

Technical Bottleneck Analysis: Q-Learning Algorithm (FrozenLake)

Overview: The Q-learning implementation provided in the GitHub repository (<https://github.com/ronanmmurphy/Q-Learning-Algorithm>) is a tabular reinforcement learning solution applied to the FrozenLake environment, which is a discrete, low-dimensional gridworld.

This document identifies the primary technical bottlenecks in the code, with a focus on runtime performance, resource constraints, and algorithmic limitations.

Identifying Computational Bottlenecks

The code implements Q-learning for the FrozenLake environment using a deterministic 4x4 grid. The primary computational bottlenecks include: [GitHub+1Japhari Mbaru+1Japhari Mbaru+1Chegg+1](#)

- **Q-Table Updates:** The Q-value update formula involves accessing and modifying the Q-table for each state-action pair, which can be computationally intensive as the number of episodes increases. [GitHub+1Japhari Mbaru+1](#)
 - **Exploration Strategy:** The epsilon-greedy strategy requires generating random numbers to decide between exploration and exploitation, adding overhead during each decision-making process. [GitHub](#)
 - **Reward Processing:** Determining rewards and updating the Q-table based on the agent's actions and resulting states can introduce delays, especially if the environment is complex. [GitHub](#)
-

2. Evaluation of Suggestions

The suggestions to optimize the code are reasonable and align with common practices in reinforcement learning:

- **Optimizing Q-Table Access:** Implementing more efficient data structures or parallel processing techniques can reduce the time spent on Q-table updates.
- **Adjusting Exploration Strategy:** Dynamically adjusting the exploration rate (epsilon) can balance exploration and exploitation more effectively, potentially improving learning

efficiency.[GitHub+1Japhari Mbaru+1](#)




- **Reward Processing Optimization:** Caching rewards or precomputing certain values can minimize delays during reward processing.[Japhari Mbaru](#)

These suggestions are practical and can lead to performance improvements in the Q-learning algorithm.

3. Hardware Implementation Proposal

To address the Q-table update bottleneck, a hardware implementation can be proposed:

- **Hardware Design:** Develop a custom hardware module that can perform Q-value updates in parallel for multiple state-action pairs. This module would interface with the main processor to receive state-action pairs and rewards, compute the updated Q-values, and store them back in memory.[GitHub](#)
- **Benefits:** This hardware acceleration would reduce the time spent on Q-table updates, allowing the software to focus on other aspects of the learning process.

 Change	 Buggy Version	 Fixed Version
1. is_slippery setting	Default (<code>True</code>)	Explicitly set <code>is_slippery=False</code>
2. env.step() unpacking	Sometimes incorrect unpacking	Correct: <code>state, reward, terminated, truncated, _ = env.step()</code>
3. done flag	Sometimes missing or misused	<code>done = terminated or truncated</code> — fully handles Gymnasium's new API
4. Learning update	May not update if <code>done=True</code> too early	Ensures proper Q-learning update at each step
5. Exploration decay	Possibly too aggressive or not present	Smooth decay: <code>epsilon = max(min_epsilon, epsilon * decay)</code>
6. Reward tracking	Appended wrong/zero values	Appends actual <code>total_reward</code> from episode

7. Moving average plot

Flat line at 0

Accurate moving average using `np.convolve()`

8. Clear terminal output

None or too verbose

Prints progress every 500 episodes for clarity

Why `is_slippery=False`?

In FrozenLake-v1, the default behavior has **stochastic transitions** (`slippery = True`). That means:

- Even if you choose to go **Right**, it might randomly go **Down** instead.
- For beginners and learning algorithms, it **makes convergence much harder**.

So setting `is_slippery=False` makes the environment deterministic and easier to learn. It's like switching to "training wheels" — great for verifying your algorithm is correct.

Q-Learning Algorithm in SystemVerilog (SV)

In the code I provided, **Q-learning** is implemented in **SystemVerilog (SV)**. Here's a breakdown of the implementation:

1. **Q-table**: The Q-table is implemented as a 2D array in SystemVerilog, where rows represent states, and columns represent actions. The Q-values for each state-action pair are stored in this table.
2. **Update Logic**: The **Q-learning update rule** is implemented inside an `always_ff` block, which ensures that the Q-values are updated on every clock cycle. The Q-value is updated based on the current reward and the maximum Q-value of the next state.
3. **State-Action Update**:
 - For each state-action pair, the current Q-value is fetched from the Q-table.

- The algorithm finds the maximum Q-value for the next state (after taking an action).
 - The Q-value is updated according to the formula mentioned above using a learning rate (**alpha**) and discount factor (**gamma**).
4. **Reset Mechanism:** The Q-table and other registers are reset when the **reset** signal is asserted, initializing all Q-values to zero.

Algorithm Used in SystemVerilog

In the SystemVerilog code, I implemented the **Q-learning algorithm** using a **tabular method** to store Q-values in a 2D array. The update rule is applied to modify the Q-values iteratively as the agent interacts with the environment.

1. **Q-table Initialization:** At the beginning of the simulation, the Q-table is initialized to zeros, representing that the agent has no prior knowledge of the environment.
2. **State-Action Updates:** After each action, the Q-values for the state-action pair are updated using the learning rule, and the agent's experience is stored in the Q-table.
3. **Policy Generation:** Although the code doesn't explicitly generate a policy (as this would be done during the exploration phase of Q-learning), it would be derived by selecting the action with the highest Q-value for each state after the learning process is complete.

Summary

- **Q-Learning** is a **reinforcement learning** algorithm used to find the optimal policy by iteratively updating Q-values using the **Bellman equation**.
- The **Frozen Lake problem** is a simple reinforcement learning task where the agent has to navigate a grid with obstacles to reach a goal, with rewards and penalties.
- In the **SystemVerilog implementation**, I used a **Q-table** (2D array) to store Q-values and applied the **Q-learning update rule** in a hardware description language to simulate the learning process in a time-driven manner.

In hardware, **Q-learning** is implemented as a clock-driven, iterative process where the **Q-table** is updated after each action taken by the agent. The **Q-values** are stored in memory and updated using arithmetic and comparison logic to reflect the expected future rewards. The hardware-based system must handle state transitions, rewards, and Q-value updates in real time, leveraging control logic, memory access, and computational resources in a time-driven manner.