Homework 4

CMU 10-703: Deep Reinforcement Learning (Fall 2024)
OUT: November 9, 2024
DUE: November 18, 2024 by 11:59pm ET

Instructions: START HERE

- **Collaboration policy:** You may work in groups of up to three people for this assignment. It is also OK to get clarification (but not solutions) from books or online resources after you have thought about the problems on your own. You are expected to comply with the University Policy on Academic Integrity and Plagiarism[1].
- **Late Submission Policy:** You are allowed a total of 8 grace days for your homeworks. However, no more than 3 grace days may be applied to a single assignment. Any assignment submitted after 3 days will not receive any credit. Grace days do not need to be requested or mentioned in emails; we will automatically apply them to students who submit late. We will not give any further extensions so make sure you only use them when you are absolutely sure you need them. See the Assignments and Grading Policy here for more information about grace days and late submissions: `https://cmudeeprl.github.io/703website_f24/logistics/`
- **Submitting your work:**
  - **Gradescope:** Please write your answers and copy your plots into the provided LaTeX template, and upload a PDF to the GradeScope assignment titled "Homework 4." Additionally, export the code from your Colab notebook ([File → Export .py]) and upload it the GradeScope assignment titled "Homework 4: Code." Each team should only upload one copy of each part. Regrade requests can be made within one week of the assignment being graded.
  - **Autolab:** Autolab is not used for this assignment.

[1] `https://www.cmu.edu/policies/`

Problem 1: Model-Based Reinforcement Learning with PETS (100 pts)

In previous homeworks, you implemented REINFORCE, N-Step A2C, and DQN, which are all model-free methods. For this homework, you will implement a model-*based* reinforcement learning (MBRL) method called **PETS** which stands for probabilistic ensemble and trajectory sampling [1]. You will find the the original PETS paper to be very useful for this assignment, so we encourage you to read this paper thoroughly. An overview of MBRL with PETS is shown in Algorithm 1.

Algorithm 1 MBRL with PETS

procedure MBRL(# training iterations K_{train})
    Initialize empty data array D and initialize probabilistic ensemble (PE) of models.
    Sample 100 episodes from the environment using random actions and store into D.
    Train PE for 10 steps using D.
    repeat for K_{train} iterations:
        Sample 1 episode using MPC and latest PE and add to D.
        Train PE for 1 step using D.
end procedure

There are 3 main components to MBRL with PETS:

1. **Probabilistic ensemble of networks:** you will be using probabilistic networks that output a distribution over resulting state given a state and action pair.
2. **Trajectory sampling:** propagate hallucinated trajectories through time by passing hypothetical state-action pairs through different networks of the ensemble.
3. **Model predictive control:** use the trajectory sampling method along with a cost function to perform planning and select good actions.

We will go into more detail on each component below.

Part 1: Probabilistic Ensemble

You are provided starter code in `model.py` that specifies that model architecture that you will be using for each member of the ensemble. In this homework an ensemble of 2 networks should be sufficient. Specifically, each network is a fully connected network with 3-hidden layers, each with 400 hidden nodes. If you have trouble running this network, a smaller network may work as well but may require additional hyperparameter tuning. The starter code also includes a method for calculating the output of each network, which will return the mean and log variance of the next state distribution conditioned on a current state and action. We recommend using a diagonal log variance matrix for models in the PE.

The loss that you should use to train each network for a minibatch of size B is the negative log likelihood of the observed next states under the predicted mean and variance

from the network, conditioned on the observed current states and actions (we recommend using $B = 128$ and using Adam as the optimizer with learning rate 1e-3).

The training routine for the ensemble is shown in Algorithm 2. You will need to implement this in the `train_model` function in `model.py`.

Algorithm 2 Training the Probabilistic Ensemble

procedure TRAIN_STEP(data $\mathcal{D}$, # networks N, minibatch size B)
  for n in $1 : N$:
    Uniformly sample (with replacement) minibatch of size B from $\mathcal{D}$
    Take a gradient step of the loss for sampled minibatch
end procedure

Part 2: Trajectory Sampling

For your implementation of PETS, you will use the TS1 sampling method which is shown in Algorithm 3. Note that this algorithm only determines which networks will be used for which time steps to perform the state prediction. It is only one component of model predictive control and will be combined with the model and action optimization method in the next section. You can think of this trajectory sampling method as returning P particles that each

Algorithm 3 Trajectory Sampling with TS1

procedure TS1(# networks N, # particles P, plan horizon T)
  Initialize array S of dimension $P \times T$ to store network assignments for each particle.
  for p in $1 : P$:
    Randomly sample a sequence s of length T where each $s_i \in \mathcal{S}$ for $i \in \{1, \ldots, N\}$.
    Set $S[p, :] = s$.
end procedure

represent a path of length T where at each time step, a random network in the ensemble is used to generate the next state.

Part 3: Action Selection with Cross Entropy Method

In order to perform action selection, we need a cost function to evaluate the fitness of different states and action pairs. Defining the right cost function is often the hardest part of getting model-based reinforcement learning to work, since the action selection and resulting trajectories from MPC depend on the cost function. Note that it is not appropriate to use the negative of the return for our problem as rewards are so sparse (we only receive rewards when the box reaches the goal). For this reason, we use surrogate costs (negative rewards) that are less sparse. For the `Pushing2D-v1` environment, you will be using the following cost function:

$$\text{cost}(\textbf{pusher}, \textbf{box}, \textbf{goal}) = d(\textbf{pusher}, \textbf{box}) + 2d(\textbf{box}, \textbf{goal}) + 5\left|\frac{x_{\textbf{box}}}{y_{\textbf{box}}} - \frac{x_{\textbf{goal}}}{y_{\textbf{goal}}}\right| \quad (1)$$

where d denotes Euclidian distance and **pusher**, **box**, **goal** denote the tuples of (x, y) coordinates of the pusher, box and goal at time t respectively, and x_{obj} denotes the x coordinate of an object (similar for y). Feel free to play around with other cost functions to see if you can do better.

We can now use TS1, along with the cost function, to estimate the cost of a particular action sequence $a_{1:T}$ from state s_0. Let TS be the output of TS1 for our settings of N, P, T and let $s_{(p,t)} \sim \text{model}_{TS[p,t]}.\text{predict}(s_{(p,t-1)}, a_{t-1})$; that is, the next state for a given particle is the predicted state from the model indicated by TS for the given particle and time step applied to the last state of the particle with the given action from $a_{1:T}$ (remember that we are using a probabilistic model so the output is a sample from a normal distribution). We can now calculate the cost of a state and action sequence pair as the sum of the average of the cost over all particles:

$$\text{C}(a_{1:T}, s) = \frac{1}{P} \sum_{p=1}^{P} \sum_{t=1}^{T} \text{cost}(s_{(p,t)}) \tag{2}$$

where $s_{(p,t)}$ is calculated as described above.

Finally, we can optimize the action sequence with the cross entropy method (CEM), which you implemented in Homework 3. We have included a version of CEM which you are free to use.

Algorithm 4 CEM

```
1: procedure CEM(population size M, # elites e, # CEM iters K_CEM, μ, σ)
2:     Generate M action sequences according to μ and σ from normal distribution.
3:     for i in 1 : K_CEM:
4:         for m in 1 : M:
5:             Calculate the cost of a_{m,1:T} according to Equation 2.
6:         Update μ and σ using the top e action sequences.
7:     return: μ
8: end procedure
```

Finally: Tying It All Together

The only missing piece left to implement Algorithm 1 is line 6, i.e., how to sample an episode using MPC; this is shown in Algorithm 5. We proceed by starting with an initial μ and σ and use that as input to CEM. Then we take the updated μ from CEM and execute the action in the first time step in the environment to get a new state that we use for MPC in the next time step. We then update the μ that is used for the next timestep to be the μ from CEM for the remaining steps in the plan horizon and initialize the last time step to 0. Finally, we return all the state transitions gathered so that they can be appended to D. Please use a planning horizon of $T = 5$.

Note that on line 11, $\mu[1:T]$ uses list indexing following Python indexing conventions (i.e. using zero indexing and not including T). For this homework, please respond to the prompts below:

Algorithm 5 Generating an episode using MPC

procedure MPC(env, plan horizon T)
    `transitions` = []
    s = `env.reset()`
    $\mu = \mathbf{0}$, $\sigma = \mathbf{1}$
    while not done:
        μ =CEM(200, 20, 5, μ, σ)
        $a = \mu[0, :]$
        $s' =$ `env.step(a)`
        `transitions.append`(s, a, s')
        $s = s'$
        $\mu = \mu[1 : T]$`.append(0)`
    return: `transitions`
end procedure

Problem 1.1: Model-based Predictive Control (25 pts)

Before you begin to implement, we recommend going through how `run.py` works in a top-down manner. See how each component of the code works toghether. For all the questions in 2.1, we provide the starter code in the `ExperimentGTDynamics` class in `run.py`.

1. (10 pts) For this question, you need to implement the `predict_next_state_gt` function in `mpc.py`. Then test CEM with the ground-truth dynamics on the Pushing2D-v1 environment. The CEM policy will also be used later for planning over a learned dynamics model. All the hyper-parameters are provided in the code. Report the percentage of success over 50 episodes.

2. (10 pts) Instead of CEM, plan with random action sequences where each action is generated independently from a normal distribution $\mathcal{N}(0, 0.5I)$, where I is an identity matrix. Use the same number of trajectories for planning as CEM, i.e. for each state, sample $M * K_{\text{CEM}}$ trajectories of length T and pick the best one. Report the percentage of success over 50 episodes. How does its performance compare to CEM?

3. (5 pts) Which algorithm (MPC vs. open-loop control) would perform better on what environments? Why? Discuss the pros and cons of MPC.

Problem 1.2: Single probabilistic network (40 pts)

Train a single probabilistic network on transitions from 1000 randomly sampled episodes. The loss that you should use to train each network is the negative log likelihood of the actual resulting state under the predicted distribution of the network. Specifically, Given state transition pairs s_t, a_t, s_{t+1}, we want to minimize the negative log probability of s_{t+1} under the Gaussian distribution $\mathcal{N}(\mu_\theta(s_n, a_n), \Sigma_\theta(s_n, a_n))$, where μ_θ and Σ_θ are outputs of the network. This loss function is also written out explicitly in Section 4 of [1]. You will need to implement this loss in the `get_loss` function in `model.py`.

1. (10 pts) Plot the loss vs. number of iterations trained for the single network.

2. (12 pts) Combine your model with planning using randomly sampled actions. In `mpc.py`, implement `predict_next_state_model` (vectorizing this will make a big difference). Evaluate the performance of your model when planning using a time horizon of 5 and 2000 possible action sequences. Do this by reporting the percent successes on 50 episodes.

3. (12 pts) Combine your model with planning using CEM. Evaluate the performance of your model when planning using a time horizon of 5, a population of 200, 20 elites, and 5 CEM iterations. Do this by reporting the percent successes on 50 episodes.

4. (6 pts) Which planning method performs better, random or CEM? How did MPC using this model perform in general? When is the derived policy able to succeed and when does it fail?

Problem 1.3: MBRL with PETS (35 pts)

(Note that this section will take about 30 minutes on CPU and 15 minutes on GPU to generate the required results)

1. (10 pts) Run Algorithm 1 for 500 iterations (i.e., collect 100 initial episodes and then 500 episodes using MPC). Plot the loss vs. the number of iterations trained.

2. (20 pts) Every 50 iterations, test your model with MPC on 20 episodes and report the percent of successes. Plot this as a function of the number of iterations of PETS.

3. (5 pts) What are some limitations of MBRL? Under which scenarios would you prefer MBRL to policy gradient methods like the ones you implemented in the previous homeworks?

Feedback

Feedback: You can help the course staff improve the course by providing feedback. What was the most confusing part of this homework, and what would have made it less confusing?

Time Spent: How many hours did you spend working on this assignment? Your answer will not affect your grade.

References

[1] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *arXiv preprint arXiv:1805.12114*, 2018.