Overview of Model-Based Reinforcement Learning

Chen Gong

18 October 2021

目录

1 Introduction to Model-based RL from Dynamics **1**

2 Shooting methods: RS, PETS, POPLIN **2**

2.1 What is the shooting method? 2

2.2 PETS: Probabilistic Ensembles with Trajectory Sampling 3

2.3 POPLIN: Model-based Policy Planning 3

3 Theoretical analysis **4**

3.1 $V^{\pi,M^*} \geq V^{\pi,\widehat{M}} - D(\widehat{M},\pi)$ 4

3.2 SLBO: Stochastic lower bound optimization 5

3.3 MBPO: Bound based on Model & Policy Error 6

4 Backpropagation through path: SVG and MAAC **7**

4.1 Deterministic Policy Gradient 7

4.2 From Deterministic to Stochastic 7

本文是 weinan zhang 老师在 RLCamp 上的关于 model-based reinforcement learning 的一个简单的笔记。基于 model based 的 RL 在机器人领域用的还挺多的，于是小编就看了一下相关的内容。有过一定 RL 基础同学都知道，RL 主要是通过环境的交互来进行试错学习，而环境主要是两部分组成：1. state dynamics: $p(s'|s,a)$，决定在当前状态 s 下采取动作 a，下一个状态 s' 的分布。2. reward function：$r(s,a)$，衡量在状态 s 做动作 a，是好还是坏。而 model based 的方法则是利用和环境交互的数据去模拟环境，得到 approximated state dynamics and reward function，$\hat{p}(s'|s,a)$，$\hat{r}(s,a)$。

1 Introduction to Model-based RL from Dynamics

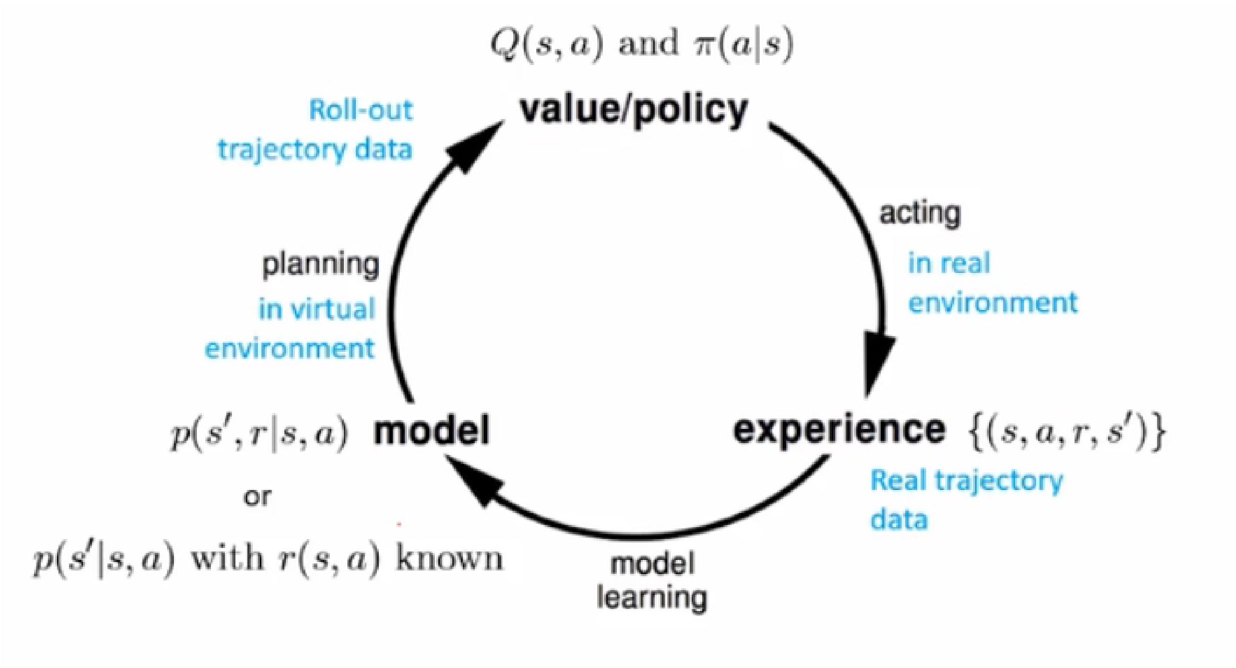

图 1: Model-Based 算法流程

Model-based 方法首先初始化一个价值函数或者策略函数 $Q(s,a)/\pi(a|s)$，用其与环境进行交互收集到经验 experience $\{(s,a,r,s')\}$，通过这些真实的 trajectory data 来进行模型学习得到 $\hat{p}(s',r|s,a)$。通过 simulated 环境来产生新的数据来更新 $Q(s,a)$ 或 $\pi(a|s)$。而 Dynamic Q 算法的思想也很简单，即为将 Model-Free 的方法和 Model based 的方法结合起来。不仅利用模拟出来的经验更新价值函数和策略函数，还利用和环境交互获得的真实经验来更新。两者达到相辅相成。

算法流程为：(a) 首先得到一个状态 S；(b) 然后利用价值函数得到一个将要执行的动作 $A \leftarrow \epsilon-\text{greedy}(S,Q)$；(c) 在环境中执行该动作 A，得到获得的奖励 R 和下一个时刻的状态 S'；(d) 利用 TD 更新价值函数：$Q(S,A) \leftarrow Q(S,A)+\gamma[R+\gamma\max_a Q(S',a)-Q(S,A)]$；(e) 学习 model：$Model(S,A) \leftarrow R,S'$；(f) 重复 n 次下面的操作，(首先，拿出一个之前见过的状态 S，和当时做的动作 A，利用模型来估计 $R,S' \leftarrow Model(S,A)$，并做一个 Q-Planning 操作：$Q(S,A) \leftarrow Q(S,A)+\gamma[R+\gamma\max_a Q(S',a)-Q(S,A)]$。)

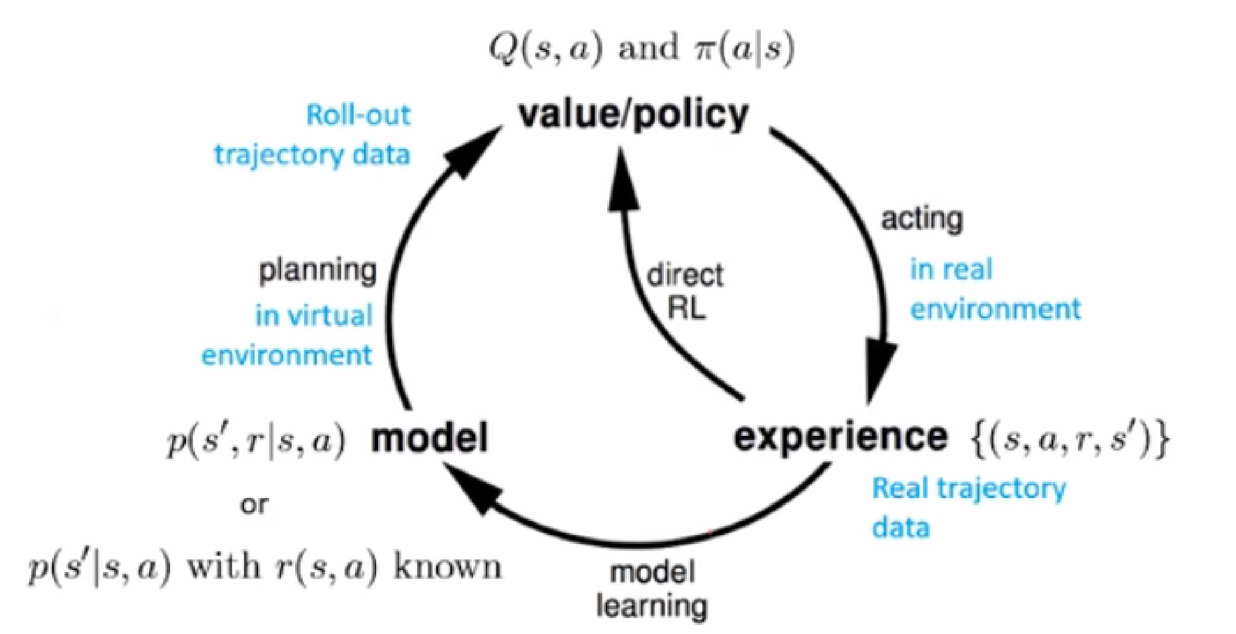

图 2: Dynamic Q 算法流程

2 Shooting methods: RS, PETS, POPLIN

2.1 What is the shooting method?

模型可以用来帮助 decision making。在 smart city 或者其他决策问题中用的比较多，比如我想知道扩宽一条道路对之后交通流的影响，这就可以通过 simulated model 来进行模拟。

那么说 **shooting method** 是什么意思呢？也就是从当前状态 s_0 出发，给定一个长度为 T 的动作序列：

$$[a_0, a_1, a_2, \cdots, a_T]$$

通过和 Model 进行交互，可以得到一条轨迹：

$$[s_0, a_0, \hat{r}_0, \hat{s}_1, a_1, \hat{r}_1, \hat{s}_2, a_2, \hat{r}_2, \cdots, \hat{s}_T, a_T, \hat{r}_T]$$

这样我们就可以看到执行这个动作序列的终局是什么。那么，我们求解这个问题就变成了选择一个动作序列来得到最高的估计 return：

$$\hat{Q}(s,a) = \sum_{t=0}^{T} \gamma^t \hat{r}_t, \qquad \pi(s) = \arg\max_a \hat{Q}(s,a).$$

在这里我有一点 confused 的是，这个给定的动作序列是怎么来的。而且比如在状态 s_0 下执行 $left$ 动作，和后面一系列动作获得的奖励最高，但是后面的动作序列换一下呢？是不是可能在状态 s_0 下执行另一个动作 $right$ 获得的累计奖励会比较高？这么做的好处就是，动作序列可以随机的得到，而且计算的时候也不需要梯度更新，也不要在乎轨迹的长度。但是缺点也很明显，就像我之前提到的那样，方差很大，这个方法得到的策略和初始给定的动作轨迹有关，我们并不一定可以找到最优解。**这就是 MPC，不会显式地学习一个 policy 或者 value function，只用 model 就可以完成决策**。

2.2 PETS: Probabilistic Ensembles with Trajectory Sampling

在强化学习中，与智能体交互的环境是一个动态系统，所以拟合它的环境模型也通常是一个动态模型。我们通常认为一个系统中有两种不确定性，分别是偶然不确定性（aleatoric uncertainty）和认知不确定性（epistemic uncertainty）。偶然不确定性是由系统中本身存在的随机性引起的，而认知不确定性是由“见”过的数据较少所导致的自身认知的不足而引起的。而在，PET 算法中，环境模型的构建会同时考虑这两种不确定性。首先，我们定义环境模型的输出为一个高斯分布，用来捕捉偶然不确定性。令环境模型为 $\hat{P}$，其参数为 θ，那么基于现在状态动作 (s_t, a_t)，下一个状态 s_{t+1} 的分布可以写为：

$$\hat{P}_\theta(s_{t+1}|s_t, a_t) = \mathcal{N}(\mu(s_t, a_t), \Sigma_\theta(s_t, a_t)) \tag{1}$$

这样我们就可以用神经网络来构建 μ_θ 和 Σ_θ，利用极大似然估计法，可以得到神经网络的损失函数为：

$$\mathcal{L}(\theta) = \sum_{n=1}^{N}[\mu_\theta(s_n, a_n) - s_{n+1}]^\top \Sigma_\theta^{-1}(s_n, a_n)[\mu_\theta(s_n, a_n) - s_{n+1}] + \log\det\sum_\theta(s_n, a_n) \tag{2}$$

这里我们就得到了一个由神经网络表示的环境模型。在此基础之上，我们选择用集成（ensemble）方法来捕捉认知不确定性。具体而言，我们构建多个网络框架一样的神经网络，它们的输入都是状态动作对，输出都是下一个状态的高斯分布的均值向量和协方差矩阵。但是它们的参数采用不同的随机初始化方式，并且当每次训练时，会从真实数据中随机采样不同的数据来训练。

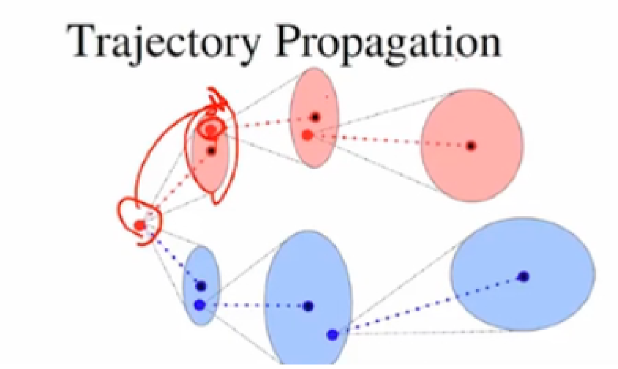

图 3: PET 算法 trajectory sampling

有了环境模型的集成后，MPC 算法会用其来预测奖励和下一个状态，从而获得整条采样轨迹。具体来说，每一次预测会从多个模型集成中挑选一个来进行预测，直到遇到终止状态，因此一条轨迹的采样会用到多个环境模型。

2.3 POPLIN: Model-based Policy Planning

2.2 小节中描述的 PET 算法有个很显著的问题，就是我们在生成动作序列的时候，是随机挑选的。这样肯定是不对的，自然而然的想法就是 maintain 一个 policy 来给定相应的 simulated state 来采样动作，来增加训练效率。

POPLIN 是 maintain 一个策略，来对给定的 simulated state 进行采样，

$$\mathcal{R}(s_i, a_i) = \mathbb{E}\left[\sum_{t=i}^{i+\tau} r(s_t, a_t)\right], \quad \text{where } s_{t+1} \sim f_\phi(s_{t+1}|s_t, a_t)$$

其中，有两种探索的方法（这个在 model free 的 RL 里面很常见）文章的实验表明后者的探索效果更好，

$$\mathcal{R}(s_i,\delta_i)=\mathbb{E}\left[\sum_{t=i} r(s_t,\hat{a}_t+\delta_t)\right],\qquad \mathcal{R}(s_i,\omega_i)=\mathbb{E}\left[\sum_{t=i} r(s_t,\pi_{\theta+\omega_t}(s_t))\right]$$

通过 MPC 以后，我们可以知道在一个状态 s，最优的动作是 $a*$，并得到 (s,a^*) 这么一个状态动作对。于是就可以把这个当成专家数据，利用模仿学习来更新策略了，

$$\min_\theta \mathbb{E}_{s,a\in\mathcal{D}}\|\pi_\theta(s)-a\|^2$$

我理解的就是 POPLIN 和 PET 的一个主要的区别就是在生成动作序列，做 planning 的这个过程，PET 是简单的随机分配，而 POPLIN 是学了一个策略，根据当前策略来生成。所以，我觉得 POPLIN 中是存在显示的策略的。

3 Theoretical analysis

3.1 $V^{\pi,M^*}\geq V^{\pi,\widehat{M}}-D(\widehat{M},\pi)$

其中，V^{π,M^*} 代表策略和真实环境 M^* 交互得到的价值，$V^{\pi,\widehat{M}}$ 代表策略和模拟环境 $\widehat{M}$ 交互得到的价值。而 $D(\widehat{M},\pi)$ 代表策略和真实环境 M^*，模拟环境 $\widehat{M}$ 交互得到的轨迹的 discrepancy。而且，以下上的等式成立需要满足三个条件：

- R_1: $V^{\pi,M^*}\geq V^{\pi,\widehat{M}}-D_{\pi_{\mathrm{ref}},\delta}(\widehat{M},\pi)$, $\forall\pi$ s.t.$d(\pi,\pi_{\mathrm{ref}})\leq\delta$。个人觉得这里的 π_{ref} 表示的是采样策略（可以理解为 behavior policy），这里要求的是 behavior policy 和 learning policy 之间的差距不能太大。这种约束主要是对 off-policy 的算法，在 on-policy 的算法中是 behavior policy 和 learning policy 是一样的。

- R_2: $\widehat{M}=M^*\Longrightarrow D_{\pi_{\mathrm{ref}}}(\widehat{M},\pi)=0,\forall\forall\pi,\pi_{\mathrm{ref}}$，张老师说这是一个强假设。我感觉这里是为了满足距离的定义。

- R_3: $D_{\pi_{\mathrm{ref}}}(\widehat{M},\pi)$ is of the form $\mathbb{E}_{\tau\sim\pi_{\mathrm{ref}},M^*}[f(\widehat{M},\pi,\tau)]$。这里实际是极大似然估计的形式，例如 $D_{\pi_{\mathrm{ref}}}(\widehat{M},\pi)=L\cdot\mathbb{E}_{S_0,\cdots,S_t\sim\pi_{\mathrm{ref}},M^*}[\|\widehat{M}(S_t)-S_{t+1}\|]$。这里描述的是利用采样策略 π_{ref} 和真实模型 M^* 交互得到的轨迹，来检测真实值与预测值之间的差距。其中，L 是李普西斯常数，定义为一段函数中最陡峭的位置的梯度。那么，新的问题来了，π 去哪了，,,,,,，这里小编看的有点懵逼。所以，小编觉得这里是不是写的不太对，应该是 $D_{\pi_{\mathrm{ref}}}(\widehat{M},\pi)=L\cdot\mathbb{E}_{S_0,\cdots,S_t\sim\pi_{\mathrm{ref}},M^*}[\|\widehat{M}(S_t,\pi(S_t))-S_{t+1}\|]$，hihihi。

于是，这里就可以导出，Meta-Algorithm for Model-based RL. MBRL 可以写为：

$$\pi_{k+1},M_{k+1}=\arg\max_{\pi\in\Pi,M\mathcal{M}} V^{\pi,M}-D_{\pi_k,\delta}(M,\pi),\quad \text{s.t. } d(\pi,\pi_k)\leq\delta \tag{3}$$

其中，采用的是 TRPO 的方法来对当前策略进行更新。然后，交替的更新策略 π 和环境模型 M。这里关于 $d(\pi,\pi_k)\leq\delta$ 的约束，小编有话说，对于这个约束的理解，小编也是学习过 offline rl 之后，理解了 distributional shift 问题以后才明白的。在 value evaluation 中，我们更新当前的策略 π 的数据是来自 behavior policy，当时利用这个数据来更新当前的策略是不对的，存在一个偏差，最直观的就是，

需要用到贝尔曼方程来对价值函数进行更新。而在计算贝尔曼残差时，需要从学习策略中采样下一个时刻的状态和动作，$Q(s_t, a_t) = r_t + \gamma \mathbb{E}_{a_{t+1} \sim \pi}[Q(s_{t+1}, a_{t+1})]$。显然，我们只能估计由行为策略生成的 (s, a) 的 Q 值。然而，如果学习策略生成的 (s, a) 与数据集代表的策略生成的 (s, a) 相差太大，则无法可靠地做出预测，会有很大的误差。所以，需要让 $d(\pi, \pi_k)$ 之间的距离不用太大。

接下来就是老套路了，证明策略提升定理：

Theorem 3.1. Suppose that $M^* \in \mathcal{M}$, that D and d satisfy equation, and the optimization problem in equation (3) is solvable at each iteration. Then we will have a sequence of policies $\pi_0, \cdots, \pi_T$ with monotonically increasing values:

$$V^{\pi_0, M^*} \leq V^{\pi_1, M^*} \leq \cdots \leq V^{\pi_T, M^*} \tag{4}$$

Moreover, as $k \to \infty$, the value $V^{V^{\pi_k, M^*}}$ converges to some V^{π, M^*}, where $\tilde{\pi}$ is a local maximum of V^{π, M^*} in domain Π.

这个证明也很简单，和 SAC 里面证明策略提升的方法差不多。因为

$$\pi_{k+1}, M_{k+1} = \arg \max_{\pi \in \Pi, M \in \mathcal{M}} V^{\pi, M} - D_{\pi_k, \delta}(M, \pi)$$
$$\text{s.t. } d(\pi, \pi_k) \leq \delta$$

根据 R_1 和 R_2 有，

$$\begin{aligned} V^{\pi_{k+1}, M^*} &\geq V^{\pi_{k+1}, M_{k+1}} - D_{\pi_k}(M_{k+1}, \pi_{k+1}) \\ &\geq V^{\pi_k, M^*} - D_{\pi_k}(M^*, \pi_k) \\ &= V^{\pi_k, M^*} \end{aligned}$$

但是这个理论分析有两个条件，**首先就是 argmax 可以取得到**，其实这个挺难做到的，理论分析中就先不考虑这个问题。第二个要求是**神经网络的表达能力要足够强**，M^* 要能用当前的网络表示。其实这两个条件还挺难达到的。

3.2 SLBO: Stochastic lower bound optimization

Algorithm 2 Stochastic Lower Bound Optimization (SLBO)

```
Initialize model network parameters φ and policy network parameters θ
Initialize dataset D ← ∅
for n_outer iterations do
    D ← D ∪ { collect n_collect samples from real environment using π_θ with noises }
    for n_inner iterations do                    ▷ optimize (6.2) with stochastic alternating updates
        for n_model iterations do
            optimize (6.1) over φ with sampled data from D by one step of Adam
        for n_policy iterations do
            D' ← { collect n_trpo samples using \hat{M}_φ as dynamics }
            optimize π_θ by running TRPO on D'
```

图 4: SLBO 算法伪代码

其实代码还是挺简单的，其中 model learning 的 loss 是：

$$\mathcal{L}_{\phi}^{(H)}((s_{t:t+h}, a_{t:t+h}); \phi) = \frac{1}{H} \sum_{i=1}^{H} \|(\hat{s}_{t+i} - \hat{s}_{t+i-1}) - (s_{t+i} - s_{t+i-1})\|_2. \tag{5}$$

然后，优化公式 (3) 的损失函数为：

$$\max_{\phi,\theta} V^{\pi_\theta, \mathrm{sg}(\widehat{M}_\phi)} - \lambda \mathbb{E}_{(s_{t:t+h}, a_{t:t+h}) \sim \pi_k, M^*} \left[\mathcal{L}_\phi^{(H)}((s_{t:t+h}, a_{t:t+h}); \phi) \right] \tag{6}$$

3.3 MBPO: Bound based on Model & Policy Error

这是 ucb 的 sergey 在 2019 NIPS 上提出来的算法，“When to Trust Your Model: Model Based Policy Optimization.” 首先文章中给了一个 bound，描述的是，真实的策略和我们用 model 学出来的策略之间存在一个 error。而这个 error 由 learning policy 和 behavior policy 之间的差距，和 learned model 和 TRUE model 之间的差距组成。

$$\eta[\pi] \geq \hat{\eta}[\pi] - \underbrace{\left[\frac{2\gamma r_{\max}(\epsilon_m + 2\epsilon_\pi)}{(1-\gamma)^2} + \frac{4 r_{\max} \epsilon_\pi}{(1-\gamma)} \right]}_{C(\epsilon_m, \epsilon_\pi)}$$

$$\begin{aligned} \epsilon_\pi &= \max_s D_{TV}(\pi \| \pi_D) \\ \epsilon_m &= \max_\ell \mathbb{E}_{s \sim \pi_{D,t}} \left[D_{TV}\left(p(s', r \mid s, a) \| p_\theta(s', r \mid s, a) \right) \right] \end{aligned}$$

其中，如果我们在使用 simulated model 时，不 rollout 到终止状态，而只向后 rollout k 个状态，于是真实的策略和我们用 model 学出来的策略之间存在一个 error 可以写为：

$$\eta[\pi] \geq \eta^{\text{branch}}[\pi] - 2 r_{\max} \left[\frac{\gamma^{k+1} \epsilon_\pi}{(1-\gamma)^2} + \frac{\gamma^k + 2}{(1-\gamma)} \epsilon_\pi + \frac{k}{1-\gamma} (\epsilon_m + 2\epsilon_\pi) \right]. \tag{7}$$

通过求导发现，当 $k = 0$ 的时候，这个 error 最小，也就是说最好是不使用 model，这无疑是非常悲观的。但是求这个 bound 有个非常重要的点，大家可以 ϵ_m 的表达式，我们需要从 behavior policy: π_D 上进行采样。但是，这里如果改变一下，不是利用 π_D 进行采样，而是采用当前的策略 π_t 来进行采样，计算 ϵ_m，我们可以得到，

$$\epsilon_{m'} = \max_t \mathbb{E}_{s \sim \pi_t} [D_{TV}(p(s', r | s, a) \| p_\theta(s', r | s, a))]$$

而这样之后，bound 可以被写为：

$$\eta[\pi] \geq \eta^{\text{branch}}[\pi] - 2 r_{\max} \left[\frac{\gamma^{k+1} \epsilon_\pi}{(1-\gamma)^2} + \frac{\gamma^k \epsilon_\pi}{(1-\gamma)} + \frac{k}{1-\gamma} (\epsilon_{m'}) \right] \tag{8}$$

我们仔细观测公式 (7) 和公式 (8) 可以发现，最大的不同在于这么一操作，$\frac{k}{1-\gamma}$ 只和 $\epsilon_{m'}$ 有关，前面两项随着 k 的增大误差在减小，后面一项随着 k 的增大，误差在增大。两者之间有希望找到一个平衡。所以，经过求导发现 $\frac{d\epsilon_{m'}}{d\epsilon_\pi}$ 足够的小，最优的 k 是大于 0 的。这也意味着，用 model 来 rollout k 步可以减小真实的策略和我们用 model 学出来的策略之间的 error。

于是基于这个理论发展了 MBPO 算法，其实此算法挺简单的。不过这里的用来更新模型的数据都是由当前策略采样出来的。但是，这里没有研究出怎么找到 optimal k，还是有机会做文章的。而且，MBPO 中是用的 SAC 来来更新的策略，SAC 算法的性能是好于 PPO 的。

Algorithm 2 Model-Based Policy Optimization with Deep Reinforcement Learning

Initialize policy π_ϕ, predictive model p_θ, environment dataset $\mathcal{D}_{\text{env}}$, model dataset $\mathcal{D}_{\text{model}}$
for N epochs **do**
  Train model p_θ on $\mathcal{D}_{\text{env}}$ via maximum likelihood
  for E steps **do**
    Take action in environment according to π_ϕ; add to $\mathcal{D}_{\text{env}}$
    for M model rollouts **do**
      Sample s_t uniformly from $\mathcal{D}_{\text{env}}$
      Perform k-step model rollout starting from s_t using policy π_ϕ; add to $\mathcal{D}_{\text{model}}$
    for G gradient updates **do**
      Update policy parameters on model data: $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi, \mathcal{D}_{\text{model}})$

图 5: MBPO 算法伪代码

4 Backpropagation through path: SVG and MAAC

这些算法目前还比较的新。首先简单对 DPG 和 DDPG 做一个回顾。

4.1 Deterministic Policy Gradient

在 DPG 中，假设都是 determinate 的策略，$a = \pi_\theta(s)$。那么根据确定性策略梯度理论，我们需要最大化：

$$J(\pi_\theta) = \mathbb{E}_{s\sim\rho^\pi}[Q^\pi(s,a)] \tag{9}$$

我们相对 π 的参数 θ 求导，而 Q 函数里面，动作的生成是和 π 相关的。所以，我们可以先对 Q 函数的动作求导，在利用 chain rule 对 θ 求导，即为，

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim\rho^\pi}\left[\nabla_\theta \pi_\theta(s) \nabla_a Q^\pi(s,a)\big|_{a=\pi_\theta(s)}\right] \tag{10}$$

4.2 From Deterministic to Stochastic

首先我们假设动作是 $a = \pi(s;\theta)$，转移函数为 $s' = f(s,a)$，尽管这看起来像是一个确定性函数，但是我们可以通过对输入进行扰动来让其变成一个随机的输出。然后假设，下标代表对下标变量求导，比如，$g_x = \partial g(x,y)/\partial x$。那么，贝尔曼方程可以表示为：

$$V(s) = r(s,a) + \gamma V'(f(s,a)) \tag{11}$$

为了清晰一点表示变量直接的关系，我将上述公式 specifically 表示一下，

$$V(s) = r(s, a = \pi(s;\theta)) + \gamma V'(f(s, a = \pi(s;\theta))) \tag{12}$$

那么，如果对 s 求导，就有

$$V_s = r_s + r_a \cdot \pi_s + \gamma V'_{s'}(f_s + f_a \cdot \pi_s) \tag{13}$$

这里都是简单的数学推导，利用导数的基本求导法则，于是对 θ 求导就有，

$$V_\theta = r_a \cdots \pi_\theta + \gamma V'_{s'} \cdot f_a \cdot \pi_\theta + \gamma V'_\theta \tag{14}$$

其中最后一项 V'_θ，是对 V 函数自己进行求导。为了实现对概率密度函数求导需要使用到重参数技巧，

$$
\begin{aligned}
p(y \mid x) &= \mathcal{N}\left(y \mid \mu(x), \sigma^2(x)\right) \\
y &= \mu(x) + \sigma(x)\xi, \text{ where } \xi \sim \mathcal{N}(0,1)
\end{aligned}
$$

其中，μ 和 σ 函数都是完全 deterministic 的。于是，我们可以把上述的公式 (13)-(14) 转成 stochastic 的版本。

$$
\begin{aligned}
\mathbf{a} = \pi(\mathbf{s}, \eta; \theta) \quad \mathbf{s}' &= \mathbf{f}(\mathbf{s}, \mathbf{a}, \xi) \quad \eta \sim \rho(\eta) \text{ and } \xi \sim \rho(\xi) \\
V(\mathbf{s}) &= \mathbb{E}_{\rho(\eta)}\left[r(\mathbf{s}, \pi(\mathbf{s}, \eta; \theta)) + \gamma \mathbb{E}_{\rho(\xi)}\left[V'(f(\mathbf{s}, \pi(\mathbf{s}, \eta; \theta), \xi))\right]\right] \\
V_{\mathbf{s}} &= \mathbb{E}_{\rho(\eta)}\left[r_{\mathbf{s}} + r_{\mathbf{a}}\pi_{\mathbf{s}} + \gamma \mathbb{E}_{\rho(\xi)} V'_{\mathbf{s}'}\left(\mathbf{f}_{\mathbf{s}} + \mathbf{f}_{\mathbf{a}}\pi_{\mathbf{s}}\right)\right], \\
V_\theta &= \mathbb{E}_{\rho(\eta)}\left[r_{\mathbf{a}}\pi_\theta + \gamma \mathbb{E}_{\rho(\xi)}\left[V'_{\mathbf{s}'}\mathbf{f}_{\mathbf{a}}\pi_\theta + V'_\theta\right]\right].
\end{aligned}
$$

基于上述分析就有了 SVG 算法。

Algorithm 1 SVG(∞)

```
Given empty experience database 𝒟
for trajectory = 0 to ∞ do
    for t = 0 to T do
        Apply control a = π(s, η; θ), η ~ ρ(η)
        Insert (s, a, r, s') into 𝒟
    end for
    Train generative model f̂ using 𝒟
    v'_s = 0 (finite-horizon)
    v'_θ = 0 (finite-horizon)
    for t = T down to 0 do
        Infer ξ|(s, a, s') and η|(s, a)
        v_θ = [r_a π_θ + γ(v'_{s'} f̂_a π_θ + v'_θ)]|_{η,ξ}
        v_s = [r_s + r_a π_s + γ v'_{s'}(f̂_s + f̂_a π_s)]|_{η,ξ}
    end for
    Apply gradient-based update using v^0_θ
end for
```

图 6: SVG 算法伪代码

他的目标就是令 $V(s, \pi_\theta(s))$ 尽可能的大，于是直接强行对这个函数的参数 θ 进行求导。