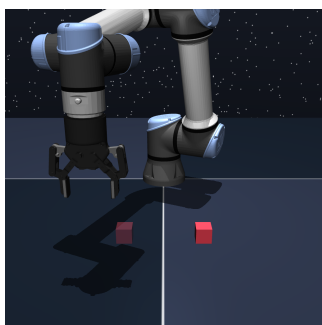


Assignment 5: Offline RL

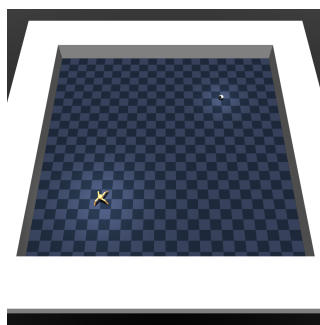
Due April 19, 11:59 pm

1 Introduction

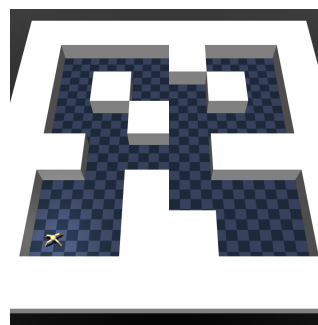
In this assignment, you will implement three offline reinforcement learning (RL) algorithms and evaluate them on continuous control tasks. In part 1 and part 2, you will implement two offline RL algorithms that train Gaussian policies: SAC+BC (Haarnoja et al., 2018a; Fujimoto and Gu, 2021) and IQL (Kostrikov et al., 2022). In part 3, you will implement an offline RL algorithm that trains expressive flow policies: FQL (Park et al., 2025b). You will tune and evaluate these algorithms on simulated robotic tasks from the OGBench task suite (Park et al., 2025a). These tasks have continuous state and action spaces (with actions bounded in $[-1, 1]$).



cube-single



antsoccer-arena



antmaze-medium

This assignment may require a non-trivial amount of compute for training and hyperparameter tuning. Each individual training run may take up to about 6 hours, and the total compute required for hyperparameter tuning across multiple algorithms and tasks can add up quickly. Hence, we recommend starting early and planning your experiments appropriately to make the best use of your compute resources.

2 Offline RL with Behavioral Regularization

One straightforward approach to offline RL is to regularize the learned policy by adding a behavioral cloning (BC) term to the policy loss of an existing off-policy RL algorithm, such as DDPG (Lillicrap et al., 2016), TD3 (Fujimoto et al., 2018), or SAC (Haarnoja et al., 2018a). This regularization makes the policy not deviate too far from the dataset's behavioral action distribution, mitigating the distributional shift issue in offline RL. This enables existing off-policy RL algorithms to be used in the offline setting.

TD3+BC (Fujimoto and Gu, 2021) is one representative example of this approach, which adds a BC term to an existing off-policy actor-critic algorithm, TD3 (Fujimoto et al., 2018). In this part of the assignment, you will implement a variant of TD3+BC called SAC+BC, which uses SAC (Haarnoja et al., 2018a) as the base algorithm instead of TD3. The difference between TD3+BC and SAC+BC is generally not significant, but the latter allows us to reuse the SAC implementation from the previous assignment.

2.1 Algorithm

SAC+BC trains two Q functions $Q_1, Q_2 : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, a Gaussian policy $\pi : \mathcal{S} \rightarrow \Delta(\mathcal{A})$, and a learnable entropy coefficient $\beta > 0$, where $\Delta(\mathcal{A})$ denotes the set of probability distributions over the action space $\mathcal{A}$.

The Q functions are trained to minimize the following Bellman error:

$$\mathcal{L}(Q) = \sum_{i=1}^2 \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}, a' \sim \pi(\cdot|s')} [(Q_i(s,a) - y)^2], \quad (1)$$

$$\text{where } y = r + \frac{\gamma}{2} \sum_{j=1}^2 \bar{Q}_j(s', a'), \quad (2)$$

where $\bar{Q}_1$ and $\bar{Q}_2$ are target networks for Q_1 and Q_2 , respectively. The target networks are updated with Polyak averaging (*i.e.*, soft updates) after every gradient step. We omit the entropy term in the Bellman backup (deviating from the original SAC algorithm), as it generally works better in offline RL settings. We also use the average (instead of the minimum) of the two target Q values in the Bellman backup, which often works better in practice.

The policy is trained to minimize the following loss:

$$\mathcal{L}(\pi) = \mathbb{E}_{(s,a) \sim \mathcal{D}, a^\pi \sim \pi(\cdot|s)} \left[-\frac{1}{2} \sum_{i=1}^2 Q_i(s, a^\pi) + \alpha \cdot \frac{1}{|\mathcal{A}|} \|a - a^\pi\|_2^2 + \beta \cdot \log \pi(a^\pi | s) \right], \quad (3)$$

where α is a hyperparameter that controls the strength of the behavioral cloning term. Intuitively, the first term maximizes the Q values, the second term regularizes the policy to dataset actions, and the third term maximizes the entropy of the policy. The main difference from the original SAC actor loss is the addition of the behavioral cloning term (the second term). As in SAC, the policy is trained via the reparameterization trick.

Finally, the (learnable) entropy coefficient β is trained via dual gradient descent to match a target entropy $\bar{\mathcal{H}} \in \mathbb{R}$, by minimizing the following loss:

$$\mathcal{L}(\beta) = \mathbb{E}_{s \sim \mathcal{D}, a^\pi \sim \pi(\cdot|s)} [\beta \cdot (-\log \pi(a^\pi | s) - \bar{\mathcal{H}})]. \quad (4)$$

In practice, we typically set the target entropy to $\bar{\mathcal{H}} = -|\mathcal{A}|/2$, where $|\mathcal{A}|$ is the action dimension. If you're not familiar with this automatic entropy tuning technique (or dual gradient descent in general), we recommend reading Section 6 of the paper by Haarnoja et al. (2018b). In this assignment, we don't ask you to implement automatic entropy tuning, and provide an implementation of it in the starter code.

The most important hyperparameter in SAC+BC (and any other behavior-regularized methods in general) is the BC coefficient α . The performance is generally *highly* sensitive to the choice of α , and it needs to be tuned carefully for each task. One of the main goals of this assignment is to understand the effect of α on the performance of behavior-regularized offline RL methods, and to have hands-on experience tuning it.

2.2 Implementation

The first part of this assignment is to implement SAC+BC and tune its hyperparameter α on two continuous control tasks.

The starter code for this assignment can be found at

https://github.com/berkeleydeeprlcourse/homework_spring2026/tree/main/hw5

You will implement SAC+BC in `src/agents/sacbc_agent.py`. We recommend starting by reading the following files thoroughly:

- `src/scripts/run.py` defines the main training loop.
- `src/networks/rl_networks.py` defines the neural network architectures for the Q functions and the policy.
- `src/configs/sacbc_config.py` builds networks and generates configuration for SAC+BC.

There is a new package requirement for the tasks in this assignment, `ogbench`, beyond the standard packages used in previous assignments. Make sure to install all packages properly with `uv` before starting the assignment.

What you'll need to do:

- Implement SAC+BC by filling in TODOs in `src/agents/sacbc_agent.py`.
- Tune the BC coefficient α to achieve the following performance at 1M training steps on OGBench tasks:

- Success rate above 75% on `cube-single-play-singletask-task1-v0`

```
uv run src/scripts/run.py --run_group=q1 --base_config=sacbc --env_name=cube-
single-play-singletask-task1-v0 --seed=<YOUR SEED> --alpha=<YOUR ALPHA>
```

- Success rate above 5% on `antsoccer-arena-navigate-singletask-task1-v0`

```
uv run src/scripts/run.py --run_group=q1 --base_config=sacbc --env_name=
antsoccer-arena-navigate-singletask-task1-v0 --seed=<YOUR SEED> --alpha=<
YOUR ALPHA>
```

- (Only for debugging, optional) Success rate above 80% on `antmaze-medium-navigate-singletask-task1-v0`

```
uv run src/scripts/run.py --run_group=q1 --base_config=sacbc --env_name=antmaze-
medium-navigate-singletask-task1-v0 --seed=<YOUR SEED> --alpha=<YOUR ALPHA>
```

Deliverables:

- Your implementation of SAC+BC.
- Your logs of the best-performing SAC+BC agents on the following tasks:
 - `cube-single-play-singletask-task1-v0`
 - `antsoccer-arena-navigate-singletask-task1-v0`
- A brief report including:
 - Training curves (training steps vs. success rate) of your best SAC+BC agents on the above two tasks.
 - Training curves with different values of α (of your choice, at least 3 different values) on `cube-single`.

Tips:

- We recommend starting to tune α from $\{30, 100, 300, 1000\}$ on `cube-single` and $\{1, 3, 10, 30\}$ on `antsoccer-arena` and `antmaze-medium`. However, the optimal value of α may differ substantially from these suggestions depending on your implementation, so feel free to explore other values as well. In particular, depending on whether you take the **mean** or **sum** over the action dimension in the BC term of the policy loss, the optimal value of α may differ by a factor of the action dimension (the α suggestions above are based on taking the **mean** over the action dimension; see Equation (3)).
- You may use `antmaze-medium` as a debugging task. With the correct implementation of SAC+BC and the best choice of α (and with the default hyperparameters), you should be able to achieve a success rate above 80% at 100K training steps.
- It is normal to achieve a low success rate on `antsoccer-arena` (but it should be non-zero), since this task is more challenging than the other two.
- When tuning α , one of the most useful training metrics is the mean squared error (MSE) between the dataset actions and the policy actions. A good sanity check is to ensure that the MSE decreases as α increases (up to the point where the policy loss is dominated by the BC term). Generally, the best value of α is achieved when the MSE is **only slightly above the minimum possible MSE** (achieved by behavioral cloning alone).

- Other useful metrics are `q_min` and `q_max`, the minimum and maximum Q values over the batch. In our environments, the reward is always either -1 or 0 at each step, so both `q_min` and `q_max` should generally be bounded between $-\frac{1}{1-\gamma}$ and 0 . With the correct implementation of SAC+BC on `antmaze-medium` or `antsoccer-arena`, `q_max` should converge to 0 and `q_min` should converge to $-\frac{1}{1-\gamma} = -100$ (on `cube-single`, `q_min` may converge to a value between -50 and -70 instead of -100 due to the short-horizon nature of the task).
- Don't forget to incorporate the `done` signal in the Bellman backup, which is omitted in the value loss above. This applies to other algorithms in this assignment as well.
- When submitting your logs, feel free to use any random seed of your choice. In actual research (especially in offline RL), it is important to report results properly averaged over multiple random seeds to ensure statistical significance. However, in this assignment, we only ask for results from a single (best) random seed to reduce the amount of compute required.
- You may turn off `@torch.compile` when debugging your code.
- For hyperparameter sweeps, you may train more than one agent in parallel on a single GPU. **We highly recommend running 4 agents in parallel on a single GPU for all experiments in this assignment.** This will not significantly slow down training of each agent. We provide a helper script that allows you to easily run multiple agents in parallel, and see `README.md` for more details.
- Training 4 SAC+BC agents in parallel for 1M steps takes about 4–6 hours on a single NVIDIA T4 GPU and 3–5 hours on a single NVIDIA A10 GPU.
- See `README.md` in the starter code for more details on how to run the code on your machine and Modal.

3 Offline RL with Implicit Q-Learning

Implicit Q-learning (IQL) (Kostrikov et al., 2022) is a simple and stable offline RL algorithm that learns a value function via expectile regression and extracts a policy via advantage-weighted regression. Expectile regression in IQL's value loss enables taking the maximum over actions in the Bellman backup without querying the Q function with out-of-distribution actions. Advantage-weighted regression in IQL's policy loss enables stable extraction of a policy via weighted supervised learning (*i.e.*, weighted behavioral cloning). These two components make IQL particularly robust, stable, and easy to tune.

3.1 Algorithm

IQL trains two Q functions $Q_1, Q_2 : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, a value function $V : \mathcal{S} \rightarrow \mathbb{R}$, and a Gaussian policy $\pi : \mathcal{S} \rightarrow \Delta(\mathcal{A})$. The Q and value functions are trained to minimize the following losses, respectively:

$$\mathcal{L}(Q) = \sum_{i=1}^2 \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} [(Q_i(s,a) - r - \gamma V(s'))^2], \quad (5)$$

$$\mathcal{L}(V) = \mathbb{E}_{(s,a) \sim \mathcal{D}} \left[\ell_2^\tau \left(V(s) - \min_{i=1,2} \bar{Q}_i(s,a) \right) \right], \quad (6)$$

where $\bar{Q}_1$ and $\bar{Q}_2$ are target networks for Q_1 and Q_2 , respectively, $\ell_2^\tau(x) = |\tau - \mathbb{I}(x > 0)|x^2$ is the expectile loss with an expectile parameter $\tau \in [0.5, 1)$, and $\mathbb{I}(\cdot)$ is the 0-1 indicator function. Intuitively, the asymmetric expectile loss approximates the maximum operator over actions in the Bellman backup.

The IQL policy is trained to minimize the following advantage-weighted regression loss:

$$\mathcal{L}(\pi) = \mathbb{E}_{(s,a) \sim \mathcal{D}} \left[-\min(e^{\alpha A(s,a)}, M) \log \pi(a | s) \right], \quad (7)$$

$$\text{where } A(s,a) = \min_{i=1,2} Q_i(s,a) - V(s), \quad (8)$$

where $\alpha > 0$ is an inverse temperature parameter that controls the strength of advantage weighting, and $M > 0$ is a constant that clips the weights to improve stability. Clipping the exponentiated advantages is

crucial in preventing excessively large weights from dominating the policy loss. In practice, we typically use $M = 100$.

IQL has two important hyperparameters: the expectile τ and the inverse temperature α . The expectile τ interpolates between the SARSA operator ($\tau = 0.5$) and the Bellman optimality operator ($\tau \rightarrow 1$). The inverse temperature α interpolates between behavioral cloning ($\alpha = 0$) and greedy maximization of the Q function ($\alpha \rightarrow \infty$). While IQL has one more (main) hyperparameter than SAC+BC, it is generally less sensitive to the choice of hyperparameters, making it easier to tune in practice. In this assignment, we will use a fixed value of $\tau = 0.9$ and focus on the tuning of α .

3.2 Implementation

The second part of this assignment is to implement IQL and tune its hyperparameter α on two continuous control tasks.

What you'll need to do:

- Implement IQL by filling in TODOs in `src/agents/iql_agent.py`.
- Tune the inverse temperature α to achieve the following performance at 1M training steps on OGBench tasks:

- Success rate above 60% on `cube-single-play-singletask-task1-v0`

```
uv run src/scripts/run.py --run_group=q2 --base_config=iql --env_name=cube-
single-play-singletask-task1-v0 --seed=<YOUR SEED> --alpha=<YOUR ALPHA>
```

- Success rate above 5% on `antsoccer-arena-navigate-singletask-task1-v0`

```
uv run src/scripts/run.py --run_group=q2 --base_config=iql --env_name=antsoccer-
arena-navigate-singletask-task1-v0 --seed=<YOUR SEED> --alpha=<YOUR ALPHA>
```

- (Only for debugging, optional) Success rate above 70% on `antmaze-medium-navigate-singletask-task1-v0`

```
uv run src/scripts/run.py --run_group=q2 --base_config=iql --env_name=antmaze-
medium-navigate-singletask-task1-v0 --seed=<YOUR SEED> --alpha=<YOUR ALPHA>
```

Deliverables:

- Your implementation of IQL.
- Your logs of the best-performing IQL agents on the following tasks:
 - `cube-single-play-singletask-task1-v0`
 - `antsoccer-arena-navigate-singletask-task1-v0`
- A brief report including:
 - Training curves (training steps vs. success rate) of your best IQL agents on the above two tasks.
 - Training curves with different values of α (of your choice, at least 3 different values) on `cube-single`.
 - A short discussion (1–2 sentences) that compares SAC+BC and IQL in terms of (1) the best performance and (2) sensitivity to α .

Tips:

- We recommend starting to tune α from $\{1, 3, 10\}$.
- You may use `antmaze-medium` as a debugging task. With the correct implementation of IQL and the best choice of α (and with the default hyperparameters), you should be able to achieve a success rate above 60% at 200K training steps.

- Training 4 IQL agents in parallel for 1M steps takes about 3–5 hours on a single NVIDIA T4 GPU and 2–4 hours on a single NVIDIA A10 GPU.

4 Offline RL with Flow Policies

So far, you have implemented two offline RL algorithms that learn a unimodal Gaussian policy. While any Markov decision process admits an optimal deterministic (in particular, unimodal) policy in theory, it is often beneficial to consider more *expressive* policy classes in practice, especially in offline RL and data-driven RL settings.

There are several reasons for this. First, in offline RL, we do not just maximize the expected return but also constrain the learned policy to stay close to the behavior policy, either explicitly or implicitly. When the behavior policy is diverse and multimodal (which is often the case in practice), a unimodal Gaussian policy may not be able to represent multiple modes of the behavior policy. This potentially leads to suboptimal performance. Second, we often want to further fine-tune an offline RL policy with online rollouts. In this case, having a more expressive policy can improve exploration during fine-tuning, as the policy can capture multiple useful modes present in the offline dataset.

In this part, you will implement flow Q-learning (FQL) (Park et al., 2025b), an offline RL algorithm that learns an expressive *flow* policy. Before explaining FQL, we first introduce a simpler (yet naïve) approach to offline RL with flow policies, flow behavior-regularized actor-critic (FBRAC) (Wang et al., 2023; Park et al., 2025b). We will then see why FBRAC may not be ideal in practice, and how FQL addresses its limitations.

4.1 An Initial Attempt: FBRAC

Flow behavior-regularized actor-critic (FBRAC) (Wang et al., 2023; Park et al., 2025b) is an algorithm that extends SAC+BC to flow policies in a straightforward manner, by simply replacing the BC loss in SAC+BC with a flow-matching loss. This way, we can train a flow policy that maximizes the expected return while staying close to the behavioral distribution.

FBRAC trains two Q functions $Q_1, Q_2 : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ and a flow policy defined by a time-dependent vector field $v : \mathcal{S} \times \mathcal{A} \times [0, 1] \rightarrow \mathcal{A}$, where $\mathcal{A} = \mathbb{R}^{|\mathcal{A}|}$ is the action space. We denote the corresponding flow policy induced by v as $\pi_v : \mathcal{S} \times \mathbb{R}^{|\mathcal{A}|} \rightarrow \mathbb{R}^{|\mathcal{A}|}$, which is obtained by solving the ordinary differential equation (ODE) defined by v from 0 to 1 with the Euler method with $F = 10$ steps. This policy transforms the standard Gaussian distribution $\mathcal{N}(0, I_{|\mathcal{A}|})$ into the target action distribution in $\mathcal{A}$, where $I_{|\mathcal{A}|}$ is the identity matrix of size $|\mathcal{A}| \times |\mathcal{A}|$ (for action dimension $|\mathcal{A}|$).

The Q functions are trained with the standard Bellman loss:

$$\mathcal{L}(Q) = \sum_{i=1}^2 \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}, z \sim \mathcal{N}(0, I_{|\mathcal{A}|})} [(Q_i(s, a) - y)^2], \quad (9)$$

$$\text{where } y = r + \frac{\gamma}{2} \sum_{j=1}^2 \bar{Q}_j(s', \pi_v(s', z)), \quad (10)$$

where $\bar{Q}_1$ and $\bar{Q}_2$ are target networks for Q_1 and Q_2 , respectively. Note that the next action a' in the target value is computed by solving the ODE defined by π_v .

The flow policy is trained to minimize the following loss:

$$\mathcal{L}(v) = \mathbb{E}_{\substack{(s,a) \sim \mathcal{D}, \\ z \sim \mathcal{N}(0, I_{|\mathcal{A}|}), \\ t \sim \text{Unif}([0,1])}} \left[-\frac{1}{2} \sum_{i=1}^2 Q_i(s, \pi_v(s, z)) + \frac{\alpha}{|\mathcal{A}|} \|v(s, \tilde{a}, t) - (a - z)\|_2^2 \right], \quad (11)$$

$$\text{where } \tilde{a} = (1 - t)z + ta, \quad (12)$$

where $\text{Unif}([0, 1])$ denotes the uniform distribution over $[0, 1]$ and α is a hyperparameter that controls the strength of the flow-matching term. Intuitively, the FBRAC policy loss interpolates between Q maximization

(the first term) and behavioral flow matching (the second term), just like how SAC+BC interpolates between Q maximization and behavioral cloning.

While FBRAC is a simple and natural extension of SAC+BC to flow policies, there is a potential issue with it. Recall that $\pi_v(s, z)$ is obtained by solving the ODE defined by v . This means that, when we compute the gradient of the first term in Equation (11) with respect to the parameters of v , we need to backpropagate the gradients through the entire ODE integration process. This is an example of *backpropagation through time* (BPTT), which is generally unstable and computationally expensive in practice because it requires multiplying many (e.g., $(\# \text{ layers}) \times (\# \text{ ODE integration steps})$) Jacobian matrices together.

4.2 A Better Solution: FQL

Flow Q-learning (FQL) (Park et al., 2025b) provides a more stable and effective solution to flow-based offline RL, which addresses the BPTT issue by training an additional *one-step* policy. One-step policies are standard feedforward networks that directly map a state and noise to an action without any iterative process.



Here is the key idea of FQL. Instead of directly interpolating between Q maximization and flow matching as in FBRAC, which involves BPTT ((a) in the figure above), FQL interpolates between Q maximization and *behavioral distillation* with an additional one-step policy ((b) in the figure above). Since the one-step policy is a standard feedforward network, we can avoid BPTT when maximizing the Q function, while maintaining the expressivity and multimodality of the flow policy (note that the one-step policy is conditioned on the same noise variable as the flow policy, so it can also represent multimodal distributions). We provide more details below, and refer to Section 3 of the original paper (Park et al., 2025b) for further explanations.

4.3 Algorithm

FQL trains two Q functions $Q_1, Q_2 : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, a *behavioral* flow policy defined by a time-dependent vector field $v : \mathcal{S} \times \mathcal{A} \times [0, 1] \rightarrow \mathcal{A}$, and a *one-step* policy defined by a standard feedforward network $\pi_\omega : \mathcal{S} \times \mathbb{R}^{|\mathcal{A}|} \rightarrow \mathbb{R}^{|\mathcal{A}|}$, where $\mathcal{A} = \mathbb{R}^{|\mathcal{A}|}$ is the action space. As before, we denote the behavioral flow policy induced by v as $\pi_v : \mathcal{S} \times \mathbb{R}^{|\mathcal{A}|} \rightarrow \mathbb{R}^{|\mathcal{A}|}$.

The behavioral flow policy is trained with the standard flow-matching loss for behavioral cloning:

$$\mathcal{L}(v) = \mathbb{E}_{\substack{(s,a) \sim \mathcal{D}, \\ z \sim \mathcal{N}(0, I_{|\mathcal{A}|}), \\ t \sim \text{Unif}([0,1])}} \left[\frac{1}{|\mathcal{A}|} \|v(s, \tilde{a}, t) - (a - z)\|_2^2 \right], \quad (13)$$

$$\text{where } \tilde{a} = (1 - t)z + ta. \quad (14)$$

The Q functions are trained with the standard Bellman loss:

$$\mathcal{L}(Q) = \sum_{i=1}^2 \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}, z \sim \mathcal{N}(0, I_{|\mathcal{A}|})} [(Q_i(s, a) - y)^2], \quad (15)$$

$$\text{where } y = r + \frac{\gamma}{2} \sum_{j=1}^2 \bar{Q}_j(s', \pi_\omega(s', z)), \quad (16)$$

where $\bar{Q}_1$ and $\bar{Q}_2$ are target networks for Q_1 and Q_2 , respectively. Note that the policy used in the Bellman backup is the one-step policy π_ω , not the behavioral flow policy π_v .

Finally, the one-step policy is trained to minimize the following loss:

$$\mathcal{L}(\pi_\omega) = \mathbb{E}_{\substack{(s,a) \sim \mathcal{D}, \\ z \sim \mathcal{N}(0, I_{|\mathcal{A}|})}} \left[-\frac{1}{2} \sum_{i=1}^2 Q_i(s, \pi_\omega(s, z)) + \frac{\alpha}{|\mathcal{A}|} \|\pi_\omega(s, z) - \pi_v(s, z)\|_2^2 \right], \quad (17)$$

where α is a hyperparameter that controls the strength of the distillation term. Here, recall that π_ω is a standard feedforward network, and π_v is an ODE integration of the vector field v from 0 to 1. Gradients do not backpropagate through π_v in the distillation term. Intuitively, FQL’s one-step policy loss above interpolates between Q maximization (the first term) and distillation from the behavioral flow policy (the second term). Since FQL’s Q maximization term only involves the one-step policy π_ω , it is free from BPTT!

As in SAC+BC, the BC coefficient α is the most important hyperparameter in FQL, which needs to be tuned for each task.

4.4 Implementation

The final part of this assignment is to implement FQL and tune its hyperparameter α on two continuous control tasks.

What you’ll need to do:

- Implement FQL by filling in TODOs in `src/agents/fql_agent.py`.
- Tune the distillation coefficient α to achieve the following performance on OGBench tasks:
 - Final success rate above 80% on `cube-single-play-singletask-task1-v0` at 1M training steps

```
uv run src/scripts/run.py --run_group=q3 --base_config=fql --env_name=cube-
single-play-singletask-task1-v0 --seed=<YOUR SEED> --alpha=<YOUR ALPHA>
```

- Best success rate above 30% on `antsoccer-arena-navigate-singletask-task1-v0` within 1M training steps

```
uv run src/scripts/run.py --run_group=q3 --base_config=fql --env_name=antsoccer-
arena-navigate-singletask-task1-v0 --seed=<YOUR SEED> --alpha=<YOUR ALPHA>
```

- (Only for debugging, optional) Final success rate above 80% on `antmaze-medium-navigate-singletask-task1-v0` at 1M training steps

```
uv run src/scripts/run.py --run_group=q3 --base_config=fql --env_name=antmaze-
medium-navigate-singletask-task1-v0 --seed=<YOUR SEED> --alpha=<YOUR ALPHA>
```

Deliverables:

- Your implementation of FQL.
- Your logs of the best-performing FQL agents on the following tasks:
 - `cube-single-play-singletask-task1-v0`
 - `antsoccer-arena-navigate-singletask-task1-v0`
- A brief report including:
 - Training curves (training steps vs. success rate) of your best FQL agents on the above two tasks.

Tips:

- We recommend starting to tune α from $\{30, 100, 300, 1000\}$ on `cube-single` and $\{1, 3, 10, 30\}$ on `antsoccer-arena` and `antmaze-medium`. However, the optimal value of α may differ substantially from these suggestions depending on your implementation, so feel free to explore other values as well. In particular, as in SAC+BC, the α suggestions above are based on taking the **mean** (not **sum**) over the action dimension.

- You may use `antmaze-medium` as a debugging task. With the correct implementation of FQL and the best choice of α (and with the default hyperparameters), you should be able to achieve a success rate above 80% at 200K training steps.
- As always, it is useful to monitor the MSE between the dataset actions and the one-step policy actions when tuning α , and is helpful to look at the Q value statistics (*e.g.*, `q_min` and `q_max`) for sanity checks.
- Don't forget to clip the actions produced by the flow policy to be within the valid action range (*i.e.*, between -1 and 1) when feeding them to the Q functions.
- However, you should **not** clip the actions produced by the one-step policy in the distillation term of the one-step policy loss. Otherwise, the one-step policy cannot be corrected when it produces out-of-bound actions.
- Remember that the output of this algorithm is the one-step policy. Hence, you should use the one-step policy π_ω (not the behavioral flow policy π_v) for **both** evaluation and in the Bellman backup.
- To achieve strong performance on `antsoccer-arena`, it is important to use the **average** (instead of the minimum) of the two target Q values in the Bellman backup, as stated in Equation (16).
- Training 4 FQL agents in parallel for 1M steps takes about 5–7 hours on a single NVIDIA T4 GPU and 4–6 hours on a single NVIDIA A10 GPU.

5 Submitting the Code and Experiment Runs

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

- A directory named `exp` with all the experiment runs from this assignment. **You should choose your best run for each question based on the success rates achieved, and *rename* the directories accordingly as specified below.**
- The `src` folder with all the `.py` files, with the same names and directory structure as the original homework repository.

The unzipped version of your submission should have the following file structure. **Make sure that the `exp` directory has the exact structure as shown below and that the `submit.zip` file is below 100MB.**

```
submit.zip
├── exp
│   ├── q1
│   │   ├── antsoccer-arena
│   │   │   ├── agent.pt
│   │   │   ├── eval.csv
│   │   │   ├── flags.json
│   │   │   ├── log.pkl
│   │   │   └── train.csv
│   │   ├── cube-single
│   │   │   ├── agent.pt
│   │   │   ├── eval.csv
│   │   │   ├── flags.json
│   │   │   ├── log.pkl
│   │   │   └── train.csv
│   └── q2
│       ├── antsoccer-arena
│       │   ├── agent.pt
│       │   ├── eval.csv
│       │   ├── flags.json
│       │   ├── log.pkl
│       │   └── train.csv
│       ├── cube-single
│       │   ├── agent.pt
│       │   ├── eval.csv
│       │   ├── flags.json
│       │   ├── log.pkl
│       │   └── train.csv
│       └── q3
│           ├── antsoccer-arena
│           │   ├── agent.pt
│           │   ├── eval.csv
│           │   ├── flags.json
│           │   ├── log.pkl
│           │   └── train.csv
│           ├── cube-single
│           │   ├── agent.pt
│           │   ├── eval.csv
│           │   ├── flags.json
│           │   ├── log.pkl
│           │   └── train.csv
├── src
│   ├── agents
│   │   ├── ...
│   │   ├── fql_agent.py
│   │   ├── iql_agent.py
│   │   ├── sacbc_agent.py
│   │   └── ...
│   └── ...
├── pyproject.toml
├── uv.lock
└── README.md
```

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 Report**.

References

- Scott Fujimoto and Shixiang Shane Gu. A minimalist approach to offline reinforcement learning. In *Neural Information Processing Systems (NeurIPS)*, 2021.
- Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning (ICML)*, 2018.
- Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning (ICML)*, 2018a.
- Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, G. Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft actor-critic algorithms and applications. *ArXiv*, abs/1812.05905, 2018b.
- Ilya Kostrikov, Ashvin Nair, and Sergey Levine. Offline reinforcement learning with implicit q-learning. In *International Conference on Learning Representations (ICLR)*, 2022.
- Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Manfred Otto Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2016.
- Seohong Park, Kevin Frans, Benjamin Eysenbach, and Sergey Levine. Ogbench: Benchmarking offline goal-conditioned rl. In *International Conference on Learning Representations (ICLR)*, 2025a.
- Seohong Park, Qiyang Li, and Sergey Levine. Flow q-learning. In *International Conference on Machine Learning (ICML)*, 2025b.
- Zhendong Wang, Jonathan J Hunt, and Mingyuan Zhou. Diffusion policies as an expressive policy class for offline reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2023.