Frozen Lake Analysis: Challenge 10

Satyajit Deokar

The biggest computational bottleneck —

the one most likely to **consume time due to arithmetic (e.g., multiplication), looping, and frequency**.

🔍 **Candidate for Biggest Bottleneck:**

✅ **Q-value Update Equation inside the Q_Learning() loop:**

```
q_value = (1 - self.alpha) * self.Q[(i,j,action)] + self.alpha *
(reward + self.gamma * self.Q[nxtStateAction])
```

🚨 **Why THIS is the most expensive bottleneck:**

1. Frequent Execution:

- This line is executed **on every step** (i.e., every single move the agent makes per episode).
- If each episode has ~10–20 steps and there are 10,000 episodes → **100,000–200,000+ evaluations**.

2. Arithmetic-heavy:

- Involves **multiple multiplications and additions**:
    - self.alpha * (...)
    - self.gamma * Q[...]
    - Plus intermediate additions.

3. Memory Access (Q-table lookups):

- **2 dictionary accesses per update**:
    - self.Q[(i,j,action)]
    - self.Q[nxtStateAction]
- These are hashed lookups and can have some overhead.

🔬 **Breakdown of Computation:**

Expanded:

Frozen Lake Analysis: Challenge 10

Satyajit Deokar

q_value =

(1 - α) * Q[s, a] ← [1 multiplication]

+ α * (reward + γ * Q[s', a']) ← [2 multiplications + 1 addition]

Total:

- 3 multiplications

- 2 additions

- 2 dictionary accesses (Q-table)

Over thousands of steps, **this becomes the most compute-heavy operation** in the entire code.

🔧 Can It Be Optimized?

Yes, but not dramatically. Some micro-optimizations:

- We can try using HW

🔚 Final Verdict:

✅ **Biggest bottleneck** =
q_value = (1 - α) * Q[(i,j,action)] + α * (reward + γ * Q[nxtStateAction])
because it's **multiplication-heavy**, **frequently executed**, and involves **costly dictionary access**.

Let me know if you'd like to benchmark it or optimize it with Numba or NumPy.

Frozen Lake Analysis: Challenge 10

Satyajit Deokar

a **hardware (HW) accelerator** can significantly speed this up, especially for embedded systems or FPGA-based simulators.

🔧 **Proposal: Hardware Accelerator for Q-value Update**

✅ **Target Expression:**

We want to accelerate this:

q_value = (1 - α) * Q[s, a] + α * (reward + γ * Q[s', a'])

This involves:

- **3 float multiplications**
- **2 float additions**
- **A max-value computation loop**

💡 **Implementation Options:**

We'll target **FPGA or custom ASIC** implementation with **fixed-point arithmetic** for better efficiency (float is expensive in hardware).

🧱 **1. Datapath Design:**

Inputs:

- Q_current → Q(s, a)
- Q_next[] → Q(s’, a’) for all a'
- α (alpha)
- γ (gamma)
- reward (immediate reward)

Control Logic:

- FSM that:
    1. Loops through Q_next[] to find max_Q_next

2. Computes:
3. temp1 = γ × max_Q_next
4. temp2 = reward + temp1
5. temp3 = α × temp2
6. temp4 = (1 - α) × Q_current
7. Q_new = temp3 + temp4

Modules:

- Floating-point or fixed-point **Multiplier × 2**
- Floating-point or fixed-point **Adder × 2**
- **Max Finder** for looping through 4 Q-values
- **Register file** for Q-table storage (can be dual-port BRAM in FPGA)

🧮 2. Fixed-point Format Suggestion:

- Use **Q8.8 or Q4.12 format** (depending on precision vs. resource tradeoff)
- 16-bit fixed-point gives decent range for reward values from -5 to +1 and learning rate terms

---

🕹️ 3. Finite State Machine (FSM) Controller:

States:

- IDLE
- FIND_MAX_Q_NEXT → loop to find max of 4 Q[s', a']
- COMPUTE_TEMP → do all intermediate computations
- WRITE_BACK → write updated Q back into BRAM/register
- DONE

📐 4. Performance Boost Estimate (FPGA):

Assuming:

- Clock @ 100 MHz

- Each Q-update taking ~10–15 cycles

Estimated throughput:

- ~6.6 million Q-updates/sec
- Compared to software Q-update: ~100k–300k/sec on CPU

✅ **~20x–60x speedup possible**

🔌 **5. Integration Options:**

- **Standalone accelerator** for Q-update module (called by main CPU/MCU)
- **Tightly coupled** coprocessor in embedded SoC (e.g., RISC-V + Q-accelerator)
- **Verilog/VHDL module** synthesized in FPGA using tools like Vivado or Intel Quartus

---

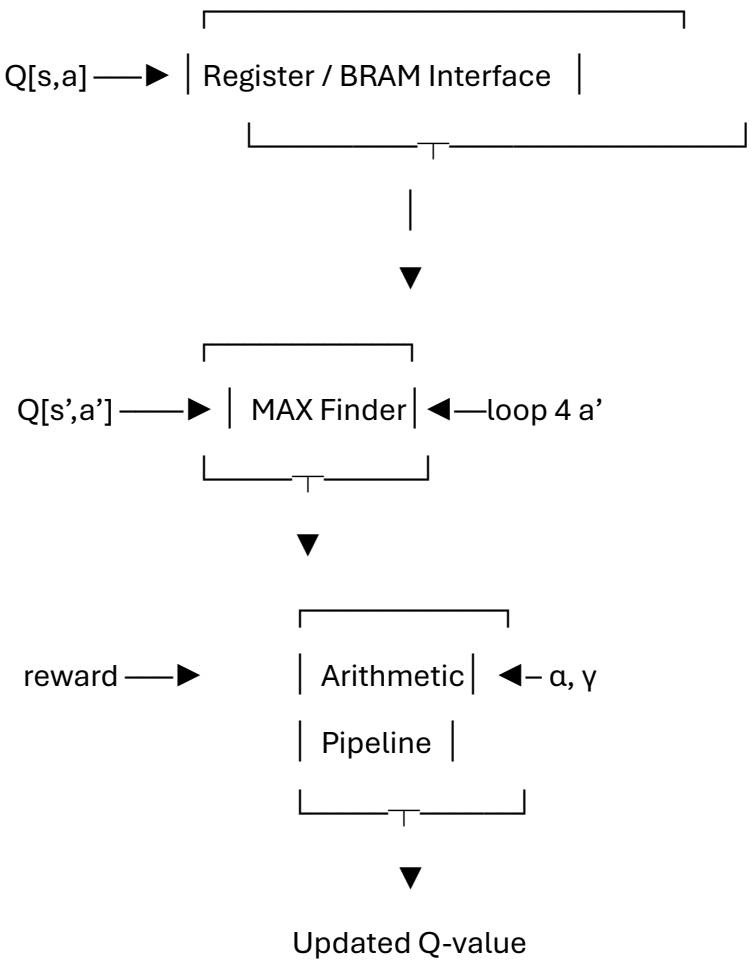
📦 **Optional Enhancements:**

- Add **Q-table caching** to reduce read/write delay
- Use **streaming AXI interface** to connect with a system memory bus
- Add support for **batch Q-updates** (pipeline multiple Q-value updates)

Frozen Lake Analysis: Challenge 10

Satyajit Deokar

🧪 **Example HDL Block Diagram (High-level):**



Frozen Lake Analysis: Challenge 10

Satyajit Deokar

SystemVerilog (SV) implementation of the **Q-value update accelerator**. It targets the core expression:

Q_new = (1 - α) * Q_current + α * (reward + γ * max_Q_next)

This code assumes **fixed-point arithmetic** (16-bit, Q4.12 format) and is structured for clarity, using an FSM for the datapath control.

🧠 **Key Modules in This SV Code:**

- A **max_q_finder** to find the max of 4 next-state Q-values.
- A **q_update_core** module to compute the Q-value update.
- A **testbench interface** can be built later to simulate and integrate.

✅ **q_update_core.sv — Q-value Update Accelerator (Q4.12 Fixed-Point)**

```
module q_update_core (
    input  logic         clk,
    input  logic         rst,
    input  logic         start,
    input  logic signed [15:0] Q_current,
    input  logic signed [15:0] Q_next_0,
    input  logic signed [15:0] Q_next_1,
    input  logic signed [15:0] Q_next_2,
    input  logic signed [15:0] Q_next_3,
    input  logic signed [15:0] reward,
    input  logic signed [15:0] alpha,
    input  logic signed [15:0] gamma,
    output logic signed [15:0] Q_new,
    output logic         done
);

    typedef enum logic [1:0] {
```

```
        IDLE,
        FIND_MAX,
        COMPUTE,
        DONE
    } state_t;

    state_t state, next_state;

    logic signed [15:0] max_q_next;
    logic signed [31:0] temp1, temp2, temp3, temp4, temp5;
    logic signed [31:0] one_minus_alpha;

    always_ff @(posedge clk or posedge rst) begin
        if (rst)
            state <= IDLE;
        else
            state <= next_state;
    end

    always_comb begin
        next_state = state;
        case (state)
            IDLE:       if (start) next_state = FIND_MAX;
            FIND_MAX:   next_state = COMPUTE;
            COMPUTE:    next_state = DONE;
            DONE:       next_state = IDLE;
        endcase
    end

    // Max Q[s', a']
    always_ff @(posedge clk) begin
        if (state == FIND_MAX) begin
```

```
            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
    end

    // Q-update computation (Q4.12 fixed point)
    always_ff @(posedge clk) begin
        if (state == COMPUTE) begin
            one_minus_alpha <= (16'h1000 - alpha);  // 1.0 in
Q4.12 = 0x1000
            temp1 <= (gamma * max_q_next) >>> 12;    // γ *
max_Q_next
            temp2 <= reward + temp1[15:0];           // reward + γ
* max_Q_next
            temp3 <= (alpha * temp2[15:0]) >>> 12;   // α *
(reward + γ * max_Q_next)
            temp4 <= (one_minus_alpha * Q_current) >>> 12; // (1-
α) * Q_current
            temp5 <= temp3 + temp4;                 // final Q_new
        end
    end

    // Output & done flag
    always_ff @(posedge clk) begin
        if (state == DONE) begin
            Q_new <= temp5[15:0];
        end
    end

    assign done = (state == DONE);

endmodule
```

Frozen Lake Analysis: Challenge 10

Satyajit Deokar

📦 **Notes:**

- **Fixed-point format**: Q4.12 → 4 integer bits, 12 fractional bits.
    - 1.0 = 0x1000
- >>> 12 used to normalize after multiplication.
- Can be connected to a Q-table memory (RAM/BRAM) externally.
- Assumes start signal triggers a single update operation.

🧪 **Example Input Encodings (Q4.12 format):**

Value	Q4.12 Hex
1.0	0x1000
0.5	0x0800
0.9	0x0E66
0.1	0x0199
-5.0	0xF000

Frozen Lake Analysis: Challenge 10

Satyajit Deokar

block diagram to understand