

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 reinforcement learning 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 do the latter. 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 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).

2.1 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

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

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:

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

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

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

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

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

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

2.2 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:

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

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:

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

in which $\mathbf{A}^{(k)} = (a_t^{(k)}, \dots, a_{t+H-1}^{(k)})$ 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 prevents 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 $\mathbf{A}^{(0)}, \dots, \mathbf{A}^{(K-1)}$, 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.

2.3 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  $f_\theta$  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  $\pi_{\text{MPC}}$  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
```

2.4 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 $\{f_{\theta_n}\}_{n=1}^N$, and average their predictions to get your final predictions

$$f(s_t, a_t) = \frac{1}{N} \sum_{n=1}^N f_{\theta_n}(s_t, a_t). \quad (9)$$

In this assignment, you'll train an ensemble of networks and compare how different values of N effect the model's performance.

3 Code

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

3.1 Overview

Obtain the code from https://github.com/berkeleydeeprlcourse/homework_fall2021/tree/master/hw4.

You will add code to the following three files: `agents/mb_agent.py`, `models/ff_model.py`, and `policies/MPC_policy.py`. You will also need to edit these files by copying code from past homeworks or Piazza: `infrastructure/rl_trainer.py` and `infrastructure/utils.py`.

Problem 1

What you will implement:

Collect a large dataset by executing random actions. Train a neural network dynamics model on this fixed dataset and visualize the resulting predictions. The implementation that you will do here will be for training the dynamics model, and comparing its predictions against ground truth. You will be reusing the utilities you wrote for HW1 (or Piazza) for the data collection part (look for “get this from Piazza” markers).

What code files to fill in:

1. cs285/agents/mb_agent.py
2. cs285/models/ff_model.py
3. cs285/infrastructure/utils.py
4. cs285/policies/MPC_policy.py (just one line labeled TODO(Q1) for now)

What commands to run:

```
python cs285/scripts/run_hw4_mb.py --exp_name q1_cheetah_n500_arch1x32
--env_name cheetah-cs285-v0 --add_sl_noise --n_iter 1 --
batch_size_initial 20000 --num_agent_train_steps_per_iter 500 --
n_layers 1 --size 32 --scalar_log_freq -1 --video_log_freq -1 --
mpc_action_sampling_strategy 'random'
python cs285/scripts/run_hw4_mb.py --exp_name q1_cheetah_n5_arch2x250
--env_name cheetah-cs285-v0 --add_sl_noise --n_iter 1 --
batch_size_initial 20000 --num_agent_train_steps_per_iter 5 --
n_layers 2 --size 250 --scalar_log_freq -1 --video_log_freq -1 --
mpc_action_sampling_strategy 'random'
python cs285/scripts/run_hw4_mb.py --exp_name q1_cheetah_n500_arch2x250
--env_name cheetah-cs285-v0 --add_sl_noise --n_iter 1 --
batch_size_initial 20000 --num_agent_train_steps_per_iter 500 --
n_layers 2 --size 250 --scalar_log_freq -1 --video_log_freq -1 --
mpc_action_sampling_strategy 'random'
```

Your code will produce plots inside your logdir that illustrate your model prediction error (MPE). The code will also produce a plot of the losses over time. For the first command, the loss should go below 0.2 by the iteration 500. These plots illustrate, for a fixed action sequence, the difference between your model’s predictions (red) and the ground-truth states (green). Each plot corresponds to a different state element, and the title reports the mean mean-squared-error across all state elements. As illustrated in the commands above, try different neural network architectures as well different amounts of training. Compare the results by looking at the loss values (i.e., itr_0.losses.png), the qualitative model predictions (i.e., itr_0.predictions.png), as well as the quantitative MPE values (i.e., in the title of itr_0.predictions.png).

What to submit: For this question, submit the qualitative model predictions (itr_0_predictions.png) for each of the three runs above. Comment on which model performs the best and why you think this might be the case.

Note that for these qualitative model prediction plots, 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/policies/MPC_policy.py` (all lines labeled TODO (Q2), i.e. everything except the CEM section)

What commands to run:

```
python cs285/scripts/run_hw4_mb.py --exp_name
q2_obstacles_singleiteration --env_name obstacles-cs285-v0 --
add_sl_noise --num_agent_train_steps_per_iter 20 --n_iter 1 --
batch_size_initial 5000 --batch_size 1000 --mpc_horizon 10 --
mpc_action_sampling_strategy 'random'
```

Recall the overall flow of our `rl_trainer.py`. We first collect data with our policy (which starts as random), we then train our model on that collected data, and we then evaluate the resulting MPC policy (which now uses the trained model). To verify that your MPC is indeed doing reasonable action selection, run the command above and compare `Train_AverageReturn` (which was the execution of random actions) to `Eval_AverageReturn` (which was the execution of MPC using a model that was trained on the randomly collected training data). You can expect `Train_AverageReturn` to be around -160 and `Eval_AverageReturn` to be around -70 to -50.

What to submit:

Submit this run as part of your `run_logs`, and include a plot of `Train_AverageReturn` and `Eval_AverageReturn` in your pdf. Note that these will just be single dots on the plot, since we ran this for just 1 iteration.

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 `rl_trainer.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_mb.py --exp_name q3_obstacles --env_name
obstacles-cs285-v0 --add_sl_noise --num_agent_train_steps_per_iter
20 --batch_size_initial 5000 --batch_size 1000 --mpc_horizon 10 --
n_iter 12 --mpc_action_sampling_strategy 'random'

python cs285/scripts/run_hw4_mb.py --exp_name q3_reacher --env_name
reacher-cs285-v0 --add_sl_noise --mpc_horizon 10 --
num_agent_train_steps_per_iter 1000 --batch_size_initial 5000 --
batch_size 5000 --n_iter 15 --mpc_action_sampling_strategy 'random'

python cs285/scripts/run_hw4_mb.py --exp_name q3_cheetah --env_name
cheetah-cs285-v0 --mpc_horizon 15 --add_sl_noise --
num_agent_train_steps_per_iter 1500 --batch_size_initial 5000 --
batch_size 5000 --n_iter 20 --mpc_action_sampling_strategy 'random'
```

You should expect rewards of around -25 to -20 for the obstacles env (takes 40 minutes), rewards of around -250 to -300 for the reacher env (takes 2-3 hours), and rewards of around 250-350 for the cheetah env takes 3-4 hours. All numbers assume no GPU.

What to submit:

Submit these runs as part of your `run_logs`, and include the performance 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_mb.py --exp_name q4_reacher_horizon5 --
    env_name reacher-cs285-v0 --add_sl_noise --mpc_horizon 5 --
    mpc_action_sampling_strategy 'random' --
    num_agent_train_steps_per_iter 1000 --batch_size 800 --n_iter 15 --
    mpc_action_sampling_strategy 'random'
python cs285/scripts/run_hw4_mb.py --exp_name q4_reacher_horizon15 --
    env_name reacher-cs285-v0 --add_sl_noise --mpc_horizon 15 --
    num_agent_train_steps_per_iter 1000 --batch_size 800 --n_iter 15 --
    mpc_action_sampling_strategy 'random'
python cs285/scripts/run_hw4_mb.py --exp_name q4_reacher_horizon30 --
    env_name reacher-cs285-v0 --add_sl_noise --mpc_horizon 30 --
    num_agent_train_steps_per_iter 1000 --batch_size 800 --n_iter 15 --
    mpc_action_sampling_strategy 'random'

python cs285/scripts/run_hw4_mb.py --exp_name q4_reacher_numseq100 --
    env_name reacher-cs285-v0 --add_sl_noise --mpc_horizon 10 --
    num_agent_train_steps_per_iter 1000 --batch_size 800 --n_iter 15 --
    mpc_num_action_sequences 100 --mpc_action_sampling_strategy 'random'
python cs285/scripts/run_hw4_mb.py --exp_name q4_reacher_numseq1000 --
    env_name reacher-cs285-v0 --add_sl_noise --mpc_horizon 10 --
    num_agent_train_steps_per_iter 1000 --batch_size 800 --n_iter 15 --
    mpc_num_action_sequences 1000 --mpc_action_sampling_strategy '
    random'

python cs285/scripts/run_hw4_mb.py --exp_name q4_reacher_ensemble1 --
    env_name reacher-cs285-v0 --ensemble_size 1 --add_sl_noise --
    mpc_horizon 10 --num_agent_train_steps_per_iter 1000 --batch_size
    800 --n_iter 15 --mpc_action_sampling_strategy 'random'
python cs285/scripts/run_hw4_mb.py --exp_name q4_reacher_ensemble3 --
    env_name reacher-cs285-v0 --ensemble_size 3 --add_sl_noise --
    mpc_horizon 10 --num_agent_train_steps_per_iter 1000 --batch_size
    800 --n_iter 15 --mpc_action_sampling_strategy 'random'
python cs285/scripts/run_hw4_mb.py --exp_name q4_reacher_ensemble5 --
    env_name reacher-cs285-v0 --ensemble_size 5 --add_sl_noise --
    mpc_horizon 10 --num_agent_train_steps_per_iter 1000 --batch_size
    800 --n_iter 15 --mpc_action_sampling_strategy 'random'
```

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, and be sure to generate your plots by extracting the corresponding performance numbers from your saved tensorboard eventfiles.

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/policies/MPC_policy.py

What commands to run:

```
python cs285/scripts/run_hw4_mb.py --exp_name q5_cheetah_random --
    env_name 'cheetah-cs285-v0' --mpc_horizon 15 --add_sl_noise --
    num_agent_train_steps_per_iter 1500 --batch_size_initial 5000 --
    batch_size 5000 --n_iter 5 --mpc_action_sampling_strategy 'random'

python cs285/scripts/run_hw4_mb.py --exp_name q5_cheetah_cem_2 --
    env_name 'cheetah-cs285-v0' --mpc_horizon 15 --add_sl_noise --
    num_agent_train_steps_per_iter 1500 --batch_size_initial 5000 --
    batch_size 5000 --n_iter 5 --mpc_action_sampling_strategy 'cem'
    --cem_iterations 2

python cs285/scripts/run_hw4_mb.py --exp_name q5_cheetah_cem_4 --
    env_name 'cheetah-cs285-v0' --mpc_horizon 15 --add_sl_noise --
    num_agent_train_steps_per_iter 1500 --batch_size_initial 5000 --
    batch_size 5000 --n_iter 5 --mpc_action_sampling_strategy 'cem'
    --cem_iterations 4
```

You should expect rewards of 800 or higher when using CEM on the cheetah env. The final CEM run takes 2-3 hours on GPU, and over twice as long without GPU, so **we recommend getting started early and using a GPU (e.g. on Colab) for this problem!**

What to submit:

- 1) Submit these runs as part of your run_logs.
- 2) Include a plot comparing random-shooting with CEM, as well as captions that describe how CEM affects results for different numbers of sampling iterations (2 vs. 4).

Be sure to include a title and legend on your plot, and be sure to generate your plot by extracting the corresponding performance numbers from your saved tensorboard eventfiles.

Submission

3.2 Submitting the PDF

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

3.3 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
│   ├── hw4_q1_...
│   │   └── events.out.tfevents.1567529456.e3a096ac8ff4
│   └── ...
├── cs285
│   ├── agents
│   │   └── mb_agent.py
│   │   └── ...
│   ├── policies
│   │   └── ...
│   └── ...
├── README.md
└── ...
```

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**.