AAI4160 Homework 6: Model-based RL

Juii, Kim
2020118019

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 the learned model to decide what to do. Model predictions can be used to directly find an action that maximizes predicted rewards, or to learn a policy.

In this assignment, you will implement learning a dynamics model and planning actions with the learned model. Specifically, you will implement CEM, which generates the action only by planning through the model, without any explicit policy.

2 Code structure overview

The code structure is similar to that of the previous homework. In this assignment, you mainly work with the following codes:

- `aai4160/scripts/run_hw6.py`: the main training loop for your implementation.
- `aai4160/agents/model_based_agent.py`: the learner you will implement.

3 Model-based reinforcement learning

We provide a brief overview of model-based reinforcement learning (MBRL), and the specific type of MBRL you will be implementing in this homework. MBRL primarily consists of two parts: (1) learning a dynamics model (Section 3.1) and (2) using the learned dynamics models to plan and execute actions that maximize a reward function (Section 3.2).

3.1 Learning dynamics model

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

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

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

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

See this 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:

$$L(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \in \mathcal{D}} \left[\|(\mathbf{s}_{t+1} - \mathbf{s}_t) - f_\theta(\mathbf{s}_t, \mathbf{a}_t)\|_2^2 \right]. \tag{3}$$

In practice, it is helpful to normalize the target values of a neural network. Thus, we will train the network to predict a normalized version of the change in state:

$$L(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \in \mathcal{D}} \left[\|\text{Normalize}(\mathbf{s}_{t+1} - \mathbf{s}_t) - f_\theta(\mathbf{s}_t, \mathbf{a}_t)\|_2^2 \right]. \tag{4}$$

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

$$\mathbf{s}_{t+1} = \mathbf{s}_t + \text{Unnormalize}(f_\theta(\mathbf{s}_t, \mathbf{a}_t)). \tag{5}$$

For simplicity, we define $F_\theta(\mathbf{s}, \mathbf{a}) = \mathbf{s} + \text{Unnormalize}(f_\theta(\mathbf{s}, \mathbf{a})) = \mathbf{s}'$.

3.2 Action optimization

Given the learned dynamics model f_θ, we now want to select and execute actions that maximize a known reward function. The optimal actions can be found by solving the following optimization:

$$\mathbf{a}_t = \text{argmax}_{\mathbf{a}_{t:\infty}} \sum_{t'=t}^{\infty} r(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}), \text{where } \hat{\mathbf{s}}_{t'+1} = F_\theta(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}) \text{ and } \hat{\mathbf{s}}_t = \mathbf{s}_t. \tag{6}$$

However, solving Equation (6) 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, we can solve the following finite-sequence, finite-length optimization problem:

$$A^* = \text{argmax}_{\{A^{(0)}, \cdots A^{(K-1)}\}} \sum_{t'=t}^{t+H-1} r(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}), \text{where } \hat{\mathbf{s}}_{t'+1} = F_\theta(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}) \text{ and } \hat{\mathbf{s}}_t = \mathbf{s}_t, \tag{7}$$

where $A^{(k)} = (\mathbf{a}_t^{(k)}, \cdots, \mathbf{a}_{t+H-1}^{(k)})$ is a random action sequence of length H. What Equation (7) 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 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 action sequences that are sampled from. We first randomly initialize a set of K action sequences $\{A^{(0)}, \cdots, A^{(K-1)}\}$, as 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.

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 timestep we perform random shooting or CEM to select the best H-step action sequence, but we execute only the first action from that sequence and replan at the next timestep using updated state information. This reduces the effect of compounding errors when using our approximate dynamics model to plan a 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 do not 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 collection

```
Input: Run base policy $\pi_0(\mathbf{a}_t|\mathbf{s}_t)$ (e.g. random policy) to collect $D = \{\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1}\}$
while not converged do
    // Expand dataset using model rollouts
    Train $f_\theta$ using $\mathcal{D}$ (Equation (4))
    $\mathbf{s}_t \leftarrow$ current state
    for $M$ times do
        for $t = 0, \cdots, T$ do
            $A^* = \pi_{\text{MPC}}(\mathbf{a}_t \mid \mathbf{s}_t)$, where $\pi_{\text{MPC}}$ is obtained from random shooting or CEM
            $\mathbf{a}_t \leftarrow$ first action in $A^*$
            Get $\mathbf{s}_{t+1}$ by executing $\mathbf{a}_t$ in the environment
            Add $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ to $\mathcal{D}$
```

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 train N independently initialized networks: $\{f_\theta^i\}_{i=1}^N$. At test time, for each candidate action sequence, we generate N independent rollouts and average the rewards of these rollouts to choose the best action sequence.

4 Learning dynamics model

In this section, you will train a model using a dataset collected by executing random actions. Specifically, you will train an ensemble of dynamics model using normalization, as mentioned in Section 3.1. A common practice for training the model is also learning the reward of the environment. However, in this homework, you only learn the dynamics and use the true reward function for simplicity.

4.1 Implementation

Let's implement the model training code in `run_hw6.py` and `model_based_agent.py`:

- `aai4160/scripts/run_hw6.py`: implement all TODOs.
- `aai4160/agents/model_based_agent.py`: implement `update()`.

4.2 Experiments

- Train a model on `Halfcheetah` with the configuration `experiments/mpc/halfcheetah_0_iter.yaml`.

  ```
  python aai4160/scripts/run_hw6.py -cfg experiments/mpc/halfcheetah_0_iter.yaml
  ```

- This config will only train the ensemble of dynamics models; thus, there is not policy evaluation. For this experiment, the model loss should go below 0.2 by iteration 500.
- Plot the model loss (**use** `itr_0_loss_curve.png` **inside your logging directory, NOT the tensorboard**).

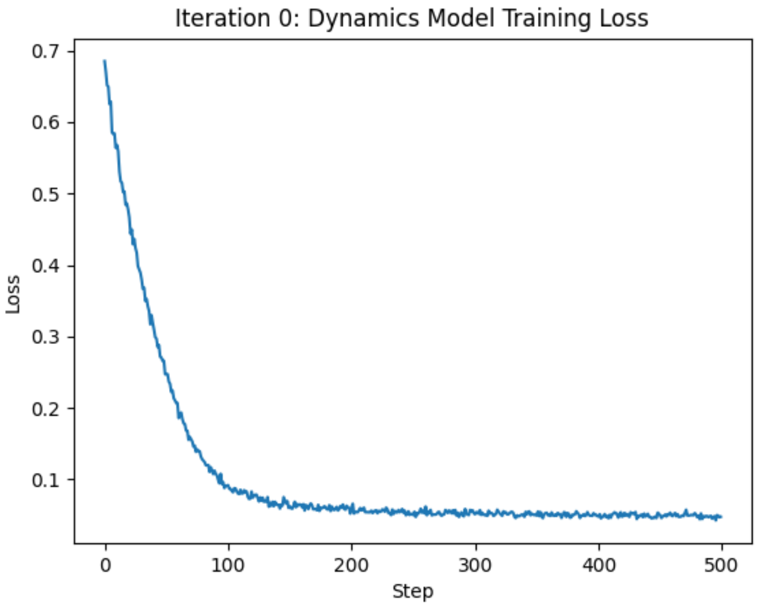

Figure 1: **Loss of Dynamics Model**

5 Random shooting and CEM

5.1 Implementation

In this section, you will implement MPC with two methods: random shooting and CEM.

- `cs285/agents/model_based_agent.py`: please implement all TODOs.

5.2 Experiments

- Train the dynamics model with on-policy data collected from random shooting with the configuration `halfcheetah_multi_iter.yaml` on `Halfcheetah`. You would have rewards 300 or higher.

```
python aai4160/scripts/run_hw6.py -cfg experiments/mpc/halfcheetah_multi_iter.yaml
```

- Train the dynamics model with on-policy data collected from CEM with the configuration `halfcheetah_cem.yaml` on `Halfcheetah`. You would have rewards 800 or higher.

```
python aai4160/scripts/run_hw6.py -cfg experiments/mpc/halfcheetah_cem.yaml
```

- **CEM experiments will take approximately two hours to finish.** If you are using a RTX 3090, we strongly recommend to execute the scripts for the next problem concurrently to maximize the GPU utilization. Otherwise, you would have to wait 6 hours for all experiments.

- Report the results (mean and std. of average returns) for random shooting and CEM. Compare and explain your results.

  Answer:

Table 1: Performance with random shooting and CEM.

	Mean	Standard Deviation
Random shooting	**875.3**	**48.53**
CEM	**2454**	**63.16**

The difference in the results between random shooting and the CEM is due to the optimizatino approaches. Random shooting samples actions randomly, leading to a lower mean reward (875 in table) and higher variance. In contrast, CEM iteratively refines its sampling based on top-performing (elite) action sequences, resulting in a higher mean reward (2454 in table) and more consistent outcomes.

- Train an agent with CEM with the configuration with different hyperparameters on `Halfcheetah`. Specifically, run the code with $H = 1$ and $K = 50$ by executing the code as below (the original value is $H = 15$, $K = 1000$).

```
python aai4160/scripts/run_hw6.py -cfg experiments/mpc/halfcheetah_cem_H-1.yaml
python aai4160/scripts/run_hw6.py -cfg experiments/mpc/halfcheetah_cem_K-50.yaml
```

- Report the CEM results (mean and std. of average returns) with the **3 settings** (original, $H = 1$, $K = 50$). Compare and explain your results.

Answer:

Table 2: Performance of CEM with 3 different hyperparameters.

	Mean	Standard Deviation
CEM	**2454**	**63.16**
CEM ($H = 1$)	**-19445**	**6108**
CEM ($K = 50$)	**1186**	**31.24**

The differences in results stem from how the hyperparameters H (planning horizon) and K (of action sequences) affect the optimization in CEM. With the original setting (H=15, K=1000), the CEM algorithm effectively balances exploration and exploitation, yielding a high mean reward (2484) and relatively low variance (std 70.02). When H is 1, the algorithm plans only one step ahead, significantly limiting its ability to foresee long-term rewards, resulting in a much lower mean reward (-17287) and higher variance (std 13892.74). This shows that a short horizon fails to capture the necessary future state information for effective planning. Reducing K to 50 decreases the number of sampled action sequences, limiting the algorithm's exploration capabilities, and resulting in a lower mean reward (1218) and slightly lower variance (std 50.95). This demonstrates that fewer samples reduce the likelihood of finding optimal action sequences, leading to sub-optimal performance. Thus, both a sufficient planning horizon and a large number of action sequences are crucial for CEM to perform well.

6 Discussion

Please provide us a rough estimate, in hours, for each problem, how much time you spent. This will help us calibrate the difficulty for the class next year.

- **Learning the model: 5 hours**
- **Random shooting and CEM: 6 hours**

Feel free to share your feedback here, if any: **We would really appreciate your feedback to improve the reinforcement learning class.**

7 Submission

Please submit the code, tensorboard logs, and the **"report"** in a single zip file, `hw6_[YourStudentID].zip`. The zip file should be smaller than 20MB. The structure of the submission file should be:

```
hw6_[YourStudentID].zip
├─ hw6_[YourStudentID].pdf
├─ data
│  ├─ cheetah-aai4160-v0_cheetah_...
│  │  └─ events.out.tfevents.1567529456.e3a096ac8ff4
│  └─ ...
├─ aai4160
│  ├─ agents
│  │  └─ model_based_agent.py
│  └─ ...
└─ ...
```