# **Challenge #10: Identifying Computational Bottlenecks in Q-Learning (FrozenLake)**

## **Objective**

This challenge investigates the computational inefficiencies of a Q-learning implementation using OpenAI Gym’s FrozenLake-v0 environment. We use an LLM (ChatGPT-4) to:

1. Identify computational bottlenecks in the source code
2. Evaluate the accuracy of the LLM’s insights
3. Propose a hardware (HW) implementation of the most significant bottleneck
4. Generate synthesizable SystemVerilog code for that HW solution

## **Step 1: Identifying Bottlenecks**

Code Source:<https://github.com/ronanmmurphy/Q-Learning-Algorithm>

File Analyzed: q\_learning\_frozen\_lake.py

### Key Bottlenecks Identified

| Bottleneck | Description |
| --- | --- |
| Nested Loops | Episode and step loops (num\_episodes × max\_steps) run serially, limiting scalability |
| Epsilon-Greedy Action Selection | Frequent calls to random.uniform and np.argmax per step introduce latency |
| Q-Table Update | Arithmetic operation at every step, including np.max() lookup and floating-point math |
| No Vectorization | The Q-table update is fully scalar; no batching or GPU acceleration used |

## **Step 2: Evaluation of LLM’s Suggestions**

### **Reasoning Accuracy:**

* + The LLM's identification of bottlenecks aligns well with real-world RL performance issues, especially for environments with:
    - Small state spaces but long training loops
    - Floating-point Q-value updates

### **Depth of Insight:**

* + The LLM not only highlighted the loops and Q-value update equation but also pointed out:
    - Redundancy in np.max(q\_table[new\_state, :])
    - The potential gains from pipelining or caching

**Verdict: Accurate, meaningful, and practically useful.**

**Step 3: Proposed Hardware Implementation**

### Focus

The Q-table update equation was identified as the most compute-heavy operation:

### 

### Hardware Accelerator (Summary)

| Component | Description |
| --- | --- |
| Q-table Memory | Dual-port BRAM storing 16×4 entries |
| Max Finder | Combinational logic to compute max(Q[s’,:]) |
| Fixed-Point ALU | For multiply-add logic (α, γ) |
| Controller FSM | Manages sequencing of read → compute → write |

## 

## **Step 4: Generated SystemVerilog Code**

module q\_update\_unit #(

parameter STATE\_BITS = 4,

parameter ACTION\_BITS = 2,

parameter DATA\_WIDTH = 16 // Q4.12 fixed-point

)(

input logic clk,

input logic rst,

input logic start,

input logic [STATE\_BITS-1:0] state,

input logic [ACTION\_BITS-1:0] action,

input logic [STATE\_BITS-1:0] new\_state,

input logic signed [DATA\_WIDTH-1:0] reward,

input logic signed [DATA\_WIDTH-1:0] alpha,

input logic signed [DATA\_WIDTH-1:0] gamma,

output logic done

);

logic signed [DATA\_WIDTH-1:0] q\_table [0:63];

logic signed [DATA\_WIDTH-1:0] q\_sa;

logic signed [DATA\_WIDTH-1:0] q\_next [0:3];

logic signed [DATA\_WIDTH-1:0] max\_q\_next;

logic signed [DATA\_WIDTH-1:0] target, delta, update, new\_q;

typedef enum logic [2:0] { IDLE, READ\_Q, COMPUTE\_MAX, CALC\_DELTA, WRITE\_BACK, DONE } state\_t;

state\_t current\_state, next\_state;

always\_ff @(posedge clk or posedge rst) begin

if (rst) current\_state <= IDLE;

else current\_state <= next\_state;

end

always\_comb begin

case (current\_state)

IDLE: next\_state = start ? READ\_Q : IDLE;

READ\_Q: next\_state = COMPUTE\_MAX;

COMPUTE\_MAX: next\_state = CALC\_DELTA;

CALC\_DELTA: next\_state = WRITE\_BACK;

WRITE\_BACK: next\_state = DONE;

DONE: next\_state = IDLE;

endcase

end

always\_ff @(posedge clk) begin

if (rst) done <= 0;

else case (current\_state)

READ\_Q: begin

q\_sa <= q\_table[{state, action}];

q\_next[0] <= q\_table[{new\_state, 2'b00}];

q\_next[1] <= q\_table[{new\_state, 2'b01}];

q\_next[2] <= q\_table[{new\_state, 2'b10}];

q\_next[3] <= q\_table[{new\_state, 2'b11}];

end

COMPUTE\_MAX: begin

max\_q\_next <= q\_next[0];

if (q\_next[1] > max\_q\_next) max\_q\_next <= q\_next[1];

if (q\_next[2] > max\_q\_next) max\_q\_next <= q\_next[2];

if (q\_next[3] > max\_q\_next) max\_q\_next <= q\_next[3];

end

CALC\_DELTA: begin

target <= reward + ((gamma \* max\_q\_next) >>> 12);

delta <= target - q\_sa;

update <= (alpha \* delta) >>> 12;

new\_q <= q\_sa + update;

end

WRITE\_BACK: begin

q\_table[{state, action}] <= new\_q;

end

DONE: done <= 1;

default: done <= 0;

endcase

end

endmodule

## **Conclusion**

This challenge successfully demonstrated how a large language model (LLM) can:

* Accurately profile reinforcement learning code
* Highlight real computational inefficiencies
* Propose and synthesize an RTL-level hardware module to address the bottleneck

The final SystemVerilog module is suitable for FPGA acceleration of the Q-learning training loop and can be extended to larger environments or vectorized action/state spaces.