CSE 599B-SP24: Reinforcement Learning
Homework 3 - Model Based RL

Due Wednesday May 31 @ 11:59pm PST

The goal of this homework is to get a basic understanding of model based RL (MBRL). In particular, you will be implementing random MPC with shooting, model-predictive path integral (MPPI) and MPPI with ensembles of models to factor in uncertainty. Please refer to the git link for the assignment `https://github.com/WEIRDLabUW/cse542-sp24-hw3`

Related Lectures: This homework largely involves knowledge from Lecture 10 and Lecture 11 on Model Based RL.

Collaboration: Students can discuss questions, but each student MUST write up their own solution, and code their own solution. We will be checking code/PDFs for plagiarism.

Late Policy: This assignment may be handed in up to 5 days late. If you have used up your 10 late days this quarter, there will be a penalty of 10% of the maximum grade per day.

1 Code Overview

The starter code is written in Python and depends on NumPy and Matplotlib as well as Pytorch. If you are new to Pytorch, please refer to the following tutorial [Link]. The README describes how to install packages in a conda environment to solve this assignment. We also provide a link to a Google Colab notebook below that can be used for cloud execution. We recommend either using a Linux machine or using the Colab, rather than Windows. This section gives a brief overview and the README provides detailed instructions.

- `main.py` - overall launcher with environment creation and policy alternation [DO NOT MODIFY]
- `environment.yml` - Conda env file to install dependencies [DO NOT MODIFY]
- `utils.py` - functions for data collections and networks. [DO NOT MODIFY]
- `evaluate.py` - Evaluating learned policy reward and success rate [DO NOT MODIFY]
- `planning_mbrl.py` - implementation of single and ensemble mbrl [TODO]
- `train_model.py` - training networks [TODO]

Environment Details This is the same conda environment as in HW1, so feel free to reuse that environment. Otherwise, to get started, you must install the correct dependencies for the code to run. There are several ways to do this - if you have a Linux machine, install a conda environment (install Anaconda from here `https://docs.anaconda.com/free/anaconda/install/index.html`) with the correct dependencies from the provided environment.yml file using the following command

```
conda env create -f environment.yml
conda activate cse542a1
```

This should give you an environment with all the necessary dependencies, and you can run the code by running `main.py` as described in each section below.

If you do not have a Linux machine, it is advisable for you to use Google Colab instead. This will provide you with a single GPU instance that should be sufficient to run this code. The Colab instance can be found `https://colab.research.google.com/drive/1O0eyKIKSaOfctpCb9ei8-H0Oldogqn2M`, and all the installs are completed by just running the initial cells inline. Please come talk to the course staff if you are having environment setup issues!

The gym reacher environment is the same as in homework 1, which is 2D environment where a double-jointed arm aims to move its end effector to a target location. You may refer to this follow link for further details: [Link].

2 Random MPC with Shooting [30 points]

```
$ python main.py --model_type single --plan_mode random_mpc
```

Model Predictive Control (MPC) is a method that iteratively optimizes for a sequence of actions that achieve the best performance according to the reward function. It predicts the future states or observations of the system using a learned model and adjusts its actions based on the prediction. In this part, you will implement one technique in model-predictive control for model-based RL: random MPC with shooting. In particular, instead of using a deterministic optimization method to find the best action, you sample N sequences of actions randomly from the action space. Each sequence of actions is stepped forward through the learned model, and the outcomes are evaluated by calculating the total rewards of the predicted future states from the learned model. The trajectory that yields the highest total reward is selected and its first action will be your action. This process is repeated at each step, with the first action of the best action sequence being executed. Please refer to the pseudocode for further details. Note that we have provided model training code for you here, you simply implement planning.

2.1 Random MPC with Shooting Pseudocode

```
def plan_model_random_shooting(env, state, ac_size, horizon, model, reward_fn, n_samples_mpc=100):
    #Initialize state and sample random_actions
    state_repeats = repeat(state, n_samples_mpc)
    random_actions = sample_uniform_random_actions(n_samples_mpc, horizon, ac_size, env.action_space)

    # Roll out based on the sampled random actions
    all_states, all_rewards = rollout_model(model, state_repeats, random_actions, horizon, reward_fn)

    # Compute the total rewards for each trajectory
    total_rewards = sum(all_rewards, axis=-1)

    # Select the trajectory with the highest total reward
    best_ac_idx = argmax(total_rewards)

    # use the first action from the best trajectory
    best_ac = random_actions[best_ac_idx, 0]

def rollout_model(model, initial_states, actions, horizon, reward_fn):
    all_states = []
    all_rewards = []
    curr_state = initial_states
```

```
    for j in range(horizon):
        curr_ac = actions[:, j]
        next_s = model(curr_state, curr_ac)
        next_r = reward_fn(next_s, curr_ac)

        all_states.append(next_s)
        all_rewards.append(next_r)
        curr_state = next_s

    return all_states, all_rewards
```

2.2 Expected Results

We provide an example below which serves as a reference for the expected output; You are expected to get a success rate higher than 0.25. please note that the success rate and reward may fluctuate:

```
$ python main.py  --model_type single --plan_mode random_mpc

using device cuda
Episode: 0, reward: -44.39803820644873, max path length: 50
Episode: 1, reward: -17.73652850726456, max path length: 50
Episode: 2, reward: -16.54873229390969, max path length: 50
Episode: 3, reward: -17.09515906142727, max path length: 50
Episode: 4, reward: -13.370320994343928, max path length: 50
Episode: 5, reward: -16.10938280377134, max path length: 50
Episode: 6, reward: -15.057747536611572, max path length: 50
Episode: 7, reward: -15.66250521516793, max path length: 50
Episode: 8, reward: -16.894985734482116, max path length: 50
Episode: 9, reward: -16.185416048796206, max path length: 50
Episode: 10, reward: -16.945894300410362, max path length: 50
Episode: 11, reward: -15.591242310833053, max path length: 50
Episode: 12, reward: -14.094809423579566, max path length: 50
Episode: 13, reward: -13.313228855386575, max path length: 50
Episode: 14, reward: -14.6755356699752, max path length: 50

...
Success rate:  0.28
Average reward (success only):  -13.936932248085684
Average reward (all):  -15.29830527289742
```

2.3 Execution

1. Fill in the blanks in the code marked with TODO in the `plan_model_random_shooting` function in `planning_mbrl.py`.
2. Fill in the blanks in the code marked with TODO in the `rollout_model` function in `planning_mbrl.py`.
3. Plot the reward during the training with default hyperparameters. Report the success rate and average reward using the evaluate function that is provided to you.

3 Model Predictive Path Integral (MPPI) [30 points]

```
$ python main.py --model_type single --plan_mode mppi
```

Random MPC with shooting provides a simple yet effective way to handle control problems by randomly sampling action sequences and selecting the best one based on predicted rewards, but it may not be the best planning method for more complex environments since it is simply doing random search.

Model Predictive Path Integral (MPPI) Control is a more efficient approach compared to MPC. Unlike random MPC that samples actions uniformly from the action space, MPPI iteratively optimizes the sampled action sequences using importance sampling, focusing on trajectories with higher rewards rather than sampling uniformly. This method focuses the exploration of the action space, helping us to converge faster to optimal actions. In particular, the idea of standard MPPI implementations is to refit the action mean to sample around by computing a weighted sum according to weights computed by the exponential of returns. This helps to focus on trajectories with higher expected rewards and is faster to find the optimal actions. Please refer to the pseudocode below for details, and implement the single model version of MPPI in `planning_mbrl.py`

3.1 MPPI Pseudocode

```
def plan_model_mppi(env, state, ac_size, horizon, model, reward_fn,
                      n_samples_mpc=100, n_iter_mppi=10,
                      gaussian_noise_scales):
    #Initialize state and sample random_actions
    state_repeats = repeat(state, n_samples_mpc)
    random_actions = sample_uniform_random_actions(n_samples_mpc, horizon, ac_size, env.action_space)

    # Compute initial rollout using the model
    all_states, all_rewards = rollout_model(model, state_repeats, random_actions, horizon, reward_fn)
    all_returns = sum(all_rewards along the last axis)

    # Iterate through a few iterations of MPPI
      for iter in range(n_iter_mppi):
        # Weight trajectories by exponential of returns
        weights = compute_exponential_return(all_return)

        # Compute weighted sum of actions
        weighted_sum = sum(random_actions * weights, axis=0)

        # compute mean and std of the best trajectories
        action_mean = weighted_sum.flatten()
        action_std = ones(weighted_sum.shape) * gaussian_noise_scales[iter]

        # Sample new actions
        random_actions = sample_actions(action_mean, action_std, n_samples_mpc)

        # Perform rollout with new actions using the model
        all_states, all_rewards = rollout_model(model, state_repeats, random_actions, horizon,
            reward_fn)
        all_returns = sum(all_rewards)

    # Identify the best action based on final rollouts
    # Select the trajectory with the highest total reward
    best_ac_idx = argmax(total_rewards)
    # use the first action from the best trajectory
    best_ac = random_actions[best_ac_idx, 0]
```

```
    return best_ac, random_actions[best_ac_idx]
```

3.2 Expected Results

We provide an example below which serves as a reference for the expected output; You are expected to get a success rate higher than 0.25. please note that the success rate and reward may fluctuate:

```
$ python main.py  --model_type single --plan_mode mppi

using device cuda
Episode: 0, reward: -45.0685292711599, max path length: 50
Episode: 1, reward: -8.028816349416811, max path length: 50
Episode: 2, reward: -11.181899229796787, max path length: 50
Episode: 3, reward: -10.187376193755673, max path length: 50
Episode: 4, reward: -7.974100628605427, max path length: 50
Episode: 5, reward: -12.418430370016777, max path length: 50
Episode: 6, reward: -9.837710231454908, max path length: 50
Episode: 7, reward: -8.775532875691537, max path length: 50
Episode: 8, reward: -10.247916344380988, max path length: 50
Episode: 9, reward: -9.131945952749149, max path length: 50
Episode: 10, reward: -5.747245489228223, max path length: 50
Episode: 11, reward: -8.331408758062988, max path length: 50
Episode: 12, reward: -8.474610788412223, max path length: 50
Episode: 13, reward: -7.870262004116912, max path length: 50
Episode: 14, reward: -9.202479269493242, max path length: 50
...
Success rate:  0.53
Average reward (success only):  -5.447100768498477
Average reward (all):  -7.627708552630046
```

3.3 Execution

1. Fill in the blanks in the code marked with TODO START-MPPI in the `plan_model_mppi` function in `planning_mbrl.py`.
2. Plot the reward during the training with default hyperparameters. Report the success rate and average reward using the evaluate function that is provided to you.

4 Ensemble MPPI [30 points]

```
$ python main.py --model_type ensemble --plan_mode mppi
```

So far, you've implemented the standard single MPPI method where you train and use a single dynamics model. We can further improve this by introducing model ensembles to incorporate uncertainty. As discussed in lecture, ensemble MPPI is simply using an ensemble multiple networks instead of just one network so as to incorporate uncertainty. By leveraging multiple network predictions, the ensemble captures model uncertainty.

In this setting, MPPI will not just take returns over a single rollout but rather will average the returns from rollouts across multiple elements of the ensemble. In doing so we take the expected return over ensemble elements rather than a single, potentially wrong model. Planning with ensemble MPPI is very similar to single MPPI but we will just roll out K different models with the same actions and average rewards across them for MPPI.

Now when training ensemble models, ensuring diversity is very important to prevent model collapse. To do so, we can have different models optimized differently. For example, each model can have their separate optimizer, batch size and learning rate, or use different architectures. The diversity helps the predictions are not overly biased by the single network, leading to a more stable and accurate policy. In this part, you only need to add the ensemble method in your MPPI implementation (in the `plan_model_mppi` function in `planning_mbrl.py`), but take a look at our implementation on how we ensure diversity when training Ensemble MPPI.

4.1 Expected Results

When you execute the code, you may get similar outputs as below. You are expected to get a success rate higher than 0.4. Please note that the success rate and reward may fluctuate.

```
$ python main.py  --model_type ensemble --plan_mode mppi

Episode: 0, reward: -43.46004802481881, max path length: 50
Episode: 1, reward: -10.8674283885565, max path length: 50
Episode: 2, reward: -12.883924277131664, max path length: 50
Episode: 3, reward: -8.933863638550225, max path length: 50
Episode: 4, reward: -8.313068222586896, max path length: 50
Episode: 5, reward: -7.589171558519515, max path length: 50
Episode: 6, reward: -9.222474745213445, max path length: 50
Episode: 7, reward: -7.393583362317796, max path length: 50
Episode: 8, reward: -6.579779976928853, max path length: 50
Episode: 9, reward: -7.697173480373474, max path length: 50
Episode: 10, reward: -6.232915190180099, max path length: 50
Episode: 11, reward: -8.142131283083632, max path length: 50
Episode: 12, reward: -5.390735122525843, max path length: 50
Episode: 13, reward: -6.648134272343903, max path length: 50
Episode: 14, reward: -5.465611595751769, max path length: 50
...
Success rate:  0.62
Average reward (success only):  -5.194262059586056
Average reward (all):  -6.475862579254006
```

4.2 Execution

1. Add Ensemble MPPI in the `plan_model_mppi` function in `planning_mbrl.py`.
2. Plot the return during the training with default hyperparameters. Report the success rate and average reward using the evaluate function that is provided to you.

5 Discussion [10 points]

1. Compare Random MPC with shooting, MPPI and Ensemble MPPI. Discuss the strengths and weaknesses of each method.
2. Why is diversity important in an ensemble model? What techniques did we use in the homework to improve diversity in the ensemble MPPI?

6 Submission

We will be using the Canvas for submission of the assignments. Please submit the written assignment answers as a PDF. For the code, submit a zip file of the entire working directory.