# Assignment 5: Exploration Strategies and Offline Reinforcement Learning

**Due Tuesday, November 21, 11:59 pm**

## 1 Analysis

In this section, we will analyze how reward bonuses can be used to manage distribution shift in offline RL. We consider an MDP $M = (\mathcal{S}, \mathcal{A}, r, p)$ with offline data $\mathcal{D}$ collected by a policy $\pi_\beta$. We denote by $p_\pi$ the distribution over states induced by $\pi$.

As a motivating example, consider the soft actor-critic (SAC) algorithm discussed in HW 3. When updating, SAC adds an adjustment of $b(\mathbf{s}, \mathbf{a}) = -\log \pi(\mathbf{a} \mid \mathbf{s})$ to the target values $\mathbb{E}_{\pi(\mathbf{a}|\mathbf{s})}[Q(\mathbf{s}, \mathbf{a}) + b(\mathbf{s}, \mathbf{a})]$ to enforce a maximum entropy regularizer on the policy. Alternatively, we could impose a similar form of entropy regularization by adding the bonus directly to the reward. In expectation, we would optimize

$$\begin{aligned}\mathbb{E}_{s,a\sim p_\pi}[r(\mathbf{s}, \mathbf{a}) + \lambda b(\mathbf{s}, \mathbf{a})] &= \tfrac{1}{H} J(\pi) - \lambda\, \mathbb{E}_{s\sim p_\pi} \mathbb{E}_{\pi(\mathbf{a}|\mathbf{s})} \log \pi(\mathbf{a} \mid \mathbf{s}) \\ &= \tfrac{1}{H} J(\pi) + \lambda\, \mathbb{E}_{s\sim p_\pi} \mathcal{H}[\pi(\mathbf{a} \mid \mathbf{s})].\end{aligned}$$

Thus, entropy regularization can also be implemented by adding the bonus $b(\mathbf{s}, \mathbf{a})$ to rewards. In the following parts, you will show how a similar reward bonus can be used to constrain distribution shift in offline RL.

We wish to learn a $Q$ function and policy $\pi$ from the offline data $\mathcal{D}$ under some constraint $D(\pi, \pi_\beta) \le \varepsilon$ with the following update:

$$Q(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \mathbb{E}_{\mathbf{a}'\sim\pi}\left[Q\left(\mathbf{s}', \mathbf{a}'\right)\right] \tag{1}$$

$$\text{where} \quad \pi = \arg\max_\pi \mathbb{E}_{\mathbf{s},\mathbf{a}\sim p_\pi}[Q(\mathbf{s}, \mathbf{a})] \;\; \text{s.t. } D(\pi, \pi_\beta) \le \varepsilon. \tag{2}$$

Directly enforcing the constraint in (2) is challenging with the environment rewards $r(\mathbf{s}, \mathbf{a})$, so we will implicitly enforce the constraint with a Lagrangian, modifying the reward to $\bar{r}(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \lambda b(\mathbf{s}, \mathbf{a})$ in (1). The overall optimization then becomes:

$$Q(\mathbf{s}, \mathbf{a}) \leftarrow \bar{r}(\mathbf{s}, \mathbf{a}) + \mathbb{E}_{\mathbf{a}'\sim\pi}\left[Q\left(\mathbf{s}', \mathbf{a}'\right)\right] \tag{3}$$

$$\text{where} \quad \pi = \arg\max_\pi \mathbb{E}_{\mathbf{s},\mathbf{a}\sim p_\pi}[Q(\mathbf{s}, \mathbf{a})]. \tag{4}$$

You may assume that $\lambda > 0$ is selected appropriately to enforce the constraint as follows:

$$\left(\arg\max_\pi \mathbb{E}_{\mathbf{s},\mathbf{a}\sim p_\pi}[Q(\mathbf{s}, \mathbf{a})] - \lambda D(\pi, \pi_\beta)\right) = \left(\arg\max_\pi \mathbb{E}_{\mathbf{s},\mathbf{a}\sim p_\pi}[Q(\mathbf{s}, \mathbf{a})] \;\; \text{s.t. } D(\pi, \pi_\beta) \le \varepsilon\right).$$

You may also assume access to the distributions $\pi(\mathbf{a} \mid \mathbf{s})$ and $\pi_\beta(\mathbf{a} \mid \mathbf{s})$ in your answers.

1. Suppose we wish to learn $\pi$ under a KL-divergence constraint, i.e.,

   $$D(\pi, \pi_\beta) = \mathbb{E}_{\mathbf{s}\sim p_\pi} D_{KL}[\pi(\mathbf{a} \mid \mathbf{s}) \| \pi_\beta(\mathbf{a} \mid \mathbf{s})].$$

   How can we enforce this constraint by adding a bonus $b(\mathbf{s}, \mathbf{a})$ to the reward $\bar{r}(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \lambda b(\mathbf{s}, \mathbf{a})$? Write your answer as an expression for $b(\mathbf{s}, \mathbf{a})$.

2. The $f$-divergence is a generalization of the KL-divergence that can be defined for distributions $P$ and $Q$ by

   $$D_f[P \| Q] = \int Q(x) f\left(\frac{P(x)}{Q(x)}\right) \mathrm{d}x$$

   where $f$ is a convex function with zero at 1. We can state an $f$-divergence policy constraint as

   $$\begin{aligned}D(\pi, \pi_\beta) &= \mathbb{E}_{\mathbf{s}\sim p_\pi} D_f[\pi(\mathbf{a} \mid \mathbf{s}) \| \pi_\beta(\mathbf{a} \mid \mathbf{s})] \\ &= \mathbb{E}_{\mathbf{s}\sim p_\pi} \mathbb{E}_{\pi_\beta(\mathbf{a}|\mathbf{s})} f\left(\frac{\pi(\mathbf{a} \mid \mathbf{s})}{\pi_\beta(\mathbf{a} \mid \mathbf{s})}\right).\end{aligned}$$

The $f$-divergence constraint allows us to specify many alternative constraints on the divergence between $\pi$ and $\pi_\beta$. For example, by taking $f(x) = \frac{1}{2}|x-1|$, the $f$-divergence becomes equivalent to total variation distance (TVD), and by taking $f(x) = -(x+1)\log\left(\frac{x+1}{2}\right) + x\log x$ it reduces to the Jensen-Shannon divergence (JSD). When $f(x) = x\log x$ we recover the standard KL divergence.

How can you extend your answer from part (1) to account for an arbitrary $f$-divergence? Your answer should be a more general alternate expression for $b(\mathbf{s}, \mathbf{a})$ in terms of $f$.

3. Suppose $M$ has a finite horizon $H$ and we want to constrain divergence in the distribution of *trajectories of states* under $\pi$ and $\pi_\beta$. We can express the KL divergence between the (state) trajectory distributions for $\tau = (\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_H)$ as follows:

$$D(\pi, \pi_\beta) = D_{KL}[p_\pi(\tau) \| p_{\pi_\beta}(\tau)].$$

What expression for $b(\mathbf{s}, \mathbf{a})$ enforces this constraint? You may assume access to the dynamics $p(\mathbf{s}' \mid \mathbf{s}, \mathbf{a})$. [*Hint: marginalize over actions to get a bonus* $b(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ *that depends on* $\mathbf{s}'$*. How can this be converted to a bonus* $b(\mathbf{s}, \mathbf{a})$ *that can be used with* $r(\mathbf{s}, \mathbf{a})$*?*]

# 2 Coding

This section requires you to implement and evaluate a pipeline for exploration and offline learning. You will first implement an exploration method called random network distillation (RND) and collect data using this exploration procedure, then perform offline training on the data collected via RND using conservative Q-learning (CQL), Advantage Weighted Actor Critic (AWAC), and Implicit Q-Learning (IQL). This assignment will be easier to run on a CPU as we will be using gridworld domains of varying difficulties to train our agents.

The questions will require you to perform multiple runs of offline RL training, which can take quite a long time as we ask you to analyze the empirical significance of specific hyperparameters and thus sweep over them. Furthermore, depending on your implementation, you may find it necessary to tweak some of the parameters, such as learning rates or exploration schedules, which can also be very time consuming. We would **highly** recommend starting early on the coding to allocate enough time to finish the assignment effectively.

## 2.1 File overview

The starter code for this assignment can be found at

https://github.com/berkeleydeeprlcourse/homework_fall2023/tree/master/hw5

All files needed to run your code are in the `hw5` folder. You will be writing new code in the following files (all in the `hw5/cs285/agents` folder):

- `random_agent.py`
- `rnd_agent.py`
- `dqn_agent.py`
- `cql_agent.py`
- `awac_agent.py`
- `iql_agent.py`

## 2.2 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

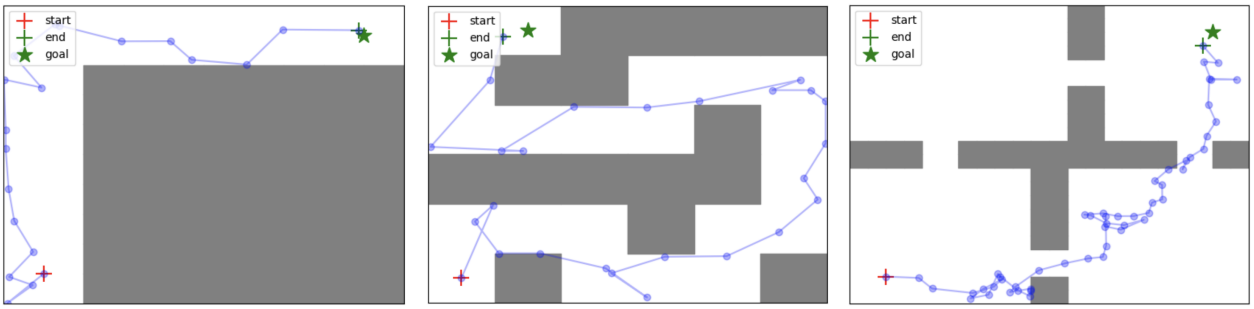


Figure 1: Figures depicting the `easy` (left), `medium` (middle) and `hard` (right) environments.

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.

## 2.3 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:

$$\phi^* = \arg\min_\phi \mathbb{E}_{s,a,s' \sim \mathcal{D}} \left[ \underbrace{||\hat{f}_\phi(s') - f_\theta^*(s')||}_{\mathcal{E}_\phi(s')} \right]. \tag{5}$$

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 target $f_\theta^*$ using an alternative initialization scheme in `agents/rnd_agent.py`.

## 2.4 Conservative Q-Learning (CQL) Algorithm

For the first portion of the offline RL part of this assignment, we will implement the conservative Q-learning (CQL) algorithm. The goal of CQL is to preventing overestimation of the policy value. In order to do that, a conservative, lower-bound Q-function is learned by additionally minimizing Q-values alongside a standard Bellman error objective. This is done by augmenting the Q-function training with a regularizer that minimizes the soft-maximum of the Q-values $\log\left(\sum_a \exp(Q(s, a))\right)$ and maximizes the Q-value on the state-action pair seen in the dataset, $Q(s, a)$. The overall CQL objective is given by the standard TD error objective augmented with the CQL regularizer weighted by $\alpha$: $\alpha \left[\frac{1}{N} \sum_{i=1}^N \left(\log\left(\sum_a \exp(Q(s_i, a))\right) - Q(s_i, a_i)\right)\right]$. You will tweak this value of $\alpha$ in later questions in this assignment.

## 2.5 Advantage Weighted Actor Critic (AWAC) Algorithm

For the second portion of the offline RL part of this assignment, we will implement the AWAC algorithm. This augments the training of the policy by utilizing the following actor update:

$$\theta \leftarrow \arg\max_{\theta} \mathbb{E}_{s,a\sim\mathcal{B}}\left[\log \pi_\theta(a|s)\exp(\frac{1}{\lambda}\mathcal{A}^{\pi_k}(s,a))\right]. \tag{6}$$

This update is similar to weighted behavior cloning (which it resolves to if the Q function is degenerate). But with a well-formed Q estimate, we weight the policy towards selecting actions that are high under our learnt q function. In the update above, the agent regresses onto high-advantage actions with a large weight, while almost ignoring low-advantage actions. This actor update amounts to weighted maximum likelihood (i.e., supervised learning), where the targets are obtained by re-weighting the state-action pairs observed in the current dataset by the predicted advantages from the learned critic, without explicitly learning any parametric behavior model, simply sampling (s, a) from the replay buffer $\beta$.

The Q function is learnt with a Temporal Difference (TD) Loss. The objective can be found below.

$$\mathbb{E}_D[(Q(s,a) - r(s,a) + \gamma\mathbb{E}_{a'\sim\pi}[Q_{\phi_{k-1}}(s',a')])^2] \tag{7}$$

Note that next actions $a'$ are sampled from the learned policy $\pi$, meaning that OOD actions will not be sampled if $\pi$ does a good job of fitting the (weighted) behavior policy.

## 2.6 Implicit Q-Learning (IQL) Algorithm

For the second portion of the offline RL part of this assignment, we will implement the IQL algorithm. IQL decouples the problem of learning the critic from the policy learning by using an *implicit* Bellman backup rather than explicitly considering the backup under a particular policy. It does this by learning an *expectile* of $Q$, which is a statistic similar to a quantile. This is a "soft" version of the *maximum* value attained by a distribution. For a random variable $X$, the expectile $m_\tau(X)$ is given as to minimize the following:

$$\arg\min_{m_\tau} \mathbb{E}_{x\sim X}[L_2(x - m_\tau)], L_2^\tau(\mu) = |\tau - \mathbb{1}\{\mu \leq 0\}|\mu^2$$

This backup will act optimistically with respect to actions taken in the dataset. To avoid being optimistic to *state* transitions, we need to learn a separate value function $V(s)$ that performs the optimism, then regress $Q(s,a) \leftarrow r + \gamma V(s')$ with a regular MSE loss. All together:

$$L_V(\phi) = \mathbb{E}_{(s,a)\sim D}[L_2^\tau(Q_\theta(s,a) - V_\phi(s))] \tag{8}$$

$$L_Q(\theta) = \mathbb{E}_{(s,a,s')\sim D}[(r(s,a) + \gamma V_\phi(s') - Q_\theta(s,a))^2] \tag{9}$$

Note that the critic learning process has two nice properties:

1. It never queries out-of-distribution actions (e.g. actions $a'$ from an arbitrary policy), which completely avoids the OOD overestimation issue.
2. It can be conducted without an actor update, so if you want you can first train a critic and then only train the actor at the end.

The actor update is decoupled from the critic update (hence *implicit* $Q$-learning), and is learned with the same objective as AWAC:

$$L_\pi(\psi) = -\mathbb{E}_{s,a\sim\mathcal{B}}\left[\log \pi_\psi(a|s)\exp(\frac{1}{\lambda}\mathcal{A}^{\pi_k}(s,a))\right]. \tag{10}$$

## 2.7 Relevant Literature

For more details about the algorithmic implementation, feel free to refer to the following papers: Conservative Q-Learning for Offline Reinforcement Learning (CQL), Accelerating Online Reinforcement Learning with Offline Datasets (AWAC), Offline Reinforcement Learning with Implicit Q-Learning (IQL), and Exploration by Random Network Distillation (RND).

### 2.8 Implementation

The first part in this assignment is to implement a working version of Random Network Distillation. The default code will run the `easy` environment with reasonable hyperparameter settings. Look for the `# TODO(student)` markers in the files listed above for detailed implementation instructions.

### 2.9 Evaluation

Once you have a working implementation of RND, CQL, AWAC, and IQL, you should prepare a report. The report should consist of one figure for each question below (each part has multiple questions). You should turn in the report as one PDF and a zip file with your code. If your code requires special instructions or dependencies to run, please include these in a file called `README` inside the zip file.

## 3 Exploration

In RL, our agent needs to see high-reward transitions at some point during training to understand that they exist. Your previous assignments (PG, DQN, SAC) have just used random exploration, possibly with some state-dependent noise (like in SAC). Here, you'll instead implement a policy that explicitly maximizes state coverage to explore the entire space.

Later, we'll use data collected from these exploration policies with several different offline RL algorithms.

### 3.1 Running a random policy

Just to get a sense for the three environments, implement `get_action` for the `RandomAgent` in `random_agent.py`. Run the random policy to generate random exploration in each of the three environments:

```
python cs285/scripts/run_hw5_explore.py \
    -cfg experiments/exploration/pointmass_easy_random.yaml
    --dataset_dir datasets/
python cs285/scripts/run_hw5_explore.py \
    -cfg experiments/exploration/pointmass_medium_random.yaml \
    --dataset_dir datasets/
python cs285/scripts/run_hw5_explore.py \
    -cfg experiments/exploration/pointmass_hard_random.yaml \
    --dataset_dir datasets/
```

These scripts will save visualizations of the final dataset in the `exploration` directory, as well as intermediate results in the Tensorboard logs. Include the final dataset visualization in your report.

**My Solution:** The visualizations are shown in following figures:

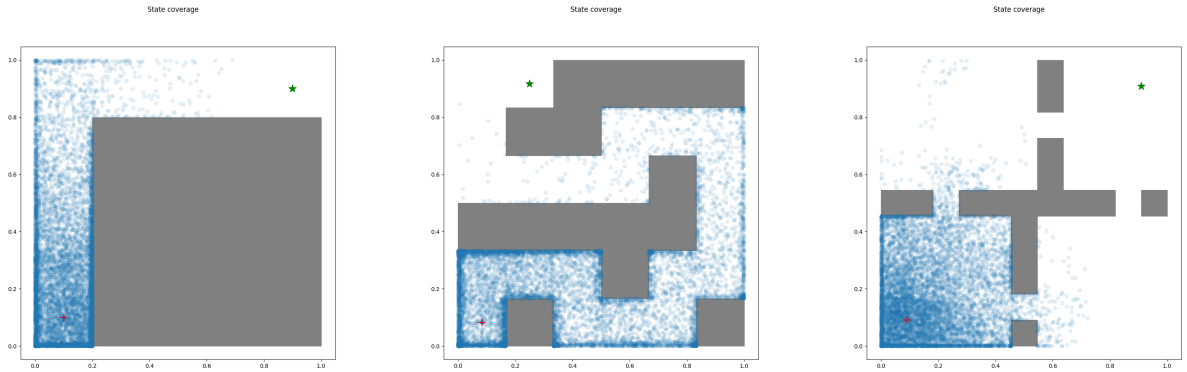


Figure 2: Visualizations of the final dataset in the `easy` (left), `medium` (middle), and `hard` (right) environments.

### 3.2 Random Network Distillation

**What you will implement**: the RND algorithm for exploration. You will need to change `cs285/agents/rnd_agent.py`. In addition, you should also thoroughly read through the training scheme in `run_hw5_explore.py`. It's very similar to the DQN training scheme you implemented in HW3.

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.

The experiment logs contain visualizations of RND error computed at each point in the environment as well as a scatter plot of visited states. As exploration progresses, a working RND algorithm should accumulate low error in all reachable states and high error in unreachable states (e.g. walls).

First, make sure your RND implementation works in the easy environment. Then, run it in all three environments:

```
python cs285/scripts/run_hw5_explore.py \
    -cfg experiments/exploration/pointmass_easy_rnd.yaml
    --dataset_dir datasets/
python cs285/scripts/run_hw5_explore.py \
    -cfg experiments/exploration/pointmass_medium_rnd.yaml \
    --dataset_dir datasets/
python cs285/scripts/run_hw5_explore.py \
    -cfg experiments/exploration/pointmass_hard_rnd.yaml \
    --dataset_dir datasets/
```

Again, include the visualizations in your final report. **These visualizations, particularly the RND error map, will be very helpful for debugging!**

**My Solution:** The visualizations are shown in the following figures:

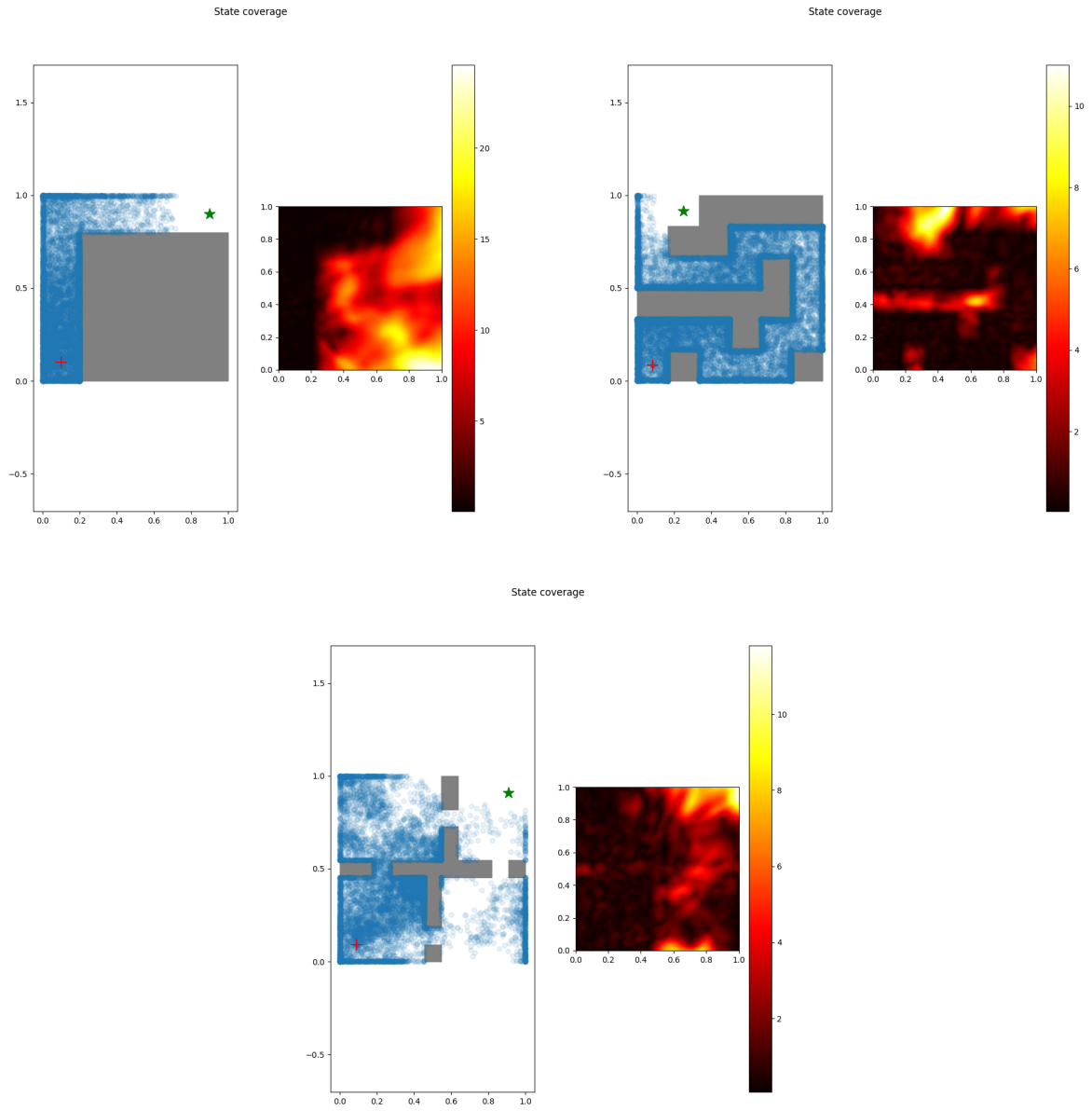


Figure 3: Visualizations of the final dataset in the `easy` (left), `medium` (middle), and `hard` (right) environments.

# 4 Offline RL

## 4.1 CQL

Now that we have implemented RND for collecting exploration data that is (likely) useful for performing exploitation, we will perform offline RL on this dataset and see how close the resulting policy is to the optimal policy. To begin, you will implement the conservative Q-learning algorithm in this `cs285/agents/cql_agent.py`.

Then, run both a standard DQN agent and your new CQL agent in the offline setting with the datasets you collected earlier.

```
python ./cs285/scripts/run_hw5_offline.py \
    -cfg experiments/offline/pointmass_easy_cql.yaml \
```

```
    --dataset_dir datasets
python ./cs285/scripts/run_hw5_offline.py \
    -cfg experiments/offline/pointmass_medium_dqn.yaml \
    --dataset_dir datasets

python ./cs285/scripts/run_hw5_offline.py \
    -cfg experiments/offline/pointmass_easy_cql.yaml \
    --dataset_dir datasets
python ./cs285/scripts/run_hw5_offline.py \
    -cfg experiments/offline/pointmass_medium_dqn.yaml \
    --dataset_dir datasets
```

On the Medium environment, create several experiment variations in which the value of the $\alpha$ parameter is varied, from $\alpha = 0$ (equivalent to DQN) to $\alpha = 10$. Show both resulting $Q$-values and evaluation performances in a plot. In the caption, describe how the $\alpha$ parameter affects training and performance in offline RL.

**My Solution:** For the first part, we compare the rewards and $Q$-values of DQN agent and CQL agent in the offline settings, with easy and medium environments. The results are shown in the following figures:

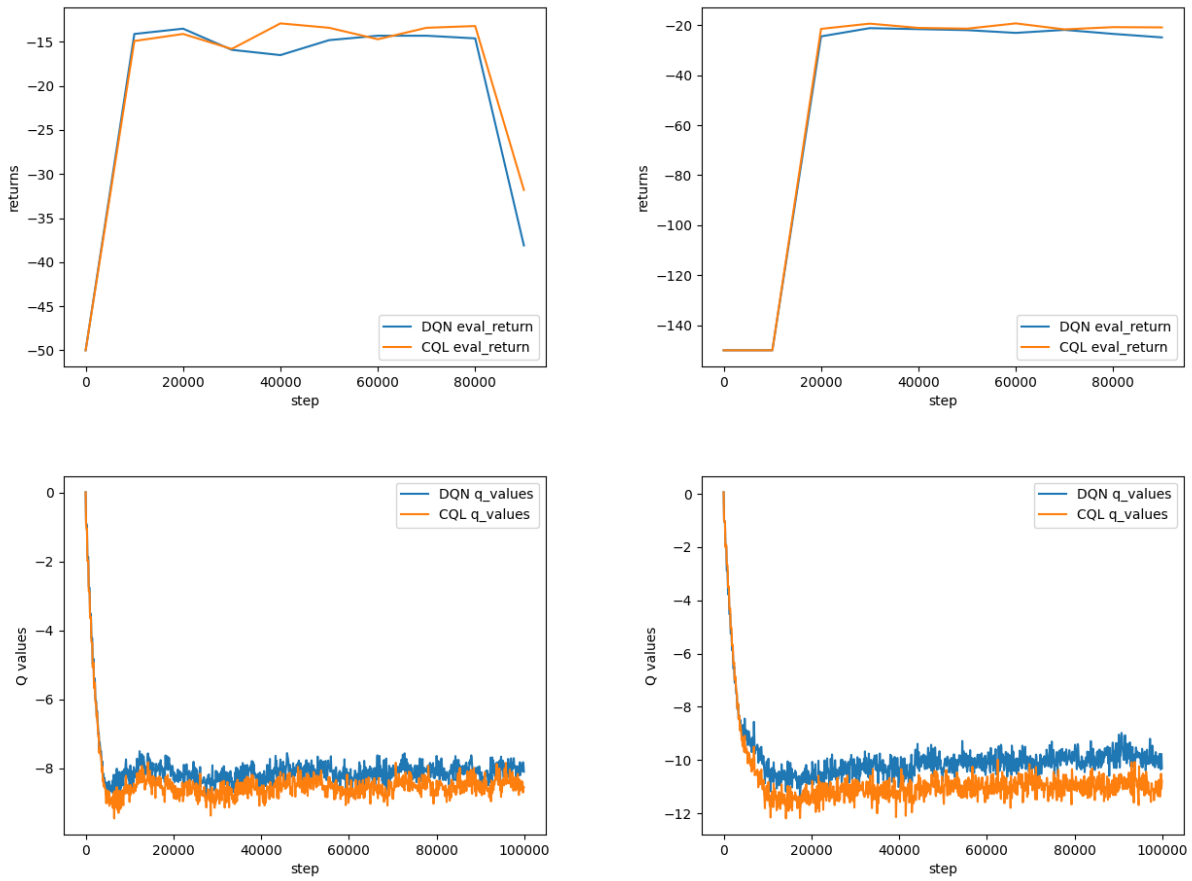


Figure 4: Comparison of rewards (top) and $Q$-values (bottom) of CQL agent and DQN agent in the offline settings, with easy (left) and medium (right) environments.

For the study on $\alpha$, I indeed have to mention that, for a proper comparison, $\alpha$ should range from 0 to 1.0, instead of 0 to 10. With this change, the results are shown in the following figure.

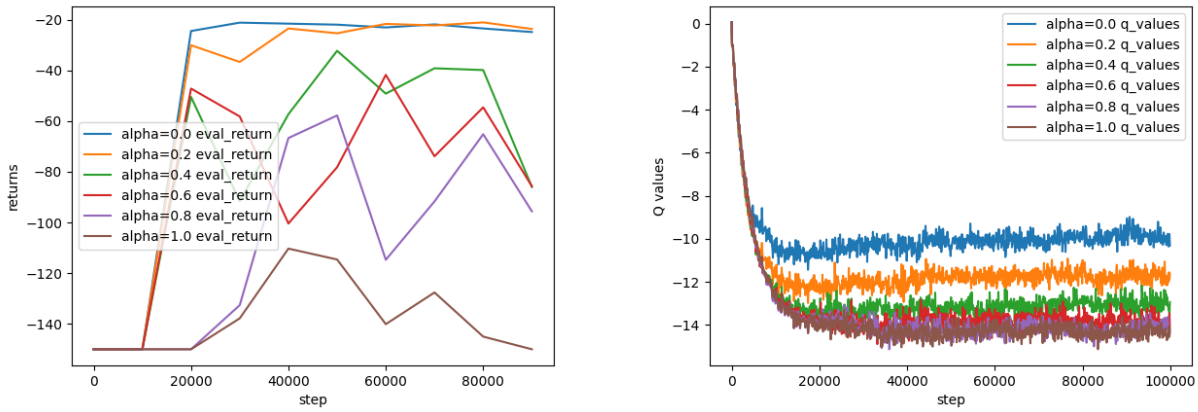


Figure 5: Comparison of rewards and $Q$-values of CQL agent with different $\alpha$ values in the offline settings, with medium environment.

From the figure, it can be found that $\alpha \to 0$ leads to the best results. This may be a little disappointing, since the CQL agent is expected to significantly outperform the DQN agent (which corresponds to $\alpha = 0$).

## 4.2 Policy Constraint Methods: IQL and AWAC

While CQL learns an actor via a modification to actor-critic algorithms like DQN that regularizes actions towards those found in the dataset by decreasing OOD $Q$-values, AWAC learns an in-distribution policy directly by performing weighted behavior cloning on the dataset.

Implement AWAC in `cs285/agents/awac_agent.py`, and IQL in `cs285/agents/iql_agent.py`. Compare them using the IQL and AWAC configuration files in the `experiments` directory. Report evaluation curves for both approaches.

**My Solution:** Here are the reward curve for AWAC and IQL agents in the medium and hard environment:

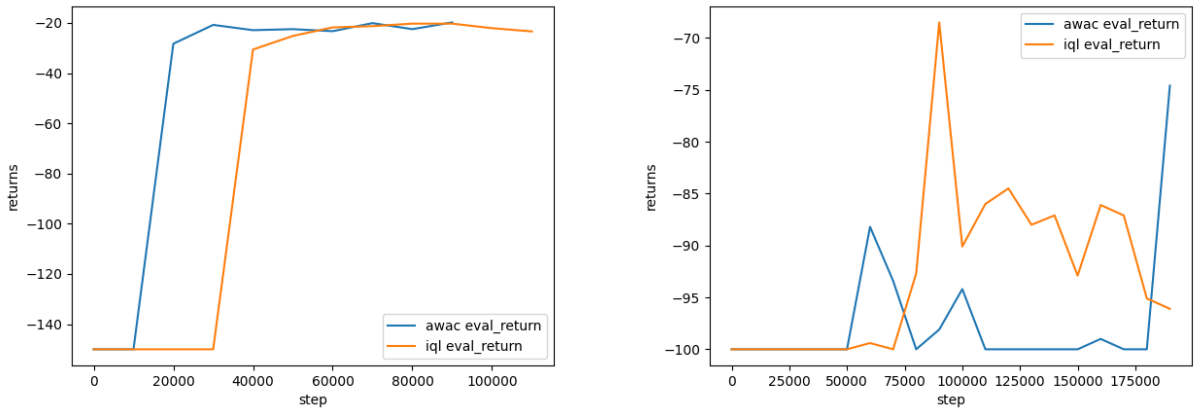


Figure 6: Rewards of AWAC agent and IQL agent in the offline settings, with medium (left) and hard (right) environments.

During the experiment, it is worth noting that the correct temperature should be around 0.1, instead of the initial value of 10.0. **However, for simplicity, I didn't change the configuration file; instead, I just change the code from `*1/temperature` to `*temperature` in the `awac_agent.py` file.** (Sorry for that)

### 4.3 Data ablations

Finally, compare the performance of offline RL under several different sizes of dataset. Run RND with `total_steps` 1000, 5000, 10000, and 20000 on either the Medium or Hard environment, creating a new dataset for each variation (you will need to make several .yaml config files for this). Then, train a CQL agent on each dataset and report its performance as well as evaluation curves.

**My Solution:** The results are shown in the following figure:

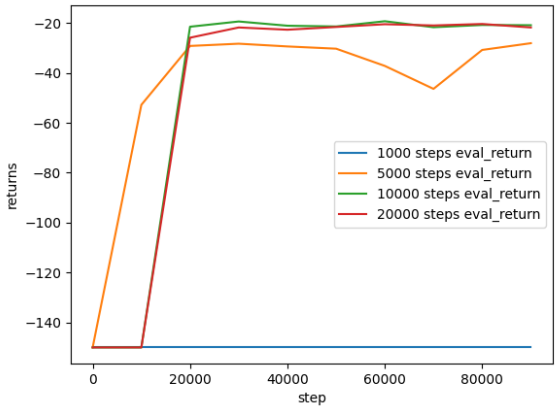


Figure 7: Rewards of CQL agent in the offline settings, with medium (left) and hard (right) environments, with different dataset sizes.

Clearly, the performance of the CQL agent increases with the dataset size. However, the improvement is not significant when the dataset size is larger than 5000.

## 5 Online Fine-Tuning

So far we only support training an algorithm purely offline, using data collected in a previous run. In `run_hw5_finetune.py`, implement online fine-tuning by first loading an offline dataset and taking a fixed number of training steps with an offline RL algorithm of your choice (IQL, CQL, or AWAC) and then switching to online learning while keeping all of the data in the dataset to initialize your replay buffer.

Report results as evaluation returns, clearly indicating the point at which online fine-tuning begins. An example configuration is provided for you in `experiments/finetune/pointmass_hard_cql_finetune.yaml`.

With online fine-tuning, your policy should be able to (stably) reach high reward (at least -20) on `PointmassHard-v0`.

**My Solution:** The results are shown in the following figure:

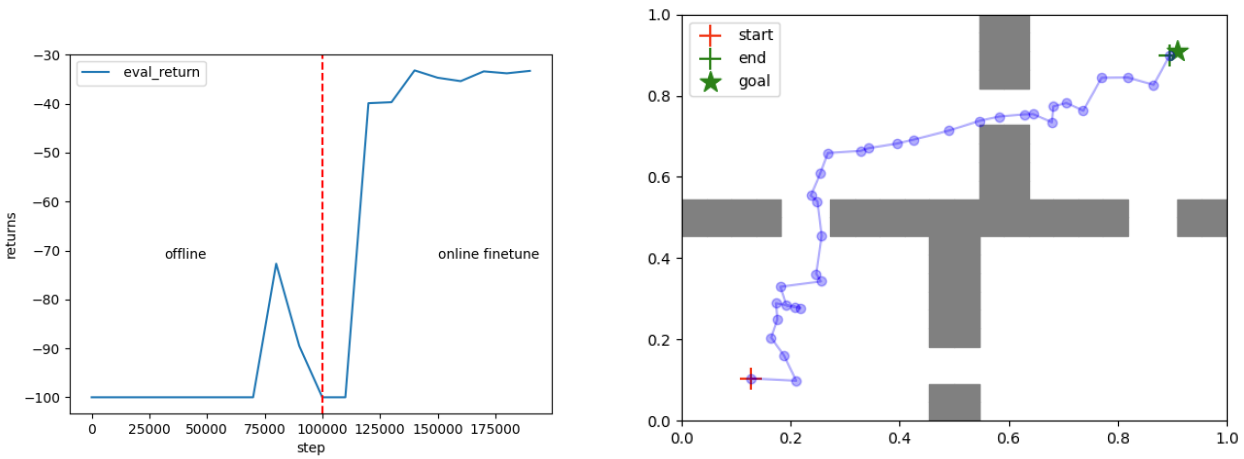


Figure 8: Rewards of CQL agent in the online fine-tuning settings (left) and an example succeeding image (right), within the hard environment.

I can't make it to reach −20 since I am too lazy to tune the hyperparameters. However, the results are still acceptable.

# 6 Submitting the code and experiment runs

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. Video logging is not utilized in this assignment, as visualizations are provided through plots, which are outputted during training.**
- The `cs285` folder with all the `.py` files, with the same names and directory structure as the original homework repository (excluding the `data` folder). Also include any special instructions we need to run in order to produce each of your figures or tables (e.g. "run python myassignment.py -sec2q1" to generate the result for Section 2 Question 1) in the form of a README file.

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 **HW5 Code**, and upload the PDF of your report to **HW5**.

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 `q1_`, `q2_`, `q3_`, etc.**

```
submit.zip
├── data
│   ├── q1...
│   │   └── events.out.tfevents.1567529456.e3a096ac8ff4
│   └── ...
├── cs285
│   └── ...
└── ...
```