Dyna-Q算法

简介

基于模型的强化学习设定：

强化学习中的“模型”，通常指与智能体进行交互的环境模型，即对环境状态转移概率和奖励函数的建模

根据环境模型的有无强化学习算法分为**基于模型的强化学习**和**无模型的强化学习**

无模型强化学习在时序差分算法的简介中已经介绍了，所以接下来详细说一下基于模型的强化学习

在基于模型的强化学习中，模型可以是事先已知，也可以根据智能体与环境交互采样到的数据学习得到，然后利用该模型进行策略提升或者价值估计

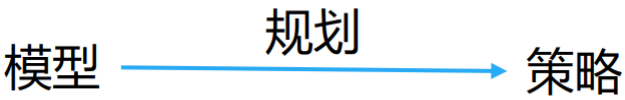
基于模型的强化学习算法：

第四章讨论的动态规划算法中的策略迭代和价值迭代就是基于模型的强化学习算法，这两种算法中的环境模型已知，本章介绍的**Dyna-Q算法**也是基于模型的强化学习基本的算法，不过它的环境模型的通过采样数据估计得到

但是采样数据估计的模型或者已知模拟出的环境模型可能并不准确，不能完全代替真实环境，因此基于模型的强化学习算法收敛的期望回报不如无模型的强化学习算法

规划

输入一个模型，输出一个策略的搜索过程，就是planning（规划）



这个过程比较抽象，下面是规划具体的分类：

- 状态空间的规划
  - 主要是依据状态的转换、推理来做相应的规划，在状态空间搜索最佳策略
- 规划空间的规划（使用不多）
  - 在规划空间搜索最佳策略，包括遗传算法和偏序规则
  - 这时，一个规划就是一个动作集合以及动作顺序的约束
  - 这时的状态就是一个规划，目标状态就是完成任务的规划

通过对模型采样生成的模拟数据，建立**规划的通用框架**，即基于模型得到模拟经验，通过回溯得到每个状态比较精准的值函数，基于值函数最终确定策略

它的优点在于任何时间都可以被打断或者重定向，比如在下棋时棋子落下这个状态，我们就马上针对这个状态重新开始进行规划，其次在做需要限时完成的事情，根据经验（采样模拟经验）我们可以估计这段时间能不能完成来估计是否值得（回溯值函数），从而采取合适的行动（策略）

即规划是一种即插即用的比较好的一种方法，在复制问题下，进行小而且增量式的时间步规划是非常有效的，比如针对这个状态稍微多想向前的几步判断优劣进行规划，那么最后策略将会有较好的提升

规划与学习：

规划和学习是两个概念，二者具有相同点也有不同点

不同点：

- 规划：利用模型产生的模拟经验，来进行策略的提升
- 学习：利用环境产生的真实经验，来进行策略的提升

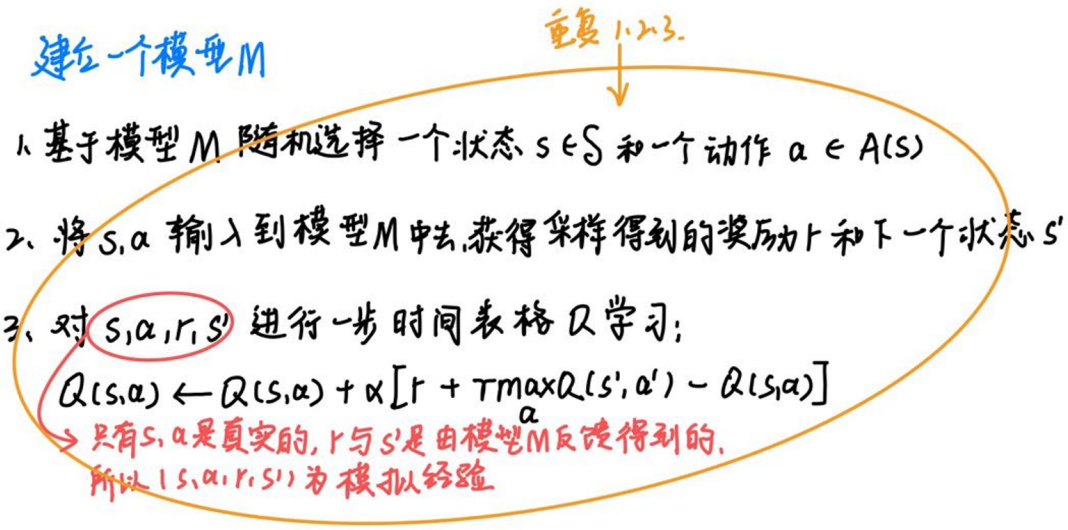
对一个环境建立比较好的模型，利用该模型采样出来模拟经验，基于经验来去提升策略。就相当于人的大脑在对一个环境进行演绎，然后推敲接下来采取如何行动使之达到更好的目标，来提升策略，这就是规划。比如我们做的人生规划、职业规划等等……

相同点：

- 通过回溯（back-up）更新值函数估计，无论是看到的数据，还是模拟出的数据都可以去判断优劣估计价值
- 统一来看，学习的方法可以用在模拟经验上

Q-planning：

Q-planning是一个非常基础的做规划的方法，也可以叫做基于表格的Q规划



所以Q-planning为什么不叫Q-learning就是因为在进行迭代更新学习时使用的是模拟经验（见上图红色分析）而不是真实经验

Dyna-Q

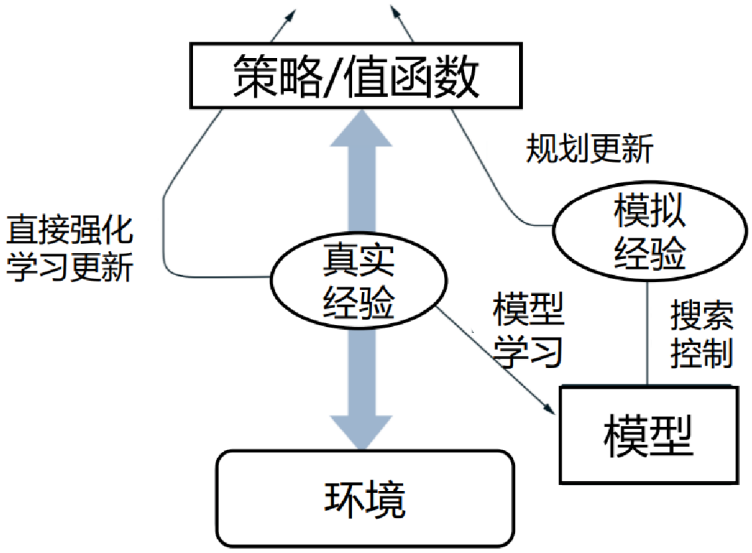
原理：

Dyna-Q是基于规划算法非常经典非常基础的算法

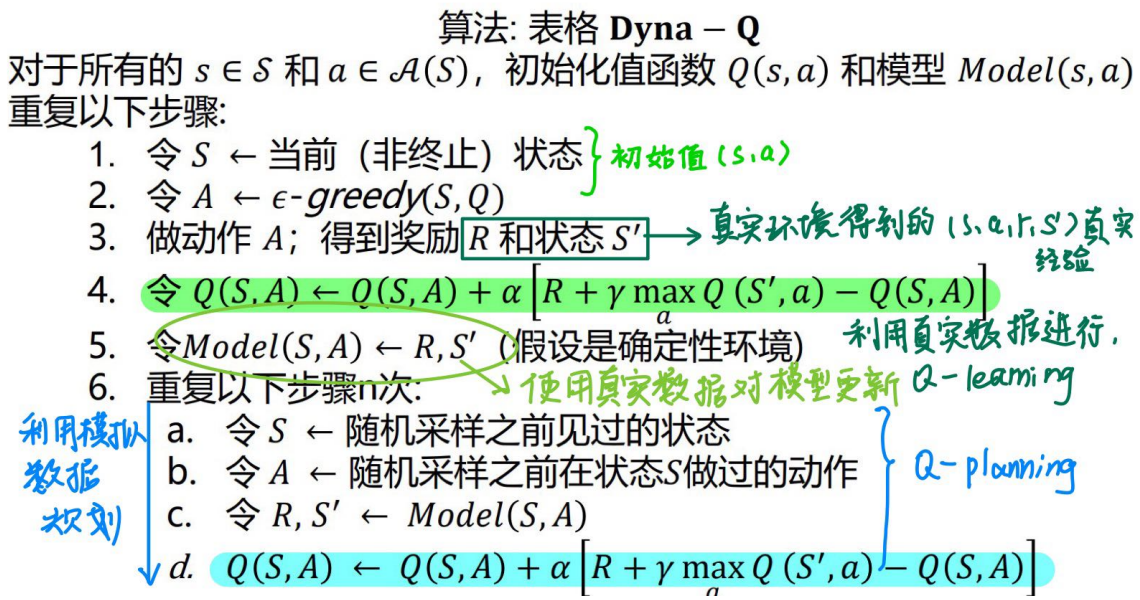
Dyna一词指集成了**规划、决策、学习**为一体的学习框架

- 首先我们可以通过智能体与环境交互得到的真实的数据来更新我们的模型M（模型学习）
- 接下来我们可以根据我们的模型M去做相应的Q-planning或者直接根据真实数据的经验本身我们可以做直接的强化学习，即Q-learning

- 二者同时做的话就是Dyna-Q算法，所以规划学习和直接强化学习对策略和值函数的更新的配合，就是Dyna算法调整的关键



算法流程：



在每次与环境进行交互执行一次 Q-learning 之后，Dyna-Q 会做n次 Q-planning。其中 Q-planning 的次数**N**是一个事先可以选择的超参数，当其为 0 时就是普通的 Q-learning

上述 Dyna-Q 算法是执行在一个离散并且确定的环境中，所以当看到一条经验数据（s,a,r,s'）时，可以对模型做出更新： M(s,a) ← r,s'

实现：

```
class DynaQ:
    """ Dyna-Q算法 """
    def __init__(self,
                 ncol,
                 nrow,
                 epsilon,
                 alpha,
```

```
                gamma,
                n_planning,
                n_action=4):
        self.Q_table = np.zeros([nrow * ncol, n_action])  # 初始化Q(s,a)表格
        self.n_action = n_action  # 动作个数
        self.alpha = alpha  # 学习率
        self.gamma = gamma  # 折扣因子
        self.epsilon = epsilon  # epsilon-贪婪策略中的参数

        self.n_planning = n_planning  #执行Q-planning的次数, 对应1次Q-learning
        self.model = dict()  # 环境模型

    def take_action(self, state):  # 选取下一步的操作
        if np.random.random() < self.epsilon:
            action = np.random.randint(self.n_action)
        else:
            action = np.argmax(self.Q_table[state])
        return action

    def q_learning(self, s0, a0, r, s1):
        td_error = r + self.gamma * self.Q_table[s1].max(
        ) - self.Q_table[s0, a0]
        self.Q_table[s0, a0] += self.alpha * td_error

    def update(self, s0, a0, r, s1):
        self.q_learning(s0, a0, r, s1)
        self.model[(s0, a0)] = r, s1  # 将数据添加到模型中
        for _ in range(self.n_planning):  # Q-planning循环
            # 随机选择曾经遇到过的状态动作对
            (s, a), (r, s_) = random.choice(list(self.model.items()))
            self.q_learning(s, a, r, s_)
```