CS285 Deep Reinforcement Learning HW4: Model-Based RL

Due November 3rd, 11:59 pm

1 Introduction

The goal of this assignment is to get experience with model-based reinforcement learning. In general, model-based RL consists of two main parts: learning a dynamics function to model observed state transitions, and then using predictions from that model in some way to decide what to do (e.g., use model predictions to learn a policy, or use model predictions directly in an optimization setup to maximize predicted rewards).

In this assignment, you will get both theoretical and practical experience with model-based RL. In the analysis section, you will analyze the effectiveness of a simple count-based model. Before doing that section, it will be greatly beneficial to first go over lecture 17 of this course on basics of RL theory (if you wish to complete this section before the lecture, the same lecture from past years will be sufficient); another beneficial resource will be section 2 of this textbook. Then, in the coding section, you will implement both the process of learning a dynamics model, as well as the process of creating a controller to perform action selection through the use of these model predictions. For references to this type of approach, see this paper and this paper.

2 Analysis

Setting. We have a discounted tabular MDP $M = (\mathcal{S}, \mathcal{A}, P, r, \gamma)$ where $\mathcal{S}$, $\mathcal{A}$ are a finite set of states and actions, P is the dynamics model (where $P(\cdot \mid s, a)$ is a probability distribution over states), r is a reward function (where rewards are between $[0, 1]$), and $\gamma \in (0, 1)$ is the discount factor.

Learning a dynamics model. We consider the most naive model-based algorithm. Suppose we have access to a simulator of the environment, and at each state-action pair (s, a), we call our simulator N times retrieving samples $s' \sim P(\cdot \mid s, a)$. Then, we build a dynamics model of the environment as simply:

$$\widehat{P}(s' \mid s, a) = \frac{\mathsf{count}(s, a, s')}{N}$$

where $\mathsf{count}(s, a, s')$ is the number of times we observed (s, a) transitioning to s'. For tabular MDP M, we can view $\widehat{P}$ as a matrix of size $|\mathcal{S}||\mathcal{A}| \times |\mathcal{S}|$.

Additional notation. Let $\widehat{M}$ be an MDP identical to M, except where the true dynamics P is replaced by model $\widehat{P}$. Let $\widehat{V}^\pi$, $\widehat{Q}^\pi$, $\widehat{V}^*$, and $\widehat{Q}^*$ denote the value function and state-action value function, and optimal value and state-action value function, in $\widehat{M}$, respectively.

Problem 2.1. In lecture 17, we saw a proof of a lemma called the *Simulation Lemma*, which states that for any policy π:

$$Q^\pi - \widehat{Q}^\pi = \gamma(I - \gamma \widehat{P}^\pi)^{-1}(P - \widehat{P})V^\pi \,.$$

Prove the following similar lemmma, which we dub the *Alternative Simulation Lemma*:

(Alternative Simulation Lemma) For any policy π, we have:

$$Q^\pi - \widehat{Q}^\pi = \gamma(I - \gamma P^\pi)^{-1}(P - \widehat{P})\widehat{V}^\pi \,.$$

Problem 2.2 In lecture 17, we saw how to bound $||Q^\pi - \widehat{Q}^\pi||_\infty$ using the Simulation Lemma and standard concentration arguments. We will attempt to do the same with the Alternative Simulation Lemma derived in Problem 2.1. Which of the following statements (may be multiple) are correct?

Hint: A statement is correct if the inequalities referenced are applied correctly, and if their assumptions hold before applying them.

Hint 2: For each observed transition from (s, a) to s', you can define a random variable $X = \mathbb{I}_{s'} \cdot V$ that is the dot product between $\mathbb{I}_{s'} \in \mathbb{R}^{|\mathcal{S}|}$ an indicator vector at s' and vector $V \in \mathbb{R}^{|\mathcal{S}|}$, and whose expected value is $\mathbb{E}[X] = P(\cdot \mid s, a) \cdot V$. What does Hoeffding's inequality look like when applied to all N observed transitions from (s, a) in this way? Can Hoeffding's inequality be applied for any vector V?

1. For any policy π and $\delta > 0$, the following holds with probability at least $1 - \delta$,

$$\begin{aligned} ||(P - \widehat{P})\widehat{V}^\pi||_\infty &\leq \max_{s,a} ||P(\cdot \mid s, a) - \widehat{P}(\cdot \mid s, a)||_1 ||\widehat{V}^\pi||_\infty \\ &\leq \frac{1}{1-\gamma} \sqrt{\frac{2|\mathcal{S}| \log(2|\mathcal{S}||\mathcal{A}|/\delta)}{N}}, \end{aligned}$$

   where we use Hoeffding's inequality and the union bound in the second inequality.

2. For any policy π and $\delta > 0$, the following holds with probability at least $1 - \delta$,

$$||(P - \widehat{P})\widehat{V}^\pi||_\infty \leq \frac{1}{1-\gamma} \sqrt{\frac{2 \log(2|\mathcal{S}||\mathcal{A}|/\delta)}{N}},$$

   using Hoeffding's inequality and the union bound.

3. For $\delta > 0$, the following holds with probability at least $1 - \delta$,

$$||(P - \widehat{P})V^*||_\infty \leq \frac{1}{1-\gamma} \sqrt{\frac{2 \log(2|\mathcal{S}||\mathcal{A}|/\delta)}{N}},$$

   where the inequality arises from Hoeffding's inequality and the union bound.

4. For $\delta > 0$, the following holds with probability at least $1 - \delta$,

$$||(P - \widehat{P})\widehat{V}^*||_\infty \leq \frac{1}{1-\gamma} \sqrt{\frac{2 \log(2|\mathcal{S}||\mathcal{A}|/\delta)}{N}},$$

   where we use Hoeffding's inequality and the union bound.

3 Model-Based Reinforcement Learning

We will now provide a brief overview of model-based reinforcement learning (MBRL), and the specific type of MBRL you will be implementing in this homework. Please see Lecture 11: Model-Based Reinforcement Learning (with specific emphasis on the slides near page 9) for additional details.

MBRL consists primarily of two aspects: (1) learning a dynamics model and (2) using the learned dynamics models to plan and execute actions that minimize a cost function (or maximize a reward function).

3.1 Dynamics Model

In this assignment, you will learn a neural network dynamics model f_θ of the form

$$\hat{\Delta}_{t+1} = f_\theta(\mathbf{s}_t, \mathbf{a}_t) \tag{1}$$

which predicts the change in state given the current state and action. So given the prediction $\hat{\Delta}_{t+1}$, you can generate the next prediction with

$$\hat{\mathbf{s}}_{t+1} = \mathbf{s}_t + \hat{\Delta}_{t+1}. \tag{2}$$

See the previously referenced paper for intuition on why we might want our network to predict state differences, instead of directly predicting next state.

You will train f_θ in a standard supervised learning setup, by performing gradient descent on the following objective:

$$\mathcal{L}(\theta) = \sum_{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \in \mathcal{D}} \|(\mathbf{s}_{t+1} - \mathbf{s}_t) - f_\theta(\mathbf{s}_t, \mathbf{a}_t)\|_2^2 \tag{3}$$

$$= \sum_{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \in \mathcal{D}} \|\Delta_{t+1} - \hat{\Delta}_{t+1}\|_2^2 \tag{4}$$

In practice, it's helpful to normalize the target of a neural network. So in the code, we'll train the network to predict a *normalized* version of the change in state, as in

$$\mathcal{L}(\theta) = \sum_{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \in \mathcal{D}} \|\text{Normalize}(\mathbf{s}_{t+1} - \mathbf{s}_t) - f_\theta(\mathbf{s}_t, \mathbf{a}_t)\|_2^2. \tag{5}$$

Since f_θ is trained to predict the normalized state difference, you generate the next prediction with

$$\hat{\mathbf{s}}_{t+1} = \mathbf{s}_t + \text{Unnormalize}(f_\theta(\mathbf{s}_t, \mathbf{a}_t)). \tag{6}$$

3.2 Action Selection

Given the learned dynamics model, we now want to select and execute actions that minimize a known cost function (or maximize a known reward function). Ideally, you would calculate these actions by solving the following optimization:

$$\mathbf{a}_t^* = \arg\min_{\mathbf{a}_{t:\infty}} \sum_{t'=t}^{\infty} c(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}) \quad \text{where} \quad \hat{\mathbf{s}}_{t'+1} = \hat{\mathbf{s}}_{t'} + f_\theta(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}). \tag{7}$$

However, solving Eqn. 7 is impractical for two reasons: (1) planning over an infinite sequence of actions is impossible and (2) the learned dynamics model is imperfect, so using it to plan in such an open-loop manner will lead to accumulating errors over time and planning far into the future will become very inaccurate.

Instead, one alternative is to solve the following gradient-free optimization problem:

$$\mathbf{A}^* = \arg\min_{\{\mathbf{A}^{(0)}, \ldots, \mathbf{A}^{(K-1)}\}} \sum_{t'=t}^{t+H-1} c(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}) \quad \text{s.t.} \quad \hat{\mathbf{s}}_{t'+1} = \hat{\mathbf{s}}_{t'} + f_\theta(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}), \tag{8}$$

in which $\mathbf{A}^{(k)} = (a_t^{(k)}, \ldots, a_{t+H-1}^{(k)})$ are each a random action sequence of length H. What Eqn. 8 says is to consider K random action sequences of length H, predict the result (i.e., future states) of taking each of these action sequences using the learned dynamics model f_θ, evaluate the cost/reward associated with each candidate action sequence, and select the best action sequence. Note that this approach only plans H steps into the future, which is desirable because it prevent accumulating model error, but is also limited because it may not be sufficient for solving long-horizon tasks.

A better alternative to this random-shooting optimization approach is the cross-entropy method (CEM), which is similar to random-shooting, but with iterative improvement of the distribution of actions that are sampled from. We first randomly initialize a set of K action sequences $\mathbf{A}^{(0)}, ..., A^{(K-1)}$, like in random-shooting. Then, we choose the J sequences with the highest predicted sum of discounted rewards as the "elite" action sequences. We then fit a diagonal Gaussian with the same mean and variance as the "elite" action sequences, and use this as our action sampling distribution for the next iteration. After repeating this process M times, we take the final mean of the Gaussian as the optimized action sequence. See Section 3.3 in this paper for more details.

Additionally, since our model is imperfect and things will never go perfectly according to plan, we adopt a model predictive control (MPC) approach, where at every time step we perform random-shooting or CEM to select the best H-step action sequence, but then we execute only the first action from that sequence before replanning again at the next time step using updated state information. This reduces the effect of compounding errors when using our approximate dynamics model to plan too far into the future.

3.3 On-Policy Data Collection

Although MBRL is in theory off-policy—meaning it can learn from any data—in practice it will perform poorly if you don't have on-policy data. In other words, if a model is trained on only randomly-collected data, it will (in most cases) be insufficient to describe the parts of the state space that we may actually care about. We can therefore use on-policy data collection in an iterative algorithm to improve overall task performance. This is summarized as follows:

Algorithm 1 Model-Based RL with On-Policy Data

Run base policy $\pi_0(\mathbf{a}_t, \mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})\}$
while *not done* **do**
  Train f_θ using $\mathcal{D}$ (Eqn. 4)
  $\mathbf{s}_t \leftarrow$ current agent state
  for *rollout number* $m = 0$ *to* M **do**
    for *timestep* $t = 0$ *to* T **do**
      $\mathbf{A}^* = \pi_{\text{MPC}}(\mathbf{a}_t, \mathbf{s}_t)$ where π_{MPC} is obtained from random-shooting or CEM
      $\mathbf{a}_t \leftarrow$ first action in $\mathbf{A}^*$
      Execute $\mathbf{a}_t$ and proceed to next state $\mathbf{s}_{t+1}$
      Add $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$ to $\mathcal{D}$
    end
  end
end

3.4 Ensembles

A simple and effective way to improve predictions is to use an ensemble of models. The idea is simple: rather than training one network f_θ to make predictions, we'll train N independently initialized networks $\{f_{\theta_n}\}_{n=1}^N$. At test time, for each candidate action sequence, we'll generate N independent rollouts and average the rewards of these rollouts to choose the best action sequence.

4 Code

You will implement the MBRL algorithm described in the previous section.

4.1 Overview

Obtain the code from https://github.com/berkeleydeeprlcourse/homework_fall2023/tree/master/hw4. You will be implementing a model-based RL agent in `cs285/agents/model_based_agent.py`. Make sure to also read the following files:

- `cs285/env_configs/mpc_config.py`: generates all of the configuration for the model-based agent.

Problem 1

What you will implement:
Collect a large dataset by executing random actions. Train a neural network dynamics model on this fixed dataset. The implementation that you will do here will be for training the dynamics model.

What code files to fill in:

1. `cs285/agents/model_based_agent.py`: up to (and including) `update_statistics`.
2. `cs285/scripts/run_hw4.py`: everything except for `collect_mbpo_rollout` at the top of the file.

What command to run:

```
python cs285/scripts/run_hw4.py -cfg experiments/mpc/halfcheetah_0_iter.yaml
```

This config will only run the first iteration without actually evaluating the policy, meaning it will only train the ensemble of dynamics models. The code will produce plots inside your logdir that illustrate the full learning curve of the dynamics models. For the first command, the loss should go below 0.2 by iteration 500.

Modify `experiments/mpc/halfcheetah_0_iter.yaml` to change some hyperparameters. Try at least two other configurations of hyperparameters that affect dynamics model training (e.g., number of layers, hidden size, or learning rate).

What to submit: For this question, submit the learning curve for 3 runs total: the initial run with provided hyperparameters as well as 2 of your own.

Note that for these learning curves, we intend for you to just copy the png images produced by the code.

Problem 2

What will you implement:
Action selection using your learned dynamics model and a given reward function.

What code files to fill in:

1. `cs285/agents/model_based_agent.py`: the rest of the file, except for the CEM strategy in `get_action`.

What commands to run:

```
python cs285/scripts/run_hw4.py -cfg experiments/mpc/obstacles_1_iter.yaml
```

Recall the overall flow of our training loop. We first collect data with our policy (which starts as random), we train our model on that collected data, we evaluating the resulting MPC policy (which now uses the trained model), and repeat. To verify that your MPC is indeed doing reasonable action selection, run one iteration of this process using the command above. This will evaluate your MPC policy, but not use it to collect data for future iterations. Look at `eval_return`, which should be greater than -70 after one iteration.

What to submit:
Submit this run as part of your run_logs, and report your `eval_return`.

Problem 3

What will you implement:

MBRL algorithm with on-policy data collection and iterative model training.

What code files to fill in:

None. You should already have done everything that you need, because `run_hw4.py` already aggregates your collected data into a replay buffer. Thus, iterative training means to just train on our growing replay buffer while collecting new data at each iteration using the most newly trained model.

What commands to run:

```
python cs285/scripts/run_hw4.py -cfg experiments/mpc/obstacles_multi_iter.yaml

python cs285/scripts/run_hw4.py -cfg experiments/mpc/reacher_multi_iter.yaml

python cs285/scripts/run_hw4.py -cfg experiments/mpc/halfcheetah_multi_iter.yaml
```

You should expect rewards of around -25 to -20 for the obstacles env, rewards of around -300 to -250 for the reacher env, and rewards of around 250-350 for the cheetah env.

What to submit:
Submit these runs as part of your run_logs, and include the return plots in your pdf.

Problem 4

What will you implement:
You will compare the performance of your MBRL algorithm as a function of three hyperparameters: the number of models in your ensemble, the number of random action sequences considered during each action selection, and the MPC planning horizon.

What code files to fill in:
None.

What commands to run:

```
python cs285/scripts/run_hw4.py -cfg experiments/mpc/reacher_ablation.yaml
```

Modify (or make copies of) the YAML file to ablate each of the hyperparameters. For each hyperparameter, do at least 1 run with it increased and 1 with it decreased from the default (so 7 runs total). Make sure to keep the other hyperparameters the same when studying the effect of one of them.

What to submit:

1. Submit these runs as part of your run_logs.
2. Include the following plots (as well as captions that describe your observed trends) of the following:
    - effect of ensemble size
    - effect of the number of candidate action sequences
    - effect of planning horizon

Be sure to include titles and legends on all of your plots.

Problem 5

What will you implement:
You will compare the performance of your MBRL algorithm with action selecting performed by random-shooting (what you have done up to this point) and CEM.

Because CEM can be much slower than random-shooting, we will only run MBRL for 5 iterations for this problem. We will try two hyperparameter settings for CEM and compare their performance to random-shooting.

What code files to fill in:

1. `cs285/agents/model_based_agent.py`: the CEM action selection strategy.

What commands to run:

```
python cs285/scripts/run_hw4.py -cfg experiments/mpc/halfcheetah_cem.yaml
```

You should expect rewards around 800 or higher when using CEM on the cheetah env. Try a `cem_iterations` value of both 2 and 4, and compare results.

What to submit:
1) Submit these runs as part of your run_logs.

2) Include a plot comparing random shooting (from Problem 3) with CEM, as well as captions that describe how CEM affects results for different numbers of sampling iterations (2 vs. 4).

Problem 6

What will you implement:

In this homework you will also be implementing a variant of MBPO. Another way of leveraging the learned model is through generating additional samples to train the policy and value functions. Since RL often requires many environment interaction samples, which can be costly, we can use our learned model to generate additional samples to improve sample complexity. In MBPO, we build on your SAC implementation from HW3 and use the learned model you implemented in the earlier questions for generating additional samples to train our SAC agent. We will try three settings:

1. Model-free SAC baseline: no additional rollouts from the learned model.
2. Dyna (technically "dyna-style" - the original Dyna algorithm is a little different): add single-step rollouts from the model to the replay buffer and incorporate additional gradient steps per real world step.
3. MBPO: add in 10-step rollouts from the model to the replay buffer and incorporate additional gradient steps per real world step.

What code files to fill in:

1. `cs285/scripts/run_hw4.py`: the `collect_mbpo_rollout` function at the top of the file.

What commands to run:

```
python cs285/scripts/run_hw4.py -cfg experiments/mpc/halfcheetah_mbpo.yaml --sac_config_file
    experiments/sac/halfcheetah_clipq.yaml
```

Edit `experiments/sac/halfcheetah_clipq.yaml` to change the MBPO rollout length. The model-free SAC baseline corresponds to a rollout length of 0, The Dyna-like algorithm corresponds to a rollout length of 1, and full MBPO corresponds to a rollout length of 10.

You should be able to reach returns around 700 or higher with full MBPO with a rollout length of 10.

What to submit:
1) Submit these 3 runs as part of your run_logs.

2) Include a plot to show a comparison between the 3 runs, and explain any trends you see.

Submission

4.2 Submitting the PDF

Your report should be a PDF document containing the plots and responses indicated in the questions above.

4.3 Submitting the Code and Logs

In order to turn in your code and experiment logs, create a folder that contains the following:

- A folder named `data` with all the experiment runs from this assignment. **Do not change the names originally assigned to the folders, as specified by `exp_name` in the instructions.** To minimize submissions size, please include runs with video logging disabled. If you would like to reuse your video logging runs, please see the script provided in `cs285/scripts/filter_events.py`.
- The `cs285` folder with all the `.py` files, with the same names and directory structure as the original homework repository (not include the `data/` folder). A plotting script should also be submitted, which should be a python script (or jupyter notebook) such that running it can generate all plots from your pdf. This plotting script should extract its values directly from the experiments in your `run_logs` and should not have hardcoded reward values.

As an example, the unzipped version of your submission should result in the following file structure. **Make sure that the submit.zip file is below 15MB and that they include the prefix `hw4_mb_`.**

```
submit.zip
├── data
│   ├── cheetah...
│   │   └── events.out.tfevents.1567529456.e3a096ac8ff4
│   └── ...
├── cs285
│   ├── agents
│   │   ├── model_based_agent.py
│   │   └── ...
│   ├── policies
│   │   └── ...
│   └── ...
├── README.txt
└── ...
```

If you are a Mac user, **do not use the default "Compress" option to create the zip**. It creates artifacts that the autograder does not like. You may use `zip -vr submit.zip submit -x "*.DS_Store"` from your terminal.

Turn in your assignment on Gradescope. Upload the zip file with your code and log files to **HW4 Code**, and upload the PDF of your report to **HW4**.