CS 330 Autumn 2021/2022 Homework 4

Exploration in Meta-Reinforcement Learning

Due Monday, November 8th, 11:59 PM PT

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

In this assignment, we will be exploring meta-reinforcement learning algorithms. In particular, you will

1. Experiment with black-box meta-RL methods [1,2] trained end-to-end.
2. Answer conceptual questions about a popular decoupled meta-RL algorithm, PEARL [4].
3. Conceptually compare end-to-end with decoupled optimization.
4. Implement components of DREAM [3] to achieve the best of both groups of approaches.

To answer the questions in this homework, it may be helpful to refer to the DREAM paper, which we've included in the zip of the starter code under `dream_paper.pdf`.

Submission: To submit your homework, submit one PDF report to Gradescope containing written answers/plots to the questions below and a zip file of your code. The PDF should also include your name and any students you talked to or collaborated with.

Setup: Please download and unzip the starter code, and then follow the instructions in the README. Download the dependencies, which we recommend you do via a `virtualenv` as follows:

```
$ python3 -m virtualenv venv
$ source venv/bin/activate
$ pip install -r requirements.txt
```

Please ensure that you're using Python3.7. Otherwise, installing the dependencies will not work.

Code Overview: The main entry point for the code is via `dream.py` and `rl2.py`, which are the training scripts for DREAM and RL^2 respectively. Both of these can be invoked as follows:

```
$ python3 {script}.py {experiment_name} -b environment=\"map\"
```

In this invocation, {`script`} can either be `dream.py` or `rl2.py` and {experiment_name} can be any string with no white spaces. Results from this invocation are saved under `experiments/`{`experiment_name`}. For example, to launch the DREAM training script and save the results to `experiments/dream`, you would run:

```
$ python3 dream.py dream -b environment=\"map\"
```

You can pass the `--force_overwrite` flag to run another experiment with the same experiment name, which will overwrite any previously saved files at the corresponding experiment directory. If you do not pass this flag, the scripts will not allow you to run two experiments with the same experiment name.

There are two important subdirectories in each experiment directory:

- **Tensorboard:** Each experiment includes a `experiments/experiment_name/tensorboard` subdirectory, which will log important statistics about the training run, including the meta-testing returns under the `rewards/test` tag and the meta-training returns under the `rewards/training` tag. To view these, point Tensorboard at the appropriate directories. All curves are plotted with two versions, one where the x-axis is number of meta-training trials under `tensorboard/episode` and one where the x-axis is the number of environment steps `tensorboard/step`.
- **Visualizations:** Each experiment also includes a `experiments/experiment_name/visualize` subdirectory. This directory includes `.gif` videos of the agent during meta-testing and is structured as follows. The top level of subdirectories identify how many meta-training trials have elapsed before the video.

  In experiments run with `dream.py`, the exploration episode is saved under {`video_num`}`-exploration.gif` and the exploration episode is saved under {`video_num`}`-exploitation.gif`. For example, the video under `experiments/dream/visualize/10000/0-exploration.gif` is the first exploration meta-testing episode after 10000 meta-training trials.

  In experiments run with `rl2.py`, {`video_num`}`.gif` contains both the exploration and exploitation episodes, with the exploration episode first. For example, the video under `experiments/rl2/visualize/10000/0.gif` contains the first exploration and exploitation episode after 10000 meta-training trials.

You will implement two short methods inside the `embed.py` file.

Problem 0: Grid World Navigation with Buses (4 Points)

We consider a grid world illustrated in Figure 1. From a high level, the agent is given a goal each episode and must reach it in as few steps as possible. To quickly get to the goal, the agent may ride a bus. This brings the agent to the destination of that bus, which is the other bus of the same color. In different tasks, the buses in the corners permute, while the buses in the center remain fixed. For example, in the left task in Figure 1, the center blue bus's destination is the bottom right corner, while in the right task, its destination is the top right corner. Note that the goal is not part of the task, and all four corners are potential goals in all tasks. There is also a map at a fixed location in all of the tasks, which tells the agent the destination of each bus, when visited.

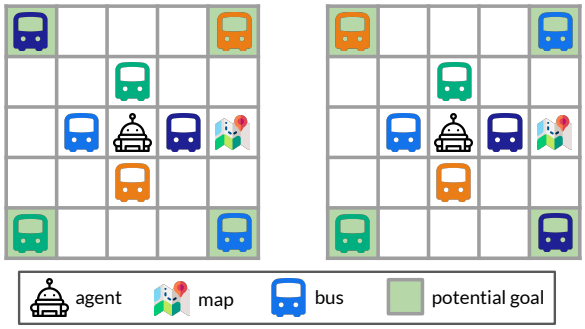

Figure 1: Two example tasks in the grid world domain. There are 4! different tasks, corresponding to different permutations of the buses in the corners.

More concretely, the **state** consists of 4 components

- The agent's (x, y)-position in the grid
- A one-hot indicator of the object at the agent's current position (none, bus, map).
- A one-hot goal g corresponding to one of the four possible goal locations in the corners (shown in green).
- A one-hot that is equal to the problem ID μ (defined below) if the agent is standing on the map, and 0 otherwise. Standing on the map effectively encodes the destination of each bus.

The agent begins every episode at the center of the grid, as in Figure 1. During an episode, the goal is held fixed, while it is re-sampled uniformly across the 4 potential goal locations in each new episode.

At each timestep, the agent can take one of 5 **actions**:

- Move one cell up, down, left or right.
- Ride the bus that the agent is currently on. This teleports the agent to the other bus of the same color.

The agent receives $+1$ **reward** for reaching the correct goal position. The agent receives -0.3 **reward** at each timestep it is not at the correct goal, incentivizing it to reach the goal as

quickly as possible. The episode ends if either the agent goes to any goal location (correct or incorrect) or if 20 timesteps have passed.

Each task is associated with a **problem ID** μ. The only thing that changes between tasks is the bus destinations: i.e., which colored bus appears in which outer corner. Therefore, there are $4! = 24$ different tasks. These tasks are **uniformly sampled** during meta-training and meta-testing.

Throughout the assignment, we consider the meta-RL setting with one *exploration episode* and one *exploitation episode*. The objective is to maximize the returns achieved in the exploitation episode, which we refer to as the *exploitation returns*. Note that the returns achieved in the exploration episode do not matter. During the exploitation episode, the agent is allowed to condition on the exploration episode $\tau^{\text{exp}} = (s_0, a_0, r_0, \ldots)$.

a) What returns are achieved by only taking the move action to get to the goal, without riding any buses: i.e., directly walking to the goal? (**1 point**)

b) If the bus destinations (i.e., the problem ID) were known, what is the optimal returns that could be achieved in a single exploitation episode? Describe an exploitation policy that achieves such returns given knowledge of the bus destinations. (**1 point**)

c) Describe the exploration policy that discovers all of the bus destinations within the fewest number of timesteps. (**1 point**)

d) Given your answers in b) and c), what is the optimal exploitation returns achievable by a meta-RL agent? (**1 point**)

For Problems 1 and 3, note that in the visualizations saved under `experiments/experiment_name/visualize`:

- The agent is rendered as a red square.
- The grid cells that the agent has visited in the episode are rendered as small origin squares.
- There are four pairs of buses, rendered as blue, pink, cyan, and yellow squares.
- The map is rendered as a black square.
- The goal state is rendered as a green square, which obscures one of the buses.

Problem 1: End-to-End Meta-Reinforcement Learning (5 Points)

In this problem, we'll analyze the performance of end-to-end meta-RL algorithms on the grid world. To do this, start by running the RL^2 agent on the grid world navigation task for $50,000$ trials by running the below command. This should take approximately 1 hour.

```
python3 rl2.py rl2 -b environment=\"map\"
```

a) Examine the Tensorboard results under the tag `reward/test` in the `experiments/rl2` directory. To 1 decimal place, what is the average meta-testing exploitation returns RL^2 achieves after training? (**1 point**)

b) Examine the videos saved under `experiments/rl2/visualize/36000/`. Describe the exploration and exploitation behaviors that RL^2 learns. (**2 points**).

c) Does RL^2 achieve the optimal returns? Based on what you know about end-to-end meta-RL, do these results align with your expectations? Why or why not? (**2 points**).

Problem 2: Decoupled Meta-Reinforcement Learning (8 Points)

In this problem, we'll examine PEARL [4], a common decoupled meta-RL algorithm. Recall that PEARL consists of 3 key components: a *prior* over latent z's $\mathrm{p(z)}$, a *policy* $\pi(a \mid s, z)$ that conditions on the latents, and a *posterior* $q(z \mid \{\tau_i\}_{i=1}^N)$ over the latents, given the past N episodes $\tau_1, \ldots, \tau_N$. Intuitively, the latent z can be thought of as an approximation of the task, so the prior and posterior model uncertainty over what the current task is, and the policy attempts to achieve high returns, assuming that z identifies the current task.

On a new task, PEARL explores via Thompson sampling. First, PEARL samples a $z \sim p(z)$ from the prior, and rolls out an episode from the policy $\pi(a \mid, s, z)$, which yields an episode τ_1. Then, a new latent is sampled from the posterior $z \sim q(z \mid \tau_1)$ and another episode is rolled out from the policy given the new z. This process is repeated for as many episodes as is allowed in the setting. Since we're considering only a single exploration episode and single exploitation episode, a latent is only sampled once from the posterior in our setting.

In this problem, we consider an idealized version of PEARL, where z is actually the problem ID μ, and the policy $\pi(a \mid s, \mu)$ is the optimal policy for each task μ. That is, $\pi(a \mid s, \mu)$ acts optimally assuming that μ is the true problem ID. Below, we'll directly refer to μ instead of z.

a) In the grid world, the prior $p(\mu)$ is uniform over the 24 tasks. After observing a single exploration episode τ, what is the new posterior over tasks $p(\mu \mid \tau)$? Hint: think about what the policy does. (**2 points**)

b) What is the expected returns achieved by PEARL given a single exploration episode? Hint: there are two cases to consider. Show your work. (**4 points**)

c) Does this idealized version of PEARL achieve the optimal returns? Based on what you know about decoupled meta-RL algorithms, do these results align with your expectations? Why or why not? (**2 points**)

Problem 3: DREAM (9 Points)

In Problem 1d) and Problem 2c) we observed some shortcomings of end-to-end and existing decoupled meta-RL approaches. In this problem, we'll implement some components

of DREAM, which attempts to address these shortcomings, given the assumption that each meta-training task is assigned a unique *problem ID* μ. During meta-testing, DREAM does not assume access to the problem ID.

From a high level, DREAM works by separately learning exploration and exploitation. To learn exploitation, DREAM learns an exploitation policy $\pi_\theta^{\text{task}}(a \mid s, z)$ that tries to maximize returns during exploitation episodes, conditioned on a task encoding z. DREAM learns a *encoder* $F_\psi(z \mid \mu)$ to produce the task encoding z from the problem ID μ. Critically, this encoder is trained in such a way that z contains only the information necessary for the exploitation policy π_θ^{task} to solve the task and achieve high returns.

By training the encoder in this way, DREAM can then learn to explore by trying to recover the information contained in z. To achieve this, DREAM learns an exploration policy π_ϕ^{exp}, which produces an exploration trajectory $\tau^{\text{exp}} = (s_0, a_0, r_0, \ldots)$ when rolled out during the exploration episode. To recover the information contained in z, DREAM tries to maximize the mutual information between the encoding z and the exploration trajectory τ^{exp}:

$$\max_\phi I(F_\psi(z \mid \mu), \tau^{\text{exp}}).$$

This objective can be optimized by maximizing the following variational lower bound:

$$\mathcal{J}(\omega, \phi) = \mathbb{E}_{\mu, z \sim F_\psi, \tau^{\text{exp}} \sim \pi_\phi^{\text{exp}}}[\log q_\omega(z \mid \tau^{\text{exp}})]$$

where $q_\omega(z \mid \tau^{\text{exp}})$ is a learned *decoder*. Note that this decoder enables us to convert an exploration trajectory τ^{exp} into a task encoding z that the exploitation policy uses. This is critical for meta-test time, where the problem ID is unavailable, and z cannot be computed via the encoder $F_\psi(z \mid \mu)$.

The objective $\mathcal{J}(\omega, \phi)$ is optimized with respect to both the decoder q_ω and the exploration policy π_ϕ^{exp}:

a) For simplicity, we parametrize the decoder $q_\omega(z \mid \tau^{\text{exp}})$ as a Gaussian $\mathcal{N}(g_\omega(\tau^{\text{exp}}), \sigma^2 I)$ centered at a learned $g_\omega(\tau^{\text{exp}})$ with unit variance. Then, $\log q_\omega(z \mid \tau^{\text{exp}})$ equals negative mean-squared error $-\|g_\omega(\tau^{\text{exp}}) - z\|_2^2$ plus some constants independent of ω. Overall, *maximizing* $\mathcal{J}(\omega, \phi)$ with respect to the decoder parameters ω is equal to *minimizing* the below mean-squared error **with respect to** ω:

$$\mathbb{E}_{z \sim F_\psi(\mu)} \left[\|g_\omega(\tau^{\text{exp}}) - z\|_2^2 \right].$$

Code: Fill in the `_compute_losses` method of the `EncoderDecoder` in `embed.py` to implement the above equation for optimize $\mathcal{J}(\omega, \phi)$ with respect to the decoder parameters ω. **(2 points)**

b) To optimize $\mathcal{J}(\omega, \phi)$ with respect to the exploration policy parameters ϕ, we expand out $\mathcal{J}(\omega, \phi)$ as a telescoping series:

$$\mathcal{J}(\omega, \phi) = \mathbb{E}_{\mu, z \sim F_\psi(\mu)}[\log q_\omega(z \mid s_0)] + \mathbb{E}_{\mu, z \sim F_\psi(\mu), \tau^{\text{exp}} \sim \pi^{\text{exp}}} \Big[\sum_{t=0}^{T-1} \log q_\omega(z \mid \tau_{:t+1}^{\text{exp}}) - \log q_\omega(z \mid \tau_{:t}^{\text{exp}}) \Big],$$

where $\tau^{\text{exp}}_{:t}$ denotes the exploration trajectory up to timestep t: $(s_0, a_0, r_0, \ldots, s_t)$. Only the second term depends on the exploration policy, and since it occurs per timestep, it can be maximized by treating it as the following intrinsic reward function r^{exp}_t, which we can maximize with standard reinforcement learning:

$$r^{\text{exp}}_t(a_t, r_t, s_{t+1}, \tau^{\text{exp}}_{t-1}; \mu) = \mathbb{E}_{z \sim F_\psi(\mu)} \left[\log q_\omega(z \mid \tau^{\text{exp}}_{:t+1}]) - \log q_\omega(z \mid \tau^{\text{exp}}_{:t}) \right]. \quad (1)$$

Note that $\tau^{\text{exp}}_{:t+1}$ is equal to $\tau^{\text{exp}}_{:t}$ with the additional observations of (a_t, r_t, s_{t+1}). The exploration policy is optimized to maximize this intrinsic reward r^{exp}_t instead of the extrinsic rewards r_t, which will maximize the objective $\mathcal{J}(\omega, \phi)$. Intuitively, r^{exp}_t is the "information gain" representing how much additional information about z – which encodes all the information to solve the task – the tuple (a_t, r_t, s_{t+1}) contains over what was already observed in $\tau^{\text{exp}}_{:t}$.

Code: Implement the reward function $r^{\text{exp}}_t(a_t, r_t, s_{t+1}, \tau^{\text{exp}}_{t-1}; \mu)$ by filling in the `label_rewards` function of `EncoderDecoder` in `embed.py`. Note that you'll need to make the same substitution for $\log q_\omega(z \mid \tau^{\text{exp}})$ in Equation (1) that we used in part a). **(2 points)**

c) Check your implementation by running DREAM:

```
python3 dream.py dream -b environment=\"map\"
```

Submit the plot for test returns under the tag `rewards/test` from the `experiments/dream` directory. Submit the plot under `tensorboard/step`, not the plot under `tensorboard/episode`. If your implementation in part a) and b) is correct, you should see the test returns training curve improve within 10 minutes of training. By around 25 minutes, the test returns curve should begin to look different from RL^2. **(2 points)**

d) Does DREAM achieve optimal returns in your results from c)? Based on what you know about DREAM, do these results align with your expectations? Why or why not? **(2 points)**

e) Inspect the videos saved under `experiments/dream/visualize/28000` or a later step after DREAM converges. Describe the exploration and exploitation behaviors that DREAM has learned. **(1 point)**

References

[1] Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, Pieter Abbeel. RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning. https://arxiv.org/abs/1611.02779

[2] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, Matt Botvinick. Learning to Reinforcement Learn. The Annual Meeting of the Cognitive Science Society. (CogSci) 2017. https://arxiv.org/abs/1611.05763

[3] Evan Zheran Liu, Aditi Raghunathan, Percy Liang, Chelsea Finn. Decoupling Exploration and Exploitation for Meta-Reinforcement Learning without Sacrifices. https://arxiv.org/abs/2008.02790

[4] Kate Rakelly, Aurick Zhou, Deirdre Quillen, Chelsea Finn, Sergey Levine. Efficient Off-Policy Meta-Reinforcement Learning via Probabilistic Context Variables. https://arxiv.org/abs/1903.08254