时序差分算法

简介

无模型强化学习设定：

动态规划算法要求马尔可夫决策过程是已知的，但大部分场景并不现实，对于实际应用中MDP的**状态转移模型P**和**奖励模型R**都是未知的

我们只能根据智能体与环境交互的轨迹中发生的动作和得到的奖励采样到的数据进行学习，而对于这种MDP里P和R未知的情况，往往有两种思路来解决

一种是基于**基于模型**，而另一种是与**模型无关**

就拿魏老师视频中的例子——计算班上学生的平均年龄来说明如何解决模型未知怎么求解🦫

假设我们知道班级中学生的年龄概率分布模型，通过加权求期望就可以得到。但是现在并不知道年龄的概率模型分布，我们只能在学生们上体育课后在空无一人的教室门口等着，然后等他们体育课下课时来一个学生询问对应的年龄进行统计，即采样数据

在得到全班的年龄时，我们可以根据数据构建年龄分布的概率模型，然后再进行计算，而这就是**基于模型**的解决思路，另一种思路是要计算班上学生的平均年龄，那么直接统计的数据加和除以总数不就得到平均值了吗？🐬非要和模型过不去？而这种思想就是**模型无关**的求解（蒙特卡洛方法）

为什么一直对模型有着执念，是因为刚刚学习的动态规划算法就是基于已知的模型来进行不断迭代求解。所以刚刚也说了，我们仍然可以通过采样数据来建立模型，但当MDP的过程有上万亿个时还能进行友好的建模吗？🐬

所以面对现实吧，MDP的模型无法写出时，也就无法直接进行动态规划时，在这种情况下，智能体只能和环境进行交互，通过采样到的数据来学习，这类学习方法统称为**无模型的强化学习**

无模型强化学习算法：

无模型的强化学习算法不需要事先知道环境的奖励函数和状态转移函数，而是直接使用和环境交互的过程中采样到的数据来学习，在这一章介绍两种无模型的强化学习的经典算法：Sarsa和Q-learning，这两种算法都是基于**时序差分**的强化学习算法

在使用算法时智能体的学习分为**在线策略学习**和**离线策略学习**两种。通常来说，在线策略学习要求使用在当前策略下采样得到的样本进行学习，一旦策略被更新，当前的样本就被放弃了，就好像在水龙头下用自来水洗手；而离线策略学习使用经验回放池将之前采样得到的样本收集起来再次利用，就好像使用脸盆接水后洗手。因此，离线策略学习往往能够更好地利用历史数据，并具有更小的样本复杂度（算法达到收敛结果需要在环境中采样的样本数量），这使其被更广泛地应用。

还没开始学，不过书上已经引入了这个概念，mark一下🦔

时序差分

时序差分学习和蒙特卡洛方法差不多但又不太一样，相似之处在于可以从样本数据中学习，不需要事先知道环境；不同之处是时序差分算法可以直接从**经验片段中进行学习**，而蒙特卡洛方法必须要等整个序列采样结束之后才能计算得到累积回报，而时序差分算法只需要当前步结束就可以进行计算

蒙特卡洛方法的增量更新为：

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)]$$

而这里将G_t累积回报换做 $r_t+\gamma V(s_{t+1})$就可得时序差分的增量更新：

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)]$$

（这里α = 1/N(s)）

从更新增量中可以看出，时序差分并不需要计算累积回报G_t，而之所以能这样替换的原因是：

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k}|S_t = s] \\ &= \mathbb{E}_\pi[R_t + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_t + \gamma V_\pi(S_{t+1})|S_t = s] \end{aligned}$$

蒙特卡洛方法将上式第一行作为更新的目标，而时序差分算法将上式最后一行作为更新的目标，所以时序差分更新的是当前的预测值（$r_t+\gamma V(s_{t+1})$）使其去接近估计累积奖励（G_t），所以这里的累积奖励是当前奖励加上未来的猜测，而并非真实值

其中**$r_t + \gamma V(s_{t+1}) - V(s_t)$**通常被称为时序差分误差

MC与TD的优缺点：

时序差分：能够在知道最后结果之前进行学习

- 时序差分能够在每一步之后进行在线学习
- 蒙特卡洛必须等待片段结束，直到累计奖励已知

时序差分：能够无需最后结果地进行学习

- 时序差分能够从不完整的序列中学习
- 蒙特卡洛只能从完整的序列中学习
- 时序差分在连续（无终止的）环境下工作
- 蒙特卡洛只能在片段化的（有终止的）环境下工作

蒙特卡洛具有高方差，对于结果无偏差

- 良好的收敛性质
    - 使用函数近似时依然如此
- 对初始值不敏感
- 易于理解使用

时序差分具有低方差，有偏差

- 通常比蒙特卡洛更加高效
- 时序差分最终收敛到$V^\pi(S_t)$
    - 但使用函数近似并不总是如此
- 比蒙特卡洛对初始值更加敏感

Sarsa算法

原理：

sarsa算法是时序差分算法的具体应用，时序差分算法在蒙特卡洛更新价值函数的基础上将序列结束后的累积回报替换成了当前奖励加下一个状态的价值乘折扣因子来对价值函数更新迭代，所以时序差分算法对价值函数的求解可以概括为动规中策略迭代的策略评估（计算价值函数）

所以策略评估可以通过时序差分算法实现，那么在不知道奖励函数和状态转移函数的情况下如何进行策略提升呢？

策略评估的本质是计算状态价值函数，策略提升的概括的说通过最优状态价值函数得到最优的动作价值函数然后贪心地在每一个状态下选择相较于该状态的价值函数，动作价值最大的动作来作为新的策略。所以利用时序差分算法直接求出最优的动作价值函数，然后贪婪算法来选取在某个状态下动作价值最大的那个动作，得到新的策略

$$\pi'(s) = \arg\max_a Q^{\pi}(s,a)$$

所以如何利用时序差分算法来计算动作价值函数？对比动作价值函数的计算：

$$\begin{aligned}
Q(s,a) &= E[G_t \mid S_t=s, A_t=a] \\
&= E[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} \cdots \mid S_t=s, A_t=a] \\
&= E[R_t + \gamma(R_{t+1} + \gamma R_{t+2} \cdots) \mid S_t=s, A_t=a] \\
&= E[R_t + \gamma G_{t+1} \mid S_t=s, A_t=a] \\
&= E[R_t + \gamma Q(S_{t+1}, a_{t+1}) \mid S_t=s, A_t=a]
\end{aligned}$$

$$\therefore G_t \approx R_t + \gamma Q(S_{t+1}, a_{t+1})$$

所以可以利用累计回报的近似值得动作价值函数的时序差分算法更新迭代的公式：

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$$

对于上述公式中我们需要当前的状态s_t以及将要发生的动作a_t，然后当前状态s_t执行动作后得到的奖励r_t，然后更新迭代时需要（s_t，a_t）→ s_{t+1}，当前状态的下一个状态s_{t+1}以及它要执行的动作a_{t+1}

将这些符号拼接在一起就能得到算法名称——sarsa，这也就是Sarsa算法的由来

存在问题以及解决方案：

我们利用更新迭代后的最优动作价值函数来决定策略时，还需要注意以下几个问题：

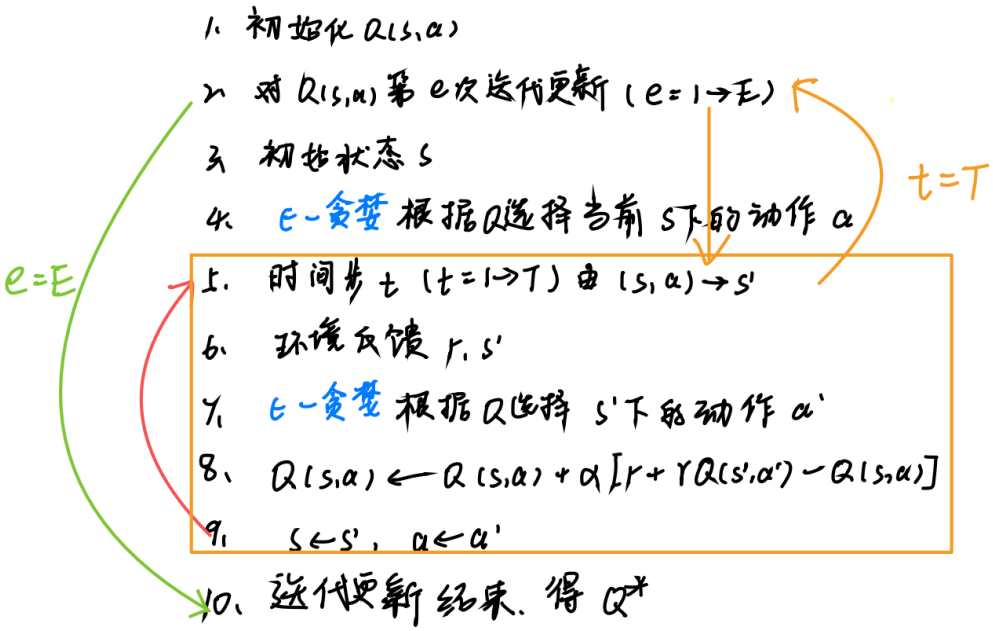
首先，时序差分算法在估计h策略的状态价值函数时，是需要极大量的样本序列片段来进行更新的，而这无疑增加了大量计算使得算法效率低下

所以对于这个问题我们不需要把那么的的样本来进行更新，而是选取其中的一些样本来评估策略然后更新策略。这样不用担心估计出的状态价值函数不准确，因为策略提升可以在策略评估未完成的情况下进行——类比价值迭代，这是一种**广义策略迭代**

（价值迭代：在策略评估进行一轮更新后就进行策略提升）

其次，如果在策略提升中一直根据贪婪算法得到一个确定性策略，可能会导致某些状态动作对(s,a)没有出现在决策上，所以就不能保证当前的认知是最优的，而针对这个问题在第二章中有详细的探讨，（ε—贪婪算法、上置信界算法 UCB、汤普森采样算法）所以我们可以利用ε—贪婪算法来解决这个问题，增加策略的多样性和可能性

算法流程：



迭代更新结束后得到的动作价值函数一般是收敛的，然后通过收敛的动作价值函数得到对应策略，在整个算法过程中没有用到状态转移函数以及奖励函数，就把较优的策略求解出来

实现：

```python
class Sarsa:
    """ Sarsa算法 """
    def __init__(self, ncol, nrow, epsilon, alpha, gamma, 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

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

```

```
    def best_action(self, state):  # 用于打印策略
        Q_max = np.max(self.Q_table[state])
        a = [0 for _ in range(self.n_action)]
        for i in range(self.n_action):  # 若两个动作的价值一样,都会记录下来
            if self.Q_table[state, i] == Q_max:
                a[i] = 1
        return a

    def update(self, s0, a0, r, s1, a1):
        td_error = r + self.gamma * self.Q_table[s1, a1] - self.Q_table[s0, a0]
        self.Q_table[s0, a0] += self.alpha * td_error
```

多步Sarsa算法:

时序差分算法（TD）与蒙特卡洛算法（MC）相比在值的估计上是有一定的偏差的，而MC是无偏估计，这是因为MC利用当前状态后的每一步的奖励而不使用估计值，TD则是将累计回报替换成了预测值而不是其真实的价值，所以是有偏的

但是相比MC它的方差是比较小的，这是因为MC的每一步状态转移具有不确定性，而每一步状态采取的动作所得到的不一样的奖励最终都会加起来，这会极大影响最终的价值估计；而TD只需一步状态转移，只关注下一个状态采取动作得到的奖励，所以方差是比较小的，稳定的

结合这两者的优势就是——**多步时序差分算法**

将累计回报：

$$G_t = r_t + \gamma Q(s_{t+1}, a_{t+1})$$

替换成：

$$G_t = r_t + \gamma r_{t+1} + \cdots + \gamma^n Q(s_{t+n}, a_{t+n})$$

此时动作状态价值函数的更新公式为：

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma r_{t+1} + \cdots + \gamma^n Q(s_{t+n}, a_{t+n}) - Q(s_t, a_t)]$$

Q-learning算法

介绍:

Q学习算法也是一种非常著名的基于时序差分算法的强化学习算法，它与sarsa算法的区别在于将时序差分的公式更新为：

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

即，该算法更新公式用的**Q(s',a')**中的**a'**不再是根据ε—贪婪算法选取的动作，而是对于给定任意的**（s,a,r,s'）**直接以最大的**Q(s',a')**中的**a'**代替

与Sarsa的区别：

这两个算法的区别在于评估策略时，Q-learining算法与Sarsa算法的区别在于策略评估，Sarsa算法采取的是ε—贪婪算法来选取动作，而Q-learining算法采取的是完全贪心，即更新时每次都选择最大的动作价值的动作，而不考虑其他动作价值函数，即每一步都是没有留余地的，策略必须精确的按照这个路径来走才能达到最后的函数值。而ε—贪婪算法则留有余地，以悬崖漫步为例，仍然有ε的概率走随机的其他路径（存在着掉下悬崖得到糟糕奖励💀的风险），经过长期的迭代最终也会收敛，但这种收敛相较于Q-learining算法更健壮

从策略的熵来看，ε—贪婪算法的熵比完全贪心的熵要更大。一般熵大的策略，健壮性更好，但最后的奖励可能略差（当然也不一定）

其次，Sarsa为在线策略（on-policy）而Q-learining算法为离线策略（off-policy）算法

在线策略与离线策略：

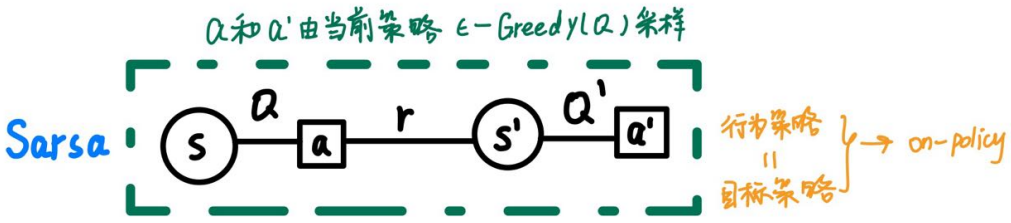
采样数据的策略为**行为策略**（behavior policy），用采样到的数据来更新的策略为**目标策略**（target policy）

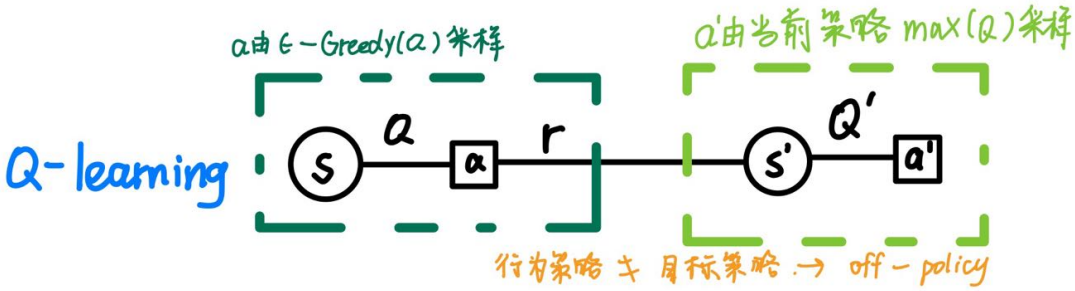
在线策略（on-policy）算法表示行为策略和目标策略是同一个策略

离线策略（off-policy）算法表示行为策略和目标策略不是同一个策略

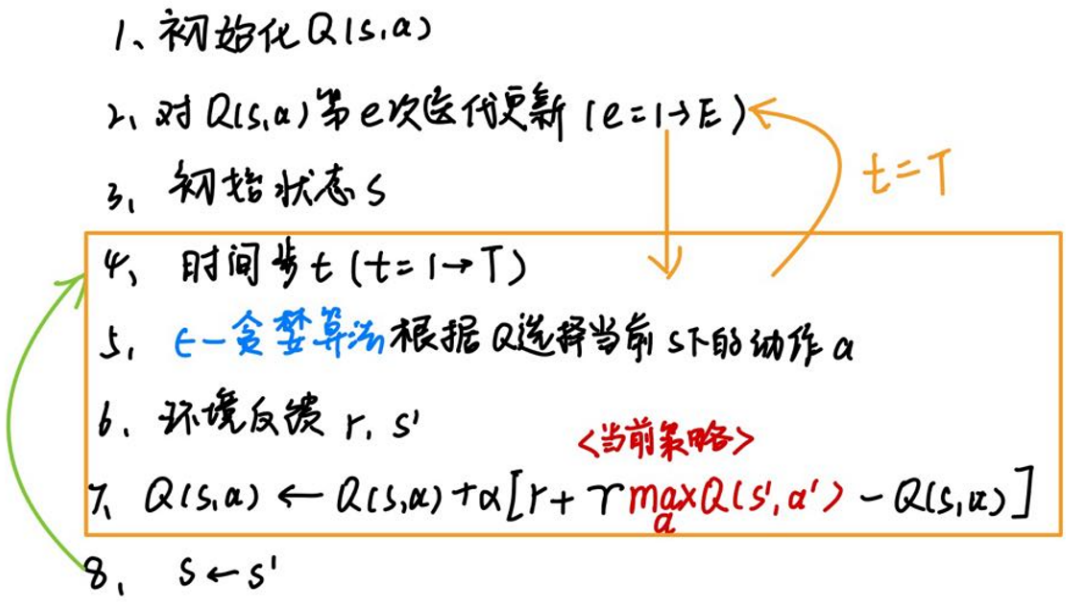
Sarsa 是典型的在线策略算法，而 Q-learning 是典型的离线策略算法。判断二者类别的一个重要手段是看**计算时序差分的价值目标的数据是否来自当前的策略**

- 对于 Sarsa更新公式必须使用来自当前策略采样得到的五元组（s,a,r,s',a'），因此它是在线策略学习方法
- 对于 Q-learning，它的更新公式使用的是四元组（s,a,r,s'）来更新当前状态动作对的价值Q(s,a)，数据中s的和a是给定的条件，r和s'皆由环境采样得到，该四元组并不一定是当前策略采样得到的数据，也可以来自行为策略得到的四元组，因此它是离线策略算法。





算法流程：



实现：

```python
class QLearning:
    """ Q-learning算法 """
    def __init__(self, ncol, nrow, epsilon, alpha, gamma, 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

    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 best_action(self, state):  # 用于打印策略
        Q_max = np.max(self.Q_table[state])
        a = [0 for _ in range(self.n_action)]
        for i in range(self.n_action):
            if self.Q_table[state, i] == Q_max:
                a[i] = 1
        return a

    def update(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
```

离线策略算法能够重复使用过往训练样本，往往具有更小的样本复杂度，也因此更受欢迎