seed = 0

random.seed(seed)

"""build target model"""

self.target\_model = QNetwork(state\_size=2, action\_size=4, seed=seed).to(self.device)

"""build eval model"""

self.eval\_model = QNetwork(state\_size=2, action\_size=4, seed=seed).to(self.device)

"""build the optimizer"""

self.optimizer = optim.Adam(self.eval\_model.parameters(), lr=self.learning\_rate)

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(len(self.memory))

else:

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()

"""copy the weights of eval\_model to the target\_model"""

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--"""

if self.step % self.EveryUpdate == 0:

self.\_learn(batch=len(self.memory))

"""---update the step and epsilon---"""

self.step += 1

self.epsilon = max(0.08, self.epsilon \* 0.43)

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

====================================================================

**QNetwork.py**

from abc import ABC

import torch.nn as nn

import torch

class QNetwork(nn.Module, ABC):

"""Actor (Policy) Model."""

def \_\_init\_\_(self, state\_size: int, action\_size: int, seed: int):

"""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, 1024),

nn.ReLU(False),

nn.Linear(1024, 1024),

nn.ReLU(False),

)

self.final\_fc = nn.Linear(1024, action\_size)

def forward(self, state):

"""Build a network that maps state -> action values."""

x = self.input\_hidden(state)

return self.final\_fc(x)

if \_\_name\_\_ == "\_\_main\_\_":

# os.environ["CUDA\_LAUNCH\_BLOCKING"] = "1"

device = torch.device("cuda:0") if torch.cuda.is\_available() else torch.device("cpu")

# device = torch.device("cpu")

net = QNetwork(2, 4, 0).to(device)

x = torch.tensor([1, 1]).float().unsqueeze(0).to(device)

#

# torch.nn.DataParallel(net, device\_ids=[0])

print(net(x))

1. **实验结果**

（实验结果，必填）

====================================================================



1. **总结**

（自评分析（是否达到目标预期，可能改进的方向，实现过程中遇到的困难，从哪些方面可以提升性能，模型的超参数和框架搜索是否合理等），**思考题，非必填**）

====================================================================

在本次实验中，我使用了广度优先搜索算法和DQN算法，最后成功实现了机器人自动走迷宫。

可能改进的方向：

1.对基础搜索算法，可以引入更高效的搜索策略，如A\*算法

2.对DQN算法，可以继续调整模型的超参数，如通过调整学习率和折扣因子等来提升性能。

实现过程中的困难：主要困难是超参数的选择，特别是学习率和折扣因子，它们对模型性能有显著影响。最后通过多次试验确定了一组合适的超参数，但这个过程相当耗时。

提升性能：模型的超参数和框架搜索在一定程度上是合理的，但仍有改进空间。通过更细致的调整和更深入的分析，可以进一步提升模型的性能。