Week-3 Challenges

Challenge -10

1. **Identifying Computational Bottlenecks in FrozenLake Q-Learning Code**

The GitHub mainly contains a Python file that does **tabular Q-learning** on the FrozenLake environment Initialize Q-table

- For each episode:
    - Reset environment
    - For each step:
        - Choose action (epsilon-greedy)
        - Take action, observe reward and next state
        - **Update Q-table**
        - If done, break
- After training, evaluate

The most computationally heavy operation is the Q-table update:

python

CopyEdit

```
Q[state, action] = Q[state, action] + alpha * (reward + gamma * np.max(Q[new_state, :]) - Q[state, action])
```

This line does:

- Fetch current Q-value for (state, action)
- Fetch the **maximum Q-value** for the new state (across all actions)
- Multiply and add terms
- Write back to the Q-table

Main bottlenecks are:

- **Finding np.max(Q[new_state, :])** → needs scanning over all possible actions
- **Q-table read-modify-write** → needs memory access, read + write

Other parts like environment step and random action choice are lightweight compared to this update.

2. Do the suggestions make sense?

Yes!

It makes sense: **Q-learning is memory bound and compute bound** mainly at the Q-table access and max computation step.

For larger problems (bigger state/action spaces), the **max operation** becomes a clear bottleneck.

3. Propose a Hardware Implementation for the Biggest Bottleneck

Since the **biggest bottleneck is finding np.max(Q[new_state, :])**, the hardware should:

- **Take all action Q-values for a given state**
- **Find the maximum value and its corresponding action efficiently**
- **Support parallel comparisons** (rather than sequentially scanning)

Thus, the hardware should implement a **parallel max-finder** (like a reduction tree).

Basic structure:

- Inputs: Q-values for all actions (say 4 actions for FrozenLake)
- Logic: Comparator tree to find maximum value
- Output: Maximum Q-value and corresponding action index

4. Generate SystemVerilog Code for the Hardware Implementation

Now, let's **generate SystemVerilog** for a **4-input max-finder** (for FrozenLake 4 actions) using a simple comparator tree:

Qlearning Hw Sv

```
99
1
2
3
4
5
6
7
8
9
```

```
// MaxFinder module: finds max Q-value and corresponding action index
module MaxFinder #(parameter WIDTH = 16) (
input logic [WIDTH-1:0] q0, q1, q2, q3,
output logic [WIDTH-1:0] max_q,
output logic [1:0] max_action
);

logic [WIDTH-1:0] max0, max1;
logic [1:0] action0, action1;

// First level comparisons
always_comb begin
if (q0 >= q1) begin
max0 = q0;
action0 = 2'd0;
end else begin
max0 = q1;
action0 = 2'd1;
end

if (q2 >= q3) begin
max1 = q2;
action1 = 2'd2;
end else begin
max1 = q3;
action1 = 2'd3;
end
end

// Second level comparison
always_comb begin
if (max0 >= max1) begin
max_q = max0;
max_action = action0;
end else begin
max_q = max1;
```

Challenge -11

Optimizing FrozenLake Q-Learning for GPU
First, some important facts:

- The original code is pure Python + NumPy.
- It's running a tabular Q-learning (Q-table is relatively small, just an array).
- GPU acceleration makes sense if you batch the episodes or steps and do updates in parallel.

```
import gym
import cupy as cp  # Using CuPy instead of NumPy
import numpy as np

# Initialize environment
env = gym.make('FrozenLake-v1', is_slippery=False)
state_space_size = env.observation_space.n
action_space_size = env.action_space.n

# Hyperparameters
alpha = 0.8
gamma = 0.95
epsilon = 1.0
epsilon_decay = 0.995
epsilon_min = 0.01
num_episodes = 5000

# Q-table initialized on GPU
Q = cp.zeros((state_space_size, action_space_size))

for episode in range(num_episodes):
    state = env.reset()[0]
    done = False

    while not done:
        if np.random.rand() < epsilon:
            action = env.action_space.sample()
        else:
            action = int(cp.argmax(Q[state, :]).get())

        next_state, reward, done, truncated, info = env.step(action)

        # GPU update
        best_next_action = cp.max(Q[next_state, :])
        Q[state, action] = Q[state, action] + alpha * (reward + gamma * best_next_action - Q[state, action])

        state = next_state

    # Decay epsilon
    if epsilon > epsilon_min:
        epsilon *= epsilon_decay

# Move Q-table back to CPU if needed
    Q_final = cp.asnumpy(Q)
```