Assignment 3: Q-Learning and Actor-Critic Algorithms

Due October 18, 11:59 pm

1 Multistep Q-Learning

Consider the N-step variant of Q-learning described in lecture. We learn $Q_{\phi_{k+1}}$ with the following updates:

$$y_{j,t} \leftarrow \left(\sum_{t'=t}^{t+N-1} \gamma^{t'-t} r_{j,t'} \right) + \gamma^N \max_{\mathbf{a}_{j,t+N}} Q_{\phi_k} \left(\mathbf{s}_{j,t+N}, \mathbf{a}_{j,t+N} \right) \tag{1}$$

$$\phi_{k+1} \leftarrow \arg\min_{\phi \in \Phi} \sum_{j,t} \left(y_{j,t} - Q_\phi(\mathbf{s}_{j,t}, \mathbf{a}_{j,t}) \right)^2 \tag{2}$$

In these equations, j indicates an index in the replay buffer of trajectories $\mathcal{D}_k$. We first roll out a batch of B trajectories to update $\mathcal{D}_k$ and compute the target values in (1). We then fit $Q_{\phi_{k+1}}$ to these target values with (2). After estimating $Q_{\phi_{k+1}}$, we can then update the policy through an argmax:

$$\pi_{k+1}(\mathbf{a}_t \mid \mathbf{s}_t) \leftarrow \begin{cases} 1 \text{ if } \mathbf{a}_t = \arg\max_{\mathbf{a}_t} Q_{\phi_{k+1}}(\mathbf{s}_t, \mathbf{a}_t) \\ 0 \text{ otherwise.} \end{cases} \tag{3}$$

We repeat the steps in eqs. (1) to (3) K times to improve the policy. In this question, you will analyze some properties of this algorithm, which is summarized in Algorithm 1.

Algorithm 1 Multistep Q-Learning

Require: iterations K, batch size B

1: initialize random policy π_0, sample $\phi_0 \sim \Phi$
2: **for** $k = 0 \dots K-1$ **do**
3: Update $\mathcal{D}_{k+1}$ with B new rollouts from π_k
4: compute targets with (1)
5: $Q_{\phi_{k+1}} \leftarrow$ update with (2)
6: $\pi_{k+1} \leftarrow$ update with (3)
7: **end for**
8: **return** π_K

1.1 TD-Learning Bias (2 points)

We say an estimator $f_{\mathcal{D}}$ of f constructed using data $\mathcal{D}$ sampled from process P is *unbiased* when $\mathbb{E}_{\mathcal{D} \sim P}[f_{\mathcal{D}}(x) - f(x)] = 0$ at each x.

Assume $\hat{Q}$ is a noisy (but unbiased) estimate for Q. Is the Bellman backup $\mathcal{B}\hat{Q} = r(s,a) + \gamma \max_{a'} \hat{Q}(s', a')$ an unbiased estimate of $\mathcal{B}Q$?

☐ Yes

☐ No

1.2 Tabular Learning (6 points total)

At each iteration of the algorithm above after the update from eq. (2), Q_{ϕ_k} can be viewed as an estimate of the true optimal Q^*. Consider the following statements:

I. $Q_{\phi_{k+1}}$ is an unbiased estimate of the Q function of the last policy, Q^{π_k}.

II. As $k \to \infty$ for some fixed B, Q_{ϕ_k} is an unbiased estimate of Q^*, i.e., $\lim_{k \to \infty} \mathbb{E}\left[Q_{\phi_k}(s,a) - Q^*(s,a)\right] = 0$.

III. In the limit of infinite iterations and data we recover the optimal Q^*, i.e., $\lim_{k, B \to \infty} \mathbb{E}\left[\|Q_{\phi_k} - Q^*\|_\infty\right] = 0$.

We make the additional assumptions:

- The state and action spaces are finite.
- Every batch contains at least one experience for each action taken in each state.
- In the tabular setting, Q_{ϕ_k} can express any function, i.e., $\{Q_{\phi_k} : \phi \in \Phi\} = \mathbb{R}^{S \times A}$.

When updating the buffer $\mathcal{D}_k$ with B new trajectories in line 3 of Algorithm 1, we say:

- When learning *on-policy*, $\mathcal{D}_k$ is set to contain only the set of B new rollouts of π (so $|\mathcal{D}_k| = B$). Thus, we only train on rollouts from the current policy.
- When learning *off-policy*, we use a fixed dataset $\mathcal{D}_k = \mathcal{D}$ of B trajectories from another policy π'.

Indicate which of the statements **I-III** always hold in the following cases. No justification is required.

	I.	**II.**	**III.**
1. $N = 1$ and ...			
(a) on-policy in tabular setting	☐	☐	☐
(b) off-policy in tabular setting	☐	☐	☐
2. $N > 1$ and ...			
(a) on-policy in tabular setting	☐	☐	☐
(b) off-policy in tabular setting	☐	☐	☐
3. In the limit as $N \to \infty$ (no bootstrapping) ...			
(a) on-policy in tabular setting	☐	☐	☐
(b) off-policy in tabular setting	☐	☐	☐

1.3 Variance of Q Estimate (2 points)

Which of the three cases ($N = 1$, $N > 1$, $N \to \infty$) would you expect to have the highest-variance estimate of Q for fixed dataset size B in the limit of infinite iterations k? Lowest-variance?

Highest variance:

- ☐ $N = 1$
- ☐ $N > 1$
- ☐ $N \to \infty$

Lowest variance:

- ☐ $N = 1$
- ☐ $N > 1$
- ☐ $N \to \infty$

1.4 Function Approximation (2 points)

Now say we want to represent Q via function approximation rather than with a tabular representation. Assume that for any deterministic policy π (including the optimal policy π^*), function approximation can represent the true Q^π exactly. Which of the following statements are true?

- ☐ When $N = 1$, $Q_{\phi_{k+1}}$ is an unbiased estimate of the Q-function of the last policy Q^{π_k}.
- ☐ When $N = 1$ and in the limit as $B \to \infty$, $k \to \infty$, Q_{ϕ_k} converges to Q^*.
- ☐ When $N > 1$ (but finite) and in the limit as $B \to \infty$, $k \to \infty$, Q_{ϕ_k} converges to Q^*.
- ☐ When $N \to \infty$ and in the limit as $B \to \infty$, $k \to \infty$, Q_{ϕ_k} converges to Q^*.

1.5 Multistep Importance Sampling (5 points)

We can use importance sampling to make the N-step update work off-policy with trajectories drawn from an arbitrary policy. Rewrite (2) to correctly approximate a Q_{ϕ_k} that improves upon π when it is trained on data $\mathcal{D}$ consisting of B rollouts of some other policy $\pi'(\mathbf{a}_t \mid \mathbf{s}_t)$.

Do we need to change (2) when $N = 1$? What about as $N \to \infty$?

You may assume that π' always assigns positive mass to each action. [Hint: re-weight each term in the sum using a ratio of likelihoods from the policies π and π'.]

2 Deep Q-Learning

2.1 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!

2.2 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 problems.
- `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.

2.3 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 learning rate from a high initial value to a lower value 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.**
  - **Frame-skip.** Keep the same constant action for 4 steps.
  - **Frame-stack.** Stack the last 4 frames to use as the input.
  - **Grayscale.** Use grayscale images.

2.4 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:

- Submit your logs of `CartPole-v1`, and a plot with environment steps on the x-axis and eval return on the y-axis.

  My Solution:

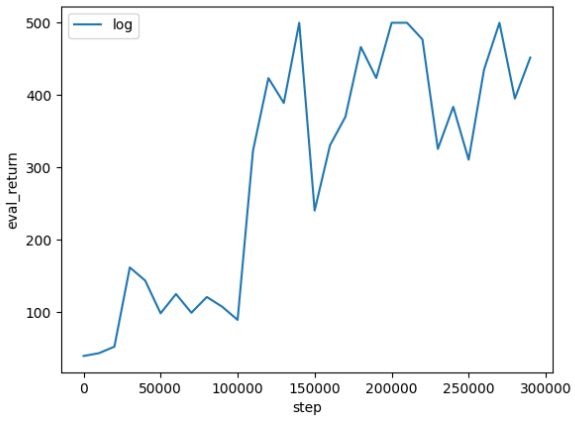


Figure 1: CartPole-v1

- 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.

My Solution: This figure will be shown in question 2.5.

- 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?

  My Solution:

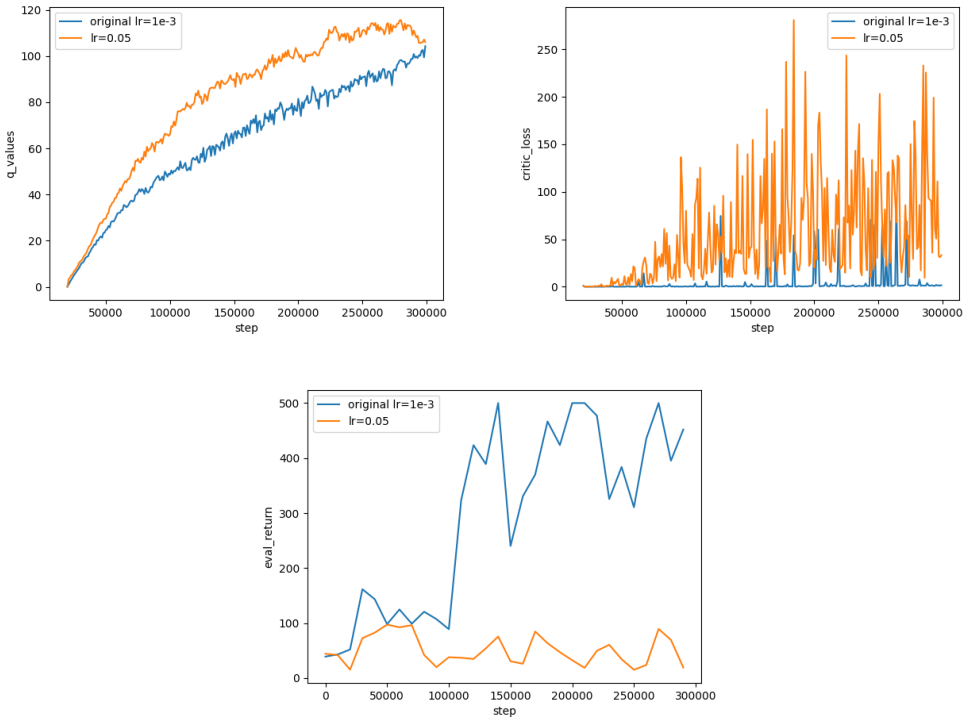


Figure 2: CartPole-v1 with learning rate 0.05 and 1e-3

The predicted Q values are larger when the learning rate is bigger (0.05). This may due to the fact that the updating speed of the target is faster, so there will be more bias. The critic loss is larger and more unstable when the learning rate is bigger. This is because the target is moving too quickly, and the model is not able to converge.

2.5 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.

  My Solution:

Figure 3: LunarLander-v2 with double Q learning

- 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.

My Solution: This experiment is no longer viable to run due to

```
gym.error.Error: We're Unable to find the game "MsPacman". Note: Gym no longer distributes
ROMs. If you own a license to use the necessary ROMs for research purposes you can download
them via `pip install gym[accept-rom-license]`. Otherwise, you should try importing "MsPacman"
via the command `ale-import-roms`. If you believe this is a mistake perhaps your copy of
"MsPacman" is unsupported. To check if this is the case try providing the environment variable
`PYTHONWARNINGS=default::ImportWarning:ale_py.roms`.

For more information see: https://github.com/mgbellemare/Arcade-Learning-Environment#
rom-management
```

2.6 Experimenting with Hyperparameters

Now let's analyze the sensitivity of Q-learning to hyperparameters. Choose one hyperparameter of your choice and run at least three other settings of this hyperparameter, in addition to the one used in Question 1, and plot all four values on the same graph. Your choice what you experiment with, but you should explain why you chose this hyperparameter in the caption. Create four config files in `experiments/dqn/hyperparameters`, and look in `cs285/env_configs/basic_dqn_config.py` to see which hyperparameters you're able to change. You can use any of the base YAML files as a reference.

Hyperparameter options could include:

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

My Solution: I choose the learning rate for the hyperparameter. The results is shown below:

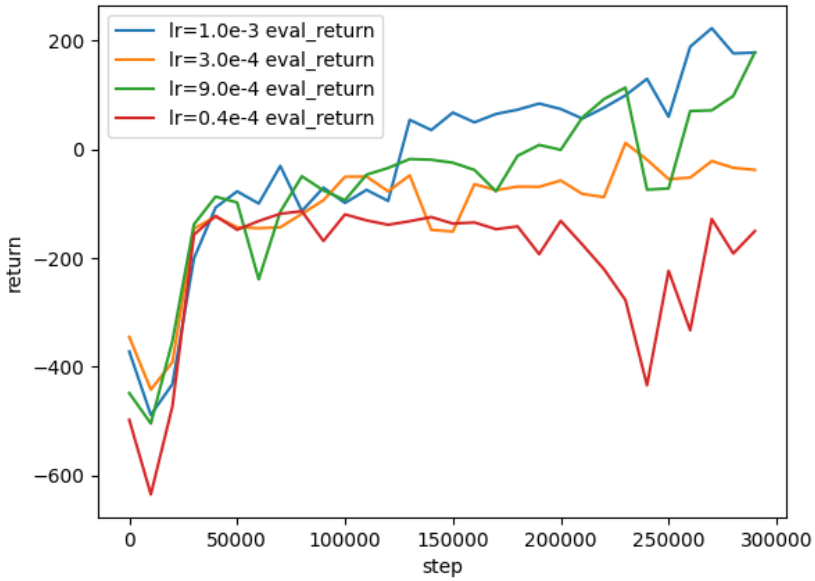


Figure 4: CartPole-v1 with different learning rates

From the result we can see that a learning rate too low will cause the return to saturate early, hence the final return is low. I can also tell that the selected learning rate are not very large, since there is no unstableness caused by the learning rate.

3 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 $\mathbb{E}_{a\sim\pi(a|s)}Q(s, a)$.

All parts in this section are run with the following command:

```
python cs285/scripts/run_hw3_sac.py -cfg experiments/sac/<CONFIG>.yaml
```

3.1 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.

You'll primarily be implementing your code in `cs285/agents/soft_actor_critic.py`.

Before implementing SAC:

- Fill in all of the `TODO`s in `cs285/scripts/run_hw3_sac.py`. This should look pretty similar to your DQN run script, as both are off-policy methods!

3.1.1 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!

3.1.2 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 $\mathcal{H}(\pi(a|s)) = \mathbb{E}_{a\sim\pi}\left[-\log\pi(a|s)\right]$. 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.

3.1.3 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.

My Solution: The result is shown below.

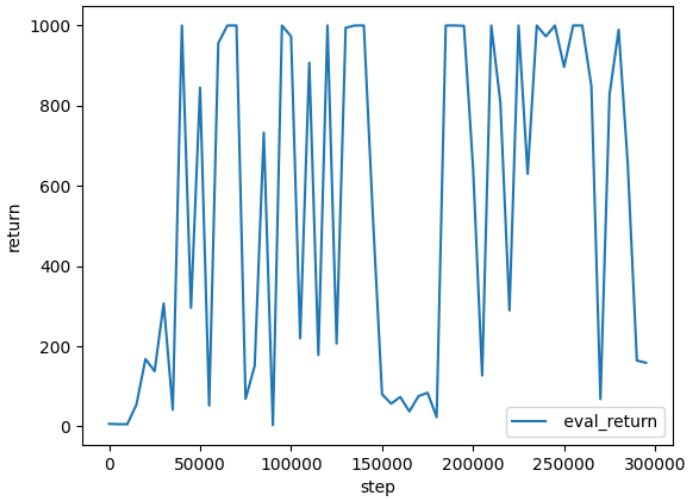


Figure 5: HalfCheetah-v4 with REINFORCE-1

The problem is that the curve is very unstable! I can't tell the reason clearly. However, I did another experiment, which turns out to make the curve stable. The result is shown below.

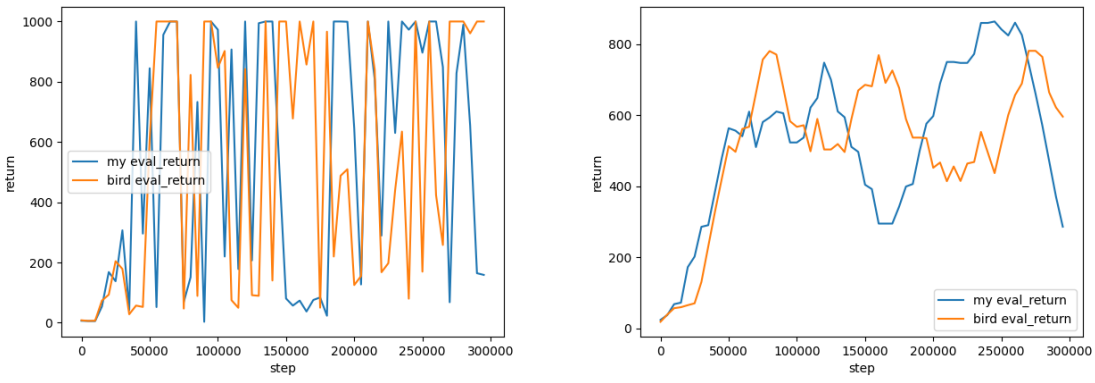


Figure 6: HalfCheetah-v4 with REINFORCE-1 (bird method); the right side is smoothed per 5 iterations

The detail of the experiment is shown below: I changed the code as

```
def actor_loss_reinforce(self, obs: torch.Tensor):
    batch_size = obs.shape[0]

    action_distribution: torch.distributions.Distribution = self.actor(obs)

    with torch.no_grad():
        action = action_distribution.sample([self.num_actor_samples])
        assert action.shape == (
            self.num_actor_samples,
            batch_size,
            self.action_dim,
        ), action.shape
        if ptu.addition_args['bird method']:
            q_values = self.critic(torch.tile(obs,dims=(self.num_actor_samples,1,1)),action)
        else:
            q_values = self.target_critic(torch.tile(obs,dims= \

                   (self.num_actor_samples,1,1)),action)

        assert q_values.shape == (
            self.num_critic_networks,
            self.num_actor_samples,
            batch_size,
        ), q_values.shape

        q_values = torch.mean(q_values, axis=0)
        advantage = q_values

    log_probs = action_distribution.log_prob(action)
    loss = -torch.mean(log_probs*advantage)

    return loss, torch.mean(self.entropy(action_distribution))
```

In this function (also similarly in the function `actor_loss_reparametrize`), If we use `bird method`, then we use the **target critic** as the critic. Otherwise, we use the **critic** as the critic.

In other words, my algorithm can be shown as

Algorithm 2 SAC Algorithm Implementation

for each step **do**
  if step $<$ some thereshold **then**
    Use random policy to collect data from the environment
  else
    Use the current policy to collect data from the environment
  end if
  Insert the data into the replay buffer
  if step $<$ another thereshold **then**
    continue
  end if
  Sample a batch of data from the replay buffer
  Update the critic for **num_critic_updates** times, using the actions from the actor
  Update the actor based on policy gradient or reparametrization trick with the objective being maximizing Q on Target Critic Network
  Perodically update (or momentum update) the target networks
end for

The `bird method` is almost the same, with only the red Target Critic Network changed to Critic Network. As XIBO (`github.com/szjzc2018`), an expert in RL, said,

> "Using the target network should be more stable."

However, this contradicts the experiment results! So no one really know which method is really better.

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.

  My Solution: This figure (and the explanation) will be shown in section 3.1.4.

3.1.4 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.

My Solution: The plot is shown below.

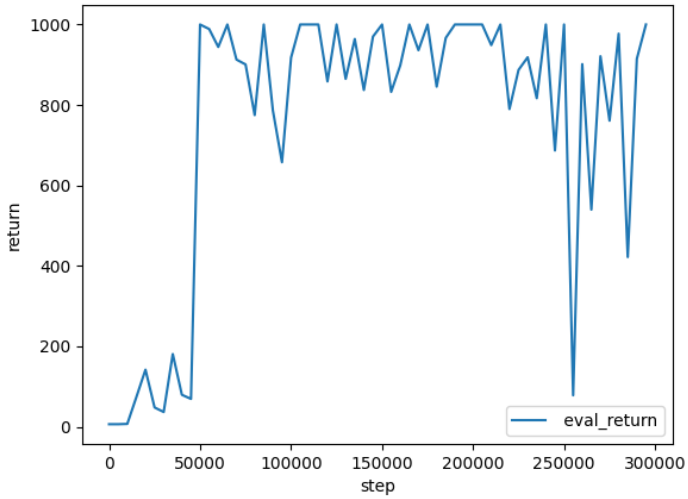


Figure 7: InvertedPendulum-v4 with REPARAMETRIZE

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.

  My Solution: The plot is shown below.

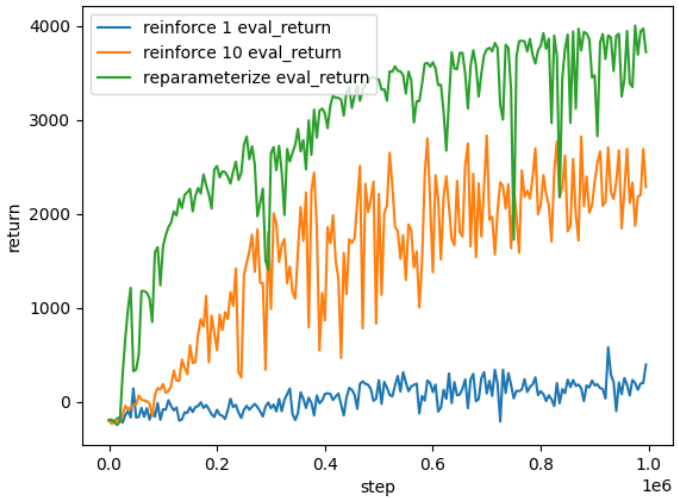


Figure 8: HalfCheetah-v4 with REPARAMETRIZE and two REINFORCE methods

For explanation (of the 3.1.3 section), the REINFORCE-10 method is better than REINFORCE-1 method, since there are more samples for REINFORCE-10 method.

- Train an agent for the `Humanoid-v4` environment with `humanoid_sac.yaml` and plot results.

  My Solution:

  This graph will be put in 3.1.5 section.

3.1.5 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.

  My Solution: Among the four implementations, double-Q and REDQ works the best. The plot is shown below.

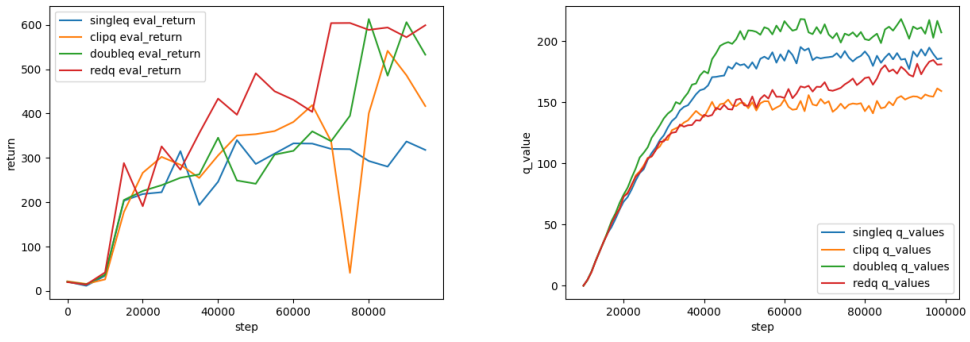


Figure 9: Hopper-v4 with single-Q, double-Q, clipped double-Q and REDQ

From the plot of the Q values, at first glance, it seems that the double Q has largest Q values. This seems to contradict the fact that double Q reduces the Q function. However, this may due to the fact that double Q method performs best, so the Q values increase very quickly. Thus, from the Q values, we can't really tell much. On the other hand, from the performance figure, we can see that double Q indeed performs the best.

- 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.*

  My Solution:

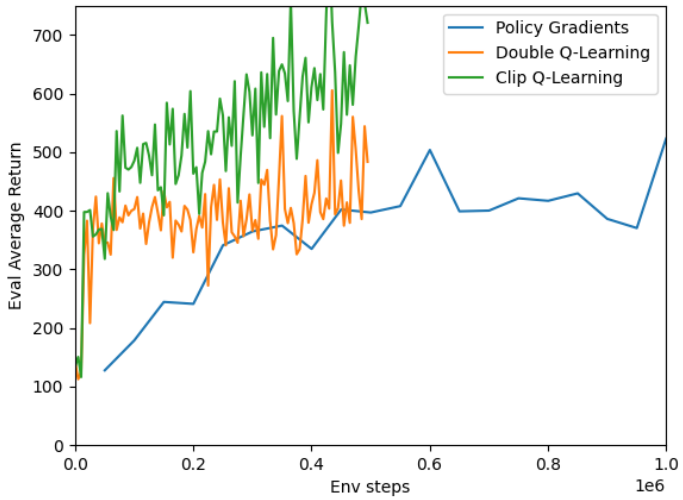


Figure 10: Humanoid-v4 with double-Q

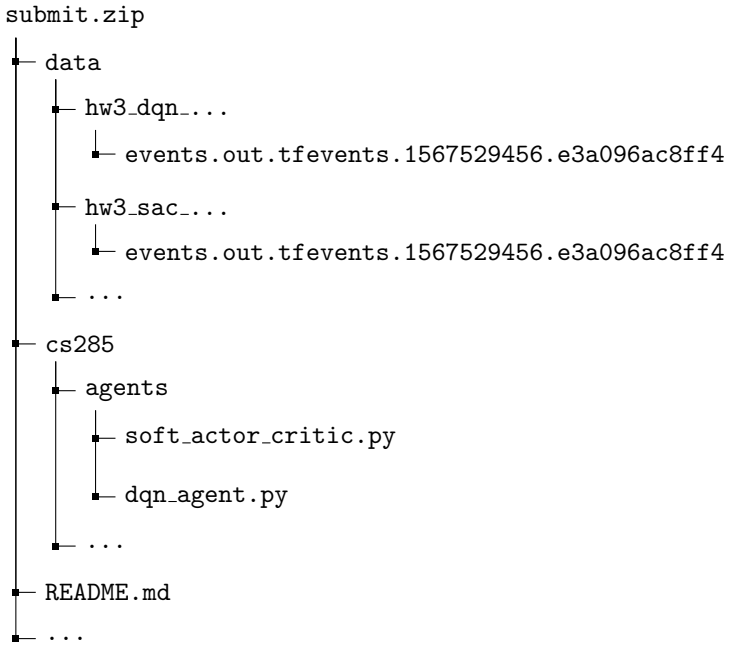
It is clear that off-policy methods are more sample efficient than on-policy methods. However, since the computation power is limited, I can't run the off-policy method to the end, so I can't tell which method is better in the end.

4 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 disabled by default in the code, but if you turned it on for debugging, you will need to run those again with `--video_log_freq -1`, or else the file size will be too large for submission.**
- 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.

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.**



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 . -x "*.DS_Store"` from your terminal from within the top-level `cs285` directory.

Turn in your assignment on Gradescope. Upload the zip file with your code and log files to **HW3 Code**, and upload the PDF of your report to **HW3**.

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.