CS224R Spring 2025 Homework 2
Online Reinforcement Learning

Due 5/2/2025

SUNet ID:
Name:
Collaborators:

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Overview

Goals: In the first part of this assignment, you will experiment with different reward functions in MuJoCo MPC (MJPC), an interactive application for real-time predictive control with MuJoCo. You will design a reward function to achieve a desired agent behavior and observe how different reward functions impact agent behavior.

In the second part of the assignment, you will solve a realistic robot task, in which the agent needs to pick up a tool and use it to hammer a nail. You will implement the main parts of an actor-critic algorithm and explore some of the design choices of the algorithm.

Submitting the PDF: Make a PDF report containing: Table 1 for your designed reward function from Question 1.1, your responses to Question 1.2, as well as your training curves from Question 2 and your responses.

Submitting the Code and Experiment Runs: For Problem 2, submit the **ac.py** file and the Logdir folder containing only the two final runs.

Gradescope: Submit both the PDF and the zipped code and experiment runs in the appropriate assignment on Gradescope. **Note: There will be no autograder score for the code/experiment submission, but these will be checked later for completion and are required for credit.**

Use of AI Tools (e.g. ChatGPT, Cursor) For the sake of deeper understanding on implementing actor-critic methods, assistance from generative models to write code for this homework is prohibited. See the course website for more details.

Sample File Submission

For the second part you should submit the **ac.py** file, as well as the logs from your final two runs, which are saved in the Logdir folder. Your submission file should look like this

```
submit.zip
├── ac.py
│
└── Logdir
    ├── run_date_agent.num_critics=2,utd=1
    │
    └── run_date_agent.num_critics=10,utd=5
```

Compute (Important, Please Read)

For this assignment, you will complete all sections on AWS EC2 instances. The CS224R AWS Guide has instructions for setting up and accessing an AWS spot instance.

We recommend only using AWS for compute, as we **do not** support setup on any other platform, such as your own local computer. You may still develop code on your local computer, which we suggest to save compute credits. You may find helpful this guide from CS221N on syncing code and other files between a virtual machine and your computer. (This was written for Azure instead of AWS, but the same principles apply.)

To complete Problem 1, you will need a **c4.4xlarge** instance. Use this link for a custom AMI that has the problem set up (ID ami-041fc055ed329e519).

To complete Problem 2, you will need a **c4.2xlarge** instance. Use this link for a custom AMI that has the problem set up (ID ami-0a2f0af48493553de).

The above AMIs already have the code for each problem, but code for the homework can also be found at `http://cs224r.stanford.edu/material/hw2_starter_code.zip`

We advise you to start as early as possible since the assignment requires longer compute times.

Problem 1: Impact of Reward Functions

To set up Problem 1, log into your AWS **c4.4xlarge** instance for this problem. Code for this problem can be found at `/home/ubuntu/hw2/mujoco_mpc`, and you should follow the instructions in `hw2/mujoco_mpc/README.md`.

Defining a reward function is often the first step of running any RL algorithm, and picking a good reward function in practice can be difficult. In this part of the homework, you will get hands-on experience with designing reward functions. Since iterating on reward functions by running RL on them can be slow, we will be using a faster model-based planner to get behavior from the defined reward functions almost instantaneously. The planner does this by leveraging the simulator itself to optimize for behavior with respect to the reward function.

1. Design a reward function for the Quadruped task such that the agent walks in a clockwise circle (watch `mujoco_mpc/videos/part1.avi` for an example of the desired behavior). The structure of the reward function for the Quadruped task is:

$$r_t = -w_0 \cdot r_{t,\text{height}} - w_1 \cdot r_{t,\text{pos}}(w_2, w_3) + c$$

   where

   - $r_{t,\text{height}}$ is the absolute difference between the agent's torso height over its feet and the target height of 1,
   - $r_{t,\text{pos}}(w_2, w_3)$ is the ℓ^2 norm of the difference between the agent's torso position and the goal position. The goal moves at each time-step according to the desired *walk speed* w_2 and *walk direction* w_3, and
   - c consists of other reward terms for balance, effort, and posture.

   You will design the reward function by choosing values for w_0, w_1, w_2, and w_3, which can be any real number. Here is how you can run the Quadruped task with $w_0 = w_1 = w_2 = w_3 = 0$:

```
./build/bin/mjpc --task="Quadruped Flat" --steps=100 \
    --horizon=0.35 --w0=0.0 --w1=0.0 --w2=0.0 --w3=0.0
```

   The program will run the simulator for the specified number of time-steps and save a video in the `mujoco_mpc/videos/` directory.

   Viewing videos: To view the generated videos, you can `scp` the `mujoco_mpc/videos/` directory to your local machine. We recommend trying multiple reward functions at a time between `scp` commands, which can slow down testing. Another way to view videos is to convert them to MP4 format via `ffmpeg` (e.g., `ffmpeg -i filename.avi filename.mp4`) and opening them with VSCode's Remote SSH plugin.

Note: The "objective" in the top right of the videos indicates the *negative* reward, also referred to as the cost.

Tip: The green sphere is a visualization of where the goal position is.

Fill in Table 1 with the parameters of your reward function.

Parameter	Value
w_0	**Your answer here**
w_1	**Your answer here**
w_2	**Your answer here**
w_3	**Your answer here**

Table 1: Parameters of your reward function.

2. In this next part, you will see how different reward functions impact the agent's behavior in the In-Hand Manipulation task. The structure of the reward function for the Hand task is:

$$r_t = -w_0 \cdot r_{t,\text{cube pos}} - w_1 \cdot r_{t,\text{cube ori}} - w_2 \cdot r_{t,\text{cube vel}} - w_3 \cdot r_{t,\text{actuator}}$$

where

- $r_{t,\text{cube pos}}$ is the ℓ^2 norm of the difference between the cube's position and the hand palm position,
- $r_{t,\text{cube ori}}$ is the ℓ^2 norm of the difference between the cube's orientation and the goal orientation,
- $r_{t,\text{cube vel}}$ is the ℓ^2 norm of the cube's linear velocity, and
- $r_{t,\text{actuator}}$ is the ℓ^2 norm of the control inputs (i.e., the forces applied that cause the robot to move).

For each part below, you will watch the videos located in `mujoco_mpc/videos`. Then, in 1-2 sentences, describe the agent's behavior and why the reward function leads to this behavior.

Note: While the planning horizon (the `--horizon` flag) does have an effect on the planned behavior, you may assume that the visualized behavior is only determined by the reward function in the following parts.

(a) Watch `mujoco_mpc/videos/part2a.avi`, which was generated with the parameters $w_0 = 20$, $w_1 = 3$, $w_2 = 10$, and $w_3 = 0.1$:

```
./build/bin/mjpc --task="Hand" --steps=100 \
    --horizon=2.5 --w0=20.0 --w1=3.0 --w2=10.0 --w3=0.1
```

Tip: The cube on the right is a visualization of the goal orientation.

Describe the agent's behavior and why the reward function leads to this behavior in 1-2 sentences.

(b) Watch `mujoco_mpc/videos/part2b.avi`, which was generated with the parameters $w_0 = 20$, $w_1 = 3$, $w_2 = 10$, and $w_3 = 1$:

```
./build/bin/mjpc --task="Hand" --steps=100 \
    --horizon=0.25 --w0=20.0 --w1=3.0 --w2=10.0 --w3=1.0
```

Describe the agent's behavior and why the reward function leads to this behavior in 1-2 sentences.

(c) Watch `mujoco_mpc/videos/part2c.avi`, which was generated with the parameters $w_0 = 0$, $w_1 = 0$, $w_2 = 0$, and $w_3 = 1$:

```
./build/bin/mjpc --task="Hand" --steps=100 \
    --horizon=2.5 --w0=0.0 --w1=0.0 --w2=0.0 --w3=1.0
```

Hint: An episode terminates when the cube falls out of the hand.

Describe the agent's behavior and why the reward function leads to this behavior in 1-2 sentences.

Problem 2:

To get started log into your AWS **c4.2xlarge** instance for this problem. Code for this homework can be found at `/home/ubuntu/hw2/ac/`. There is a setup script at `hw2/ac/setup.sh`, but you will not need to run this if you use the custom AMI.

In this problem we consider a realistic task in which a Sawyer robot needs to grasp a tool and used it to hammer in a nail. The agent controls a 4-DOF Sawyer robot with a continuous action space and receives environment states as observations. Providing dense rewards for real world problems is difficult, so in this environment the agent receives a sparse reward of 1.0 upon fully completing the task and no intermediate rewards.

We will implement a general actor-critic algorithm that learns to solve the task with almost 100% success rate. At each step the agent processes observations o_t, which consists of the environment states (e.g., pose of the robot gripper and hammer, etc.) from the last two environment steps. The full agent consists of:

- The actor-critic algorithm utilizes critic networks $Q_{\theta^i}(o_t, a_t)$. This is implemented as the **Critic** class in **ac.py**. We use an ensemble of N different critic networks. Using an ensemble of critic networks can help reduce error in the Q-value estimates.
- Each critic network has a corresponding *target* critic network $\bar{Q}_{\theta^i}(o_t, a_t)$, which is used for computing the Bellman targets for critic updates. We use target critic networks, which are updated more slowly, to have a more stable target for learning.
- The final component of the algorithm is an the Actor or policy $\pi_\theta(o_t)$. This is implemented as the **Actor** class in **ac.py**.

You should familiarize yourself with these functions and their inputs and outputs. You should not modify any other files besides **ac.py**.

1. Optimizing behavior in environments with sparse rewards is difficult due to limited reward supervision. To alleviate this, we provide the agent with 20 successful demonstrations, which we will use to pre-train with behavior cloning. In **ac.py** complete the **bc** function of the **ACAgent** class to train the policy using supervised behavior cloning. That is, given state-action pairs o_t, a_t optimize the loss

$$\mathcal{L}_{\pi_\theta, f_\theta}(o_t, a_t) = -\log \pi_\theta(a_t|o_t)$$

   We will also use this to update the actor during later RL training, to improve stability.

2. In the second part, we will try to improve the performance of the policy with additional fine-tuning with reinforcement learning. Reinforcement learning allows the robot to improve using its own experience, without needing additional demonstrations. Your implementation will be in the **ACAgent** class.

   - We begin by implementing the **update_critic** method using the Bellman objective. Consider transitions $(o_t, a_t, r_t, o_{t+1}, \gamma)$ and implement the following steps:
     (a) Sample next state actions from the policy $a'_{t+1} \sim \pi_\theta(o_{t+1})$.
     (b) Compute the Bellman targets

$$y = r_t + \gamma \min\{\bar{Q}_{\theta^i}(o_{t+1}, a'_{t+1}), \bar{Q}_{\theta^j}(o_{t+1}, a'_{t+1})\}$$

     where $\bar{Q}_{\theta^i}$ and $\bar{Q}_{\theta^j}$ are two randomly sampled target critics. We take the minimum over two Q-values to mitigate critic overestimation errors.
     (c) Compute the loss:

$$\mathcal{L}_{Q_\theta, f_\theta} = \sum_{i=1}^{N} (Q_{\theta^i}(o_t, a_t) - \mathbf{sg}(y))^2$$

     where sg stands for the stop gradient operator.
     Note: We compute the loss for all N critic networks.
     (d) Take a gradient step with respect to the critic parameters.
     (e) Update the target critic parameters using exponential moving average.

$$\bar{Q}_{\theta^i} = (1 - \tau)\bar{Q}_{\theta^i} + \tau Q_{\theta^i}$$

     By using an exponential moving average, our target critic parameters are updated more slowly than those of the main critic.
     Note: Check the soft_update_params function in utils.py.

   - Next, we will improve the policy in the **update_actor** method. Sample an action from the actor $a'_t \sim \pi_\theta(o_t)$ and compute the objective that optimizes the actor to maximize the Q-value estimates from the critics:

$$\mathcal{L}_{\pi_\theta} = -\frac{1}{N}\sum_{i=1}^{N} Q_{\theta^i}(o_t, a'_t)$$

Take a gradient step on this objective with respect to the policy only.

- Once you are done, run the RL fine-tuning with

```
python train.py agent.num_critics=2 utd=1
```

To view Tensorboard plots on your AWS instance from your local computer, run the following:

```
ssh -i PATH_TO_AWS_PEM_KEY -L 6006:localhost:6006 -t \
    ubuntu@YOUR_AWS_INSTANCE
cd /home/ubuntu/hw2/ac
micromamba activate AC
tensorboard --logdir Logdir/
```

Attach the plot "eval/episode_success" for this run from Tensorboard. You should achieve a success rate of at least 90% before 100K environment steps.

- In the final part, we will explore some optimization parameter choices. The update-to-data (UTD) ratio stands for the number of critic gradient steps we do, with respect to each environment step. So far we have used only 2 critics and UTD of 1. Repeat the previous part with:

```
python train.py agent.num_critics=10 utd=5
```

Here, we perform critic gradient steps 5x more often. However, too many gradient steps can make learning unstable by increasing error in the critic Q-values.

Attach the plot "eval/episode_success" for this run and compare it to the previous question. Provide an explanation of why we observe these effects.

Note: this is slower to run due to the higher computational cost and you should expect run time of about 2 hours for 50,000 steps.

Common Bugs

- If your critic loss curves remain flat throughout training, make sure the critics are being updated.
- If your critic losses become very large, make sure the dimensions of your critic predictions and targets are correct, and/or tensor broadcasting is done correctly.
- Make sure all critics are updated each iteration, not just those corresponding to the two target critics that are sampled for computing Bellman targets.