# Reinforcement Learning for Load Balancing in Distributed Systems

Khoa Nguyen
nguyen.khoa@northeastern.edu

Hung Nguyen
nguyen.hun@northeastern.edu

## I. Problem Statement

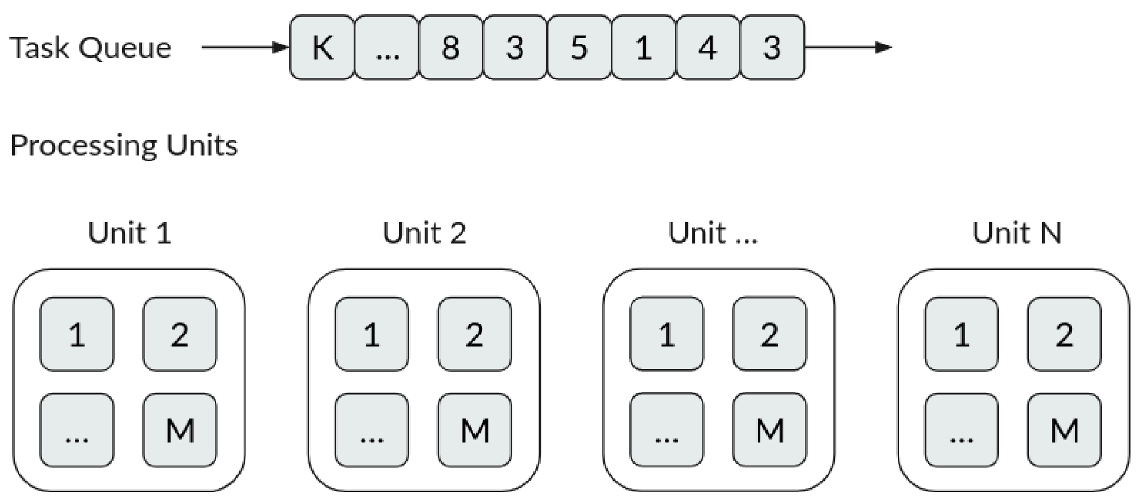
Load balancing is a popular problem that is usually encountered within the computer science space, whether it's thread/cpu utilization, or equal server distribution. It is also prevalent in daily life, as a lot of services require customers/clients to queue themselves up behind counters and wait for their turn. Load balancing can also be utilized here to make sure that everyone gets their turn in a reasonable amount of time, considering when they were first queued.

The objective of this project is to evaluate the effectiveness of reinforcement learning algorithms in load balancing tasks across multiple processors. Specifically, we consider a system composed of multiple processors (PUs), each with multiple threads, where tasks with varying execution durations must be assigned to processors dynamically. The goal is to design a scheduling policy that minimizes overall task completion time while maintaining balanced processor utilization.

We hypothesize that the Deep Q-Network (DQN) agent will outperform both Random and Q-Learning policies by minimizing task completion time and achieving higher utilization scores through adaptive decision-making.

## II. Environment

The environment is modeled after a queueing system, where tasks arrive randomly, but for training purposes, that queue is limited to a fixed number $K$ of tasks. The system includes $N$ processing units (PUs), working independently of each other, along with $M$ threads for each unit. Each thread can only process *one* task at the time.



Each task is assigned a random duration, bounded by a maximum length $D$, treated as a hyperparameter.

Time is discretized into "ticks," where each tick corresponds to one unit of progress on a task (a time step) for ease of training. During each tick:

- Each busy thread reduces the remaining duration of its assigned task by 1.
- Tasks that reach zero remaining duration are considered completed and removed.

At each tick, the agent is allowed to move **one** task from the queue to one of the available processors. Tasks cannot be preempted once assigned.

## III. Method

### IIIa. MDP

- State space: each state is represented by a vector (|next task in queue|, |task in PU$_1$ Thread$_1$|, |task in PU$_1$ Thread$_2$|, … |task in PU$_N$ Thread$_3$|, |task in PU$_N$ Thread$_M$|)

State space heavily depends on the hyperparameters $N$ processing unit and $M$ threads and its dimension size equates to: $1 + (N \times M)$, each dimension would have max size of $D$. The order in which the environment evaluates the next state for each tick would be: process each task in each thread in each processing unit first, then action from the agent is performed. Terminal state is defined with a vector of zeros i.e. there are no more tasks left in the queue nor in any CPUs.

- Action space: $N$ actions

Each action allows the agent to move **one** task from the queue to one of the processing units with the matching number. So if the agent chooses action *2*, then the task on top of the queue will be moved to one of the empty threads of the processing unit *2*.

An action can also be considered illegal if the queue is empty, or the processing unit that is getting queued into is maxed out on utilization (number of busy threads ÷ number of threads). In which case, the action equates to doing nothing.

- Reward function: $[-\ 1 \text{ living reward}]\ +\ \ [-\ \lambda \times var(\text{all processing units' utilization score})]$

We have a living reward to encourage the agent queuing tasks in ways that finish the fastest, but at the same time, the tasks also have to be evenly distributed over all the available processing units. This is done by having variance of the utilization scores as a negative reward. We added λ here as a hyperparameter for better tuning the scale of the negative reward.

### IIIb. Algorithms

Everything was implemented in C++ and all algorithms were trained using a shared environment class to ensure consistent simulation dynamics. Everything is written from scratch with the exception of the neural network implemented by using LibTorch (PyTorch API library for C++). Random seeds were fixed across runs for reproducibility.

**Random Policy** is used as the baseline for evaluating the performances of the algorithms below, as we observed that even in a medium bottlenecked configuration, it performs fairly well with task completion time.

**Q-Learning** is the classical reinforcement learning we want to use to see how well it can perform in a more complex environment like this. To handle the large state-space, we opted in to use hashmap to store visited (state, action) pairs, anticipating the need for a sparse Q-table, and allocating gigabytes of memory for tensors when most of them stay zero isn't memory efficient. The agent follows an ε-greedy exploration strategy during training, with a linear ε decay schedule from 1 to 0.1 over time.

**Deep Q-Network (DQN)** agent extends Q-Learning by using neural network to approximate the Q-function, along with a replay buffer and a second target neural network. We initialize the input and output dimensions equal to the state-vector dimension and $N$ (total number of actions) respectively. There are 2 hidden layers with size 64 each, a replay buffer size of 10,000 transitions, target network update at every 1,000 steps, and also an ε-greedy explorations strategy during training with an exponential ε decay schedule, and a learning rate of 0.001.

## IIIc. Evaluation metrics

To evaluate how effective each policy distributes workload across the available processing units, two metrics were employed: Task completion time and Utilization score with the idea that they provide complementary perspectives on performance and efficiency.

**Task completion time** is the number of ticks i.e. time steps taken to go from the start state to terminal state, at which point all tasks have been completed and the queue is empty.

While task completion time reflects raw speed, it does not account for how evenly the workload is distributed across PUs. We introduce a custom **Utilization Score** metric (or "U-Score") which quantifies how evenly usage was distributed. During each episode, the environment tracks how many threads are busy at each time step. At the end of the episode, we calculate the **utilization ratio** for each processor (or "PU utilization") as the number of busy threads over the total number of threads.

$$\text{PU utilization} = \frac{\#\, Busy\, Thread}{\#\, Threads}$$

This yields a value between 0 and 1 for *each* processing unit. The Utilization score is then calculated by using the coefficient of variation of the "PU utilization". Formally:

$$\text{U-score} = 1 - \frac{\sigma}{\mu}$$

where μ is the mean utilization across all PUs and σ is the standard deviation of utilization values. The score is clamped to the [0,1] range to avoid misleading negative values when imbalance exceeds mean usage. A score of 1 indicates perfect balance (equal usage across processors). A score closer to 0 indicates significant imbalance.

## IV. Result

### IVa. Experiment Setup

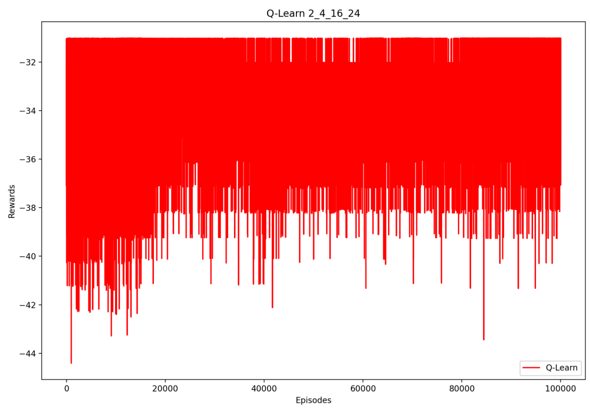
We evaluated three policies, Random, Q-Learning, and Deep Q-Network (DQN), on variations of the custom load balancing environment. The varying factor being the number of processors, number of threads, number of tasks, and max-length of each task. Specifically, we designed 2 configurations for this report:

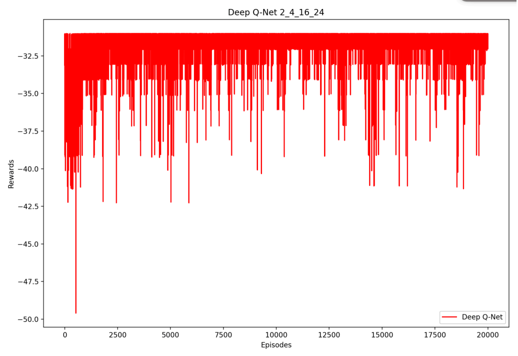
1. One is heavily bottlenecked (2PUs, 4 threads, with 16 tasks)
2. One has a larger action space (4 PUs, 4 threads, with 16 tasks)

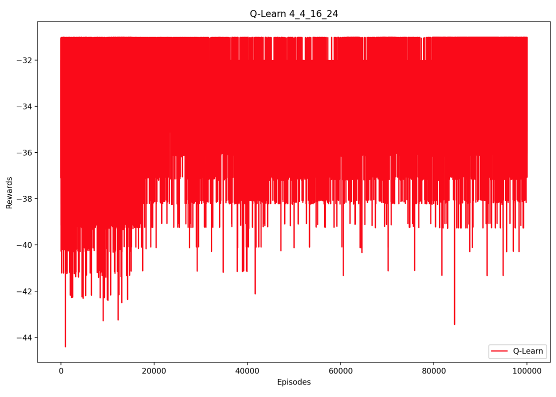
All tasks are generated with varying length. We want to observe how each algorithm scales differently.

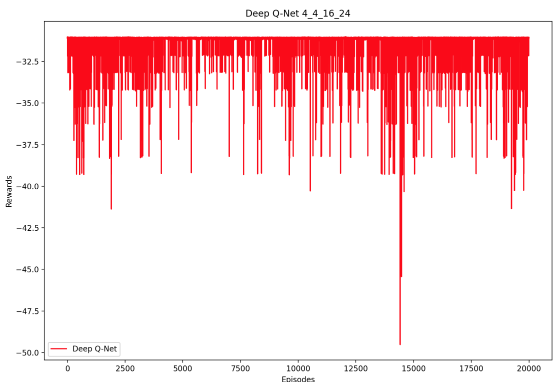
For each policy, we ran three independent trials to average out noise. For each trial, the agent trained for a fixed number of episodes (20K or 100K) and then performed a separate rollout to measure task completion time and utilization score. Average episode rewards were also recorded for plotting learning curves.  This large gap in episode numbers  allows us to compare training stability, performance, and practical usability.

### IVb. Training Stability









Deep Q-Network agent shows more stable training and seems to converge at a decent solution in the same or less amount of times to Q-Learn agent.

## IVc. Performance Results

The performance was measured using two metrics: Task completion time and Utilization score (U-Score). Additionally, we also recorded the average training time in seconds to measure practical usability. The results are summarized below:

| **2 PUs 4 Threads** | **Policy** | **# of steps** | **Avg. U-Score** | **Training time (CPU)** |
|---|---|---|---|---|
| | Random Policy | 44 | NA | NA |
| | Q Learning Policy (20K episodes) | 44 | 0.85 | < 1 minute |
| | Q Learning Policy (100K episodes) | 33 | 0.92 | ~1 minute |
| | Deep Q Network Policy (20K episodes) | 32 | 0.83 | ~1 hour 30 min |
| | Deep Q Network Policy (100K episodes) | 31 | 0.92 | ~5 hours |
| **4 PUs 4 Threads** | Random Policy (20K episodes) | 37 | NA | NA |
| | Q Learning Policy (20K episodes) | 37 | 0.79 | < 1 minute |
| | Deep Q Network Policy (20K episodes) | 37 | 0.81 | ~2 hours 30 mins |

In both environments, DQN consistently achieved the lowest average task completion time, outperforming both the Q-learning and Random policies. After 20K training episodes, Q-Learning achieved slightly higher utilization scores than DQN (e.g., 0.85 vs 0.83 in the 2PU setup). However, with additional training (100K episodes), DQN exceeded Q-Learning in utilization performance, with an average U-Score of 0.92 while maintaining faster task completion. Q-Learning initially balances workloads slightly better with limited training, DQN benefits more from extended training. However, DQN also requires significantly longer training times due to its reliance on deep neural network approximations, while Q-Learning remains computationally lightweight.

## V. Analysis

Both methods seem to work as intended, but with some significant trade-off:

In a heavy bottlenecked environment (having 16 tasks in total, while only being able to process $2 \times 4$ tasks maximum), Deep Q Network performs better on both metrics. Q learning result is slightly worse, but is still considerably better than random policy.

Moving to a less bottlenecked environment (4 processing units and 4 threads environment), DQN metrics are on par with Q Learning on both metrics, given both of them are only trained with 20,000 episodes. This is consistent with the result we saw from the last environment. They would both need to have more episodes to converge into the optimal policy, but that is besides the scope of why we want to increase the number of processing units.

Training time wise, we can see that DQN takes significantly longer to train. By increasing the number of processing units, we introduced way more parameters for the Q Network to compute, so it makes sense that it takes much longer for the same amount of episodes. The training time of Q Learning, on the other hand, does not seem to be impacted by the increment of processing units.

In conclusion, we believe that with the same environment and amount of episodes, Q-Learning seems to produce a slightly worse, if not on par, results with DQN. Q-Learning also only stores visited (state, value) pairs in a hashmap, in comparison to the large *float32* tensors that DQN uses, making it much more memory efficient. In a production environment where time is important and resources are scarce, Q Learning seems to be more ideal to be used for load balancing problems.

## VI. Discussion

For some extra discoveries we made while doing these experiments, environment setup and determining MDP probably caused us some worst problems:

Our initial state space is ($K$, Num busy threads in $PU_1$, …, Num busy threads in $PU_N$), but we figured very early on that it is a partially observable MDP. Using this state space, the agent would have no way of knowing how long the next task in queue is, and how long would it take for each processing unit to finish all their tasks, and this makes it impossible to load balance.

Action space used to have an explicit "Do nothing" action, along with the ability for the agent to move one task from one processing unit to another for each tick. From the terrible data we had earlier on, we have decided that these actions had contributed nothing, but artificially increased the training time. It also made the Q table larger, because then, we had 1 *(do nothing)* $+ \ n$ *(move from queue to processing unit)* $+ \ [n \times (n - 1)]$ *(move from one processing unit to another)* actions.

Instead of having $-\ var$(all processing unit's utilization score) to encourage distributing task evenly over all processing unit, the reward function used to have $-\ 1$ reward for each processing unit that is under $U$% utilization, with $U$ being a hyperparameter. This seemed to somewhat work, but we noticed that if $U$ was very low, then the agent only tried to load balance processing units up to $U$% of utilization, so we had a scenario like $[20\%,\ 100\%,\ 20\%,\ 20\%]$ with $U = 20$.

On the other hand, if $U$ was set too high, in a less bottlenecked environment, since the agent was hardly able to get every processing unit to reach $U$% utilization, it constantly got huge negative rewards, even when it was doing everything correctly, resulting in it failing to learn anything meaningful.

*Our source code can be found at: https://github.com/reesque/RLLoadBalancer*