CS224R Spring 2026 Homework 2
Online Reinforcement Learning

Due 4/24/2026

SUNet ID:
Name:
Collaborators:

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

Overview

Goals: In this assignment, you will implement Q-learning on a gridworld and implement RL algorithms to solve a realistic robot task, in which the agent needs to pick up a tool and use it to hammer a nail. The agent controls a 4 degree-of-freedom Sawyer robot with a continuous action space and receives environment states. Providing dense rewards for real world problems is difficult, so in this environment the agent receives a sparse reward of 1.0 upon fully completing the task and no intermediate rewards.

The implementations will be split into three parts. In the first part, you will implement a Q-Learning algorithm on a gridworld to test out the effects of different reward functions. In the second part, you will implement an on-policy algorithm. In the third part, you will implement an off-policy algorithm. This homework is intended to allow you to explore some of the design choices of online RL algorithms and how they affect performance.

Setup

For this assignment, you will complete all sections on Modal instances. **We recommend only using Modal for compute**, as we **do not** support setup on any other platform, such as your own local computer. You may still develop code on your local computer, which we suggest to save compute credits. You may find this guide helpful on how to set up your code to run on modal.

For simplicity, for this homework we already provide you with the entry points to run on modal, so you won't have to worry about setting it up yourself, but this guide will be handy for the homework.

For set up, create a conda environment that you will use to launch modal with from your local machine:

```
conda env create -f conda_env_modal.yml
conda activate cs224r-hw2
```

We also provide a conda environment for running your code locally in the case that you have a linux machine with an nvidia GPU. However, this is just a reference for those who would like to give it a shot and won't be supported by the teaching staff.

```
conda env create -f conda_env_local.yml
conda activate cs224r-hw2-local
```

Running Long Jobs on Modal: Some of the training jobs in the homework can take a long time above 1 hour. You can use `--detach` to run in in the background so you don't need to keep your terminal open (you can still see your jobs on wandb), e.g.:

```
modal run --detach modal_on_policy.py
```

To terminate the job you can go to your workspace, your live apps, and click on *Stop app*.

Weights and Biases Experiment Tracking: This homework uses Weights and Biases (Wandb) logging to track training stats. You will need a wandb account at https://wandb.ai/site/.

Step 1: Create a wandb account.
Go to https://wandb.ai/site and sign up for a free account. You can authenticate using your existing GitHub or Google credentials.

Step 2: Install the wandb library.
Install via `conda` in your terminal (or add it to your `environment.yml`):
`conda install -c conda-forge wandb`
This step should be handled automatically when installing the conda environment.

Step 3: Authenticate with your API key.
Run the login command and paste your API key when prompted:
`wandb login`
Your API key can be found at https://wandb.ai/authorize. The key is saved locally, so you only need to run this once per machine.

If you have any issues with the wandb key not found when running on modal try running the following:

```
modal secret create wandb-secret WANDB_API_KEY=<wandb_key> --force
```

Step 4: Viewing your results.
Once training starts, wandb prints a direct URL to your terminal. Open it to see your metrics update in real time or you can also go to https://wandb.ai/home and click into the

project.

Submitting the PDF, Code, and Runs: Make a PDF report containing: observed outcomes from Question 1, training curves from Question 2, and 3, and your responses.

Submit the **grid_world_q_learning.py**, **on_policy.py** and **off_policy.py** file and the csv downloaded from Wandb for the final runs.

To download the csv file, go to "Charts" on your wandb run, go to eval/episode_success section and click on the three dots on the top right corner as shown in the picture below:

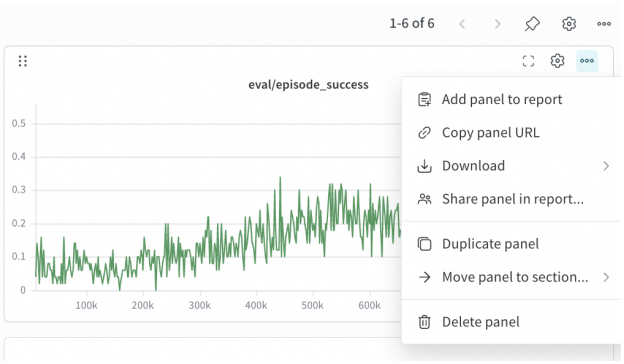

Figure 1: CSV file download instruction.

Gradescope: Submit both the PDF and the zipped code and experiment runs in the appropriate assignment on Gradescope.

Use of AI Tools (e.g. ChatGPT, Cursor) For the sake of deeper understanding on implementing actor-critic methods, assistance from generative models to write code for this homework is prohibited. See the course website for more details.

Sample File Submission

For this homework, you should submit the **gridworld_q_learning.py**, **on_policy.py** and **off_policy.py** file, as well as the logs from your final three runs, which are saved in the logging folder. Your submission file should look like this

```
submit.zip
├─ gridworld_q_learning.py
├─ on_policy.py
├─ off_policy.py
└─ CSV_files
   ├─ on_policy.csv
   ├─ off_policy.num_critics=2,utd=1.csv
   └─ off_policy.num_critics=10,utd=5.csv
```

We advise you to start as early as possible since the assignment requires longer compute times.

Problem 1 [3 points]

We implement a Q-learning agent on a 2D grid world with discrete states and discrete actions. The grid world has two goals at different distances from the start. The grid is 5×4 (i.e. there are 20 states), with the agent always starting at $(0, 0)$, **Goal 2** at $(4, 0)$ (4 steps away), and **Goal 1** at $(4, 3)$ (7 steps away). At each step the agent chooses from four actions: $\{\texttt{left}, \texttt{right}, \texttt{up}, \texttt{down}\}$, which move it one step on the grid. If the agent attempts to move off the edge, the agent stays in place.

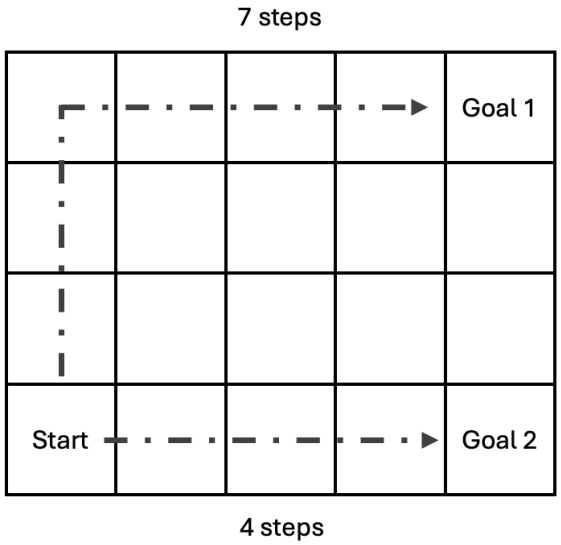

Figure 2: Visual showing the grid.

Algorithm: Tabular Q-Learning

Instead of using a neural network to represent the Q-function, the agent maintains a table of Q-values $Q(s, a)$ initialized to zero, with one Q-value for each state and action (i.e. the table contains $40 \times 4 = 160$ scalar Q-values). When making updates, instead of updating the parameters of a neural network, we update the Q-values in the table directly following this Q-learning update:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\Big]$$

where α is the learning rate and γ is the discount factor.

Your Tasks

1. **Implement** `choose_action` **in** `grid_world_q_learning.py`**.**

Complete the `choose_action` function. With probability ε the agent selects a uniformly random action; otherwise it selects the greedy action $a^* = \arg\max_a Q(s, a)$. Break ties arbitrarily.

2. **Implement `train_q_learning` in `grid_world_q_learning.py`.**

   Complete the inner loop of `train_q_learning`. For each step in an episode:

   (a) Choose an action using ε-greedy policy.

   (b) Step the environment to obtain $(s_{t+1}, r_t, \texttt{done})$.

   (c) Apply the Q-learning update above.

   (d) Terminate the episode if `done` is `True`.

3. **Three Scenarios. [1 point each scenario]**

   Run `python gridworld_q_learning.py` on each scenario below and report the learned trajectory and total reward.

   The reward received at each step depends on the action taken.

   Let $a \in \{\texttt{left}, \texttt{right}, \texttt{up}, \texttt{down}\}$ be the action, s the current state, and s' the resulting next state. The per-step reward r_{step} is always incurred, and a goal bonus is added if and only if the agent transitions into a terminal state:

$$r(s, a, s') \;=\; r_{\text{step}} + R_1 \, \mathbf{1}[s' = \text{Goal 1}] + R_2 \, \mathbf{1}[s' = \text{Goal 2}] \tag{1}$$

   Note that s' is fully determined by (s, a) via the deterministic transition function of the grid (with boundary clamping: moving into a wall leaves the agent in place but still incurs r_{step}).

   - **Scenario 1** ($r_{\text{step}} = -1$, $R_1 = 10$, $R_2 = 5$)
   - **Scenario 2** ($r_{\text{step}} = -2$, $R_1 = 10$, $R_2 = 5$)
   - **Scenario 3** ($r_{\text{step}} = +1$, $R_1 = 1$, $R_2 = 1$)

   For each scenario, report whether it reaches a goal, and if so which one it reaches. Give a one sentence explanation for why it learned that trajectory in each scenario.

Problem 2 [3 points]

We will implement an on-policy agent that learns to solve the manipulation task of hammering a nail, which collects rollouts under the current policy and then performs multiple epochs of gradient updates.

Note: "on-policy" in this context

Strictly speaking, on-policy learning requires that every batch of data is collected under the *current* policy and discarded after exactly one gradient update. The agent you implement here is not strictly on-policy by that definition: it collects rollouts under the current policy but then performs *multiple* epochs of gradient updates on the same batch. We nonetheless call it on-policy because the data is always freshly collected under (a recent snapshot of) the current policy, which contrasts sharply with the off-policy agent in the next problem, where data may have been collected by an arbitrarily old policy and is stored in a replay buffer for reuse.

The algorithm you will implement is PPO[1], which uses a clipped surrogate objective. The full agent is implemented in **on_policy.py** and consists of:

- An **actor** $\pi_\theta(a_t|s_t)$ that outputs a `TruncatedNormal` distribution over actions. A TruncatedNormal is a Gaussian distribution that has been clipped to a finite interval — here $[-1, 1]$ to match the action bounds of the environment. If you sample from a standard Normal and the sample falls outside $[-1, 1]$, it gets clipped back to the boundary rather than being accepted. We are using a Truncated Normal because the action space of the environment is between -1 and 1.
- A **value function** $V_\phi(s_t)$ that estimates the state value function. The PPO value function conditions only on states (not actions), and we compute advantages via Generalized Advantage Estimation (GAE).[2].
- A frozen **reference actor** $\pi_{\theta_{\text{ref}}}$, a snapshot of the policy taken after behavior-cloning pre-training. It is used during PPO updates to regularize the policy via a reverse-KL penalty, preventing it from drifting too far from the pre-trained initialization.

You should familiarize yourself with the `PPOAgent` class and its helper methods before starting. You should not modify any files other than **on_policy.py**.

Your Tasks

1. **Generalized Advantage Estimation (GAE).**

[1] John Schulman et al., "Proximal Policy Optimization Algorithms," 2017. arXiv:1707.06347

[2] John Schulman et al., "High-Dimensional Continuous Control Using Generalized Advantage Estimation," ICLR 2016. arXiv:1506.02438

Complete the **compute_gae** method of the **PPOAgent** class. Given a trajectory of length T, compute the GAE advantages $\hat{A}_t$ and the corresponding returns (value targets) $\hat{R}_t = \hat{A}_t + V_\phi(s_t)$.

GAE computes advantages by bootstrapping a multi-step TD residual with an exponential decay controlled by λ. Refer to the lecture on Actor Critic methods for details. Concretely, iterate *backwards* through time steps $t = T-1, \ldots, 0$:

$$\delta_t = r_t + \gamma\,(1 - d_t)\,V_\phi(s_{t+1}) - V_\phi(s_t)$$
$$\hat{A}_t = \delta_t + \gamma\,\lambda\,(1 - d_t)\,\hat{A}_{t+1}$$

where γ is the discount factor, λ is the GAE smoothing parameter (`self.gae_lambda`), and $d_t \in \{0, 1\}$ is the episode-termination flag.

2. **PPO Update.**

   Complete the two `### YOUR CODE HERE ###` blocks inside the **update** method.

   (a) **Compute GAE advantages and returns (before the epoch loop).**
   Using the full rollout stored in `obs_all`, `rew_all`, `disc_all`, `next_obs_all`, and `done_all`, obtain value estimates from the critic and call `compute_gae` to produce `advantages_all` and `returns_all`. Prevent gradient updates on these values by wrapping this block in `torch.no_grad()` since the targets must be treated as fixed.

   (b) **Clipped surrogate objective (inside the minibatch loop).**
   Given the ratio of new to old action probabilities

$$\rho_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} = \exp\big(\log \pi_\theta(a_t|s_t) - \log \pi_{\theta_{\text{old}}}(a_t|s_t)\big),$$

   compute the PPO-Clip policy loss:

$$\mathcal{L}^{\text{CLIP}}(\theta) = -\frac{1}{B}\sum_t \min\Big(\rho_t\,\hat{A}_t,\ \text{clip}(\rho_t, 1-\varepsilon, 1+\varepsilon)\,\hat{A}_t\Big)$$

   where ε is `self.clip_eps` and B is the minibatch size. Store the result in `policy_loss`.
   Step 1: Write ρ_t and set it as `ratio`. (`ratio` will then be used in `reverse_kl`.)
   Step 2: Write the PPO-Clip policy loss and set it as `policy_loss`.
   In practice, when computing the ratios, use the log-probability form on the right. Dividing probabilities directly is numerically unstable: in high-dimensional action spaces the individual probabilities $\pi_\theta(a_t \mid s_t)$ can be extremely small, leading to underflow or overflow. Working in log-space and exponentiating the difference is both numerically stable and cheaper to compute, since the actor network already outputs log-probabilities.
   Note: The quantities `new_log_prob`, `old_lp_all` (indexed by `batch_idx` as `olp_ep`), `adv_ep`, and the ratio variable `ratio` are all available in scope — you only need to implement the clipping and the loss.

3. **Training. [3 points]**

   Once your implementation is complete, train PPO with the command:

   ```
   modal run modal_on_policy.py
   ```

   Attach a screenshot of the plot `eval/episode_success` for this run up to 1 million steps from Wandb. You should achieve at least 25% success rate.

Problem 3 [4 points]

We will implement a general actor-critic algorithm that learns to solve the task with almost 100% success rate. At each step the agent processes states s_t, which consists of the environment states (e.g., pose of the robot gripper and hammer, etc.) from the last two environment steps. The full algorithm consists of the following components, each in **off_policy.py**.

- The actor-critic algorithm utilizes critic networks $Q_{\phi^i}(s_t, a_t)$. This is implemented as the **Critic** class in **off_policy.py**. We use an ensemble with $i = 1 \dots N$ different critic networks. Using an ensemble of critic networks can help reduce error in the Q-value estimates.
- Each critic network has a corresponding *target* critic network $\bar{Q}_{\phi^i}(s_t, a_t)$, which is used for computing the Bellman targets for critic updates. We use target critic networks, which are updated more slowly, to have a more stable target for learning.
- The final component of the algorithm is the Actor or policy $\pi_\theta(a_t|s_t)$. This is implemented as the **Actor** class in **off_policy.py**.

You should familiarize yourself with these functions and their inputs and outputs. You should not modify any other files besides **off_policy.py**.

Your Tasks

1. **BC Pretraining.** Optimizing behavior in environments with sparse rewards, which means the agent receives a reward signal when it completes the task and gets zero (or a constant) for every other step, is difficult due to limited reward supervision. To alleviate this, we provide the agent with 20 successful demonstrations, which we will use to pre-train with behavior cloning. Note this is the same as the reference policy used in Problem 1. In **off_policy.py** complete the **bc** function of the **ACAgent** class to train the policy using supervised behavior cloning. That is, given state-action pairs s_t, a_t optimize the loss

   $$\mathcal{L}_{\pi_\theta}(s_t, a_t) = -\log \pi_\theta(a_t|s_t)$$

   We will also use this to update the actor during later RL training, to improve stability.

2. **Update Agent.** In the second part, we will try to improve the performance of the policy with additional fine-tuning with reinforcement learning. Reinforcement learning allows the robot to improve using its own experience, without needing additional demonstrations. Your implementation will be in the **ACAgent** class.

   - We begin by implementing the **update_critic** method using the Bellman objective. Consider transitions $(s_t, a_t, r_t, s_{t+1}, \gamma)$ and implement the following steps:

     (a) Sample next state actions from the policy $a_{t+1} \sim \pi_\theta(s_{t+1})$.

(b) Compute the Bellman targets

$$y = r_t + \gamma \min\{\bar{Q}_{\phi^i}(s_{t+1}, a'_{t+1}), \bar{Q}_{\phi^j}(s_{t+1}, a'_{t+1})\}$$

where $\bar{Q}_{\phi^i}$ and $\bar{Q}_{\phi^j}$ are two randomly sampled target critics. We take the minimum over two Q-values to mitigate critic overestimation errors. (**Tip:** `random.sample(list, N)` samples without replacement).

(c) Compute the loss:

$$\mathcal{L}_{Q_\phi, f_\theta} = \sum_{i=1}^{N} (Q_{\phi^i}(s_t, a_t) - \mathbf{sg}(y))^2$$

where sg stands for the stop gradient operator.
Note: We compute the loss for all N critic networks.

(d) Take a gradient step with respect to the critic parameters.

(e) Update the target critic parameters using an exponential moving average.

$$\bar{Q}_{\phi^i} = (1 - \rho)\bar{Q}_{\phi^i} + \rho Q_{\phi^i}$$

By using an exponential moving average, our target critic parameters are updated more slowly than those of the main critic.
Note: Check the soft_update_params function in utils.py.

- Next, we will improve the policy in the **update_actor** method. Sample an action from the actor $a_t \sim \pi_\theta(\cdot|s_t)$ and compute the objective that optimizes the actor to maximize the Q-value estimates from the critics:

$$\mathcal{L}_{\pi_\theta} = -\frac{1}{N} \sum_{i=1}^{N} Q_{\phi^i}(s_t, a'_t)$$

Take a gradient step on this objective with respect to the policy only.

- **Training. [1 point]**
Once you are done, run the RL fine-tuning with

```
modal run modal_off_policy.py
```

Attach a screenshot of the plot "eval/episode_success" for this run from Wandb up to 100k steps. You should achieve a success rate of at least 90% before 100K environment steps.

- **Analysis of UTD choice. [1 point]**
In the final part, we will explore some optimization parameter choices. The update-to-data (UTD) ratio refers to the number of critic gradient steps we do after each environment step taken. So far we have used only 2 critics and UTD

of 1. Repeat the previous part but change the argv inside the subprocess of modal_off_policy.py to use 10 num_critics and utd = 5 as follows:

```
    result = subprocess.run(
        [
            "python", "-u", "train_off_policy.py",
            "--config-path", "/root/cfgs",
            "--config-name", "off_policy_config",
            "agent.num_critics=10",
            "utd=5"
        ],
```

And then run again, but now with the new parameters:

```
modal run modal_off_policy.py
```

Here, we perform critic gradient steps 5x more often.

Attach a screenshot of the plot "eval/episode_success" for this run up to 40k steps and compare it to the previous question. You should achieve a success rate of at least 90% before 40k environment steps. Provide a brief, one-sentence explanation of why we observe these effects.

Note: this is slower to run due to the higher computational cost and you should expect run time of about 2 hours for 50,000 steps.

Common Bugs

- If your critic loss curves remain flat throughout training, make sure the critics are being updated.
- If your critic losses become very large, make sure the dimensions of your critic predictions and targets are correct and/or tensor broadcasting is done correctly.
- Make sure all critics are updated each iteration, not just those corresponding to the two target critics that are sampled for computing Bellman targets.

3. **Comparison and Analysis. [2 points]**

Compare the PPO learning curve (`eval/episode_success`) from Problem 2 with the actor-critic curve from Problem 3 (2 critics, UTD=1).

In 3–5 sentences, explain at least two concrete differences you observe between the two algorithms in terms of sample efficiency and final performance. Justify each observed difference by connecting it to a specific property of each algorithm.