# Model-Based RL Lecture I Notes (Draft Version)

March 26, 2024

## 1 Outline

- What is model-based RL? Contrasting with earlier approaches discussed in class.
- Pros and cons of using models in RL.
- Considerations when choosing and training models.
- Prediction with models.
- Planning with models.
- Putting it all together: PETS
- Summary: Model choice, uncertainty estimation, planning, and open- vs. closed-loop control.

## 2 What is Model-Based RL?

Remember that a key component of an MDP is the transition dynamics $p(s' \mid s, a)$, which captures how the world evolves due to an RL agent's actions. So far in this course, we have focused on methods that only require *samples* from the true underlying dynamics. For example, the objective for policy gradient-based approaches only requires samples of entire trajectories together with reward information:

$$\theta_{t+1} \leftarrow \theta_t + \eta \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[ R(\tau) \nabla_\theta \log p_\theta(\tau) \right] \approx \theta_t + \eta \sum_{t=0}^{T} R_\tau \nabla_\theta \log \pi(a_t | s_t)$$

As another example, the DQN objective learns the optimal $Q$-function only using samples of $(s, a, r, s')$ pairs:

$$\min_\theta \sum_{i=1}^{N} \left\{ Q_\theta(s_i, a_i) - \left[ r_i + \gamma \max_{a'} Q_{\text{targ}}(s', a') \right] \right\}^2$$

These *model-free* methods ignore the fact that there is a lot of "structure" in the dynamics independent of the reward function. Said more concretely, there is a lot of information in the sequence $(s_0, a_0, s_1, a_1, \ldots, s_T)$ that these methods ignore.

Today's lecture focuses on a whole new class of RL methods collectively referred to as *model-based RL*. A *model* here refers to a learned estimate $\hat{p}(s' \mid s, a)$ of the true underlying transition dynamics $p(s' \mid s, a)$[1]. This learned model is referred to as a *transition/dynamics model*. By the end of this lecture, you should get a sense of:

[1] Sometimes, the underlying reward function as well. We assume we know the reward function.

- Why should I use model-based RL? What are the drawbacks of using model-based RL?
- What kinds of models can I use in model-based RL?
- How do I use models to solve the RL problem?

# 3 Motivating Model-Based RL

Why should we care about model learning in the first place? Here are some advantages of learning with models:

**Models are reward function agnostic.** In many cases, we might want an agent to solve multiple unrelated tasks in the same environment. We can model this situation as solving several MDPs that only differ in the underlying reward function. The same learned model can be used to solve all of these MDPs.

**Models can make more effective use of the data.** With a well-chosen family of candidate models, a learned dynamics model could generalize well beyond the data collected by the behavior policy so far. Thus, even if the learner is in areas of the state space it has never seen before, the model could still allow the agent to make reasonable return-maximizing decisions.

Unfortunately, there is no free lunch, and there are definite drawbacks to using models:

**Models and compounding errors.** Given data from a single timestep $(s, a, s')$, dynamics models are typically trained to regress $s'$ against $(s, a)$. However, at planning time, we typically need to have the model regress on its previous predictions to obtain a full predicted trajectory. This means that model-based planning procedures are subject to compounding errors, which increase in severity with longer planning horizons. This is in fact a big limiting aspect of model-based RL, and we typically see that model-based RL algorithms perform worse than their model-free counterparts in terms of asymptotic performance. Addressing long-horizon prediction is a longstanding problem that guides the design of many algorithms, as we will see both during today's and next week's lecture.

# 4 How to Train your Model

In this section, we will discuss how models are chosen and learned, as well as important considerations when dealing with the drawbacks of long-horizon planning.

## 4.1 Model choice

Modeling dynamics effectively boils down to predicting the next state given the current state and action, so why not just use standard regression techniques? While this is intuitively correct, this does not give us a distribution over the next state, like the true transition dynamics would. The following example illustrates how we can use standard regression approaches to obtain a predictive distribution.

**Example 4.1** ($L_2$ loss with neural networks)**.** Let us say we want to fit a particular neural network architecture to dynamics data. Let $\{\mu_\theta(s, a)\}$ represent the set of functions that are representable by the architecture, indexed by the parameter $\theta$. In practice, neural networks are trained to perform regression

through the use of the $L_2$ prediction loss. How can we take a neural network prediction (which is a point estimate) and obtain a predicted distribution instead over $s'$?

The idea is that we *implicitly identify* each neural network $\mu_\theta$ with the conditional distribution $y \mid x \sim \mathcal{N}(\mu_\theta(s,a), \sigma^2)$, where $\sigma^2$ is an arbitrary constant that is fixed for the entire class. One can then show that performing maximum likelihood estimation in this context is equivalent to finding a $\theta$ that solves

$$\min_\theta \frac{1}{n} \sum_{i=1}^{n} \|s_i' - \mu_\theta(s_i, a_i)\|_2^2 .$$

That is, when (1) we take the neural network model class, (2) choose a constant variance noise model, and finally, (3) use maximum-likelihood estimation, we actually obtain the $L_2$ loss we are familiar with! ⌟

What the above example shows is that standard regression approaches have an implicit modeling assumption about the noise. Since we want to carefully model the environment, we should be explicit about all these choices. In particular, when we choose a model, we have to choose three components:

1. Model architecture. This includes neural networks, as well as other models you may have already seen like linear models, decision trees, kernel SVR, and so on.

2. Noise model. We can think of this as the mapping from the output of the regression model to a distribution. In the example above, this is the constant-variance noise model.

3. Loss function. An appropriate choice usually takes into account the previous two components. A common choice and one which we will focus on is the maximum likelihood loss.

To illustrate this model selection process, let us think about potential drawbacks with using a constant variance noise model in the prior example. In many cases, it might not make sense to have constant variance additive noise. There may be parts of the state space where the dynamics are inherently more unpredictable than others. In such cases, we can introduce an input-dependent (i.e. *heteroskedastic*) noise model, as the following example describes:

**Example 4.2** (Neural networks with heteroskedastic noise)**.** Instead of fixing the variance to a constant, we could have just as easily defined the model class to be $\{\mathcal{N}(\mu_\theta(s,a), \Sigma_\theta(s,a))\}$, where $\Sigma_\theta(s,a)$ denotes the $(s,a)$-dependent noise variance. MLE with the full Gaussian likelihood then gives the following optimization problem:

$$\min_\theta \frac{1}{n} \sum_{i=1}^{n} \left[ (s_i' - \mu_\theta(s_i,a_i)) \Sigma_\theta(s_i,a_i)^{-1} (s_i' - \mu_\theta(s_i,a_i)) + \log\det \Sigma_\theta(s,a) \right]. \tag{1}$$

In practice, this class is implemented by a neural network where one part of the output is used to compute the mean function $\mu_\theta$, and the other part is used to compute the covariance $\Sigma_\theta$. The covariance $\Sigma_\theta$ is usually restricted to be diagonal, because (a) ensuring $\Sigma_\theta$ is positive-definite is easier, and (b) computing the log-determinant and the inverse is easier in (1). With all of these considerations, the three components of the model are then:

| Component | Choice |
|---|---|
| Architecture | NN mapping $(s,a) \mapsto (\mu_\theta(s,a), \Sigma_\theta(s,a))$ |
| Noise Model | $(\mu_\theta(s,a), \Sigma_\theta(s,a)) \mapsto \mathcal{N}(\mu_\theta(s,a), \Sigma_\theta(s,a))$ |
| Loss | Maximum Likelihood |

Table 1: Components of an NN-based heteroskedastic noise model

⌟

## 4.2 Model Errors, Uncertainty Estimation, and Bootstrapping

A key problem with model-based RL is that we are relying on the model being a good approximation of reality. However, since we are always working with a finite amount of data, there will always be finite-sample errors. Can our models tell us how unsure they are about the prediction? What does being "uncertain" about the prediction mean in the first place?

**Remark 4.1** (A common misconception)**.** We might say that by modeling the noise explicitly, the model is already giving us a notion of uncertainty. But this is not the kind of uncertainty we are looking for! Noise from the modeling step represents unpredictable noise inherent in the environment. However, what we are looking for here is the uncertainty from the learning process *due to not having enough data*. Quantifying this uncertainty is useful because it might tell us where we should gather more data, or taking a different perspective, where we cannot trust model predictions. ⌟

### 4.2.1 Bias-Variance Decomposition

To mathematically ground our discussion of model errors, we will rely on the *bias-variance* decomposition, which you may have already seen in a prior machine learning class. The setting is as follows:

- We have a randomly sampled training dataset $\mathcal{D}$.
- Some training procedure is used to obtain a predictor $\hat{f}_{\mathcal{D}}$.
- We want to evaluate the error on a fixed test input $x$ whose label is generated as $f(x) + \varepsilon_x$ (the noise distribution depends on $x$).

The bias-variance decomposition then tells us the error we should expect for the training procedure on a typical sampled dataset $\mathcal{D}$:

$$\mathbb{E}_{\mathcal{D},\varepsilon_x}\left[(y - \hat{f}(x))^2\right] = \underbrace{\left(f(x) - \mathbb{E}_{\mathcal{D}}\left[\hat{f}(x)\right]\right)^2}_{\text{bias}} + \underbrace{\text{Var}_{\mathcal{D}}\left[\hat{f}(x)\right]}_{\text{variance}} + \underbrace{\mathbb{E}\left[\varepsilon_x^2\right]}_{\text{irreducible error}}$$

What do these terms represent?

**Bias.** The bias represents the error resulting from a misspecified model class that does not capture the ground truth. We control for this by using expressive function approximators (e.g. neural networks).

**Variance.** The variance represents how much our prediction varies if we were to have gotten a different training dataset. With a larger dataset, variance tends to 0.

**Irreducible Error.** The irreducible error results from inherent noise in the sampling procedure, or in our case, inherent noise in the environment.

Out of all these three terms, the kind of "uncertainty" we are interested in lies in the second term, as it measures how different our predictions could have been with a different draw of the training set. Intuitively, our "uncertainty" about the correct model should tend to 0 as we get more data, and indeed, the variance term in the bias-variance decomposition decreases with a larger dataset.

#### 4.2.2 Uncertainty Estimation via Bootstrapping and Model Ensembling

We now understand that our uncertainty in the quality of the model is captured by the variance in the classical bias-variance decomposition. But, how do we actually measure this variance in practice? Looking at the definition, it seems like we need multiple independent dataset draws, which seems infeasible and detracts from the data-efficiency gains from model-based RL.

The key technique here is to make use of a classical technique known as *bootstrapping*. Given a dataset $\mathcal{D}$ of size $N$, we sample $m$ bootstrapped datasets $\mathcal{D}_1, \ldots, \mathcal{D}_m$ of size $N$ uniformly with replacement from $\mathcal{D}$. We then train an *ensemble* of $m$ models $\hat{p}_1, \ldots, \hat{p}_m$. Since we expect $\mathcal{D}$ to be a "representative" of the true sampling distribution, these $m$ bootstrap datasets can be thought of as approximating sampling $m$ independent dataset samples from the original distribution.

Bootstrapping and ensembled model training are key techniques for improving the stability and performance for model-based RL algorithms.

# 5 How to Use Models: Trajectory Prediction and Model Predictive Control

In this section, we will outline how we use models in RL algorithms. We first discuss the problem of trajectory prediction, and then use a trajectory prediction subroutine to create control policies.

## 5.1 Trajectory Prediction

Now that we have a model, what do we want to actually do with it? Ultimately, one can think of many RL algorithms as modeling the outcome of applying a particular policy to an environment. For example, Q-functions capture the return from rollout out a particular policy. Since we have a dynamics model, we can go beyond predicting returns: we can predict the resulting trajectory itself.

Let us consider the problem of predicting the resulting trajectory from applying an action sequence $(a_0, a_1, \ldots, a_{T-1})$ from a starting state $s_0$. Assume that we have an ensemble of models $\hat{p}_1, \ldots, \hat{p}_m$ trained using bootstrapping. Our prediction method needs to incorporate two aspects of the model ensemble:

- $\hat{p}_1, \ldots, \hat{p}_m$ all model the inherent environment noise.
- Each individual model represents a different "belief" of the true dynamics (since they are trained on different datasets).

The idea we will take is known as *particle-based sampling*. We initialize some predetermined number of particles $P$ at $s_0$. For each particle, we choose a member of the ensemble uniformly at random, and sample an entire trajectory from this model after applying $(a_0, \ldots, a_{T-1})$. We then repeat this procedure for each of the particles[2]. Note that using one model per particle ensures that every particle is fully consistent with one possible model of the dynamics. Sampling more particles allows us to (1) better predict the effect of dynamics stochasticity, and (2) incorporate finite-sample uncertainty from model learning. We detail this trajectory prediction approach below:

[2]Efficient implementations batch the trajectory prediction process, so that all particles are propagated at the same time.

**Algorithm 1** Particle-based Trajectory Prediction

**Require:** Starting state $s_0$, action sequence $(a_0, \ldots, a_{T-1})$, number of particles $P$, model ensemble $(\hat{p}_1, \ldots, \hat{p}_m)$

1: **for** particle $i = 1, \ldots, P$ **do**
2: Sample a model $\hat{p}$ from the ensemble uniformly at random.
3: Sample a trajectory $(s_0^{(i)}, s_1^{(i)}, \ldots, s_T^{(i)})$ for the $i^{\text{th}}$ particle by letting $s_0^{(i)} = s_0$ and sampling $s_{t+1}^{(i)}$ from $\hat{p}(s_{t+1}^{(i)} | s_t^{(i)}, a_t)$.
4: **return** predicted trajectories for all particles.

Extensions to predicting the trajectory of a policy requires very minor changes to the algorithm outlined here. We note that there are other possible methods to trajectory prediction in the literature (c.f. moment matching).

## 5.2 Control Policies

Now that we know how to predict trajectories using models, let us go one step further and create control policies. We will describe three schemes, ordered in increasing complexity.

### 5.2.1 Open-Loop Control

A preliminary idea that works well in problems with minimal dynamical stochasticity is to find a sequence of actions that achieves maximal reward. Mathematically, we consider the optimization problem

$$\max_{a_0, a_2, \ldots, a_{T-1}} \mathbb{E}\left[\sum_{t=0}^{T-1} r(s_t, a_t)\right].$$

Since we have a model of the dynamics, we can estimate the expectation to arbitrary precision for any action sequence using Monte Carlo estimation. This suggests the following algorithm:

**Algorithm 2** Open-Loop Control

**Require:** Number of sequences $m$, action sequence proposal distribution $\mu$, dynamics model $\hat{p}$.

1: Sample i.i.d. action sequences $\left\{(a_0^{(i)}, a_1^{(i)}, \ldots, a_{T-1}^{(i)})\right\}_{i=1,\ldots,m}$ from $\mu$.
2: **for** $i = 1, \ldots, m$ **do**
3: Sample from the model $\hat{p}$ to perform Monte Carlo estimation of

$$R_i = \mathbb{E}\left[\sum_{t=0}^{T-1} r(s_t, a_t^{(i)})\right].$$

4: $i^* \leftarrow \text{argmax}_i \, R_i$.
5: Apply action sequence $(a_0^{(i^*)}, a_1^{(i^*)}, \ldots, a_{T-1}^{(i^*)})$ to the environment.

Note that once the action sequence is chosen, we do not make use of any feedback from the MDP to update our decision-making. Since we are not making use of a feedback loop, this is commonly referred to as *open-loop control*.

**When does this work well?** Open-loop control makes two key assumptions about the environment: (1) the environment dynamics have minimal stochasticity, and (2) the dynamics model is very accurate. Without (1), a fixed action sequence would generally be suboptimal because it might be beneficial to change the chosen action sequence depending on the realized environment noise. Similarly, without (2), the effect of the action sequence may be very different from model predictions, and thus it would be useful to modify the plan on-the-fly.

## 5.3 Closed-Loop Control via MPC

A simple modification that accounts for the previously outlined limitations is to allow the agent to *replan* its action sequence at every time step. This allows the agent to plan for unexpected transitions due to either model inaccuracies, or inherent stochasticity, or both. This approach is often known as *model predictive control (MPC)*. Since the agent takes into account the feedback from the environment and acts on that feedback, we say that this is a type of *closed-loop control* (since we are "closing" the feedback loop by acting on it). The modified algorithm is given below:

**Algorithm 3** Model-Predictive Control

**Require:** Number of sequences $m$, action sequence proposal distribution $\mu$, dynamics model $\hat{p}$.

1: **for** every timestep $t$ **do**

2: Sample i.i.d. action sequences $\left\{(a_t^{(i)}, a_{t+1}^{(i)}, \ldots, a_{t+T-1}^{(i)})\right\}_{i=1,\ldots,m}$ from $\mu$.

3: **for** $i = 1, \ldots, m$ **do**

4: Sample from the model $\hat{p}$ to perform Monte Carlo estimation of

$$R_i = \mathbb{E}\left[\sum_{s=0}^{T-1} r(s_{t+s}, a_{t+s}^{(i)}) \;\middle|\; s_t\right].$$

5: Apply the first action from the best sequence to the environment.

## 5.4 MPC with Policies

Although MPC as presented above accounts for possible model inaccuracies and stochasticity in the environment itself, there is actually still some mismatch between how the agent evaluates each action sequence, and how the agent actually rolls out. In particular, the evaluation of each action sequence does not account for the fact that the agent may replan upon taking an action!

In fact, we have already seen how to implement behaviors that act based on a current state at any given time: policies. To handle the mismatch above, we can simply use model predictive control as before, but instead of finding the best action sequence, we instead find the best policy parameters within a class.

**Algorithm 4** Model-Predictive Control with Policies

**Require:** Number of policy candidates $m$, policy proposal distribution $\mu$, dynamics model $\hat{p}$.

1: **for** every timestep $t$ **do**

2: Sample $m$ policy candidates $\pi_{\theta_1}, \ldots, \pi_{\theta_m}$ from $\mu$.

3: **for** $i = 1, \ldots, m$ **do**

4: Sample from the model $\hat{p}$ to perform Monte Carlo estimation of

$$R_i = \mathbb{E}\left[\sum_{s=0}^{T-1} r(s_{t+s}, a_{t+s}) \;\middle|\; s_t\right], \quad a_{t+s} \sim \pi_{\theta_i}(s_{t+s}).$$

5: Apply the action from the best policy.