Chapter 5 Model-Based Reinforcement Learning

5.1 Introduction to Model-Based reinforcement learning

What we have covered so far can be categorized as "model-free" reinforcement learning. This is because in previous discussions the transition probabilities are unknown and we did not even attempt to learn the transition probabilities. In reinforcement learning, we have the objective of maximizing the expectation of reward along the trajectory given as follows:

$$\pi_\theta(\tau) = p_\theta\left(\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T\right) = p\left(\mathbf{s}_1\right) \prod_{t=1}^{T} \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right) p\left(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t\right) \tag{5.1.1}$$

$$\theta^\star = \arg\max_\theta E_{\tau \sim p_\theta(\tau)} \left[\sum_t r\left(\mathbf{s}_t, \mathbf{a}_t\right) \right] \tag{5.1.2}$$

The transition probabilities $p\left(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t\right)$ is not known in all the model-free RL algorithms that we have learned such as Q-learning and policy gradients. But what if we know the transition dynamics? Recall that at the very beginning of the notes we drew an analogy of RL and control theory. In many cases, we do know the system's internal transition. For example, in games (e.g., Atari games, chess, Go), easily modeled systems (e.g., navigating a car), and simulated environments (e.g., simulated robots, video games), the transitions are given to us.

Moreover, it is not uncommon to learn the transition models: in classic robotics, system identification fits unknown parameters of a known model to learn how the system evolves, and one could also imagine a deep learning approach where we could potentially fit a general-purpose model to observed transition data for later use.

It holds true that knowing the transition dynamics does make things easier, and this is what we are going to deal with. In model-based reinforcement learning, we are going to learn the transition dynamics, and then figure out how to choose actions. In this section we will mainly focus on how can we make decisions if we *know* the dynamics, specifically the following two topics:

1. How can we choose actions under perfect knowledge of the system dynamics?
2. Optimal control, trajectory optimization, planning.

So in this chapter we assume the transition model is **known** or **approximated** to further help planning.

5.2 Optimal Control and Planning

5.2.1 Optimal Control

Optimal control is a task that we come across when we are well aware of the transition probabilities and we try to learn how to control the system optimally. In optimal control, there are two different categories of controller design: the first one is open-loop control, where we do not have any state feedbacks, and we roll out a sequence of actions based on the current state that we observe. The second one is called closed-loop control, where we determine the action at each time step based on the current state, and how we determine the action to apply is based on state feedbacks. In terms of transition model, there are also two categories depending on whether the

model is deterministic or stochastic. While in reinforcement learning, we generally deal with stochastic transition models and close loop control.

In a open loop controller, if we have a deterministic transition in our system such that $\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t)$, then our action sequence should be determined by choosing those that can return the maximum rewards:

$$\mathbf{a}_1, \dots, \mathbf{a}_T = \arg\max_{\mathbf{a}_1,\dots,\mathbf{a}_T} \sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t) \quad \text{s.t. } \mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t) \tag{5.2.1}$$

In stochastic scenarios, the transition function is a probabilistic distribution, where we have $p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)$, and the action sequence should be chosen based on expectation of the rewards:

$$\begin{gathered} p_\theta(\mathbf{s}_1, \dots, \mathbf{s}_T \mid \mathbf{a}_1, \dots, \mathbf{a}_T) = p(\mathbf{s}_1) \prod_{t=1}^{T} p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t) \\ \mathbf{a}_1, \dots, \mathbf{a}_T = \arg\max_{\mathbf{a}_1,\dots,\mathbf{a}_T} E\left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \mid \mathbf{a}_1, \dots, \mathbf{a}_T\right] \end{gathered} \tag{5.2.2}$$

Note that this policy might be suboptimal, since future states may reveal more information helpful to decision making. While in closed-loop stochastic cases we roll out all actions to apply only based on the initial state marginal and we do not consider any state-feedback.

In a closed-loop controller, however, we keep interacting with the world, so we need a policy function that can tell us the action to apply if we input the current state: $\pi(\mathbf{a}_t \mid \mathbf{s}_t)$, which we call a state-feedback. We choose our policy function as follows:

$$\begin{gathered} p(\mathbf{s}_1, \mathbf{a}_1, \dots, \mathbf{s}_T, \mathbf{a}_T) = p(\mathbf{s}_1) \prod_{t=1}^{T} \pi(\mathbf{a}_t \mid \mathbf{s}_t)\, p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t) \\ \pi = \arg\max_\pi E_{\tau \sim p(\tau)}\left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)\right] \end{gathered} \tag{5.2.3}$$

Note that the form of policy π may vary, including nerual nets, or simply takes a time-varying linear form: $\mathbf{K}_t \mathbf{s}_t + \mathbf{k}_t$.

5.2.2 Open-Loop Planning

We'll start from open-loop planning first. It has the minimal assumption about the dynamics: it does not care about whether the transition model is continuous or discrete, deterministic or stochastic, or whether it is differentiable.

Recall the objective of stochastic open-loop planning, we roll out a sequence of actions by doing arg max on the sum of rewards:

$$\mathbf{a}_1, \dots, \mathbf{a}_T = \arg\max_{\mathbf{a}_1,\dots,\mathbf{a}_T} \sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t) \text{ s.t. } \mathbf{a}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t) \tag{5.2.4}$$

We can abstract away the control and planning part:

$$\mathbf{a}_1, \dots, \mathbf{a}_T = \arg\max_{\mathbf{a}_1,\dots,\mathbf{a}_T} J(\mathbf{a}_1, \dots, \mathbf{a}_T) \tag{5.2.5}$$

Or compactly, since we do not care about whether the action is sequential or discrete, we can say:

$$\mathbf{A} = \arg\max_{\mathbf{A}} J(\mathbf{A}) \tag{5.2.6}$$

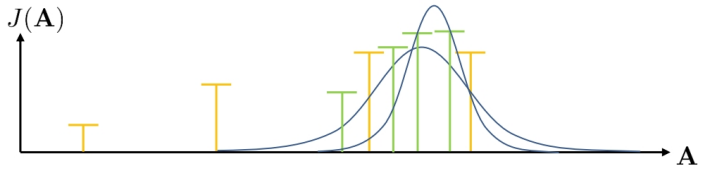
The simplest method is guess and check, or you can call it random shooting method:

1. pick $\mathbf{A}_1, \dots, \mathbf{A}_N$ from some distribution (e.g., uniform)
2. choose $\mathbf{A}_i$ based on $\arg\max_i J(\mathbf{A}_i)$

Random shooting method has the advantage that it is super simple to implement, though it could be highly inefficient in that we are not improving what we sample, so we might get stuck in some mediocre action sequence.

Cross-entropy method

We can dramatically improve this random shooting method by using cross-entropy method (CEM) Instead of randomly selecting samples, in CEM we first randomly initialize a set of samples and then evaluate the objective function for each sample. Next, we use these objective function values to update the sampling distribution, so that the samples are more likely to fall in regions with higher objective function values. This image illustrates the process of iteratively sampling from regions with higher objective function values.



Algorithm 21 Cross-entropy method

1: **repeat**
2: sample $\mathbf{A}_1, \ldots, \mathbf{A}_N$ from $p(\mathbf{A})$
3: evaluate $J(\mathbf{A}_1), \ldots, J(\mathbf{A}_N)$
4: pick the elites $\mathbf{A}_{i_1}, \ldots, \mathbf{A}_{i_M}$ with the highest value, where $M < N$
5: refit $p(\mathbf{A})$ to the elites $\mathbf{A}_{i_1}, \ldots, \mathbf{A}_{i_M}$
6: **until** Satisfactory result

When we "fit" a distribution, what we are actually doing is to iteratively search for the best parameters to find a distribution from which we can sample to give us the best cost-to-go value. Cross-entropy method is also very fast if parallelized, and also extremely simple to implement. However it has a harsh limit on dimensionality. And it only works for open-loop scenarios.

Monte Carlo tree search (MCTS)

In discrete scenarios, we use another method called Monte Carlo tree search, which is very popular in planning in stochastic games.

The gist of this method is that in discrete action space, we are essentially expanding out a tree, but we can not fullly explore the tree due to the exponential computational cost. One way to save the computational cost is to partially expand the tree (for example explore to a fixed depth in each branch) and then use a given policy (e.g., random distribution) to simulate a trajectory from the last expanded node.

Then the question would be: how do we decide on where to search first?

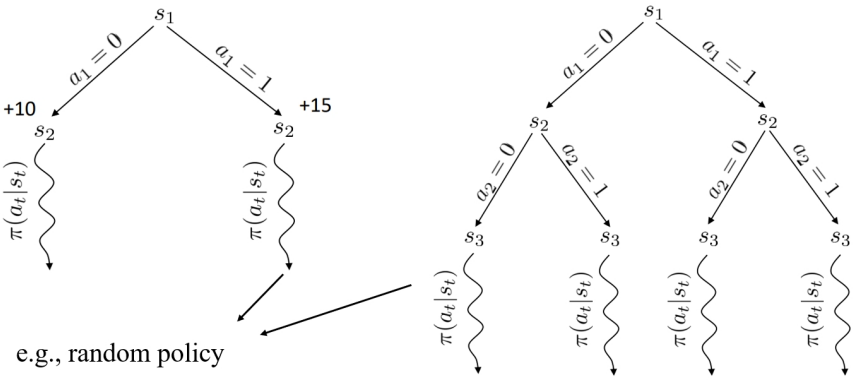


Figure 5.1: Monte Carlo tree search

Suppose we have two actions to take, $a = 0$ and $a = 1$. After taking either action, the agent follows a fixed policy to complete the episode, and the rewards obtained are 10 and 15, respectively.Note that the values of 10 and 15 are not the true values of the rewards, since this is a stochastic process and we have only executed it once.

Our intuition tells us that we should further exploring the action that gave us a higher reward, which is $a = 1$. Besides, if there is a third action that we have not yet explored, then we should also explore this unknown path. That is to say, we should choose nodes with best reward, but also prefer rarely visited nodes. This intuition actually gives the skectch of generic MCTS:

Algorithm 22 Generic MCTS algorithm

1: **repeat**
2: find a leaf s_l using TreePolicy (s_1)
3: evaluate the leaf using DefaultPolicy (s_l)
4: update all values in tree between s_1 and s_l
5: refit $p(\mathbf{A})$ to the elites $\mathbf{A}_{i_1}, \ldots, \mathbf{A}_{i_M}$
6: **until** take best action from s_1

A frequently used TreePolicy is called UCT TreePolicy: if s_t is not fully expanded, choose the remaining new action a_t; else, choose child branch with best score $S(s_{t+1})$, where the score is calculated by:

$$\text{Score}\left(s_t\right) = \frac{Q\left(s_t\right)}{N\left(s_t\right)} + 2C\sqrt{\frac{2\ln N\left(s_{t-1}\right)}{N\left(s_t\right)}} \tag{5.2.7}$$

Here Q for reward, and N for total steps of a chosen trajectory.

5.2.3 Trajectory Optimization with Derivatives

We are going to use the set of notation more commonly used in control therory for the following discussions. (using $\mathbf{u}_t$ for action and $\mathbf{x}_t$ for state, minimizing cost instead of maximizing reward)

$$\min_{\mathbf{u}_1,\ldots,\mathbf{u}_T} \sum_{t=1}^{T} c\left(\mathbf{x}_t, \mathbf{u}_t\right) \text{ s.t. } \mathbf{x}_t = f\left(\mathbf{x}_{t-1}, \mathbf{u}_{t-1}\right) \tag{5.2.8}$$

if we plug in the transition constraint, we have:

$$\min_{\mathbf{u}_1,\ldots,\mathbf{u}_T} c\left(\mathbf{x}_1, \mathbf{u}_1\right) + c\left(f\left(\mathbf{x}_1, \mathbf{u}_1\right), \mathbf{u}_2\right) + \cdots + c\left(f(f(\ldots)\ldots), \mathbf{u}_T\right) \tag{5.2.9}$$

5.3 Model-Based Reinforcement Learning

5.3.1 Basics

In this section, we are going to cover a rather simpler case of model-based RL. Specifically, we are going to talk about a technique to learn a model of the system first, and then use the optimal control technique we covered to improve the model. Furthermore, we will learn to address uncertainty in the model such as model mismatch and imperfection.

Why do we learn the model? We can learn the model so that we know $f(s_t, a_t) = s_{t+1}$ (in deterministic case) or $p(s_{t+1}|s_t, a_t)$ (in stochastic case), we could use the tools from optimal control to maximize our rewards.

Algorithm 23 Model-based Reinforcement Learning Version 0.5

Require: Some base policy for data collection π_0
1: Run base policy $\pi(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$ (supervised learning)
2: Learn dynamics model $f(s, a)$ to minimize $\sum_i ||f(s_i, a_i) - s_i'||^2$
3: Plan through $f(s, a)$ to choose actions

Our first attempt is naive: learn $f(s_t, a_t)$ from data, and then plan through it. We call this approach model-based RL version 0.5, or vanilla model-based RL, as shown in Alg. 23. This is essentially what people do in system identification, which is a technique used in classic robotics, and it is effective when we can hand-engineer a dynamics representation using our knowledge of physics, and fit just a few parameters. However, it does not work in general cases because of distribution mismatch, i.e., $p_{\pi_0}(s_t) \neq p_{\pi_f}(s_t)$. Furthermore, the distribution mismatch exacerbates as we use more expressive model classes (e.g., neural networks), since more complicated models would fit more tightly to the demonstration data (which might be inaccurate).

How to make $p_{\pi_0}(s_t) = p_{\pi_f}(s_t)$? We need to collect data from $p_{\pi_f}(s_t)$. We can do this by keep updating the dataset by running the current model, and then update the model accordingly. Take a look at the updated model-based RL algorithm in Alg. 24.

Algorithm 24 Model-based Reinforcement Learning Version 1.0

Require: Some base policy for data collection π_0
1: Run base policy $\pi(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
2: **while** True **do**
3: Learn dynamics model $f(s, a)$ to minimize $\sum_i ||f(s_i, a_i) - s'_i||^2$
4: Plan through $f(s, a)$ to choose actions
5: Execute those actions and add the resulting data $\{(s, a, s')_j\}$ to $\mathcal{D}$

Version 1.0 addresses the model mismatch issue and drives the current model as close as possible to the true dynamics model. However, we are still blindly following a trajectory in step 5 of Alg. 24, and if we made a mistake, we would follow the wrong step which makes the mistake exacerbate. Therefore, we need to somehow adjust our plan as time goes on. One way to do this is to borrow some ideas from modern control theory: Model Predictive Control (MPC).

In MPC, we are given the system's dynamics model, and we are trying to design an adaptive controller by solving a finite time constrained optimal control problem at each time step, and take only the first action in the generated sequence of actions. Then we replan based on the new state. We essentially aims to take one action in the planned sequence and only observe one new state, and then append the observed transition to our dataset $\mathcal{D}$. The improvement is shown in Alg. 25.

Algorithm 25 Model-based Reinforcement Learning Version 1.5

Require: Some base policy for data collection π_0, hyperparameter N
1: Run base policy $\pi(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
2: **while** True **do**
3: Learn dynamics model $f(s, a)$ to minimize $\sum_i ||f(s_i, a_i) - s'_i||^2$
4: **for** N times **do**
5: Plan through $f(s, a)$ to choose actions
6: Execute the first planned action, observe resulting state s' (MPC)
7: Append (s, a, s') to dataset $\mathcal{D}$

The inner loop in Alg. 25 refers to replanning in MPC, which is solving for an optimization problem at each time step after we take the first action planned. The outer loop means that we are periodically retraining the model in order to make it closer to the true underlying dynamics. Intuitively, the more frequently the agent replans, the less perfect each individual plan needs to be, because since we are frequently replanning, we are able to correct our mistakes made in previous plans more easily. Consequently, one is able to correct the plans as one increases the replanning frequency. Therefore, if we are frequently replanning, we could use shorter horizons.

5.3.2 Performance Gaps in Model-based RL

Sometimes model-based RL performs worse than model-free RL. The problem is from step 5 of Alg. 25. In this step, we plan through the model to choose actions, which means we are solving an optimization problem based on the data we collect. One could imagine that if we overfit the data, the agent might have some wrong belief about the model, thus generating wrong actions, as is illustrated in Fig. 5.2.

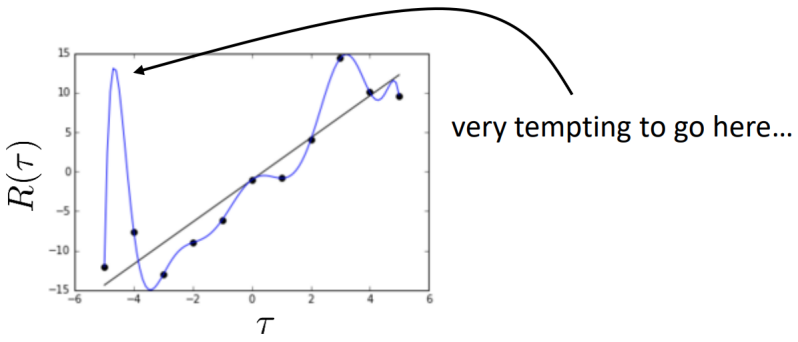


Figure 5.2: False belief about the model from overfitting

Will model-based learning run into the overfitting problem? The answer is yes, because the model does not see the true data distribution. In step 5 of Alg. 25, when we choose actions, we only take actions for which we think we will get high reward **in expectation** (with respect to uncertain dynamics). In the model-based setting, the model will quickly adapts to the distribution of these actions and converge, which avoids the model from exploring enough and results in bad actions planned.

Therefore, we need to explore to get more representative and holistic data of the model, thus preventing overfitting and false belief. Note that the expected value of the reward is not the same as optimistic or pessimistic, but it is often a good start.

5.3.3 Uncertainty-Aware Models

One way to deal with the problem of bad actions planned is to construct an uncertainty-aware model.

The uncertainty-aware model provides a way to quantitatively estimate the uncertainty in the model, so that we can assess the accuracy of the model and the planned actions. When we take uncertainty into account, the agent will be less likely to take actions that are highly uncertain, and will instead prefer actions that are both highly certain and have high reward. In the context of neural networks, this means that the agent will be less likely to take actions that are the result of overfitting, even if those actions have high predicted reward, because they are highly uncertain.

Remark. Why can uncertainty-aware models help alleviate the above problem?

Add more details

The first idea is to use entropy of output distribution, and as we know, higher entropy means higher uncertainty. We can estimate the entropy of $p(s_{t+1}|s_t, a_t)$. However, this is not enough because even when the model is wrong (e.g., an over-fitted model), we might still have low variance, thus low entropy, as long as the data points all satisfy the over-fitted model.

The reason why entropy of the output distribution alone is not expressive enough is that there are two types of uncertainty:

- Aleatoric (statistical) uncertainty, where *the data itself is noisy* .
- Epistemic (model) uncertainty, where *the model is certain about data, but we are not certain about model.*

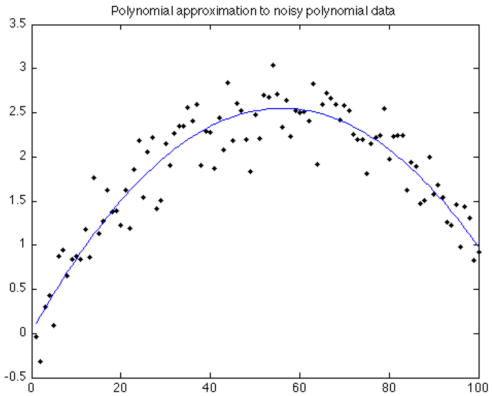


Figure 5.3: Aleatoric uncertainty

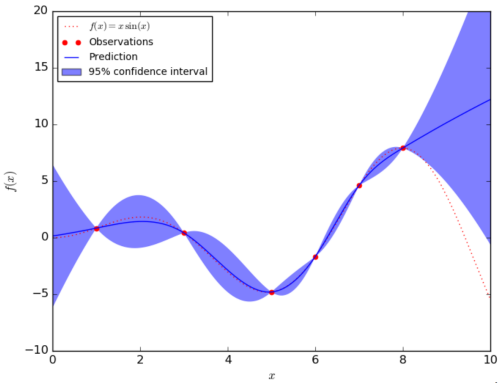


Figure 5.4: Epistemic uncertainty

The second idea is to estimate the epistemic (model) uncertainty, where we essentially estimate how uncertainty we are about the model.

Usually, we use maximum likelihood estimation, where

$$\arg\max_{\theta} \log p(\theta|\mathcal{D}) = \arg\max_{\theta} \log p(\mathcal{D}|\theta) \tag{5.3.1}$$

This is because when we are searching for the model parameters θ that are "most likely" there are actually two ways to interpret "most likely" in this context: (1) Given the model parameters θ, the probability of generating

the dataset $\mathcal{D}$ is maximized. (2) Given the dataset $\mathcal{D}$, the probability that this particular dataset was generated by the model with parameters θ is maximized. And these two interpretations essentially gives you the same parameters when performing arg max.

If we estimate the full distribution of data $p(\theta|\mathcal{D})$ instead of argmax, the entropy of the distribution gives us the model uncertainty from the data. In practice we can predict using

$$\int p(s_{t+1}|s_t, a_t, \theta)p(\theta|\mathcal{D})d\theta. \tag{5.3.2}$$

Bayesian Neural Networks

Bayesian Neural Networks

Bootstrap Ensembles

To learn the posterior distribution, we can also train bootstrap ensembles, where we use multiple networks to learn the same distribution. Formally, say we have N networks, each with a parameter θ_i to learn $p(s_{t+1}|s_t, a_t)$, we can then estimate the posterior by:

$$p(\theta|\mathcal{D}) \sim \frac{1}{N}\sum_i \delta(\theta_i) \tag{5.3.3}$$

where $\delta(\cdot)$ is the Dirac-delta function. To train it, we need to generate independent datasets to get independent models. One way to do this is to chop the original dataset into multiple subsets. Another way is to train θ_i on $\mathcal{D}_i$ sampled with replacement from $\mathcal{D}$. In reality, resampling with replacement is usually not necessary, because SGD and random initialization usually makes the model sufficiently independent.

With this ensemble of networks, we choose actions a little differently. Before, we choose actions by $J(a_1, \ldots, a_H) = \sum_{t=1}^{H} r(s_t, a_t)$, where $s_{t+1} = f(s_t, a_t)$, and now we average over the ensemble by $J(a_1, \ldots, a_H) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{H} r(s_{t,i}, a_{t,i})$, where $s_{t+1,i} = f(s_{t,i}, a_{t,i})$

In general, for candidate action sequence $a_1, \ldots, a_H$, we first sample $\theta \sim p(\theta|\mathcal{D})$, then at each time step t, we sample $s_{t+1} \sim p(s_{t+1}|s_t, a_t, \theta)$, then we calculate the reward $R = \sum_t r(s_t, a_t)$, and we repeat the aforementioned steps and accumulate the average reward.

5.3.4 Latent Space Model

In many cases, we are given very complex observations of states which we do not have full access to, such as pixel-based images. To learn the dynamics using observations, we need to learn from the latent space and infer the states from observations.

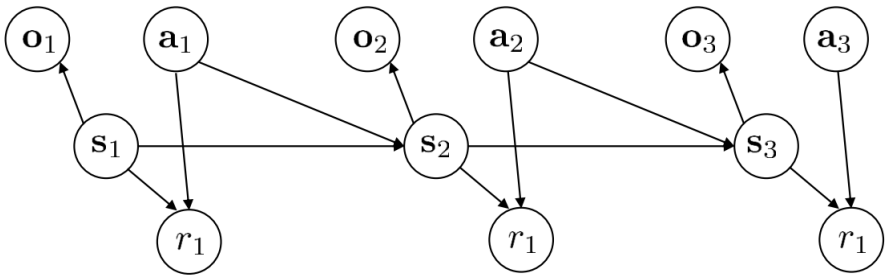


Figure 5.5: Latent space model

From Fig. 5.5, we can see that we need to learn the following models:

- $p(o_t|s_t)$, the observation model
- $p(s_{t+1}|s_t, a_t)$, the dynamics model
- $p(r_t|s_t, a_t)$, the reward model

Recall that in high level, model-based RL algorithms are basically doing a maximum likelihood estimation in training given fully observed states:

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log p_{\phi}(s_{t+1,i}|s_{t,i}, a_{t,i}) \tag{5.3.4}$$

With latent models, then, we are not sure about the actual state, so we take the expected value:

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \mathbb{E}\left[\log p_{\phi}(s_{t+1,i}|s_{t,i}, a_{t,i}) + \log p_{\phi}(o_{t,i}|s_{t,i})\right] \tag{5.3.5}$$

where the expectation is with respect to the distribution of $(s_t, s_{t+1}) \sim p(s_t, s_{t+1}|o_{1:T}, a_{1:T})$.

However, the posterior distribution $p(s_t, s_{t+1}|o_{1:T}, a_{1:T})$ is usually intractable if we have very complex dynamics. As a result, we could instead try to learn an approximate posterior $q_{\psi}(s_t|o_{1:t}, a_{1;t})$, which is called an **encoder**. We could also learn $q_{\psi}(s_t, s_{t+1}|o_{1:t}, a_{1;t})$ and $q_{\psi}(s_t|o_t)$. Learning the distribution $q_{\psi}(s_t|o_t)$ is crude, but it is simple to implement. Let us focus on $q_{\psi}(s_t|o_t)$ for now, and then the expectation becomes:

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \mathbb{E}\left[\log p_{\phi}(s_{t+1,i}|s_{t,i}, a_{t,i}) + \log p_{\phi}(o_{t,i}|s_{t,i})\right]$$

where the expectation is with respect to $s_t \sim q_{\psi}(s_t|o_t)$, $s_{t+1} \sim q_{\psi}(s_{t+1}|o_{t+1})$.

For now, let us further assume that $q(s_t|o_t)$ is deterministic (because the stochastic case requires variational inference, which will be covered in-depth in a later chapter). In deterministic case, we are training a neural net $g_{\psi}(o_t) = s_t$ using a Dirac-delta function such that $q_{\psi}(s_t|o_t) = \delta(s_t = g_{\psi}(o_t))$. Then the expectation can be simplified as

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log p_{\phi}(g_{\psi}(o_{t+1,i})|g_{\psi}(o_{t,i}), a_{t,i}) + \log p_{\phi}(o_{t,i}|g_{\psi}(o_{t,i})). \tag{5.3.6}$$

Now everything is differentiable, we can train using backpropagation.

Thus, we can slightly modify Alg. 25 so that we can deal with observations and latent space. We show the sketch of this slightly modified algorithm in Alg. 26. In line 3, we are respectively learning the dynamics, reward model, observation model, and encoder.

Algorithm 26 Model-based Reinforcement Learning with Latent States

Require: Some base policy for data collection π_0, hyperparameter N
1: Run base policy $\pi(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
2: **while** True **do**
3: Learn dynamics model $p_{\phi}(s_{t+1}|s_t, a_t), p_{\phi}(r_t|s_t), p(o_t|s_t), g_{\psi}(o_t)$
4: **for** N times **do**
5: Plan through $f(s, a)$ to choose actions
6: Execute the first planned action, observe resulting state o' (MPC)
7: Append (o, a, o') to dataset $\mathcal{D}$

5.3.5 Further Reading

For more details on model-based RL, refer to Nagabandi et al. (2018), Chua et al. (2018), Feinberg et al. (2018) and Buckman et al. (2018).

For more details for latent space models, refer to Watter et al. (2015) and Zhang et al. (2019).

5.4 Model-Based Policy Learning

So far we have covered the basics of model-based RL that we first learn a model and use a model for control. We have seen that this approach does not work well in general because of the effect of distributional shift in model-based RL. We have also seen the method to quantify uncertainty in our model in order to alleviate this

issue. The methods we covered so far do not involve learning policies. In this chapter, we will cover model-based reinforcement learning of policies. Specifically, we will learn global policies and local policies, and combine local policies into global policies using guided policy search and policy distillation. We shall understand how and why we should use models to learn policies, global and local policy learning, and how local policies can be merged via supervised learning into a global policy.

We have seen the difference between a closed-loop and open-loop controller. We also discussed why an open-loop controller is suboptimal because we are rolling out a whole sequence of actions solely based on one state observation. Therefore, it would be more ideal if we could design a closed-loop controller where state feedbacks can help us correct the mistakes we make. Recall in a stochastic environment, we are optimizing over the policy as follows:

$$p(s_1, a_1, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

$$\pi = \arg\max_{\pi} \mathbb{E}_{\tau \sim p(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

and π could take several forms: π can be a neural net, which we call a **global** policy, and it can also be a time-varying linear controller $K_t s_t + k_t$ as we saw in LQR, which we call a **local** policy.

5.5 Back-propagate into the Policy

Let us start with a simple solution for model-based policy learning. Ideally, we could build a computational graph in Tensorflow, and calculate the partial derivatives step by step so that we can backpropage into policy and optimize the policy, illustrated in Fig. 5.6.

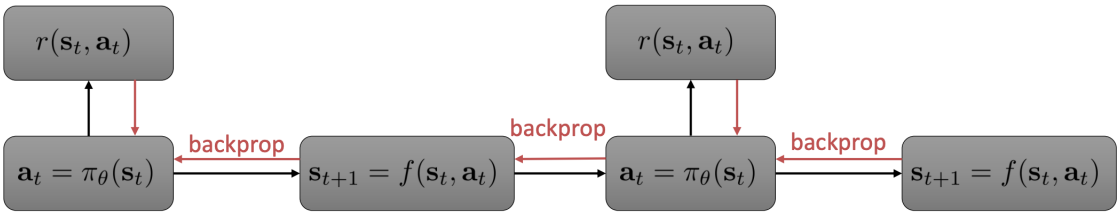


Figure 5.6: Back-propagate into policies

Then we can modify our model-based policy-free RL algorithm to accomodate this new policy learning process in Alg. 27.

Algorithm 27 Model-based Reinforcement Learning Version 1.5

Require: Some base policy for data collection π_0

1: Run base policy $\pi_0(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
2: **while** True **do**
3: Learn dynamics model $f(s, a)$ to minimize $\sum_i ||f(s_i, a_i) - s'_i||^2$
4: Backpropagate through $f(s, a)$ into the policy to optimize $\pi_\theta(a_t|s_t)$
5: Run $\pi_\theta(a_t|s_t)$, appending the visited tuples (s, a, s') to $\mathcal{D}$.

5.5.1 Vanishing and Exploding Gradients

One problem with Alg. 27, or general gradient-based optimization is that as we progress into the time steps, we might encounter vanishing or exploding gradients. Because as we apply chain rule, the gradients get multiplied by each other, so we the product may get extremely big (exploding) or extremely small (vanishing), making optimization a lot harder. Furthermore, we have similar parameter sensitivity problems as shooting methods, but we no longer have convenient second order LQR-like method, because the policy function is extremely complicated and policy parameters couple all the time steps, so no dynamic programming.

So what can we do about it? First, we can use model-free RL algorithms with synthetic samples generated by the model. Essentially, we are using models to accelerate model-free RL. Second, we can use simpler policies than neural nets such as LQR, and train local policies to solve simple tasks, and then combine them into global policies via supervised learning.

5.6 Model-free Optimization with a Model

Model-free Optimization with a Model is yet to be written.

Recall the equation from policy gradients:

$$\nabla_\theta J(\theta) \simeq \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\hat{Q}^\pi_{i,t}$$

Note that we are not doing any backprop through time in policy gradient because we are calculating the gradient with respect to an expectation, so we can just take the derivative of the probability of the samples instead of the actual dynamics function.

Then we look at the regular backprop (pathwise) gradient, we see a more chain rule-like gradient:

$$\nabla_\theta J(\theta) = \sum_{t=1}^{T}\frac{dr_t}{ds_t}\prod_{t'=2}^{t}\frac{ds_{t'}}{da_{t'-1}}\frac{da_{t'-1}}{ds_{t'-1}}$$

The two gradients are different, because the policy gradient is for stochastic systems while the backprop policy is for deterministic systems. But using variational inference, we can prove that they are calculating the same gradient differently, thus having different tradeoffs. We will talk about variational inference more in-depth in the next chapter.

Actually, given more samples to reduce variance, policy gradient is more stable because it does not require multiplying many Jacobians. However, if our models are inaccurate, the samples we use from the wrong model will be incorrect, and the mistakes are likely to exacerbate as time goes on. So it would be nice to use such model-free optimizer and keep the rolled out samples' trajectory short. This is essentially what Dyna algorithm does.

5.6.1 Dyna

Dyna is an online Q-learning algorithm that performs model-free RL with a model.

Algorithm 28 Dyna

Require: Some exploration policy for data collection π_0

1: Given state s, pick action a using exploration policy
2: Observe s' and r, to get transition (s, a, s', r)
3: Update model $\hat{p}(s'|s, a)$ and $\hat{r}(s, a)$ using (s, a, s')
4: Q-update: $Q(s, a) \leftarrow Q(s, a) + \alpha\mathbb{E}_{s',r}\left[r + \max_{a'} Q(s', a') - Q(s, a)\right]$
5: **for** K times **do**
6: Sample $(s, a) \sim \mathcal{B}$ from buffer of past states and actions
7: Q-update: $Q(s, a) \leftarrow Q(s, a) + \alpha\mathbb{E}_{s',r}\left[r + \max_{a'} Q(s', a') - Q(s, a)\right]$

In step 3 of Alg. 28, we are updating the model and reward function using the observed transition. Then in step 6, we will sample some old state and action pairs and apply the model onto the sampled pair, so the s' in step 7 are synthetic next states. Intuitively, as the models get better, the expectation estimate in step 7 also gets more accurate. This algorithm seems arbitrary in many aspects, but the gist is to keep improving models and use models to improve Q-function estimation by taking expectations.

We can also generalize Dyna to see how this kind of general Dyna-style model-based RL algorithms work. The generalized algorithm is shown in Alg. 29.

As shown in Fig. 5.7, we choose some states (orange dots) from the buffer, simulate the next states using the learned model, and then train model-free RL with synthetic data (s, a, s', r) where s is from the experience

Algorithm 29 General Dyna

Require: Some exploration policy for data collection π_0
1: Collect some data, consisting of transitions (s, a, s', r)
2: Learn model $\hat{p}(s'|s, a)$ (and optionally, $\hat{r}(s, a)$)
3: **for** K times **do**
4: Sample $s \sim \mathcal{B}$ from buffer
5: Choose action a (from $\mathcal{B}$, from π, or random)
6: Simulate $s' \sim \hat{p}(s'|s, a)$ (and $r = \hat{r}(s, a)$)
7: Train on (s, a, s', r) with model-free RL
8: (optional) take N more model-based steps

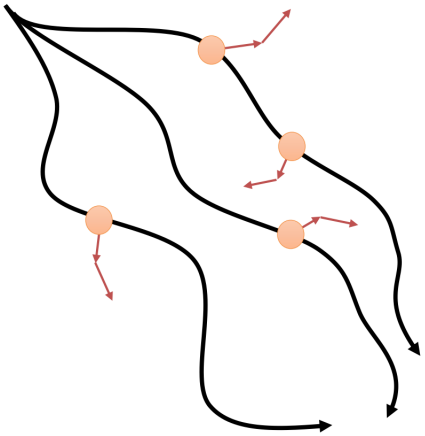


Figure 5.7: General Dyna training

buffer, s' is from the learned model. One could also take more than one step if one believes that the model is good enough for more steps.

This algorithm only requires very short (as few as one step) rollouts from model, so the mistakes will not exacerbate and accumulate much. Moreover, we explore well with a lot of samples because we still see diverse states.

5.7 Local and Global Models

Recall that in LQR, we can turn a constrianed optimization problem into an unconstrained problem:

$$\min_{u_1,\dots,u_T} c(x_1, u_1) + c(f(x_1, u_1), u_2) + \cdots + c(f(f(\dots)\dots), u_T)$$

Backpropagation is indeed a possible solution to solve this optimization problem, and we need $\frac{df}{dx_t}$, $\frac{df}{du_t}$, $\frac{dc}{dx_t}$, $\frac{dc}{du_t}$

5.7.1 Local Models

Since LQR gives us a state-feedback controller for a linear system, we can keep linearizing the system and iteratively apply LQR to generate local models. We fit $\frac{df}{dx_t}$, $\frac{df}{du_t}$ around the current trajectory or policy. Say the model is a Gaussian $p(x_{t+1}|x_t, u_t) = \mathcal{N}(f(x_t, u_t), \Sigma)$, then we can approximate the model as a linear function $f(x_t, u_t) \simeq A_t x_t + B_t u_t$, and we can use $\frac{df}{dx_t}$ as A_t, and $\frac{df}{du_t}$ as B_t.

Iterative LQR produces $\hat{x}_t, \hat{u}_t, K_t, k_t$, where $u_t = K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t$. We can execute the controller using a Gaussian $p(u_t|x_t) = \mathcal{N}(K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t, \Sigma_t)$ because we can add noise to the iLQR controller so that all samples do not look the same. Practically, we can set $\Sigma_t = Q_{u_t,u_t}^{-1}$. We can fit the model $p(s_{t+1}|s_t, a_t)$ using Bayesian linear regression, and use the global model as prior.

We also need to stay close to old controller if we go too far. If trajectory distribution is close, then dynamics will be close too. Close here means the KL-divergence is small $D_{KL}(p(\tau)||p(\bar{\tau})) \leq \epsilon$.

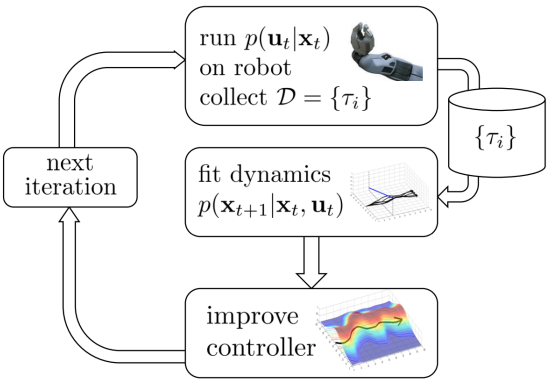


Figure 5.8: Local models fitting

Algorithm 30 Guided Policy Search

1: **while** True **do**
2: Optimize each local policy $\pi_{LQR,i}(u_t|x_t)$ on initial state $x_{0,i}$ with respect to $\tilde{c}_{k,i}(x_t, u_t)$
3: Use samples from the previous step to train $\pi_\theta(u_t|x_t)$ to mimic each $\pi_{LQR,i}(u_t|x_t)$
4: Update cost function $\tilde{c}_{k+1,i}(x_t, u_t) = c(x_t, u_t) + \lambda_{k+1} \log \pi_\theta(u_t|x_t)$

5.7.2 Guided Policy Search

The high level idea of guided policy search is to use some simpler local policy such as local LQR controller to help and guide the learning process of more complex global policy learner. Essentially, we would use the local models trajectories as the training data for a supervised learning neural net that can solve all the tasks.

However, one problem is that the local policies might not be able to be reproduced using a single neural net. Therefore, after training the global policy with supervised learning, we need to reoptimize the local policies using the global policy so that the policies are consistent with each other. The sketch of guided policy search is shown in Alg. 30. Note that the cost function $\tilde{c}_{k,i}$ is the modified cost function to keep π_{LQR} close to π_θ.

In Divide and Conquer RL, the idea is similar, except that we are replacing the local LQR controllers with local neural net.

5.7.3 Distillation

In RL, we borrow some ideas from supervised learning to achieve the task of learning a global policy from a bunch of local policies.

Recall in supervised learning, we use model ensemble to make our predictions more robust and accurate. However, keeping a lot of models is expensive during test time. Is there a way to train just one model that can behave as well as a meta-learner?

The idea, proposed by Hinton in **?**, is to train a model on the ensemble's predictions as "soft" targets using:

$$p_i = \frac{\exp(z_i/T)}{\sum_j (z_j/T)}$$

where T is called temperature. The new labels here can be intuitively explained using the example of MNIST dataset. For example, a handwritten digit "2" looks like a 2 and a backward 6. Therefore, the soft-labels that we use to train the distilled model is going to be "80% chance being 2 and 20% chance being 6".

In RL, to achieve multi-task global policy learning, we can use something similar called policy distillation. The idea is to train a global policy using a bunch of local tasks:

$$\mathcal{L} = \sum_a \pi_{E_i}(a|s) \log \pi_{AMN}(a|s)$$

where the meta-policy π_{AMN} can be trained in a supervised learning fashion.