CS 224R Spring 2022/2023 Homework 4
Goal-Conditioned Reinforcement Learning & Meta-Reinforcement Learning
Due Wednesday May 31st, 11:59 PM PT

SUNet ID: 06661759
Name: Paul-Emile Giacomelli
Collaborators: Yoni Gozlan, Josselin Sommerville

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

Overview

This assigment consists of two parts.

In Part 1, we will be looking at goal-conditioned reinforcement learning and hindsight experience replay (HER). In particular, you will:

1. Adapt an existing model (Deep Q-Network) to be goal-conditioned.
2. Run goal-conditioned DQN on two environments.
3. Implement Hindsight Experience Replay (HER) [1,2] on top of goal-conditioned DQN.
4. Compare the performance with and without HER.

In Part 2, we will be exploring meta-reinforcement learning algorithms. In particular, you will:

1. Experiment with a black-box meta-RL method [3] trained end-to-end.
2. Implement components of DREAM [4] to replace the end-to-end optimization objective.
3. Compare the performance of end-to-end versus decoupled optimization.

We have provided you with starter code, which can be downloaded from the course website. All code for Part 1 is located in the directory `hw4/goal_conditioned_rl` and all code for Part 2 is located in `hw4/meta_rl`.

Submitting the PDF: Submit a PDF report to Gradescope containing the written answers, plots, and Tensorboard graphs (screenshots are acceptable) to the questions. The PDF should include your name and any students you talked to or collaborated with.

Submitting the Code and Experiment Runs: In order to turn in your code and experiment logs, create a folder that contains the following:

- `data/goal_conditioned_rl` folder with all logged runs for Part 1 of the assignment. **Note: Please remove any redundant run folders. Remove any empty/incomplete logs that correspond to interrupted/failed runs.**
- `data/meta_rl` folder with all logged (tensorboard only) runs for Part 2 of the assignment. **Note: Please remove any redundant run folders. Remove any empty/incomplete logs that correspond to interrupted/failed runs. Finally, please only include tensorboard logs.**
- `code/goal_conditioned_rl` folder with the files `trainer.py` and `run_episode.py` from Part 1.
- `code/meta_rl` folder with the file `encoder_decoder.py` from Part 2.

Gradescope: Submit both the PDF and the code and experiment runs in the appropriate assignments on Gradescope. An autograder will be provided to evaluate the performance of your policies from the generated tensorboard files.

Use of GPT/Codex/Copilot: For the sake of deeper understanding on implementing imitation learning methods, assistance from generative models to write code for this homework is prohibited.

Part 1: Goal-Conditioned Reinforcement Learning

In Part 1, we will be running goal-conditioned Q-learning on two problems: (a) a toy problem where we flip bits in a bit vector to match it to the current goal vector and (b) move the end effector of a simulated robotic arm to the desired goal position. Applying hindsight relabeling to a goal-conditioned reinforcement learning setting is one of the most promising and commonly-used ways to improve sample efficiency and exploration challenges of RL!

Setup: Part 1 requires the use of physics simulator MuJoCo, and installing the related python bindings. While we have provided instructions to setup MuJoCo and install the corresponding python bindings `mujoco-py` in `hw4/goal_conditioned_rl/README.md`, the setup can be dependent on the machine and it may require additional steps to be setup successfully. **We recommend completing the setup as soon as possible**. In particular, we recommend setting up the virtual environment and installing the requirements, and then running `python test_installation.py`. The script should print `Dependencies successfully installed!` if the setup was successful. In case the local setup is unsuccessful after following the troubleshooting guide, we recommend completing the assignment by creating a virtual machine on AWS. Please follow the detailed instructions in `hw4/goal_conditioned_rl/README.md`.

Part 1 Code Overview: The code consists of several files to enable Q-learning. All code for Part 1 is located in the directory `hw4/goal_conditioned_rl` You are expected to write the code in the following files: `run_episode.py, trainer.py`. A brief description for the codebase is provided here:

- `trainer.py`: The main training loop `train`. Alternates between collecting transitions using Q-value networks and training the networks on collected transitions.
- `run_episode.py`: Collects an episode using the current Q-value network, and returns the transitions, collected reward and whether the agent was successful during the episode.
- `replay_buffer.py`: Buffer for storing the transitions collected in the environment.
- `q_network.py`: Creates a feedforward neural network with one hidden layer. This neural net represents our Q-network.
- `bit_flip_env.py`: The source code for the bit flipping environment, which is set up to follow the gym API with an `__init__`, `reset` and `step` functions.
- `sawyer_action_discretize.py`: Wraps the `SawyerReachXYEnv` from `multiworld` environment and converts the continuous action space into a discrete action space with a simplified 2D observation space.
- `main.py`: Main file to configure the experiment.

A detailed description for every function can be found in the comments. You are not expected to change any code except for sections marked with `TODO`. Next, we provide description of the environments followed by the exact expectations for this assignment.

Environments

You will be running RL methods on two environments:

Environment 1: Bit Flipping Environment

In the bit-flipping environment, the state is a binary vector with length `n`. The goal is to reach a known goal vector, which is also a binary vector with length `n`. At each step, we can flip a single bit in the vector (changing a 0 to 1 or a 1 to 0). This environment can very easily be solved without reinforcement learning, but we will use the DQN algorithm to understand how adding hindsight relabelling (HER) can improve performance.

The bit flipping environment is an example of an environment with sparse rewards. At each step, we receive a reward of -1 when the goal and state vector do not match and a reward of 0 when they do. With a larger vector size, we receive fewer non-negative rewards.

Environment 2: 2D Sawyer Arm

The Sawyer Arm is a multi-jointed robotic arm for grasping and reaching (https://robots.ieee.org/robots/sawyer/). The arm operates in a 2D space, and the goal is to move the robot to a set of coordinates. The sawyer reach is an example of a dense reward environment, where the reward is given by negative Euclidean distance between the robot arm and the goal state. The end-effector (EE) is constrained to a 2-dimensional rectangle parallel to a table. The action controls EE position. The state is the XY position of the EE and the goal is an XY position of the EE.

Problem 1: Implementing Goal-conditioned RL

We start this problem with a goal-conditioned implementation of DQN. The Q-function takes in the concatenated state and goal as input. You can think of the goal-conditioned implementation as an extended Markov decision process (MDP), where your state space contains both the original state and the goal. We will use this goal-conditioned Q-network to collect episodic data in `run_episode.py`.

For this part of the assignment, complete `run_episode.py` and run the following command:

```
python main.py --env bit_flip --num_bits 6 --num_epochs 250 --her_type no_hindsight
```

The evaluation metrics should be available in tensorboard events logged in `logs/` by default. **Verify** the `eval_metrics`, that is `total_reward` should be above -40.0 and `success_rate` should be 1.0. This plot illustrates the performance without using HER. You do **not** need to include this in the homework.

Implementation notes:

- For simplicity, we will only consider the *greedy* action with respect to Q-network. Pass this action to `env.step`.
- The `env.step` function returns `next_state, reward, done, info`, where `info` is a dictionary containing the a boolean under the key `'successful_this_state'`, indicating whether the state was successful or not. Use this value to update `succeeded`, such that `run_episode` returns `True` if the *policy was successful at any step of the episode*. To understand more about `env.step`, read the documentation for the function in `bit_flip_env.py`.
- Ensure that floating point numpy arrays use `np.float32`. You may need to recast some of the arrays to ensure that.

Problem 2: Adding HER to Bit Flipping

With HER, the model is trained on the actual (state, goal, reward) tuples along with (state, goal, reward) tuples where the goal has been relabeled after the fact. The goals are relabeled to the state that was *actually reached* and the rewards should be relabeled correspondingly. In other words, we pretend that the state we reached was our goal all along. HER gives us more examples of actions that lead to positive rewards, effectively doubling our experience (since we use the original episode as well as the relabeled one). The reward function for relabeled goals is the same as the environment reward function; for the bit flipping environment, the reward is -1 if the state and goal vector do not match and 0 if they do match.

There are three different variations of HER: `final`, `random`, and `future`. Suppose the data collected in an episode consists of the following (state, goal, reward) tuples: $\{(s_t, g, r_t)\}_{t=0}^{T}$, where t indicates each time step in the episode. Given a (state, goal, reward) tuple (s_i, g_i, r_i) each variation of HER relabels the goal g_i differently:

- `final`: The final state of the episode is used as the goal.
- `random`: A random state in the episode is used as the goal.
- `future`: A random future state of the episode is used as the goal. Specifically, if you want to relabel the goal in the tuple (s_i, g_i, r_i), only states s_j with $j > i$ can be used.

Implementation Notes:

- Always use `copy()` when assigning an existing numpy array to a new variable. NumPy does not create a new copy by default.
- When choosing a new goal for relabelling, choose the state corresponding to `next_state` from the experience.

More details of these three implementations are in the comments of `trainer.py`. Implement the three variations of HER in the function `update_replay_buffer` in `trainer.py`. You **do not** need to submit a plot for this.

Problem 3: Analyzing HER for Bit Flipping Environment

Once you have completed the previous problems, we can analyze the role of HER in goal-conditioned RL. We analyze the performance of HER as the size of the bit vector is increased from 6 to 25. For each of the parts (a) to (d), submit a tensorboard screenshot showing the `eval_metrics` for different runs on the same plot (check the correct event files). A total of 4 screenshots should be submitted for this section.

a) Run the following commands:

```
python main.py --env=bit_flip --num_bits=6 --num_epochs=250 --her_type no_hindsight
python main.py --env=bit_flip --num_bits=6 --num_epochs=250 --her_type final
```

Include your screenshot here.

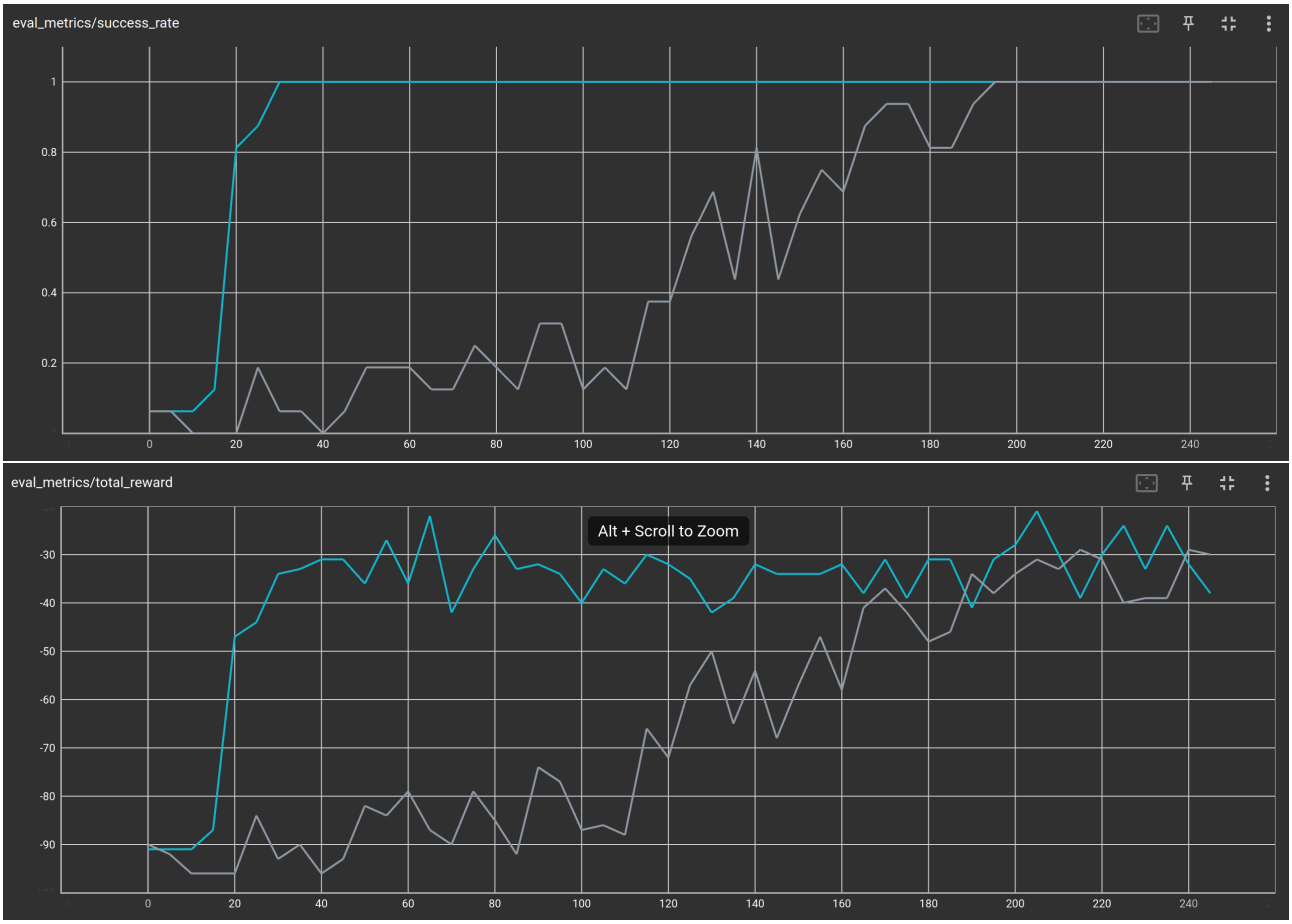

Figure 1: Bit Flipping with HER (type "final" - blue) and without HER (gray) – 6 bits long

b) Run the following commands:

```
python main.py --env=bit_flip --num_bits=15 --num_epochs=500 --her_type no_hindsight
python main.py --env=bit_flip --num_bits=15 --num_epochs=500 --her_type final
```

Include your screenshot here.

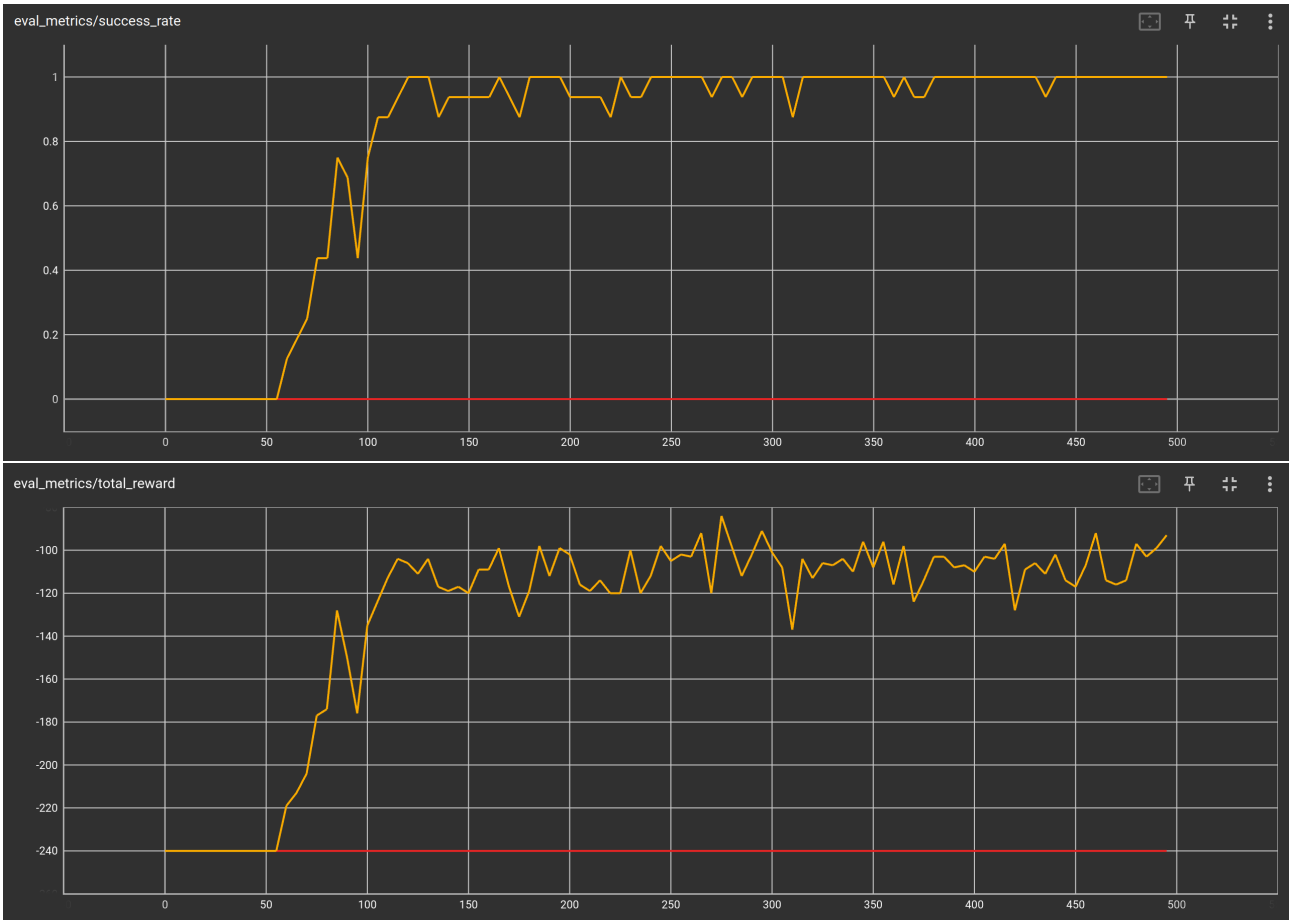

Figure 2: Bit Flipping with HER (type "final" - orange) and without HER (red) – 15 bits long

c) Run the following commands:

```
python main.py --env=bit_flip --num_bits=25 --num_epochs=1000 --her_type no_hindsight
python main.py --env=bit_flip --num_bits=25 --num_epochs=1000 --her_type final
```

Include your screenshot here.

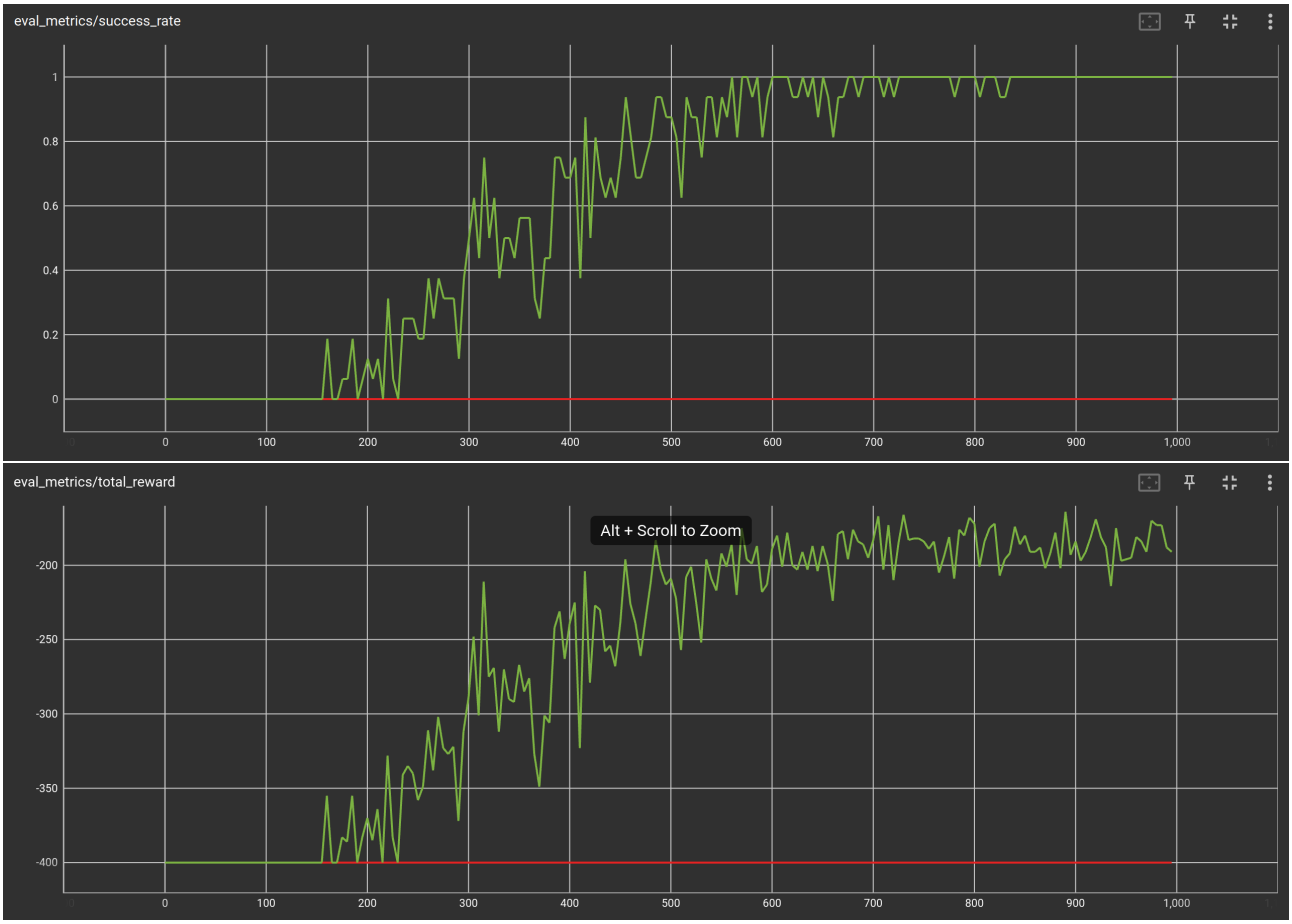

Figure 3: Bit Flipping with HER (type "final" - green) and without HER (red) – 25 bits long

Explain your findings in parts (a)-(c) and why you expect the methods to perform in the observed manner for the varying numbers of bits.

Write your response here.

For a length of 6 bits, the agent is able to learn with and without HER. However the learning is faster with HER.
For a length of 15 and 25 bits, the agent is unable to learn without HER, and is able to learn with HER.

These observations can be explained by several reasons:

- First, the BitFlip environment provides sparse rewards. With a bit length of 6, there is enough random exploration for the DQN without HER to occasionally stumble upon the correct action sequence. However, as the bit length increases to 15 and 25, the size of the action space grows exponentially, making it increasingly difficult for the agent to stumble on the correct action sequence. Therefore,

the agent without HER struggles to find a successful trajectory, and thus has a poor learning performance.

- Then, it can be challenging for the agent in environments with sparse rewards to explore effectively and find the optimal solution. HER helps by generating additional goals for the agent to learn from, even when it fails to reach the original goal. So the agent can still extract valuable information from unsuccessful trajectories, which leads to a more efficient learning and an improved performance.

d) Finally, we will compare the three versions of HER, with the baseline of not using HER:

```
python main.py --env=bit_flip --num_bits=15 --num_epochs=500 --her_type no_hindsight
python main.py --env=bit_flip --num_bits=15 --num_epochs=500 --her_type final
python main.py --env=bit_flip --num_bits=15 --num_epochs=500 --her_type random
python main.py --env=bit_flip --num_bits=15 --num_epochs=500 --her_type future
```

Since two of the commands (`her_type no_hindsight` and `her_type final`) are identical to part (b), you do **not** need to run them again.

Explain your findings from these four runs and provide justification as to why you expect the methods to perform the way it did for the varying relabelling strategies.

Write your response here.

We observe different performance for each case:

- without HER: Same case as in the previous question. The agent relies on sparse rewards and the exploration strategy to learn. As the environment is 15 bits long, the agent struggles to learn and ends up not learning at all and acieves poor performance.
- with 'final' HER: Same case as in the previous question. The agent learns from unsuccessful trajectories that did not reach the original goal state. That way, the agent can extract valuable information and improve its learning process. Therefore, the performance of DQN with 'final' HER is better is without HER.
- with 'random' HER: It introduces more diversity in training samples compared to the 'final' HER. By considering alternative goals, the agent can explore different successful trajectories and learn a more robust policy. This is why the 'random' HER agent achieves better performance than the 'final' HER agent, as it encourages exploration and generalization by training on a wider range of successful trajectories.
- with 'future' HER: This approach encourages the agent to learn from a mix of successful and unsuccessful trajectories. By training on both types of trajectories, the agent can gain a more comprehensive understanding of the environment and achieve better performance. The 'future' HER agent outperforms both 'final' and 'random' HER agents, as it combines the benefits of both relabeling successful goals and exploring diverse successful trajectories.

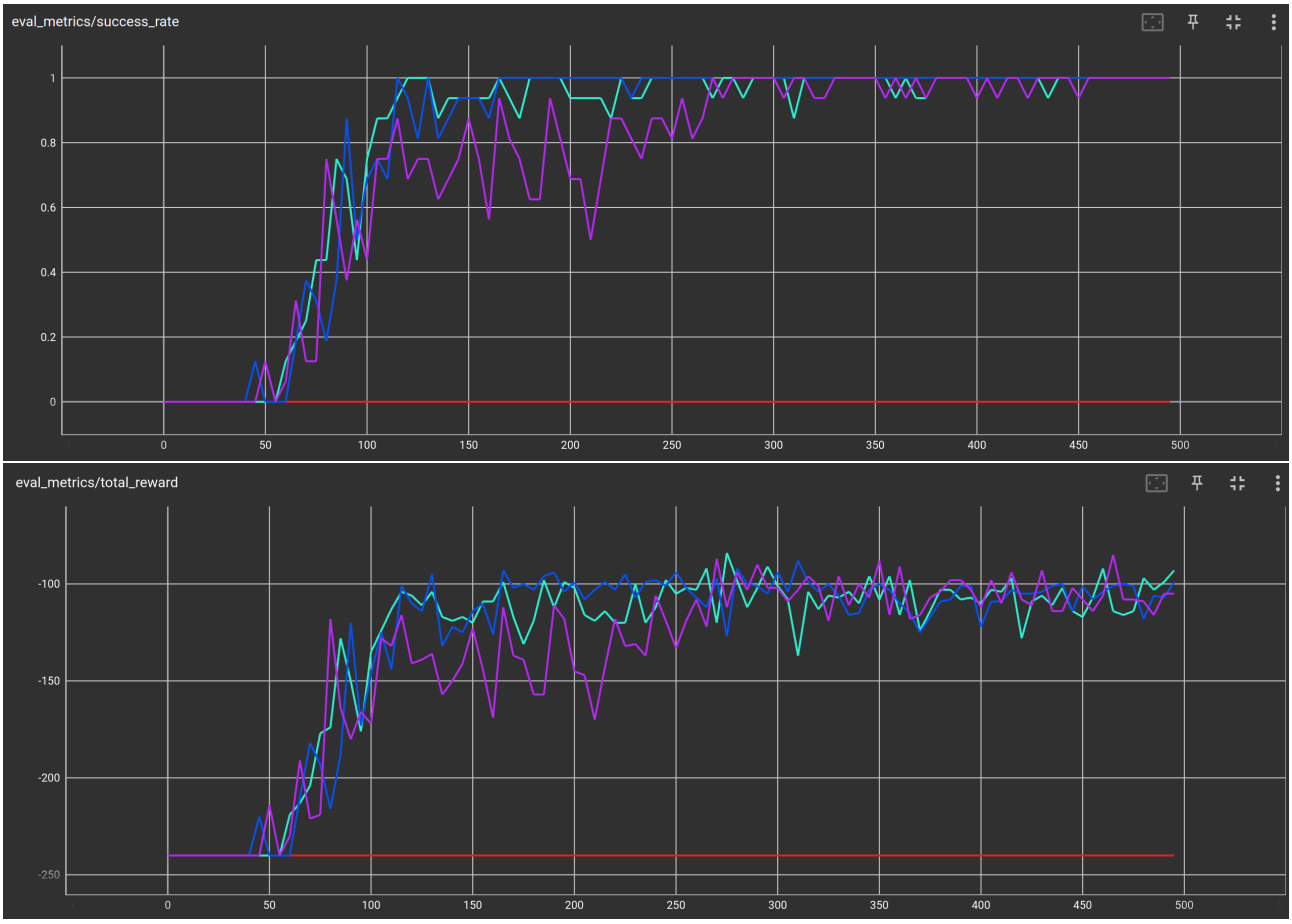

Figure 4: Bit Flipping with HER (type ”final” - purple || type ”random” - blue || type ”future” - cyan) and without HER (red) – 15 bits long

Problem 4: Analyzing HER for Sawyer Reach

If implemented correctly, HER should work for the second environment, `Sawyer Reach`.

Compare the performance of the Sawyer arm with and without HER. Run the following commands:

```
python main.py --env=sawyer_reach --num_epochs=1000 --her_type no_hindsight
python main.py --env=sawyer_reach --num_epochs=1000 --her_type final
```

a) Submit the tensorboard screenshot comparing the `eval_metrics` in your report. Include your screenshot here.

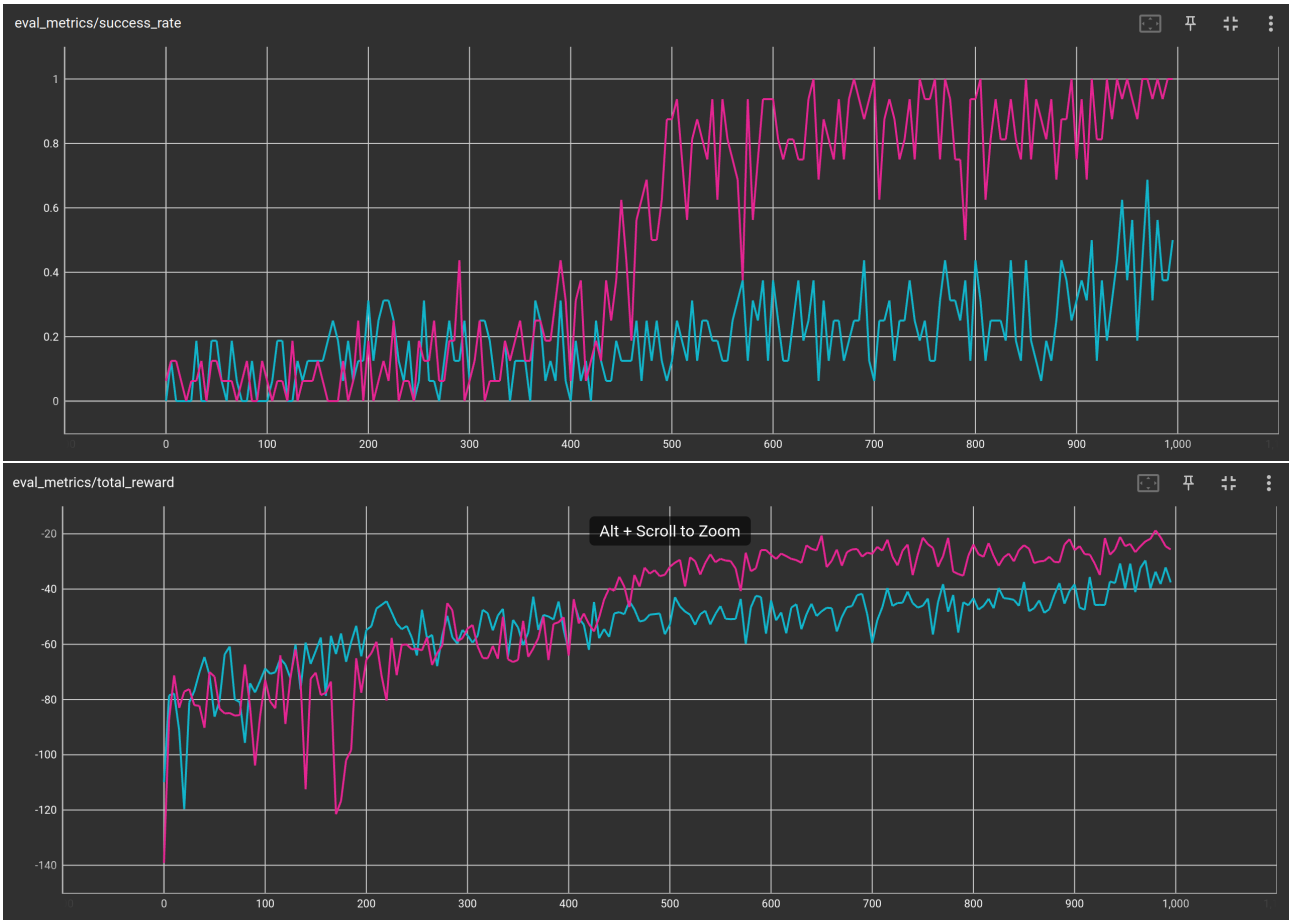

Figure 5: Sawyer Reach with HER (type "final" - pink) and without HER (blue)

b) Discuss your findings: Compare the role of HER in Bit Flipping Environment and Sawyer Reach. Comment on the differences between the contribution of HER, if any. Write your response here.

The reward structure of the environments plays an important role in the effectiveness of HER. In the BitFlip environment, the rewards are sparse, as the agent receives a positive reward only when it successfully flips all the bits. The 'final' HER variant is beneficial in this case, as it allows the agent to learn from unsuccessful trajectories that did not reach the original goal state. On the other hand, the Sawyer_reach environment has a dense reward structure, where the agent receives feedback throughout the trajectory. In such cases, the impact of 'final' HER might be less significant, as the agent can learn directly from the intermediate rewards provided by the environment. In our case, we can see that the agent with HER performs way better than the agent without HER as using HER allows the agent to have more examples when it actually completes the task.

Part 2: Meta-Reinforcement Learning

Setup: Please navigate to `hw4/meta_rl` and follow the instructions in the README. **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 `scripts/dream.py` and `scripts/rl2.py`, which are the training scripts for DREAM and RL^2 respectively. Both of these can be invoked as follows:

```
$ python scripts/{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:

```
$ python scripts/dream.py dream -b environment=\"map\"
```

To overwrite previous results, you would run:

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. An example command for running DREAM is below:

```
$ python scripts/dream.py dream -b environment=\"map\" --force_overwrite
```

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/encoder_decoder.py file.

Problem 0: Grid World Navigation with Buses

We consider a grid world illustrated in the Homework 4 PDF. 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, the center light 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.

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. 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 $\tau^{\text{exp}} = (s_0, a_0, r_0, \ldots)$.

1. 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?
   Write your response here.
   By only taking the move action to get to the gpoal without riding any bus, the returns that are achieved are within the range [-6, +0.1].
   Indeed, the worst possible trajectory would be to do 20 steps without reaching any goal. This would lead to a return of $20 \times -0.3 = -6$. On the other hand, the best trajectory would be to take 4 steps to the right goal. This would lead to a return of $3 \times -0.3 + 1 = 0.1$.

2. 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.
   Write your response here.
   In this case, the optimal return that could be achieved in a single exploitation episode is 0.7. Indeed, the optimal trajectory in this case is to take one step to the bus that would lead to the right goal: $1 \times -0.3 + 1 = 0.7$.

3. Describe the exploration policy that discovers all of the bus destinations within the fewest number of timesteps.
   Write your response here.
   The exploration policy that discovers all of the bus destinations within the fewest number of steps is the following:

   - Take two steps to the right to find the map
   - Go through the different buses to discover their respective destinations. This represents 7 steps.

   However, after a certain amount of learning steps, the agent will now which configuration bus stops location corresponds to which environment number. From this

moment on, the agent will only go to the map directly by taking two steps to the right.

4. Given your answers in b) and c), what is the optimal exploitation returns achievable by a meta-RL agent?
   Write your response here.
   The optimal exploitation return achievable by a meta-RL agent is 0.7. This corresponds to one step to the bus leading to the right goal, and taking this bus: $1 \times -0.3 + 1 = 0.7$.

For Problems 1 and 2, 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

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 2 hours.

```
python scripts/rl2.py rl2 -b environment=\"map\"
```

1. 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?
   Include your screenshot here.

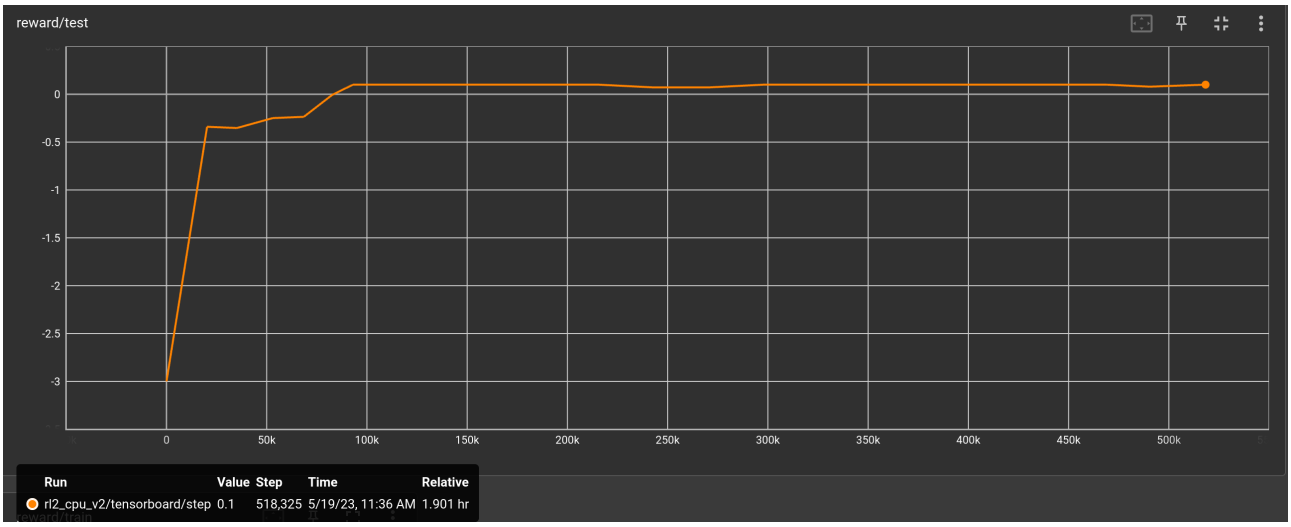

Figure 6: RL^2 test reward (orange)

Write your response here.
The average meta-testing exploitation returns RL^2 achieves after training is 0.1.

2. Examine the videos saved under `experiments/rl2/visualize/36000/`. Describe the exploration and exploitation behaviors that RL^2 learns. Does RL^2 achieve the optimal returns?
Write your response here.
This value of 0.1 corresponds to the trajectory described in Problem 0 Question 1. It seems like the agent is acting as if the exploration phase didn't occur. Exploitation and exploration appear to be uncorrelated/independent here.

Problem 2: DREAM

In Problem 1, 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}:

1. 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 $-\|g_\omega(\tau^{\text{exp}}) - \text{stop\_gradient}(z)\|_2^2$ 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:

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

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

2. 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 $\tau_{:t}^{\text{exp}}$ 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_t^{exp}, which we can maximize with standard reinforcement learning:

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

   Note that $\tau_{:t+1}^{\text{exp}}$ is equal to $\tau_{:t}^{\text{exp}}$ with the additional observations of (a_t, r_t, s_{t+1}). The exploration policy is optimized to maximize this intrinsic reward r_t^{exp} instead of the

extrinsic rewards r_t, which will maximize the objective $\mathcal{J}(\omega, \phi)$. Intuitively, r_t^{exp} 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 $\tau_{:t}^{\text{exp}}$.

Code: Implement the reward function $r_t^{\text{exp}}(a_t, r_t, s_{t+1}, \tau_{t-1}^{\text{exp}}; \mu)$ by filling in the `label_rewards` function of `EncoderDecoder` in `encoder_decoder.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).

3. Check your implementation by running DREAM :

```
python scripts/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 30 minutes of training. By around 40 minutes, the test returns curve should begin to look different from RL^2 . The total run should take around 2 hours. It may take 1-2 hours longer if training locally.
Include your screenshot here.

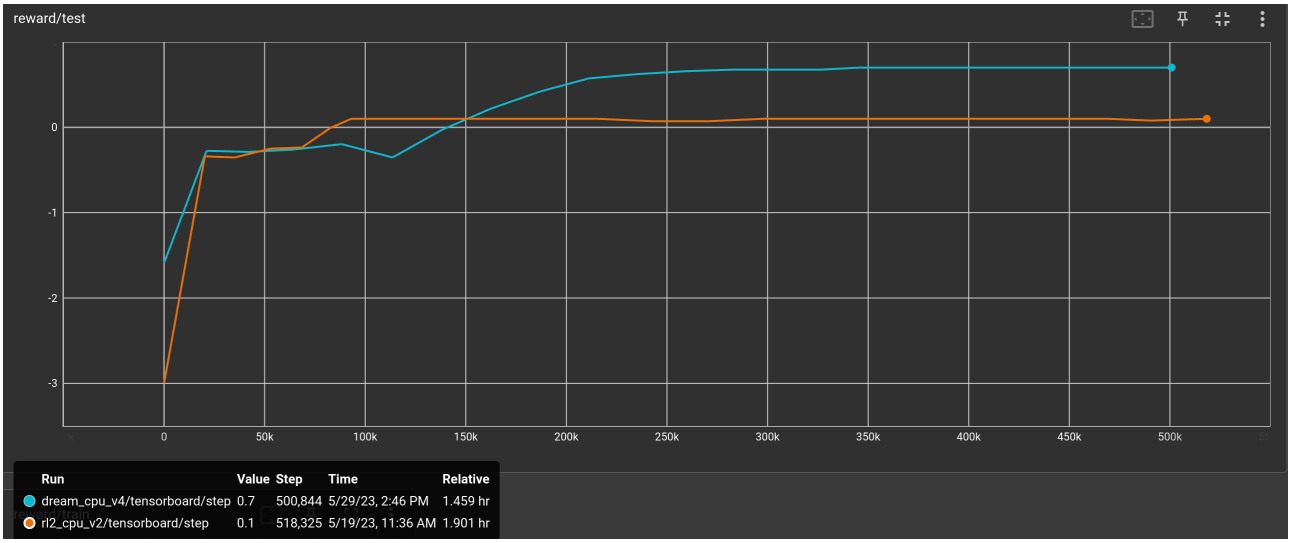

Figure 7: DREAM (blue) and RL^2 (orange) test reward

4. 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?
Write your response here.
Dream does achieve optimal results as the reward gets to 0.7, which is the expected value as described in Problem 0. As the exploration and exploitation phases are correlated/dependent here, in opposition to RL^2 , this value is expected.

5. 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.
   Write your response here.
   In the exploration phase, the agent goes directly to the map by moving two steps to the right. This is the expected behavior that takes the minimal number of steps, which is described in Problem 0.
   In the exploitation phase, the agent goes directly to the right bus, which leads to a reward of 0.7. This is the expected optimal behavior as described in Problem 0.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, Wojciech Zaremba. Hindsight Experience Replay. Neural Information Processing Systems (NeurIPS), 2017. https://arxiv.org/abs/1707.01495

[2] Leslie Kaelbling. Learning to Achieve Goals. International Joint Conferences on Artificial Intelligence (IJCAI), 1993. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.3077

[3] 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

[4] Evan Liu, Aditi Raghunathan, Percy Liang, Chelsea Finn. Explore then Execute: Adapting without Rewards via Factorized Meta-Reinforcement Learning. https://arxiv.org/abs/2008.02790v1