Lab 3: Reinforcement Learning in Frozen Lake

Reinforcement Learning Course

1 Overview

The FrozenLake Environment

The `FrozenLake-v1` environment, part of the OpenAI Gym library, is a small grid-world that models a stochastic navigation problem on a frozen lake. The agent must move from the *start* position in the upper-left corner to the *goal* in the lower-right corner without falling into any holes.

Each cell of the grid can be one of the following types:

- **S** — the starting square (top-left corner);
- **F** — a frozen, safe square (ice);
- **H** — a hole in the ice (absorbing, episode ends);
- **G** — the goal square (absorbing, episode ends).

In this lab we use the standard 4×4 map, illustrated in Figure 1 below. The agent has four possible actions: `LEFT`, `DOWN`, `RIGHT`, and `UP`. If the agent moves into a hole (**H**) or the goal (**G**), the episode ends and the environment resets for the next run.

Deterministic and Stochastic Transitions

The environment can be used in two modes, controlled by the parameter `is_slippery`:

- **Non-slippery** (`is_slippery=False`): The agent moves deterministically in the chosen direction.
- **Slippery** (`is_slippery=True`): The movement is stochastic due to the ice being slippery. The agent *intends* to move in the chosen direction, but the actual motion is selected at random from a small set of possible directions:
  - With probability 1/3 the agent moves in the intended direction.
  - With probability 1/3 the agent slides 90° to the left of the intended direction.
  - With probability 1/3 the agent slides 90° to the right of the intended direction.

  The probabilities are uniform, and there is no "stay in place" option.

Thus, if the agent chooses the action `RIGHT`, the next movement could be: `UP`, `RIGHT`, or `DOWN`, each with equal probability. Similarly, choosing `UP` may result in moving `LEFT`, `UP`, or `RIGHT`. In this lab, we set `is_slippery=True`.

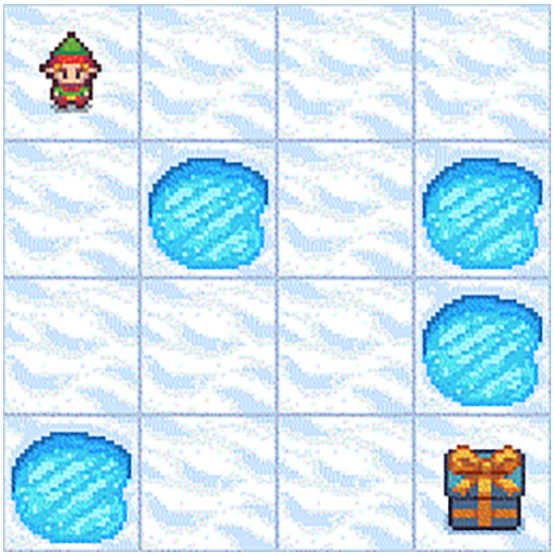

Figure 1: FrozenLake 4×4 map. Cells: **S** (start), **F** (frozen), **H** (hole), **G** (goal).

Boundary Conditions

The grid has rigid boundaries. If an intended or slipped move would take the agent outside the grid, the agent simply remains in its current cell. For example:

- If the agent is in the leftmost column and tries to move `LEFT`, it stays in place.
- If it is in the top row and tries to move `UP`, it stays in place.

This ensures all transitions are well-defined for every action in every state.

Rewards and Episodes

Rewards are very sparse:

- The agent receives a reward of 1 when it reaches the goal square (**G**).
- It receives a reward of 0 for all other transitions, including those that end in a hole.

Episodes terminate immediately upon entering a hole or the goal.

The Meaning of the Value Function and the Choice of $\gamma = 1$

In reinforcement learning, the value function $V^\pi(s)$ of a policy π is defined as the expected discounted return when starting from state s and following policy π thereafter:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \,\middle|\, S_0 = s \right],$$

where $\gamma \in [0, 1]$ is the *discount factor*. The discount factor controls how much the agent values future rewards relative to immediate ones: If γ is small (e.g., $\gamma = 0.6$), future rewards are strongly discounted. If γ is close to 1, the agent values future rewards almost equally to immediate ones.

Why We Use $\gamma = 1$?

In the FrozenLake environment, all rewards are either 0 or 1:

$$R_{t+1} = \begin{cases} 1, & \text{if the agent reaches the goal,} \\ 0, & \text{otherwise.} \end{cases}$$

Episodes always terminate when the agent either falls into a hole or reaches the goal. Thus, even if $\gamma = 1$, the total return $\sum_t \gamma^t R_{t+1}$ is bounded (it is either 0 or 1). This means setting $\gamma = 1$ is safe and well-defined.

Using $\gamma = 1$ gives the value function a very intuitive interpretation:

$$V^\pi(s) = \Pr_\pi\big(\text{eventually reaching the goal before a hole} \,\big|\, S_0 = s\big).$$

In words: *The value of a state equals the probability of successfully reaching the goal if we start from that state and follow policy π.*

This interpretation is particularly clear and pedagogically useful for understanding the prediction problem:

- It removes the abstract notion of discounted sums.
- It connects directly to a tangible outcome: "How likely am I to succeed from here?"

Policy Used in Task 1, Task 2, and Task 4

In the lab we use the **uniform-random policy**:

$$\pi(a|s) = \frac{1}{4}, \quad \text{for all actions } a \in \{\texttt{LEFT}, \texttt{DOWN}, \texttt{RIGHT}, \texttt{UP}\} \text{ and all non-terminal states } s.$$

This policy is state-independent and purely exploratory. It allows the agent to visit all states over time and serves as a neutral baseline for studying prediction algorithms (Task 1 and Task 2), and it is also reasonable behavioral policy for Q-learning (Task 4).

Task 1: Understanding the Random Policy and Computing the True Value

Objective. The purpose of this task is to understand what the random policy represents and to compute its exact value function V^π analytically. You will also develop intuition about which states have higher or lower probability of success and what it means for a policy to be proper.

Setup. We analyze the uniform-random policy in the 4×4 slippery version of FrozenLake with $\gamma = 1$. To compute V^π we represent the environment as a finite Markov chain, partitioning the states into transient (non-terminal) and absorbing (holes and goal) states. If Q is the transition matrix among transient states and r is the expected immediate reward vector, the value function for transient states satisfies:

$$(I - Q)v = r.$$

The system can be solved directly for v as $v = (I - Q)^{-1}r$, provided the matrix $(I - Q)$ is non-singular. If it is singular, the policy is not proper, meaning that some states have a non-zero probability of remaining transient forever.

Procedure.

1. Run the provided code cell that computes V^π for the random policy.
2. Inspect the resulting 4×4 heatmap showing the value of each state.
3. Reflect on how values increase toward the goal and why holes and the goal have value zero.

Be prepared to explain.

- What is the meaning of the number in the value function (e.g, at value $V^\pi(1,2)$) in terms of probability of reaching the goal? And why?
- What the random policy does in this environment and why it is proper.
- Which states have higher success probability and why.
- What it means if $(I - Q)$ is singular and how this relates to properness.

Task 2: Monte Carlo and TD(0) Prediction

Objective. In this task we approximate the same value function V^π using two prediction algorithms: Monte Carlo (MC) and Temporal-Difference learning (TD(0)). You will visualize how both methods learn over time, compare their mean-squared error (MSE) against the true values, and study the effect of different TD step-size schedules.

Algorithms.

- **Monte Carlo (First-Visit):** Estimates $V(s)$ by averaging the total return G_t observed after each state's first visit in an episode. Updates occur only at the end of each episode.
- **TD(0):** Updates $V(s)$ after each step using a bootstrapped target $r + \gamma V(s')$. This allows faster learning but introduces bias that depends on the step-size parameter.

Step-size schedules. For TD(0) we experiment with three common update rules:

$$\alpha_t(s) = \begin{cases} \alpha_0, & \text{constant step-size,} \\ \dfrac{\alpha_0}{1 + N_t(s)}, & \text{inverse schedule,} \\ \dfrac{\alpha_0}{\sqrt{1 + N_t(s)}}, & \text{inverse-square-root schedule,} \end{cases}$$

where $N_t(s)$ is the number of times state s has been visited so far.

Procedure.

1. Run the MC cell to visualize snapshots of the estimated value function after different numbers of episodes (including the initialization at episode 0). Try adjusting the number of episodes and the snapshot interval.
2. Run the TD(0) cell and change the step-size strategy by editing `SCHEDULE` and `ALPHA0`. Observe how learning speed and stability change.
3. Use the comparison cells to plot the mean-squared error versus episodes for both methods (linear and log-scale views).

Be prepared to explain.

- How MC and TD(0) differ in when and how they update their value estimates.
- Why TD(0) with a constant α can oscillate or stabilize above zero error.
- How diminishing step-size schedules affect TD(0) stability and convergence.
- Which method converges faster in early episodes and which one is more stable at the end.
- Which method would be more suitable for continuing task? (that is task that don't have clear ending or episodes)

Note. All experiments evaluate the same random policy as in Task 1, and the true value function computed previously is used as a reference for computing the MSE. Students are encouraged to experiment with different episode counts, snapshot intervals, and step-size settings to observe how learning dynamics change.

Task 3: SARSA vs. Q-learning under ε-greedy Exploration

Objective. Compare two tabular control algorithms,**SARSA** (on-policy) and **Q-learning** (off-policy). In both methods, we use ε-greedy exploration.

Algorithms. Let $Q(s, a)$ denote the action-value function. For transitions $(S_t, A_t, R_{t+1}, S_{t+1})$:

- **SARSA (on-policy)** updates toward the value of the *actual* next action A_{t+1} sampled from the same ε-greedy policy:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) \; + \; \alpha\Big[R_{t+1} + \gamma\, Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big].$$

- **Q-learning (off-policy)** updates toward the *greedy* next action (independent of the sampled A_{t+1}):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) \; + \; \alpha\Big[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)\Big].$$

On terminal transitions (entering a hole or the goal), the target reduces to R_{t+1} because the episode ends.

ε-greedy behavior (with random tie-breaking). During training, actions are selected ε-greedily from Q:

$$\pi_\varepsilon(a|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}|}, & \text{if } a \in \arg\max_{a'} Q(s, a'), \\ \frac{\varepsilon}{|\mathcal{A}|}, & \text{otherwise.} \end{cases}$$

Important: when multiple actions tie for the maximum $Q(s, \cdot)$, we break ties *uniformly at random* among the max actions (rather than always picking the first index). This prevents systematic bias and improves early exploration.

Evaluation protocol. Training is periodically paused every `log_every` episodes (e.g., every 50 episodes). At each pause, we run a batch of *greedy* evaluation episodes (typically a few hundred) and report the **success rate**: the fraction of evaluation episodes that reach the goal. To reduce plot noise, we display a **moving average** of recent success-rate measurements (averaging over a fixed-size window k), which smooths stochastic fluctuations and reveals the overall learning trend.

Setup.

- Environment: `FrozenLake-v1`, 4×4, `is_slippery=True`; rewards are 1 on reaching the goal and 0 otherwise; episodes terminate at holes or goal.
- Discount: $\gamma = 1$ (undiscounted episodic return is bounded and equals the probability of eventual success).
- Exploration: shared ε-greedy behavior policy for both algorithms (constant ε is sufficient here).
- Tie-breaking: random among arg max actions when choosing greedy actions.

Procedure.

1. Train **SARSA** and **Q-learning** for the same number of episodes using identical (α, ε).
2. Every `log_every` episodes, run greedy evaluation rollouts to estimate success rate; record these points.
3. Plot success rate vs. training episodes for both algorithms; optionally smooth with a moving average (window k).

Be prepared to explain.

- Summarize the main steps of the SARSA and Q-learning algorithms.
- Explain how SARSA and Q-learning differ. Which one is on-policy and which one is off-policy, and why?
- Discuss how the parameters ε and α influence the learning process — in particular, their effects on speed and stability for both algorithms.

Task 4: Offline Q-learning from a Random Behavioral Policy

Objective. Understand how **Q-learning** can learn an optimal action-value function *off-policy*, using experience generated by a different (non-greedy) behavioral policy. Unlike online learning, the agent here never interacts with the environment during training—it learns entirely from a fixed dataset of transitions collected beforehand.

Background. Q-learning is an **off-policy** algorithm: its update target

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)\big]$$

uses the value of the *greedy* next action, even though the data may come from a different policy. This allows the algorithm to learn about the optimal policy while following another policy. In this case, the *behavioral policy* that generated the dataset.

Behavioral policy and dataset. Before learning, we collect a dataset $\mathcal{D}$ of transitions $(s, a, r, s', \text{done})$ by acting with a simple *random behavioral policy* $\pi_b(a|s)$ that samples uniformly among all actions. This policy explores the state–action space without bias but does not aim to reach the goal efficiently.

Offline Q-learning. Once the dataset $\mathcal{D}$ is collected, learning proceeds *offline*:

1. Initialize $Q(s, a) = 0$ for all (s, a).
2. For multiple passes (epochs) over the fixed dataset, perform the standard Q-learning update for each transition in $\mathcal{D}$.
3. After each epoch, derive the current **greedy policy** $\pi_Q(s) = \arg\max_a Q(s, a)$ and compute its **true value function** V^{π_Q} using the same analytical method from Task 1.
4. Track the *start-state value* $V^{\pi_Q}(s_{\text{start}})$ across epochs to observe convergence.

Evaluation and visualization.

- Plot the **true start-state value** $V^{\pi_Q}(s_{\text{start}})$ as a function of training epoch to visualize convergence.
- Visualize the learned Q-values as a heatmap of $\max_a Q(s, a)$ (the value implied by the learned table).
- Compute and display the true V^{π_Q} of the final greedy policy as a heatmap; compare it with the analytical ground truth.

Be prepared to explain.

- What is the behavioral policy?
- Is it a good behavioral policy? More generally, what would make a good one?
- The algorithm only learns the optimal Q-table, not the optimal policy. How do we derive a policy from a Q-table, and how do we act greedily with respect to Q?
- Interpret the value heatmap for the greedy policy. Is it optimal?