**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:

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.