**程序报告**

学号： 3190100923 姓名：陈志博

1. **问题重述**

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

本次实验希望分别通过搜索算法，Q-learning和Deep Q-learning算法分别来完成机器人走迷宫的机器人智能体。在Q-learning中，我们通过维护一张Q值，用通过贝尔曼方程对其进行不停的迭代尝试直至收敛，然后根据Q值表获取智能体在每个状态下的最优策略。但Q值表在状态和动作空间都是有限且低维的时候适用，当状态-动作空间高维且连续时，维护一张无限庞大的Q值表是不现实的。因此。DQN提出将Q-Table的更新问题变成一个函数拟合问题，相近的状态将得到相近的动作输出，即使用神经网络对动作-状态的Q值进行建模估计。在任一位置可执行动作包括：向上走 'u'、向右走 'r'、向 走 'd'、向左走 'l'。执行不同的动作后，有撞墙，走到出口，其余情况等3种情况，且每种情况会对应不同的reward。通过对reward函数的方程对objective function进行更新，使其收敛到最优解。

1. **设计思想**

对于迷宫这一对象最基础的算法是搜索算法，包括深度优先和广度优先搜索等算法。但对于过大的迷宫来说每次走迷宫都需要遍历迷宫的所有分支，运行时间较长。Q-learning是较为经典的强化学习方法，但Q-learning是对于有限状态的一个优化过程，仅能应付规模较小的迷宫。对于大规模的迷宫构造出所有状态的Table的代价过大，因此采用Deep Q-learning的算法，通过函数来近似状态函数，从而减小存储数据的开销。DQN的实现我采用了torch的架构来搭建了一个两层的简易神经网络，先对数据集进行训练将Q值收敛，再根据收敛所得的q值来进行动作选择，做出走迷宫的最终决策。

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

1. **代码内容**

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 = 0.91 # 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 = 32

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": -50.,

"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 \* 3, 1e4)

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

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(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=32)

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

self.step += 1

self.epsilon = max(0.01, self.epsilon \* 0.9)

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\_\_":

""" create maze"""

maze1 = Maze(maze\_size=5)

maze\_size = maze1.maze\_size

maze1.reward = {

"hit\_wall": 10.0,

"destination": -2 \* maze\_size \*\* 2,

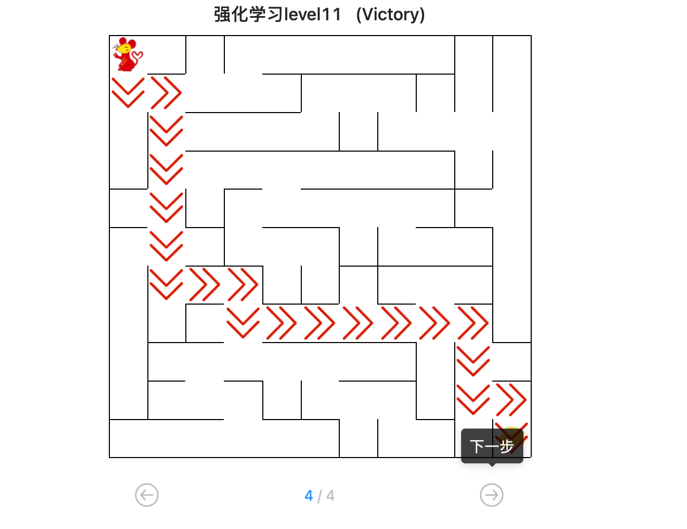
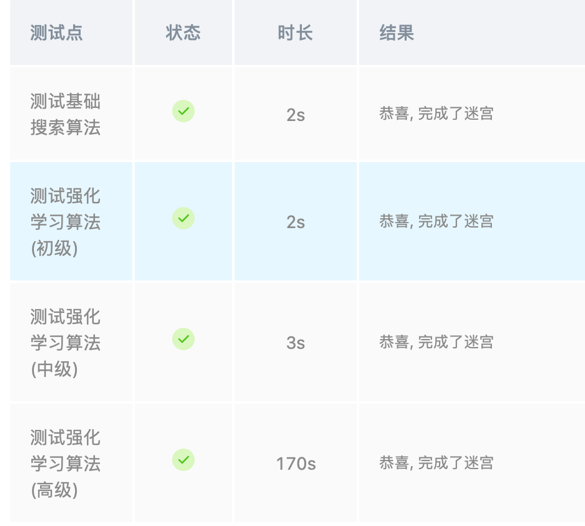
"default": 0.1,

}

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

1. **实验结果**

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



1. **总结**

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

实验过程中我发现DQN的神经网络的参数调整是十分不容易的，虽然所能调整的参数不多，但要综合探索和选择目前最优解的trade-off来更新数据。agent把对环境的感知，放入经验回放中，作为神经网络的样本集合。如果经验回放集合尺寸太小了，必然要选择丢弃部分经验，如果选择丢弃的经验是很重要的，就会给训练带来不稳定。因此，经验回放集合越大越好。同时因为时间原因我构建的神经网络结构较为简易，这也导致了对于大部分参数网络的实践效果并不突出，使得参数的调整更加困难。