UNIVERSITY OF PENNSYLVANIA

ESE 650: LEARNING IN ROBOTICS

[04/23] HOMEWORK 4

DUE: 05/10 FRI 11.59 PM ET

NO LATE DAYS FOR THIS HOMEWORK

Changelog: This space will be used to note down updates/errata to the homework problems.

Instructions. Read the following instructions carefully before beginning to work on the homework.

- You will submit solutions typeset in LaTeX on Gradescope (strongly encouraged). You can use hw_template.tex on Canvas in the "Homeworks" folder to do so. If your handwriting is *unambiguously legible*, you can submit PDF scans/tablet-created PDFs.
- Please start a new problem on a fresh page and mark all the pages corresponding to each problem. Failure to do so may result in your work not graded completely.
- Clearly indicate the name and Penn email ID of all your collaborators on your submitted solutions.
- **For each problem in the homework, you should mention the total amount of time you spent on it. This helps us gauge the perceived difficulty of the problems.**
- You can be informal while typesetting the solutions, e.g., if you want to draw a picture feel free to draw it on paper clearly, click a picture and include it in your solution. Do not spend undue time on typesetting solutions.
- You will see an entry of the form "HW 4 PDF" where you will upload the PDF of your solutions. You will also see entries like "HW 4 Problem 1 Code" where you will upload your solution for the respective problems. **For each programming problem, you should create a fresh Python file**. This file should contain all the code to reproduce the results of the problem and you will upload the .py file to Gradescope. If we have installed Autograder for a particular problem, you will use the Autograder.

- **You should include all the relevant plots in the PDF, without doing so you will not get full credit.** You can, for instance, export your Jupyter notebook as a PDF (you can also use text cells to write your solutions) and export the same notebook as a Python file to upload your code.
- **Your PDF solutions should be completely self-contained. We will run the Python file to check if your solution reproduces the results in the PDF.**

Credit. The points for the problems add up to 100. You only need to solve for 100 points to get full credit, i.e., your final score will be $\min(\text{your total points}, 100)$.

Problem 1 (Mujoco Model Predictive Control Library, 25 points (No Autograder)). In this problem, we will learn to use the Mujoco MPC (MJPC) library (https://github.com/google-deepmind/mujoco_mpc). MJPC allows the user to solve surprisingly complex robotics tasks using non-learning based planners, e.g., cross-entropy method, or policy gradients where one samples a lot of trajectories and simply executes the one that obtains the best return in a receding horizon fashion.

(a) [10 points] Use the Installation instructions on Github in the Readme to install mjpc. If you have a Linux or Mac laptop, this should be easy. Windows is also supposed to work according to the original authors. You should read the paper at https://arxiv.org/abs/2212.00541. You should also listen to this seminar by Yuval Tassa who is one of the main authors https://www.youtube.com/watch?v=J-JO-lgaKtw. You do not need VS Code like the Readme says. You can simply do

```
$ git clone https://github.com/google-deepmind/mujoco_mpc.git
$ cd mujoco_mpc
$ mkdir build; cd build;
$ cmake ..
$ make -j
```

After this, you should have a file "build/bin/mjpc" which is the program you will run as "./mjpc" from your terminal.

(b) [10 points] The goal of this problem is to see a lot of different dynamical systems in action and appreciate the power of fast rollouts in the simulators. You should watch the video that we have recorded at
https://drive.google.com/file/d/1gc9nc5LgSYeVYMSDUhS7Y9iHkXP9OwY1/view?usp=drive_link
that describes the different parts of mjpc and how to use the widgets. There are a lot of intricate and cool aspects to the different algorithms implemented in MJPC and I would strongly encourage you to play with this program.

For Cartpole, Acrobot and the Walker, you will create a video just like the video we created above (Zoom screen capture works great...). A sequence of screen shots that convinces us that you were able to run the program and control these three systems is also okay.

(c) [5 points] There is a system named "Humanoid Stand" in MJPC. Try to make the humanoid stand up. You are free to modify any setting within MJPC.

Problem 2 (PPO for a Walker, 75 points (No Autograder)). In this problem, you will use make the Walker walk using reinforcement learning. We will implement PPO.

(a) [5 points] We will be using Deepmind's Control Suite to implement our algorithms. You should install this from https://github.com/google-deepmind/dm_control; you can simply do "pip install dm_control". You might find it useful to glance at https://arxiv.org/abs/2006.12983 to understand what the broad goals of the library are. DM Control is a software package that wraps around the Mujoco simulator (not MJPC). This is useful to implement a number of control and reinforcement learning algorithms.

We have provided you some starter code to set up the environment. For example, the few lines below initialize the Walker, create a task (i.e., a problem) called "walk" (which tells DM Control to give us rewards that correspond to good walking) and uses a random controller to undertake actions. You can run the code to see the Walker in action inside a visualizer.

```
"""
Setup walker environment
"""
r0 = np.random.RandomState(42)
e = suite.load('walker', 'walk',
                   task_kwargs={'random': r0})
U=e.action_spec();udim=U.shape[0];
X=e.observation_spec();xdim=14+1+9;


#Visualize a random controller
def u(dt):
    return np.random.uniform(low=U.minimum,
                             high=U.maximum,
                             size=U.shape)
viewer.launch(e,policy=u)
```

You will next implement rollouts using a neural network-based controller. See the code below to understand how one should sample trajectories from the environment.

```
def rollout(e,uth,T=1000):
    """
    e: environment
    uth: controller
    T: time-steps
    """

    traj=[]
    t=e.reset()
    x=t.observation
    x=np.array(x['orientations'].tolist()+[x['height']]+x['velocity'].tolist())
    for _ in range(T):
        with th.no_grad():
            u,_=uth(th.from_numpy(x).float().unsqueeze(0))
        r = e.step(u.numpy())
        x=r.observation
        xp=np.array(x['orientations'].tolist()+[x['height']]+x['velocity'].tolist())

        t=dict(xp=xp,r=r.reward,u=u,d=r.last())
        traj.append(t)
        x=xp
        if r.last():
            break
    return traj
```

Line 48 samples the control u from a PyTorch-based neural network and runs the simulator one step forward by calling e.step(u). This returns a data structure that contains the state (in this case, the orientations of different joints, height of the center of mass, and velocities of joints). It is a 24 dimensional state. The simulator also returns, the reward and whether this was the last state or not (r.last()). We collect all of returns from successive time steps for implementing the RL algorithm.

Your neural network-based controller should take in all 24 states as input and return 6 controls as the output; check above that udim = 6. We should also build a stochastic controller, so the network will output a 6-dimensional mean and a 6-dimensional standard deviation for a Gaussian controller.

(b) [50 points] It might seem difficult to implement PPO but it is actually quite easy if you proceed systematically. You can refer to the implementation of PPO in Spinning Up https://github.com/openai/spinningup/tree/master/spinup/algos/pytorch/ppo as you begin to work on the problem. This implementation is less than 200 lines of code (if we ignore the boilerplate). You should also read the description of the algorithm in
https://spinningup.openai.com/en/latest/algorithms/ppo.html.

Important: Spinning Up contains some of the best implementations of RL algorithms. But it is quite old and unmaintained now and therefore you will not be able to execute the code. It also uses parallelization to sample more trajectories but we do not need to do so. Use the Spinning Up implementation as a resource to understand how to implement the algorithm, do not simply code from there and expect it to work. We are using DM Control while Spinning Up is using Open AI Gym; the two are very different frameworks, different rewards, slightly different systems, and certainly syntax etc.

There is a minor difference in how we taught PPO in the lecture notes and how it is implemented here. Notice that at each iteration we want to calculate the gradient of

$$\mathop{\mathrm{E}}_{x\sim d^{\theta'}} \mathop{\mathrm{E}}_{u\sim u^{\theta'}} \left[\frac{u_\theta(u \mid x)}{u_{\theta'}(u \mid x)} \, a^{\theta'}(x,u) \right] - \lambda \mathop{\mathrm{E}}_{x\sim d^{\theta'}} \left[\mathrm{KL}(u_\theta(\cdot \mid x), u_{\theta'}(\cdot \mid x)) \right]$$

where the expectation is taken the states and controls along the trajectories in the rollout. In principle, we should fit a value function $q^{\theta'}(x,u)$ and a value function $v^{\theta'}(x)$ to calculate the advantage. But Spinning Up uses a shortcut and sets $q^{\theta'}(x,u)$ to be the return of the trajectory that was rolled out starting from state x and control u. This will be a poor estimate of q but it is certainly valid—and it is quite simple to implement. It also normalizes the returns to come of all the states by subtracting the mean returns of all states in the rollout and dividing by the standard deviation; see Line 81. They fit a critic $v^{\theta'}(x)$ using the Bellman iteration on Line 248.

A suggested order for implementing things:

(1) Implement the neural networks for the controller and the value function. Be careful of conversions from PyTorch to Numpy and dimensions of all quantities (controller takes in a mini-batch of states, outputs a mini-batch of controls along with their log-likelihoods, the value function takes in a mini-batch of states as input and outputs a mini-batch of values). It is important to appreciate that we do need the gradient of the log-likelihood of the control in PPO, so you need to use the PyTorch routines to sample the . See lines 127-132 in core.py for how the control is sampled with th.no_grad and Line 231 for obtaining the controls with the log-likelihood.

(2) Code up the loss function for PPO correctly. Check whether the gradient computed using backward is non-trivial (not NaNs, correct dimensions etc.).

(3) You should train for at least 1 million time-steps. Since the Walker is going to fall down quickly at the beginning, each episode will be short. As the controller learns, the episodes become longer (the code given to you has a maximum length of 1000). So 1 million timesteps amounts to approx. 500 rollouts.

(4) Describe in detail your implementation and any changes/variations that you made in the PDF.

(b) [20 points] Plot the average return of the trajectories sampled during rollout as a function of the number of timesteps. See an example plot at https://spinningup.openai.com/en/latest/spinningup/bench.html. Your numerical values of the returns will be different because DM Control has a different reward than Open AI's Gym. You should include screenshots of the Walker walking in the DM Control viewer in the PDF.

> If you want to first attempt a simpler problem than walking, you can do the task named "stand" for the Walker.