深度强化学习方法第四次作业说明文件：**Maze 迷宫游戏的 Dyn-Q 算法实现**

一、问题重述

本次实验要求通过 Dyna 算法解决迷宫游戏，其中迷宫游戏为一个 9×6 的方格迷宫，其中分布着障碍物，agent 在每个状态/方格处可以采取上下左右四个动作，从 S 出发到 G 的路径越短越好，到达 G 点的 Reward=1，要求画出 planning=0，10，50 的迭代次数，画出迭代第 1-10 步的最优 policy、value 值。

在第七讲中，我们通过 n 步时序差分方法统一了 MC 和 TD(0)两类方法，一种类似资格迹的方法，学习了 n 步 Sarsa 以及 n 步期望、n 步期望 Sarsa、回溯树等算法。在本章中，我们从一个统一的视角来学习考虑之前的一系列强化学习方法，既有**具备环境模型**的方法，如动态规划和启发式搜索，也包括**没有环境模型**的方法，如蒙特卡洛方法和时序差分方法。这些方法分别被称为基于模型和无模型的强化学习方法。基于模型的方法将**规划**作为其主要组成部分，而无模型的方法则主要依赖于**学习**。虽然这两种方法之间存在着很大的差异，但它们也有很多相似之处，特别是这两类方法的核心都是价值函数的计算。同时所有的方法都基于对未来事件的展望，来计算一个回溯价值，然后使用它作为目标来更新一个近似价值函数。

其中值得一提的是：我们所说的环境的模型，指的是一个智能体可以用来预测环境对其动作的反应的任何事物。给定一个状态和一个动作，作为环境的反应结果，模型就会产生后继状态和下一个收益的预测。如果模型是随机的，那么后继状态和下一个收益有好几种可能，每个都有一定的发生概率。一些模型生成对所有可能的结果的描述及其对应的概率分布，这种模型被称为**分布模型**。另一种模型从所有可能性中生成一个确定的结果，这个结果通过概率分布采样得到，我们称这种模型为**样本模型**。前者比后者能力更强，可以生成后者，但是在许多场景下获得样本模型会更加容易。我们所说的规划，是指任何以环境模型为输入，并生成或改进与它进行交互的策略的计算过程，分为状态空间规划，以及方案空间规划。基于此，Dyna 算法是一类集成在一起的规划、动作和学习方法。对一个规划智能体来说，实际经验至少扮演了两个角色：它可以用来改进**模型**（使模型与现实环境更精确地匹配）；另外，它可用于直接改善前几章中介绍的强化学习中的**价值函数和策略**。前者称为模型学习，即所谓通过模型规划，进行间接强化学习。后者则称为直接强化学习（direct RL）。经验、模型、价值和策略之间可能的关系如下图所示，每个箭头表示产生影响和改进的关系方向。可以看到“经验”是如何直接地或通过模型间接地改善价值函数和策略的。

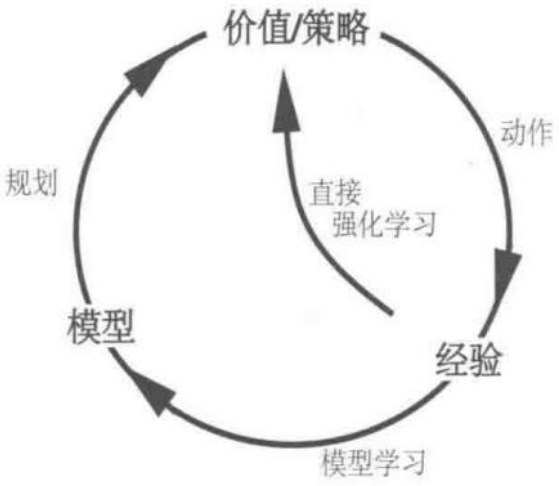


图 1：经验、模型和价值学习的关系

Dyna-Q 包括了上图及上述所有过程：规划、动作、模型学习和直接强化学习，该过程会持续迭代。这里的规划方法是随机采样单步 Q 规划方法。一般采用基于表格型的策略，在每一次$S_t, A_t \to R_{t+1}, S_{t+1}$之后，模型在它的表格中会为$S_t, A_t$建立条目，记录环境在这种情况下产生的转移结果的预测值R_{t+1}, S_{t+1}。因此，如果对模型之前经历过的“状态—动作”二元组进行查询，将返回最后观察到的后继状态和后继收益作为其预测值。

从概念上讲，规划、动作、模型学习和直接强化学习在 Dyna 的智能体中是同时发生并行进行的。然而，对于在串行计算机中的具体实现，在每一步我们都需要指定它们发生的顺序。在 Dyna-Q 中，动作执行、模型学习和直接强化学习过程只需要很少的计算，我们假设它们只消耗一小部分时间。每一步的剩余时间都可以用于规划过程，而这个过程是计算密集型的。我们假设在每个步骤之后，即动作执行、模型学习和直接强化学习之后，都有时间来完成 Q-规划算法的 n 次迭代（算法步骤 a ~ c）。在下面的框中给出了 Dyna-Q 的算法流程伪代码，其中，Model（s，a）表示基于“状态—动作”二元组（s，a）预测的后继状态和收益的内容。直接强化学习、模型学习和规划分别由步骤（d）、（e）和（f）来实现。如果省略（e）和（f），则剩下的算法是单步表格型 Q 学习算法。Dyna-Q 算法如下所示：

表格型 Dyna-Q 算法

对所有的 $s \in \mathcal{S}$ 和 $a \in \mathcal{A}(s)$，初始化 $Q(s,a)$ 和 $Model(s,a)$

无限循环：

(a) $S \leftarrow$ 当前 (非终止) 状态

(b) $A \leftarrow \varepsilon$-贪心(S,Q)

(c) 采取动作 A；观察产生的收益 R 以及状态 S'

(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

(e) $Model(S,A) \leftarrow R, S'$ （假设环境是确定的）

(f) 重复 n 次循环：

$S \leftarrow$ 随机选择之前观察到的状态

$A \leftarrow$ 随机选择之前在状态 S 下采取过的动作 A

$R, S' \leftarrow Model(S,A)$

$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

图 2：Dyna-Q 算法

在本次实验中，考虑一个 9×6 的迷宫游戏，要求从起点到终点的路径越短越好，到达 G 点 Reward=1。Agent 共有 54 个可能状态，状态下有上下左右四个动作。要求画出 planning=0, 10, 50 的迭代次数，以及迭代第 1-10 步的最优 policy, value 值。

二、实验环境

硬件环境 PC 机,CPU Intel(R)Core (TM) I7-9750H@2.60GHz 内存 16GB

软件环境 Visual Studio Code，python 版本 3.7.6

三、实验方法

1、 Maze 迷宫问题建模:

在本次实验中，我们将迷宫地图看作一个有（6 rows×9 columns）54 个状态，4 个动作的有限状态马尔可夫过程，其中地图如下图所示，其中有 7 个地方是障碍物，模拟在迷宫中从初始点到终点的寻路。agent 目标是从起点 S 到终点 G，agent 的动作有上下左右四个，只有走到终点获得 reward=1。Dyna-Q 学习中采样策略的方法是ε贪婪法。

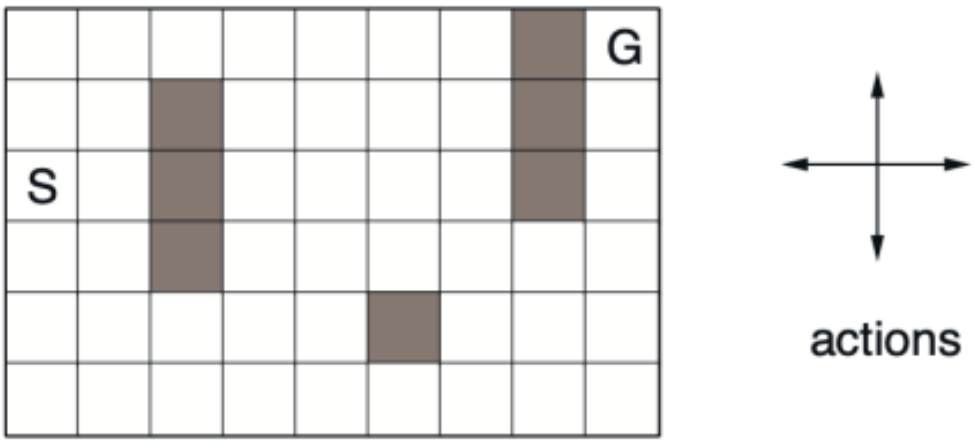


图 3：迷宫地图

因此在实验代码中会初始化一个有着 4 个动作, rows×columns 个状态环境类，并且环境类中包含着对是否是终止状态、是否是非法状态（出界以及遇到障碍物）、状态转移和 reward 获取等相关函数，类的构成思想类似第三次作业中的 cliffwalking 问题。

```
class Env:
    def __init__(self,environment,start,goal) -> None:
        self.start = start
        self.goal    = goal
        self.environment = environment
        self.actions = {"r":(0,1),
                        "l":(0,-1),
                        "d":(1,0),
                        "u":(-1,0)}  # basically there are four actions ,and these are represented as keys and the values re nothing but the indices by which state will change
        self.state = self.start # by default the state is start, and it will be updated over the time

        self.x_limit,self.y_limit = self.environment.shape
        self.states = np.array([state for state in product(np.arange(self.x_limit),np.arange(self.y_limit ))]) # every index of the environment
        self.tuple_sum = lambda state,action:tuple(map(sum,zip(state,self.actions[action]))) # does tuple sum

    def is_terminal(self,state):
        # state is in the form of indices
        # only goal is treated as the terminal state. When the agent reaches the goal, it return to the start state,and begins new episode

        if state == self.goal:
            return True
```

```
            return False

        def next_state(self,state,action):
            # For each of the four actions, the agent deterministically moves to the
neighboring corresponding states
            # except when movement is blocked by an obstacle or the edge of the maze, in
which case the agent
            # remains where it is
            nxt_state = self.tuple_sum(state,action)
            # check if next state is inside boundary or is not an obstacle
            # in a valid next_state
            if (nxt_state in map(tuple,self.states)) and (self.environment[nxt_state] == 0):
                self.state = nxt_state
                return nxt_state
            # in a invalid state so keep the original state
            else:
                self.state = state
                return state

        def reward(self,state,action):
            # Reward is zero on all transitions, except those into the goal state, on which it is
+1.
            next_state = self.next_state(state,action)
            if next_state == self.goal:
                return 1
            else:
                return 0

        def transition(self,state,action):
            # S_t, A_t -> S_t+1, R_t+1
            return self.reward(state,action),self.next_state(state,action)

        def reset(self):
            # Whenever the robot transitioned to terminal state , reset the position to start
position and begin new round
            self.state = self.start
            return    self.state
```

2、 问题解决与结果展示：

Dyna-Q：

在 Dyna-Q 算法中，选取初始动作价值为零，$\gamma = 0.95, \alpha = 0.1$以及$\varepsilon = 0.1$做为默认参数，其中前两者为 Dyna-Q 价值迭代过程中的折扣参数和步长加权参数，ε是贪心算法的超参数，这样的选取是符合一般情况的。在 Dyna-Q 类中，我们在初始化函数时传入参数，并且初始化得到状态-动作价值 Q 字典模型。代码如下所示：

```
class DynaQ:
    def __init__(self,env,gamma,alpha,n,epsilon,max_episodes = 50) -> None:
        self.env = env
        self.gamma = gamma
        self.alpha = alpha
        self.n = n   # number of planning steps
        self.epsilon = epsilon
        self.max_episodes = max_episodes # max number of episodes the simulation is run
for

    def initialize(self):
        Q = {}
        Model = defaultdict(dict)
        for state in self.env.states:
            state = tuple(state)
            #checking is that action can be taken because of the boundaries
            Q[state] = {action:0 for action in self.env.actions}# if
self.env.tuple_sum(state,action) in map(tuple,self.env.states) }
            # Model[state] = {action:(0,None) for action in self.env.actions} # initially we
do not know the model
        return Q,Model
```

除了初始化函数外，此类中还含有 Dyna-Q 方法的迭代过程函数，在 50 步的迭代过程中，每次迭代中通过 n=0.10.50 分别传入函数为三个不同的 planning 值，得到过程中的步数和最优策略，价值函数。最后可视化得到结果。其中 Dyna-Q 函数与算法的对应关系在代码中注释，关键的算法中**价值函数更新代码**以及 **planning 循环**的关键代码加粗表示：

```
def dynaQ(self):
    Q,Model = self.initialize()
    episode = 0
    episodes =[i+1 for i in range(self.max_episodes)] # store episode
    steps = []
    while episode<self.max_episodes:
        episode+=1
        S = self.env.start # self.env.state # in the beginning it is same as start
        step = 0
        while S != self.env.goal:
            step+=1
            # S = self.env.state
            # # (a) S current (non-terminal) state
            A = self.epsilon_greedy(S,Q)
            # (b) A "-greedy(S,Q)
            R,S_ = self.env.transition(S,A)
            # (c) Take action A; observe resultant reward, R, and state, S_
            S_temp = S_
            Q[S][A] = Q[S][A] + self.alpha*(R+self.gamma*max(Q[S_].values())-
```

```
Q[S][A])
                # (d) Update Q like you do in Q-learning
                Model[S][A] =R,S_
                # (e) Model(S,A) R, S0 (assuming deterministic environment)
                # (f) Loop repeat n times:(planning steps)
                for _ in range(self.n):
                    S = random.sample(Model.keys(),1)[0] # S - random previously observed state
                    A = random.sample(Model[S].keys(),1)[0] # A - random action previously taken in S
                    R,S_ = Model[S][A] # R, S_ from Model(S,A)
                    Q[S][A] = Q[S][A] + self.alpha*(R+self.gamma*max(Q[S_].values())-Q[S][A])# Update Q like you do in Q-learning
                # print(step,S_temp)
                S = S_temp
            steps.append(step) #steps per episode
            # print(episode, step)
        return episodes,steps
```

实验过程中 planning=0，10，50 的迭代次数如下图所示，实验结果类似书中的样例图值得一提的是，对于所有 n 值，第一幕所需要的迭代次数是相同的，这个数据没有体现在图中。在第一幕之后，对于所有的 n 值，性能都有所改善，但是对于更大的 n 值来说则改善要快得多。n = 0 的智能体是一个无规划的智能体，只使用直接强化学习（单步 Q 学习）方法。它是在这个问题上最慢的智能体。无规划智能体用了大约 25 幕才取得最优性能，然而 n = 10 的智能体只用了 5 幕，n = 50 时只用了 3 幕。

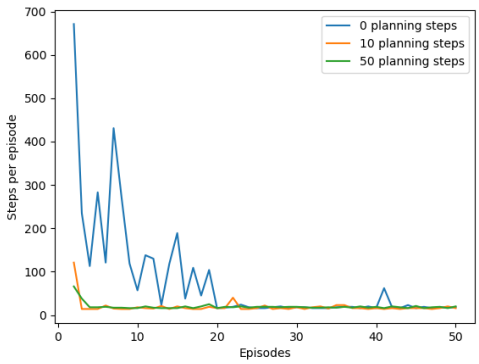
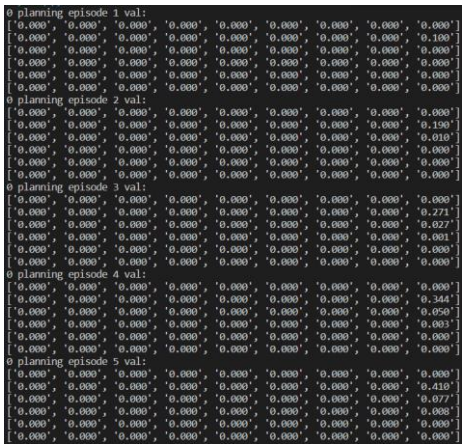


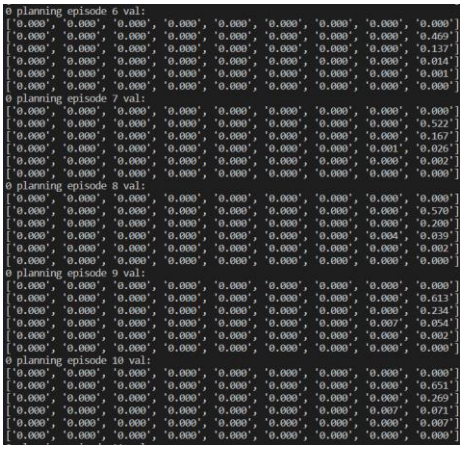
图 4：planning=0,10,50 的迭代次数

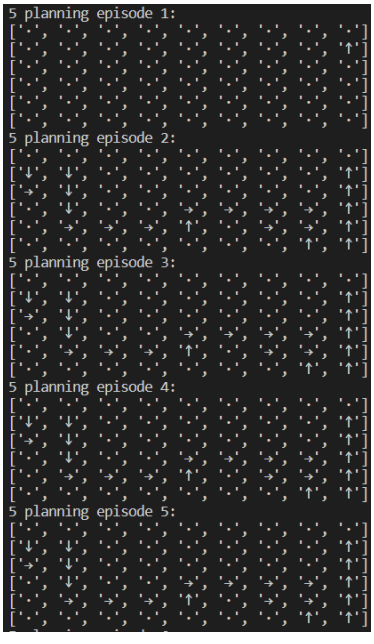
下图显示了 planning=0,10,50 的最优 policy,value 值。也反应出了带规划的 agent 为什么比无规划的智能体能更快地找到解决方案。图中显示的是 n = 0,10,50 时的智能体在第 1-10 幕的中间所找到的策略。如果没有规划（n = 0），则每一幕只会给策略学习增加一次学习机会，即智能体仅仅在一个特定时刻（该幕最后一个时刻）进行了学习。而有规划的时候，虽然在第一幕中仍然只有一步的学习，但是到第二幕时，可以学出一个宽泛的策略，于是不用等到这一幕结束就可以做多次回溯更新计算，这些计算几乎可以回溯到初始状态。这个策略是智能体在初始状态附近徘徊时通过规划过程构建的。到第三幕结束时，planning=50 的带规划智能体将找到一个完整的最优策略，达到完美的性能表现


0 planning episode 1:
0 planning episode 2:
0 planning episode 3:
0 planning episode 4:
0 planning episode 5:


0 planning episode 6:
0 planning episode 7:
0 planning episode 8:
0 planning episode 9:
0 planning episode 10:


0 planning episode 1 val:
0 planning episode 2 val:
0 planning episode 3 val:
0 planning episode 4 val:
0 planning episode 5 val:


0 planning episode 6 val:
0 planning episode 7 val:
0 planning episode 8 val:
0 planning episode 9 val:
0 planning episode 10 val:


5 planning episode 1:
5 planning episode 2:
5 planning episode 3:
5 planning episode 4:
5 planning episode 5:


5 planning episode 6:
5 planning episode 7:
5 planning episode 8:
5 planning episode 9:
5 planning episode 10:

图 5：planning=0,10,50 的最优 policy,value 值

3、额外说明

本次的代码在 dyna-q 文件中，直接运行可以生成 planning=0,10,50 的迭代次数的图片以及在命令行输出中得到最优 policy 和 value 值。代码仓库亦可见

https://github.com/Mu-Yanchen/rl_hw , 代码参考 https://github.com/shivakumar-tekumatla/Dyna-Q