POLYTECHNIC UNIVERSITY OF TURIN

Master's Degree in Mathematical Engineering

Master's Degree Thesis

Deep Reinforcement Learning for Dynamic Job-Shop Scheduling in High-Utilization Systems

Supervisors

Prof. Paolo BRANDIMARTE

Prof. Edoardo FADDA

Dr. Lorenzo MAZZA

Candidate

Valerio FIRMANO

July 2024

Summary

Dynamic job-shop scheduling is a significant challenge in the quickly evolving field of manufacturing technology. It requires flexible and optimized solutions that can adapt to changes in operations in real time. The use of deep reinforcement learning (DRL) to solve the Dynamic Job-Shop Scheduling Problem (DJSP) in high-utilization systems is investigated in this thesis. The goal of the study is to build a stable environment based on the OpenAI Gym structure, dynamically modeling machine operations and job arrivals. By leveraging Double Deep Q-Learning (DDQN), an advanced DRL technique, the research aims to train an intelligent agent capable of optimizing job sequences to minimize weighted tardiness and improve overall scheduling efficiency.

The framework that has been developed integrates essential elements like action selection, reward computation, and state representation, specifically designed to manage the unpredictable nature of job arrivals and processing times. The DDQN agent's performance is assessed using a simulation-based method in comparison to conventional dispatching rules under various utilization scenarios. The outcomes show that in high-utilization environments, the DDQN agent performs noticeably better than heuristic-based techniques, indicating its potential to improve real-world manufacturing processes. This work contributes to the larger field of operations research and industrial engineering by laying the foundation for future studies in the application of reinforcement learning to intricate, dynamic scheduling problems.

Acknowledgements

ACKNOWLEDGMENTS

I would like to thank Professors Fadda and Brandimarte,
as well as Lorenzo and Alessia for their support in the
completion of this project.

Table of Contents

List of Figures

List of Symbols

A_i	Arrival time of J_i
AM_k	Available time of M_k, $AM_k = \sum_{J_i \in J^k} t_{ik} - CO_k$
C_i	Completion time of J_i
CO_k	Time that M_k has spent on the current operation
D_i	Due date of J_i
J	Set of all jobs, $J = \{J_i : i = 1, \ldots, n\}$
J^k	Set of jobs being or queuing to be processed by M_k, $J^k = \{J_a, \ldots, J_z\}$
M	Set of all machines, $M = \{M_k : k = 1, \ldots, m\}$
O_i	Number of operations of J_i
S_i	Slack time of J_i, $S_i = TTD_i - WR_i$
SQC_i	Operation sequence of J_i, $SQC_i = [M_a, \ldots, M_z]$
t_j^i	Processing time of the j-th operation of J_i
T_i	Tardiness of J_i, $T_i = \max(C_i - D_i, 0)$
T_{sum}	Cumulative tardiness, the objective of this research
t_{ik}	Process time of J_i on machine M_k
TTD_i	Time-till-due of J_i
WR_i	Work-remaining of J_i, $WR_i = \sum_{j=x}^{O_i} t_j^i$, where x is the index of J_i's next operation
NOW	Current time in the system

Chapter 1

Introduction

1.1 Motivation

At a time when technological progress is advancing very rapidly, industries cannot afford not to embrace innovation and integrate it into their production systems. In fact, applied mathematics makes a huge innovative contribution to the world of manufacturing with studies on optimizing its processes. Although Scheduling problems are classic problems in Operations Research and have been addressed extensively, the literature has focused more on combinatorial solving of static cases, giving little consideration to problems involving dynamic and stochastic new arrivals, addressed more with heuristics. However, recent advances in the field of artificial intelligence, particularly Reinforcement Learning, have made it possible to re-address the problem using this technology, attempting to create agents capable of addressing the scheduling problem even in dynamic and stochastic contexts. This thesis fits exactly into this context, seeking to lay the groundwork for using the RL framework in the search for new solutions for Dynamic Scheduling problems.

1.2 Problem Definition

As mentioned before, we are in the context of Dynamic Scheduling problems, in particular we are considering the Job-Shop Dynamic problem. The Job-shop Scheduling Problem (JSP) represents a classical challenge in operational research and production management, involving the allocation of jobs to resources at specific times. Each job consists of a sequence of operations that must be processed on specific machines in a predetermined order. The primary objective is typically to optimize a performance criterion, such as minimizing the total completion time (makespan), minimizing delays, or maximizing machine utilization. In its static form, JSP assumes that all jobs, machines, and constraints are known in advance and the problem consist into finding the best combinatory order for the Jobs on the machine, respecting the order constraints of the operations.

However, real-world manufacturing environments are rarely static. They are dynamic and often subject to unpredictable changes, such as machine breakdowns, urgent job arrivals, changes in job priorities, and variations in processing times. This variability introduces significant complexity, requiring adaptive and flexible scheduling approaches. The dynamic Job-shop Scheduling Problem (DJSP) extends the static version by incorporating these real-time changes and uncertainties, thereby providing a more realistic and applicable model for practical scenarios.

The shift from static to dynamic scheduling is not merely a theoretical exercise but a practical necessity for modern manufacturing systems striving for agility and responsiveness. While static scheduling provides foundational insights and solutions, dynamic scheduling aims to reflect the operational realities and enhances the system's ability to maintain high performance under fluctuating conditions.

1.3 Reinforcement Learning for Dynamic Job-Shop Scheduling Problems

As we have already seen, the main complexity of this type of problem is its dynamic nature. This can happen in many ways, from the stochastic arrival of new orders to the breakdown of some machines or the arrival of particularly urgent jobs. The simplest way to deal with this dynamism is to use so-called dispatching rules, heuristics that choose to schedule one job before another using simple priority rules (the most famous is the so-called FIFO or First In First Out, which involves scheduling jobs in the order in which they arrive on the system, with the first one to arrive being executed first). We can easily imagine that these rules, while providing agile and fast methods, provide sub-optimal solutions, sometimes far from the best solution.

Can better solutions be found? The literature explores the path of reinforcement learning to find agents that are able to provide the best possible action (i.e. which and how to schedule the next job) given the state we are in at the moment this decision needs to be made.

This problem is very complex and has a large number of variables to be taken into account: the nature of the scheduling problem (there is not only the classical job shop), the nature of the dynamic component (new arrivals, machine breakdowns, priority changes,...), the presence of stochastic factors (new arrivals at random times, random processing times,...), different objective functions to be minimised (tardiness, weighted tardiness, makespan, just in time,...). For this reason, the solutions found in the literature can be very different and will be examined in more detail in the following sections.

1.4 Contribution

In this thesis we focus on the Job-Shop problem in which the dynamic factor lies in the arrival at stochastic times of new jobs with stochastic processing times. The main contribution is on two fronts. On one side the development of an Environment from scratch that models the problem, following the reset-action-step-reward structure of OpenAI's *gym* library, making the use of the same Environment very agile and easily adaptable to different agents. On the other hand, I developed a Framework for training an Automated Agent using double deep Q-Learning (DDQN), one of the most advanced deep reinforcement learning techniques to date, resulting in training the agent and testing its performance on different instances compared with dispatching rules.

Chapter 2

Literature Review

As I introduced in the previous section, the problem of dynamic scheduling with RL is really broad and complex. Even just small differences in problem definition and/or instances result in very different and specific approaches, especially in the case of Reinforcement Learning.

This is due to the fact that as much as Reinforcement Learning tries to be an “Artificial Intelligence” method and we therefore think that it is able to solve problems in a general way, it actually finds solutions that depend very much on the state-action combination that we define in our problem and on the condition of the problem itself. To understand this better, let us take an example concerning this work: when we are in a dynamic Job-Shop Environment with stochastic arrivals of new jobs and these jobs have stochastic processing times, it means that each new job that arrives at the system is (almost certainly) different from the previous one and each instance generated is different from the others. This means that the "job space" can be potentially infinite and can quickly become overwhelming for the RL agent if not trained properly. To address this issue, some approaches involve selecting only a subset of jobs at each step or defining the state as a set of aggregate features of the available jobs to have always a fixed dimension of the state space. The definitions of actions also vary across different approaches. This illustrates how even minor changes in the problem can lead to significantly different solutions. In the next section, we will explore these various solutions.

2.1 Job-Shop Dynamic Problem

Let’s give a broader and more technical introduction to the problem of Dynamic Job-Shop. Note that in this thesis we will not consider its Flexible version but only the pure Job-Shop. Starting with a general notation introduction:

- **Machines:** M is the set of m machines, k is the index of a machine, $k =$

$1, 2, \ldots, m$

$$M = \{M_1, M_2, \ldots, M_k, \ldots, M_m\}$$

- **Jobs:** J is the set of n jobs, i is the index of a job, $i = 1, 2, \ldots, n$

$$J = \{J_1, J_2, \ldots, J_i, \ldots, J_n\}$$

- **Operations:** Each job J_i has a sequence of n_i operations, $O_{i,j}$ is the j-th operation of job J_i

$$O_{J_i} = \{O_{i,1}, O_{i,2}, \ldots, O_{i,j}, \ldots, O_{i,n_i}\}$$

The Job Shop Problem (JS) is defined as: Each job in J has its distinct route to be processed on each of m machines, all J_i have the same number of operations, and $n_i = m$. The Dynamic Job Shop Problem (DJSP) is a variant of the classical Job Shop Scheduling (JSP) problem where the job and machine environment changes over time. Unlike the static job-shop problem, where all jobs are known in advance and do not change, the dynamic version incorporates real-time events, in our case we consider only new job arrivals.

The objective of scheduling is to optimize resource use to meet due dates at minimal cost, enhancing inventory management, resource utilization, and customer satisfaction. These measurements are done through a so-called objective function, which measures how good that schedule is relative to the objective to be achieved. Let r_i, d_i, and c_i be the release date, due date, and completion date of J_i, respectively. In some cases, there is a weight w_i associated with J_i to address its importance or value. Some common objective functions to be minimized are:

- **Makespan:** $C_{\max} = \max\{c_i \mid i = 1, \ldots, n\}$
- **Flow time:** $F_i = (c_i - r_i)$
- **Lateness:** $L_i = c_i - d_i$
- **Earliness:** $E_i = \max\{0, d_i - c_i\}$
- **Tardiness:** $T_i = \max\{0, c_i - d_i\}$
- **Absolute deviation:** $D_i = |c_i - d_i|$
- **Unit penalty:**

$$U_i = \begin{cases} 1, & \text{if } c_i > d_i \\ 0, & \text{otherwise} \end{cases}$$

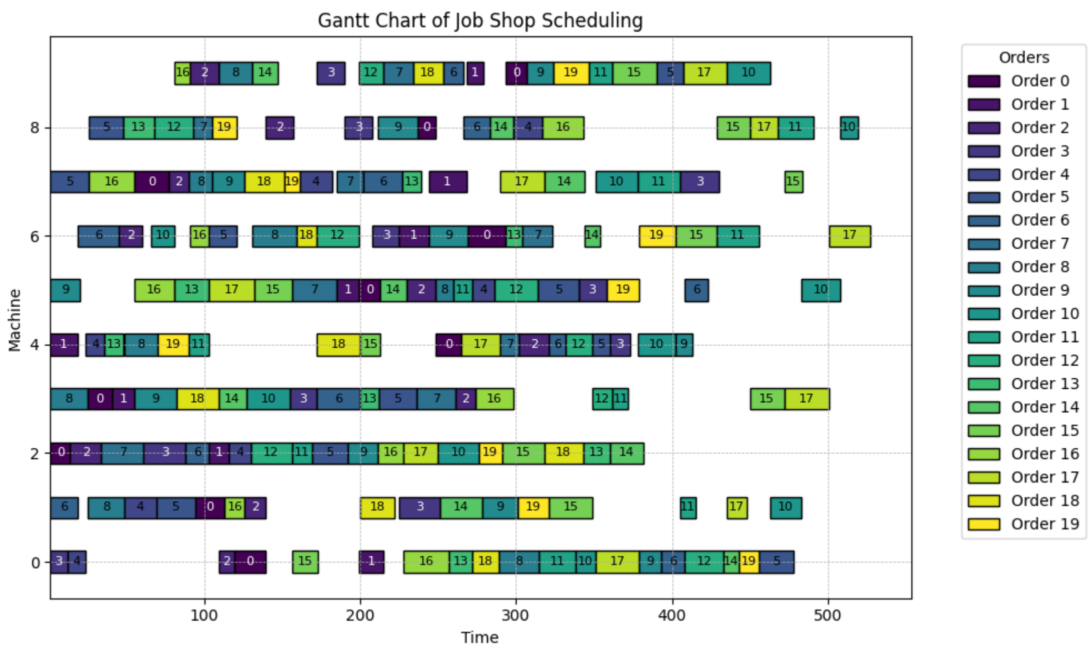

Figure 2.1: An example of a solution of a Job-Shop scheduling problem with 20 jobs. This kind of representation is called Gantt chart, on the vertical axis we have the 10 different machines, on the horizontal the time. Each colored bar represent an operation of a specific job (*order*) and we can see from this chart the flow of jobs and their position in the schedule

Schedulers usually use maximum, sum, average or weighted sum of above-mentioned performance measures or some combinations of them.

Before the development of RL to solve DJSP, solutions were mainly based on a set of dispatching rules heuristics (very easy and fast to use but clearly sub-timed) and meta-heuristics that however were often complex and computationally heavy and slow. So let's give a look to some works that develop RL algorithms to solve DJSP.

2.2 Reinforcement Learning in Dynamic Job-Shop Scheduling problems

Let's dive deep into the latest researches on Reinforcement Learning in Dynamic Job-Shop Scheduling problems. In this section I report only papers related to the Job-Shop problem in the strict sense without considering its flexible variant.

The first work by Inal et al. [1] propose a multi-agent system with reinforcement learning (MAS-RL) for the DJSS problem. Their approach is designed to minimize tardiness and flow time, improving the dynamic scheduling capabilities compared to traditional dispatching rules. The study compares the MAS-RL approach with FIFO, SPT, and EDD rules across various performance metrics such as the proportion of tardy jobs, mean tardiness, and mean flow time. The results demonstrate that the MAS-RL outperforms these traditional methods, particularly under high workload conditions. However, this work has limitations since although they are considered stochastic time arrivals, new jobs are only taken from a finite closed set. This represents a major simplification since having a finite representation of the state a Q-learning algorithm can be applied, which however cannot be generalized to the case where new jobs have unfixed and randomly generated processing times.

Also in the work of Wang at al. [2] fixed instances are considered to measure the performance of their algorithm, although in a final section they show how randomly changing some instance operations maintains good performance with quick retraining of the agent. A policy iteration algorithm is used here, thus involving policy training, rather than in updating state-action pair values as in value learning algorithms (e.g., Q-learning). In particular they use PPO algorithm that is the state-of-the-art for policy iteration. Their objective in this case is the simple Makespan, so this reward structure incentivizes the agent to develop policies that effectively schedule jobs to minimize the overall completion time. They use as benchmark some dispatching rules to test their framework.

Wang and Liao (2023) [3] present an interesting approach in their study where they simulate the dynamic arrival of jobs in a discrete event simulation (DES) system. In their system, both the arrival times of jobs and their processing times on the machines are simulated from exponential and uniform distributions, respectively. This results in each job being different, leading to a potentially infinite state space and causing a "dimensional explosion" that can confuse the agent in reinforcement learning (RL) problems. To address this problem, they opt for an aggregate state representation. The state is represented by aggregate features-specifically, the minimum, maximum, mean, and standard deviation of certain job attributes. These attributes include the number of jobs, their processing times, and the state of the machines. This aggregated state information helps the RL agent generalize across different scenarios, thereby increasing the scalability and robustness of the model. Due to the aggregate nature of the state representation, the agent's actions cannot directly involve selecting one job over another. Instead, at each decision point, the agent chooses one of 11 possible scheduling rules. These rules represent common scheduling heuristics. The Proximal Policy Optimization (PPO) algorithm is then used to learn the optimal policy for scheduling the jobs.

Wu et al. [4] also use PPO as learning algorithm but with a different approach on feature representation side. To deal with the variability of the state dimension they represent the state using two matrices, the machine matrix and the remaining time matrix. The machine matrix captures the status of machines, while the remaining time matrix tracks the processing time left for jobs. These matrices are treated as image-like inputs to a convolutional neural network (CNN) within the SPP-Net which converts variable-sized inputs into fixed-length feature vectors. The action space in their model consists of paired priority dispatching rules (PDRs), which include common scheduling heuristics such as Shortest Processing Time (SPT), Most Work Remaining (MWKR), and Longest Remaining Machine time (LRM), among others.

Liu et al. [5] propose another interesting approach to solve the scheduling problem. Like Wang and Liao, they simulate both the arrival times and processing times of jobs, leading to a potentially infinite state space. However, unlike Wang and Liao, Riu et al. do not aggregate job features to create a fixed-size state. Instead, they maintain a job-level state representation. To address the challenge of having a possible different number of jobs to choose from at each step, they developed a method to ensure the state always has the same dimensions. Their experiments showed that the agent rarely had to choose among more than four jobs at a time. Based on this insight, they defined four dispatching rules, each selecting one job from the buffer. If fewer than four jobs are in the buffer, some jobs are repeated in the state. They also included the option to wait for a job that is about to finish its current operation on the previous machine, adding this to the state. This approach ensures that the state always has a dimension of five, and the action corresponds directly to selecting one of these five jobs. They use a Double Deep Q-learning algorithm (value learning) with tardiness as the objective function and a reward based on the queuing time of a job at each machine.

Chapter 3

Solution Framework and Implementation

3.1 Markov Decision Process Framework for a RL solution

Before going into the details of our problem, it is necessary to make a small introduction to how RL problems are modeled through Markov Decision Processes. An MDP provides a formalism for decision-making in environments that evolve over time following both deterministic and stochastic factors. It is defined by the following components:

- **States (S):** The set of all possible states in the environment. A state $s \in S$ represents a specific configuration of the environment at a given time.
- **Actions (A):** The set of all possible actions. An action $a \in A(s)$ is a decision or move that an agent can take when in state s.
- **Transition Function (T):** The probability of transitioning from one state to another, given a specific action. Formally, this is defined as $T(s, a, s') = P(s' \mid s, a)$, which represents the probability of reaching state s' from state s when action a is taken.
- **Reward Function (R):** The immediate reward received after transitioning from state s to state s' due to action a. It is denoted as $R(s, a, s')$. In some cases, a simpler form $R(s, a)$ is used, representing the expected reward for taking action a in state s.
- **Policy (π):** A policy defines the behavior of the agent. It is a mapping from states to a probability distribution over actions. A deterministic policy is

denoted as $\pi(s) = a$, where the agent always takes action a in state s. A stochastic policy is represented as $\pi(a \mid s)$, the probability of taking action a in state s.

- **Discount Factor (γ):** A factor $\gamma \in [0, 1]$ that discounts future rewards. It represents the degree to which future rewards are considered less valuable than immediate rewards. A discount factor of 0 means the agent only cares about immediate rewards, while a discount factor close to 1 means the agent values future rewards almost as much as immediate ones.

The goal of an RL agent is to find an optimal policy π^* that maximizes the expected cumulative reward, often referred to as the return. The return G_t is defined as the sum of discounted rewards from time step t onwards:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

3.2 Problem Description

This study addresses the challenges associated with Dynamic Job Shop Scheduling Problems (DJSP). Initially, the workshop consists of $n_{initial} = 10$ jobs $\{J_1, .., J_{10}\}$ and $m = 10$ machines $\{M_1, \ldots, M_{10}\}$. Each job J_i is composed of n_i operations (the number can vary for each job) $\{O_{i1}, \ldots, O_{in_i}\}$ that must be processed sequentially on the machines. In this dynamic workshop environment, where n_{new} jobs arrive randomly, the processing and sequencing information for each job J_i can only be determined after it enters the system. The processing time for the operation O_{ij} is denoted as OP_{ij}. We define a due date tightness factor, f_i, each job has an arrival time A_i, and the due date D_i is generated based on OP_{ij} and f_i.

$$D_i = A_i + f_i \cdot \sum_{j=1}^{n_i} OP_{ij} \tag{3.1}$$

Let C_i represent the completion time of job J_i, and T_i denote its corresponding tardiness and w_i his weight. Thus, the optimization objective weighted total tardiness $T^{(w)}_{sum}$ is calculated as:

$$T_i = \max(C_i - D_i, 0) \tag{3.2}$$

$$T^{(w)}_{sum} = \sum_{i=1}^{n_{initial}+n_{new}} w_i T_i \tag{3.3}$$

A complete and comprehensive list of the nomenclature, description and symbols of all quantities involved in the problem can be found in the List of Symbols at the beginning of the document.

3.3 Modelling of the Simulation-Based dynamic Job-Shop problem

In Reinforcement Learning (RL), training involves an interactive process between an intelligent agent and its environment. Consequently, designing a smart scheduling framework requires focusing on three essential components: the environment, the agent, and the Markov Decision Process (MDP) model. The MDP, characterized by its defined states, actions, and rewards, comprehensively describes the intricate learning process.

I decided to model the environment through the concept of machine buffers: within the environment, there are $m = 10$ buffers, each corresponding to a specific machine. These buffers contain the set of job operations waiting to be executed on their respective machines. Whenever a job completes an operation on a machine, it is transferred to the buffer of the next machine in its execution sequence (machine setup time and job movement time are ignored). Additionally, when a new job arrives in the system, its first operation is placed in the buffer of the machine designated for its execution. The agent's task is to select the best possible operation from the buffer of a machine whenever that machine becomes idle. Thus, differently from the static problem where the whole set of operations is considered to optimize the schedule and the action becomes the entirety of the schedule itself, the dynamic problem configuration requires selecting one action at a time. Each chosen action is then incorporated into the schedule sequentially, continuing this process until the entire simulation is complete.

In our case the 10 static jobs are solved with Gurobi Solver, giving an initial solution. Anyway I decided that the instant the first of the new jobs arrives at the system, all operations that have not yet been executed from the static schedule can be rescheduled (even if they belong to the initial static jobs) and they are considered as "dynamic jobs" arrived at time 0.

Now we can break down the main components of the Environment with their implementation

3.3.1 Objective and Scenarios

For this project i chose to minimize the total weighted tardiness of all jobs:

$$\text{Minimize } T_{\text{sum}}^{(w)} = \sum_{i=1}^{n} w_i \cdot T_i$$

where w_i is the weight indicating the importance of J_i's tardiness.

The development and validation of scheduling strategy are simulation-based; three types of factors need to be specified to create the simulation environment:

- Arrival times of the jobs distributed by an exponential distribution of rate β. This rate (as will be reported more precisely in the numerical experiments), is chosen in dependence on the average time of simulated job operations: Let $E(t)$ denote the expected processing time of operations, In a DJSP in which jobs visit all machines, the expected utilization rate of the job shop is calculated as:

$$E(utilization\_rate) = \frac{E(t)}{\beta}\dot{1}00\%$$

  I preferred to select the right beta in order to have high utilization rate since in these scenarios the difference between a trained Agent and dispatching rules can be better seen.

- Each job has processing times on the machines distributed uniformly so $t_{i,k} \sim U[1, upper\_value]$.

- I defined the due date as the sum of the working times of the operations of each job times a multiplication factor$D_i = \alpha_i \times \sum_{j=1}^{O_i} t_{i,j}$. The multiplication tightness factor $\alpha_i = 1.5$ for each job.

3.3.2 State representation

As we introduced earlier our Environment consists of a set of 10 machines, each of which corresponds to a buffer of operations-jobs waiting to be processed on that machine. In the case of a DJSP it comes naturally to think of the action as the choice of one of these operations to be mounted on the machine when the machine becomes idle, so the state will have to somehow give us a description of the characteristics of the jobs in the buffer so that the agent, "reading" the state, can make its choice. As explained in the literature review, though, this way of conceiving the environment brings with it a major problem regarding resolution with RL, since it is impossible to establish a fixed size of the state since the number of operations within a machine's buffer is constantly changing. While there are those who solve this issue by using a fixed number of aggregate features of operations in the buffer, such as [3], I decided to take a different approach, as described in [5], using a Minimum Repetition algorithm.

To understand this algorithm first we need to observe (as is pointed out in the paper [5]) how in most cases the action to be taken is trivial (there are 0 or 1 jobs in the queue when the machine gets idle) or there are 2 to 4 jobs in the buffer, with a few cases where 4 jobs are exceeded. Therefore it is decided as a first step to set the number of jobs that are considered in our state to 4 whenever an action is to be taken. With this fixed number, each time a machine becomes idle, 4 jobs must be selected from the buffer. If the buffer contains fewer than 4 jobs (but more than zero, as zero implies a null action), some jobs will be selected multiple times. If there are more than 4 jobs, only a subset of 4 will be chosen.

There is a need to clarify how this selection called "Minimum Repetition" (MR) occurs.

Some additional variables are introduced to have more information within the system: (1) for job J_i, the available time of its succeeding machine is denoted as SAM_i; (2) the time J_i has waited in the current queue is denoted as CQ_i.

Six features are extracted for each candidate job: (1) $t_{i,k}$ represents the time required to process the candidate job, which also extends the queuing time for other jobs if selected; (2) WR_i indicates the minimal time before the completion of a job, showing the potential to reduce system congestion; (3) S_i is the slack time, measuring the urgency of a job; (4) SAM_i acts as a surrogate measure of the queuing time for the job's next operation; (5) CQ_i represents the time the agent has detained the job, indicating the agent's responsibility for any job tardiness ; (6) w_i, the weight of the candidate job, since, differently from [5], we want to consider the weight of the jobs in the tardiness.

Than we introduce 4 different well-known sequencing rules are used to select candidate jobs, whose information would be filled into the state space. These rules focus on different aspects of jobs to ensure that the candidates possess at least one unique feature that makes them more urgent or more suitable than other jobs. The rationales are introduced as follows:

Shortest processing time (SPT): selects the job that can be processed in the shortest time, to minimize the slack time consumption for other queuing jobs.

Least work remaining (LWKR): selects the job that could exit the system in the shortest estimated time, to reduce the overall congestion level.

Minimal slack (MS): selects the most urgent job to protect it from being tardy or incurring too much tardiness.

Work in queue (WINQ): selects the job that exposes its succeeding operation to the least congestion, to balance the workload of machines and avoid wasting machine idle time.

Now the MR approach is just an application of these four sequencing rules to the candidate jobs, included without repetition when the number of jobs in queue is greater than 3; otherwise, the jobs that conform to more rules would appear more times in state space. The Pseudocode of MR is presented in **Algorithm 1**, while its implementation is presented in Appendix A.1:

We finally obtain a (4x6) state composed of 4 candidate jobs with 6 corresponding features each.

Algorithm 1 Minimal-repetition approach to build the state space for M_k

```
Require: $J_k$
 1: Initialize empty state tensor: $s_t \leftarrow [\,]$
 2: Initialize the buffer job set: $J^{\text{buffer}} \leftarrow J^k$
 3: Initialize the rule set: $Rules \leftarrow [SPT, LWKR, MS, WINQ]$
 4: Initialize the rule count: $x \leftarrow 1$, $y \leftarrow 1$
 5: while $J_{\text{buffer}} \neq$ empty and $x \leq 4$ do
 6:     Select $J_a$ from $J^{\text{buffer}}$ by $x^{th}$ rule in $Rules$
 7:     Append feature vector $[t_{a,k}, WR_a, S_a, SAM_a, CQ_a, w_a]$ to $s_t$
 8:     $x \leftarrow x + 1$
 9:     Delete $J_a$ from $J_{\text{buffer}}$
10: end while
11: while $x \leq 4$ do
12:     Select $J_b$ from $J^k$ by $y$th rule in $Rules$
13:     Append feature vector $[t_{b,k}, WR_b, S_b, SAM_b, CQ_b, w_b]$ to $s_t$
14:     $x \leftarrow x + 1$, $y \leftarrow y + 1$
15: end while
16: return $st$
```

3.3.3 Action

Following the state definition outlined above, defining the action space becomes straightforward, as it corresponds directly to the state space. Whenever an action is required, the agent will select a job from those presented in the state, as determined by the MR algorithm.

3.3.4 Reward

A significant challenge in DRL-based scheduling lies in the design of the reward function.

In a dynamic scheduling problem focused on minimizing cumulative tardiness, the production performance results from the collective behavior of agents over an extended period. During this time, a series of reward signals (realized tardiness of jobs) appear and must be associated with the actions that caused them. Additionally, the information about tardiness is quite sparse, as it is only experienced at the end of each job. To address this, it is necessary to find a way to distribute the responsibilities of tardiness across each action taken during the whole process.

The idea for Reward shaping arises from the fact that the final tardiness comes from the resultant of all the times a job was held in a machine's buffer. In high utilization environments, with many jobs arriving dynamically, tardiness is itself unavoidable but each decision made by machine agents in the job's sequence of operations affects subsequent states, subsequent decisions, and the final value of tardiness itself.

While dispatching rules are "constructive" methods, which look neither at the past nor the future of the schedule but make a decision only on the current state situation, in this paper an attempt is made to develop a method that succeeds in somehow understanding how the agent's action on each machine affects the actions of subsequent agents and the final tardiness of the job.

Therefore, again inspired by the work of [5], I decided to implement an Asynchronous Reward system, based on the queueing times (the times spent in the buffer) of each job on the respective machines. This system is called asynchronous since only at the end of the job's path in the system I can compute its tardiness and queueing times on each machine.

In contrast to a classical MDP, where an action taken in a given state yields an immediate reward, or a multi-agent system, where multiple agents take actions simultaneously to receive an immediate joint reward, this system operates differently. Here, multiple agents (each machine) make sequential decisions on which job to select from the buffer. They receive a joint reward only after several asynchronous time steps, as illustrated in fig.3.1.

So I implemented a reward-shaping algorithm based on queue time, where the joint reward is given upon job completion, once the effects of all actions have been fully realized. At this stage, the production history is backtracked, and the reward for each agent's action is determined based on the duration they contributed to the job. Some of the crucial points for reward shaping are:

1. All agents along a job's trajectory incur penalties for any tardiness, but there are no rewards for early completion. While offering rewards for timeliness might speed up policy convergence, it could also lead to a focus on maximizing earliness, thereby decreasing the incidence of tardy jobs but potentially increasing the overall tardiness accumulated.

2. Sequencing choices impact not only the jobs in the current buffer but also influence other agents by determining the next buffer the job will join and its timing. If a job is tardy, agents share responsibility for both the time spent in their own buffer and the time spent in the subsequent buffer. Agents learn to avoid overloading other agents' buffers by adjusting their own buffer times, validated with a shift-back ratio of $\alpha = 0.2$.

3. As previously discussed, queuing is unavoidable and tardiness is a common occurrence in the given training and validation scenarios. To prevent undue penalization, agents are not penalized for slack time consumption beyond their control. The queuing time is only counted once jobs are ready for processing.

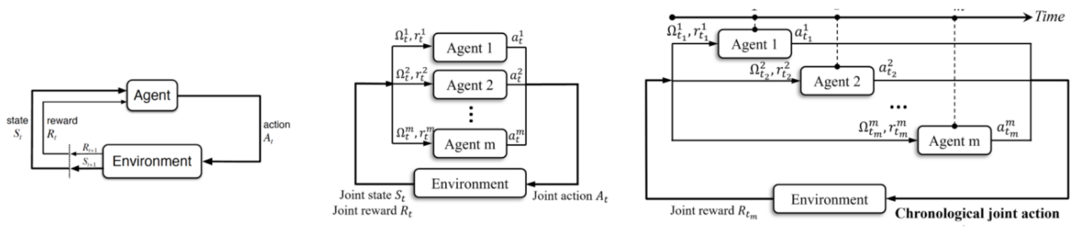

Figure 3.1: Different state-action-reward system between MDP, Multi-Agent MDP and Asynchronous MDP

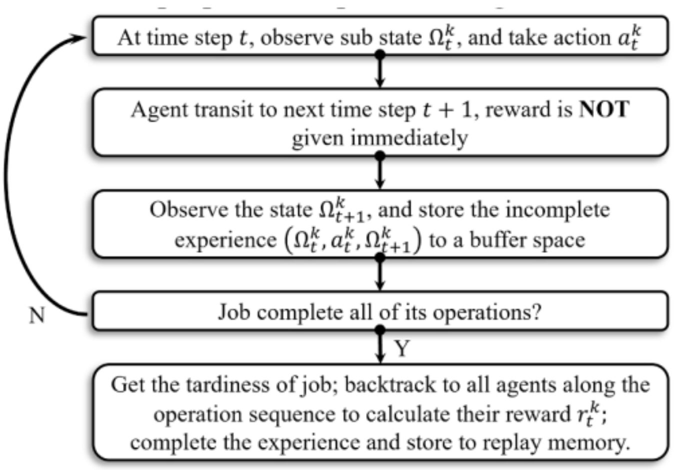
At time step t, observe sub state Ω_t^k, and take action a_t^k
Agent transit to next time step $t+1$, reward is **NOT** given immediately
Observe the state Ω_{t+1}^k, and store the incomplete experience $(\Omega_t^k, a_t^k, \Omega_{t+1}^k)$ to a buffer space
Job complete all of its operations?
N
Y
Get the tardiness of job; backtrack to all agents along the operation sequence to calculate their reward r_t^k; complete the experience and store to replay memory.

Figure 3.2: Transition and experience building in proposed algorithm

4. Agents incur harsher penalties for holding up jobs with shorter slack times. The initial slack time of a job is converted into a criticality factor $\beta \in (0, 2)$ using a sigmoid function. The sensitivity of β to slack time is controlled by a parameter δ.

5. An additional criticality factor is introduced to impose greater penalties on jobs with higher weights, through a "weighting factor" ν. This factor is defined as $\nu_i = 1 + w_i$, where defining it as one plus the weight (where $w_i \in$ (0,1)) ensures that the reward value is not diminished excessively, maintaining the effectiveness of the criticality factor since both factors are multiplied.

6. The magnitude of queue time is scaled by a factor ϕ, ensuring that most reward values lie within the range $[-1, 0]$, which is suitable for neural network inputs. The square of the waiting time is used to impose severe penalties for long waiting durations.

For each job J_i we define some extra notations: (1) queue time before each operation, denoted as vector $Q_i = [Q_i^1, .., Q_i^{O_i}]$, (2) slack time at the time of job arrival at each buffer $SA_i = [S_i^1, .., S_i^{O_i}]$ and the results will be stored in the reward vector $R_i = [R_i^1, .., R_i^{O_i}]$. The pseudo-algorithm for reward computation is presented in **Algorithm 2**, while the implementation is in *Appendix*A.2

Algorithm 2 Calculation of reward upon the completion of job J_i

```
Require: $T_i$, $Q_i$ and $SA_i$
 1: Initialize empty reward vector $R_i \leftarrow [\ ]$
 2: if $T_i == 0$ then
 3:     Fill reward vector with 0s: $R_i \leftarrow [0, \ldots, 0]$, $|R_i| = O_i$
 4: else
 5:     for $j \leftarrow 1$ to $O_i$ do
 6:         Re-construct the queue time: $RQ_i^j = (1 - \alpha) \times Q_i^j + \alpha \times Q_i^{j+1}$
 7:         Compute the criticality factor: $\beta = 1 - \left(\frac{S_i^j}{|S_i^j| + \delta}\right)$
 8:         Compute the weighting factor: $\nu = 1 + w$
 9:         Get the reward: $R_j^t = -\left(\beta \times \nu \times \frac{RQ_i^j}{\phi}\right)^2$
10:         Clip the $Rt_j$ to $[-1, 0]$
11:         Append $Rt_j$ to $R_i$
12:     end for
13: end if
14: return $R_i$
```

3.4 Environment Implementation

We saw how all the components of an MDP have been defined in the project, starting from the ideas of [5] with some new components. One key part of the project is missing to be presented is to show how all these components (state-action-reward) interact with each other in the Dynamic Job-Shop system we have described. The place where these interactions take place, which evolves the dynamics of the system, which houses the agents and their actions, and which stores the information of the states is clearly the Environment. This class is the core of the code implementation of this work, around which all other objects are built. It is therefore necessary that its implementation be as general, robust, and modular as possible, so that different agents and different solution methodologies can be tested within it.

Even if the description of his implementation is not crucial for the results, I think it's important to explain how the Environment code have been designed in order to be used eventually in the future for other researches.

To achieve the most general implementation possible, the best approach was to follow the structure of the Gymnasium (formerly Gym) library developed by OpenAI. This framework provides a standardized way to implement environments in reinforcement learning by defining several common methods, such as:

- **init** : as in any declaration of a class, the first method to be implemented is always initialization, within which we instantiate all the initial variables necessary for the unfolding of the problem
- **reset**: the reset function is in charge of generating a new episode of the problem, resetting all variables to their initial conditions in order to re-start the system evolution process. It is a crucial function especially in training-evaluation phases since in these situations we have to generate new episodes several times.
- **step**: the step function is the central function of the environment, since it is responsible for its evolution. It takes in the action taken by the agent and evolves the current state of the system to the next step, changing all the variables internal to the environment itself.

The pseudo-code of a gym-like Environment that we followed is shown below:

```python
class DynamicPlant(gym.Env):

    def __init__(self, instance_name):
        super().__init__()
        "implement the initialization here"

    def step(self, action):

        reward = compute_reward(action)
        obs = get_next_state(action)

        info = {}
        self.current_event += 1
        done = done_condition

        return obs, reward, done, info

    def reset(self, seed=None):

        self.current_event = 0
        self.scenario = simulate_new_scenario
        obs = get_initial_state

        return obs
```

And the structure of the main function is something like:

```
if __name__ == '__main__':

    env = DynamicPlant(instance_name, n_new_orders=10)
    agent = Agent(env)

    done = False
    obs = env.reset()

    while not done:
        action = agent.get_action(obs)
        obs, reward, done, _ = env.step(action)
        print(reward, obs)
```

This outline presents only the structural foundation of the algorithm. The problem we are addressing is significantly more complex and involves numerous intricate challenges that require in-depth examination. Therefore, the following sections will detail each function I have implemented, providing insight into the solutions developed for this problem. Before deep diving into the implementation of these functions , it's important to note that the DJSP can be modeled and approached in various ways. The method I chose for this project addresses the problem using a "single" approach, focusing on the evolution of the system one operation at a time. When an operation is completed on a machine, the agent's task is to select only the next operation for that machine, repeating this process until the end of the simulation. An alternative approach is to reschedule the entire set of future operations whenever a new job enters the environment. This method is not "single," as it aims to provide a complete schedule rather than just determining the next operation.

3.4.1 Init Function

The initialization of the environment class contains some essential steps:

- As we mentioned earlier our system consists of 10 static jobs to which dynamic jobs are added later via stochastic arrivals. In the initialization of the environment, the first step that is performed is to read the static instance (and also the dynamic one, which, however, can be regenerated multiple times in the reset in case of multiple training or evaluation) and its solution via the commercial solver Gurobi that determines the best initial solution.
- Some functions such as *timing, envmanager, and reward* are initialized that will be needed later in the step function
- The dimensions of the state and action tensors needed for the Deep Q-learning algorithm are defined

The full implementation of the Init function can be found in *Appendix* A.3.1

3.4.2 Reset Function

The reset function serves several purposes:

- It generates new dynamic instances when we are in a training or evaluation scenario where we want to keep the static solution unchanged (or avoid recomputing it, as it is particularly time-consuming) and only generate new dynamic jobs.
- During the reset, all system variables are initialized, establishing the connection between the static and dynamic parts. At the moment of the first dynamic arrival, the number of completed operations for each job up to that point is counted, machine buffers are filled, and all necessary information for the simulation's start is defined.

  This includes one of the fundamental elements of the whole code, which is the event list. Within this list are entered all the arrival times of the simulated dynamic jobs, and will then be entered during the evolution of the system the end-of-processing events of a machine when a job is mounted on it.

The pseudoalgorithm 3 shows the main steps of the reset function, the full implementation of the Reset function can be found in *Appendix* A.3.3

Algorithm 3 Env Reset Function

```
1: if mode == 'train' then
2:     Generate dynamic jobs instance with train parameters
3: end if
4: if mode == 'evaluate' then
5:     Generate dynamic jobs instance with evaluation parameters
6: end if
7: Initialize buffers, finished op per job counter, event list
8: Fill the event list with dynamic arrivals
9: Pop the first arrival event with his timestamp
10: Compute the number of finished op per job at timestamp
11: Fill the buffers at timestamp
12: Create the current "machine end operation" events
13: Select the first machine where to take action
14: Get first state
```

3.4.3 Step Function

The step function is the core of the evolution of the environment. It receives incoming action taken by the agent and is responsible for making sure that this is recorded in the environment and that it evolves accordingly. Four main operations take place within it:

The first is the "timing" operation. It is so called since the action taken by the agent corresponds just the operation to be mounted on the machine next, but without any timing indication. This function then takes the operation that is chosen by the agent and places it appropriately in the schedule, assigning the starting and ending time. Clearly in the "single" solving method we have adopted this operation is trivial since the start time corresponds with the instant the decision is made, adding the processing time of the operation we get the end time.[1]

Then we have the *"next"* function. This function performs several critical tasks:

- It updates the buffers when a machine finishes processing a job, either moving the job to its new buffer or removing it from the system.
- It advances the time by moving the event clock forward.

[1]The timing operation is clearly more important in the case of a "non-singular" solving method in which the action returned to me by the agent is the ordered sequence of all operations until the end of the schedule on each machine without any indication of time. At that point my timing fuction must construct the times of the entire schedule from the order of operations set in the action.

- It handles job arrival events by placing the new job in the buffer corresponding to its first operation.

Algorithm 4 Env Step Function

```
 1: if Action $a$ is not None then
 2:     Add action $a$ to the schedule (timing)
 3:     Compute partial reward $\tilde{r}$ for actual state-action
 4:     Create machine end operation event for the current action
 5: end if
 6: if there is any idle machines with non-empty buffer $M_{idle} = [..., M_i, ...]$ then
 7:     Pop first idle machine $M_k = M_{idle}[0]$
 8:     Use MR algorithm to get the state from $M_k$ buffer
 9: else ( so no idle machines $\rightarrow M_{idle} = \emptyset$ )
10:     Evolve the Env to the next event:
11:     if next event type is end operation $O_l$ of the job $J_i$ on machine $M_j$ then
12:         Move job $J_i$ to the next machine buffer or outside the Env if $O_l$ is the
        last operation
13:         Update $n\_finished\_operation(J_i) += 1$
14:         Set $M_j$ as idle
15:     else if next event type is new job $J_n$ arrival then
16:         Add first operation $O_1$ of $J_n$ to the buffer of the respective machine
17:         Set $n\_finished\_operation(J_n) = 0$
18:     end if
19: end if
```

At each step, this function returns the current state, allowing the agent to take appropriate actions.

The full implementation of step function can be found in *Appendix* A.3.2

3.5 Double Deep Q-network (DDQN) Agent

So far, we have explored the structure of the Dynamic Job-Shop Scheduling Problem (DJSP), the definition of all the components of the Markov Decision Process (MDP), and their interactions within the environment, including the implementation details. Now, it is time to introduce the agent responsible for making decisions. Here, I will provide a brief explanation of the reinforcement learning framework used for the decision agent in this project: Double Deep Q-Learning, building upon Deep Q-learning. More detailed information about the agent's training and testing will be provided in the next chapter.

3.5.1 Double Deep Q-learning

Deep Q-learning (DQN) [6] has established itself as a powerful technique in the field of Reinforcement Learning (RL), particularly for handling environments with high-dimensional state spaces as in our case. However, one significant limitation of DQN is its tendency to overestimate action values, a bias that can lead to suboptimal policy learning.

This overestimation arises from DQN's use of the maximum action value as part of its Q-value update rule. Specifically, the same network is used to both select and evaluate the best action, which can result in overly optimistic value estimates. To address this issue, Double Deep Q-learning (DDQN) was introduced in van Hasselt et al. paper [7] of 2016 , incorporating a key modification to decouple action selection from action evaluation, thereby reducing the overestimation bias.

These are the key steps and components of the training of a DDQN :

- **Experience Replay:** The agent stores its experiences, defined as tuples (s_t, a_t, r_t, s_{t+1}), in a replay buffer $\mathcal{D}$. This buffer helps in breaking the correlation between consecutive samples and enables the agent to learn from a diverse set of experiences.

- **Mini-Batch Learning:** From the replay buffer $\mathcal{D}$, we sample a mini-batch of N experiences $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1}^{N}$. This mini-batch is used to perform updates on the network.

- **Target Network:** The DDQN maintains two separate neural networks:

  - The online Q-network $Q(s, a; \theta)$ with parameters θ
  - The target Q-network $Q'(s, a; \theta^-)$ with parameters θ^-

  The target network's parameters θ^- are periodically updated to match the online network's parameters θ to provide stable Q-Value targets.

- **Bellman Equation:** The Bellman equation is used to compute the target Q-Value y_i for each experience in the mini-batch. For DDQN, the target Q-Value is computed as:

  $$y_i = r_i + \gamma Q'(s_{i+1}, \arg\max_a Q(s_{i+1}, a; \theta); \theta^-)$$

  where:

  - r_i is the reward received
  - γ is the discount factor

  - $Q'(s_{i+1}, \arg\max_a Q(s_{i+1}, a; \theta); \theta^-)$ is the value of the next state s_{i+1}, using the action chosen by the online network and evaluated by the target network.

- **Double Q-Learning Update:** In Double Q-Learning, the action a' that maximizes the Q-Value for the next state is selected using the online network, and the value of this action is obtained using the target network:

$$a' = \arg\max_a Q(s_{i+1}, a; \theta)$$

$$y_i = r_i + \gamma Q'(s_{i+1}, a'; \theta^-)$$

- **Loss Function:** The difference between the predicted Q-Values $Q(s_i, a_i; \theta)$ and the target Q-Values y_i is measured using a chosen loss function (in this case is Huber Loss):

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} l(y_i, Q(s_i, a_i; \theta))$$

  where the Huber Loss l is defined as:

$$l(y, Q) = \begin{cases} \frac{1}{2}(y - Q)^2 & \text{for } |y - Q| \leq \delta, \\ \delta |y - Q| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

  Here, δ is a threshold parameter that controls the point at which the loss function transitions from quadratic to linear.

- **Optimization:** To minimize the loss $\mathcal{L}(\theta)$, we use backpropagation to compute the gradients of the loss with respect to the network parameters θ:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$$

  where α is the learning rate, and $\nabla_\theta \mathcal{L}(\theta)$ represents the gradients of the loss function.

The primary distinction between DDQN and DQN is the method used to address overestimation bias. In DQN, the Q-value update uses:

$$y = r + \gamma \max_a Q_{\theta'}(s', a).$$

This formulation can lead to overestimation because the same network is used to determine both the action and its value. In contrast, DDQN's update rule:

$$y = r + \gamma Q_{\theta'}(s', \arg\max_a Q_\theta(s', a)),$$

separates the action selection process (handled by the online network) from the action evaluation process (handled by the target network). This separation reduces the likelihood of overestimation, leading to more accurate Q-value estimates and, consequently, more reliable policy learning.

By mitigating the overestimation bias, DDQN not only enhances the stability of the learning process but also improves the overall performance of the agent across various environments. This modification, while conceptually straightforward, has a profound impact on the effectiveness of the learning algorithm, making DDQN a robust choice for many RL applications.

3.5.2 DDQN Agent implementation

As we saw in the previous section, the Double Deep Q-Learning algorithm works through two neural networks, so-called Q-networks, which are updated during training. The network that is returned at the end of the training and works as the agent, providing action values based on the current state, is referred to as the "Online Network."

The distinction between these two Q-networks is made to enhance the stability and accuracy of the learning process and every few steps, one network's weights are updated with the weights of the other to ensure they remain aligned. So even if we have two separate networks, they work exactly in the same way, thus we need them to be identical.

For this task, it is unnecessary to use overly complex networks that demand extensive training and large datasets. Instead, simpler networks that can still make sensible decisions based on the available data are preferable. An ideal choice in this scenario is the Multi-Layer Perceptron (MLP), the simplest network structure. This structure includes:

- An input normalization and concatenation layer to process the data efficiently. Normalization ensures that each state feature is appropriately weighted. Without normalization, features such as tardiness weight might be undervalued compared to features like slack time, which can be three orders of magnitude higher.
- Six hidden layers of neurons, respectively with dimensions $[64, 48, 48, 36, 24, 12]$
- Each neuron is activated by a *tanh* activation function and the loss function used is the Huber Loss function.

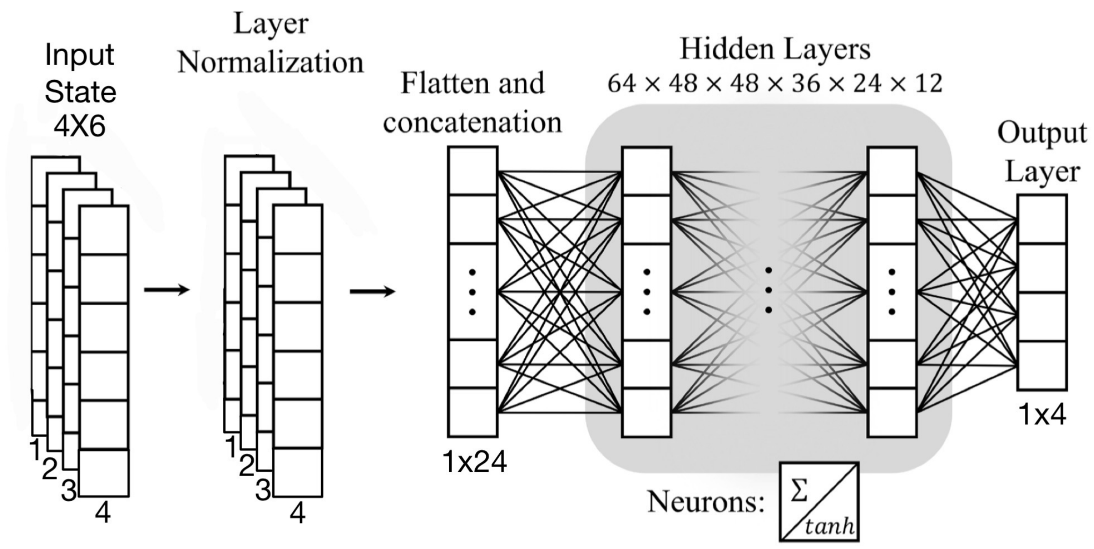

Figure 3.3: Visula representation of the Network. The input state is a 4x6 tensor (x64 that is the dimension of minibatch). Then we have a normalization layer that normalizes within the minibatch and a concatenation layer that transforms the 4x6 tensor into a 1x24. Then an MLP structure with 64x48x48x36x24x12 Hidden Layers and an output of dimension 4 (4 possible jobs to choose).

Chapter 4

Numerical Results

4.1 Agent Training

In the previous chapter we saw how DDQN training takes place theoretically, in this section we will focus on its implementation and numerical results.

Indeed, it is crucial to focus meticulously on all the hyperparameters involved in these kinds of algorithms since they are very sensitive to them. One has to choose the right learning rate, the length of the training episodes and their number, how often to update the weights of the online network (do a training step) and how often to synchronize the target network, after how many steps to update the replay memory with new data, and how big the minibatches extracted from the replay memory should be. Several tests were done, observing the performance of the trailed model and the loss time series to find the best combination of hyperparameters.

The best result was found by generating 1 training episode, consisting of 3600 jobs (each job has between 7 and 10 operations, considering also the steps where there are no possible actions to be taken we get about 36k steps). The first 3600 steps are so-called "warmup" steps, i.e., actions are chosen randomly instead of via the online network, to ensure a fair degree of exploration and to initially fill the replay memory. The replay memory has a maximum size of 1024 and at each training step a minibatch of size 64 is sampled. Every 2 steps an iteration of training is done (so we have about 18k iterations of training). The replay memory is updated with new experiences every 100 steps[1] and the target network is synchronized with

[1]This parameter proved crucial for the success of the training process. In previous sections, I explained that rewards are "asynchronous" and can only be computed after several steps. Setting the number of these steps too low poses a problem: many jobs within a short time frame would not reach completion, making it impossible to determine if they are tardy or not. Consequently, we wouldn't be able to assign the correct reward for different actions, rendering several data points unusable. On the other hand, using too high a value results in the replay memory being updated

the online network every 500 steps. Finally, the best learning rate found is 0.005 which is updated every 4000 steps with a decay rate of 0.9 up to 0.001. The complete set of hyperparameters can be found in table 4.1

Hyperparameter	**Value**
Number of Episodes	1
Number of Jobs per Episode	3600
Number of Operations per Job	7 to 10
Interarrival time of new jobs β	28.5
Operation processing time uniform distribution	Between[1,50]
Total Steps per Episode	~36,000
Warmup Steps	3600
Replay Memory Size	1024
Mini-batch Size	64
Training Steps Interval	Every 2 steps
Training Iterations	~18,000
Replay Memory Update Interval	Every 100 steps
Target Network Update Interval	Every 500 steps
Learning Rate	0.005
Learning Rate Update Interval	Every 4000 steps
Learning Rate Decay Rate	0.9
Final Learning Rate	0.001
Gamma (Discount Factor)	0.95
Epsilon (Initial Exploration Rate)	0.4
Epsilon Decay Rate	0.95
Minimum Epsilon	0.1

Table 4.1: Hyperparameters for DDQN Training

In Fig.4.1 I show the time series of the loss function of the training experiment.

The pseudocode 5 explains the main steps of the DDQN training framework. The full implementation of the training framework is shown and explained in Appendix A.4

too infrequently, causing the agent to overfit on the limited data available, which significantly degrades performance. After several attempts, I found that updating the replay memory every 100 steps provides the best balance and yields optimal results.

Algorithm 5 Duble deep QNetwork learning

```
 1: Initialize environment $E$, online network $Q_\theta$, and target network $Q'_{\theta^-}$
 2: Initialize target network $Q'_{\theta^-}$ with the same parameters as online network $Q_\theta$
 3: Initialize replay memory $\mathcal{D}$
 4: for episode in range(num_episodes) do
 5:     Reset environment $E$
 6:     Clear replay memory $\mathcal{D}$
 7:     Initialize Done to False and step_count to 1
 8:     while not Done do
 9:         if step_count < warmup_steps then
10:             Select random action $a'$
11:         else
12:             Get action with online network $a' = \arg\max_a Q(s, a; \theta)$
13:         end if
14:         Execute action $a'$, observe temporary reward $\tilde{r}$ and next state $s'$
15:         Append state $s$, action $a'$, and temporary reward $\tilde{r}$ to memory $\mathcal{D}$
16:         Set $s = s'$ and increment step_count
17:         if step_count mod memory_step_update == 0 then
18:             Compute final reward for the last memory_step_update state-action pairs
19:             Update the memory $\mathcal{D}$ with the final reward
20:         end if
21:         if step_count mod train_step == 0 then
22:             Sample minibatch of 64 state, action, reward, next state $(s, a, r, s')_{i=1,\dots,64}$ from $\mathcal{D}$
23:             for each sample $i$ in minibatch do
24:                 Choose action $a' = \arg\max_a Q(s', a; \theta)$
25:                 Compute $y_i = r_i + \gamma Q'(s', a'; \theta^-)$
26:                 Compute loss $\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} l(y_i, Q(s_i, a_i; \theta))$
27:                 Perform a training step $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$
28:             end for
29:         end if
30:         if step_count mod update_step == 0 then
31:             Sync weights of $Q_\theta$ and $Q'_{\theta^-}$
32:         end if
33:         if step_count mod update_params_step == 0 then
34:             Update learning rate $\alpha$ and $\epsilon$
35:         end if
36:     end while
37: end for
```

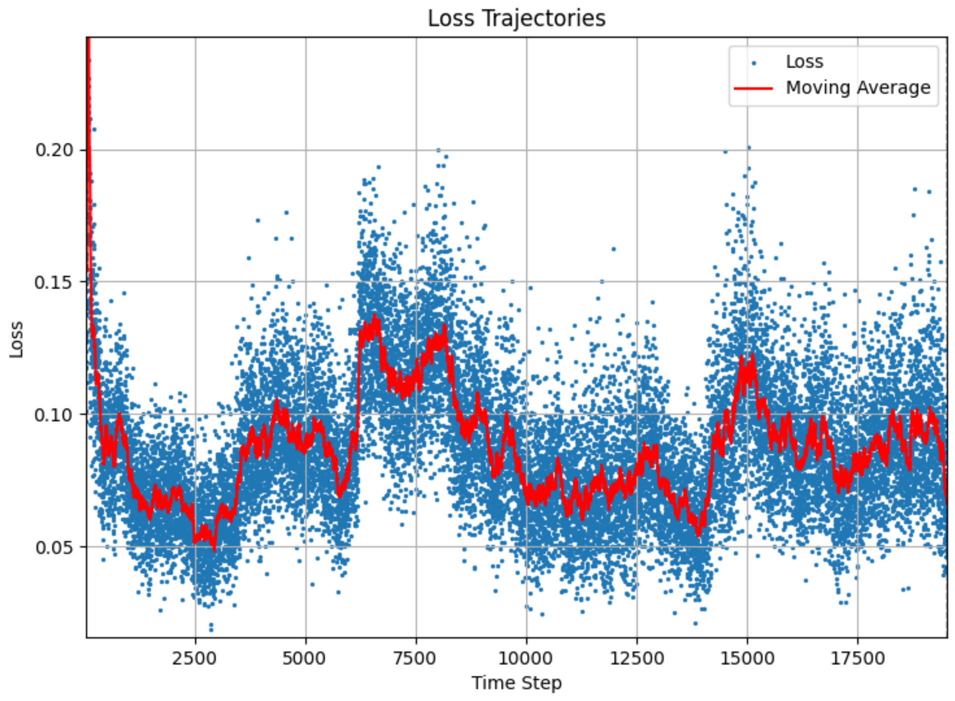

Figure 4.1: This plot shows the time series of the moving average of the loss during training. During the approximately 18,000 training iterations (x-axis), we observed a significant decrease in the loss value. The loss decreased from an initial value of about 0.25 to values around 0.07, marking a reduction of approximately 70 percent. However, occasional jumps in the moving average of the loss are observed. These jumps are likely due to the sampling of state-action-reward data points from the replay memory that contain particularly high (in the sense of particularly low negative) rewards, resulting in large fluctuations in the loss value.

4.2 Agent performances evaluation

Having reached this point we can finally show the performance of the DDQN trained agent in different evaluation settings. As a first thing, we should mention that this agent was trained in a high-utilization system. This means having processing times evenly distributed between [1,50] (with mean value 15.5) and an average dynamic job interarrival time $\beta = 28$. As we already pointed out in Sec.3.3.1 this means an utilization rate of $\sim 90\%$, so an high-utilization rat

So, of course, I will test it under the specified optimal conditions to highlight its strengths. Additionally, I will expose it to various other environments and conditions to assess its robustness. This dual approach not only emphasizes the agent's capabilities but also reveals its weaknesses. It's important to remember that reinforcement learning (RL) is a numerical and model-free algorithm, making it particularly sensitive to changes in data and parameters. Although training such an algorithm can be computationally intensive, achieving the best results necessitates retraining whenever conditions are altered.

The agent will be tested against different dispatching rules agents:

- **FIFO**: the baseline benchmark is clearly the FIFO agent. This agent just choose, each time an action must be taken, the job that entered the system before, i.e. the job with the lowest Job id. Using FIFO is clearly the easiest way to solve a DJSP problem so we need to have better performances respect to it with our custom agent.

- **WINQ**: The second agent chosen follows the least Work In Queue rule. As observed previously, this agent selects the job with the minimum SAM, or "Available time of the Succeeding Machine." For each job in the buffer, the agent identifies the machine required for the next operation and computes the total processing times of both the ongoing operation and those queued in the buffer of that machine. This calculation serves as an indicator of potential congestion at the succeeding machine, which the agent aims to minimize when selecting a job. This approach helps in avoiding bottlenecks and ensuring smoother workflow.

- **PT+WINQ+S**: This is the first of the two Composed Agents developed, referred to in the plots as ComposedAgent1. This agent selects the job that minimizes the sum of three components: the processing time of the job's operation in the buffer, the job's SAM (Available time of the Succeeding Machine explained before), and its slack time.

- **PT+LWKR+S** This is the second of the two Composed Agents developed, referred to in the plots as ComposedAgent2. This agent, similarly to the previous one, selects the job that minimizes the sum of three components: the processing time of the job's operation in the buffer, the job's Work Remaining (the sum of the processing times of all the operations not processed yet), and its slack time.

All evaluation tests involve generating 100 distinct dynamic scenarios for each agent, with each scenario containing 300 jobs. We chose a simulation with a high number of jobs to better highlight the differences between the agents. With a small number of jobs, it's more likely to encounter lucky instances where simpler strategies, like FIFO, outperform more complex agents. However, with a large number of jobs, such fluctuations are less frequent, as only agents consistently making good decisions throughout the simulation achieve favorable results, eliminating the impact of a few fortunate choices. The initial static solution remains consistent across all scenarios, and a seed ensures that each agent is tested under identical conditions. The evaluation scenarios are controlled by two key parameters: the interarrival time (β) and the extreme values of the processing time distribution on the machines. Together, these parameters determine the utilization rate of the simulation. The subsequent sections will present the results.

4.2.1 DDQN in high-utilization systems outperforms Dispatching rules

First, we decided to test the agent in a system as similar as possible to the one in which we trained it. We set the interarrival times to $\beta = 28$ and evenly distributed the processing times between [1,50], achieving a utilization rate of 90%. As shown in plot 4.2, the QNetwork Agent demonstrates superior performance in this configuration, significantly outperforming all other agents. Specifically, over three-quarters of the QNetwork's performance results fall below the median value achieved by FIFO. The median weighted tardiness value for the QNetwork is approximately 115, which represents an improvement of over 30% compared to the average performance of FIFO, as highlighted in Fig 4.3

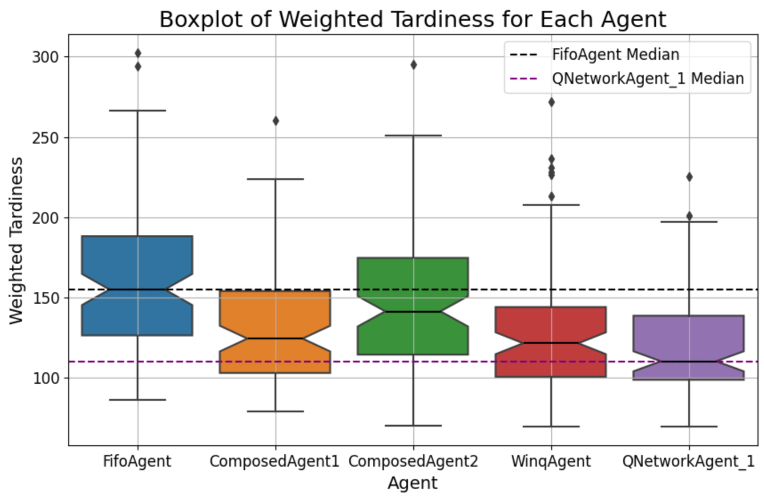

Figure 4.2: The plot shows the differences in the weighted tardiness between all the different agents with 90% utilization rate ($\beta = 28, t : U \sim$ [1,50]). The dashed black line represent the median performance of FIFO Agent that is used as benchmark, the dashed purple line represent the median performance of QNetwork agent that is our result

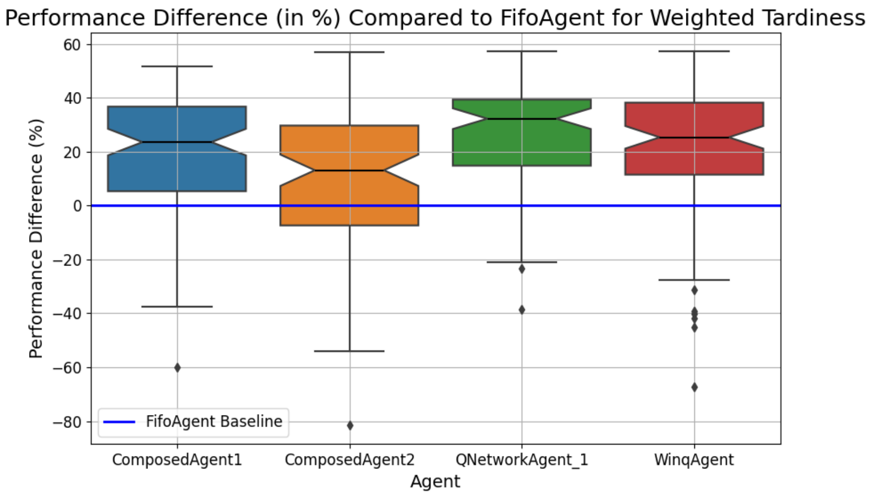

Figure 4.3: This plot, like the previous one, aims to illustrate the performance differences among the agents under the same settings, 90% utilization rate ($\beta = 28, t : U \sim$ [1,50]). However, unlike the last plot, this one presents the performance differences in terms of percentages relative to the benchmark of FIFO's mean weighted tardiness. The blue horizontal line is the FIFO's mean weighted tardiness value used as a baseline.

While it's important to assess the "average" performance to understand an agent's overall effectiveness, analyzing the outcomes of individual instances is also crucial. We seek not only that an agent performs well on average but also that it demonstrates reliability. This means the agent should not produce extremely poor results in any instance, and its performance should remain within a reasonable range without excessive fluctuations. Additionally, a good agent should consistently add value relative to other agents, frequently outperforming them in the same scenarios. Even when it does not come out on top, its performance should not drastically lag behind that of the winning agent.

As can be seen from the plot 4.4 in this setting the QNetwork agent is not only the agent that has the best performance "on average" but is also the one that performs best overall on individual instances since it has the lowest weighted tardiness (it is therefore the "winner") 49 times out of 100. We can also observe an important finding about how FIFO , in this high utilization setting does not result in even a single instance out of 100 in which it has the best performance. So for these settings we can say that QNetwork is for sure the best Agent among the ones chosen for the comparison, followed by Winq. In fact Winq has the best performance in almost 1/3 of the 100 simulations.

Therefore, I have included a last plot 4.5 in this section that compares the

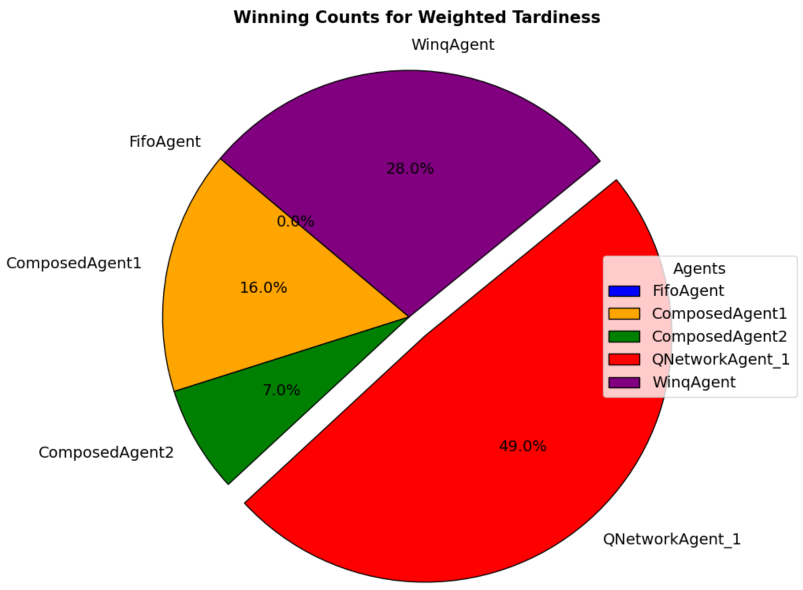

Figure 4.4: The winning counts for weighted tardiness on 100 simulated instances. A seed is fixed before the simulation so each Agent is tested on the same instances. As we can see the best performing Agent is QNetwork, followed by Winq, while Fifo doesn't win in a single instance.

performance of QNetwork and Winq across all individual instances by measuring the difference in their weighted tardiness. Since lower tardiness values are preferable, the plot specifically illustrates the discrepancies in performance between the two agents. It highlights that when QNetwork underperforms relative to Winq, the difference in their tardiness is lower, indicating that QNetwork's performance is closer to that of Winq. Conversely, when Winq underperforms, the difference is more pronounced, showing that Winq tends to be significantly more tardy than QNetwork, showing also some spikes of very bad performances. This visual representation effectively demonstrates the consistency and relative performance of QNetwork compared to Winq.

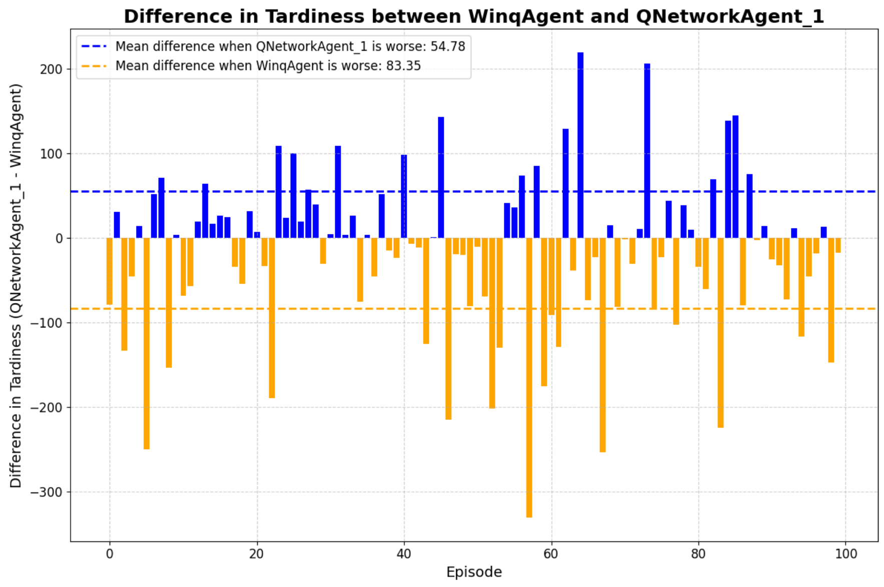

Figure 4.5: This plot displays the performance differences ($QNet - Winq$) between the QNetwork and Winq agents. The blue bars represent episodes where the QNetwork Agent performs worse, meaning it is more tardy than Winq, resulting in a positive difference. Conversely, the orange bars indicate episodes where the Winq Agent performs worse, meaning it is more tardy than QNetwork, leading to a negative difference. As observed, the average difference is approximately 83 when Winq performs worse, and around 54 when QNetwork is less effective. This indicates that even when QNetwork underperforms compared to Winq, the degree of its tardiness is relatively moderate, underscoring its competitive reliability against Winq.

4.2.2 DDQN in low-utilization systems is outperformed by Dispatching rules

As expected, not in all cases the DDQN agent performs better than traditional dispatching rules. In fact, we had already anticipated that we expected a worse performance when the degree of system utilization would drop, that is, in cases where the ratio of the average processing time to the average arrival time drops. We generated an evaluation setting with $\beta = 36.5$ and processing times uniformly distributed between [1,50]. So we are in a utilization setting of $\sim 70\%$, quite lower than the level we utilized to train the agent. As we can see in Fig. 4.6 the QNetwork Agent performs again better than Fifo, but with poor performances if compared with previous setting and wit other dispatching rule Agents.

This is probably due to several factors: first in a system with a lower utilization rate the number of jobs that are buffered when we have to make a decision can often be lower and this leads somewhat to leveling the performance of the various agents since it is less likely to make many different choices. Additionally, the agent was trained in a high-utilization environment characterized by quick reactions and frequent turnovers of operations on the machines. This training environment encouraged the agent to prioritize specific job characteristics that are advantageous in highly congested settings—such as selecting jobs with low work-in-queue to prevent system clogging. However, these strategies may not yield the same benefits in systems where the utilization level is lower, where such selective decision-making is less critical.

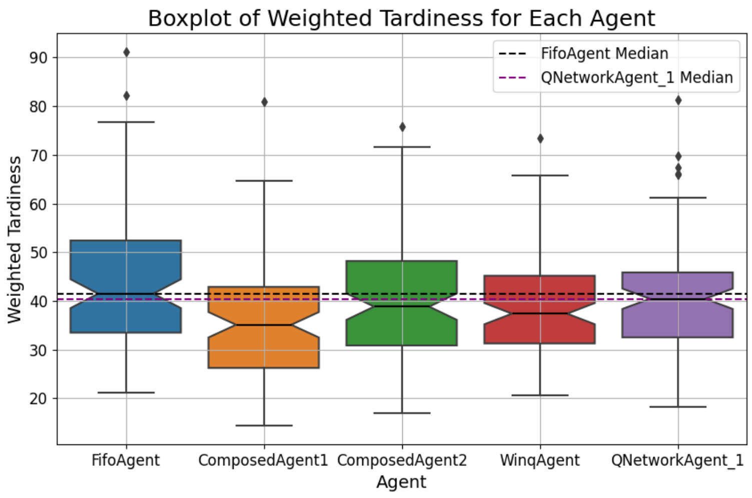

Figure 4.6: This plot, similarly to the previous one, wants to show the differences in performance of the Agents with these settings. differently from the last plot i wanted to show the performance difference in percentage respect to the benchmark of FIFO mean weighted tardiness. Clearly the best result is again the ComposedAgent1 that beats FIFO average performance on 75% of the cases

Another curious observation is that, despite FIFO being by definition a particularly greedy agent, it actually achieves the best performance in 25% of cases according to the plot 4.8. This is because FIFO, as defined in this system, is highly subject to statistical fluctuations: FIFO always chooses the operation of the job that entered the system first, thus favoring the earliest jobs and often causing the later ones to be tardy. Since jobs are generated with random processing times and especially random weights, there can be particularly fortunate scenarios where many jobs towards the end of the simulation have low weights, resulting in very low weighted tardiness, and making FIFO the best performer. On the flip side, just as there are very fortunate scenarios, there are also very unfortunate ones, where

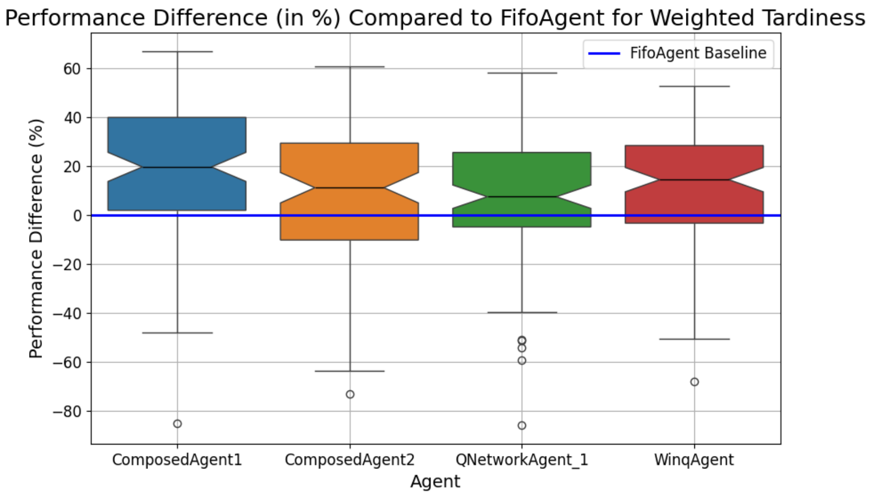

Figure 4.7: This plot shows the differences in performance of the Agents with these 70% utilization settings. We can see how the performance of QNetwork agent are just sligthly better than FIFO because the distribution of results is tighter so avoids very high tardiness results. For sure the best performing agent is ComposedAgent1 with these settings.

the weighted tardiness is extremely high for the opposite reason.

Other agents, including QNetwork, adopt a more balanced approach, which may not always yield the best result but certainly has a tighter distribution of outcomes. They might not always achieve the best solution but avoid disastrous results in unfortunate cases.

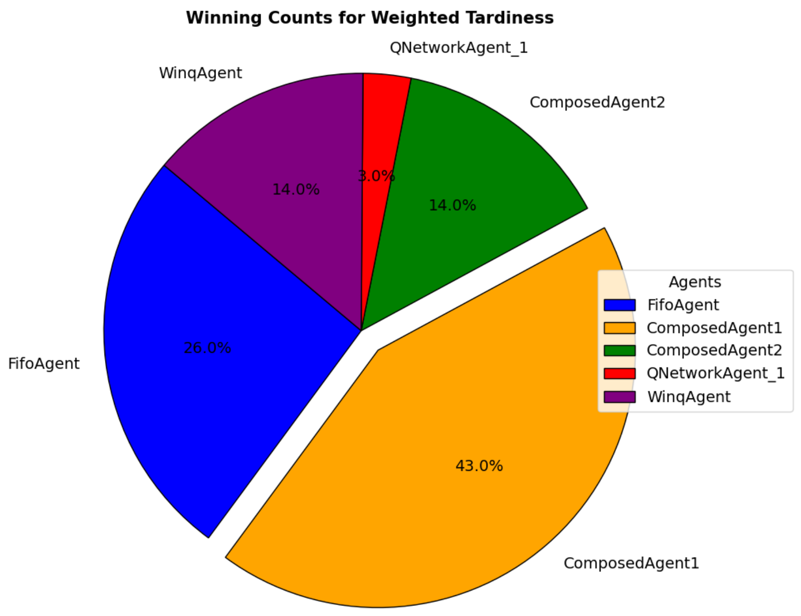

Figure 4.8: The winning counts for weighted tardiness on 100 simulated instances for 70% utilization rate. A seed is fixed before the simulation so each Agent is tested on the same instances. As we can see the best performing Agent is ComposedAgent1, instead in this case QNetwork Agent has quite poor performances in terms of winning counts.

4.2.3 DDQN in extremely high-utilization systems outperforms Dispatching rules

Another very interesting result, in my opinion, is the one concerning systems with an even higher utilization rate. In the title of the section, it has been defined as "extreme" because a value of β was chosen such that the utilization rate exceeds even 100%. In a real setting, this is hard to imagine since a production plant would hardly accept orders at a faster rate than it can process them, certainly incurring in tardiness in deliveries. However, we can observe that with the parameters of $\beta = 22$ and the usual processing times distributed as $U \sim [1{,}50]$, resulting in a utilization rate of 115%, the QNetwork agent performs excellently compared to other heuristic agents. As we can see from fig.4.9, with these settings, all the other agents do not have comparable performances, unlike the setting with a utilization rate of 90% where Winq had performances close to our QNetwork agent.

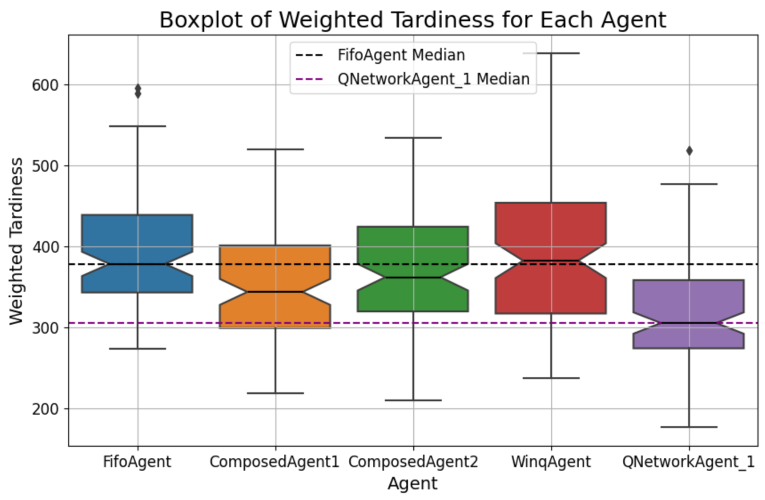

Figure 4.9: The plot shows the differences in the weighted tardiness between all the different agents with 115% utilization rate ($\beta = 22, t : U \sim$ [1,50]). The dashed black line represent the median performance of FIFO Agent that is used as benchmark, the dashed purple line represent the median performance of QNetwork agent that is our result

In particular the QNetwork agent gains more than 20% (see fig 4.10) of performance respect to Fifo, solidly outperforming the other agents.

Even from the pie chart fig.4.11, we can see how, in terms of winning rate, QNetwork is dominant compared to the other agents with a winning rate of as much as 60% in this setting.

Therefore, it is understood from this further test how the QNetwork agent has somehow learned during training to manage highly congested environments, with high utilization frequency and a wider range of operations to choose from in the queues, rather than working in low-utilization systems where it is easily outperformed by heuristics.

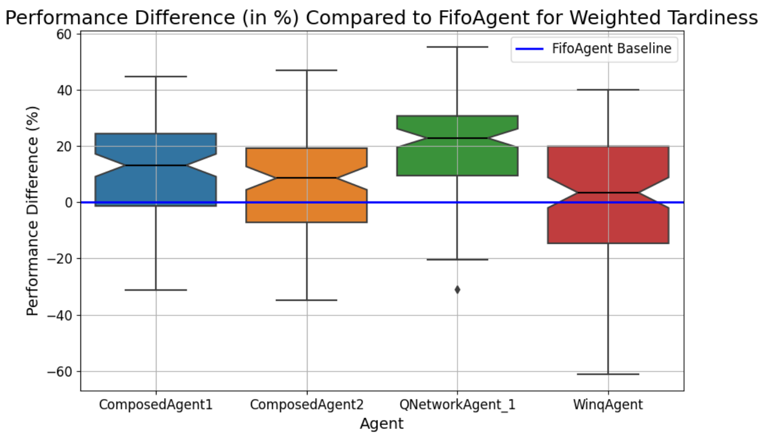

Figure 4.10: This plot, like the previous one, aims to illustrate the performance differences among the agents under the same settings, 115% utilization rate ($\beta = 22, t : U \sim [1{,}50]$). However, unlike the last plot, this one presents the performance differences in terms of percentages relative to the benchmark of FIFO's mean weighted tardiness. The blue horizontal line is the FIFO's mean weighted tardiness value used as a baseline.

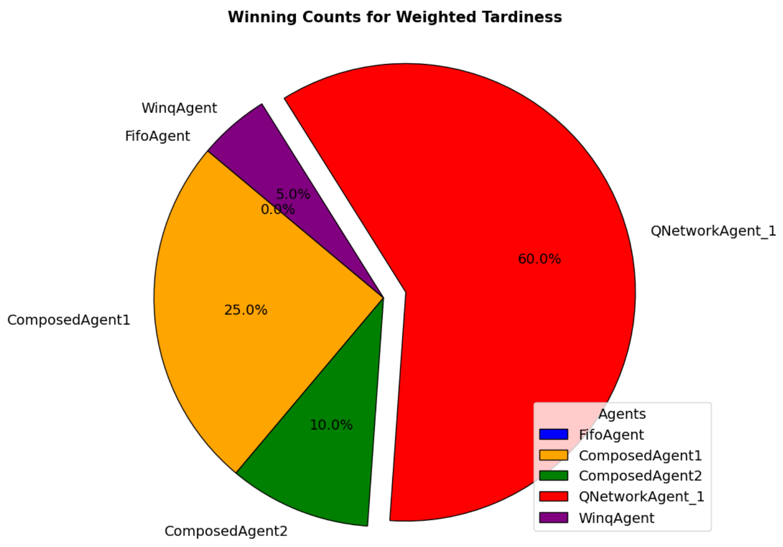

Figure 4.11: The winning counts for weighted tardiness on 100 simulated instances for 115% utilization rate. A seed is fixed before the simulation so each Agent is tested on the same instances. As we can see the best performing Agent is QNetwork with 60%, while Fifo doesn't win in a single instance.

Chapter 5

Conclusions

This thesis work contributes to the field of Operations Research, specifically focusing on dynamic scheduling. The primary contributions are twofold: both in the areas of implementation and modeling, and in the development of a benchmark solution method inspired by the work of Liu et al. [5].

Firstly, this work has enabled a flexible and robust modeling of the problem with a "single operation" approach that can be utilized in future research and for the development of new agents easily integrated into the code. This achievement was made possible through extensive literature research on state-of-the-art methods for modeling and solving the Dynamic Job Shop Scheduling Problem (DJSP), as well as a computational and IT study of the best frameworks and structures to ensure flexibility and robustness.

Secondly, it focused on developing a reinforcement learning solution using Double Deep Q-learning to surpass simple heuristic solutions.As observed from the results, when the DDQN agent is tested in a high-utilization environment, particularly with an utilization rate around 90%, it yields significantly positive outcomes. The agent improves performance in terms of weighted tardiness by an average of 30% compared to FIFO, with a winning rate of nearly 50%. However, it is also noted that slightly more sophisticated dispatching rules (such as WINQ) can deliver good performances, with (clearly) a higher degree of explainability and interpretability compared to an agent based on a neural network.

In contrast, in systems with a lower utilization rate, the performance of the agent predictably declines, and some dispatching rules outperform it. This is because reinforcement learning is a numerical algorithm that allows the agent to prioritize certain state features over others, and when trained in a high-utilization system and tested in a low-utilization one, the learned policy might no longer be optimal.

Probably, seeing how different agents can have the best performance in various situations, future improvements in dynamic scheduling might involve developing systems capable of selecting among different agents based on the state features each time an action needs to be taken, to fully leverage the potential of each individual agent.

In conclusion, this work leaves considerable scope for improvement in the potential development of new agents and methods that can be compared with those defined here and used as benchmarks.

Appendix A

Code Implementation

A.1 MR algorithm implementation

```python
def get_state(self):
    # Check if there is no actual machine in the environment
    if self.env.actual_machine is None:
        # Return a zero matrix state
        state = np.zeros((4, 6), dtype=np.float32)
        return state
    else:
        buffer_operations = []

        # Iterate through the operations in the buffer of the actual machine
        for op in self.env.buffers[self.env.actual_machine]:
            job_id = op['job_id']
            operations = self.env.all_operations[job_id]
            finished_operations = self.env.n_finished_operations_per_job[job_id]
            processing_time = op['processing_time']
            buffer_start_time = op['buffer_starting_time']

            # Calculate the remaining processing time
            remaining_processing_time = sum(op['processing_time'] for op in self.env.all_operations[job_id][finished_operations:])

            # Calculate slack time
            slack_time = self.env.all_jobs[job_id]['due_date'] - self.env.next_event.timestamp - sum(op['processing_time'] for op in operations[finished_operations:])

            # Determine the next machine
            next_machine = operations[finished_operations + 1]['machine'] if finished_operations + 1 < self.env.max_operations[job_id] else None

            if next_machine is not None:
                # Sum the processing times for operations in the next machine's buffer
                sum_time_next_machine = sum(op['processing_time'] for op in self.env.buffers[next_machine])

                # Find the current operation in the next machine's schedule
                next_machine_current_op = self.env.current_schedule[
                    (self.env.current_schedule['machine'] == next_machine) &
                    (self.env.current_schedule['t_start'] <= self.env.next_event.timestamp) &
                    (self.env.current_schedule['t_end'] > self.env.next_event.timestamp)
                ]

                # Calculate the available time for the next machine
                if not next_machine_current_op.empty:
                    available_time_next_machine = sum_time_next_machine + (next_machine_current_op['t_end'].values[0] - self.env.next_event.timestamp)
                else:
                    available_time_next_machine = sum_time_next_machine
            else:
                available_time_next_machine = 0

            # Calculate the current queue time
            current_queue_time = self.env.next_event.timestamp - buffer_start_time

            # Get the weight (tardiness penalty) of the job
            weight = self.env.all_jobs[job_id]['tardiness_penalty']

            # Append the calculated features of the operation to the buffer_operations list
            buffer_operations.append([processing_time, remaining_processing_time, slack_time, available_time_next_machine, current_queue_time, weight])

        # Copy the buffer operations to process them
        buffer_copy = buffer_operations[:]
        state_list = []

        # Find the operation with the minimum value for the selected
    attributes
        for i in range(4):
            min_list = min(buffer_copy, key=lambda x: x[i])
            state_list.append(min_list)  # Save the list
            buffer_copy.remove(min_list)

            # Check if all operations have been processed
            if not buffer_copy:
                buffer_copy = buffer_operations[:]

        # Return the state as a numpy array
        return np.array(state_list, dtype=np.float32)
```

A.2 Reward computation

As extensively explained in the Reward section, the computation of the latter is done asynchronously. At each step the queueing time and slack time corresponding to the job chosen in the action is saved, then every tot steps (depends on the training algorithm, the best number so far seems to be 100) the reward on the last 100 actions is computed. Since we want to punish only tardy jobs we can consider only the actions that correspond to completed jobs or that are not completed but already have a negative slack time (since slack time is a never-increasing function, at most it can stay the same, if the slack time at some point becomes negative it means that certainly the job will be tardy). So of the actions taken in the last 100 steps we must discard the null actions (no jobs in the queue) and the actions of jobs whose tardiness we are not yet certain of.

This is the function to update queueing times and slack times

```
def update_reward_list(self, action):
    # Update the queueing table based on the action taken.
    if action:
        # Extract necessary values to avoid repeated lookups
        job_id = action['job_id']
        current_timestamp = self.env.next_event.timestamp
        buffer_starting_time = action['buffer_starting_time']
        job_info = self.env.all_jobs[job_id]
        job_due_date = job_info['due_date']

        # Calculate remaining processing time
        finished_operations = self.env.n_finished_operations_per_job[
    job_id]
        remaining_operations = self.env.all_operations[job_id][
    finished_operations:]
```

```python
        remaining_processing_time = sum(op['processing_time'] for op in remaining_operations)

        # Calculate queueing and slack times
        queueing_time = current_timestamp - buffer_starting_time
        slack_time = job_due_date - current_timestamp - remaining_processing_time

        # Create a new row as a dictionary
        new_row = {
            'job_id': job_id,
            'queueing_time': queueing_time,
            'slack_time': slack_time,
            'is_tardy': True,
            'is_completed': False
        }
        self.queueing_list.append(new_row)  # Accumulate the new row

        # Check if all operations for this job are finished
        if finished_operations == self.env.max_operations[job_id]-1:
            #print('job completed', job_id)
            # Calculate tardiness
            tardiness = current_timestamp + action['processing_time'] - job_due_date
            for row in self.queueing_list:
                if row['job_id'] == job_id:
                    row['is_completed'] = True
                    if tardiness <= 0:
                        row['is_tardy'] = False
    else:
        new_row = {
            'job_id': None,
            'queueing_time': None,
            'slack_time': None,
            'is_tardy': None,
            'is_completed': None
        }
        self.queueing_list.append(new_row)
```

This is the function used to compute the final rewards

```python
def compute_final_reward(self, queueing_list):
    params = {
        'alpha': 0.8,
        'delta': 150,
        'phi': 50,
    }
    self.set_params(params)
    # Compute the final reward after the episode is done. to
understand where to put this method
    queueing_df=pd.DataFrame(queueing_list)
    #clean the queueing_list
    self.clean_reward_list()
    # set job_id =-1 when there are no jobs in buffer so the action
is None and the reward should not be considered
    queueing_df['job_id']=queueing_df['job_id'].fillna(-1)
    queueing_df.fillna(0, inplace=True)

    # Compute the queueing time of the next operation of the job
    queueing_df['next_queueing_time'] = queueing_df.groupby('job_id')
['queueing_time'].shift(-1)

    # Fill NaN values in next_queueing_time with 0 (or any default
value as per your requirement)
    queueing_df['next_queueing_time'] = queueing_df['
next_queueing_time'].fillna(0)

    # Compute queueing_time_composed
    queueing_df['queueing_time_composed'] = queueing_df['
queueing_time'] * self.params['alpha'] + queueing_df['
next_queueing_time'] * (1-self.params['alpha'])

    # Drop the temporary column
    queueing_df.drop(columns=['next_queueing_time'], inplace=True)

    # Compute the criticality factor
    queueing_df['criticality_factor'] = 1 - (queueing_df['slack_time'
] / (abs(queueing_df['slack_time']) + self.params['delta']))

    #compute the weight factor
    queueing_df['weight_factor'] = 1 + queueing_df['job_id'].map(
        lambda job_id: self.env.all_jobs[job_id]['tardiness_penalty']
 if job_id != -1 else 0)

    # Compute the final reward
    queueing_df['reward'] = -(queueing_df['queueing_time_composed'] *
 queueing_df['criticality_factor'] *
                                    queueing_df['weight_factor'] / self
.params['phi'])**2

    queueing_df['reward'] = queueing_df['reward'].clip(-1, 0)  # Clip
 the reward to [-1, 1]

    # Initialize the is_good column to True
    queueing_df['is_good'] = True

    # compute the is_completed=True for the jobs that are not -1 and have the slack time of the last operation negative
    # Filter the dataframe based on the conditions
    filtered_df = queueing_df[(queueing_df['job_id'] != -1) & (queueing_df['is_completed'] == False)]

    # Select the job_ids of the last rows of each job with negative slack time
    result_filtered = filtered_df[filtered_df['slack_time'] < 0].groupby('job_id').last().reset_index()

    # select the job ids that are tardy also if they are not completed yet
    early_tardy_job_ids = result_filtered['job_id'].tolist()

    # Set is_completed=True for the jobs that are tardy also if they are not completed yet
    queueing_df.loc[queueing_df['job_id'].isin(early_tardy_job_ids), 'is_completed'] = True

    queueing_df.loc[queueing_df['job_id'] == -1, 'is_good'] = False
    queueing_df.loc[queueing_df['is_completed'] == False, 'is_good'] = False

    # set to 0 the reward of the non-tardy jobs
    queueing_df.loc[queueing_df['is_tardy'] == False, 'reward'] = 0

    return queueing_df
```

A.3 Environment implementation

A.3.1 Init Function

```
def __init__(self, instance_name):
    super().__init__()

    main_folder = os.path.join(
        '.', 'data',
        instance_name
    )

    # read the instance settings, static instance and dynamic instance
    # the dataframe variants ('df_') of some of the data are used only for solving the static instance
```

```
        self.n_machines, self.n_sku, self.n_static_jobs, self.n_dynamic_jobs, self.setup, df_setup = read_instance_settings(main_folder)
        self.static_jobs, df_static_jobs, self.static_operations, self.machines_initial_state = read_instance_static_jobs(main_folder)
        subfolder_name = 'subtest'
        self.dynamic_jobs, df_dynamic_jobs, self.dynamic_operations = read_instance_dynamic_jobs(main_folder, subfolder_name)

        self.settings= {
            'n_machines': self.n_machines,
            'n_sku': self.n_sku,
            'n_static_jobs': self.n_static_jobs,
        }
        #read train/evaluation settings in case of trainig/evaluation
        with open('data_interfaces/dynamic_jobs_training_settings.json', 'r') as file:
            self.dynamic_jobs_training_setting = json.load(file)
        with open('data_interfaces/dynamic_jobs_evaluate_settings.json', 'r') as file:
            self.dynamic_jobs_evaluate_setting = json.load(file)

        # create a unique set of all jobs (both static and dynamic)
        self.all_jobs = {**self.static_jobs,**self.dynamic_jobs}
        self.all_operations = self.static_operations+self.dynamic_operations

        # creates an instance of the static data to solve the initial problem
        self.inst = InstanceJobShopSetUp(self.n_static_jobs, self.n_machines, self.machines_initial_state, df_static_jobs, df_setup, self.static_operations)

        solver = SolveJITJSSST(self.inst)
        solver.solve(
            verbose = True,
            time_limit = 40,
        )
        _, self.static_schedule, self.setups, _ = solver.get_solution()
        self.current_schedule = self.static_schedule

        # instantiate the EventManager
        self.envmanager = SingleEnvManager(self)
        self.timing = SingleTiming(self)
        self.reward_function = TardinessReward()

        self.action_space = gym.spaces.Discrete(4)  # 4 possible actions
        self.observation_space = gym.spaces.Box(low=-np.inf, high=np.inf, shape=(4, 6), dtype=np.float32)  # state shape 4x6
```

A.3.2 Step Function

```
def step(self, action):
    # the action defines the new current schedule

     self.current_schedule = self.timing.timing_schedule(action)

    reward_value = self.reward_function.compute(action)

    # move forward the time using envmanager.next
    # in order to have the same signature of the next function i
    upload all the env variables inside next
    obs=self.envmanager.next(action)

    #reward_value = 0
    #done = len(self.events) == 0
    done = self.n_finished_operations_per_job == self.max_operations

     info = {}
     return obs, reward_value, done, info
```

single timing function

```
def timing_schedule(self, action):
    # Returns the timing schedule of the operations chosen by the
    agent when the agent choses a single action.
    # Support variable
     action_df = pd.DataFrame()

     if action is not None:
         #for operation in action:
         # create the new row of the schedule with the operation
    chosen
         machine=action['machine']
         job=action['job_id']
         op=(self.env.n_finished_operations_per_job[action['job_id']],
     action['job_id'])
         t_start=self.env.next_event.timestamp
         t_end=t_start+action['processing_time']
         new_op_schedule = pd.DataFrame({'machine': [machine], 'order'
    : [op[1]], 'op': [op], 't_start': [t_start], 't_end': [t_end], '
    earliness': 0, 'tardiness': 0})
         action_df=pd.concat([action_df, new_op_schedule])
     else:
```

```python
        timed_action=self.env.current_schedule
        return timed_action


    # if the action_df is not empty insert the new row in the schedule that will be the final action of the agent
    timed_action=pd.concat([self.env.current_schedule,action_df]).sort_values(['machine','t_start']) #TODO: there is for sure a best way instead of append+sort

    return timed_action
```

single next function

```python
def next(self, action):
    '''Process the next event based on the given action.'''

    # Create the end machine event for the action if provided
    if action:
        self.env.events.append(Event(action['job_id'], self.env.next_event.timestamp + action['processing_time'], 'machine_end_operation'))
    self.env.events.sort()
    #print(self.env.events)

    # Handle the next available machine or process the next event
    if self.env.free_machines_with_buffer:
        self.env.actual_machine = self.env.free_machines_with_buffer.pop(0)
    else:
        # Process all events with the same timestamp
        if self.env.events:
            first_event_timestamp = self.env.events[0].timestamp

            while self.env.events and self.env.events[0].timestamp == first_event_timestamp:
                self.env.next_event = self.env.events.pop(0)

                if self.env.next_event.type == 'machine_end_operation':
                    self._handle_machine_end_operation()
                elif self.env.next_event.type == 'job_arrival':
                    self._handle_job_arrival()

                self._buffer_next_operation()

        # Update the list of free machines with buffer
        self._update_free_machines()

        if self.env.free_machines_with_buffer:
            self.env.actual_machine = self.env.
   free_machines_with_buffer.pop(0)
        else:
            self.env.actual_machine = None

    next_state=self.get_state()
    return next_state


def _handle_machine_end_operation(self):
    job_id = self.env.next_event.job_id
    machine_ended = self.env.all_operations[job_id][self.env.
   n_finished_operations_per_job[job_id]]['machine']

    # Remove the completed operation from the machine's buffer
    self.env.buffers[machine_ended] = [op for op in self.env.buffers[
   machine_ended] if op['job_id'] != job_id]

    # Update the counter of finished operations
    self.env.n_finished_operations_per_job[job_id] += 1

    # Set the machine as free
    # self.env.free_machines_with_buffer.append(machine_ended)  #
   Uncomment if required to mark the machine as free

def _handle_job_arrival(self):
    self.env.id_next_job += 1
    self.env.n_finished_operations_per_job[self.env.next_event.job_id
   ] = 0

def _buffer_next_operation(self):
    job_id = self.env.next_event.job_id
    operations = self.env.all_operations[job_id]
    finished_operations = self.env.n_finished_operations_per_job[
   job_id]

    if finished_operations < len(operations):
        next_operation = operations[finished_operations]

        # Prepare the next operation details
        operation_details = {
            **next_operation,
            'job_id': job_id,
            'buffer_starting_time': self.env.next_event.timestamp,
            'remaining_processing_time': sum(op['processing_time']
    for op in operations[finished_operations:]),
            # 'slack_time': self.env.all_jobs[job_id]['due_date'] -
    self.env.next_event.timestamp - sum(op['processing_time'] for op
    in operations[finished_operations:]),
            # 'work_in_queue': sum(op['processing_time'] for op in
    self.env.buffers[self.env.all_operations[job_id][
    finished_operations+1]['machine']])
        }

        # Add the operation to the buffer of the respective machine
        machine = next_operation['machine']
        self.env.buffers[machine].append(operation_details)

def _update_free_machines(self):
    # Update the list of free machines with buffer
    working_machines = np.unique(self.env.current_schedule[
        (self.env.current_schedule['t_start'] <= self.env.next_event.
    timestamp) &
        (self.env.current_schedule['t_end'] > self.env.next_event.
    timestamp)
    ]['machine'])
    free_machines = list(np.setdiff1d(np.arange(self.env.n_machines),
    working_machines))
    self.env.free_machines_with_buffer = [machine for machine in
    free_machines if self.env.buffers[machine]]
```

A.3.3 Reset Function

```
    def reset(self, seed=None, mode=None):
        # if mode is training each reset creates a new instance of
    the dynamic problem, keeping the same static instance

        if mode == 'train':

            self.dynamic_jobs, df_dynamic_jobs, self.
    dynamic_operations = simulate_instance_dynamic_jobs_no_save(self.
    dynamic_jobs_training_setting ,self.settings)
            self.all_jobs = {**self.static_jobs,**self.dynamic_jobs}
            self.all_operations = self.static_operations+self.
    dynamic_operations

        elif mode == 'evaluate':
            self.dynamic_jobs, df_dynamic_jobs, self.
    dynamic_operations = simulate_instance_dynamic_jobs_no_save(self.
    dynamic_jobs_evaluate_setting ,self.settings)
```

```
            self.all_jobs = {**self.static_jobs,**self.dynamic_jobs}
            self.all_operations = self.static_operations+self.dynamic_operations

        # dictionary: for every key (machine id), specifies the list of operations in the queue
        self.queues = {}

        # dictionary: for every key (job id), specifies the number of operations already completed
        self.n_finished_operations_per_job = {}

        # list of events happening in the future (every event is an object of class Event)
        self.events = []

        # define the buffers of the machines: for every machine, the list of operations that are waiting in the buffer
        self.buffers = {machine : [] for machine in range(self.n_machines)}

        # for every job in the static instance set the finished operations to 0
        for job_id in self.static_jobs:
            self.n_finished_operations_per_job[job_id] = 0

        self.reward_function.initialize(self)

        # id of next job arriving
        self.id_next_job = len(self.static_jobs) + 1

        for job_id in self.dynamic_jobs:
            # for every job in the dynamic instance it defines the event of its future arrival
            self.events.append(Event(job_id,self.dynamic_jobs[job_id]['arrival_date'], 'job_arrival'))
        # sorts event based on their order of happening
        self.events.sort()

        self.next_event = self.events.pop(0) #the first event is always a job arrival

        #add the counter of finished operations equal to 0 for the new job
        self.n_finished_operations_per_job[self.next_event.job_id] = 0

        self.current_schedule = self.static_schedule

        # set the current schedule as the schedule before the first event
        self.current_schedule = self.current_schedule[self.current_schedule['t_start']< self.next_event.timestamp]

        #set the intial finished operations indices for every job
        for job_id in self.n_finished_operations_per_job:
            self.n_finished_operations_per_job[job_id] = self.current_schedule[(self.current_schedule['t_end'] <= self.next_event.timestamp) & (self.current_schedule['order']==job_id)]['op'].count()

            if self.n_finished_operations_per_job[job_id] < len(self.all_operations[job_id]):

                operations = self.all_operations[job_id]
                finished_operations = self.n_finished_operations_per_job[job_id]

                next_operation = operations[finished_operations]

                # Prepare the next operation details
                operation_details = {
                    **next_operation,
                    'job_id': job_id,
                    'buffer_starting_time': self.next_event.timestamp,
                    'remaining_processing_time': sum(op['processing_time'] for op in operations[finished_operations:]),
                }

                # Add the operation to the buffer of the respective machine
                machine = next_operation['machine']
                self.buffers[machine].append(operation_details)

        #set free machines in the first time instant
        working_machines = np.unique(self.current_schedule[(self.current_schedule['t_start'] <= self.next_event.timestamp)
                                                            & (self.current_schedule['t_end'] > self.next_event.timestamp)]['machine'])
        free_machines = list(np.setdiff1d(np.arange(self.n_machines), working_machines))

        self.free_machines_with_buffer = [machine for machine in free_machines if self.buffers[machine]]

        # if there are free machines with buffer, set the actual machine as the first free machine with buffer
        if self.free_machines_with_buffer:
            self.actual_machine = self.free_machines_with_buffer.pop(0)
        else:
            self.actual_machine = None

        #create initial machine_end operation events for the machines working at first time instant
        for machine in working_machines:
            working_operation=self.current_schedule[(self.current_schedule['machine']==machine) & (self.current_schedule['t_end'] > self.next_event.timestamp)]
            self.events.append(Event(working_operation['order'].values[0],working_operation['t_end'].values[0], 'machine_end_operation'))

        # this stopping criteria just takes the max number of operations for every job and checks if the number of finished operations is equal to that
        self.max_operations ={}
        for i in self.all_jobs:
            self.max_operations[i] = len(self.all_operations[i])

        obs=self.envmanager.get_state()

        return obs
```

A.4 DDQN Training script

```

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class QNetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(QNetwork, self).__init__()

        self.input_size = input_size
        self.output_size = output_size
        self.flattened_input_size = torch.tensor(self.input_size).prod()
        layer_1 = 64
        layer_2 = 48
        layer_3 = 48
        layer_4 = 36
        layer_5 = 24
        layer_6 = 12
        #TODO: metti questi parametri come input

        self.norm_layer = nn.Sequential(
            nn.LayerNorm(self.input_size),
            nn.Flatten()
        )

        self.FC_layers = nn.Sequential(
            nn.Linear(self.flattened_input_size, layer_1),
            nn.Tanh(),
            nn.Linear(layer_1, layer_2),
            nn.Tanh(),
            nn.Linear(layer_2, layer_3),
            nn.Tanh(),
            nn.Linear(layer_3, layer_4),
            nn.Tanh(),
            nn.Linear(layer_4, layer_5),
            nn.Tanh(),
            nn.Linear(layer_5, layer_6),
            nn.Tanh(),
            nn.Linear(layer_6, output_size)
        )

        self.network = nn.ModuleList([self.norm_layer, self.FC_layers])

    def forward(self, inp):
        x1 = self.norm_layer(inp)
        x1 = self.FC_layers(x1)
        return x1

class Memory:
    def __init__(self, memory_len):
        self.rewards = collections.deque(maxlen=memory_len)
        self.state = collections.deque(maxlen=memory_len)
        self.action = collections.deque(maxlen=memory_len)
        self.is_done = collections.deque(maxlen=memory_len)
         # Temporary storage for state, action until reward is available
        self.temp_state = []
        self.temp_action = []
        self.temp_is_done = []
        #self.temp_reward = []

    def append_temp(self, state, action, reward, done, real_action):
        # Append state, action, done flag to temporary storage

        # the state should be appended as a tensor 1x4x6 for later computation
        self.temp_state.append(torch.tensor(state).unsqueeze(0))

        # action and is done are appended to lists
        self.temp_action.append(action)
        self.temp_is_done.append(done)

        # since in this project the reward is asyncronous to the state-action, i cannot append the reward at each step as i do
        # with the state, action, done flag. The reward variable here is a list of rewards that is updated at each step
        self.temp_reward = reward
    def update(self, final_reward_df):

        # get the indexes of good state-action pairs using the final_reward_df
        # the good indexes are the indexes of the state-action pairs where the action is not None (no job available to be chosen)
        # and the reward can be computed since we know if the job is tardy or not
        good_indexes = final_reward_df[final_reward_df['is_good'] == True].index

        # extend state, action, done flag lists with temporary storage, pay attention that temp_state is a list of tensors
        good_action=[self.temp_action[i] for i in good_indexes]
        good_state=[self.temp_state[i] for i in good_indexes]
        good_is_done=[self.temp_is_done[i] for i in good_indexes]

        self.state.extend(good_state)
        self.action.extend(good_action)
        self.is_done.extend(good_is_done)

        # append the final reward to the rewards list
        temp_reward=final_reward_df.loc[good_indexes]['reward'].tolist()
        self.rewards.extend(temp_reward)

        #clear temporary storage
        self.temp_state = []
        self.temp_action = []
        self.temp_is_done = []


    def sample(self, batch_size):

        # store the lenght of the replay memory in n
        n = len(self.is_done)

        # select a number of random indexes equal to the batch size from the replay memory
        idx = random.sample(range(0, n-1), batch_size)

        #sampled_states is a tensor batch_size x 4 x 6 (state dimension is 4x6)
        sampled_states = torch.cat([self.state[i] for i in idx], dim=0).to(device)

        #next_states is a tensor batch_size x 4 x 6 (state dimension is 4x6)
        next_states = torch.cat([self.state[i+1] for i in idx], dim=0).to(device)

        #actions is a tensor batch_size x 1
        actions = torch.LongTensor([self.action[i] for i in idx]).to(device)

        #rewards is a tensor batch_size x 1
        rewards = torch.Tensor([self.rewards[i] for i in idx]).to(device)

        #dones is a tensor batch_size x 1
        dones = torch.Tensor([self.is_done[i] for i in idx]).to(device)

        return sampled_states, actions, next_states, rewards, dones

    def reset(self):
        self.rewards.clear()
        self.state.clear()
        self.action.clear()
        self.is_done.clear()
        self.temp_state = []
        self.temp_action = []
        self.temp_is_done = []


def get_action(model, env, state, eps):

    # This function is called in the forward process to collect an action given the state
    # the state has to be a tensor 1x4x6
    state = torch.tensor(state,dtype=torch.float).unsqueeze(0).to(device)

```

```python
    # the actual machine is the machine where we are going to take the action. If the machine is None, it
    # means a new job has arrived on a busy machine
    if (env.actual_machine is not None):
        machine_buffer=env.buffers[env.actual_machine]

        # if the machine buffer is not empty, we can take an action
        if (len(machine_buffer) > 0):
            with torch.no_grad():
                values = model(state)
            if random.random() <= eps:
                action = np.random.randint(0, env.action_space.n)
            else:
                action = np.argmax(values.cpu().numpy())
            #convert action index to real action
            chosen_op = convert_real_action(state, action, machine_buffer)
        else:
            action=None
            chosen_op = None
    else :
        action=None
        chosen_op = None
    # the function returns action (tensor needed for algorithm training)
    # and real_action (conversion of the abstract action into real data that are used by the env to evolve)
    return action, chosen_op


def get_random_action( env, state):

    # This function is called in the warm-up process to collect a random action given the state

    state = torch.tensor(state,dtype=torch.float).unsqueeze(0).to(device)

    if (env.actual_machine is not None):
        machine_buffer=env.buffers[env.actual_machine]
        if (len(machine_buffer) > 0):
            action = np.random.randint(0, env.action_space.n)
            #convert action index to rel action
            chosen_op = convert_real_action(state, action, machine_buffer)
        else:
            action=None
            chosen_op = None
    else :
            action=None
            chosen_op = None
        return action, chosen_op


def convert_real_action(state,action, machine_buffer):
        # This function is called to convert the action index to the real action data (job_id, operation_id, machine_id,...)
        # in order to be used in the environment step function

        # Extract the criteria values from the tensor
        processing_time_criteria = state.squeeze(0)[action][0].item()
        remaining_processing_time_criteria = state.squeeze(0)[action][1].item()

        # Find the matching dictionary
        matching_dict = None
        for job in machine_buffer:
            if job['processing_time'] == processing_time_criteria and job['remaining_processing_time'] == remaining_processing_time_criteria:
                matching_dict = job
                return matching_dict

def train(batch_size, current, target, optim, memory, gamma):

    # Sample a batch of experiences from the replay memory with dimensions specified in sample function
    states, actions, next_states, rewards, is_done = memory.sample(batch_size)

    #q_values of the batch of states-actions so the output is a tensor batch_size x action_space (64x4)
    q_values = current(states)

    #next_q_values of the batch of next_states so the output is a tensor batch_size x action_space (64x4)
    next_q_values = current(next_states)

    #next_q_state_values of the batch of next_states so the output is a tensor batch_size x action_space (64x4)
    next_q_state_values = target(next_states)

    # q_value of each state when you choose the action taken from the replay memory for each state so the output is a tensor batch_size (64)
    q_value = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)

    # next_q_value of each state when you choose the action that maximizes the q_value of the next state so the output is a tensor batch_size (64)
    next_q_value = next_q_state_values.gather(1, torch.max(next_q_values, 1)[1].unsqueeze(1)).squeeze(1)

    # expected_q_value is the target value for the q_value so the output is a tensor batch_size (64)
    expected_q_value = rewards + gamma * next_q_value * (1 - is_done)

    # Compute the loss between q_value and expected_q_value and update the Q-network
    loss = F.mse_loss(q_value, expected_q_value.detach())
    optim.zero_grad()
    loss.backward()
    optim.step()

    return loss.item()

def evaluate(Qmodel, env, repeats):
    #implemented function to evaluate performances during training process

def update_parameters(current_model, target_model):
    target_model.load_state_dict(current_model.state_dict())

def save_data(...):
    #implemented function to save results

def main(gamma=0.95, lr=0.005, eps=0.4, eps_decay=0.95, eps_min=0.1, update_step=500, batch_size=64, update_eps_steps=5000,
         train_step=4, memory_step_update=100, num_episodes=2, seed=100, max_memory_size=1024, lr_gamma=0.9, lr_step=5000, evaluate=True, measure_step=3000,
         measure_repeats=10, warmup_steps=3600, print_step=50, checkpoint_dir='save_results/checkpoints_8', final_save_dir='save_results/save_8',
         save=True, save_checkpoint=True, ):
    """
    Main function for training a DQN agent on the specified environment.
    """

    # Create checkpoint directory to save intermediate results
    if not os.path.exists(checkpoint_dir):
        os.makedirs(checkpoint_dir)

    # Initialize environment and set seeds for reproducibility
    #env = gym.make(env_name)
    env=DynamicPlant("test_instance5")
    evaluation_env = DynamicPlant("test_instance5")
    #torch.manual_seed(seed)
    #env.seed(seed)

    # Initialize Q-networks
    Q_1 = QNetwork(input_size=env.observation_space.shape, output_size=env.action_space.n).to(device)
    Q_2 = QNetwork(input_size=env.observation_space.shape, output_size=env.action_space.n).to(device)

    # Copy parameters from Q_1 to Q_2 and freeze Q_2 parameters
    update_parameters(Q_1, Q_2)
    for param in Q_2.parameters():
        param.requires_grad = False

    # Initialize optimizer and learning rate scheduler
    optimizer = optim.SGD(Q_1.parameters(), lr=lr, momentum=0.9)
    scheduler = StepLR(optimizer, step_size=lr_step, gamma=lr_gamma)

    # Initialize replay memory
    memory = Memory(max_memory_size)

    # List to store performance measurements
    performance = []

    loss_trajectories = {}

    step_count_across_episodes = 0

    # Main training loop
    for episode in range(1,num_episodes+1):
        #Measure performance with an evaluation after each episode (measure_step=1), since we use long episodes

        # Initialize state
        #here state is a numpy array 4x6
        state = env.reset(mode='train')
        print('reset')
        memory.reset()

        '''append state,action,done to temporary memory -
        later to be updated in the replay memory since the reward is asyncronous to the state,action,done.
        you cannot update the replay memory with the reward until the reward is available.'''

        done = False
```

```python
        # Initialize step count to 1 (not 0) beacuse later we will call step_count % memory_step_update == 0 and i don't want it to be true at the first step
        step_count = 1

        episode_losses = []

        # Run episode
        while not done:

            # in order to fill the replay memory with an initial set of state-action pairs, we select random actions for the first warmup_steps
            if episode==1 and step_count < warmup_steps:
                action,real_action = get_random_action(env, state)  # Select random action during warm-up period
            else:
                #print('warmup finished')
                action,real_action = get_action(Q_2, env, state, eps)  # Select action using epsilon-greedy policy, real_action is the conversion of the action index to the real action
            next_state, reward, done, _ = env.step(real_action)  # Take action in the environment

            # Update memory
            memory.append_temp(state, action, reward, done, real_action)

            state = next_state

            # Update replay memory each memory_step_update steps
            if step_count % memory_step_update == 0:
                final_reward_df = env.reward_function.compute_final_reward(memory.temp_reward)

                memory.update(final_reward_df)

            # Train Q-networks each 5 steps
            if step_count % train_step == 0 and step_count > warmup_steps:
                #print('training_step')
                loss=train(batch_size, Q_1, Q_2, optimizer, memory, gamma)
                episode_losses.append(loss)

            # print loss each print_step steps
            if step_count % print_step == 0 and step_count > warmup_steps:
                print("Step: ", step_count)
                print("Loss: ", loss)

            # Update Q_2 with Q_1 parameters each 250 steps
            if step_count % update_step == 0:
                update_parameters(Q_1, Q_2)

            step_count += 1
            step_count_across_episodes += 1

            # Update learning rate and epsilon each episode
            if step_count > warmup_steps:
                scheduler.step()
                if (step_count_across_episodes+1) % update_eps_steps == 0:
                    eps = max(eps * eps_decay, eps_min)
                    print(f"Updated learning rate to {scheduler.get_last_lr()[0]} and epsilon to {eps} at step {step_count_across_episodes}")

            # Measure performance
            if (step_count % measure_step == 0): #and episode >= min_episodes:
                #measure performances

        # Save the loss trajectory for the current episode
        loss_trajectories[episode] = episode_losses

        # # Update learning rate and epsilon
        # scheduler.step()
        # eps = max(eps * eps_decay, eps_min)

         # Save checkpoint at the end of each episode
        if save_checkpoint: ...


    # Save model parameters, performance data, and loss trajectories
    if save:...
```

Bibliography

[1] Ali Fırat İnal, Çağrı Sel, Adnan Aktepe, Ahmet Kürşad Türker, and Süleyman Ersöz. «A Multi-Agent Reinforcement Learning Approach to the Dynamic Job Shop Scheduling Problem». In: *Sustainability* 15.8262 (2023) (cit. on p. 6).

[2] Libing Wang, Xin Hu, Yin Wang, Sujie Xu, Shijun Ma, Kexin Yang, Zhijun Liu, and Weidong Wang. «Dynamic job-shop scheduling in smart manufacturing using deep reinforcement learning». In: *Computer Networks* 190 (2021), p. 107969 (cit. on p. 7).

[3] Ziqing Wang and Wenzhu Liao. «Job Shop Scheduling Problem Using Proximal Policy Optimization». In: *2023 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*. IEEE. 2023, pp. 1517–1521 (cit. on pp. 7, 12).

[4] Xinquan Wu and Xuefeng Yan. «A spatial pyramid pooling-based deep reinforcement learning model for dynamic job-shop scheduling problem». In: *Computers & Operations Research* 160 (2023), p. 106401 (cit. on p. 7).

[5] Renke Liu, Rajesh Piplani, and Carlos Toro. «A deep multi-agent reinforcement learning approach to solve dynamic job shop scheduling problem». In: *Computers & Operations Research* 159 (2023), p. 106294 (cit. on pp. 8, 12, 13, 15, 18, 41).

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. «Playing atari with deep reinforcement learning». In: *arXiv preprint arXiv:1312.5602* (2013) (cit. on p. 23).

[7] Hado Van Hasselt, Arthur Guez, and David Silver. «Deep reinforcement learning with double q-learning». In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016 (cit. on p. 23).