EARIN LABORATORY 6
VARIANT 5: FROZEN LAKE

AKSA FATIMA, 317740

Introduction

FrozenLake Problem

The FrozenLake problem is a classic grid-world scenario where an agent aims to cross a frozen lake from a start point to a goal without falling into any holes. Due to the lake's slippery nature, the agent may not always move in the intended direction, introducing an element of uncertainty and challenge.

Environment Setup

The agent starts at the top-left corner, position `[0, 0]`, and must navigate to the goal, located at the bottom-right corner, position `[7, 7]`, in the 8x8 FrozenLake environment. Along the way, there are randomly distributed holes that can lead to failure if the agent steps into them.

Slippery Surface

In a typical setup, the lake is slippery (`is_slippery=True`), which means the agent's movement may not align perfectly with the chosen action. Instead, there's a chance the agent will slide in an unintended direction, which requires planning and adaptability.

Action Space and Movement

The agent's action space consists of four possible moves:

- 0: Move left
- 1: Move down
- 2: Move right
- 3: Move up

These actions allow the agent to navigate the lake in search of the goal.

Rewards and Episode Termination

The reward structure is simple:

- A reward of 1 is given when the agent reaches the goal.
- No reward is given if the agent falls into a hole or remains on the ice.

And finally, each episode ends under the following conditions:

1. Termination:

   a) The agent falls into a hole.
   b) The agent reaches the goal at [7, 7].

2. Truncation:

a) The episode exceeds a predefined length (e.g., 200 steps for an 8x8 environment, 100 steps for a 4x4 environment).

The environment, with its randomness and slippery nature, makes it a suitable challenge for reinforcement learning experiments and requires the agent to balance exploration and exploitation while navigating the frozen lake.

Q-learning Algorithm

A reinforcement learning algorithm based on value functions, the Q-learning algorithm involves estimating the expected reward for taking a specific action in each state. Represented as Q(s,a), this value indicates the expected benefit when an agent, in state (s) takes action (a). The environment provides feedback in the form of rewards depending on the agent's actions. The algorithm's core concept revolves around maintaining a Q-table, where states represent rows, and actions represent columns, with the table storing the Q-values. By updating the Q-table based on the rewards received from each action, the agent can determine which action yields the highest expected reward, thus guiding its behavior towards optimal outcomes.

Q-learning is classified as an off-policy algorithm because it uses the maximum operation to compute the expected value for the next state, selecting the best possible action based on the Q-table. However, the policy used to generate actions and collect experience does not always follow the optimal path. In other words, the policy that drives the agent's behavior during learning may differ from the policy derived from the Q-table. This difference between learning and behavior policies is what defines off-policy learning, allowing flexibility in exploration strategies while learning an optimal policy through the Q-table.

Details of the code:

A) Initialization of Q-table

The Q-table is a two-dimensional array with dimensions **(state_no, action_no)**, where **state_no** is the number of states in the environment, and **action_no** is the number of possible actions. The initialization occurs at the beginning of the code, creating a table where each state-action pair starts with a Q-value of zero. This is done to ensure the agent learns from scratch by updating the Q-table based on its experiences during training.

```
self.q_table = np.zeros((state_no, action_no))  # Initialize Q-table with zeros
```

B) Choosing Actions

The **choose_action** method selects actions for the agent to take during training. It uses an exploration-exploitation strategy to balance trying new actions and leveraging known high-value actions. If the exploration rate (epsilon) is greater than a random value between 0 and 1, the agent chooses a random action (exploration). Otherwise, it selects the action with the highest Q-value for the current state (exploitation).

```
if rng.random() < epsilon:
    action = env.action_space.sample()  # Random action (exploration)
else:
    action = np.argmax(self.q_table[state, :])  # Best-known action (exploitation)
```

C) Updating the Q-table

The **update_q_**table method updates the Q-table after the agent takes an action and observes the reward and next state. It uses the Q-learning update rule to adjust the Q-value for the current state-action pair, accounting for the reward and the maximum Q-value in the next state. This update reflects the agent's learning as it experiences the environment.

```
current_q = self.q_table[state, action]  # Current Q-value
max_q = np.max(self.q_table[next_state, :])  # Max Q-value for next state
new_q = (1- self.learning_rate) * current_q + self.learning_rate * (
    reward + self.discount_factor * max_q)  # Q-learning update rule
self.q_table[state, action] = new_q  # Update Q-table with new Q-value
```

D) Calculating the Reward

The reward structure in the FrozenLake environment defines the feedback given to the agent based on its actions. A positive reward is typically given for reaching the goal, while no reward or negative rewards occur for other outcomes (like falling into a hole). The Q-values are updated based on the observed reward and the state transition.

E) Training Loop

The training loop runs for a specified number of episodes, where the agent interacts with the environment, selects actions, and updates the Q-table accordingly. This loop controls the overall training process, managing the exploration rate (which decays over time), and records data for analysis.

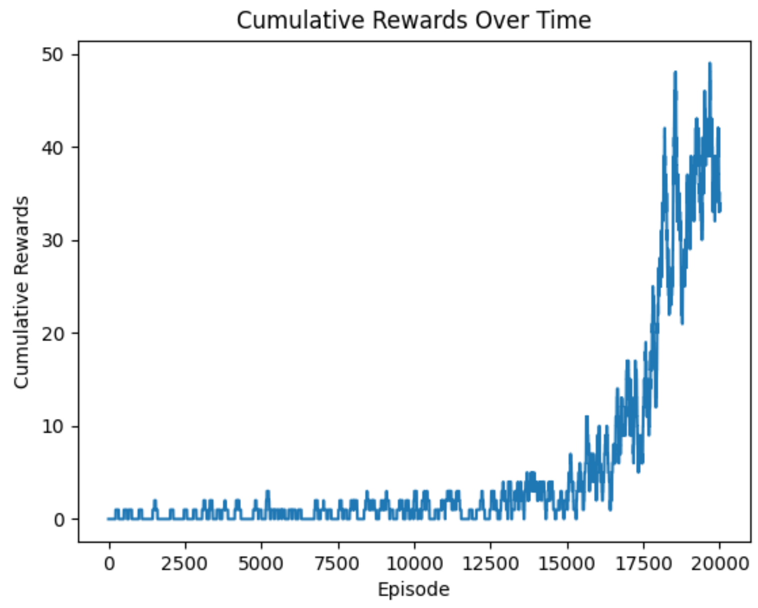
```
def train(env, num_episodes):
    rewards_per_episode = []  # To record rewards
    for episode in range(num_episodes):
        state = env.reset()[0]
        terminated = False
        total_reward = 0
        while not terminated:
            action = choose_action(state)  # Choose an action
            new_state, reward, terminated, _ = env.step(action)  # Take the action
            update_q_table(state, action, reward, new_state)  # Update the Q-table
            total_reward += reward  # Accumulate reward
            state = new_state  # Transition to the new state

        # Decay epsilon (exploration rate) after each episode
        epsilon = max(epsilon- epsilon_decay_rate, 0)
        rewards_per_episode.append(total_reward)  # Record total reward for the episode
```

F) Analysis and Testing

After training, the results (episode lengths, rewards, etc.) are returned for analysis. These results help evaluate the performance of the agent during training and guide further adjustments to improve learning. The code also allows for testing the trained Q-values to assess the agent's behavior under the learned policy.

```
# Return the rewards for each episode
return rewards_per_episode
```

Final Output:



From the output, we can observe the following trends:

- **Epsilon Decay:** The effect of epsilon (exploration rate) decay is visible. Initially high, allowing for broad exploration, epsilon is reduced over time, shifting the balance towards exploitation of learned strategies. The graph reflects this shift with increased reward accumulation as the agent begins to leverage the most rewarding actions learned.
- **Convergence:** The increasing trend towards the latter episodes could suggest approaching convergence towards an optimal policy, where the agent has effectively learned to navigate the environment with minimal error.
- **Learning Rate and Discount Factor: The** steepness of the reward increase might also be influenced by the relatively high learning rate (0.99) and discount factor (0.99), which emphasize the importance of future rewards and rapid integration of new information into the agent's decision-making process.

In conclusion, we can say that the graph effectively visualizes the learning and optimization process of a Q-learning algorithm, demonstrating initial exploration, gradual learning, and eventual exploitation of a learned policy in a complex environment.