GCB6206 Homework 4: Soft Actor-Critic

[Your name]
[Your student ID]

1 Introduction

DQN is great for discrete action spaces. However, it requires calculating $\max_{\mathrm{a}} Q(\mathrm{s}, \mathrm{a})$. Doing this is trivial for discrete action spaces, where you can exhaustively find which of the n actions has the highest Q-value. However, in continuous action spaces, this can be a complex nonlinear optimization problem.

Actor-critic methods get around this by learning an explicit policy, π, that is trained to maximize $\mathbb{E}_{\mathrm{a}\sim\pi(\mathrm{a}|\mathrm{s})} Q(\mathrm{s}, \mathrm{a})$. Throughout this assignment, you will learn to implement one of the most popular algorithms in this line, Soft Actor-Critic (SAC).

All experiments in this assignment can be executed with the following command:

```
python gcb6206/scripts/run_hw4.py -cfg experiments/sac/<CONFIG>.yaml
```

2 Code Structure Overview

The code structure is similar to that of the previous homework. In this assignment, you mainly work with the following codes:

- `gcb6206/scripts/run_hw4.py`: the main training loop for your SAC implementation.
- `gcb6206/agents/sac_agent.py`: the SAC learner you will implement.
- `gcb6206/networks/state_action_value_critic.py`: a simple MLP-based critic network, $Q(\mathrm{s}, \mathrm{a})$. Note that unlike the DQN's critic, which maps a state to an array of Q-values for all actions, this critic maps (s, a) to a single Q-value.
- `gcb6206/env_configs/sac_config.py`: base configuration (and list of hyperparameters).
- `experiments/sac/*.yaml`: configuration files for the experiments.

3 Training Loop

Before you start implementing SAC, you first need to complete all `TODO`s in `gcb6206/scripts/run_hw4.py`. This is very similar to the DQN script in Homework 3, as both algorithms are off-policy methods!

4 Bootstrapping

As in DQN, we train our critic by "bootstrapping" from a target critic. Using the tuple $(\mathrm{s}_t, \mathrm{a}_t, r_t, \mathrm{s}_{t+1}, d_t)$ (where d_t is the flag for whether the trajectory terminates after this transition), we set a target value y:

$$y \leftarrow r_t + \gamma(1 - d_t) Q_\phi(\mathrm{s}_{t+1}, \mathrm{a}_{t+1}), \text{ where } \mathrm{a}_{t+1} \sim \pi(\mathrm{a}_{t+1} \mid \mathrm{s}_{t+1}). \tag{1}$$

Then, we use this bootstrapped target, y, to update our critic, Q_ϕ, by minimizing the L2 loss:

$$\min_\phi (Q_\phi(\mathrm{s}_t, \mathrm{a}_t) - y)^2. \tag{2}$$

In practice, we stabilize learning by using a separate target network, $Q_{\phi'}$. There are two common strategies for updating the target network, as covered in lectures:

- Hard update (implemented in DQN): set $\phi' \leftarrow \phi$ every K steps.
- Soft update: ϕ' is continually updated towards ϕ with Polyak averaging (exponential moving average). After each update, we perform the following operation (τ is typically very small, e.g., 0.005):

$$\phi' \leftarrow \phi' + \tau(\phi - \phi'). \tag{3}$$

4.1 Implementation

Let's first implement the critic update in `gcb6206/agents/sac_agent.py`:

- Implement the bootstrapped critic update explained above in the `update_critic` method. You do not need to implement `next_action_entropy` at this point.
- Update the critic for `num_critic_updates` times in the `update` method.
- Implement soft and hard target network updates in `update`.

4.2 Experiments

- Train an agent on `Pendulum-v1` with the configuration `experiments/sac/sanity_pendulum_1.yaml`.

```
python gcb6206/scripts/run_hw4.py -cfg experiments/sac/sanity_pendulum_1.yaml
```

- It will not get high reward yet as you are not training an actor. But, the Q-values should stabilize at some large negative number (like, -700). If the Q-values go to minus infinity or stay close to zero, there might be something wrong with your code.
- The "do-nothing" reward for this environment is about -10 per step. Let's assume our agent always get "do nothing", so that they agent receives -10 every step and never terminates. Calculate (approximately) what should be the average Q-value you would expect over training batches considering the discount factor $\gamma = 0.99$.

  Answer: Add your explanation.

5 Entropy Bonus and Soft Actor-Critic

In DQN, we use an ϵ-greedy strategy to decide which action to take at a given time. In continuous spaces, we have several options for generating exploration noise.

One of the most common approaches is providing an entropy bonus to encourage the actor to have high entropy (i.e. to be more random), scaled by a "temperature" coefficient β. For example, in the REPARAMETRIZE case:

$$\mathcal{L}_\pi = Q(\mathrm{s}, \mu_\theta(\mathrm{s}) + \sigma_\theta(\mathrm{s})\epsilon) + \beta\mathcal{H}(\pi_\theta(\cdot \mid \mathrm{s})), \text{ where } \epsilon \sim \mathcal{N}(0, 1), \tag{4}$$

where entropy is defined as $\mathcal{H}(\pi(\cdot \mid \mathrm{s})) = \mathbb{E}_{\mathrm{a}\sim\pi}\left[-\log \pi(\mathrm{a} \mid \mathrm{s})\right]$.

To make sure the entropy term is also factored into the Q-function, we use soft Q target values:

$$y \leftarrow r_t + \gamma(1 - d_t)\left[Q_\phi(\mathrm{s}_{t+1}, \mathrm{a}_{t+1}) + \beta\mathcal{H}(\pi(\cdot \mid \mathrm{s}_{t+1}))\right]. \tag{5}$$

This soft Q-values result in behaviors where the actor will choose more random actions when it is unsure of what action to take. Please refer to the SAC paper: https://arxiv.org/abs/1801.01290 for more details.

Note that maximizing entropy $\mathcal{H}(\pi_\theta) = -\mathbb{E}[\log \pi_\theta]$ requires differentiating through the sampling distribution. We can do this via the "reparametrization trick" from lecture – if you'd like a refresher, refer to the Section 7.

5.1 Implementation

In this section, you need to compute (estimate) entropy of a policy and implement soft Q target values in `gcb6206/agents/sac_agent.py`:

- Implement the `entropy` method to calculate the approximate the entropy of an action distribution with one sample:

$$\mathcal{H}(\pi_\theta(\cdot \mid \mathrm{s})) = \mathbb{E}_{\mathrm{a}\sim\pi_\theta}\left[-\log \pi_\theta(\mathrm{a} \mid \mathrm{s})\right] \approx -\log \pi_\theta(\hat{\mathrm{a}} \mid \mathrm{s}), \text{ where } \hat{\mathrm{a}} \sim \pi_\theta(\cdot \mid \mathrm{s}). \tag{6}$$

- Add the entropy term to the target critic values in `update_critic`.
- Call `update_actor` in `update` to train the actor to maximize the entropy.

5.2 Experiments

- Train an agent on `Pendulum-v1` with the configuration `experiments/sac/sanity_pendulum_2.yaml`, which learns soft Q-values and maximizes a policy entropy:

```
python gcb6206/scripts/run_hw4.py -cfg experiments/sac/sanity_pendulum_2.yaml
```

- The experiments log `entropy` during the actor updates. Since the agent only maximizes its entropy, it should achieve (close to) the maximum possible entropy for a 1-dimensional action space. The entropy is maximized when it is a uniform distribution:

$$\mathcal{H}(\mathcal{U}[-1, 1]) = \mathbb{E}[-\log p(x)] = -\log \frac{1}{2} = \log 2 \approx 0.69. \tag{7}$$

  Because currently our actor loss only consists of the entropy bonus (we have not implemented anything to maximize rewards yet), the entropy should increase until it reaches roughly this level. If the entropy is higher than this, or significantly lower, there can be something wrong with the code.

- There is no result to report in this section.

6 Actor Update with REINFORCE

To train our actor to maximize rewards, we can use the REINFORCE gradient estimator that we used in Homework 2!

The objective of the policy is to maximize the expected return and we need its gradient:

$$\nabla_\theta \mathbb{E}_{\mathrm{s}\sim\mathcal{D}, \mathrm{a}\sim\pi_\theta(\mathrm{a}|\mathrm{s})}\left[Q_\phi(\mathrm{s}, \mathrm{a})\right]. \tag{8}$$

As we learned in lecture and Homework 2, we can use REINFORCE to compute the policy gradient:

$$\mathbb{E}_{\mathrm{s}\sim\mathcal{D},\mathrm{a}\sim\pi_\theta(\mathrm{a}|\mathrm{s})}\left[\nabla_\theta \log(\pi_\theta(\mathrm{a}\mid\mathrm{s}))Q_\phi(\mathrm{s},\mathrm{a})\right]. \tag{9}$$

Note that the actions a are sampled from π_θ and we do not require real data samples. This means that to reduce the variance of this policy gradient, we can sample more actions from π_θ for any given state! You will implement this in your code using the `num_actor_samples` parameter.

6.1 Implementation

In this section, you need to implement the REINFORCE policy gradient estimator in `gcb6206/agents/sac_agent.py`:

- Implement the REINFORCE gradient estimator in the `actor_loss_reinforce` method.

6.2 Experiments

- Please note that each of the following experiments would take at least an hour.
- Train an agent on `InvertedPendulum-v4` using `sanity_invertedpendulum_reinforce.yaml`. You should achieve a total reward close to 1000, which corresponds to staying upright for all time steps.

```
python gcb6206/scripts/run_hw4.py \
    -cfg experiments/sac/sanity_invertedpendulum_reinforce.yaml
```

Answer:

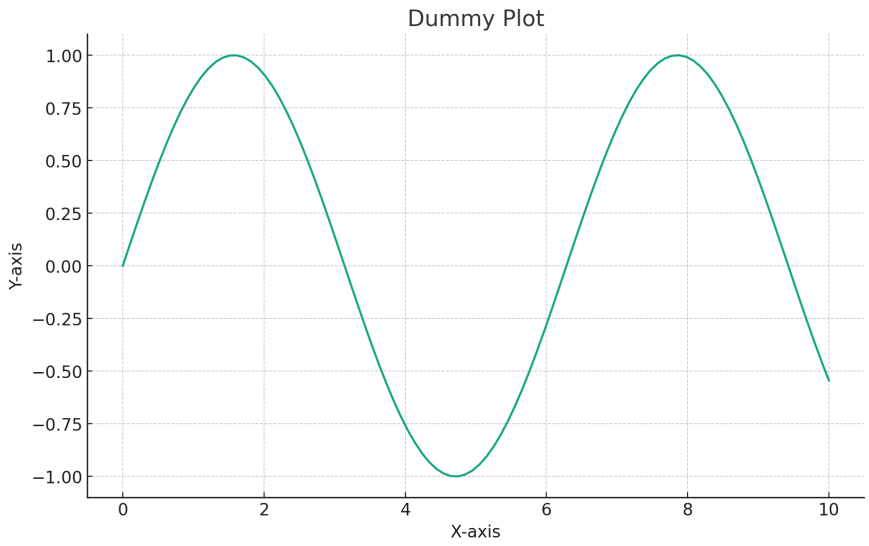

Figure 1: Replace this with your plot, and add the caption.

Add your explanation.

- Train an agent on `HalfCheetah-v4` using the provided config (`halfcheetah_reinforce1.yaml`). Note that this configuration uses only one sampled action for estimating policy gradients. The result should reach positive rewards in 500K steps.

```
python gcb6206/scripts/run_hw4.py -cfg experiments/sac/halfcheetah_reinforce1.yaml
```

- Train another agent with `halfcheetah_reinforce10.yaml`. This configuration takes 10 action samples from the actor for computing the REINFORCE gradient. We call this REINFORCE-10 and the single-sample version above REINFORCE-1. REINFORCE-10 should achieve above 500 evaluation returns in 200K steps.

- Plot the results (evaluation return over time) of REINFORCE-1 and REINFORCE-10. Compare and explain your results.

```
python gcb6206/scripts/run_hw4.py -cfg experiments/sac/halfcheetah_reinforce10.yaml
```

Answer:

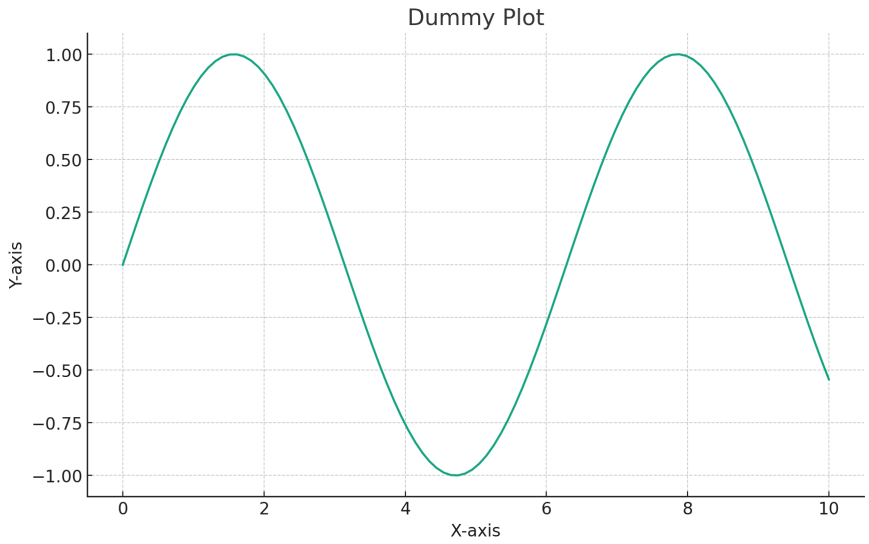

Figure 2: Replace this with your plot, and add the caption.

Add your explanation.

7 Actor Update with REPARAMETRIZE

REINFORCE works quite well with many samples, but particularly in high-dimensional action spaces, it starts to require a lot of samples to give low variance. We can improve this by using the reparametrized gradient.

If we parameterize π_θ as $\mu_\theta(\mathrm{s}) + \sigma_\theta(\mathrm{s})\epsilon$, where ϵ is normally distributed, then we can write:

$$\nabla_\theta \mathbb{E}_{\mathrm{s}\sim\mathcal{D},\mathrm{a}\sim\pi_\theta(\mathrm{a}|\mathrm{s})}\left[Q(\mathrm{s},a)\right] = \nabla_\theta \mathbb{E}_{\mathrm{s}\sim\mathcal{D},\epsilon\sim\mathcal{N}}\left[Q(\mathrm{s},\mu_\theta(\mathrm{s})+\sigma_\theta(\mathrm{s})\epsilon))\right] = \mathbb{E}_{\mathrm{s}\sim\mathcal{D},\epsilon\sim\mathcal{N}}\left[\nabla_\theta Q(\mathrm{s},\mu_\theta(\mathrm{s})+\sigma_\theta(\mathrm{s})\epsilon))\right]. \tag{10}$$

This gradient estimator often provides a much lower variance, so it can be used with few samples (in practice, just using a single sample works very well).

7.1 Implementation

- Implement `actor_loss_reparametrize()` in `gcb6206/agents/sac_agent.py`. Be careful to use the reparametrization trick for sampling (`*.rsample()`)!

7.2 Experiments

- Please note that each of the following experiments would take at least an hour.
- Make sure you can solve `InvertedPendulum-v4` (use `sanity_invertedpendulum_reparametrize.yaml`). You should achieve reward close to 1000, which corresponds to staying upright for all time steps.

```
python gcb6206/scripts/run_hw4.py \
    -cfg experiments/sac/sanity_invertedpendulum_reparametrize.yaml
```

- Train (once again) on `HalfCheetah-v4` with `halfcheetah_reparametrize.yaml`. Plot results for all three gradient estimators (REINFORCE-1, REINFORCE-10, and REPARAMETRIZE) on the same set of axes, with number of environment steps on the x-axis and evaluation return on the y-axis.

```
python gcb6206/scripts/run_hw4.py \
    -cfg experiments/sac/halfcheetah_reparametrize.yaml
```

Answer:

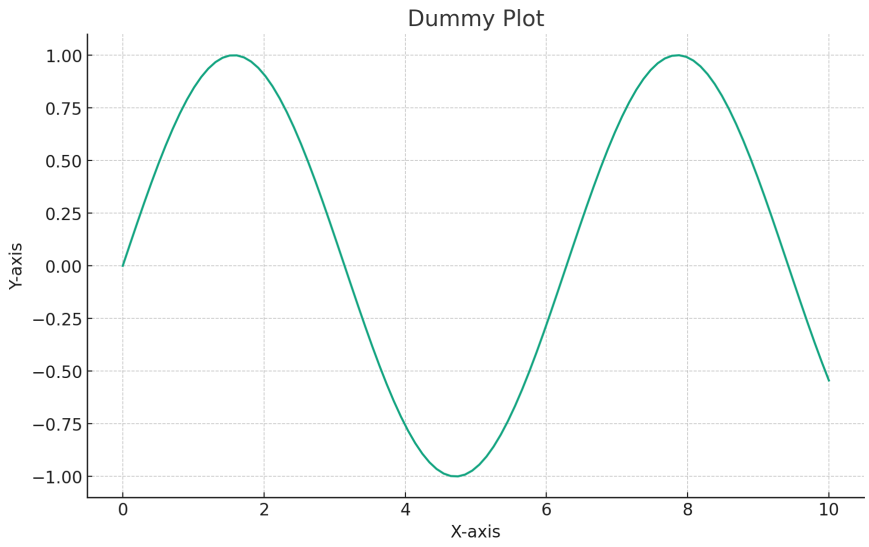

Figure 3: Replace this with your plot, and add the caption.

Add your explanation.

- (Optional) Train an agent for the `Humanoid-v4` environment with `humanoid.yaml` and plot results.

```
python gcb6206/scripts/run_hw4.py -cfg experiments/sac/humanoid.yaml
```

8 Stabilizing Target Values

As in DQN, the target Q with a single critic exhibits overestimation bias! There are a few commonly-used strategies to combat this:

- Double-Q: learn two critics Q_{ϕ_A}, Q_{ϕ_B}, and keep two target networks $Q_{\phi'_A}, Q_{\phi'_B}$. Then, use $Q_{\phi'_A}$ to compute target values for Q_{ϕ_B} and vice versa (it's slightly different from Double DQN):

$$y_A = r + \gamma Q_{\phi'_B}(\mathrm{s}', \mathrm{a}')$$

$$y_B = r + \gamma Q_{\phi'_A}(\mathrm{s}', \mathrm{a}')$$

- Clipped double-Q: learn two critics Q_{ϕ_A}, Q_{ϕ_B} (and keep two target networks). Then, compute the target values as $\min(Q_{\phi'_A}, Q_{\phi'_B})$.

$$y_A = y_B = r + \gamma \min\left(Q_{\phi'_A}(\mathrm{s}', \mathrm{a}'), Q_{\phi'_B}(\mathrm{s}', \mathrm{a}')\right)$$

8.1 Implementation

- Implement double-Q and clipped double-Q in the `q_backup_strategy` function in `sac_agent.py`.

8.2 Experiments

- Please note that each of the following experiments would take at least two hours.
- Run single-Q, double-Q, and clipped double-Q on `Hopper-v4` using the corresponding configuration files. Which one works best? Plot the logged `eval_return` from each of them as well as `q_values`. Discuss how these results relate to overestimation bias. The default seed should work well, but if not, you may report results using a seed that shows the better tendency.

```
python gcb6206/scripts/run_hw4.py -cfg experiments/sac/hopper.yaml --seed 48
python gcb6206/scripts/run_hw4.py -cfg experiments/sac/hopper_doubleq.yaml \
    --seed 48
python gcb6206/scripts/run_hw4.py -cfg experiments/sac/hopper_clipq.yaml --seed 48
```

Which one works the best? Your answer.

Answer:

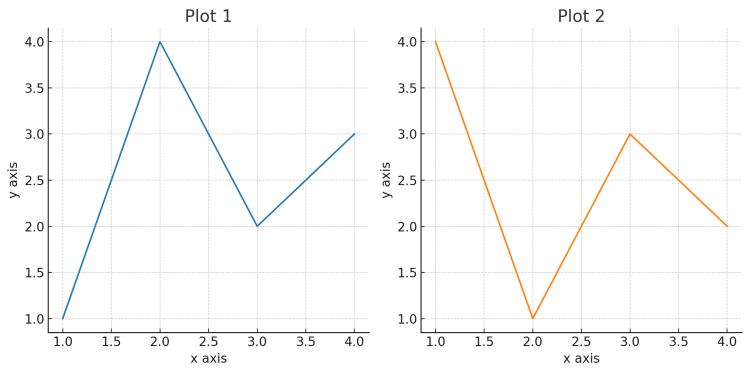

Figure 4: Replace this with your plots (eval return and Q-values), and add the caption.

Add your explanation.

- (Optional) Pick the best configuration among single-Q, double-Q, and clipped double-Q, and run it on `Humanoid-v4` using `humanoid.yaml` (edit the config to use the best option). Plot the default configuration (in the previous section) and the best configuration together with environment steps on the x-axis and evaluation return on the y-axis.

  Note: if you'd like to run training to completion (5M steps, 24h of training), you should get a proper, walking humanoid! You can run with videos enabled by using `-nvid 1`.

9 Submitting the code and experiment runs

Please submit the code, tensorboard logs, and the "report" in a single zip file, `hw4_[YourStudentID].zip`. Do not include videos as the file size should be less than 50MB. The structure of the submission file should be:

```
hw4_[YourStudentID].zip
├── hw4_[YourStudentID].pdf
├── data
│   ├── hw4_sac_...
│   │   └── events.out.tfevents.1567529456.e3a096ac8ff4
│   └── ...
├── gcb6206
│   ├── agents
│   │   └── sac_agent.py
│   └── ...
├── README.md
└── ...
```