CS 234: Assignment #3

Due date: Feb 20, 2026 at 6:00 PM (18:00) PST

These questions require thought but do not require long answers. Please be as concise as possible.

We encourage students to discuss in groups for assignments. **However, each student must finish the problem set and programming assignment individually, and must turn in her/his assignment.** We ask that you abide by the class Honor Code (see the course website), university Honor Code and the Computer Science department Honor Code, and make sure that all of your submitted work is done by yourself. If you have discussed the problems with others, please include a statement saying who you discussed problems with. Failure to follow these instructions will be reported to the Office of Community Standards. We reserve the right to run a fraud-detection software on your code.

Please review any additional instructions posted on the assignment page at http://web.stanford.edu/class/cs234/assignments.html. When you are ready to submit, please follow the instructions on the course website.

Note: We are now requiring students to typeset their homeworks.

Submission guidelines

You will be submitting the following on Gradescope:

1. PDF of this writeup
2. Latex submission .zip file. Do **not** include an enclosing top-level folder; the ZIP file must unpack directly to:

```
main.tex
img/
    hopper.png
    hopper_rlhf.png
    hopper_dpo.png
```

3. Code submission .zip file. See starter code README for instructions.

An introduction to reinforcement learning from human preferences

Reinforcement learning from human preferences (RLHF) was a key tool used in enabling the impressive performance of ChatGPT. However, the concept of RLHF came earlier, and is best known from a seminal paper "Deep reinforcement learning from human preferences." The goal of this assignment is to give you some hands-on experience with RLHF in the context of a robotics task in MuJoCo. You will also get a chance to explore the performance of Direct Preference Optimization (DPO), an alternative to RLHF that allows one to directly learn from preferences without inferring a reward model. You will implement, compare, and contrast several different approaches, including supervised learning (behavior cloning). These methods are all popular approaches used in large language model training, and many other machine learning tasks.

Environment setup:
Please see the starter code README for setting up the environment for this *entire* assignment!

1 Reward engineering (13 pts writeup)

In Assignment 2 you applied PPO to solve an environment with a provided reward function. The process of deriving a reward function for a specific task is called reward engineering. Each question in this problem shall be answered **concisely** with a few sentences.

1.1 Written Questions

(a) Why is reward engineering usually hard? What are potential risks that come with specifying an incorrect reward function? Provide an example of a problem and a reward function that appears to be adequate but may have unintended consequences.

Answer:
Reward engineering is hard because requires translating human goals and aspects of desired behavior into mathematical functions, wich is sometimes difficult to formulate.
Some risks of specifying an incorrect reward function include reward hacking, where the agent fins unintended ways to mazimize reward that don't align with the true objective, unwanted side effects that not being penalized and may cause harm in achieving the goal, or rewards that work well in training, but fail in deployment.
For example, rewarding a cleaning robot only for the amount of dirt collected can incentivice it to spill dirt first, then clean it, damage furniture to access dirt that is hard to reach, or act noisy, dangerously, etc.

(b) Read the description of the Hopper environment. Using your own words, describe the goal of the environment, and how each term of the reward functions contributes to encourage the agent to achieve it.

Answer:
Goal: The Hopper is a one-legged robot that should learn to hop forward as fast as possible while maintaining balance.
Reward function: $r = r_{\text{healthy}} + r_{\text{forward}} - r_{\text{ctrl}}$
where

- **Healthy reward** ($r_{\text{healthy}} = 1$): constant reward given at each timestep when the hopper is in a "healthy" state, to encourage the agent to not fall.
- **Forward reward** ($r_{\text{forward}} = v_x \cdot \text{forward\_reward\_weight}$): proportional to the hopper's forward velocity, which incentivizes the agent to move forward as fast as possible.
- **Control cost** ($r_{\text{ctrl}} = \text{ctrl\_cost\_weight} \cdot \|\text{action}\|_2^2$): to discourage the agent from taking large control actions, promoting smooth movements.

(c) By default, the episode terminates when the agent leaves the set of "healthy" states. What do these "healthy" states mean? Name one advantage and one disadvantage of this early termination.

Answer:
The healthy states are defined by the following conditions:

- Observations are contained in the interval $[-100, 100]$
- The hopper's height is within $[0.7, +\infty)$ meters
- The torso angle is within $[-0.2, 0.2]$ radians

These constraints ensure the hopper remains upright and doesn't fallen over.
Advantage: speeds up training by not wasting time simulating states where the robot has already fallen and cannot recover.
Disadvantage: can bias the learned policy by preventing the agent from learning how to recover from near-failure states, never learning to operate near the boundaries of the healthy region.

1.2 Code-Related Questions

(d) Use the provided starter code to train a policy using PPO to solve the Hopper environment for 3 different seeds. Do this with and without early termination. **Each seed can take up to 90 minutes to run so please start this early!**

```
python ppo_hopper.py [--early-termination] --seed SEED
```

Attach here the plot of the episodic returns along training, with and without early termination. You can generate the plot by running

```
python plot.py --directory results --seeds 1,2,3 --output results/hopper.png
```

where `SEEDS` is a comma-separated list of the seeds you used. Also include the plot in your Latex submission folder as `hopper.png` (in your `img/` folder). Comment on the performance in terms of training epochs and wall time. Is the standard error in the average returns high or low? How could you obtain a better estimate of the average return on Hopper achieved by a policy optimized with PPO?

Answer:

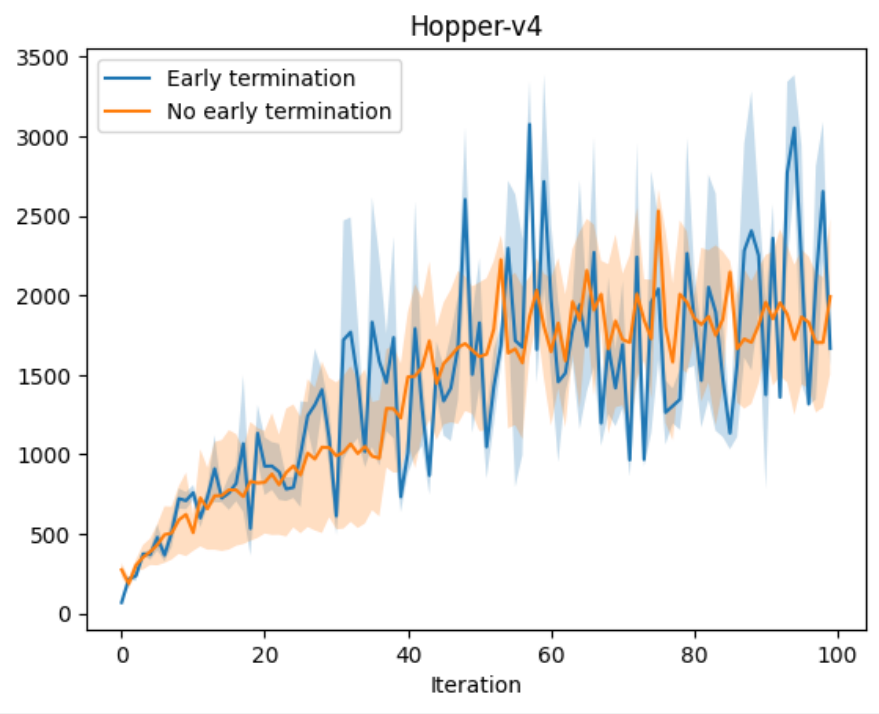

Figure 1: PPO training on Hopper with and without early termination across 3 seeds.

Performance in terms of training epochs and wall time:

- **With early termination**: Wall time ≈ 55 minutes per seed. Each epoch completes faster because episodes terminate early when the agent falls, allowing more episodes to be collected per unit time. The agent reaches good performance earlier in training (fewer epochs needed to learn balance).

- **Without early termination**: Wall time ≈ 55–65 minutes per seed. Episodes run to full length even after falling, so fewer episodes are collected per epoch. However, the agent eventually achieves higher final returns (mean 1991.77 vs 1666.04) as it learns from the complete state space.

The standard error is low for early termination (Std = 150.76) and high without early termination (Std = 690.03). The high variance without early termination suggests that some seeds find good policies while others get stuck, indicating sensitivity to initialization and exploration. To obtain a better estimate of the average return, we could:

- Use more random seeds to reduce variance in the estimate
- Run more evaluation episodes per checkpoint

(e) Pick one of the trained policies and render a video of an evaluation rollout.

```
# on linux
MUJOCO_GL=egl python render.py --checkpoint [PATH TO MODEL CHECKPOINT]

# on mac/windows
python render.py --checkpoint [PATH TO MODEL CHECKPOINT]
```

Does the agent successfully complete the assigned task? Does it complete it in the way you would expect it to, or are you surprised by the agent behavior?

Answer:
Inspecting *Hopper-v4-early-termination=False-seed=1*, I observe that the agent does move forward at a relatively fast pace, but not in the expected manner. Instead, it appears to drag itself along the floor, occasionally making contact with it, before recovering just enough to continue progressing. At first glance, this behavior was unexpected; however, the absence of early termination may explain it, as the agent can learn policies that operate near the boundary of the healthy state constraints while still remaining valid.

(f) Render another video with another policy. How do the two rollouts compare? Do you prefer one over the other?

Answer:
Comparing the policy from `Hopper-v4-early-termination=True-seed=1` with the previous one, the two rollouts exhibit different hopping styles: the early-termination policy displays a more "natural" hopping gait but eventually falls and fails to recover, while the non-early-termination policy drags itself along the floor in an unconventional manner but maintains faster forward progress. I prefer the early-termination policy aesthetically since it resembles natural hopping, but the non-early-termination policy actually achieves better forward velocity according to the reward function, demonstrating that multiple valid solutions can emerge with qualitatively different behaviors.

2 Learning from preferences (19 pts writeup + 8 pts coding)

In the previous part you trained multiple policies from scratch and compared them at the end of training. In this section, we will see how we can use human preferences on two roll-outs to learn a reward function. We will follow the framework proposed by Deep Reinforcement Learning from Human Preferences. A reward function $r : \mathcal{O} \times \mathcal{A} \to \mathbb{R}$ defines a preference relation $\succ$ if for all trajectories $\sigma^i = (o_t^i, a_t^i)_{t=0,\ldots,T}$ we have that

$$\left(\left(o_0^1, a_0^1\right), \ldots, \left(o_T^1, a_T^1\right)\right) \succ \left(\left(o_0^2, a_0^2\right), \ldots, \left(o_T^2, a_T^2\right)\right)$$

whenever

$$r\left(o_0^1, a_0^1\right) + \cdots + r\left(o_T^1, a_T^1\right) > r\left(o_0^2, a_0^2\right) + \cdots + r\left(o_T^2, a_T^2\right).$$

Following the Bradley-Terry preference model, we can calculate the probability of one trajectory σ^1 being preferred over σ^2 as follows:

$$\hat{P}\left[\sigma^1 \succ \sigma^2\right] = \frac{\exp \sum \hat{r}\left(o_t^1, a_t^1\right)}{\exp \sum \hat{r}\left(o_t^1, a_t^1\right) + \exp \sum \hat{r}\left(o_t^2, a_t^2\right)},$$

where $\hat{r}$ is an estimate of the reward for a state-action pair. This is similar to a classification problem, and we can fit a function approximator to $\hat{r}$ by minimizing the cross-entropy loss between the values predicted with the above formula and ground truth human preference labels μ.

$$\mathcal{L}(\hat{r}) = -\sum_{(\sigma^1, \sigma^2, \mu) \in \mathcal{D}} \mu \log \hat{P}\left[\sigma^1 \succ \sigma^2\right] + (1-\mu) \log\Big(1 - \hat{P}\left[\sigma^1 \succ \sigma^2\right]\Big).$$

Once we have learned the reward function[1], we can apply any policy optimization algorithm (such as PPO) to maximize the returns of a model under it.

2.1 Written questions

We parameterize $\hat{r}(o, a)$ with a neural network θ, i.e. $\hat{r}_\theta(o, a)$. Now we want to derive $\nabla_\theta \mathcal{L}(\hat{r})$, the gradient of the loss w.r.t. θ.

(a) First write $\nabla_\theta \log \hat{P}[\sigma^1 \succ \sigma^2]$ as a function of $\nabla_\theta \hat{r}_\theta(o_i^j, a_i^j)$ where $i = 0, ..., T$ and $j = 1, 2$. You may find it useful to rewrite $\hat{P}$ as $\sigma(z_\theta)$ where σ is sigmoid and z_θ is some function of θ.

Answer:
Define cumulative rewards:

$$R^1 = \sum_{i=0}^{T} \hat{r}_\theta(o_i^1, a_i^1), \quad R^2 = \sum_{i=0}^{T} \hat{r}_\theta(o_i^2, a_i^2)$$

Then we can write:

$$\hat{P}[\sigma^1 \succ \sigma^2] = \frac{e^{R^1}}{e^{R^1} + e^{R^2}} = \frac{1}{1 + e^{-(R^1 - R^2)}} = \sigma(z_\theta)$$

where $z_\theta = R^1 - R^2$.
Taking the log:

$$\log \hat{P}[\sigma^1 \succ \sigma^2] = \log \sigma(z_\theta)$$

[1] Recent work on RLHF for reinforcement learning suggests that the pairwise feedback provided by humans on partial trajectories may be more consistent with regret, and that the learned reward function may be better viewed as an advantage function. See Knox et al. AAAI 2024 "Learning optimal advantage from preferences and mistaking it for reward." https://openreview.net/forum?id=euZXhbTmQ7

Using the chain rule and the derivative of log-sigmoid: $\frac{d}{dz}\log\sigma(z) = 1 - \sigma(z)$:

$$\nabla_\theta \log \hat{P}[\sigma^1 \succ \sigma^2] = (1 - \sigma(z_\theta))\nabla_\theta z_\theta$$

Since $\nabla_\theta z_\theta = \nabla_\theta R^1 - \nabla_\theta R^2 = \sum_{i=0}^{T} \nabla_\theta \hat{r}_\theta(o_i^1, a_i^1) - \sum_{i=0}^{T} \nabla_\theta \hat{r}_\theta(o_i^2, a_i^2)$:

$$\nabla_\theta \log \hat{P}[\sigma^1 \succ \sigma^2] = \left(1 - \hat{P}[\sigma^1 \succ \sigma^2]\right)\left(\sum_{i=0}^{T} \nabla_\theta \hat{r}_\theta(o_i^1, a_i^1) - \sum_{i=0}^{T} \nabla_\theta \hat{r}_\theta(o_i^2, a_i^2)\right)$$

(b) Then write $\nabla_\theta \mathcal{L}(\hat{r})$ as a function of $\nabla_\theta \hat{r}_\theta(o_i^j, a_i^j)$ where $i = 0, ..., T$ and $j = 1, 2$.

Answer:
The loss is:

$$\mathcal{L}(\hat{r}) = -\sum_{(\sigma^1,\sigma^2,\mu)\in\mathcal{D}} \mu \log \hat{P}[\sigma^1 \succ \sigma^2] + (1-\mu)\log\Big(1 - \hat{P}[\sigma^1 \succ \sigma^2]\Big)$$

Taking the gradient:

$$\nabla_\theta \mathcal{L}(\hat{r}) = -\sum_{(\sigma^1,\sigma^2,\mu)\in\mathcal{D}} \mu\nabla_\theta \log \hat{P}[\sigma^1 \succ \sigma^2] + (1-\mu)\nabla_\theta \log\Big(1 - \hat{P}[\sigma^1 \succ \sigma^2]\Big)$$

From part (a): $\nabla_\theta \log \hat{P}[\sigma^1 \succ \sigma^2] = (1 - \hat{P}[\sigma^1 \succ \sigma^2])\nabla_\theta z_\theta$
For the second term, using $1 - \hat{P}[\sigma^1 \succ \sigma^2] = \sigma(-z_\theta)$:

$$\nabla_\theta \log\Big(1 - \hat{P}[\sigma^1 \succ \sigma^2]\Big) = \nabla_\theta \log \sigma(-z_\theta) = (1 - \sigma(-z_\theta))(-\nabla_\theta z_\theta) = -\hat{P}[\sigma^1 \succ \sigma^2]\nabla_\theta z_\theta$$

Substituting and simplifying (where $P = \hat{P}[\sigma^1 \succ \sigma^2]$):

$$\begin{aligned}
\nabla_\theta \mathcal{L}(\hat{r}) &= -\sum_{(\sigma^1,\sigma^2,\mu)} [\mu(1-P)\nabla_\theta z_\theta - (1-\mu)P\nabla_\theta z_\theta] \\
&= -\sum_{(\sigma^1,\sigma^2,\mu)} (\mu - \mu P - P + \mu P)\nabla_\theta z_\theta \\
&= -\sum_{(\sigma^1,\sigma^2,\mu)} (\mu - P)\nabla_\theta z_\theta
\end{aligned}$$

Since $\nabla_\theta z_\theta = \sum_{i=0}^T \nabla_\theta \hat{r}_\theta(o_i^1, a_i^1) - \sum_{i=0}^T \nabla_\theta \hat{r}_\theta(o_i^2, a_i^2)$:

$$\nabla_\theta \mathcal{L}(\hat{r}) = \sum_{(\sigma^1,\sigma^2,\mu)\in\mathcal{D}} \Big(\hat{P}[\sigma^1 \succ \sigma^2] - \mu\Big)\left(\sum_{i=0}^T \nabla_\theta \hat{r}_\theta(o_i^1, a_i^1) - \sum_{i=0}^T \nabla_\theta \hat{r}_\theta(o_i^2, a_i^2)\right)$$

2.2 Coding questions

In this problem we are trying to solve the same task as in the previous part, but this time we will learn a reward function from a dataset of preferences rather than manually specifying a reward function.

(c) You can load a sample from the provided **long preferences** dataset and render a video of the two trajectories using the following command

```
# on linux
MUJOCO_GL=egl python render.py --dataset data/long-prefs-hopper.npz --idx IDX

# on mac/windows
python render.py --dataset data/long-prefs-hopper.npz --idx IDX
```

where `IDX` is an index into the preference dataset (if ommitted a sequence will be chosen at random). Bear in mind that each sequence in the dataset has 200 timesteps which results in an 8-second video.

Load 5 different samples from the dataset. For each, take note of which sequence was labeled as preferred (for the coming parts it is helpful to know that 0 means the first sequence was preferred, 1 means the second one, and 0.5 means neither is preferred over the other). Note: you may see that the Hopper terminates / freezes even though it is upright (there is a narrow healthy angle range for the Hopper's torso); assume in this situation that the robot has reached an "unhealthy" state, so it rightfully terminated. Do you agree with the labels (that is, if shown the two trajectories, would you have ranked them the same way they appear in the dataset, knowing that we are trying to solve the Hopper environment)?

From your answers, how often do you estimate you agree with whoever ranked the trajectories? Based on this estimate, would you trust a reward function learned on this data?

Answer:
After viewing 5 different samples from the long preferences dataset, I can say that I agree with 4 out of the 5 labels. For sample with IDX = 1, I preferred the one labeled as "Not Preferred". I guess my estimated agreement rate is approximately 80%. I would trust in the reward function on this data, but with caveats. This high agreement suggests that the labels capture reasonable human intuitions about good hopping behavior, tough some edge cases may still seem ambiguous.

(d) Implement the functions in the `RewardModel` class (`run_rlhf.py`), which is responsible for learning a reward function from preference data.

(e) Train a model using PPO and the learned reward function with 3 different random seeds. Plot the average returns for both the original reward function and the learned reward function and include it in your response and your LaTeX source submission folder as `hopper_rlhf.png` (in the `img/` folder).

```
# run rlhf
# Note: You may want to look at the original_scores.png
# and learned_scores.png generated in the output folder
python run_rlhf.py --seed SEED

# plot
python plot.py --rlhf-directory results_rlhf \
    --output results_rlhf/hopper_rlhf.png --seeds 1,2,3
```

Do the two correlate?

Answer:

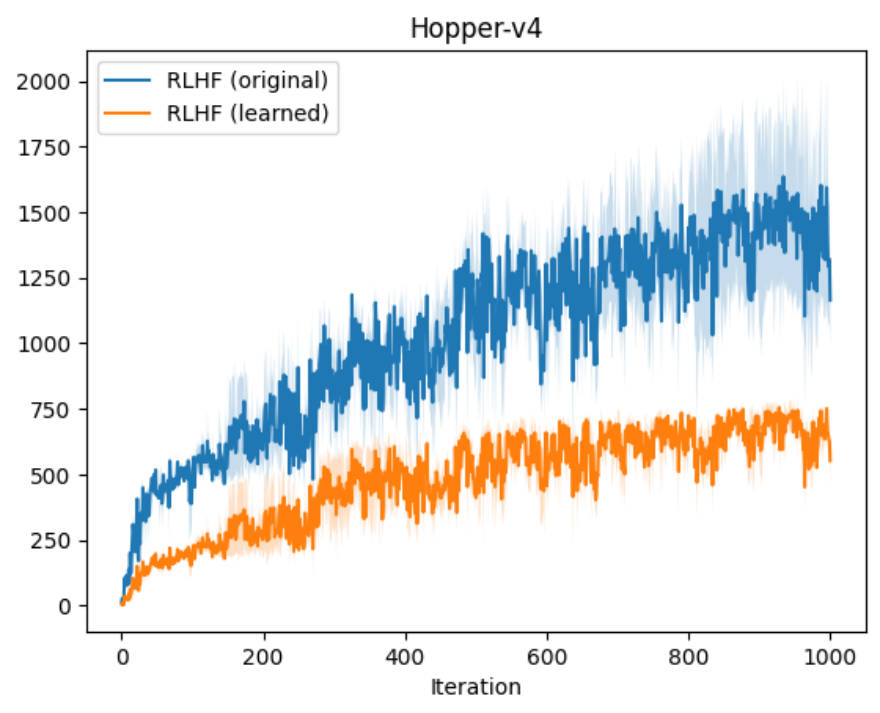

Figure 2: RLHF training on Hopper: original reward vs learned reward across 3 seeds.

Yes, the two strongly correlate. Looking at the training curves, when the learned reward goes up, the original reward also goes up. For example, in seed 1, the agent reaches a final original return of 1495.20 and learned return of 436.41, the absolute values differ because the learned reward is scaled to $[0, 1]$, but both increase together throughout training. This means our learned reward function successfully captures what "good hopping" looks like: optimizing for the learned reward also improves performance on the true reward, even though the reward

model never saw the original reward function directly.

(f) Given enough preference pairs sampled under the Bradley-Terry model, can we recover the original reward function they were derived from?

Answer:
No. The Bradley-Terry model only compares differences in cumulative rewards between trajectories, so we cannot recover the exact original reward. For example, adding a constant to all rewards ($r' = r + c$) or scaling them ($r' = 2r$) would produce the same preferences. However, we can recover a reward function that leads to the same optimal policy, which is what matters in practice.

(g) Pick one of the policies and render a video of the agent behavior at the end of training.

```
# on linux
MUJOCO_GL=egl python render.py --checkpoint [PATH TO MODEL CHECKPOINT]

# on mac/windows
python render.py --checkpoint [PATH TO MODEL CHECKPOINT]
```

How does it compare to the behavior of the policies trained with the ground truth reward in problem 1? How does it compare to the demonstrations you've seen from the dataset?

Answer:
The behavior of the RLHF-trained agent demonstrates smoother and more stable hopping, better than results from problem 1 (both types). In particular, *Hopper-v4-rlhf-seed=1* maintains a much better balance, though at the cost of reduced forward velocity.
Compared againts the dataset demonstrations, I can say that they seem like the preferred sequence, though these new results do not freeze, and are more complete.
Overall, RLHF successfully learns a policy that solves the hopping task, validating that preferences can be used as an effective alternative to hand-crafted reward functions.

3 Direct preference optimization (6 pts writeup + 19 pts coding)

In the previous question we saw how we could train a model based on preference data. In general you may be given access to a pre-trained model and the corresponding preference data. Learning a reward model and then optimizing a policy for that reward model can have some limitations. An alternative is Direct Preference Optimization (DPO): directly optimize a policy using the preference data, without learning a reward model. DPO in its original form is focused on bandit problems, and proposes optimizing the policy using the following loss:

$$\mathcal{L}_{\text{DPO}}\left(\pi_\theta; \pi_{\text{ref}}\right) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta\left(y_w \mid x\right)}{\pi_{\text{ref}}\left(y_w \mid x\right)} - \beta\log\frac{\pi_\theta\left(y_l \mid x\right)}{\pi_{\text{ref}}\left(y_l \mid x\right)}\right)\right],$$

.

where x is the context/state, and y_w and y_l are two actions (in LLM terms, responses) sampled from the reference policy π_{ref}. y_w is the response/action that was preferred (the "winning" response/action) and y_l is the other ("losing") response. π_θ is the policy to be learned, and σ is the sigmoid function.
In this part of the assignment you will get to use DPO for learning policies for the same MuJoCo task. While DPO is designed for bandit problems, we will use a simple adaptation of it to handle our RL setting.
First, to provide some context, consider the general approach for RLHF for text generation:

1. Train a large language model (LLM) to do next token prediction given a context (the tokens that came previously).
2. Given a fixed context x, generate possible next token sequence predictions y_1 and y_2, and store the triple (x, y_1, y_2).
3. Ask human supervisors to rank y_1 and y_2 given x according to individual preference.
4. Update the LLM to maximize the probability of giving the preferred answers using reinforcement learning.

In a similar way, given an observation x we could have two ranked sequences of actions $a^1_{1:T}$ and $a^2_{1:T}$, train the model to generate the preferred sequence of actions, and then execute them all[2]. If the length of the generated action sequence is equal to the environment time horizon, this is called open-loop control. However, this approach lacks robustness, since the plan of actions will not change in response to disturbances or compounding errors. Instead we will use the common approach of receding horizon control (often also called model predictive control), where a multi-step action plan is computed, the first action in that plan is taken, and then a new action plan is computed. MPC/RHC is well known to improve performances, since it allows the agent to react to disturbances.

Note that there are other algorithms that directly tackle learning from preferences in the multi-step RL setting, such as later work on Contrastive Preference Learning (CPL). Indeed CPL showed that DPO can be viewed as a special case of their setting, for the bandit setting. In this homework we focus on DPO because it is widely used in LLM training, and is a simpler setting which still provides key insights into the difference between RLHF and learning policies directly from preferences.

3.1 Coding questions

In this coding question you will need to modify the `run_dpo.py` file. You do not need to modify any other files.

[2]To understand why we are considering sequences of actions rather than a single action for the next time, recall that 50 actions corresponded to 2 seconds of video. If you found it difficult to rank a sequence of 50 actions based on such a short video, imagine ranking the effect of a single action!

(a) Implement the `ActionSequenceModel` class instance methods. When called, the model should return a probability distribution for the actions over the number of next time steps specified at initialization. Use a multivariate normal distribution for each action, with mean and standard deviation predicted by a neural network (see the starter code for more details).[3]

(b) Implement the `update` method of the `SFT` class. This class will be used to pre-train a policy on the preference data by maximizing the log probabilities of the preferred actions given the observations in the dataset.

(c) Implement the `update` method of the `DPO` class. This should minimize the DPO loss described above.

(d) Run SFT and DPO for 3 different random seeds each, and plot the evolution of returns over time.

```
# run SFT
python run_dpo.py --seed SEED --algo sft

# run DPO
python run_dpo.py --seed SEED --algo dpo

# plot
python plot.py --dpo-directory results_dpo \
    --output results_dpo/hopper_dpo.png --seeds 1,2,3
```

Include that plot in your response and your LaTeX source submission folder as `hopper_dpo.png` (in the `img/` folder). How does it compare to the returns achieved using RLHF? Comment on the pros and cons of each method applied to this specific example.

Answer:

[3] We have prepared a notebook to illustrate the behavior of `torch.distributions.Independent`.

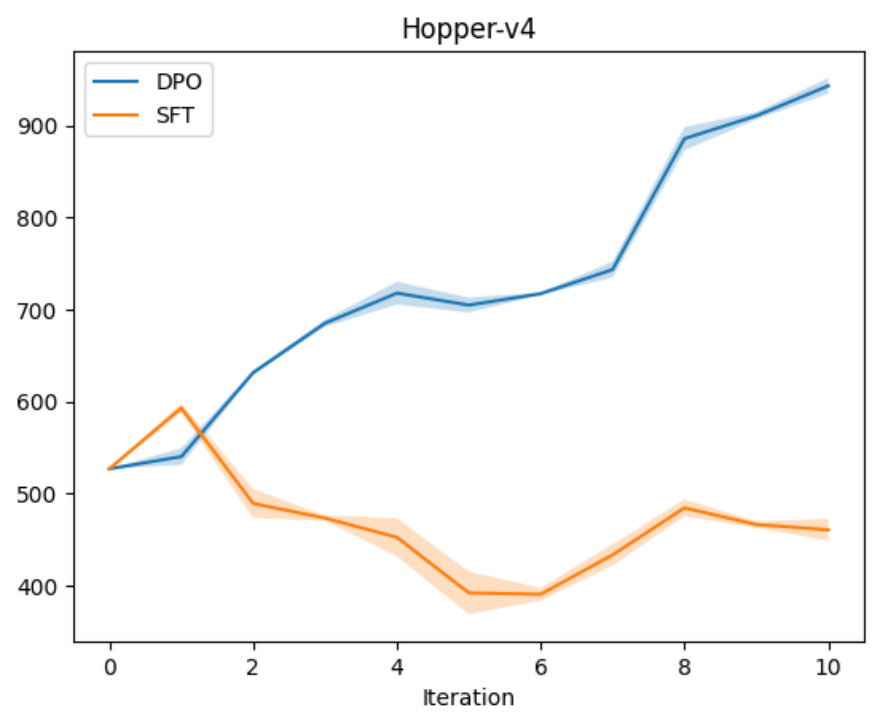

Figure 3: SFT and DPO training on Hopper across 3 seeds.

Results: DPO significantly outperforms SFT. SFT achieves final returns of ∼440–480, while DPO achieves ∼930–960 (roughly 2x better). For comparison, RLHF achieved ∼930–1495.
Comparison with RLHF: DPO reaches similar performance to RLHF on some seeds, but RLHF has higher potential (seed 1 reached 1495). This is because RLHF uses online RL to actively explore, while DPO is limited to offline data.
Pros and cons:

- **RLHF**: Can achieve higher returns through online exploration, but requires training a separate reward model and is more computationally expensive.
- **DPO**: Simpler (no reward model needed), more stable training, but limited to behaviors in the dataset and cannot discover new strategies.
- **SFT**: Simplest approach (just imitation), but ignores preference information and performs worst.

(e) Take the best DPO training run and render videos of episodes generated by the SFT policy and the DPO policy. The following command will render 10 episodes from SFT DPO side-by-side. The left one is from SFT policy while the right one is from the DPO policy.

```
# Note: --checkpoint argument should be the path to the dpo.pt the script will
# find a sft.pt with the same seed following our default naming convention

# on linux
MUJOCO_GL=egl python render.py --dpo --checkpoint [PATH TO DPO CHECKPOINT]

# on mac/windows
python render.py --dpo --checkpoint [PATH TO DPO CHECKPOINT]
```

How do they compare? Note that both of them may not look great because they are only trained on a small amount of offline data and have not interacted with the environment during training. But you may still observe that one is slightly better than the other in some videos.

Answer:
The best DPO training run was *Hopper-v4-dpo-seed=2*. For that one, DPO (right video) seemed to perform better than SFT. Even if none look great, DPO still manages to maintain balance for a couple more seconds than SFT (which falls down pretty quickly), and even when DPO falls, it still manages to keep moving forward by dragging itself, almost seeming like it wants to get up as well.

4 Best Arm Identification in Multi-armed Bandit (25 pts writeup)

In many experimental settings we are interested in quickly identifying the "best" of a set of potential interventions, such as finding the best of a set of experimental drugs at treating cancer, or the website design that maximizes user subscriptions. Here we may be interested in efficient pure exploration, seeking to quickly identify the best arm for future use.

In this problem, we bound how many samples may be needed to find the best or near-optimal intervention. We frame this as a multi-armed bandit with rewards bounded in $[0, 1]$. Recall a bandit problem can be considered as a finite-horizon MDP with just one state ($|\mathcal{S}| = 1$) and horizon 1: each episode consists of taking a single action and observing a reward. In the bandit setting – unlike in standard RL – the action (or "arm") taken does not affect the distribution of future states. We assume a simple multi-armed bandit, meaning that $1 < |\mathcal{A}| < \infty$. Since there is only one state, a policy is simply a distribution over actions. There are exactly $|\mathcal{A}|$ different deterministic policies. Your goal is to design a simple algorithm to identify a near-optimal arm with high probability.

We recall Hoeffding's inequality: if $X_1, \ldots, X_n$ are i.i.d. random variables satisfying $0 \leq X_i \leq 1$ with probability 1 for all i, $\overline{X} = \mathbb{E}[X_1] = \cdots = \mathbb{E}[X_n]$ is the expected value of the random variables, and $\widehat{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ is the sample mean, then for any $\delta > 0$ we have

$$\Pr\left(|\widehat{X} - \overline{X}| > \sqrt{\frac{\log(2/\delta)}{2n}}\right) < \delta. \tag{1}$$

Assuming that the rewards are bounded in $[0, 1]$, we propose this simple strategy: pull each arm n_e times, and return the action with the highest average payout $\widehat{r}_a$. The purpose of this exercise is to study the number of samples required to output an arm that is at least ϵ-optimal with high probability. Intuitively, as n_e increases the empirical average of the payout $\widehat{r}_a$ converges to its expected value $\overline{r}_a$ for every action a, and so choosing the arm with the highest empirical payout $\widehat{r}_a$ corresponds to approximately choosing the arm with the highest expected payout $\overline{r}_a$.

(a) We start by bounding the probability of the "bad event" in which the empirical mean of some arm differs significantly from its expected return. Starting from Hoeffding's inequality with n_e samples allocated to every action, show that:

$$\Pr\left(\exists a \in \mathcal{A} \quad \text{s.t.} \quad |\widehat{r}_a - \overline{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}}\right) < |\mathcal{A}|\delta. \tag{2}$$

Note that, depending on your derivation, you may come up with a tighter upper bound than $|\mathcal{A}|\delta$. This is also acceptable (as long as you argue that your bound is tighter), but showing the inequality above is sufficient.

Answer:

Let the "bad event" for arm a be:

$$B_a = \left\{|\widehat{r}_a - \overline{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}}\right\}$$

By Hoeffding's inequality, for each arm a with n_e samples:

$$\Pr(B_a) = \Pr\left(|\widehat{r}_a - \overline{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}}\right) < \delta$$

The event tjat some arm has a large deviation, "$\exists a \in \mathcal{A}$ s.t. $|\widehat{r}_a - \overline{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}}$", is exactly

$\bigcup_{a \in \mathcal{A}} B_a$.
Applying the union bound:

$$\Pr\left(\bigcup_{a \in \mathcal{A}} B_a\right) \leq \sum_{a \in \mathcal{A}} \Pr(B_a) < \sum_{a \in \mathcal{A}} \delta = |\mathcal{A}| \cdot \delta$$

Therefore:

$$\Pr\left(\exists a \in \mathcal{A} \text{ s.t. } |\widehat{r}_a - \overline{r}_a| > \sqrt{\frac{\log(2/\delta)}{2n_e}}\right) < |\mathcal{A}|\delta$$

(b) After pulling each arm (action) n_e times our algorithm returns the arm with the highest empirical mean:

$$a^\dagger = \arg\max_a \widehat{r}_a \tag{3}$$

Notice that $a^\dagger$ is a random variable. Let $a^\star = \arg\max_a \overline{r}_a$ be the true optimal arm. Suppose that we want our algorithm to return at least an ϵ-optimal arm with probability at least $1 - \delta'$, as follows:

$$\Pr\left(\overline{r}_{a^\dagger} \geq \overline{r}_{a^\star} - \epsilon\right) \geq 1 - \delta'. \tag{4}$$

How accurately do we need to estimate each arm in order to pick an arm that is ϵ-optimal? Then derive how many total samples we need total (across all arms) to return an ϵ-optimal arm with prob at least 1- δ' (that satisfies Equation 4). Express your result as a function of the number of actions, the required precision ϵ and the failure probability δ'.

Answer:
To guarantee ϵ-optimality, we need each estimate to be within $\frac{\epsilon}{2}$ of its true mean.
Suppose $|\widehat{r}_a - \overline{r}_a| \leq \frac{\epsilon}{2}$ for all a (good event). Then:

$$\begin{aligned}
\overline{r}_{a^\dagger} &\geq \widehat{r}_{a^\dagger} - \frac{\epsilon}{2} && \text{(by accuracy bound)} \\
&\geq \widehat{r}_{a^\star} - \frac{\epsilon}{2} && \text{(since } a^\dagger = \arg\max_a \widehat{r}_a\text{)} \\
&\geq \overline{r}_{a^\star} - \frac{\epsilon}{2} - \frac{\epsilon}{2} && \text{(by accuracy bound)} \\
\overline{r}_{a^\dagger} &\geq \overline{r}_{a^\star} - \epsilon
\end{aligned}$$

So $\overline{r}_{a^\dagger}$ is ϵ-optimal whenever all arms are estimated within $\epsilon/2$
From part (a), we need:

$$\sqrt{\frac{\log(2/\delta)}{2n_e}} \leq \frac{\epsilon}{2}$$

Solving for n_e:

$$n_e \geq \frac{2\log(2/\delta)}{\epsilon^2}$$

To ensure the "good event" (all estimates accurate) holds with probability $\geq 1 - \delta'$, we set $|\mathcal{A}|\delta = \delta'$, giving $\delta = \frac{\delta'}{|\mathcal{A}|}$.
Substituting:

$$n_e \geq \frac{2\log(2|\mathcal{A}|/\delta')}{\epsilon^2}$$

Therefore, the total samples across all $|\mathcal{A}|$ arms is:

$$N_{\text{total}} = |\mathcal{A}|n_e \geq \frac{2|\mathcal{A}|\log(2|\mathcal{A}|/\delta')}{\epsilon^2}$$

(c) (Optional challenge, will not be graded) The above derivation only assumed the outcomes were bounded between 0 and 1. In practice people often assume outcomes are drawn from a parametric distribution, and under mild assumptions, one can use the central limit theorem to assume the average outcomes for an arm will follow a normal distribution. Repeat the above analysis under this assumption, for a multi-armed bandit with two arms. Is the resulting number of samples significantly smaller under these assumptions? In real settings it is often very expensive to run experiments. Do you think the method and bound derived in (a-b) would be preferable to making a normal assumption and why or why not?

Answer:

5 Stated vs. Revealed Preferences (4 pts writeup)

Context: You are designing a reinforcement learning algorithm to power a news recommendation app. The app has two types of data about users:

- User Profiles: Users specify what news topics they are interested in reading about, and their preferred format (videos, podcasts, etc).
- Interaction data: what news articles the user lingered on, how long they lingered on it, and if and who they shared articles/podcasts.

Additionally, the app has metadata about users, such as:

- Location, browser type, etc.
- Engagement Metrics: Frequency of app usage, average time spent on the app
- Revenue Potential: Whether they are paying users and their likelihood to subscribe to premium services.

(a) Which of the given data about users represent the stated preferences of the user, and which represents the revealed preferences?

Answer: **Stated preferences** (what users explicitly say they want):

- User Profiles: topics users say they are interested in
- User Profiles: preferred format (videos, podcasts, etc.)

Revealed preferences (what users' actions demonstrate they want):

- Interaction data: which articles users lingered on
- Interaction data: how long they lingered
- Interaction data: what they shared and with whom
- Engagement metrics: frequency of app usage
- Engagement metrics: average time spent on app

Neither stated nor revealed preferences (metadata about users, not preferences):

- Location, browser type
- Revenue potential / subscription status

(b) What reward function might a company pick?

Answer: A company might pick a reward function that combines multiple objectives:

$$r = w_1 \cdot \text{engagement} + w_2 \cdot \text{retention} + w_3 \cdot \text{revenue}$$

Specific components could include:

- **Click-through rate**: Did the user click on the recommended article?
- **Dwell time**: How long did the user spend reading/watching?
- **Session length**: Total time spent on app per session
- **Return visits**: Probability user returns within 24 hours
- **Shares**: Did the user share content (viral growth)?
- **Subscription conversion**: Did the user upgrade to premium?
- **Ad revenue**: Revenue generated from ads shown during session

Companies often prioritize revealed preferences (engagement metrics) over stated preferences because they directly correlate with business metrics. However, this can lead to optimizing for addictive rather than genuinely valuable content.

(c) Assume the company wants to optimize its news feed to cater to user's preferences. What are the ethical considerations of prioritizing the user's stated preferences vs revealed preferences?

Answer: **Ethical considerations for revealed preferences**:

- – May exploit psychological vulnerabilities (doom-scrolling, outrage addiction)
- – Can create filter bubbles and echo chambers
- – Optimizes for what captures attention, not what's beneficial
- – May promote sensationalist or misleading content that generates clicks
- – Users may regret time spent (revealed $\neq$ reflective preference)
- \+ Reflects actual behavior rather than aspirational self-image

Ethical considerations for stated preferences:

- \+ Respects user autonomy and explicit choices
- \+ Aligns with users' reflective, considered preferences
- \+ Less likely to exploit cognitive biases
- – Users may not know what they actually want
- – Stated preferences may be aspirational but not realistic
- – Can miss content users would genuinely enjoy but didn't know to request

Recommendation: A balanced approach that primarily respects stated preferences while using revealed preferences to surface relevant content within those boundaries—never overriding explicit user choices with engagement optimization.

(d) Suggest a way to incorporate exploration to test whether a user's preferences (stated or revealed) might evolve over time.

Answer: Several exploration strategies can help detect evolving preferences:

1. **ϵ-greedy exploration**:
   - With probability ϵ (e.g., 5-10%), show content outside the user's typical profile
   - Track engagement with exploratory content to detect new interests
   - Decay ϵ for users with stable preferences, increase for new users
2. **Thompson Sampling / UCB**:
   - Maintain uncertainty estimates for user preferences across topics
   - Explore topics with high uncertainty about user interest
   - Naturally balances exploitation of known preferences with exploration
3. **Temporal preference modeling**:
   - Use time-decayed weights on historical interactions
   - More recent behavior weighted higher than older behavior
   - Periodically "forget" old preferences to allow rediscovery
4. **Contextual bandits**:
   - Model preferences as context-dependent (time of day, device, location)
   - Explore when context changes significantly
   - Allow different preference profiles for different contexts
5. **Explicit preference refresh**:
   - Periodically prompt users to update stated preferences
   - A/B test whether stated preferences have drifted from revealed behavior