程序报告

学号：　　3190100923　　　　　姓名：陈志博

一、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。

二、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。

==

三、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):
```

```python
        """
            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()
```

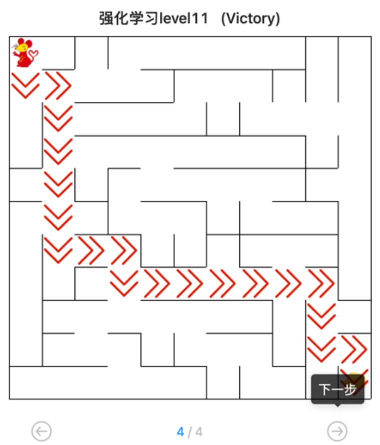
```
        """Get max predicted Q values (for next states) from target model"""
        Q_targets_next = self.target_model(next_state).detach().min(1)[0].unsqueeze(1)

        """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,
    }
```
==

四、实验结果

==

测试点	状态	时长	结果
测试基础搜索算法	✓	2s	恭喜, 完成了迷宫
测试强化学习算法(初级)	✓	2s	恭喜, 完成了迷宫
测试强化学习算法(中级)	✓	3s	恭喜, 完成了迷宫
测试强化学习算法(高级)	✓	170s	恭喜, 完成了迷宫



五、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。