# Homework 1

CMU 10-703: Deep Reinforcement Learning (Fall 2024)

OUT: September $4^{th}$

DUE: September $23^{th}$ by 11:59pm ET

## Instructions: START HERE

- **Collaboration policy:** You may work in groups of up to three people for this assignment. It is also OK to get clarification (but not solutions) from books or online resources after you have thought about the problems on your own. You are expected to comply with the University Policy on Academic Integrity and Plagiarism[1].
- **Late Submission Policy:** You are allowed a total of 8 grace days for your homeworks. However, no more than 3 grace days may be applied to a single assignment. Any assignment submitted after 3 days will not receive any credit. Grace days do not need to be requested or mentioned in emails; we will automatically apply them to students who submit late. We will not give any further extensions so make sure you only use them when you are absolutely sure you need them. See the Assignments and Grading Policy for more information about grace days and late submissions [2]
- **Submitting your work:**
  - **Gradescope:** Please write your answers and copy your plots into the provided LaTeX template, and upload a PDF to the GradeScope assignment titled "Homework 1." Additionally, export your code ([File → Export .py (if using Colab notebook)]) and upload it the GradeScope assignment titled "Homework 1: Code." Each team should only upload one copy of each part. Regrade requests can be made within one week of the assignment being graded.

[1] `https://www.cmu.edu/policies/`

[2] `https://cmudeeprl.github.io/703website_f24/logistics/`

# Collaborators

Please list your name and Andrew ID, as well as those of your collaborators.

# Problem 1: Value Iteration & Policy Iteration (30 pts)

## Problem 1.1: Contraction Mapping (3 pts)

Answer the true/false questions below, providing one or two sentences for **explanation**. You can use theorems and information mentioned in lecture slides.

1. If $f : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ has a fixed point, then it is a contraction mapping w.r.t. the euclidean norm.

2. Let $S, \rho$ be a complete metric space, where $S$ is the set and $\rho$ is the metric. If a function $f : S \longrightarrow S$ is a contraction mapping then there exists a unique $x^*$ such that $f(x^*) = x^*$.

3. Let $\{\pi_k\}$ be the sequence of of policies generated by the policy iteration algorithm. Then $F^{\pi_{k+1}} = F^{\pi_k}$ if and only if $F^{\pi_k} = F^{\pi_*}$, where $F$ is the Bellman expectation backup operator.

## VI/PI implementation (27 pts)

In this problem, you will implement value iteration and policy iteration. Throughout this problem, initialize the value functions as zero for all states and break ties in order of the action numbering (L, D, R, U), if you break ties randomly, you will not get the correct answer.

We will be working with a different version of the OpenAI Gym environment `Deterministic-*-FrozenLake-v0`[3], defined in `code/frozen_lake/lake_init.py`. You can check `README.md` for specific coding instructions. Starter code is provided in `code/frozen_lake/pi_vi.py`, with useful helper functions at the end of the file!

We have provided two different maps, a $4 \times 4$ map and a $8 \times 8$ map:

```
FHSF
FGHF
FHHF
FFFF
```

```
FFFFFSFF
FFFFFFFF
HHHHHHFF
FFFFFFFF
FFFFFFFF
FHFFFHHF
FHFFHFHH
FGFFFFFF
```

[3] `https://gymnasium.farama.org/environments/toy_text/frozen_lake/`

There are four different tile types: Start (S), Frozen (F), Hole (H), and Goal (G).

- The agent starts in the Start tile at the beginning of each episode.
- When the agent lands on a Frozen or Start tile, it receives 0 reward.
- When the agent lands on a Hole tile, it receives 0 reward and the episode ends.
- When the agent lands on the Goal tile, it receives +1 reward and the episode ends.

States are represented as integers numbered from left to right, top to bottom starting at zero. For example in a $4 \times 4$ map, the upper-left corner is state 0 and the bottom-right corner is state 15:

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{array}$$

**Note:** Be careful when implementing value iteration and policy evaluation. Keep in mind to make sure the reward function ($r(s, a, s')$) is correctly considered. Also, terminal states are slightly different. Think about the backup diagram for terminal states and how that will affect the Bellman equation.

In this section, we will use the **deterministic** versions of the FrozenLake environment. Answer the following questions for the maps `Deterministic-4x4-FrozenLake-v0` and `Deterministic-8x8-FrozenLake-v0`.

In the following sub-problems (**Problems 1.2** to **1.5**) you will implement *synchronous* and *asynchronous* versions of policy iteration and value iteration. The main difference between the sync and async versions is: whether all the updates are performed in-place (async) or not (sync). Take the synchronous value iteration for example, at some time step $k$, you would maintain two separate vectors $V_k$ and $V_{k+1}$ and perform updates of the following form for each value update:

$$V_{k+1}(s) \longleftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V_k(s')].$$

For asynchronous updates, you don't need two copies of the vector as above.

The pseudo-codes for policy iteration and value iteration are provided in Figure 1 and Figure 2, respectively. Please note that, strictly speaking, the pseudo-code implements *asynchronous* versions. You will need to make appropriate modifications to implement the synchronous versions where prompted.

**Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$**

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s)) \big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      *old-action* $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 1: Policy iteration, taken from Section 4.3 of Sutton & Barto's RL book (2018).

**Value Iteration, for estimating $\pi \approx \pi_*$**

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\textit{terminal}) = 0$

Loop:
|  $\Delta \leftarrow 0$
|  Loop for each $s \in \mathcal{S}$:
|  $v \leftarrow V(s)$
|  $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$
|  $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$

Figure 2: Value iteration, taken from Section 4.4 of Sutton & Barto's RL book (2018).

## Problem 1.2: Synchronous Policy Iteration

1. (4 pts) For each domain, find the optimal policy using **synchronous policy iteration**. Specifically, you will implement `policy_iteration_sync()` in `code/frozen_lake/pi_vi.py`, writing the policy evaluation steps in `evaluate_policy_sync()` and policy improvement steps in `improve_policy()`. Record (1) the number of policy improvement steps and (2) the total number of policy evaluation steps. Use a discount factor of $\gamma = 0.9$. Use a stopping tolerance of $\theta = 10^{-3}$ for the policy evaluation step.

| Environment | # Policy Improvement Steps | Total # Policy Evaluation Steps |
|---|---|---|
| Deterministic-4x4 | | |
| Deterministic-8x8 | | |

2. (2 pts) Show the optimal policy for the Deterministic-4x4 and 8x8 maps as grids of letters with "L", "D", "R", "U" representing the actions left, down, right, up respectively. See Figure 3 for an example of the 4x4 map. Helper: `display_policy_letters()`.

3. (2 pts) Find the value functions of the policies for these two domains. Plot each as a color image, where each square shows its value as a color. See Figure 4 for an example for the 4x4 domain. Helper function: `value_func_heatmap()`.

```
LLLL
RRRR
UUUU
DDDD
```

Figure 3: An example (deterministic) policy for a $4 \times 4$ map of the `FrozenLake-v0` environment. L, D, R, U represent the actions left, down, right, up respectively.

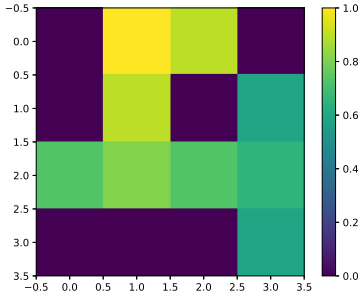


Figure 4: Example of value function color plot for a $4 \times 4$ map of the `FrozenLake-v0` environment. Make sure you include the color bar or some kind of key.

## Problem 1.3: Synchronous Value Iteration

1. (3 pts) For both domains, find the optimal value function directly using **synchronous value iteration**. Specifically, you will implement `value_iteration_sync()` in `code/frozen_lake/pi_vi.py`. Record the number of iterations it took to converge. Use $\gamma = 0.9$ Use a stopping tolerance of $10^{-3}$.

| Environment | # Iterations |
|---|---|
| Deterministic-4x4 | |
| Deterministic-8x8 | |

2. (2 pts) Plot these two value functions as color images, where each square shows its value as a color. See Figure 4 for an example for the 4x4 domain.

3. (2 pts) Convert both optimal value functions to the optimal policies. Show each policy as a grid of letters with "L", "D", "R", "U" representing the actions left, down, right, up respectively. See Figure 3 for an example of the expected output for the 4x4 domain.

## Problem 1.4: Asynchronous Policy Iteration

1. (4 pts) Implement **asynchronous policy iteration** using two heuristics:

   (a) The first heuristic is to sweep through the states in the order they are defined in the gym environment. Specifically, you will implement `policy_iteration_async_ordered()` in `code/frozen_lake/pi_vi.py`, writing the policy evaluation step in `evaluate_policy_async_ordered()`.

   (b) The second heuristic is to choose a random permutation of the states at each iteration and sweep through all of them. Specifically, you will implement `policy_iteration_async_randperm()` in `code/frozen_lake/pi_vi.py`, writing the policy evaluation step in `evaluate_policy_async_randperm()`.

   Fill in the table below with the results for Deterministic-8x8-FrozenLake-v0. Run one trial for the **first** ("async_ordered") heuristic. Run **ten trials** for the **second** ("async_randperm") heuristic and report the **average**. Use $\gamma = 0.9$. Use a stopping tolerance of $10^{-3}$.

| **Heuristic** | **Policy Improvement Steps** | **Total Policy Evaluation Steps** |
|---|---|---|
| Ordered | | |
| Randperm | | |

## Problem 1.5: Asynchronous Value Iteration

1. (4 pts) Implement **asynchronous value iteration** using two heuristics:

   (a) The first heuristic is to sweep through the states in the order they are defined in the gym environment. Specifically, you will implement `value_iteration_async_ordered()` in `code/frozen_lake/pi_vi.py`.

   (b) The second heuristic is to choose a random permutation of the states at each iteration and sweep through all of them. Specifically, you will implement `value_iteration_async_randperm()` in `code/frozen_lake/pi_vi.py`.

   Fill in the table below with the results for Deterministic-8x8-FrozenLake-v0. Run one trial for the **first** ("async_ordered") heuristic. Run **ten trials** for the **second** ("async_randperm") heuristic and report the **average**. Use $\gamma = 0.9$. Use a stopping tolerance of $10^{-3}$.

| **Heuristic** | **# Iterations** |
|---|---|
| Ordered | |
| Randperm | |

2. (4 pts) Now, you can use a domain-specific heuristic for asynchronous value iteration (`value_iteration_async_custom()`) to beat the heuristics defined in Q1.5.1. Specifically, you will sweep through the entire state space ordered by Manhattan distance to goal.

(a) Fill in the table below (use a stopping tolerance of $10^{-3}$).

| Env | # Iterations |
|---|---|
| Deterministic-4x4 | |
| Deterministic-8x8 | |

(b) Name at least one case(s) that you expect this "goal distance" heuristic to perform well? Briefly explain why.

# Problem 2: Bandits (36 pts)

This problem will explore a few different exploration algorithms for bandits. The aim of the problem is to understand how each exploration algorithm works and the requirements of each. We provide a simple template in `code/bandits/bandits.py`.

We will use the "10-armed Testbed" described in Sutton+Barto Section 2.3. There are $k = 10$ arms. The average reward for each arm is sampled from $\mathcal{N}(1, 1)$. When the agent pulls arm $a$, they observe a noisy reward $r \sim \mathcal{N}(r(a), 1)$ (i.e., they observe $r(a)+\mathcal{N}(0, 1)$). For the following questions, each time you make a run of an algorithm, you should re-instantiate the mean rewards.

1. **[8 pts]** Implement $\epsilon$-greedy exploration. Plot the expected reward (Y-axis) over 1000 time steps (X-axis) for the following values of $\epsilon$: [0, 0.001, 0.01, 0.1, 1.0]. The results for each value of $\epsilon$ should be shown as different lines on the same plot. Since each experiment is noisy, run your algorithm 1000 times for each value of $\epsilon$ and take the average over these 1000 experiments. Note that you can compute the expected reward analytically as $\sum_a r(a)\pi(a)$, where $r(a)$ is the true reward. Remember to label each line and the axes.

2. **[8 pts]** Implement optimistic initialization and evaluate the following initial values: [0, 1, 2, 5, 10]. Plot the expected reward over 1000 time steps for each value. Your experiments with optimistic initialization should *not* use epsilon greedy exploration. You should again run your algorithm 1000 times for each initial value and take the average.

3. **[8 pts]** Implement UCB exploration. Plot the expected reward over 1000 time steps for the following values for $c$: [0, 1, 2, 5]. Your experiments should *not* use epsilon greedy exploration or optimistic initialization. You should again run your algorithm 1000 times for each value of $c$ and take the average.

4. **[8 pts]** Implement Boltzmann exploration. Plot the expected reward over 1000 time steps for the temperature: [1, 3, 10, 30, 100]. Your experiments should *not* use epsilon greedy exploration, optimistic initialization, or UCB exploration. You should again run your algorithm 1000 times for each temperature value and take the average. Please refer to the Boltzmann policy below for implementation

$$\pi = p(A_t = a) = \frac{exp(\tau \hat{q}_{a,t})}{\sum_{a' \in \mathcal{A}} exp(\tau \hat{q}_{a,t})}$$

where $\tau$ represents the temperature. You can also refer to S&B chapter 2.8 for additional information related to Boltzmann exploration.

5. [**8 pts**] Compare the exploration strategies by plotting the best-performing hyperparameter setting for each method. That is, create a single plot showing expected reward over 1000 time steps with four lines, corresponding to the best hyperparameters for $\epsilon$-greedy, optimistic initialization, UCB, and Boltzmann exploration.

6. [**4 pts**] In 2-3 sentences, explain a setting where you might *not* want to use the best-performing exploration strategy you found above.

# Feedback

**Feedback**: You can help the course staff improve the course by providing feedback. What was the most confusing part of this homework, and what would have made it less confusing?

**Time Spent**: How many hours did you spend working on this assignment? Your answer will not affect your grade.