强化学习：作业二

庄镇华 502022370071

2022 年 10 月 29 日

1 作业内容

在 gridworld 环境中实现 Q-learning 算法。

2 实现过程

2.1 实验过程

本次实验实现了 Q-Learning 算法，Q-Learning 如图 1所示，是强化学习中基于值的算法，Q 即为 Q（s,a）就是在某一时刻的 s 状态下，采取动作 a 能够获得收益的期望，环境会根据 agent 的动作反馈相应的回报 r。

$Q_0 = 0$, initial state
for i=0, 1, ...
$a = \pi_\epsilon(s)$
s', r = do action a
a' = $\pi(s')$
$Q(s,a)$+= $\alpha(r + \gamma Q(s',a') - Q(s,a))$
$\pi(s) = \arg\max_a Q(s,a)$
$s = s$'
end for

图 1: Q-learning 算法

算法的主要思想就是将 State 与 Action 构建成一张 Q-table 来存储 Q 值，然后根据 Q 值来选取能够获得最大收益的动作，需要注意的是 Q-Learning 算法最终的策略是不带 ϵ 探索的，Q-Learning 算法属于 off-policy 的算法。

首先在 algo.py 中补全 Q-Learning 的相关代码，具体代码如下：

```
class QLearningAgent(QAgent):
        def __init__(self):
                super().__init__()
                self.gamma = 0.99
                self.qtable = defaultdict(lambda : [0., 0., 0., 0.])

        def select_action(self, ob):
                o = str(ob[0]) + str(ob[1])
                all_q = np.array(self.qtable[o])
                idxes = np.argwhere(all_q == np.max(all_q))
                return random.choice(idxes)[0]

        def update(self, ob, a, r, next_ob, step):
                o = str(ob[0]) + str(ob[1])
                next_o = str(next_ob[0]) + str(next_ob[1])
                next_a = self.select_action(next_o)
                old_q = self.qtable[o][a]
                new_q = r + self.gamma * self.qtable[next_o][next_a]
                self.qtable[o][a] += self.get_lr(step) * (new_q - old_q)
```

QLearningAgent 类即为实现的算法，___init___ 函数初始化算法的超参数，包括折扣因子 γ 和 Q-table，其中学习率会根据训练轮数而变化，因此放到相关改进小节阐述；select_action 函数依据当前观测和 Q-table 选择 Q 值最大的动作，当多个动作均取得最大值时进行随机选择；update 函数负责利用公式 $Q(s,a)+=\alpha(r+\gamma Q(s',a')-Q(s,a))$ 更新 Q-table。

然后对 main.py 中的一些地方进行修改，主要修改三个地方，首先是将 QAgent 替换为自己实现的 QLearningAgent，并在每次和环境交互之后进行模型更新（agent.update）；由于 ϵ 取值过大，导致算法偏向于探索，难以收敛，因此我将 ϵ 值取到 0.0001；最后是删去绘图代码中关于 query 的部分，因为本次算法实现没有针对专家进行查询。

```
agent = QLearningAgent()
# start to train your agent
```

```
for i in range(num_updates):
        obs = envs.reset()
        for step in range(args.num_steps):
                epsilon = 1e-4
                if np.random.rand() < epsilon:
                        action = envs.action_sample()
                else:
                        action = agent.select_action(obs)
                obs_next, reward, done, info = envs.step(action)
                agent.update(obs, action, reward, obs_next, i * args.num_steps + step)
                obs = obs_next
                if done:
                        envs.reset()
```

2.2 相关改进

原始代码中 ϵ 探索机制中 ϵ 取值过大，导致算法震荡，难以收敛，因此本次实验中地图较为简单，在探索和利用的权衡里应该偏向利用，因此我将 ϵ 值取到 0.0001。

并且我针对不同的训练轮次设置不同的学习率，在初始时，学习率较小，防止模型往错误的梯度方向更新太远；中间阶段，逐步调大学习率，使得模型向最优点快速前进；最终模型收敛时，调低学习率，使得模型可以保持稳定在最优解附近。

```
def get_lr(self, step):
        if step <= 2000:
                return 0.1
        elif step <= 10000:
                return 1
        elif step <= 20000:
                return 0.01
        else:
                return 0.001
```

3 复现方式

在主文件夹下运行 python main.py.

4 实验效果

通过调整探索率 ϵ、学习率 α 和折扣因子 γ，在多次实验后找到了一个相对较优的参数组合，最后找到的最优结果如下：$\epsilon = 0.0001, \gamma = 0.99$，学习率随着训练轮数而变化（具体情况见相关改进小节），此时得到的训练曲线收敛较快且最终效果较为稳定。

描述累计奖励和样本训练量之间的关系?

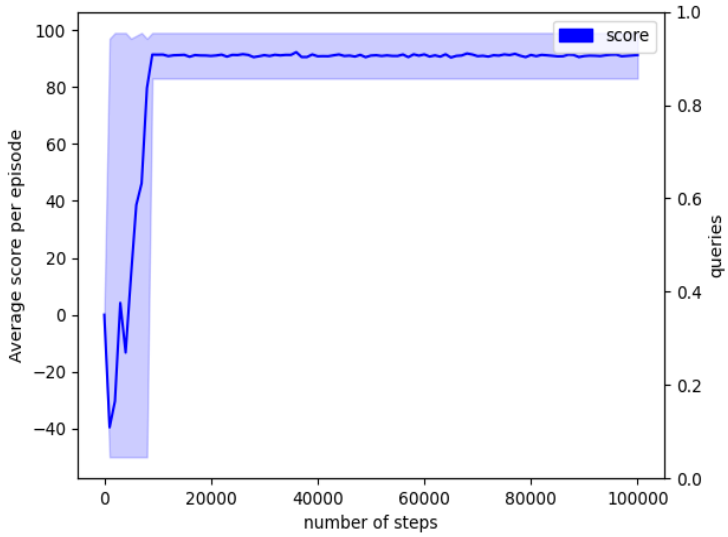

图 2: Q-leaning 算法

开始时，累计奖励随着样本训练量逐步上升，但偶有波动，且最大值和最小值差值较大，说明此时策略还不稳定；当累计奖励均值达到 92 左右开始收敛，且最大值和最小值差值较小，此时即使样本训练量继续增加，累计奖励依旧保持在最优附近，说明算法已经稳定收敛。

5 小结

在这次实验中，我发现在游戏环境相对简单，状态和动作空间相对小的情况下，无需实用深度网络，QLearning 算法本身就能取得不错的效果；

并且只要超参数调节的较为合理，也能达到较快的收敛速度，但 QLearning 算法本身也有很多限制，比如仅仅适合状态和动作空间较小且离散的问题，算法占用空间过大等，因此我更期待后续对 DQN 及其变体的学习，利用强化学习解决更多的实际问题。