# Reinforcement Learning Homework 2: Tabular Methods for Reinforcemant Learning

Mayank Dubey
*mayankd3@illinois.edu*

## I. Introduction

Reinforcement Learning (RL) is a subfield of machine learning where an agent interacts with an environment and learns based on feedback from a reward. In this homework, we focus on tabular methods of RL — where value estimates for every possible state and state-action pair are stored in a lookup table or matrix.

In this assignment, I implemented three classic model-free tabular control algorithms: First-Visit Monte Carlo (MC) control, SARSA, and Q-learning. This assignment highlights a key distinction between episodic Monte Carlo learning and temporal-difference (TD) learning. MC methods rely solely on complete returns and therefore incorporate long-term information but suffer from high variance. TD methods, like SARSA and Q-learning, bootstrap their estimates, allowing faster and more incremental learning.

These methods are implemented an environment called frozen lake. The goal is to teach an agent to navigate the lake and avoid holes. Using the gymnasium environment, not all elements that define the Markov Decision Process (MDP) will be assumed to be known; the probability transition matrix and reward functions also assumed not to be known, so the agent must learn through interactions with its environment.

The frozen lake environment has an agent that starts in the top left of a 4x4 grid map. The agent must reach the bottom right grid while avoiding the holes. The agent receives a reward of 0 when it arrives at any state, except two terminal states which are the end or falling into a hole. Falling into a hole gives a reward of -1 while reaching the end gives a reward of +1. The environment also has a is_slippery property. When true, the agent has a 1/3 chance of moving perpendicular to the intended action.

## II. Methods

As previously mentioned, three methods were explored: MC Control, SARSA, and Q Learning. These methods assume that the probability transition matrix and reward functions are not known beforehand. All three algorithms maintain a tabular action-value function $Q(s,a) \in \mathbf{R}^{|S|\times|A|}$ and improve policies using $\epsilon$-greedy policies for exploration.

### A. Monte Carlo Control (first-visit)

A Monte-Carlo Control algorithm was implemented to learn the optimal action value function. The algorithm works by updating its action value function based on the agents interactions with the environment. The algorithm rolls out multiple episodes with some arbitrary initial action value function and an $\epsilon$-greedy policy. After each episode, the action value function for each state is updated by finding the difference between the realized return and the initial expected return. Theoretically, this process would be run for an infinite number of episodes to converge onto the true optimal policy. In application, it was capped at a defined number of episodes. The algorithm implemented looks similar to the pseudo-code in figure 1.

**On-policy first-visit MC control (for $\varepsilon$-soft policies), estimates $\pi \approx \pi_*$**

Algorithm parameter: small $\varepsilon > 0$
Initialize:
  $\pi \leftarrow$ an arbitrary $\varepsilon$-soft policy
  $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
  $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
  Generate an episode following $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
      Append $G$ to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
      $A^* \leftarrow \text{argmax}_a Q(S_t, a)$ (with ties broken arbitrarily)
      For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Fig. 1. First-Visit MC Control Pseudo-code.

In my implementation, I stored the sequence of visited states, actions, and rewards, and computed the return $G_t$ for every time step by iterating backward from the end of the episode. Because this is first-visit MC, an update is performed only the first time a state-action pair appears in the episode. This helps reduces bias, but also gives large variance due to the large amount of time steps involved in one update.

The MC method is also highly sensitive to the episode length distribution. In the slippery case, episodes often become long due to random sideways movement, meaning the returns vary widely and may propagate noise into early state-action values. This explains why MC tends to learn slower and less stably than both TD methods.

### B. SARSA

SARSA is an on-policy temporal-difference method, meaning its updates are based on the actual action chosen by the $\epsilon$-greedy policy at the next time step. The way this algorithm differs is how it updates the action value function. The algorithm uses a bootstrapped TD target and updates the

action value function as the episode rolls out. This contrasts MC Control's update after the end of every episode. Figure 2 shows how the algorithm looks in more detail.

**Sarsa (on-policy TD control) for estimating $Q \approx q_*$**

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
  Loop for each step of episode:
    Take action $A$, observe $R$, $S'$
    Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until $S$ is terminal

Fig. 2. SARSA Control Pseudo-code.

The TD bootstrap allows SARSA to learn from incomplete episodes, making it much more sample-efficient than MC control. However, it uses its own estimates to update future values; this can lead to overestimation bias.

### C. Q-Learning

Q Learning is almost the same algorithm as SARSA, but with a different TD target. SARSA updates using the value of the next action actually taken under the current $\epsilon$-greedy policy (on-policy), whereas Q-learning updates toward the value of the best possible next action regardless of what was taken (off-policy). This makes Q-learning optimistic: even if the agent takes a random exploratory action, it updates using the best possible future action. As a result, Q-learning converges faster than SARSA in deterministic environments because it is not limited by its own exploratory behavior. Figure 3 shows how the algorithm looks in more detail.

**Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$**

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until $S$ is terminal

Fig. 3. Q Learning Control Pseudo-code.

During the implementation of Q learning, the same hyperparameters were used as SARSA.

All three algorithms used in this assignment share a common structure. They maintain a $Q(s, a)$ table initialized with random values, and improve policies using an $\epsilon$-greedy strategy. For all experiments, I used a discount factor of $\gamma = 0.95$ which encourages the agent to prioritize long-term reward over immediate outcomes. I also capped each episode at a maximum of 500 time steps to guarantee termination even in sequences where the agent loops among safe states.

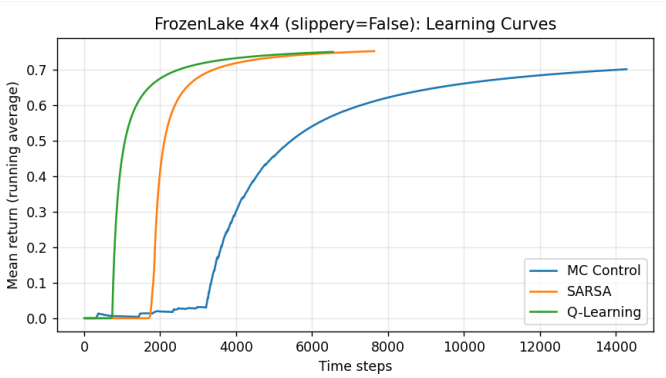


Fig. 4. Evaluation return vs. number of time steps for non-slippery environment for all three methods.

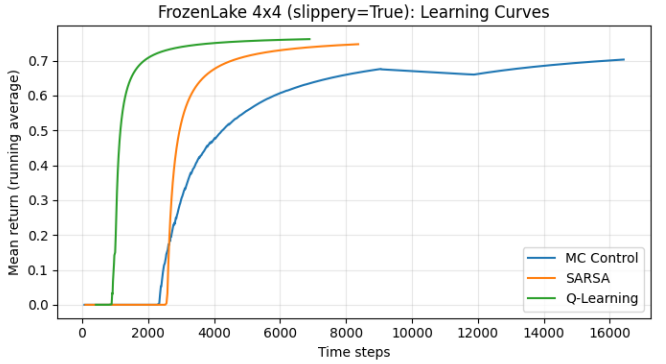


Fig. 5. Evaluation return vs. number of time steps for slippery environment for all three methods.

The exploration rate, $\epsilon$, was initialized at 0.3 in all methods. For all methods, $\epsilon$ decayed by being multiplied by 0.9 after each action, meaning the policy became greedy early in training.

## III. Results

The three algorithms were run for both cases of is_slippery=False and is_slippery=True. The evaluation return was plotted against the number of time steps. In all graphs, it can be seen that they start to converge as the time steps increases. In figures 6-12, the evaluated policies for each of the algorithms can be seen for both environments; the action space can be encoded as 0=Left, 1=Down, 2=Right, 3=Up. In figures 4 and 5, the converging training graphs can be seen.

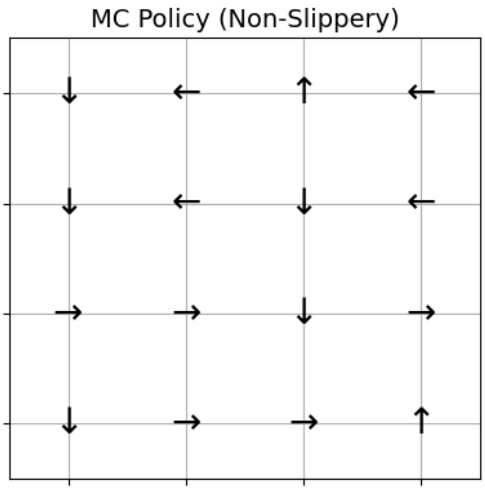


Fig. 7. MC Control Policy for Frozen Lake (no-slip

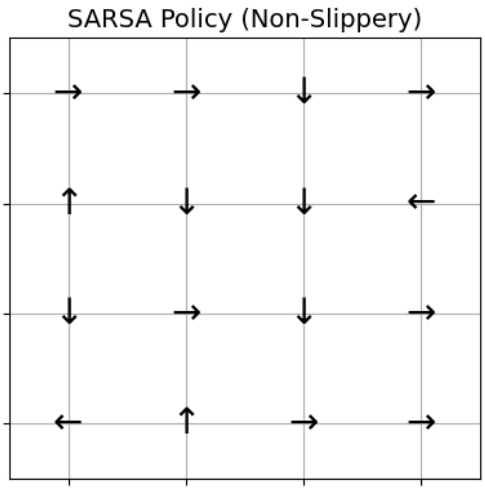


Fig. 8. SARSA Policy for Frozen Lake (no-slip

```
### FOR SLIPPERY: False
Running MC...
MC Policy: [1 0 3 0 1 0 1 0 2 2 1 2 1 2 2 3]

Running SARSA...
SARSA Policy: [2 2 1 2 3 1 1 0 1 2 1 2 0 3 2 2]

Running Q...
Q Policy: [2 2 1 0 0 2 1 2 3 2 1 2 2 2 2 0]

### FOR SLIPPERY: True
Running MC...
MC Policy: [1 0 0 2 1 3 1 3 2 2 1 0 0 2 2 1]

Running SARSA...
SARSA Policy: [2 2 1 0 1 0 1 2 3 0 1 2 3 2 2 0]

Running Q...
Q Policy: [2 2 1 0 1 3 1 1 3 2 1 1 3 1 2 3]
```

Fig. 6. Policies evaluated for slippery and not slippery environment for all 3 algorithms.

Across both environments, the evaluation return curves show clear differences in sample efficiency and stability among the three methods. In the non-slippery environment, Q-learning rapidly converges to near-optimal performance within the first few thousand time steps. SARSA converges slightly slower, as expected, because the on-policy updates incorporate exploratory moves that sometimes pull value estimates downward. Monte Carlo shows the slowest improvement because each update is based on full-episode returns. The graphs also show a slight difference in converged mean return values. This can be indicitave of the bias TD methods like SARSA and Q-learning have compared to unbiased methods like MC Control.

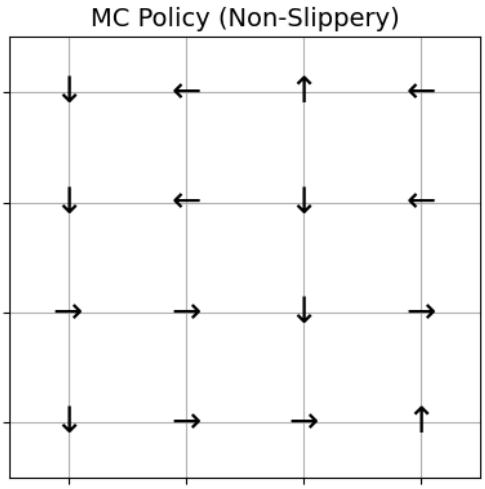


Fig. 9. MC Control Policy for Frozen Lake (no-slip

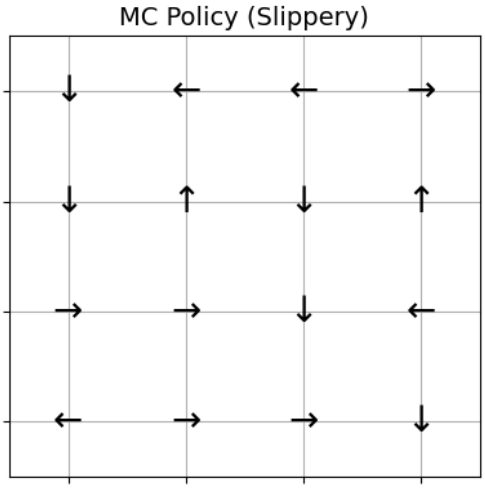


Fig. 10. MC Control Policy for Frozen Lake (slip

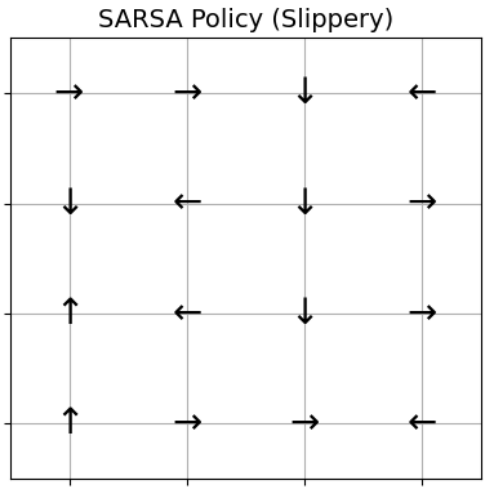


Fig. 11. SARSA Policy for Frozen Lake (slip

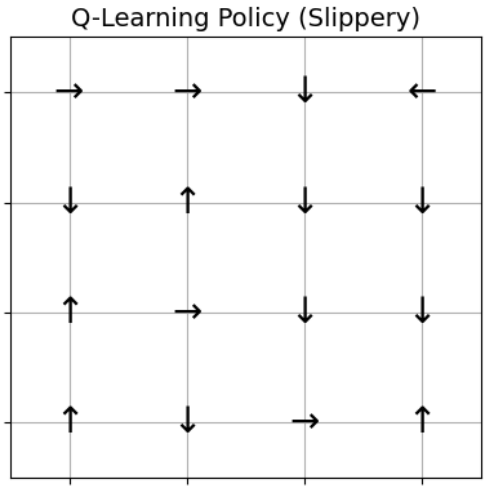


Fig. 12. Q-Learning Policy for Frozen Lake (slip

In the slippery environment, both SARSA and Q-learning initially rise quickly. SARSA's on-policy nature makes it more cautious and slightly more stable. MC control once again shows as longer episodes and slower convergence. The graphs also show the same slight difference in converged mean return values — indicitave of the bias inherent in TD methods. All policies appear to struggle more in the slippery environment. This is expected due to the stochastic nature of the environment.

The policy visualizations further highlight these differences. In the non-slippery grid, TD algorithms eventually converge toward similar goal-directed paths. MC Control seemed to vary more in the learned policies. This is a clearer distinction in how the bias in TD learning affects the learned policies. In the slippery environment, SARSA typically chooses routes that minimize exposure to holes, whereas Q-learning may prefer shorter but riskier paths because it evaluates future actions assuming optimal behavior.

## IV. Conclusion

The comparison of Monte Carlo control, SARSA, and Q-learning highlights the key tradeoffs between episodic and bootstrap-based updating. MC control, while conceptually simpler, struggles in environments with sparse rewards and stochastic transitions due to the high variance of full-episode returns. Because MC only updates after an episode finishes, it reacts slowly to new information and requires many more samples to propagate reward back through the state space.

SARSA offers more stable learning through TD updates and performs well in unknown environments because its on-policy nature accounts for exploratory moves. This produces behavior that is generally safer and more consistent, but also slightly more conservative. Although SARSA converges more quickly than MC control, it still tends to lag behind Q-learning because it evaluates future actions using the policy's own exploration, which limits the optimism in its updates.

Q-learning, on the other hand, is highly sample-efficient and consistently converged the fastest in both the slippery and non-slippery versions of the environment. Its off-policy update rule, which backs up the value of the greedy next action regardless of the behavior policy, allows it to learn optimal Q-values more aggressively. In practice, this meant Q-learning rapidly outperformed both SARSA and MC control in terms of evaluation return, even under the stochastic slippery setting.

Overall, both TD methods, SARSA and Q-learning, demonstrate much greater sample efficiency than Monte Carlo control. However, the inherent bias introduced by bootstrapping leads to different behaviors: SARSA learns policies that reflect its exploratory actions, while Q-learning learns more optimistic policies that assume greedy future behavior. In this assignment, these biases favored Q-learning, which consistently produced the best-performing policies across all experimental conditions.

## References

[1] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed., in progress. Cambridge, MA, USA: MIT Press, 2014–2015.