

Assignment 3: Q-Learning and Actor-Critic Algorithms

Due March 11, 11:59 pm

1 Introduction

The goal of this assignment is to experiment with value-based deep reinforcement learning algorithms. You will implement Deep Q-Networks (DQN) for discrete action spaces and Soft Actor-Critic (SAC) for continuous action spaces.

The starter code for this assignment can be found at

https://github.com/berkeleydeeprlcourse/homework_spring2026/tree/main/hw3

This assignment will be faster to run on a GPU, though it is possible to complete on a CPU as well. Most experiments (CartPole, LunarLander, InvertedPendulum, Hopper) are relatively cheap and can be run locally. However, MsPacman (Atari) and HalfCheetah are expensive and may take approximately 3 hours even on a GPU – we recommend using Modal for these experiments. Please start early!

Results may vary with different random seeds, and for some experiments you may need to do a couple of runs to get a good result. (Note: In actual RL research, it is generally very important to report results averaged over multiple random seeds to ensure statistical significance and avoid cherry-picking! However, for this assignment, we only require you to submit your best run to reduce computational burden.)

1.1 Running on Modal

We provide Modal scripts for running experiments on cloud GPUs. First, authenticate with Modal:

```
uv run modal token new
```

Then you can launch experiments remotely. For DQN:

```
uv run modal run src/scripts/modal_run_dqn.py -- -cfg experiments/dqn/mspacman.yaml
```

For SAC:

```
uv run modal run src/scripts/modal_run_sac.py -- -cfg experiments/sac/sanity_invertedpendulum.yaml
```

Note the `--` separator between Modal arguments and your script arguments. Extra flags (e.g. `--seed 2`) can be appended after the config file.

For long-running experiments (e.g., Atari, HalfCheetah), you can use `--detach` to run in the background so you don't need to keep your terminal open:

```
uv run modal run --detach src/scripts/modal_run_dqn.py -- -cfg experiments/dqn/mspacman.yaml
```

Experiment logs are saved to a persistent Modal volume. To download them:

```
uv run modal volume get hw3-ql-volume exp/
```

2 Deep Q-Learning

2.1 Introduction

This section 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.

2.2 File Overview

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

- `src/configs/dqn.config.py`: builds networks and generates configuration for DQN problems.
- `src/infrastructure/replay_buffer.py`: implementation of replay buffer.

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 scheduling.** 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.

2.4 Basic Q-Learning

Implement the basic DQN algorithm. You'll implement an update for the Q -network, a target network, and ϵ -greedy action selection. Recall that in DQN, the action space is discrete, so the greedy action is simply $a = \arg \max_a Q_\varphi(s, a)$ over the finite set of actions.

What you'll need to do:

- Implement a DQN critic update in `update_critic` by filling in the unimplemented sections (marked with `TODO`).
- Implement ϵ -greedy sampling in `get_action`.
- Implement the `TODOs` in `run_dqn.py`.
- 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`. It should reach a return of 500 at least once during training

```
uv run src/scripts/run_dqn.py -cfg experiments/dqn/cartpole.yaml --eval_interval 2500
```

Deliverables:

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

2.5 Double Q-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:

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

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

In our case, we'll keep using the target network $Q_{\varphi'}$ to estimate the action's value, but we'll select the action using Q_{φ} (the online Q network). This stabilizes learning and reduces the value overestimation that can cause instability in vanilla DQN.

Implement this functionality in `dqn_agent.py`.

Deliverables:

- Run double-Q DQN on **LunarLander-v2**:

```
uv run src/scripts/run_dqn.py -cfg experiments/dqn/lunarlander.yaml
```

You should expect a return of **200** at least once during training. Plot the eval return over training steps.

- 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** at least once during training.

```
uv run src/scripts/run_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.

2.6 Experimenting with Hyperparameters

Now let's analyze the sensitivity of Q-learning to hyperparameters on **LunarLander-v2**. Choose one hyperparameter of your choice and run at least three other settings of this hyperparameter, in addition to the one used in the previous questions, 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.

Hyperparameter options could include:

- Learning rate
- Network architecture
- Exploration schedule
- Target network update frequency

3 Soft Actor-Critic (SAC)

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:

```
uv run src/scripts/run_sac.py -cfg experiments/sac/<CONFIG>.yaml
```

3.1 Implementation

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

- `src/scripts/run_sac.py` - the main training loop for your SAC implementation.
- `src/agents/sac_agent.py` - the structure for the SAC learner you'll implement.

You may also find the following files useful:

- `src/networks/critic.py` - a simple MLP-based $Q(s, a)$ network. Note that unlike the DQN critic, which maps states to an array of Q-values, one per action, this critic maps one (s, a) pair to a single Q-value.
- `src/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 `src/agents/sac_agent.py`.

Before implementing SAC:

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

3.2 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:

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

$$\min_{\varphi} (Q_{\varphi}(s_t, a_t) - y)^2$$

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

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

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

What you'll need to do (in `src/agents/sac_agent.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 `InvertedPendulum-v4` with the sample configuration:

It shouldn't get high return yet (you're not training an actor), but the Q-values should stabilize at some reasonable value. If the Q-values go to infinity or stay at zero, you probably have a bug.

```
uv run src/scripts/run_sac.py -cfg experiments/sac/sanity_invertedpendulum.yaml
```

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

3.3 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 . The actor objective to maximize is:

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

In practice, we minimize the loss $\mathcal{L}_\pi = -J_\pi$. 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:

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

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>.

What you'll need to do (in `src/agents/sac_agent.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()`.
- Enable the actor update by calling `self.update_actor()` in the `update()` method.

Testing this section:

- Run the sanity check on `InvertedPendulum-v4`:

```
uv run src/scripts/run_sac.py -cfg experiments/sac/sanity_invertedpendulum.yaml
```

The code should be logging `entropy` during training. Because the actor loss **only** consists of the entropy bonus at this stage (we haven't implemented anything to maximize rewards yet via reparametrization), the entropy should increase until it approaches the maximum entropy for the action space. For a tanh-squashed 1D action space, this is approximately $\log 2 \approx 0.69$.

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

3.4 Actor Update with Reparametrization

To update the actor, we want to compute:

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

We can do this efficiently using the reparametrized gradient. Parametrize π_{θ} as $\mu_{\theta}(s) + \sigma_{\theta}(s)\epsilon$, where ϵ is normally distributed. Then we can write:

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

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.yaml`) and achieve return close to 1000, which corresponds to staying upright for all time steps.

```
uv run src/scripts/run_sac.py -cfg experiments/sac/sanity_invertedpendulum.yaml
```

Deliverables:

- Train on `HalfCheetah-v4` with `halfcheetah.yaml`. Plot results with number of environment steps on the x -axis and evaluation return on the y -axis. You should expect to achieve a return of at least **6000** at least once during training.

```
uv run src/scripts/run_sac.py -cfg experiments/sac/halfcheetah.yaml
```

3.5 Automatic Temperature Tuning

In the previous sections, you manually set the temperature coefficient β for each environment. However, finding the optimal temperature requires manual tuning per task, and the ideal value may change during training.

The SAC paper (<https://arxiv.org/abs/1812.05905>, Section 5) provides a principled approach to *automatically* adjust β by formulating temperature selection as a constrained optimization problem solved via *dual gradient descent*.

The Constrained Optimization Problem:

Instead of fixing β , we want to maximize expected return while maintaining a minimum entropy level:

$$\max_{\theta} \mathbb{E} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad \text{subject to} \quad \mathbb{E}_{(s_t, a_t) \sim \rho_{\pi}} [\mathcal{H}(\pi(\cdot | s_t))] \geq \mathcal{H}_{\text{target}}$$

Where $\mathcal{H}_{\text{target}}$ is a target entropy (typically set to $-\dim(\mathcal{A})$ for continuous action spaces).

Dual Gradient Descent:

Using Lagrange multipliers, we introduce α (the temperature) as a *dual variable* that enforces the entropy constraint. The algorithm alternates between:

- **Primal updates:** Update actor θ and critics φ to maximize expected return given current α
- **Dual updates:** Update α to enforce the entropy constraint

The dual update for α is:

$$\nabla_{\alpha} J(\alpha) = \mathbb{E}_{s_t \sim \mathcal{D}, a_t \sim \pi} [-\alpha (\log \pi(a_t | s_t) + \mathcal{H}_{\text{target}})]$$

Implementation Details:

- Parameterize as $\alpha = \exp(\log \alpha)$ to ensure $\alpha > 0$
- Make $\log \alpha$ a learnable parameter with its own optimizer
- Update $\log \alpha$ after each actor/critic update

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

- Add a learnable parameter `self.log_alpha` (initialized to match the starting temperature)
- Create a separate optimizer `self.alpha_optimizer` for `log_alpha`
- Set target entropy: `self.target_entropy = -action_dim`
- Implement `update_alpha()` method to compute and apply the dual gradient
- Implement `get_temperature()` to return the learned temperature when auto-tuning is enabled
- Call `update_alpha()` in your main `update()` loop

Testing this section:

- Test on `InvertedPendulum-v4` to verify it still achieves return close to 1000.

```
uv run src/scripts/run_sac.py -cfg experiments/sac/sanity_invertedpendulum_autotune.yaml
```

Deliverables:

- Run auto-tuned SAC on `HalfCheetah-v4`:

```
uv run src/scripts/run_sac.py -cfg experiments/sac/halfcheetah_autotune.yaml
```

- Create a figure with two subplots:
 - Eval return over training for both fixed temperature (from Section 3.4) and auto-tuned temperature
 - Temperature value (α) over training for the auto-tuned run
- Answer the following questions:
 - Does auto-tuning improve or achieve comparable performance to the fixed temperature?
 - How does the temperature evolve during training? Does it increase, decrease, or stabilize?
 - Why might the temperature change in this particular way for `HalfCheetah`?

Note: Since the default fixed temperature is already well-tuned for `HalfCheetah`, auto-tuning may not necessarily outperform it. The main benefit of auto-tuning is that it removes the need for manual temperature selection, which is especially valuable when deploying SAC on new environments where a good fixed value is unknown.

Optional experiment: To illustrate this point, try sweeping over several fixed temperature values (e.g., $\beta \in \{0.01, 0.05, 0.1, 0.5, 1.0\}$) on `HalfCheetah-v4` and plot all of them alongside the auto-tuned run. Include this plot in your report and discuss how sensitive performance is to the choice of fixed temperature, and whether auto-tuning finds a competitive value without manual search.

3.6 Stabilizing Target Values

As in DQN, the target Q with a single critic exhibits overestimation bias! To combat this, we can use clipped double- Q : learn two critics $Q_{\varphi_A}, Q_{\varphi_B}$ (and keep two target networks $Q_{\varphi'_A}, Q_{\varphi'_B}$). Then, compute the target values by taking the minimum of both critics:

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

This pessimistic estimate reduces overestimation by using the lower of the two Q -value predictions.

Implement clipped double- Q (the “min” case) in the `q_backup_strategy` function in `sac_agent.py`. The “mean” case for single- Q is already provided.

Deliverables:

- Run single- Q and clipped double- Q on Hopper-v4 using the corresponding configuration files. You should expect the clipped double- Q to achieve a return of at least **1500** at least once during training. 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.

```
uv run src/scripts/run_sac.py -cfg experiments/sac/hopper_singleq.yaml
uv run src/scripts/run_sac.py -cfg experiments/sac/hopper_clipq.yaml
```

4 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. We identify the experiment runs by their prefixes (up to the “_sd” part in the directory name), and they should exactly match the tree structure below.
- The `src` folder with all the `.py` files, with the same names and directory structure as the original homework repository.

To create the zip file, run the following:

```
zip -r submit.zip exp src README.md pyproject.toml uv.lock
```

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.** Do **not** nest the `exp` and `src` directories within another parent directory.

```
submit.zip
├── exp
│   ├── CartPole-v1_dqn_sd1_20260301_012345
│   │   ├── agent.pt
│   │   ├── flags.json
│   │   ├── log.csv
│   │   └── log.pkl
│   ├── LunarLander-v2_dqn_sd1_20260301_012345
│   ├── MsPacman_dqn_sd1_20260301_012345
│   ├── HalfCheetah-v4_sac_sd1_20260301_012345
│   ├── HalfCheetah-v4_sac_autotune_sd1_20260301_012345
│   ├── Hopper-v4_sac_singleq_sd1_20260301_012345
│   └── Hopper-v4_sac_clipq_sd1_20260301_012345
├── src
│   ├── agents
│   │   ├── ...
│   │   ├── dqn_agent.py
│   │   ├── sac_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 **HW3 Code**, and upload the PDF of your report to **HW3 Report**.