Week 7 Precept Notes: Model-Based RL

ECE 433/COS 435

April 6, 2024

1 Introduction

This week, we will be discussing model-based RL algorithms. We will first give a rough outline of an algorithm that encapsulates most MBRL algorithms in practice, and then we will discuss common algorithmic choices.

2 Generic MBRL Algorithm Outline

Algorithm 1 MBRL Algorithm Outline

Require: Model class $\mathcal{M}$, number of episodes N

1: Collect initial set of data using random actions (or other exploratory policy), add to buffer $\mathcal{B}$.
2: **for** $i \in \{1, \ldots, N\}$ **do**
3: Find model $\hat{M}_i$ of best fit to $\mathcal{B}$ within $\mathcal{M}$. ▷ Model fitting step
4: Train (or create) a policy π_i using $\mathcal{M}_i$. ▷ Model-based policy learning
5: Collect an episode from the environment using π_i, add data to buffer $\mathcal{B}$.

We outline a generic MBRL algorithm in Algorithm 1, mainly emphasizing how model training and use are situated with respect to each other. Note that the model-based policy learner can be arbitrarily general: it can make use of MPC, or learn a value function as part of obtaining the policy, and so on. Note that the policy learning step also does not have to necessarily start from scratch for every loop iteration, and can make use of learned quantities (policies, value functions, etc.) from previous timesteps.

3 Pros and Cons of MBRL

Pro: Generalizes across reward functions (with caveats). A dynamics model is independent of reward function, and can be reused across different tasks in the same environment (i.e. when only the reward function is changing). That being said, in practice, different reward functions may induce different visitation densities, so a model trained using Algorithm 1 for one reward function may not be particularly useful for another.

Pro: Faster learning (usually). Learning a dynamics model makes better use of the data, allowing the agent to potentially generalize to unseen states.

Con: Compounding model errors. Models are expected to predict more than just a single transition, resulting in compounding model errors. This is similar to the distribution shift problem with imitation learning. Since planning occurs under the model, this mismatch between the model and the true environment means that model-based methods tend to underperform model-free methods asymptotically.

Con: Added complexity; objective mismatch. Training a model adds more complexity and requires more tuning to get right. Additionally, the objective used to train a predictive model is not necessarily the same as the RL objective, so there is a form of *objective mismatch*[1]. In particular, a model that achieves low MSE is not necessarily a better one for planning.

4 Model Fitting

As discussed in lecture, we need to choose three components when selecting a model: (1) the function applied to the input (s, a) (model architecture), (2) the way the function output is converted into a distribution (noise model), and (3) the loss used to train the model. Additionally, we can choose to form an ensemble of models trained on a bootstrapped[2] dataset to understand where the model is accurate, and where more data is needed. Remember that the noise model models aleatoric uncertainty, while ensembling handles epistemic uncertainty[3].

We will focus on the use of a neural network together with a heteroskedastic noise model, trained using MLE.

4.1 Understanding the MLE loss on a heteroskedastic noise model

Assume that we have a heteroskedastic Gaussian noise model $\mathcal{N}(\mu_\theta(s,a), \Sigma_\theta(s,a))$. Then, the MLE loss on this model is given by

$$\min_\theta \frac{1}{n}\sum_{i=1}^{n}[s' - \mu_\theta(s,a)]^\top \Sigma_\theta^{-1}(s,a)[s' - \mu_\theta(s,a)] + \log\det\Sigma_\theta(s,a)$$

To understand how this loss works, let us consider the following problem: we have samples $(x_1, \ldots, x_n)$ from a distribution $\mathcal{N}(\mu^*, (\sigma^*)^2)$, and we want to estimate both parameters. The MLE loss is then

$$\min_{\mu,\sigma} \frac{1}{n}\sum_{i=1}^{n}\frac{1}{\sigma^2}(x_i-\mu)^2 + \log\sigma^2 = \min_{\mu,\sigma} \frac{1}{n}\sum_{i=1}^{n}\frac{1}{\sigma^2}(x_i-\mu)^2 + 2\log\sigma.$$

Observe that no matter what the value of σ is, μ still solves for the mean. On the other hand, σ is trained using two competing terms: Firstly, σ is encouraged to be larger by the first term, since a larger σ minimizes the impact of the squared residuals. Intuitively, if the variance of the true distribution were larger, then we would expect the deviations to be larger too, so a larger variance reduces the impact of large deviations on the loss. On the other hand, the $\log\sigma$ term encourages the σ to be smaller. These two competing terms encourages the sigma to settle on a value that is consistent with the observed noise in the data.

[1] There are several works that try to address this problem however

[2] Bootstrapping here does not refer to the "bootstrap" we have been using throughout the semester, but the bootstrap in classical statistics.

[3] One can also take a Bayesian approach to modeling epistemic uncertainty.

4.2 Common tricks for training models

1. **Having the architecture directly output $\log \sigma$ instead of σ.** This synergizes well with the $\log \sigma$ term, and this also means that the network does not need a final activation on the variance part of the model when computing losses.

2. **Having the model output state changes.** That is, we consider a model

$$(s, a) \mapsto \mathcal{N}\left(s + \mu_\theta(s, a), \Sigma_\theta(s, a)\right).$$

   This generally works well in continuous state spaces where the dynamics are "smooth", and we only expect the agent's state to change by a little bit after every transition.

3. **Applying a $\theta \mapsto (\sin\theta, \cos\theta)$ transformation to angles.** If we have a state representation with angles, we generally first map those parts of the state using the outlined transformation, and have the model predict $\Delta\theta$. This ensures that the model can take periodicity into account.

4. **Data normalization.** Data normalization plays a huge role in model training, and can be the difference between a working model and a non-working model. The normalization statistics are calculated according to the buffer, and updated as more data come in.

5 Extracting a Policy from a Model

Once we have a trained model, there are many ways to obtain a policy using said model. We can roughly categorize most of these methods as follows:

- Planning-based methods (Model only)
- "Model-augmented" model-free methods
  - Data Augmentation
  - Improving the Critic Update
  - Improving the Actor Update

5.1 Planning-based Methods

We saw an example of this during lecture in the form of model predictive control (MPC). MPC at its core boils down to finding a sequence of actions (or more generally, a policy) that achieves high returns.

Optimization in MPC. Note that the approach presented in lecture, which samples several candidate sequences, and chooses the best one according to model-based evaluation, uses a *zero-order optimization method*. We say zero-order here because we are optimizing for the action sequence only by considering function evaluations, not the gradient (unlike gradient descent, which is a *first-order method*). Zero-order methods are preferred for MPC as the optimization step is part of the action selection method, and we would not want to wait for gradient descent to converge to a local minimum every time before taking a step in the environment.

Improving on random sampling. A way to improve upon random sampling is an approach called the cross-entropy method (CEM). The approach first considers a general sampling distribution μ_θ for the action sequences, with some default initialization θ_0. The idea is to first sample candidates, evaluate each one, then update θ based on the *elites* in the candidate set, and repeat for a specified number of iterations. The idea is we want to adapt the sampling distribution to consider better actions rather than just sampling once as in standard MPC. Note that θ is resetted to θ_0 every time a new action is needed (unless there is a better initialization method, depending on the problem).

5.2 Data Augmentation

Data augmentation allows the agent to make use of the model as a way to collect more (s, a, r, s') pairs. The benefit of this is that it minimizes the impact of model errors, as errors cannot compound if we only predict one (or just a few) steps into the future. A widely-known method taking this approach is called Dyna, while a more modern algorithm that uses the same approach is known as MBPO.

5.3 Improving the Critic Update

Recall n-step TD-learning:

$$Q(s_0, a_0) \leftarrow \sum_{t=0}^{n} \gamma^t r(s_t, a_t) + \gamma^{n+1} Q(s_{n+1}, a_{n+1}).$$

We noted earlier on in the semester that an algorithm using n-step returns is necessarily on-policy, because the distribution of the first term is dependent on how the actions beyond a_0 are chosen, and thus dependent on the behavior policy. This meant that old data was not particularly useful for computing TD targets.

With a model, we are able to sidestep this issue, since we can still use the (s, a, s') data to get a dynamics model. We can then use rollouts from the model to estimate the TD target above. We can think of this approach as a form of data augmentation, allowing the critic to be updated on states beyond those collected by the current behavior policy. By selecting n, we can trade off between getting a better estimate and controlling for compounding errors.

5.4 Improving the Actor Update

We can leverage a similar idea to the one from the previous section to improve the actor update instead. However, let us first explore an idea called the *reparameterization trick*.

5.4.1 Reparameterization

Let us consider optimizing the following objective using gradient descent:

$$J(\theta) = \mathbb{E}_{x \sim p_\theta(x)} \left[L(x) \right]$$

Throughout the semester, we have discussed the use of the REINFORCE trick to obtain an unbiased estimate of the gradient:

$$\nabla_\theta J(\theta) \approx L(x) \nabla_\theta \log p_\theta(x).$$

However, we said that this is a high-variance estimate.

As it turns out, there is another way to estimate this expectation, *if p_θ has a particular form.* Assume that there is a function f and a noise variable $\varepsilon \sim q$ such that $f(\theta, \varepsilon) \sim p_\theta$. That is, we can simulate

sampling from p_θ by instead sampling noise from q (not dependent on θ), and then computing $f(\theta, \varepsilon)$. Then, observe that we can rewrite $J(\theta)$ as

$$J(\theta) = \mathbb{E}_{\varepsilon \sim q}\left[L(f(\theta, \varepsilon))\right] \implies \nabla_\theta J(\theta) = \mathbb{E}_{\varepsilon \sim q}\left[\nabla_\theta L(f(\theta, \varepsilon))\right].$$

Note that the expectation can be estimated arbitrarily well by Monte Carlo sampling.

Example 5.1. Assume that $x \sim \mathcal{N}(\mu, \sigma^2)$ in the setting above. Then, observe that we can let $\varepsilon \sim \mathcal{N}(0, 1)$, and $f((\mu, \sigma^2), \varepsilon) = \mu + \sigma \cdot \varepsilon$. In particular, we can use reparameterization to optimize Gaussian parameters under an objective function dependent on random samples from those Gaussians. ⌟

The reparameterization trick is widely used in deep learning, and is a key detail in the implementation of variational autoencoders (VAEs).

5.4.2 Backpropagation through Time

Why did we discuss reparamterization? Given a Q-value, most actor-critic methods optimize the following objective:

$$\max_\theta \mathbb{E}_{s,a \sim \pi_\theta(a|s)}\left[Q(s, a)\right].$$

Using n-step returns, we can also optimize

$$\max_\theta \mathbb{E}\left[\sum_{t=0}^{n} \gamma^t r(s_t, a_t) + \gamma^{n+1} Q(s_{n+1}, a_{n+1})\right].$$

The benefit of this is with large n, errors in the Q-function have less of an impact on the optimization (due to the γ^n).

However, how do we actually optimize this if we have a model? The key insight here is that if we use a Gaussian noise model, then we can actually use the reparameterization trick! In practice, the way this is implemented is simply by computing a forward pass, but ensuring that Gaussians are sampled using the reparameterization trick. This way, backpropagation can pass gradients through the appropriate parameters even through sampling steps.