Assignment 3: Q-Learning and Actor-Critic Algorithms

1 Deep Q-Learning

1.1 Introduction

Part 1 of this assignment requires you to implement and evaluate Q-learning for playing Atari games. The Q-learning algorithm was covered in lecture, and you will be provided with starter code. This assignment will be faster to run on a GPU, though it is possible to complete on a CPU as well. Note that we use convolutional neural network architectures in this assignment. Therefore, we recommend using the Colab option if you do not have a GPU available to you. Please start early!

1.2 File overview

The starter code for this assignment can be found at

`https://github.com/berkeleydeeprlcourse/homework_fall2023/tree/main/hw3`

You will implement a DQN agent in `cs285/agents/dqn_agent.py` and `cs285/scripts/run_hw3_dqn.py`. In addition to those two files, you should start by reading the following files thoroughly:

- `cs285/env_configs/dqn_basic.py`: builds networks and generates configuration for the basic DQN problems (cartpole, lunar lander).
- `cs285/env_configs/dqn_atari.py`: builds networks and generates configuration for the Atari DQN
- `cs285/infrastructure/replay_buffer.py`: implementation of replay buffer. You don't need to know how the memory efficient replay buffer works, but you should try to understand what each method does (particularly the difference between `insert`, which is called after a frame, and `on_reset`, which inserts the first observation from a trajectory) and how it differs from the regular replay buffer.
- `cs285/infrastructure/atari_wrappers.py`: contains some wrappers specific to the Atari environments. These wrappers can be key to getting challenging Atari environments to work!

There are two new package requirements (`gym[atari]` and `pip install gym[accept-rom-license]`) beyond what was used in the first two assignments; make sure to install these with `pip install -r requirements.txt` if you're re-using your Python environment from last assignment.

1.3 Implementation

The first phase of the assignment is to implement a working version of Q-learning, with some extra bells and whistles like double DQN. Our code will work with both state-based environments, where our input is a low-dimensional list of numbers (like Cartpole), but we'll also support learning directly from pixels!

In addition to the double Q-learning trick (which you'll implement later), we have a few other tricks implemented to stabilize performance. You don't have to do anything to enable these, but you should look at the implementations and think about why they work.

- **Exploration scheduling for ϵ-greedy actor.** This starts ϵ at a high value, close to random sampling, and decays it to a small value during training.
- **Learning rate scsheduling.** Decay the lr from a high initial value to a lower one at the end of training.
- **Gradient clipping.** If the gradient norm is larger than a threshold, scale the gradients down so that the norm is equal to the threshold.
- **Atari wrappers.**

  I **Frame-skip.** Keep the same constant action for 4 steps.
  II **Frame-stack.** Stack the last 4 frames to use as the input.
  III **Grayscale.** Use grayscale images.

1.4 Basic Q-Learning

Implement the basic DQN algorithm. You'll implement an update for the Q-network, a target network, and

What you'll need to do:

- Implement a DQN critic update in `update_critic` by filling in the unimplemented sections (marked with TODO(student)).
- Implement ϵ-greedy sampling in `get_action`
- Implement the TODOs in `run_hw3_dqn.py`.

  Hint: A trajectory can end (`done=True`) in two ways: the actual end of the trajectory (usually triggered by catastrophic failure, like crashing), or *truncation*, where the trajectory doesn't actually end but we stop simulation for some reason (commonly, we truncate trajectories at some maximum episode length). In this latter case, you should still reset the environment, but the `done` flag for TD-updates (stored in the replay buffer) should be false.
- Call all of the required updates, and update the target critic if necessary, in `update`.

Testing this section:

- Debug your DQN implementation on `CartPole-v1` with `experiments/dqn/cartpole.yaml`. It should reach reward of nearly 500 within a few thousand steps.

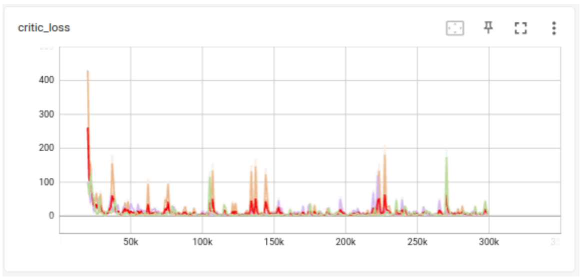
Deliverables:

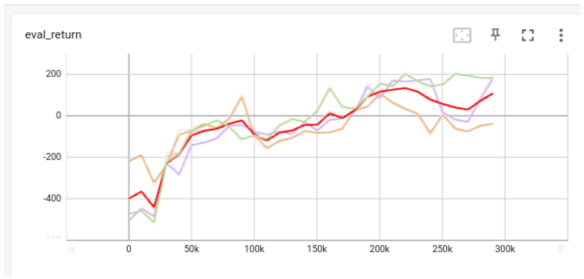
- and a plot with environment steps on the x-axis and eval return on the y-axis.

  Plot is attached with the lr comparison below.
- Run DQN with three different seeds on `LunarLander-v2`:

```
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander.yaml --seed 1
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander.yaml --seed 2
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander.yaml --seed 3
```

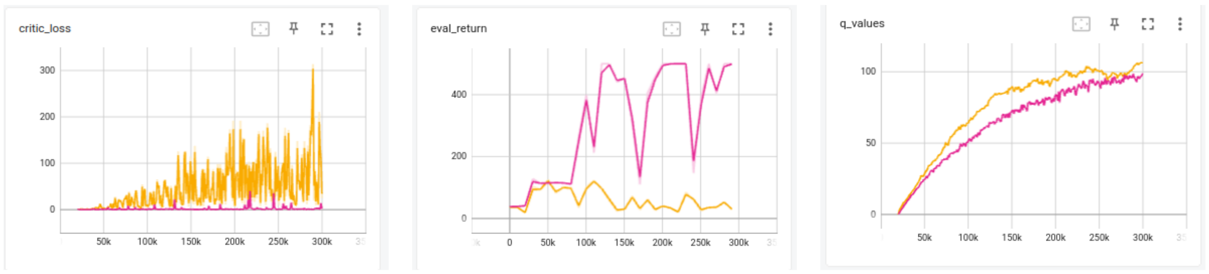
  Your code may not reach high return (200) on Lunar Lander yet; this is okay! Your returns may go up for a while and then collapse in some or all of the seeds.





LunarLander DQN results across 3 seeds. Left: Critic loss. Right: Eval return.

- Run DQN on `CartPole-v1`, but change the `learning rate` to 0.05 (you can change this in the YAML config file). What happens to (a) the predicted Q-values, and (b) the critic error? Can you relate this to any topics from class or the analysis section of this homework?



CartPole DQN with different learning rates. Pink: lr=0.001, Orange: lr=0.05.

Analysis: (a) The higher learning rate (lr=0.05, Orange) converges faster initially. The lower learning rate converges more slowly but reaches higher and more stable final returns. (b) The critic loss is significantly higher and more unstable with lr=0.05, indicating that while higher learning rates speed up learning, they introduce instability and larger prediction errors during training.

1.5 Double Q-Learning

Let's try to stabilize learning. The double-Q trick avoids overestimation bias in the critic update by using two different networks to *select* the next action a' and to *estimate* its value:

$$a' = \arg\max_{a'} Q_\phi(s', a')$$

$$Q_{\text{target}} = r + \gamma(1 - d_t) Q_{\phi'}(s', a').$$

In our case, we'll keep using the target network $Q_{\phi'}$ to estimate the action's value, but we'll select the action using Q_ϕ (the online Q network).

Implement this functionality in `dqn_agent.py`.

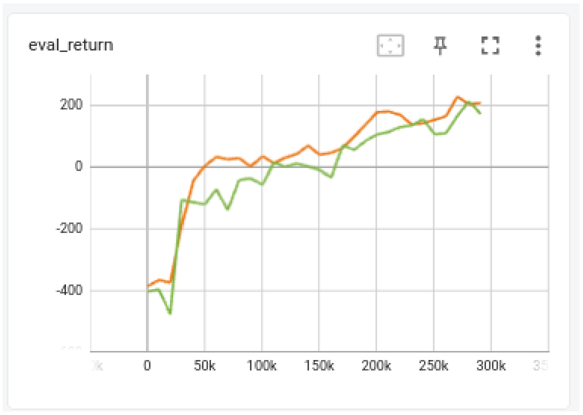
Deliverables:

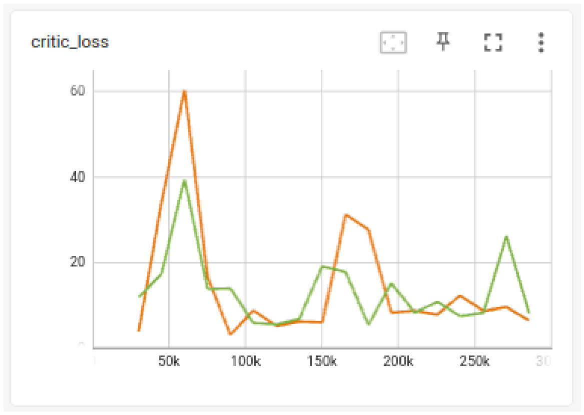
- Run three more seeds of the lunar lander problem:

```
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander_doubleq.yaml --seed 1
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander_doubleq.yaml --seed 2
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/lunarlander_doubleq.yaml --seed 3
```

  You should expect a return of **200** by the end of training, and it should be fairly stable compared to your policy gradient methods from HW2.

  Plot returns from these three seeds in ~~red~~, and the "vanilla" DQN results in ~~blue~~, on the same set of axes. Compare the two, and describe in your own words what might cause this difference.





Vanilla DQN vs Double-Q on LunarLander across 3 seeds each.

Analysis: The green curve (Double-Q) shows more stable learning compared to the orange curve (vanilla DQN). Double-Q reduces overestimation bias by using the online network to select actions and the target network to evaluate them, although in our experiments vanilla DQN achieved higher average return.

Vanilla DQN's overestimation bias can be beneficial early in training because optimistic Q-values encourage more aggressive exploration and faster initial reward accumulation. By removing this optimism, Double DQN tends to learn more conservatively, which can result in slower early performance and lower returns at the beginning.

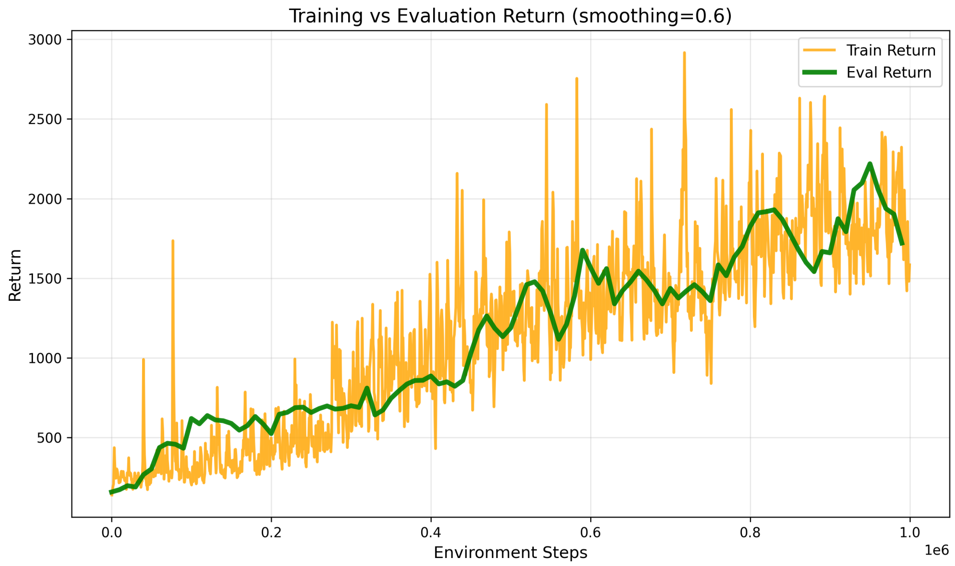
LunarLander's small discrete action space, dense rewards, and relatively low stochasticity mean that DQN's bias often does not severely harm learning and can sometimes even help, reducing the advantage of Double DQN in this environment.

Finally, using identical hyperparameters does not guarantee a fair comparison. Double DQN often prefers different settings, such as higher learning rates or less frequent target updates.

- Run your DQN implementation on the `MsPacman-v0` problem. Our default configuration will use double-Q learning by default. You are welcome to tune hyperparameters to get it to work better, but the default parameters should work (so if they don't, you likely have a bug in your implementation). Your implementation should receive a score of around **1500** by the end of training (1 million steps. **This problem will take about 3 hours with a GPU, or 6 hours without, so start early!**

```
python cs285/scripts/run_hw3_dqn.py -cfg experiments/dqn/mspacman.yaml
```

- Plot the average training return (`train_return`) and eval return (`eval_return`) on the same axes. You may notice that they look very different early in training! Explain the difference.



MsPacman training return vs evaluation return.

Analysis: The training and evaluation return differ significantly early in training.

During training, the agent uses ϵ-greedy exploration, which involves random actions that can lead to lower performance. In contrast, evaluation uses a greedy policy (no exploration), which tends to perform better initially. As training progresses and the Q-network improves, both curves converge as the learned policy becomes strong enough that exploration matters less.

1.6 Experimenting with Hyperparameters

Hyperparameter options could include:

- Learning rate
- Network architecture
- Exploration schedule (or, if you'd like, you can implement an alternative to ϵ-greedy)

2 Continuous Actions with Actor-Critic

DQN is great for discrete action spaces. However, it requires you to be able to calculate $\max_a Q(s, a)$ in closed form. Doing this is trivial for discrete action spaces (when you can just check which of the n actions has the highest Q-value), but in continuous action spaces this is potentially a **complex nonlinear optimization** problem.

Actor-critic methods get around this by learning two networks: a Q-function, like DQN, and an explicit policy π that is trained to maximize $\mathbb{E}_{a\sim\pi(a|s)}Q(s, a)$.

2.1 Implementation

First, you'll need to take a look at the following files:

- `cs285/scripts/run_hw3_sac.py` - the main training loop for your SAC implementation.
- `cs285/agents/soft_actor_critic.py` - the structure for the SAC learner you'll implement.

You may also find the following files useful:

- `cs285/networks/state_action_critic.py` - a simple MLP-based $Q(s, a)$ network. Note that unlike the DQN critic, which maps states to an array of Q-value, one per action, this critic maps one (s, a) pair to a single Q-value.
- `cs285/env_configs/sac_config.py` - base configuration (and list of hyperparameters).
- `experiments/sac/*.yaml` - configuration files for the experiments.

2.1.1 Bootstrapping

As in DQN, we train our critic by "bootstrapping" from a target critic. Using the tuple $(s_t, a_t, r_t, s_{t+1}, d_t)$ (where d_t is the flag for whether the trajectory terminates after this transition), we write:

$$y \leftarrow r_t + \gamma(1 - d_t)Q_\phi(s_{t+1}, a_{t+1}), a \sim \pi(a_{t+1}|s_{t+1})$$

$$\min_\phi (Q_\phi(s_t, a_t) - y)^2$$

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

- *Hard update* (like we implemented in DQN), where every K steps we set $\phi' \leftarrow \phi$.
- ***Soft*** *update*, where ϕ' is continually updated towards ϕ with *Polyak averaging* (exponential moving average). After each step, we perform the following operation:

$$\phi' \leftarrow \phi' + \tau(\phi - \phi')$$

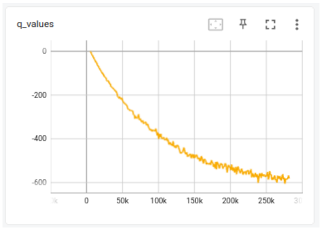
What you'll need to do (in `cs285/agents/soft_actor_critic.py`):

- Implement the bootstrapped critic update in the `update_critic` method.
- Update the critic for `num_critic_updates` in the `update` method.
- Implement soft and hard target network updates, depending on the configuration, in `update`.

Testing this section:

- Train an agent on `Pendulum-v1` with the sample configuration `experiments/sac/sanity_pendulum.yaml`. It shouldn't get high reward yet (you're not training an actor), but the Q-values should stabilize at some large negative number. The "do-nothing" reward for this environment is about -10 per step; you can use that together with the discount factor γ to calculate (approximately) what Q should be. If the Q-values go to minus infinity or stay close to zero, you probably have a bug.

Deliverables: None, once the critic is training as expected you can move on to the next section!



Stabilized Q-values.

2.1.2 Entropy Bonus and Soft Actor-Critic

In DQN, we used 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 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:

$$\mathcal{L}_\pi = Q(s, \mu_\theta(s) + \sigma_\theta(s)\epsilon) + \beta\mathcal{H}(\pi(a|s)).$$

Where entropy is defined as $\mathcal{H}(\pi(a|s)) = \mathbb{E}_{a\sim\pi}[-\log\pi(a|s)]$. To make sure entropy is also factored into the Q-function, we should also account for it in our target values:

$$y \leftarrow r_t + \gamma(1-d_t)\left[Q_\phi(s_{t+1}, a_{t+1}) + \beta\mathcal{H}(\pi(a_{t+1}|s_{t+1}))\right]$$

When balanced against the "maximize Q" terms, this results in behavior where the actor will choose more random actions when it is unsure of what action to take. Feel free to read more in the SAC paper: `https://arxiv.org/abs/1801.01290`.

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, skip to the section on REPARAMETRIZE.

What you'll need to do (in `cs285/agents/soft_actor_critic.py`):

- Implement `entropy()` to calculate the approximate entropy of an actor distribution.
- Add the entropy term to the target critic values in `update_critic()` and the actor loss in `update_actor()`.

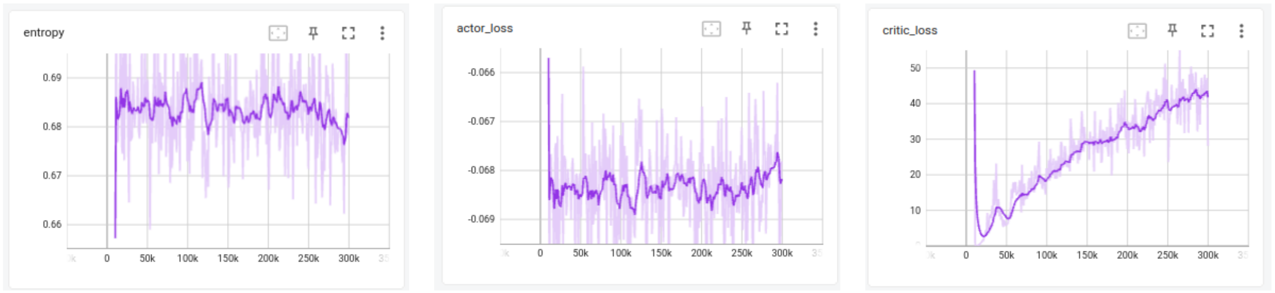
Testing this section:

- The code should be logging `entropy` during the critic updates. If you run `sanity_pendulum.yaml` from before, it should achieve (close to) the maximum possible entropy for a 1-dimensional action space. Entropy is maximized by a uniform distribution:

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

  Because currently our actor loss **only** consists of the entropy bonus (we haven't implemented anything to maximize rewards yet), the entropy should increase until it arrives at roughly this level.

  If your logged entropy is higher than this, or significantly lower, you have a bug.



The entropy $\mathcal{H}(\pi) = \mathbb{E}_{a\sim\pi}[-\log\pi(a|s)] = \log 2$ around 0.69

2.1.3 Actor with REINFORCE

We can use the same REINFORCE gradient estimator that we used in our policy gradients algorithms to update our actor in actor-critic algorithms! We want to compute:

$$\nabla_\theta \mathbb{E}_{s\sim\mathcal{D},a\sim\pi_\theta(a|s)}\left[Q(s,a)\right]$$

To do this using REINFORCE, we can use the policy gradient:

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

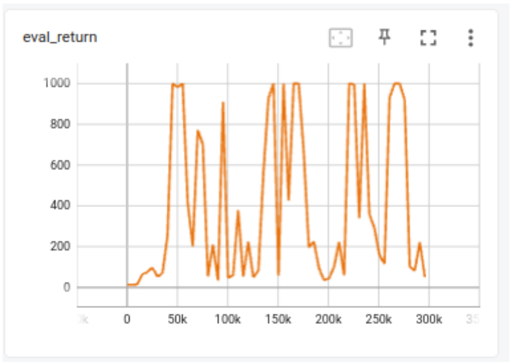
Note that the actions a are sampled from π_θ, and we do **not** require real data samples. This means that to reduce variance we can just sample more actions from π for any given state! You'll implement this in your code using the `num_actor_samples` parameter.

What you'll need to do (in `cs285/agents/soft_actor_critic.py`):

- Implement the REINFORCE gradient estimator in the `actor_loss_reinforce` method.
- Update the actor in `update`.

Testing this section:

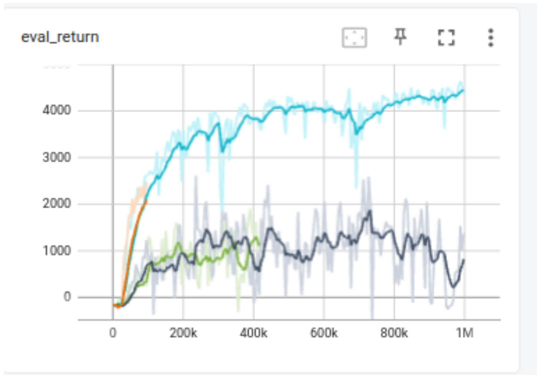
- Train an agent on `InvertedPendulum-v4` using `sanity_invertedpendulum_reinforce.yaml`. You should achieve reward close to 1000, which corresponds to staying upright for all time steps.



InvertedPendulum-v4 with REINFORCE achieves reward close to 1000.

Deliverables

- Train an agent on `HalfCheetah-v4` using the provided config (`halfcheetah_reinforce1.yaml`). Note that this configuration uses only one sampled action per training example.
- Train another agent with `halfcheetah_reinforce_10.yaml`. This configuration takes many samples from the actor for computing the REINFORCE gradient (we'll call this REINFORCE-10, and the single-sample version REINFORCE-1). Plot the results (evaluation return over time) on the same axes as the single-sample REINFORCE. Compare and explain your results.



HalfCheetah-v4 training comparison: REINFORCE-1 vs REINFORCE-10.

2.1.4 Actor 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. Parametrize π_θ as $\mu_\theta(s) + \sigma_\theta(s)\epsilon$, where ϵ is normally distributed. Then we can write:

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

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

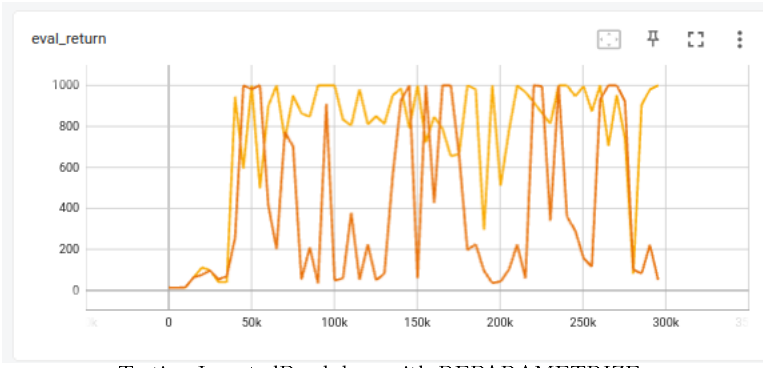
Hint: you can use `.rsample()` to get a *reparametrized* sample from a distribution in PyTorch.

What you'll need to do:

- Implement `actor_loss_reparametrize()`. Be careful to use the reparametrization trick for sampling!

Testing this section:

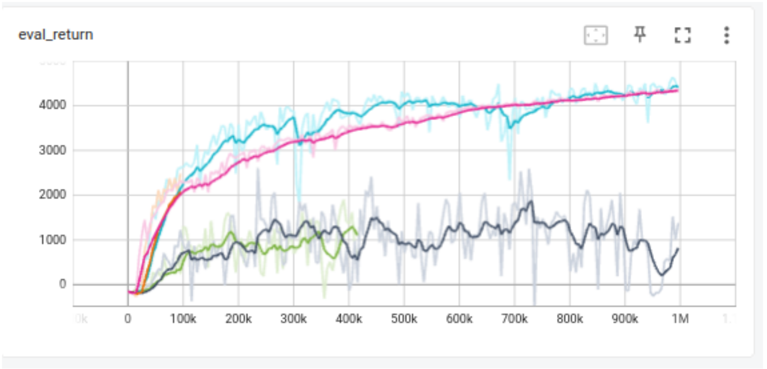
- Make sure you can solve `InvertedPendulum-v4` (use `sanity_invertedpendulum_reparametrize.yaml`) and achieve similar reward to the REINFORCE case.



Testing InvertedPendulum with REPARAMETRIZE.

Deliverables:

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

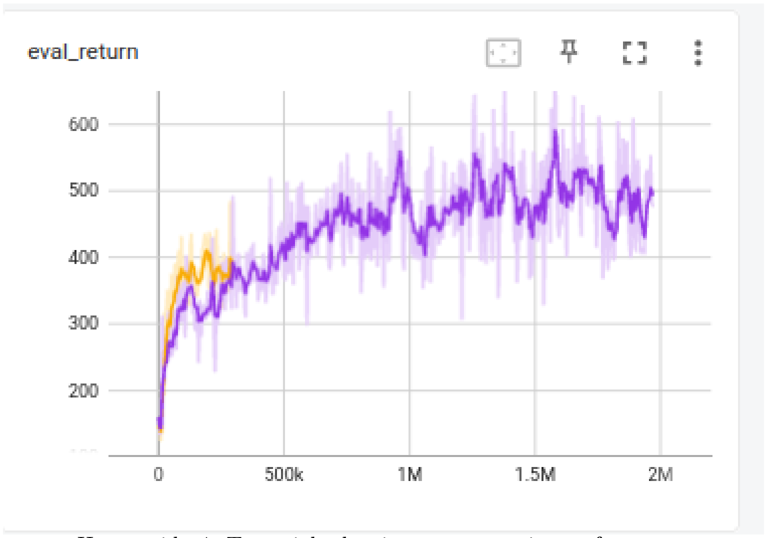


HalfCheetah-v4: Comparison of REINFORCE-1, REINFORCE-10, and REPARAMETRIZE gradient estimators.

The results demonstrate clear differences in sample efficiency and stability across gradient estimators. REPARAMETRIZE (pink) achieves the most stable convergence and highest final performance (~4200), with low variance throughout training.

REINFORCE-10 (cyan) reaches similar asymptotic performance but requires more environment steps to converge. REINFORCE-1 exhibits significantly higher variance and lower sample efficiency, with (navy) showing extreme instability.

This confirms that the reparametrization trick provides substantially lower variance gradients, enabling reliable learning with single-sample estimates, while REINFORCE requires multiple action samples to achieve comparable performance.



Humanoid-v4: Two trials showing reparametrize performance.

2.1.5 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:

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

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

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

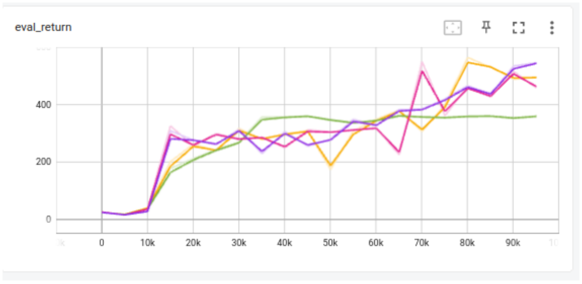
$$y_A = y_B = r + \gamma \min\left(Q_{\phi'_A}(s', a'), Q_{\phi'_B}(s', a')\right)$$

- **(Optional, bonus)** *Ensembled* clipped double-Q: learn many critics (10 is common) and keep a target network for each. To compute target values, first run all the critics and sample two Q-values for each sample. Then, take the minimum (as in clipped double-Q). If you want to learn more about this, you can check out “Randomized Ensembled Double-Q”: `https://arxiv.org/abs/2101.05982`.

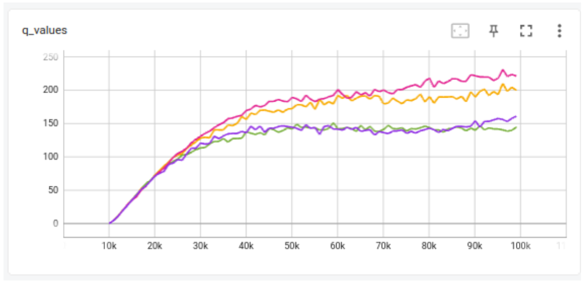
Implement double-Q and clipped double-Q in the `q_backup_strategy` function in `soft_actor_critic.py`.

Deliverables:

- 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.



Hopper-v4: Evaluation return



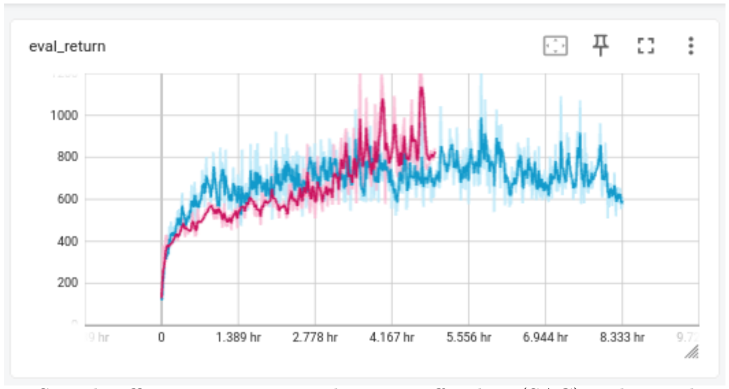
Hopper-v4: Q-values

The results clearly demonstrate the relationship between overestimation bias mitigation and policy performance. Clipped double-Q (min, blue) achieves the most stable and highest performance (∼500), while single-Q (pink) shows poor performance with high instability. Double-Q (orange/yellow) falls between the two extremes with moderate performance.

Examining the Q-values reveals the classic overestimation bias pattern: single-Q exhibits the *highest* Q-values (∼200-220) despite having the *worst* actual performance, indicating severe overestimation. In contrast, clipped double-Q shows the *lowest* Q-values. This confirms that **overestimated Q-values lead to poor policy learning when the agent becomes overconfident about suboptimal actions**. By taking the minimum of two critics, clipped double-Q produces more conservative (lower) but more *accurate* value estimates, enabling the policy to learn more effectively.

Interestingly, REDQ (green) achieves even lower Q-values than clipped double-Q but exhibits the poorest performance. This suggests that REDQ’s strategy of randomly sampling only 2 out of 10 critics may introduce too much underestimation or training instability.

- Pick the best configuration (single-Q/double-Q/clipped double-Q, or REDQ if you implement it) and run it on `Humanoid-v4` using `humanoid.yaml` (edit the config to use the best option). You can truncate it after 500K environment steps. If you got results from the humanoid environment in the last homework, plot them together with environment steps on the x-axis and evaluation return on the y-axis. Otherwise, we will provide a humanoid log file that you can use for comparison. How do the off-policy and on-policy algorithms compare in terms of sample efficiency? *Note: if you'd like to run training to completion (5M steps), you should get a proper, walking humanoid! You can run with videos enabled by using `-nvid 1`. If you run with videos, you can strip videos from the logs for submission with this script.*



Humanoid-v4: Sample efficiency comparison between off-policy (SAC) and on-policy algorithms.

Off-policy SAC demonstrates significantly better sample efficiency compared to on-policy methods, achieving meaningful learning progress within 500K environment steps. The off-policy approach benefits from replay buffer utilization, allowing each experience to be reused multiple times for training, whereas on-policy methods must discard data after each update. This advantage is particularly pronounced in complex environments like Humanoid-v4, where sample collection is expensive and off-policy learning can extract more value from limited interactions.

SAC-related questions. We wanted to address some of the common questions that have been asked regarding Question 6 of the HW. The full algorithm for SAC is summarized below, the equations listed in this paper will be helpful for you: `https://arxiv.org/pdf/1812.05905`. Some definitions that will be useful:

1. What is alpha and how to update it: Alpha is the entropy regularization coefficient denoting how much exploration to add to the policy. You should update based on Eq. 18 in Section 5 in the above paper as follows:
$$J(\alpha) = \mathop{\mathbb{E}}_{a_t \sim \pi_t}[-\alpha \log_{\pi_t}(a_t|s_t) - \alpha\bar{\mathcal{H}}].$$

2. Target entropy is the negative of the action space dimension that is used to update the alpha term.

3. SquashedNorm: This is a function that takes in mean and std as in previous homeworks, and will give you a distribution that you can sample your action from.

4. To update the critic, refer to how to update Q-function parameters in Equation 6 of the paper above as follows:
$$J_Q(\theta) = Q_\theta(s_t, a_t) - (r(s_t, a_t) + \gamma(Q_{\bar{\theta}}(s_{t+1}, a_{t+1}) - \alpha \log(\pi_\phi(a_{t+1}, s_{t+1}))))$$

5. To update the policy, follow Equation 18:
$$J(\alpha) = E_{a_t \sim \pi_t}[-\alpha \log \pi_t(a_t|s_t) - \alpha\bar{\mathcal{H}}]$$

6. You don't need to alter any parameters from the SAC run commands. The correct implementation should work with the provided default parameters.