Assignment 4: Model-Based RL and Exploration
Due November 16th, 11:59 pm

1 Introduction

This assignment requires you to implement and evaluate a pipeline for model based learning and exploration. You will first implement a pipeline for model-based reinforcement learning which consists of two main parts: learning a dynamics function to model observed state transitions, and then using predictions from that model in some way to decide what to do. You will then implement an exploration method called random network distillation

2 Model-Based Reinforcement Learning

We will now provide a brief overview of model-based reinforcement learning (MBRL), and the specific type of MBRL you will be implementing in this homework.

MBRL consists primarily of two aspects: (1) learning a dynamics model and (2) using the learned dynamics models to plan and execute actions that minimize a cost function (or maximize a reward function).

2.1 Dynamics Model

In this assignment, you will learn a neural network dynamics model f_θ of the form

$$\hat{\Delta}_{t+1} = f_\theta(\mathbf{s}_t, \mathbf{a}_t) \tag{1}$$

which predicts the change in state given the current state and action. So given the prediction $\hat{\Delta}_{t+1}$, you can generate the next prediction with

$$\hat{\mathbf{s}}_{t+1} = \mathbf{s}_t + \hat{\Delta}_{t+1}. \tag{2}$$

See the previously referenced paper for intuition on why we might want our network to predict state differences, instead of directly predicting next state.

You will train f_θ in a standard supervised learning setup, by performing gradient descent on the following objective:

$$\mathcal{L}(\theta) = \sum_{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \in \mathcal{D}} \|(\mathbf{s}_{t+1} - \mathbf{s}_t) - f_\theta(\mathbf{s}_t, \mathbf{a}_t)\|_2^2 \tag{3}$$

$$= \sum_{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \in \mathcal{D}} \|\Delta_{t+1} - \hat{\Delta}_{t+1}\|_2^2 \tag{4}$$

In practice, it's helpful to normalize the target of a neural network. So in the code, we'll train the network to predict a *normalized* version of the change in state, as in

$$\mathcal{L}(\theta) = \sum_{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \in \mathcal{D}} \|\text{Normalize}(\mathbf{s}_{t+1} - \mathbf{s}_t) - f_\theta(\mathbf{s}_t, \mathbf{a}_t)\|_2^2. \tag{5}$$

Since f_θ is trained to predict the normalized state difference, you generate the next prediction with

$$\hat{\mathbf{s}}_{t+1} = \mathbf{s}_t + \text{Unnormalize}(f_\theta(\mathbf{s}_t, \mathbf{a}_t)). \tag{6}$$

2.2 Action Selection

Given the learned dynamics model, we now want to select and execute actions that minimize a known cost function (or maximize a known reward function). Ideally, you would calculate these actions by solving the following optimization:

$$\mathbf{a}_t^* = \arg\min_{\mathbf{a}_{t:\infty}} \sum_{t'=t}^{\infty} c(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}) \text{ where } \hat{\mathbf{s}}_{t'+1} = \hat{\mathbf{s}}_{t'} + f_\theta(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}). \tag{7}$$

However, solving Eqn. 7 is impractical for two reasons: (1) planning over an infinite sequence of actions is impossible and (2) the learned dynamics model is imperfect, so using it to plan in such an open-loop manner will lead to accumulating errors over time and planning far into the future will become very inaccurate.

Instead, one alternative is to solve the following gradient-free optimization problem:

$$\mathbf{A}^* = \arg\min_{\{\mathbf{A}^{(0)},\ldots,\mathbf{A}^{(K-1)}\}} \sum_{t'=t}^{t+H-1} c(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}) \text{ s.t. } \hat{\mathbf{s}}_{t'+1} = \hat{\mathbf{s}}_{t'} + f_\theta(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}), \tag{8}$$

in which $\mathbf{A}^{(k)} = (a_t^{(k)}, \ldots, a_{t+H-1}^{(k)})$ are each a random action sequence of length H. What Eqn. 8 says is to consider K random action sequences of length H, predict the result (i.e., future states) of taking each of these action sequences using the learned dynamics model f_θ, evaluate the cost/reward associated with each candidate action sequence, and select the best action sequence. Note that this approach only plans H steps into the future, which is desirable because it prevent accumulating model error, but is also limited because it may not be sufficient for solving long-horizon tasks.

A better alternative to this random-shooting optimization approach is the cross-entropy method (CEM), which is similar to random-shooting, but with iterative improvement of the distribution of actions that are sampled from. We first randomly initialize a set of K action sequences $\mathbf{A}^{(0)}, ..., A^{(K-1)}$, like in random-shooting. Then, we choose the J sequences with the highest predicted sum of discounted rewards as the "elite" action sequences. We then fit a diagonal Gaussian with the same mean and variance as the "elite" action sequences, and use this as our action sampling distribution for the next iteration. After repeating this process M times, we take the final mean of the Gaussian as the optimized action sequence. See Section 3.3 in this paper for more details.

Additionally, since our model is imperfect and things will never go perfectly according to plan, we adopt a model predictive control (MPC) approach, where at every time step we perform random-shooting or CEM to select the best H-step action sequence, but then we execute only the first action from that sequence before replanning again at the next time step using updated state information. This reduces the effect of compounding errors when using our approximate dynamics model to plan too far into the future.

2.3 On-Policy Data Collection

Although MBRL is in theory off-policy—meaning it can learn from any data—in practice it will perform poorly if you don't have on-policy data. In other words, if a model is trained on only randomly-collected data, it will (in most cases) be insufficient to describe the parts of the state space that we may actually care about. We can therefore use on-policy data collection in an iterative algorithm to improve overall task performance. This is summarized as follows:

Algorithm 1 Model-Based RL with On-Policy Data

Run base policy $\pi_0(\mathbf{a}_t, \mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})\}$
while *not done* **do**
  Train f_θ using $\mathcal{D}$ (Eqn. 4)
  $\mathbf{s}_t \leftarrow$ current agent state
  for *rollout number* $m = 0$ *to* M **do**
    for *timestep* $t = 0$ *to* T **do**
      $\mathbf{A}^* = \pi_{\text{MPC}}(\mathbf{a}_t, \mathbf{s}_t)$ where π_{MPC} is obtained from random-shooting or CEM
      $\mathbf{a}_t \leftarrow$ first action in $\mathbf{A}^*$
      Execute $\mathbf{a}_t$ and proceed to next state $\mathbf{s}_{t+1}$
      Add $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$ to $\mathcal{D}$
    end
  end
end

2.4 Ensembles

A simple and effective way to improve predictions is to use an ensemble of models. The idea is simple: rather than training one network f_θ to make predictions, we'll train N independently initialized networks $\{f_{\theta_n}\}_{n=1}^N$, and average their predictions to get your final predictions

$$f(s_t, a_t) = \frac{1}{N} \sum_{n=1}^{N} f_{\theta_n}(s_t, a_t). \tag{9}$$

In this assignment, you'll train an ensemble of networks and compare how different values of N effect the model's performance.

3 Random Network Distillation (RND) Algorithm

A common way of doing exploration is to visit states with a large prediction error of some quantity, for instance, the TD error or even random functions. The RND algorithm, as covered in Lecture 13, aims at encouraging exploration by asking the exploration policy to more frequently undertake transitions where the prediction error of a random neural network function is high. Formally, let $f_\theta^*(s')$ be a randomly chosen vector-valued function represented by a neural network. RND trains another neural network, $\hat{f}_\phi(s')$ to match the predictions of $f_\theta^*(s')$ under the distribution of datapoints in the buffer, as shown below:

$$\phi^* = \arg\min_\phi \ \mathbb{E}_{s,a,s' \sim \mathcal{D}} \left[\underbrace{||\hat{f}_\phi(s') - f_\theta^*(s')||}_{\mathcal{E}_\phi(s')} \right]. \tag{10}$$

If a transition (s, a, s') is in the distribution of the data buffer, the prediction error $\mathcal{E}_\phi(s')$ is expected to be small. On the other hand, for all unseen state-action tuples it is expected to be large. To utilize this prediction error as a reward bonus for exploration, RND trains two critics – an *exploitation critic*, $Q_R(s, a)$, and an *exploration critic*, $Q_\mathcal{E}(s, a)$, where the exploitation critic estimates the return of the policy under the actual reward function and the exploration critic estimates the return of the policy under the reward bonus. In practice, we normalize error before passing it into the exploration critic, as this value can vary widely in magnitude across states leading to poor optimization dynamics.

In this problem, we represent the random functions utilized by RND, $f_\theta^*(s')$ and $\hat{f}_\phi(s')$ via random neural networks. To prevent the neural networks from having zero prediction error right from the beginning, we initialize the networks using two different initialization schemes marked as `init_method_1` and `init_method_2` in `hw4_part2/exploration/rnd_model.py`.

3.1 Environments

Unlike previous assignments, we will consider some stochastic dynamics, discrete-action gridworld environments in this assignment. The three gridworld environments you will need for the graded part of this assignment are of varying difficulty: `easy`, `medium` and `hard`. A picture of these environments is shown below. The easy environment requires following two hallways with a right turn in the middle. The medium environment is a maze requiring multiple turns. The hard environment is a four-rooms task which requires navigating between multiple rooms through narrow passages to reach the goal location. We also provide a very hard environment for the bonus (optional) part of this assignment.

4 Code

You will implement the MBRL algorithm and RND algorithm described in the previous sections.

4.1 Overview

Obtain the code from https://github.com/cmuroboticsdrl/16831_F23_HW/tree/main/hw4.

For the first five problems of the assignment, you will add code to the following three files: `cs16831/hw4_part1/agents/mb`
`cs16831/hw4_part1/models/ff_model.py`, and
`cs16831/hw4_part1/policies/MPC_policy.py`. You will also need to edit these files by copying code from past homeworks or
Piazza: `cs16831/hw4_part1/infrastructure/rl_trainer.py` and
`cs16831/hw4_part1/infrastructure/utils.py`.

Problem 1

What you will implement:
Collect a large dataset by executing random actions. Train a neural network dynamics model on this fixed dataset and visualize the resulting predictions. The implementation that you will do here will be for training the dynamics model, and comparing its predictions against ground truth. You will be reusing the utilities you wrote for HW1 (or Piazza) for the data collection part (look for "get this from Piazza" markers).

What code files to fill in:

1. `cs16831/hw4_part1/agents/mb_agent.py`
2. `cs16831/hw4_part1/models/ff_model.py`
3. `cs16831/hw4_part1/infrastructure/utils.py`
4. `cs16831/hw4_part1/policies/MPC_policy.py` (just one line labeled `TODO(Q1)` for now)

What commands to run:

```
python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q1_cheetah_n500_arch1x32 --env_name
    cheetah-hw4_part1-v0 --add_sl_noise --n_iter 1 --batch_size_initial 20000 --
    num_agent_train_steps_per_iter 500 --n_layers 1 --size 32 --scalar_log_freq -1 --video_log_freq
     -1 --mpc_action_sampling_strategy 'random'

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q1_cheetah_n5_arch2x250 --env_name
    cheetah-hw4_part1-v0 --add_sl_noise --n_iter 1 --batch_size_initial 20000 --
    num_agent_train_steps_per_iter 5 --n_layers 2 --size 250 --scalar_log_freq -1 --video_log_freq
    -1 --mpc_action_sampling_strategy 'random'

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q1_cheetah_n500_arch2x250 --env_name
    cheetah-hw4_part1-v0 --add_sl_noise --n_iter 1 --batch_size_initial 20000 --
    num_agent_train_steps_per_iter 500 --n_layers 2 --size 250  --scalar_log_freq -1 --
    video_log_freq -1 --mpc_action_sampling_strategy 'random'
```

Your code will produce plots inside your logdir that illustrate your model prediction error (MPE). The code will also produce a plot of the losses over time. For the first command, the loss should go below 0.2 by the iteration 500. These plots illustrate, for a fixed action sequence, the difference between your model's predictions (red) and the ground-truth states (green). Each plot corresponds to a different state element, and the title reports the mean mean-squared-error across all state elements. As illustrated in the commands above, try different neural network architectures as well different amounts of training. Compare the results by looking at the loss values (i.e., itr_0_losses.png), the qualitative model predictions (i.e., itr_0_predictions.png), as well as the quantitative MPE values (i.e., in the title of itr_0_predictions.png).

What to submit: For this question, submit the qualitative model predictions (itr_0_predictions.png) for each of the three runs above. Comment on which model performs the best and why you think this might be the case.

Note that for these qualitative model prediction plots, we intend for you to just copy the png images produced by the code.

Problem 2

What will you implement:
Action selection using your learned dynamics model and a given reward function.

What code files to fill in:

1. `cs16831/hw4_part1/policies/MPC_policy.py` (all lines labeled `TODO(Q2)`, i.e. everything except the CEM section)

What commands to run:

```
python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q2_obstacles_singleiteration --env_name
    obstacles-rob831-v0 --add_sl_noise --num_agent_train_steps_per_iter 20 --n_iter 1 --
    batch_size_initial 5000 --batch_size 1000 --mpc_horizon 10 --video_log_freq -1 --
    mpc_action_sampling_strategy 'random'
```

Recall the overall flow of our rl_trainer.py. We first collect data with our policy (which starts as random), we then train our model on that collected data, and we then evaluate the resulting MPC policy (which now uses the trained model). To verify that your MPC is indeed doing reasonable action selection, run the command above and compare Train_AverageReturn (which was the execution of random actions) to Eval_AverageReturn (which was the execution of MPC using a model that was trained on the randomly collected training data). You can expect Train_AverageReturn to be around -160 and Eval_AverageReturn to be around -70 to -50.

What to submit:
Submit this run as part of your run_logs, and include a plot of Train_AverageReturn and Eval_AverageReturn in your pdf. Note that these will just be single dots on the plot, since we ran this for just 1 iteration.

Problem 3

What will you implement:

MBRL algorithm with on-policy data collection and iterative model training.

What code files to fill in:

None. You should already have done everything that you need, because rl_trainer.py already aggregates your collected data into a replay buffer. Thus, iterative training means to just train on our growing replay buffer while collecting new data at each iteration using the most newly trained model.

What commands to run:

```
python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q3_obstacles --env_name obstacles-
    hw4_part1-v0 --add_sl_noise --num_agent_train_steps_per_iter 20 --batch_size_initial 5000 --
    batch_size 1000 --mpc_horizon 10 --n_iter 12 --video_log_freq -1 --mpc_action_sampling_strategy
     'random'

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q3_reacher --env_name reacher-hw4_part1-
    v0 --add_sl_noise --mpc_horizon 10 --num_agent_train_steps_per_iter 1000 --batch_size_initial
    5000 --batch_size 5000 --n_iter 15 --video_log_freq -1 --mpc_action_sampling_strategy 'random'

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q3_cheetah --env_name cheetah-hw4_part1-
    v0 --mpc_horizon 15 --add_sl_noise --num_agent_train_steps_per_iter 1500 --batch_size_initial
    5000 --batch_size 5000 --n_iter 20 --video_log_freq -1 --mpc_action_sampling_strategy 'random'
```

You should expect rewards of around -25 to -20 for the obstacles env (takes 40 minutes), rewards of around -250 to -300 for the reacher env (takes 2-3 hours), and rewards of around 250-350 for the cheetah env takes 3-4 hours. All numbers assume no GPU.

What to submit:
Submit these runs as part of your run_logs, and include the performance plots in your pdf.

Problem 4

What will you implement:
You will compare the performance of your MBRL algorithm as a function of three hyperparameters: the number of models in your ensemble, the number of random action sequences considered during each action selection, and the MPC planning horizon.

What code files to fill in:
None.

What commands to run:

```
python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q4_reacher_horizon5 --env_name reacher-
    hw4_part1-v0 --add_sl_noise --mpc_horizon 5 --mpc_action_sampling_strategy 'random' --
    num_agent_train_steps_per_iter 1000 --batch_size 800 --n_iter 15 --video_log_freq -1 --
    mpc_action_sampling_strategy 'random'

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q4_reacher_horizon15 --env_name reacher-
    hw4_part1-v0 --add_sl_noise --mpc_horizon 15 --num_agent_train_steps_per_iter 1000 --batch_size
     800 --n_iter 15 --video_log_freq -1 --mpc_action_sampling_strategy 'random'

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q4_reacher_horizon30 --env_name reacher-
    hw4_part1-v0 --add_sl_noise --mpc_horizon 30 --num_agent_train_steps_per_iter 1000 --batch_size
     800 --n_iter 15 --video_log_freq -1 --mpc_action_sampling_strategy 'random'


python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q4_reacher_numseq100 --env_name reacher-
    hw4_part1-v0 --add_sl_noise --mpc_horizon 10 --num_agent_train_steps_per_iter 1000 --batch_size
     800 --n_iter 15 --mpc_num_action_sequences 100 --mpc_action_sampling_strategy 'random'

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q4_reacher_numseq1000 --env_name reacher-
    hw4_part1-v0 --add_sl_noise --mpc_horizon 10 --num_agent_train_steps_per_iter 1000 --batch_size
     800 --n_iter 15 --video_log_freq -1 --mpc_num_action_sequences 1000 --
    mpc_action_sampling_strategy 'random'

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q4_reacher_ensemble1 --env_name reacher-
    hw4_part1-v0 --ensemble_size 1 --add_sl_noise --mpc_horizon 10 --num_agent_train_steps_per_iter
     1000 --batch_size 800 --n_iter 15 --video_log_freq -1 --mpc_action_sampling_strategy 'random'

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q4_reacher_ensemble3 --env_name reacher-
    hw4_part1-v0 --ensemble_size 3 --add_sl_noise --mpc_horizon 10 --num_agent_train_steps_per_iter
     1000 --batch_size 800 --n_iter 15 --video_log_freq -1 --mpc_action_sampling_strategy 'random'

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q4_reacher_ensemble5 --env_name reacher-
    hw4_part1-v0 --ensemble_size 5 --add_sl_noise --mpc_horizon 10 --num_agent_train_steps_per_iter
     1000 --batch_size 800 --n_iter 15 --video_log_freq -1 --mpc_action_sampling_strategy 'random'
```

What to submit:
1) Submit these runs as part of your run_logs.

2) Include the following plots (as well as captions that describe your observed trends) of the following:

- effect of ensemble size
- effect of the number of candidate action sequences
- efffect of planning horizon

Be sure to include titles and legends on all of your plots, and be sure to generate your plots by extracting the corresponding performance numbers from your saved tensorboard eventfiles.

Problem 5

What will you implement:
You will compare the performance of your MBRL algorithm with action selecting performed by random-shooting (what you have done up to this point) and CEM.

Because CEM can be much slower than random-shooting, we will only run MBRL for 5 iterations for this problem. We will try two hyperparameter settings for CEM and compare their performance to random-shooting.

What code files to fill in:

1. `cs16831/hw4_part1/policies/MPC_policy.py`

What commands to run:

```
python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q5_cheetah_random --env_name 'cheetah-
    hw4_part1-v0' --mpc_horizon 15 --add_sl_noise --num_agent_train_steps_per_iter 1500 --
    batch_size_initial 5000 --batch_size 5000 --n_iter 5 --video_log_freq -1 --
    mpc_action_sampling_strategy 'random'

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q5_cheetah_cem_2 --env_name 'cheetah-
    hw4_part1-v0' --mpc_horizon 15 --add_sl_noise --num_agent_train_steps_per_iter 1500 --
    batch_size_initial 5000 --batch_size 5000 --n_iter 5 --video_log_freq -1 --
    mpc_action_sampling_strategy 'cem' --cem_iterations 2

python cs16831/hw4_part1/scripts/run_hw4_mb.py --exp_name q5_cheetah_cem_4 --env_name 'cheetah-
    hw4_part1-v0' --mpc_horizon 15 --add_sl_noise --num_agent_train_steps_per_iter 1500 --
    batch_size_initial 5000 --batch_size 5000 --n_iter 5 --video_log_freq -1 --
    mpc_action_sampling_strategy 'cem' --cem_iterations 4
```

You should expect rewards of 800 or higher when using CEM on the cheetah env. The final CEM run takes 2-3 hours on GPU, and over twice as long without GPU, so **we recommend getting started early and using a GPU (e.g. on Colab) for this problem!**

What to submit:
1) Submit these runs as part of your run_logs.

2) Include a plot comparing random-shooting with CEM, as well as captions that describe how CEM affects results for different numbers of sampling iterations (2 vs. 4).

Be sure to include a title and legend on your plot, and be sure to generate your plot by extracting the corresponding performance numbers from your saved tensorboard eventfiles.

Problem 6

What code files to fill from previous assignments:

1. `cs16831/hw4_part2/infrastructure/utils.py`
2. `cs16831/hw4_part2//infrastructure/rl_trainer.py`
3. `cs16831/hw4_part2/policies/MLP_policy.py`
4. `cs16831/hw4_part2//policies/argmax_policy.py`
5. `ccs16831/hw4_part2/critics/dqn_critic.py`

What code files to fill in:

1. `cs16831/hw4_part2/exploration/rnd_model.py`
2. `cs16831/hw4_part2/agents/explore_or_exploit_agent.py`

"Unsupervised" RND and exploration performance. Implement the RND algorithm and use the argmax policy with respect to the exploration critic to generate state-action tuples to populate the replay buffer for the algorithm. In the code, this happens before the number of iterations crosses `num_exploration_steps`, which is set to 10k by default. You need to collect data using the `ArgmaxPolicy` policy which chooses to perform actions that maximize the exploration critic value.

In experiment log directories, you will find heatmap plots visualizing the state density in the replay buffer, as well as other helpful visuals. You will find these in the experiment log directory, as they are output during training. **You need to your code on the environments PointmassEasy-v0 and PointmassHard-v0. Compare RND exploration to random (epsilon-greedy) exploration.** Include all the state density plots and a comparative evaluation of the learning curves obtained via RND and random exploration in your report.

```
python cs16831/hw4_part2/scripts/run_hw4_expl.py --env_name PointmassEasy-v0 --use_rnd
--unsupervised_exploration --exp_name q6_env1_rnd

python cs16831/hw4_part2/scripts/run_hw4_expl.py --env_name PointmassEasy-v0
--unsupervised_exploration --exp_name q6_env1_random

python cs16831/hw4_part2/scripts/run_hw4_expl.py --env_name PointmassHard-v0 --use_rnd
--unsupervised_exploration --exp_name q6_env2_rnd

python cs16831/hw4_part2/scripts/run_hw4_expl.py --env_name PointmassHard-v0
--unsupervised_exploration --exp_name q6_env2_random
```

The density of the state-action pairs on this `easy` environment should be, as expected, more uniformly spread over the reachable parts of the environment (that are not occupied by walls) with RND as compared to random exploration where most of the density would be concentrated around the starting state.

BONUS: You need to implement a separate exploration strategy of your choice. This can be an existing method, but feel free to design one of your own. To provide some starting ideas, you could try out count-based exploration methods (such as pseudo counts and EX2) or prediction error based approaches (such as exploring states with high TD error) or approaches that maximize marginal state entropy. Compare and contrast the chosen scheme with respect to RND, and specify possible reasons for the trends you see in performance. The heatmaps and trajectory visualizations will likely be helpful in understanding the behavior here.

```
python cs16831/hw4_part2/scripts/run_hw5_expl.py --env_name PointmassMedium-v0
--unsupervised_exploration <add arguments for your method> --exp_name q1_alg_med

python cs16831/hw4_part2/scripts/run_hw5_expl.py --env_name PointmassHard-v0
--unsupervised_exploration <add arguments for your method> --exp_name q1_alg_hard
```

Submission

4.2 Submitting the PDF

Your report should be a PDF document containing the plots and responses indicated in the questions above.

4.3 Submitting the Code and Logs

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

- A folder named `data` with all the experiment runs from this assignment. **Do not change the names originally assigned to the folders, as specified by `exp_name` in the instructions.** To minimize submissions size, please include runs with video logging disabled. If you would like to reuse your video logging runs, please see the script provided in `cs16831/hw4_part1/scripts/filter_events.py`.
- The `cs16831` folder with all the `.py` files, with the same names and directory structure as the original homework repository (not include the `data/` folder). A plotting script should also be submitted, which should be a python script (or jupyter notebook) such that running it can generate all plots from your pdf. This plotting script should extract its values directly from the experiments in your `run_logs` and should not have hardcoded reward values.

As an example, the unzipped version of your submission should result in the following file structure. **Make sure that the submit.zip file is below 15MB and that they include the prefix `hw4_mb_`.**

```
submit.zip
├── data
│   ├── hw4_q1_...
│   │   └── events.out.tfevents.1567529456.e3a096ac8ff4
│   └── ...
├── cs16831
│   ├── hw4_part1
│   │   ├── ...
│   │   └── ...
│   ├── hw4_part2
│   │   └── ...
│   └── ...
├── README.md
└── ...
```

If you are a Mac user, **do not use the default "Compress" option to create the zip**. It creates artifacts that the autograder does not like. You may use `zip -vr submit.zip submit -x "*.DS_Store"` from your terminal.

Turn in your assignment on Gradescope. Upload the zip file with your code and log files to **HW4 Code**, and upload the PDF of your report to **HW4**.