UiA University of Agder

Master

Implementation of Reinforcement learning to solve Job-shop Scheduling Problem

RESHMA MAHARJAN

SUPERVISORS
Per-Arne Andersen and Lei Jiao

University of Agder, 2024
Faculty of Engineering and Science
Department of Information and Communication Technology

Obligatorisk gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

1.	Vi erklærer herved at vår besvarelse er vårt eget arbeid, og at vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.	Ja
2.	**Vi erklærer videre at denne besvarelsen:** • Ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. • Ikke refererer til andres arbeid uten at det er oppgitt. • Ikke refererer til eget tidligere arbeid uten at det er oppgitt. • Har alle referansene oppgitt i litteraturlisten. • Ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	Ja
3.	Vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§ 31.	Ja
4.	Vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.	Ja
5.	Vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk.	Ja
6.	Vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider.	Ja
7.	Vi har i flertall blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt. Ordinært vurderes alle deltakere i prosjektet samlet.	Nei

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).
Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Per-Arne Andersen, and co-supervisor, Lei Jiao, for their guidance and support throughout this research journey. Their expertise and wisdom have been invaluable in shaping my understanding of the Job Shop Scheduling Problem and Reinforcement Learning.

I am deeply grateful to my family and friends for their unwavering support and love. Their sacrifices and patience have enabled me to pursue my academic goals, and I am forever grateful.

I would like to thank my colleagues and friends at the University of Agder for creating a stimulating and supportive environment that fostered my growth and learning.

This thesis would not have been possible without the collective efforts of these individuals, and I am grateful for their contributions.

Additionally, I want to acknowledge the invaluable assistance provided by ChatGPT, whose insightful suggestions and contributions have enriched this thesis.

Reshma Maharjan,
May 24, 2024,
Grimstad

Abstract

The Job Shop Scheduling Problem (JSSP) consists of allocating various tasks to distinct machines, each of which has a different sequence of operations. This thesis investigates the application of Reinforcement Learning (RL) algorithms in addressing the JSSP, focusing primarily on instances from the Lawrence, Dermikol, and Taillard datasets. Particularly, the study evaluates popular RL algorithms, including Proximal Policy Optimization (PPO), Policy Gradient (PG), Advantage Actor-Critic (A2C), and Asynchronous Advantage Actor-Critic (A3C), against traditional dispatching rules and other state-of-the-art methods. The findings highlight the superior performance of the PPO approach, which consistently outperforms alternative RL algorithms and dispatching rules across various instance sizes. PPO demonstrates robustness and adaptability in navigating dynamic job-shop scheduling landscapes, positioning it as a versatile and potent solution for learning complex scheduling strategies. Insights from the training dynamics of RL agents underscore their ability to improve performance over time by learning from the environment. The increasing reward values and decreasing makespan observed across all datasets signify the adaptive nature of RL agents in optimizing scheduling policies. Remarkably, the PPO-based approach demonstrates a 6-9 times lower optimality gap compared to traditional scheduling algorithms and achieves a 2-3 times lower optimality gap than state-of-the-art approaches in all three datasets, highlighting its superiority in addressing the JSSP. This thesis contributes to the understanding of the efficacy of RL algorithms in scheduling optimization, suggesting their potential to significantly enhance operational efficiency across diverse industrial sectors.

Contents

List of Figures

List of Tables

List of Acronyms

A2C	Advantage Actor-Critic
AI	Artificial Intelligence
A3C	Asynchronous Advantage Actor-Critic
COP	Combinatorial Optimization Problem
CP	Constraint Programming
DL	Deep Learning
DRL	Deep Reinforcement Learning
EU	European Union
GP	Genetic Programming
GPHH	Genetic Programming-based Hyperheuristic
JSSP	Job Shop Scheduling Problem
LP	Linear Programming
LSTM	Long short-term memory
ML	Machine Learning
MDP	Markov Decision Process
MLP	Multilayer Perceptron
PG	Policy Gradient
PDRs	Priority Dispatch Rules
PPO	Proximal Policy Optimization
RLib	Ray library
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
SPT	Shortest Processing Time
TSP	Traveling Salesman Problem
WandB	Weights & Biases

Chapter 1

Introduction

1.1 Introduction

The Job Shop Scheduling Problem (JSSP) is a well-known optimization problem in the fields of production management and operations research, with significance in manufacturing, healthcare, supply chain management, and many other applications [1]. One of the most important tasks in the industrial sector is scheduling. Planning the length of the task or procedure and how it will handle the equipment or resources is crucial. This planning exercise determines how long tasks will take to complete and how the various work activities relate to one another. Stable planning and production, real-time production feedback, and capability control are the advantages of job shop scheduling. In particular, managers are able to optimize the way they control their production plants. They can arrange their staff, machinery, materials, and resources in accordance with this timetable.

Optimization of job shop scheduling offers a number of benefits, including faster delivery times, lower completion times and makespan, fewer resources at bottlenecks, and less idle time. When taken as a whole, these advantages improve customer happiness and operational effectiveness across a variety of businesses. The benefits of job shop scheduling are not limited to specific industries, they also affect important performance indicators in larger business disciplines. Increased customer satisfaction and operational effectiveness in a variety of business contexts can be attributed to faster delivery times, shorter completion durations, less makespan, and more efficient resource allocation during bottlenecks. Organizations can achieve a competitive advantage in dynamic market settings by optimizing resource use, minimizing idle time, and preserving flexible production processes through the implementation of effective scheduling techniques. But in order to properly take use of these advantages, it is necessary to deal with the computational difficulties and intrinsic complexity of the JSSP, opening the door for creative fixes and cutting-edge optimization strategies suited to a variety of industrial backgrounds.

1.2 Motivation

The motivation for this study stems from the ambitious objectives of the European Union's Horizon 2020 project, known as the "Rhinoceros" project, which focuses on optimizing the entire lifecycle of lithium batteries, from production to end-of-life [32]. A crucial component of the Rhinoceros project is a dedicated work package aimed at automating the sorting and dismantling processes of car batteries. This initiative seeks to significantly enhance and streamline the disassembly process, ensuring alignment with the United Nations' Sustainable Development Goals. The ultimate goal is to foster environmental sustainability and improve resource efficiency on a global scale. By advancing the automation of these processes, the Rhinoceros project aims to minimize the environmental impact of battery waste

and contribute to a more sustainable future.
To optimize the disassembly process of car batteries, which is crucial for recycling and reusing valuable components, it is essential to break down the process into several distinct tasks. Each task represents a specific operation necessary to dismantle the battery safely and efficiently. These tasks include removing the casing, disconnecting electrical components, segregating recyclable materials, etc. This disassembling problem can be in general modeled as a JSSP, which is the focus of this study.

As mentioned earlier, the JSSP, a variant of optimal job scheduling, poses one of the most formidable challenges in combinatorial optimization within computer science and operations research [37]. In this problem, a set of jobs, each comprising multiple operations, needs to be processed on a range of heterogeneous machines. Each operation is assigned to a specific machine, with the known duration required for its completion. The primary objective is to determine the optimal sequence for scheduling these operations to minimize the makespan, which is the total time for all jobs to be completed.

Despite its practical importance, the JSSP falls under the category of NP-hard problems, meaning that solving it optimally is not feasible in polynomial time, at least at this stage. Quick solutions are not possible with methods like constraint programming or integer programming because of their high processing cost. Thus, in order to effectively obtain approximations of answers, hand-engineered heuristics are frequently employed in practice. Nevertheless, creating such heuristic principles is a difficult undertaking that needs an in-depth understanding of the issue.Priority Dispatch Rules(PDRs) are a series of practical heuristics that are quick to compute and simple to apply when solving scheduling difficulties in the context of the JSP. However, it's unclear how to apply these guidelines to other situations [11].

In view of the above-mentioned drawbacks of heuristics based approaches, the focus of current research has been on applying Reinforcement Learning (RL) to automatically create domain-specific heuristics. There are various benefits when RL is applied to JSSP. Compared to conventional priority dispatching rule heuristics, whose performance might differ greatly from instance to instance, it is, first of all, more adaptable. In contrast to traditional Combinatorial Optimization Problem (COP) techniques like constraint programming (CP) and linear Programming (LP), RL environments are able to replicate stochastic decision-making situations that are encountered by actual scheduling systems. Second, by taking into account the influence of a schedule for known jobs on the new ones, RL offers the opportunity to incrementally arrange the incoming jobs as they come in the queue, in contrast to traditional scheduling approaches that concentrate exclusively on the provided set of works. The idea of lifelong learning, which allows an agent to reuse its prior learning from JSS instances in addition to optimizing a single instance, is the most promising application of RL [37].

The majority of JSSP research so far has concentrated on tackling small-scale situations, frequently finding it difficult to function well with larger ones. The purpose of this thesis is to minimize this gap by creating RL-based techniques that are specially made to manage larger JSSP instances efficiently. This work aims to push the limits of what is possible in JSSP optimization by giving priority to increased scheduling efficiency and scalability across various problem sizes. In addition, for the application area of our EU project, the job scale and the machine number are usually large. To this end, as part of this study, first, the experiments are recreated from [31] to validate the methodology and replicate the findings in a given environment. This allows to establish a foundation for this research and ensures the reliability of the results. Then, the performance of different RL algorithms is compared across various datasets and instance sizes to determine the most effective approach for solving the JSSP.

1.3 Goals and Research Questions

This thesis aims to tackle the JSSP as a single-agent RL problem by utilizing Deep Reinforcement Learning (DRL) techniques. The primary goals are to enhance scheduling efficiency in industrial settings and reduce the overall makespan by teaching a dispatcher agent to choose jobs for processing sequentially. This goal encompasses several key aspects, which are articulated as the following research questions:

- **RQ1:** How can DRL models be effectively implemented to solve the JSSP as a single-agent RL problem?
- **RQ2:** How do different RL algorithms, including Proximal Policy Optimization (PPO), Policy Gradient (PG), Advantage Actor-Critic (A2C), and Asynchronous Advantage Actor-Critic (A3C), perform in solving the JSSP across various datasets and instance sizes, particularly in terms of scheduling efficiency and reducing overall makespan?

To this end, a set of hypotheses has been formulated to address each of these questions respectively:

- **H1:** The PPO algorithm can be effectively implemented to solve the JSSP as a single-agent RL problem, achieving a lower makespan compared to other RL algorithms (PG, A2C, A3C).
- **H2:** The implementation of PPO will result in a consistent increase in rewards during the training process, indicating effective learning and adaptation to the scheduling environment.
- **H3:** The PPO algorithm will achieve lower makespan values compared to traditional dispatching rules (FIFO, MWKR, SPT) and state-of-the-art DRL approaches in the literature.
- **H4:** The PPO algorithm will demonstrate a lower optimality gap compared to traditional dispatching rules and state-of-the-art approaches, indicating that the solutions are closer to the optimal or best-known solutions.

1.4 Contributions

The following is a summary of this thesis's main contributions:

1. Implement a DRL Model: This thesis particularizes a deep learning model, in order to solve the JSSP as a single-agent RL problem. Specifically, the actor-critic Proximal Policy Optimization algorithm is used to learn a policy that maps a state to an action probability distribution.
2. Comparative analysis of different RL algorithms: This study, in addition to the implementation of a particular RL strategy, also evaluates the applicability of several RL algorithms by conducting a comparative study of them in order to solve JSSP.
3. Numerous experiments on hyperparameter sensitivity are carried out to optimize the solution.

1.5 Thesis structure

The thesis begins with an introduction delineating the motivation, goals, research questions, contributions, and overall structure. Following this, Chapter 2 delves into background theory, encompassing related work and theoretical foundations of RL algorithms. Chapter 3 outlines a RL approach to address the JSSP, covering environment setup, action selection, reward function design, state representation, dataset overview, and implementation details using the Ray library (RLib). Chapter 4 delves into the empirical analysis of RL algorithms applied to the JSSP, showcasing detailed comparisons of performance metrics such as makespan and optimality gap across diverse datasets and instance sizes. The chapter also provides insights into the training dynamics of these algorithms, illustrating their learning capabilities and improvements over iterations, thereby offering valuable guidance for algorithm selection and future research directions in scheduling optimization. Chapter 5 discusses the effectiveness of RL algorithms, particularly PPO, in addressing the JSSP, highlighting PPO's superiority in minimizing makespan and learning efficiency, while acknowledging its limitations and the importance of algorithm selection. Finally, Chapter 6 provides a comprehensive conclusion to the thesis, summarizing key findings, discussing their implications, and proposing future research directions to enhance scheduling performance and address scalability issues

Chapter 2

Background Theory

This Chapter explores background theory, including relevant literature and the conceptual foundations of RL algorithms.

2.1 Related Work

Many scheduling approaches have been developed in the literature over the last few decades with the goal of solving the Job Shop Scheduling Problem. These methodologies cover a broad spectrum of approaches, including heuristic, reinforcement learning techniques, as well as mathematical programming. DRL has gained popularity recently as a comprehensive method for solving COPs. The number of DRL applications is rapidly rising, ranging from the Traveling Salesman Problem (TSP) via Graph Optimization to the Satisfiability problem. Nevertheless, DRL's use in scheduling issues is more recent and restricted.

2.1.1 Heuristic Approaches

In part due to the curse of dimensionality and the inability to be modified in real-time, mathematical programming optimization techniques like mixed integer programming [24] and integer linear programming [19] are not practical for large-scale or dynamic scheduling problems. However, they can find the best solutions for small-scale problems. Approximate solution optimization techniques, which seek to identify nearly optimal solutions, have been used to solve the dynamic job-shop scheduling problem since exact optimization methods are unable to solve it well. DJSP is frequently solved using heuristic dispatching rules, which can produce workable but maybe suboptimal solutions in a brief amount of time [35]. Effective dispatching rules must be manually designed through a process of trial and error that takes a lot of time, code, and domain expertise [5]. The Shortest Processing Time (SPT) is a simple PDR that is simple to implement and low in time complexity. However, instance-related factors like shop configuration, operating conditions, and objective functions have a significant impact on the performance of PDR scheduling methods.

2.1.2 Metaheuristic Techniques

Random search factors are introduced to improve the exploration ability and higher quality solutions can be provided in a reasonable amount of computational time using tabu search [8], simulated annealing [36], and genetic algorithms [21]. Because the techniques mentioned above are meant to identify all scheduling assignments for a given initial condition, and because fine-tuning the parameters of these methods is particularly challenging, they are not easily applied to dynamic scheduling issues, where the conditions of the problem change continuously. According to the paper [2], these approaches should therefore be used again anytime the scheduling problem's conditions change. However, because they demand a significant amount of calculation time, they are not practicable in real-world settings.

2.1.3 Genetic Programming-based Hyperheuristic

One of the most efficient ways to solve DJSP these days is to use hyper-heuristic approaches, including Genetic Programming-based Hyperheuristic (GPHH) [25]. GPHH seeks to automatically build dispatching rules in the heuristic space rather than the solution space [5]. Nonetheless, the rules produced by GPHH are frequently extremely big, complex, and challenging to understand. In GPHH, numerous feature selection techniques are used to shorten the length of generated dispatching rules [37]. Unfortunately, the Genetic Programming (GP) approach is too time-consuming and impractical because the existing feature selection methods typically modify offline selection processes to acquire a promising subset of terminals, which necessitates extra simulation runs.

2.1.4 Reinforcement Learning for Combinatorial Problems

The reinforcement learning approach, one of the most exciting areas of machine learning, is frequently used for operations and production management issues, particularly those involving decisions in dynamic contexts. The application of deep reinforcement learning to find end-to-end solutions for combinatorial problems was originally shown in the paper [3]. The Traveling Salesman Problem was addressed by the authors using the Pointer Network inside an actor-critic architecture. With RL and a customized critic for effective learning, the work [7] expands the neural combinatorial optimization framework for the TSP without the need for Long short-term memor (LSTM). In contrast to pointer networks, this study [12] suggests an attention-based model trained with RL for combinatorial optimization heuristic learning. All of these research, nevertheless, rely on sequence-to-sequence models, which makes extensive sampling and search methods necessary for inference in order to enhance results.

Considering every machine as an agent is a traditional method of solving model scheduling issues. This work [33] demonstrates the successful application of RL with a Deep Q-Nework (DQN) agent for production scheduling. After only two training days, expert-level answers were achieved, matching established heuristics without the need for human intervention. With actor and critic networks, the paper [18] leverages deep reinforcement learning to effectively manage sequential decision-making without the need for hand-crafted features in the dynamic realm of job shop scheduling. The two previously mentioned publications highlight flexibility and suggest scenarios in which machines break down randomly.

Furthermore, an extensive number of scheduling examples can be found in various application domains including distributed computing [20] and manufacturing [17]. Many of these approaches, despite their best efforts, do not match traditional heuristics, and their scalability across a wide range of problem sizes is hampered in some studies by state representations that are limited by variables such as job size or a number of processes. Using DRL with graph neural networks, this research [37] suggests an automatic technique to train PDRs for the Job-shop scheduling problem. Even on larger instances that have not yet been seen, the method achieves great performance against PDRs and can be generalized to large-scale instances. Similarly, this research [9] proposes an adaptive scheduling framework for job shop scheduling utilizing DRL that makes use of disjunctive graphs in conjunction with dueling networks, double DQN, and prioritized experience replay. The scheduling states are represented as multi-channel images, which are then turned into a sequential decision problem

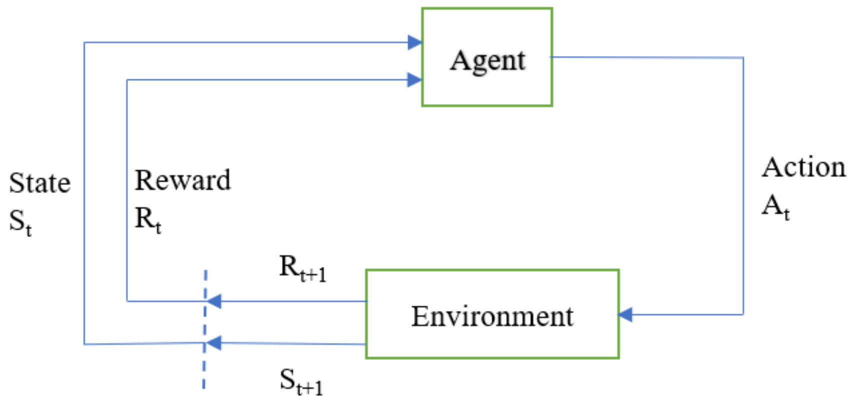

Figure 2.1: Typical Reinforcement Learning scenario framing.

using topological sorting. Nevertheless, it can be challenging to solve big examples with thousands of operations since the induced graph representations have a lot of nodes and edges that indicate all conceivable ordering of the operations in an instance.

2.2 Related Theory

This section delves into the foundational concepts that underpin RL, a paradigm facilitating autonomous agents' interaction with their environment to achieve optimal decision-making. Exploring these fundamental principles paves the way for a deeper understanding of RL methodologies and their applications in various domains.

2.2.1 Reinforcement learning

An agent interacts with an environment in RL, where the agent's actions modify the state of the environment. The goal of RL is to create a framework in which an agent, lacking any prior information, learns optimal behavior in the environment via trial and error. A reward function measures how well the agent's actions correspond with the intended result, capturing this optimal behavior. The agent's goal is to discover a strategy that maximizes rewards for particular states by investigating the environment and learning how to maximize this reward function. The typical framing of the RL scenario is shown in Figure 2.1.

2.2.2 Element of Reinforcement Learning

The goal of reinforcement learning is to determine how to make choices that maximize reward. Feedback about the agent's performance at each time step is the reward (R). At every stage, the reward and the environment-descriptive state (S) determine the agent's behavior (A). The policy (P) directs the agent's behavior by mapping states to actions. Furthermore, the value function (V) assesses each state's quality; in contrast to instant rewards, it indicates a state's long-term desirability. The key elements of reinforcement learning as listed below:

Agent

An intelligent system that learns and makes decisions within an environment, aiming to maximize its cumulative rewards over time. The agent's primary goal is to learn the optimal policy that maps states to actions to maximize rewards. Also, the agent can be a human, robot, or software program and is responsible for making decisions in the environment.

Environment

The external system that the agent is operating in, gives it feedback and affects its decisions with its dynamic. The environment can be a physical or virtual system, such as a game, a robot, or a simulation. The environment's state is the agent's input, and the agent's actions influence the environment's state. Since the agent's actions are based on the environment's state, the agent's decisions are influenced by the environment.

Action

The decisions taken by the agent impact the ensuing state transitions and are based on the observable states of the environment. The agent's actions are determined by the policy, which is a mapping from states to actions. The agent's goal is to learn the optimal policy that maximizes rewards by taking actions in the environment.

State

A representation of the setup of the current environment that includes crucial data for the agent to make decisions. The agent's actions are based on the environment's state, which is the input to the agent's decision-making process. The agent's goal is to learn the optimal policy that maximizes rewards by taking actions in the environment.

Reward

After each action, the environment provides feedback to the agent about the quality of the agent's behavior in a certain state. The agent's goal is to maximize the cumulative reward over time by learning the optimal policy. The reward is a scalar value that indicates how well the agent is performing in the environment.

Policy

Policy is Rule or tactic that specify how the agent maps states to actions in order to accomplish its goals and control its behavior. The agent's behavior function, or policy, instructs the agent on what to do in various states. It is a mapping, which might be stochastic or deterministic, from state s to action a. The policy can be deterministic or stochastic, and it is the agent's strategy for selecting actions in the environment. The policy is a mapping from states to actions that guide the agent's behavior.

- Deterministic:$\pi(s) = a$,
- Stochastic: $\pi(a|s) = \mathbb{P}_\pi[A = a|S = s]$.

Episode

A complete series of interactions that go from a starting condition to a terminal one and offer distinct learning chances in episodic real-life environments. An episode is a sequence of interactions between the agent and the environment that begins with the agent's initial state and ends with a terminal state. The agent's goal is to maximize the cumulative reward over time by learning the optimal policy.

Discount factor γ

It determines the extent to which future rewards are taken into return. γ has a value between 0 and 1. In the extreme, an agent that values $\gamma = 0$ is only interested in rewards that come in the near future, whereas an agent that values $\gamma = 1$ looks at all rewards that come in the future.

Q-value

A Q-learning and related algorithms estimate of the predicted cumulative future reward that an agent can choose to get by selecting a particular action in a particular state. The Q-value is a measure of the quality of an action in a given state, and it is used to determine the best action to take in a given state. The Q-value is the expected cumulative reward that an agent can receive by taking a particular action in a particular state. The Q-value is used to determine the best action to take in a given state.

Exploration and Exploitation

The process of trying out new behaviors to find out more about the environment and perhaps better behaviors is called "exploration". On the other hand, exploitation comprises choosing actions that are known to yield large rewards based on the agent's existing understanding. To learn effectively, one must find the perfect balance between exploitation and exploration. While excessive research can lead to inefficiency, excessive exploitation could hinder the agent from discovering more productive paths of action. Finding the best trade-off in reinforcement learning to maximize cumulative rewards and guarantee effective learning is a major challenge.

2.2.3 Value Function

Value functions use future reward predictions to evaluate how excellent a situation or action is, or how rewarding it is. The total of all future reduced rewards is the future reward, sometimes referred to as the return. The return G_t is calculated as below Eq. (2.1) and future rewards are penalized by the discounting factor $\gamma \in [0, 1]$.

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{2.1}$$

The state-value function is the expected return starting from s following policy π and is represented as:

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]. \tag{2.2}$$

The action-value function indicates how helpful it is to perform a specific action in a specific state and is represented as:

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]. \tag{2.3}$$

The difference between action-value and state-value is the action advantage function and represented as:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s). \tag{2.4}$$

The optimal value function produces the maximum return:

$$V_*(s) = \max_\pi V_\pi(s) , Q_*(s, a) = \max_\pi Q_\pi(s, a). \tag{2.5}$$

The optimal policy achieves optimal value functions:

$$\pi_* = \arg\max_\pi V_\pi(s) , \pi_* = \arg\max_\pi Q_\pi(s, a). \tag{2.6}$$

2.2.4 Markov Decision Processes

In the area of RL, a Markov Decision Process (MDP) is a mathematical framework used to model decision-making issues. An agent engages with the environment in an MDP over a number of distinct time steps. At each time step, the agent evaluates the state of the environment before selecting a course of action. The environment responds by altering into a new condition and rewarding the agent. Both the transition to the new state and the reward are influenced by the agent's actions and the present state.

The agent and environment interact at each of a number of discrete time steps, $t = 0, 1, 2, 3, ...$ At each time step t, the agent receives a representation of the environment's state, $S_t \in \mathcal{S}$, and chooses an action, $A_t \in \mathcal{A}(s)$. The agent chooses a new state, S_{t+1}, and is rewarded, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, after one time step. Consequently, the combination of the MDP [29] and agent produces a trajectory or sequence that starts as:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ... \tag{2.7}$$

Within a finite MDP, each of the sets of states, actions, and rewards ($\mathcal{S}, \mathcal{A},$ and $\mathcal{R}$) has a finite number of components. In this case, the discrete probability distributions for the random variables R_t and S_t are well-defined and exclusively reliant on the state and activity of the past. That is, for certain values of these random variables, $s^{'} \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability that these values will occur at time t given specific values of the previous state and action: (2.8)

$$p(s', r|s, a) = Pr\{S_t = s^{'}, R_t = r|S_{t-1} = s, A_{t-1} = a\}, \tag{2.8}$$

for all $s', s \in \mathcal{S}, r \in \mathcal{R}$ and $a \in \mathcal{A}(s)$. Here, the function p defines the dynamics of the MDP.

2.2.5 Features of Reinforcement Learning

In general, there are two types of RL problems: model-based and model-free. Model-based RL allows the agent to plan and simulate future actions by generating a comprehensive representation of the environment. This technique has the benefit of allowing for forward planning and logical decision-making, but it requires proper environmental modeling. Conversely, model-free reinforcement learning gains knowledge through interactions with the environment rather than relying on a predetermined framework. Because it concentrates on predicting the value or policy directly from observed experiences, it is more adaptable; yet, learning necessitates more interactions. The complexity of the environment and the trade-off between sampling efficiency and planning skills have an impact on the choice between model-based and model-free reinforcement learning.

RL environments fall into one of two categories: deterministic or stochastic. In a deterministic setting, results are repeatable and predictable since the subsequent state and reward simply depend on the current state and action. In stochastic situations, where randomness and uncertainty are introduced, the same action taken in the same state may yield various consequences. Both the reward obtained and the change to the next state are determined by a probability distribution. By making the environment as deterministic as is practical, convergence will be enhanced. Providing the agent with additional information will help achieve this.

Environments in RL can be classified as discrete or continuous depending on the properties of their state and action spaces. Discrete environments have a finite or countable number of different states and actions, whereas continuous environments have real-valued variables that represent continuous and infinite state and action spaces. Understanding the differences between discrete and continuous environments is essential because it influences the algorithms and techniques that real-time agents select to use. Discrete environments can be easily handled with tabular approaches, but continuous settings sometimes require the employment of function approximation techniques in order to manage the endless possibilities.

Mathematically defining an environment for RL requires careful attention to detail. Agents can exploit any gaps in the environment's description, especially in open-world or simulation-based environments. Rewards and penalties should only be earned by the desired actions. For example, in a video game, if there is a reward for reaching a checkpoint, it should be deleted after an agent passes through it once to prevent exploitation. By doing this, it is ensured that the agent is focused on accomplishing the intended goals rather than attempting to maximize rewards by repeatedly crossing the checkpoint.

Both on-policy and off-policy algorithms are used in RL. On-policy algorithms learn from and improve the present policy by applying the knowledge they have acquired from its implementation. The agent uses the same policy to explore and interact with its environment while simultaneously updating it. Off-policy algorithms, on the other hand, use information from a separate policy or behavior to improve and modify the policy. Both exploratory data and data collected by following an alternate policy can be used by the agent to learn new things.

2.2.6 Reinforcement Learning algorithms

An RL training algorithm's objective is to maximize the predicted cumulative reward over time by learning a policy, or a mapping from states to actions. This policy directs the agent's choices, enabling it to make decisions that, in the current environment, result in greater rewards. The algorithm modifies the policy iteratively to improve the behavior of the agent and reach optimal performance through learning from encounters. For a state s_0, we can calculate the value V_π of a policy π by using the following equation:

$$V^\pi(s_0) = \mathbb{E}^\pi\Big[\sum_{t=0}^{+\infty} \gamma^t \ R(s_t, a_t\Big], \tag{2.9}$$

where γ, discount rate, is a hyper-parameter that regulates how far the agent looks into the future and $\mathbb{E}^\pi$ is the expectation over the distribution of the acceptable trajectories $s_0, a_0, r_0, s_1, r_1, a_1$ derived by sampling the actions from the policy.

Estimating the state-value function, which estimates the total expected reward from a specific condition, is the primary objective of value-based algorithms. After obtaining this function, the action that leads to the state with the highest expected value is selected as the policy. underlined this approach, which focuses on appreciating states to guide decision-making [23]. On the other hand, policy-based algorithms attempt to discover the policy that maximizes cumulative reward through direct learning. These strategies function by raising the likelihood of choosing courses of action that have historically produced large rewards. This method places greater emphasis on the behavior of the agent and the direct optimization of incentives [29]. In modern reinforcement learning, a combination of these two approaches has emerged in the form of actor-critic methods, as discussed in paper [22]. Actor-critic approaches make use of both a policy network, called the "actor", and a value function network called the "critic". Based on the existing policy, the actor recommends courses of action, and the critic assesses them by calculating their predicted values. The combination of policy-based decision-making and value-based estimates enables effective learning and enhanced performance in complicated situations.

Policy Gradients Methods

Reinforcement learning algorithms that learn a policy directly are known as policy gradient methods. Typically, a neural network is used to represent the policy parameters. The expected cumulative payoff is to be maximized. To accomplish this, gradients of expected rewards with respect to the policy parameters are computed and updated, usually via gradient ascent.

Policy gradient [15] methods use entropy regularization to prevent premature convergence to suboptimal policies. This regularization term promotes the exploration of a varied action space and is proportional to the negative entropy of the policy distribution. By encouraging a balance between exploitation and exploration, it aids in preventing the policy from being mired in local optima.

The goal of the policy gradient approach is to directly model and optimize the policy. Typically, a parameterized function with regard to θ, $\pi_\theta(a_t|s_t)$ is used to model the policy. This policy determines the value of the reward (objective) function, and various algorithms can then be used to optimize for the optimal reward.

In discrete space, the objective function is as below:

$$J(\theta) = \ V_{\pi\theta}(S_1) = \mathbb{E}_\pi[V_1],$$

where S_1 is the initial starting state.

In continuous space, the objective function is as below:

$$J(\theta) = \sum_{s\in S} d^\pi(s)\ V^\pi(s) = \sum_{s\in S} d^\pi(s) \sum_{a\in A} \pi_\theta(\ a|s) Q^\pi(s,a),$$

where $d^\pi(s)$ is the stationary distribution of of Markov chain for π_θ.

The gradient of the reward function is calculated with respect to the policy parameters and used this information to identify the necessary changes to the policy to raise the predicted reward. The policy updating process is guided toward activities that are more likely to result in higher rewards by this gradient. The gradient of $J(\pi_\theta)$ is:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t,a_t)\right],$$

where τ is a trajectory and $A^{\pi\theta}$ is the advantage function for the current policy.
The policy gradient algorithm works by updating policy parameters via stochastic gradient ascent on policy performance:

$$\theta_{k+1} = \theta_k + \alpha\nabla_\theta J(\pi_{\theta_k}).$$

Algorithms that optimize the policy in such a way are known as policy gradient algorithms. The gradient of policy performance is denoted by $\nabla_\theta J(\pi_\theta)$.

Proximal Policy Optimization

Proximal Policy Optimization [15] is an actor-critic model that combines elements of both value-based and policy-based reinforcement learning approaches. Actor-critic models consist of two primary parts:

- Actor: The actor's role is to learn a policy that associates states with actions. Its objective is to directly modify the policy settings to maximize the projected cumulative benefit.

- Critic: The critic evaluates the morality of the actions chosen by the actor. It estimates the value function, which represents the expected cumulative reward if the policy is followed starting from a certain condition.

In PPO, the actor represents the "policy" component, which learns to choose actions in states based on the parameterized policy given by θ. The advantage function, used to calculate the worth of actions and guide policy adjustments, is analogous to the "value function" part. Similarly to how actor-critic approaches update policy parameters based on advantages determined by a critic, PPO uses the advantage function to compute policy gradients and update the policy parameters..

PPO is a policy gradient method for reinforcement learning. By limiting the size of policy updates, the PPO reinforcement learning method improves training stability. This is achieved by introducing a clipped objective function, which imposes a constraint on how far the new policy can diverge from the old one. PPO aims to maintain consistency in learning progress by limiting the magnitude of policy changes, thus promoting more dependable convergence to an optimal policy.

PPO-clip updated policies via:

$$\theta_{k+1} = \arg\max_{\theta} \underbrace{\mathbb{E}}_{s,a\sim\pi_{\theta_k}} [L(s, a, \theta_k, \theta],$$

typically taking multiple steps of (usually minibatch) SGD to maximize the objective and the loss (surrogate objective) function, L, is written as:

$$L(s, a, \theta_k, \theta) = min\big(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), g(\epsilon, A^{\pi_{\theta_k}}(s, a))\big),$$

where

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A, & A \geq 0, \\ (1-\epsilon)A, & A < 0. \end{cases}$$

Here $\theta$and θ_k are the parameters of the new and the old policy respectively.

Asynchronous Advantage Actor-Critic

Asynchronous Advantage Actor Critic is a RL policy gradient algorithm that upholds a policy $\pi(a_t|s_t;\theta)$ and a value function estimate $V(s_t, \theta_v)$. It updates the policy and the value function simultaneously while operating in the forward view and employing a combination of n-step returns [22]. The policy and the value function are updated after every t_{max} action or when a terminal state is reached. In A3C, critics learn the value function while multiple actors receive parallel instruction while they synchronize with global parameters. Similar to parallelized stochastic gradient descent, the gradients are gathered as part of the stability training process.

The title of the algorithm comes from two key concepts that A3C expands upon. Initially, by acquiring an estimate $\hat{V}_\phi$ of the value function, A3C indirectly learns an estimate $\hat{A}_\phi$ of the advantage function. Secondly, A3C presents the idea of employing several concurrent actors to engage with the surroundings in order to maintain training stability. Policy gradients are sampled by the algorithm [15]

$$\hat{\nabla}_\theta = \frac{1}{|\mathcal{D}|} \sum_{s,a \in \mathcal{D}} \hat{A}_\phi(s,a) \nabla_\theta ln\ \pi_\theta(a|s),$$

where $\mathcal{D}$ is a batch transition collected by the actors.

Advantage Actor-Critic

The A2C algorithm [15], is a synchronous version of the A3C method, in which each actor interacts synchronously with their own copy of the environment. By interacting with the environment, each actor gains experience, which it subsequently uses to adjust the global neural network's settings. Like A3C, A2C makes use of the advantage function to evaluate the actor's action quality and directs policy changes appropriately. Policy gradients are calculated using batches of transitions—state-action-reward-next state tuples—that are gathered by the actors. A2C provides stability and predictability in training by ensuring that changes are synchronized and averaged over all actors by delaying global parameter modifications until each actor has finished their experience segment. About effective reinforcement learning, A2C offers a deterministic and synchronous substitute for A3C while preserving the benefits of the actor-critic architecture.

2.2.7 Deep Reinforcement Learning

Deep reinforcement learning [28] is a subfield of machine learning that combines deep learning with reinforcement learning. DRL algorithms use deep neural networks to approximate the value function or policy function in reinforcement learning problems. By leveraging the power of deep learning, DRL algorithms can handle complex, high-dimensional input data, such as images or text, and learn more sophisticated representations of the environment. DRL has been successfully applied to a wide range of tasks, including playing video games, controlling robots, and optimizing resource allocation. DRL algorithms have achieved impressive results in many domains, outperforming traditional reinforcement learning methods and human experts in some cases.

2.2.8 Multilayer Perceptron (MLP)

A Multilayer Perceptron (MLP) [28] is a type of artificial neural network that consists of multiple layers of nodes, each of which is connected to the next layer. The nodes in the input layer receive input data, which is then passed through the network to the output layer. Each node in the hidden layers performs a weighted sum of the inputs and applies an activation function to produce an output. The weights and biases of the network are adjusted during training to minimize the error between the predicted output and the actual output. MLPs are commonly used for supervised learning tasks, such as classification and regression, and can be trained using backpropagation. MLPs are versatile and can be used for a wide range of tasks, including image recognition, natural language processing, and time series forecasting.

An advanced kind of artificial neural network made up of several layers of nodes—also known as neurons—is called a Multilayer Perceptron (MLP) [28]. The input layer, hidden layers, and output layer are the three primary categories for these layers. The initial data must be received by the input layer, where each node represents a distinct feature of the input data. Each node in an image recognition task, for instance, might be associated with a particular picture pixel value. One or more hidden layers follow the input layer; they are called "hidden" because the input or output does not explicitly display their values. The nodes in these hidden layers apply an activation function after performing a weighted sum of the inputs from the preceding layer. The model gains non-linearity from this activation function, which helps it recognize intricate patterns in the data. Rectified Linear Unit (ReLU), hyperbolic tangent (tanh), and sigmoid are examples of common activation functions. The output layer, which is the last layer, is responsible for producing the network's predictions. Whereas in regression tasks, every node in the output layer may represent a continuous value, in classification tasks, every node might represent a class label.

Learnable parameters called weights define the connections between the nodes in these levels. Furthermore, biases are applied to every node prior to the activation function being applied. During the training phase, these weights and biases are changed to reduce the error between the target values and the anticipated outputs of the network. Backpropagation, an optimization technique that uses the chain rule to determine the gradient of the loss function with respect to each weight, is commonly used in the training of multiple linear polynomials. As a result, the weights can be updated by the model to lower the loss. To improve the effectiveness of this training procedure, gradient descent optimization techniques like Adam and stochastic gradient descent are frequently employed. For supervised learning tasks like regression and classification, MLP are frequently utilized. They are flexible tools in machine learning because they can process and learn from a variety of data kinds. For example, machine learning algorithms are used in image recognition to detect and categorize objects in photographs, and in natural language processing to evaluate and interpret textual input for tasks like sentiment analysis and language translation. Time series forecasting, which predicts future values based on historical data, is another valuable use for MLP. Exameples of this type of forecasting include weather forecasts and financial market analysis.

Chapter 3

Methodology

This chapter describes how to solve the JSSP using RL. It covers how to set up the environment, choose an action, design the reward function, represent the state, provide an overview of the dataset, and implement the solution using RLlib.

In JSSP, a fnite set of n machines $\{M_k\}_{k=1}^{n}$ are used to process a finite set of m jobs $\{J_i\}_{i=1}^{m}$. Every task J_i is composed of n operations $(O_{i,1} \to O_{i,2} \to \ldots \to O_{i,n})$ that need to be completed in a specific order. The machine M_k is assigned for a specific processing time $D_{i,j}$ for each operation, $O_{i,j}$. The objective is to ascertain which processes should be scheduled in what order to minimize the makespan or overall execution time. There are several constraints for the job shop problem:

- A machine can only work on one task at a time: This implies that a single operation can be handled by each machine at a time. A machine cannot go on to another task until it has finished the one it is now processing. This limitation makes sure that machines don't take on too many jobs at once.

- No task for a job can be started until the previous task for that job is completed: A job's sequence of operations dictates that one activity cannot start until the other is completed. This constraint guarantees the proper sequence of operations within each task, for instance, if operation $O_{i,1}$ must be finished before operation $O_{i,2}$ may begin.

- A task, once started, must run to completion: This limitation guards against errors or inefficiencies in the production process by guaranteeing that once operations commence on a machine, they continue uninterrupted.

Figure 3.1 is a sample solution for a small instance with three machines and three jobs. Every job is tracked by a single machine, and each operation takes a different amount of time [31].

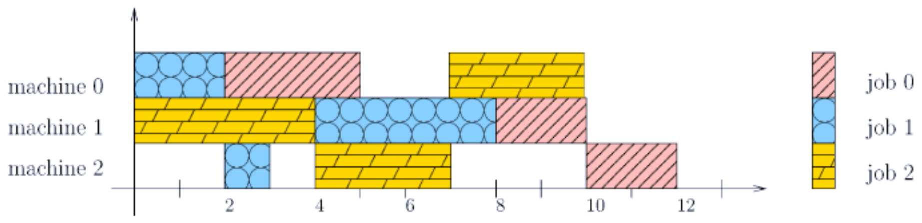

Figure 3.1: A solution for a small instance composed of three jobs and three machines [31].

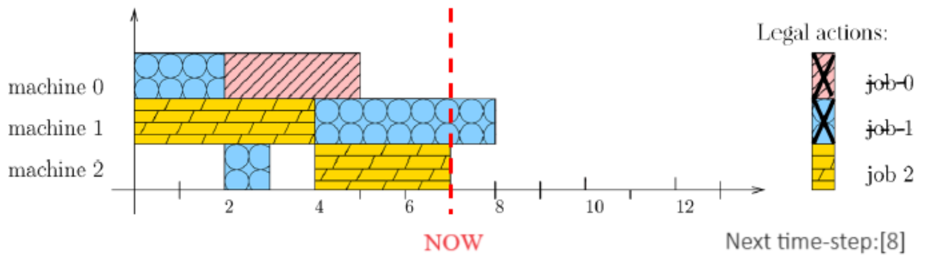

Figure 3.2: Illustration of the environment's status, indicating the current time-step, job allocations, available options, and future time-steps [31].

3.1 Job-Shop Scheduling Environment

3.1.1 Environment

The environment is intended for RL applications, where a dispatcher, or agent, makes judgments about distributing work to machines over time in order to improve scheduling. The JSSP is modeled as a single-agent problem in this environment. The list of available actions includes an additional action called No Operation, which is signified "by No-op", to move on to the next time-step without scheduling any specific operation. As a result, the discrete jobs represented by the formula $A = \{J_0, J_1, ..., J_{|J|-1},\ No - Op\}$ comprise the action space. "No-Op" basically denotes the choice to remain inactive and not carry out any particular duty, enabling the agent to advance to the subsequent time-step without carrying out any specific action. It is not always possible to assign a task at every stage of the process due to a number of constraints. The machine being used, the machine that already has work assigned to it, or the accomplishment of all job procedures could be examples of these constraints. To handle these limitations, the environment indicates which actions are allowed using a Boolean vector. The agent places a task in the order of allocation onto a stack for the next time-step it will visit after it is allocated. Afterward, if it reaches a time-step where no more jobs can be allotted or if we decide to carry out a "No-Op" action, iteratively moves to the next time-step until a machine becomes available for job allocation once again [31].

Figure 3.2 provides an overview of the current state of the environment, specifically focusing on scheduling tasks on machines. The vertical red line serves as a marker indicating the current time-step within the scheduling process. This temporal reference point is crucial for decision-making. There are two potential actions based on the current environment status. Firstly, job J_2, highlighted in yellow, can be allocated to machine M_0; however, the specific machine is not explicitly provided. Alternatively, as there's only one legal action available, advancing to the next time-step, T_8, is an option. This progression allows machine M_1 to become available, facilitating the allocation of job J_0 to it.

3.1.2 Reduction Search-Space

It is vital to guide our agents away from investigating less-than-ideal solutions in order to achieve high performance effectively. This is particularly crucial in situations where time is of the essence. Solutions that are not ideal can waste resources, cause inefficiencies, and impair overall performance. To increase efficiency and provide better results faster, the agent is directed to prioritize actions and make well-informed decisions that result in the best possible outcomes.

In order to ensure the effective use of machine resources, non-final prioritization is a technique in job scheduling that gives preference to jobs that are not at their last execution. Jobs that are not at their end operation are given priority using this method. Stated differently, when two jobs are being considered, J_2 is given priority for machine allocation if J_1 is at its end operation and J_2 is not. By doing this, the machine stays active and productive after finishing final processes and avoids idle time. Instead of dedicating time to work that are almost finished, this tactic seeks to maximize resource utilization by keeping the machine busy with jobs that require further operations.

The "No-Op" action in job scheduling requires careful consideration due to its complexity and potential for worse results compared to greedy policies [31]. If the agent is not properly organized, it might completely quit using No-Op and develop excessive greed. To address this, the first rule designates jobs as illegal for No-Op if they can be assigned in a way that ensures effective time management. Each machine calculates the minimal duration (D) of legitimate jobs it can schedule in order to further restrict No-Op. To avoid unnecessary waiting, No-Op is forbidden if a new job will arrive in fewer than D time-units. Additionally, it would be pointless to wait for a job that is in its final operation when the next job is ready for allocation. These guidelines help the agent use No-Op wisely, resulting in effective work scheduling and resource optimization. The action is prohibited if there are four or more machines with an open job or five or more jobs that can be allocated, in order to prevent improper usage of "No-Op" and save time. If the agent continues to select No-Op, it looks for new employment in succeeding time-steps until one becomes available.

3.1.3 Reward

One common statistic in the JSSP is the minimum makespan, which represents the overall time required to complete the task. However using this metric alone as the RL reward function can result in sparse feedback, where the agent only gets input at the conclusion of each episode. The agent's capacity to understand the immediate effects of its actions is hampered by this limited feedback. For example, the consequences of the agent idling a machine early on when it could be working might not become apparent until the end of the episode. The agent's ability to learn and improve its scheduling technique is hampered by this feedback latency.

An advanced reward function centered on the scheduled area is designed to overcome the above problem. After every action, the discrepancy between the duration of assigned operations and any "holes" (idle time) created on a machine is measured. The scheduled area in the context of the JSSP is the interval of time that machines are actively processing jobs. Any idle time that machines spend not in use is not included in this category. For instance, the scheduled area in a JSSP scenario with numerous jobs and machines would be made up of the time slots where the jobs' operations are delegated to the machines. A machine is not in the scheduled area if it is not in use or not operating at all during a given period of time. The reward is represent as:

$$R(s,a) = p_{aj} - \sum_{m \in M} \text{empty}_m \left(s, s^{'}\right). \tag{3.1}$$

In Eq. (3.1), s represents the current state and s' the next state resulting from taking action a. The action a denotes the jth operation $p_{a,j}$ of job J_a with a processing time $p_{a,j}$ scheduled. The function $\text{empty}_m(s, s')$ returns the duration of idle time for machine m while transitioning from state s to s'. The reward $R(s,a)$ is computed by subtracting the sum of idle times on all the machines from the processing time of the operation $p_{a,j}$.

3.1.4 State Space

The state representation in RL is the way the environment is currently represented to the RL agent. It is an important idea since the RL agent makes decisions about what to do based on this representation. All of the necessary information that the agent requires to choose the right course of action is represented by the state. In essence, it's a moment-in-time snapshot of the surroundings that gives the agent the information they need to make judgments. In the JSSP, the state representation is organized as a $(JX7)$ matrix, where J is the number of jobs. Seven properties are present in each row of the matrix, which represents a job. The state representation attributes are as follows [31]:

Boolean (a1): Job Allocation

- It indicates if the work is assignable or not.
- The RL agent needs this feature in order to determine whether to assign a task or to wait for the next time step. The job can be assigned if "True" (1) is returned; if "False" (0) is returned, the job cannot be allocated at this time step.

Remaining Time for the Present Process (a2)

- It indicates the amount of time left to complete the task at hand.
- This property shows the remaining time for the current operation of the job to finish. The representation is standardized by scaling this number by the longest operation time in the schedule, which enables the RL agent to compare progress across various workloads and schedules.

The percentage of completed operations (a3)

- It shows how the work's operations are going.
- The RL agent can learn how much of the job's operations have been finished according to this attribute. It facilitates knowledge of the work's advancement toward completion.

Remaining Time for Complete Finishing (a4)

- It represents the amount of time left before the task is finished in its entirety.
- This property shows how much work remains until the full job is finished, much like (a2). The representation is made uniform for comparison across various work durations by scaling according to the longest overall completion time.

Time Until the Next Machine Is Ready (a5):

- It indicates how long it will take for the machine required for the performance of the following project to be available.
- This property notifies the RL agent how long it will be until the machine needed for the subsequent operation is available. It aids in determining when the work can go on to the following procedure.

IDLE Duration Since Last Operation (a6)

- shows the IDLE time elapsed since the last operation.
- IDLE time shows how long a job has been waiting without any work being done on it. The RL agent can make scheduling decisions based on this characteristic, which provides information about how long the job has been waiting since its last execution.

Cumulative IDLE Time (a7)

- indicates the total IDLE time allocated to the task in the schedule.
- The amount of time the work has spent IDLE overall during the schedule can be seen by looking at the cumulative IDLE time. It is the total of all IDLE periods since the task's start. By scaling this value, the RL agent may determine how much IDLE time there is in relation to the total amount of time spent on every operation.

The characteristics described in the state representation offer a thorough understanding of the environment surrounding the JSSP. These characteristics have the potential to reconstruct the schedule's present state and provide information on the status, progress, and machine availability of each work. The accuracy with which the agent understands the dynamics of the system at any given time is guaranteed by this representation. Furthermore, it is consistent with RL's Markov requirement, which holds that all relevant data from previous states is contained in the current state. Because of its adherence to the Markov property, the RL agent may make efficient scheduling decisions by projecting future states exclusively from the present state.

3.1.5 Action Selection

In RL, action selection describes the process through which an agent decides what to do in a particular situation. It includes choosing the optimal path of action to maximize the agent's long-term reward and is a basic component of RL algorithms. A tabular matrix is used in the environment to represent the status. In order to simplify the process of calculating action distributions and state-value estimations, this matrix is converted into a vector and supplied to the agent's MLPs. In addition to the traditional MDP model, the environment includes a mask that shows the possible legal actions the agent can take in each state. This implies that the agent sets restrictions on the options accessible by knowing which activities are permitted or allowed in the current state. With the aid of this additional information, the agent's decision-making process proceeds toward legal and feasible alternatives given the circumstances of the issue.

When an agent engages in illegal activity, one popular strategy is to penalize them, with the hope that they would learn to identify and stay away from such behavior. Nevertheless, this approach frequently results in poor results. The agent must concurrently distinguish between good and harmful acts, which complicates the learning process. Alternatively, the method entails masking the neural network's output. The values associated with unlawful activity are changed by this mask into a tiny negative number, such as the smallest representable number. This method significantly reduces the chance of illegal acts to almost zero when the softmax function is then applied to compute action probabilities. Previous studies [10] on this strategy have yielded encouraging results. This strategy improves performance in the Job Shop Scheduling environment by assisting the agent in concentrating on understanding the best course of action while keeping legal activities in check.

Table 3.1: Instances of Lawrence's Datasets

Dataset	**Jobs**	**Machine**	**Lower bound**	**Upper bound**
la20	10	10	902	902
la25	15	10	977	977
la30	20	10	1355	1355
la35	30	10	1888	1888
la40	15	15	1222	1222

Table 3.2: Instances of Demirkol's Datasets

Dataset	**Jobs**	**Machine**	**Lower bound**	**Upper bound**
dmu16	30	20	3734	3751
dmu21	40	15	4280	4380
dmu26	40	20	4647	4647
dmu31	50	15	5640	5640
dmu36	50	20	5621	5621

3.2 Datasets

For the comparative analysis, this thesis considers three different benchmark datasets:

3.2.1 Lawrence Datatset

A popular benchmark dataset for the JSSP is the Lawrence dataset [13], which was first presented by Stephen Lawrence in 1984. This dataset, which consists of 40 cases that are further classified as small, medium, and large, offers a wide range of problem sizes and complexities, with 10 to 30 jobs and 10 to 15 machines. Every instance offers a thorough depiction of actual production settings by specifying a range of tasks, equipment, processing times for operations, and constraints. For the purpose of this experiment, a subset of five instances is selected, representing different problem sizes, to evaluate the performance of the proposed approach which is listed in Table 3.1.

3.2.2 Dermikol Dataset

The Demirkol dataset [6], introduced by Demirkol in 1998, is another widely-used benchmark dataset for the JSSP. Consisting of 120 instances, this dataset provides a diverse range of problem sizes and complexities, with 10 to 100 jobs and 5 to 20 machines. For the purpose of this experiment, a subset of five instances is selected, representing high complexity problem sizes, to evaluate the performance of the proposed approach which is listed in Table 3.2.

3.2.3 Taillard Dataset

The Taillard dataset [30], introduced by Éric Taillard in 1993, is a widely-used benchmark dataset for the JSSP. Consisting of 80 instances, this dataset provides a diverse range of problem sizes and complexities, with 15 to 100 jobs and 5 to 20 machines. The Taillard dataset is known for its challenging instances, which have been used to evaluate the performance of various JSSP algorithms and heuristics. Table 3.3 contains instances from Taillard's dataset. To assess the efficacy of the suggested method, a subset of 13 cases that show a good level of instance size complexity are chosen for this experiment.

Table 3.3: Instances of Taillard's Datasets

Dataset	**Jobs**	**Machine**	**Lower bound**	**Upper bound**
ta40	30	15	1651	1669
ta41	30	20	1606	2005
ta42	30	20	1884	1937
ta50	30	20	1833	1923
ta51	50	15	2760	2760
ta52	50	15	2756	2756
ta60	50	15	2723	2723
ta61	50	20	2868	2868
ta62	50	20	2869	2869
ta70	50	20	2995	2995
ta71	100	20	5464	5464
ta72	100	20	5181	5181
ta80	100	20	5183	5183

3.3 Ray

An open-source framework called Ray [16] was created to grow Python and AI applications effectively, especially for jobs like machine learning. It eliminates the need for in-depth knowledge of distributed systems and accelerates parallel processing. With scalable frameworks for operations like data preprocessing, distributed training, hyperparameter tuning, RL, and model serving, Ray reduces the complexity of managing distributed machine learning processes. Ray provides an efficient solution for the whole machine learning lifecycle, regardless of whether you're training models, adjusting hyperparameters, or deploying models at scale.

Ray's core components include Ray Core, which provides a distributed execution framework, and Ray Tune, which offers hyperparameter tuning. Ray Core is a distributed execution framework that allows you to run Python functions concurrently on multiple cores or machines. Ray Tune is a scalable hyperparameter tuning library that can be used with any machine learning framework. Ray Tune provides a simple interface for tuning hyperparameters and supports various search algorithms, including random search, grid search, and Bayesian optimization. Ray Tune also integrates with popular machine learning libraries like TensorFlow, PyTorch, and Scikit-learn.

3.4 Weights & Biases

Weights & Biases (WandB) [4] stands as a comprehensive platform tailored to enhance machine learning experimentation. Its suite of tools and services empowers researchers and practitioners by streamlining the process of logging and visualizing crucial metrics, including hyperparameters and outputs. The platform prioritizes experiment monitoring, providing users with a comprehensive understanding of their model's performance dynamics. WandB's effectiveness extends to hyperparameter optimization, offering a range of techniques such as random search, grid search, and Bayesian optimization. This breadth of options allows users to fine-tune their models efficiently, optimizing performance for diverse tasks.

Additionally, WandB facilitates smooth teamwork by offering capabilities that make code sharing and result distribution easier. Its interfaces with well-known deep learning frameworks, such as PyTorch and TensorFlow, facilitate easy workflow integration and guarantee tool compatibility. WandB carefully records experiment details and code versions in order to prioritize experiment reproducibility above and beyond facilitating cooperation. Proactive alerts and real-time monitoring provide crucial information during model training, allowing for timely modifications and optimizations. WandB is a vital tool in the constantly changing field of machine learning research, enabling researchers to push the limits of creativity and innovation with tasks ranging from model comparison and hyperparameter tuning to result display.

3.5 Implementation

The open-source RL library (RLlib) supports highly distributed, production-level RL workloads while preserving uniform and straightforward APIs for a wide range of industry applications. In this thesis, RLlib and Tenserflow are implemented on the JSSP environment [11]. WandB [4] Bayesian optimization is employed to conduct hyperparameter searches and also logs all the resulting data. The agent is trained using four well-known RL algorithms: PG, PPO, A2C, and A3C. The training is conducted on a single NVIDIA H100 Tensor Core Graphics Processing Unit(GPU), ensuring efficient computation. The makespan metric is used to evaluate the performance of the trained models. The results from the different algorithms are compared to determine the best-performing approach for the JSSP. This comprehensive setup, integrating powerful tools and methodologies, aims to provide a thorough analysis of RL algorithms' effectiveness in solving complex scheduling problems.

Chapter 4

Experiments and Results

4.1 Introduction

This chapter presents a comprehensive analysis of experiments conducted to evaluate the performance of different RL algorithms in solving the JSSP. The experiments aim to address the research questions and test the hypotheses outlined in Section 1.3. The chapter is structured as follows:

1. **Experimental Configurations**: The settings and parameters used in training the RL models, including datasets, network architectures, and hyperparameters, are detailed. This section addresses **RQ1** by explaining the implementation of DRL models for JSSP.
2. **Baseline Comparison**: The performance of RL algorithms is benchmarked against traditional dispatching rules and state-of-the-art methods. This comparison addresses **RQ2** and validates **H3** and **H4**.
3. **Performance Analysis**: The results are analyzed across three datasets: Lawrence's, Demirkol's, and Taillard's. Each dataset is evaluated to demonstrate how RL models perform on varying problem sizes and complexities. This analysis addresses **RQ2** and tests **H1**, **H2**, **H3**, and **H4**.
4. **Performance Metrics**: Makespan and reward metrics are used to evaluate the efficiency of RL algorithms, crucial for testing **H1** and **H2**.
5. **Comparative Analysis**: Optimality gaps and makespan values are compared across different RL algorithms and benchmarks, identifying the most effective approaches and testing **H3** and **H4**.

This chapter systematically organizes the results and provides in-depth analyses to offer a clear understanding of the experiments and the performance of various RL algorithms in solving the JSSP. The insights gained will contribute to answering the research questions and validating the hypotheses, advancing the development of efficient scheduling algorithms.

4.2 Experimental Configurations

The model is trained on scheduling instances of the sizes used on Taillard's [30], Demirkol's [6], and Lawrence's [13] datasets. The examples are written as JSSP $m \times n$, where m is the number of jobs and n is the number of machines. Standard benchmark instances ranging in size from 10×10 to 100×20 are used for static experiments to evaluate the method's performance; these examples are included in Table 3.3, Table 3.2 and Table 3.1.

For PPO, the approach used in this thesis involves constructing distinct Multi-Layer Perceptron architectures for the state-value prediction network and the action selection network.

These networks utilize ReLU as an activation function and consist of two hidden layers with a size of 256 neurons on each. Additionally, to maintain the unique functionality of these networks, the layers are not shared between them. Throughout the training, PPO-specific parameters are carefully adjusted, including the clipping parameters of 0.5, the number of epochs 10 for network updates, and the coefficients controlling policy loss and value function of 0.5 and 0.8 respectively. A linear decay scheduler is applied, gradually reducing the learning rate from 6.6×10^{-4} to 7.8×10^{-5} and the entropy coefficient from 2.0×10^{-3} to 2.5×10^{-4} over the training period. For both the actor and the critic networks, the Adam optimizer is employed with the discount factor γ to 1. The hyperparameters like the batch size and the dimension of the actor network's hidden layers are all carefully chosen.

The experimental configuration for PG, A2C, and A3C is similar to that of PPO; it consists of distinct MLP architectures with two hidden layers of size 256 each for action selection networks and state-value prediction networks with ReLU activation functions. The networks' distinct functions are preserved by their refusal to share layers. Linear decay scheduling is used to optimize common hyperparameters such as learning rate, discount factor γ, and entropy coefficient. The Adam optimizer is used for both actor and critic networks. Unlike PPO, these algorithms do not require specific parameters such as clipping or loss coefficients. However, Optimizing performance across all methods requires careful consideration of actor-network dimensionality and batch size selection.

For PPO, Lawrence's instances, being of small sizes, are run for 20 iterations. Demirkol's and Taillard's instances are run for 60 iterations, except for the 100×20 instances. However, the 100×20 size instances are run for 200 iterations due to their complexity. For PG, 30 iterations are run for Taillard's instances, 40 iterations for Lawrence's instances, and 60 iterations for Demirkol's instances. A2C is run for 60 iterations for all instances, and A3C is run for 460 iterations for all instances.

4.3 Baseline

The agent is compared to three well-known dispatching rules from the literature to ensure that it is capable of learning difficult dispatching strategies. These dispatching rules are typically effective because they demonstrate greediness and balance the distribution of jobs. *First In First Out* (FIFO) implies that each machine processes the job that entered the system first. *Most Work Remaining* (MWKR) selects the job with the highest remaining processing time for prioritization. *Shortest Processing Time* (SPT), commonly used to minimize job completion time, prioritizes jobs based on their shortest processing time. These rules serve as benchmarks to evaluate the agent's ability to handle complex scheduling scenarios and learn effective dispatching strategies.

The four benchmarks from the literature are selected respectively by Zhang et al. [37], Park et al. [26], Han et al. [9], and Wu et al. [34], and they evaluate their approaches on the same instances as those used in this thesis. In paper [37], the authors used DRL with graph neural networks and suggested an automatic technique to train Priority Dispatching Rules for the Job-shop scheduling problem. An RL-based real-time scheduler, ScheduleNet, was proposed in work [26] which solved the various types of multiagent scheduling problems. In work [9], the researcher proposed an adaptive scheduling framework for job shop scheduling utilizing DRL that used disjunctive graphs in conjunction with dueling networks, double DQN, and prioritized experience replay. The authors in paper [34] proposed a DRL approach utilizing Proximal Policy Optimization with hybrid prioritized experience replay for the Dynamic Job-Shop Scheduling Problem. To approximate the optimal solution and estimate the optimality gap of all algorithms, the constraint programming solver of Google OR-Tools CP-SAT [27] is utilized.

4.4 Performance Analysis

In this work, the different RL algorithms are implemented on three different datasets: Taillard [30], Dermikol [6], and Lawrence [13]. These results are curious because they highlight an important finding: the performance outcomes of the three datasets show notable similarities even when the same hyperparameters as those used for Taillard's instances are used, and Demirkol's and Lawrence's instances are not included in the hyperparameter search process. This observation is especially significant in considering the problem size similarities between Taillard's, Demirkol's, and Lawrence's situations, particularly about the number of procedures that need to be planned.

This result implies that this method demonstrates a high degree of generalizability. The approach performs consistently even when faced with examples from different datasets, proving its ability to adapt and flexibility in a range of problem scenarios. This ability to maintain similar performance levels on many datasets highlights the approach's effectiveness and adaptability to a range of optimization problems. The comparative analysis of results on each of these datasets is detailed next.

4.4.1 Performance Matrix

When evaluating the performance of RL algorithms in JSSP, two primary metrics are commonly considered: makespan and reward. The makespan, sometimes referred to as the completion time, shows how long it took to finish every task on the timetable. The goal of JSSP is to reduce the makespan, which measures how well the scheduling algorithm performs in finishing all jobs in the least amount of time. Therefore, when the agent learns to discover the best scheduling options that reduce completion time, the makespan should ideally decrease as the training iterations go on. To evaluate algorithms performance, the average optimality gap (opt_{gap}) [14] is also calculated using Eq. (4.1)

$$opt_{gap} = \frac{makespan - makespan^*}{makespan^*} \times 100, \tag{4.1}$$

where "*makespan*" is the makespan obtained from different algorithms, and "*makespan**" is either optimal or the best-known solution. The optimal solutions are derived using Or-Tools [27]. The optimality gap is a useful metric for evaluating how well an algorithm finds approximate optimal solutions to scheduling problems. While a bigger optimality gap can suggest that the algorithm's performance could be enhanced, a lower optimality gap shows that the solutions produced by the algorithm are closer to the optimal or best-known solutions.

In addition to the makespan, reward serves as another important performance metric in RL-based approaches. The agent is intended to be guided toward actions that result in desired results via the reward function. A more cumulative reward in the context of JSSP suggests that the agent is taking in information from its surroundings and making choices that improve scheduling results. The agent modifies its policy to maximize cumulative reward as it gets feedback from the environment through rewards, which eventually improves performance in terms of reduces makespan and increases overall scheduling efficiency.

Table 4.1: Make-span of Lawrence's instances by different approach.

Dataset	**PPO**	**PG**	**A2C**	**A3C**
la20	**912**	947	1024	1052
la25	**1022**	1113	1206	1343
la30	**1356**	1567	1630	1759
la35	**1895**	2043	2113	2216
la40	**1323**	1457	1519	1614

4.4.2 Performance Analysis on Lawrence's Dataset

Table 4.1 and Table 4.2 present the makespan of Lawrence's examples using different methods. A particular instance is represented by each row, which is distinguished by prefixes like "la20", "la25", and so forth. For comparison, the columns show various dispatching rules, RL algorithms, and other cutting-edge techniques. The tables show the makespan that each algorithm or method achieves for each instance. The values that are bolded signify the optimal makespan that was attained for that specific algorithm. These tables provide a thorough analysis of how various approaches performed in distinct Lawrence cases, shedding light on the relative merits of various strategies for resolving scheduling problems.

Comparison of RL Algorithms Based on Makespan

The baselines for PPO, PG, A2C, and A3C RL algorithms on Lawrence's dataset are compared in Tabel 4.1. Lawrence instances of various sizes, listed in Table 3.1, were used for the investigation. Lower makespan values indicate better performance, as they represent shorter total processing times for the scheduling instances. A clear trend emerges from the comparison: as the instance size increases, so does the makespan, suggesting that larger instances are more challenging to solve. Among the RL algorithms, PPO consistently outperforms the others, demonstrating its superior performance. PG follows closely behind PPO, while A2C and A3C trail behind.

It is shown in Fig. 4.1, Fig. 4.2, and Fig. 4.3 that the makespan of Lawrence's instances of RL algorithms: PPO, PG, and A2C, respectively. Each figure displays the makespan on the y-axis and the number of iterations on the x-axis. The plots demonstrate the learning process of the agent, where the makespan decreases as the number of iterations increases. This downward trend indicates that the agent is learning and improving its scheduling strategy, resulting in reduced makespans. However, as the instance sizes are increased, the makespan also increases, indicating that larger instances are more challenging to solve. Notably, PPO exhibits superior performance compared to PG, A2C, and A3C, which is why we focus on PPO for further comparison and analysis.

PPO performs better than other methods because it can effectively explore the action space and learn a more precise policy. The clipped surrogate objective function in PPO aids in preventing significant policy modifications, which occasionally have negative consequences. PPO may learn a more stable and dependable policy because to this clipping process, which improves performance in challenging scheduling issues like the ones in Lawrence's dataset. PPO performs better still because of its trust region optimization technique, which also strengthens the policy updates' robustness. On the other hand, as PG, A2C, and A3C algorithms are more prone to exhibiting unstable behavior or becoming stuck in local optima, they might have trouble with bigger instance sizes.

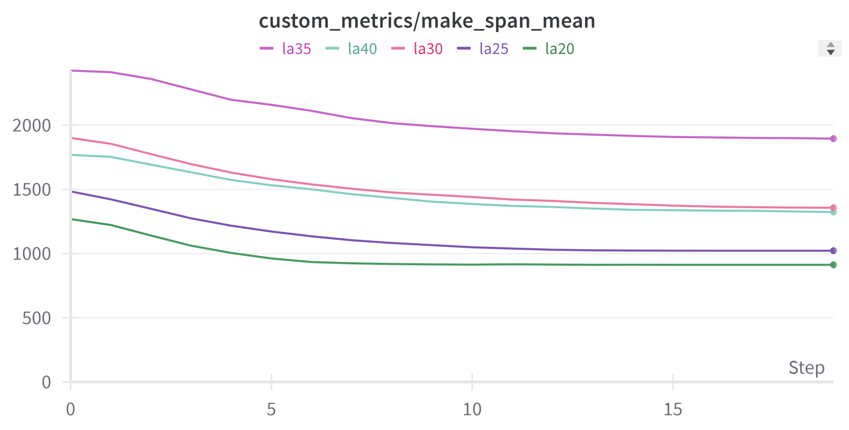

Figure 4.1: Makespan of the Lawrence's dataset of PPO.

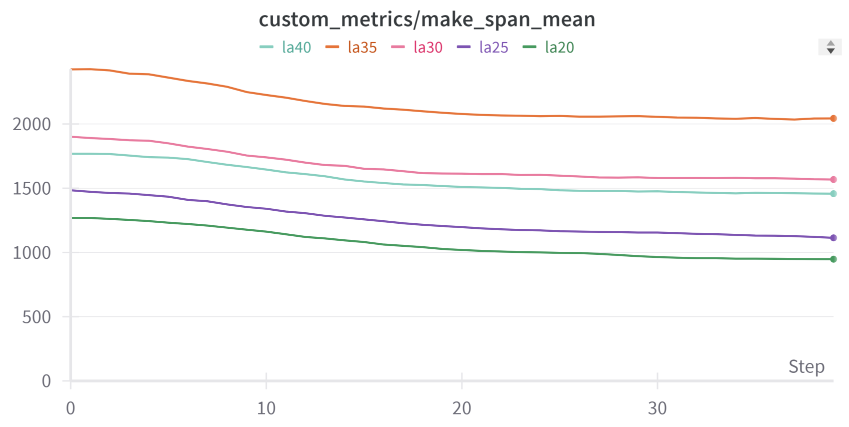

Figure 4.2: Makespan of the Lawrence's instances of PG.

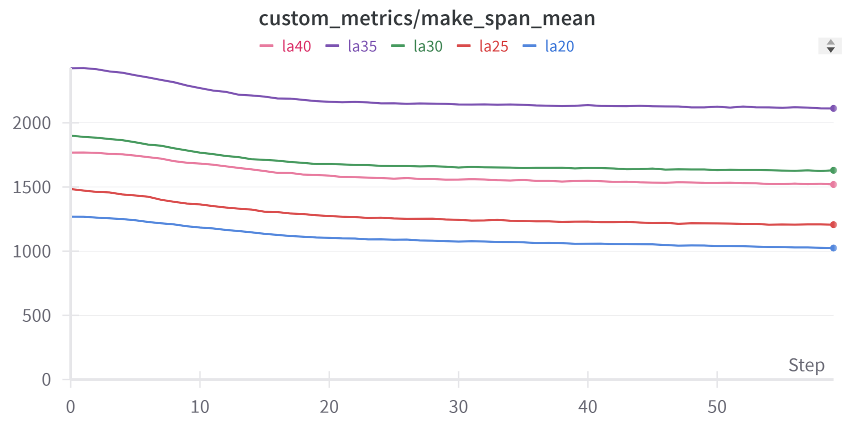

Figure 4.3: Makespan of the Lawrence's instances of A2C.

Comparison of Baselines based on Makespan

It is presented in Table 4.2 a comparative analysis of various approaches described in Sec. 4.3 on the Lawrence dataset, showcasing their performance in terms of makespan. Comparing different RL approaches is challenging because they often have different goals, settings, and algorithms. These variations can make it difficult to assess their performance accurately and meaningfully. But stated in paper [9], the agent was trained using the same benchmark instances, allowing for a more direct comparison, their comparison provides an improved

Table 4.2: The comparison results on Lawrence's instances with different benchmarks.

Dataset	**PPO**	**FIFO**	**MWKR**	**SPT**	**Park et al. (2021)**	**Han et al. (2020)**	**Or Tool**
la20	**912**	1272	1059	1262	-	-	902
la25	**1022**	1283	1203	1253	1117	1067	980
la30	**1356**	1648	1533	1775	1490	1417	1355
la35	**1865**	2138	2073	2464	1969	1941	1800
la40	**1323**	1435	1450	1481	1350	1336	1222

Table 4.3: The optimality gap Lawrence's instances with different benchmarks.

Dataset	**PPO**	**FIFO**	**MWKR**	**SPT**	**Park et al. (2021)**	**Han et al. (2020)**
la20	1.10	41.02	17.41	39.91	-	-
la25	4.29	30.92	22.76	27.86	19.98	8.88
la30	0.07	21.62	13.14	31	9.96	4.58
la35	3.61	18.78	15.17	36.78	9.39	7.83
la40	8.27	17.43	18.66	21.19	10.47	9.33
Average	**3.47**	**25.95**	**17.42**	**31.35**	**10.95**	**7.65**

evaluation.

The results demonstrate that the proposed PPO-based tool outperforms the traditional scheduling algorithms, including FIFO, MWKR, and SPT, across all instance sizes. Notably, PPO achieves the lowest makespan values in most cases, highlighting its effectiveness in solving complex scheduling problems. In comparison to the state-of-the-art approaches, PPO demonstrates superior performance, outdoing [26] and [9] in most instances.

The optimality gap of Lawrence instances for PPO and other techniques detailed in Sec. 4.3 is displayed in Table 4.3 and calculated using Eq. (4.1). The average optimality gap is calculated for the comparisons of the algorithms and from Table 4.3, PPO has a lower average optimality gap than other approaches means PPO is closer to the optimal or best-known solutions. The PPO-based approach significantly outperforms the traditional scheduling algorithms, FIFO, MWKR, and SPT, with an average optimality gap of 3.47, compared to 25.95, 17.42, and 31.35, respectively. Notably, the PPO-based approach achieves a 6-9 times lower optimality gap than the traditional algorithms. In comparison to the state-of-the-art approaches, the PPO-based approach significantly outperforms [26] and [9], with an average optimality gap of 3.47, compared to 10.95 and 7.65, respectively. Remarkably, the PPO-based approach achieves a 2-3 times lower optimality gap than the state-of-the-art approaches.

Reward of Lawrence dataset

It is depicted in Fig. 4.4, Fig. 4.5, and Fig. 4.6 that the reward of all Lawrence's instances during the training of the agent using PPO, PG, and A2C algorithms, respectively. Each figure displays the reward on the y-axis and the number of iterations on the x-axis. Initially, the reward is very low, with some instances even exhibiting negative rewards, but as the agent is trained more, the reward increases, indicating that the agent is learning the environment. It is evident from the figures that as the training iterations increase, the reward also increases, as clearly shown in all figures. Moreover, the PPO algorithm consistently achieves higher rewards compared to PG and A2C, demonstrating its superiority in finding optimal solutions.

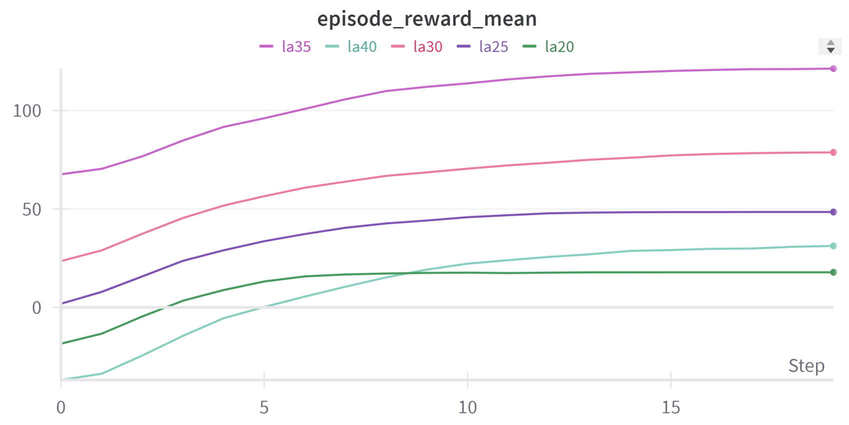

Figure 4.4: Reward of the Lawrence's dataset of PPO.

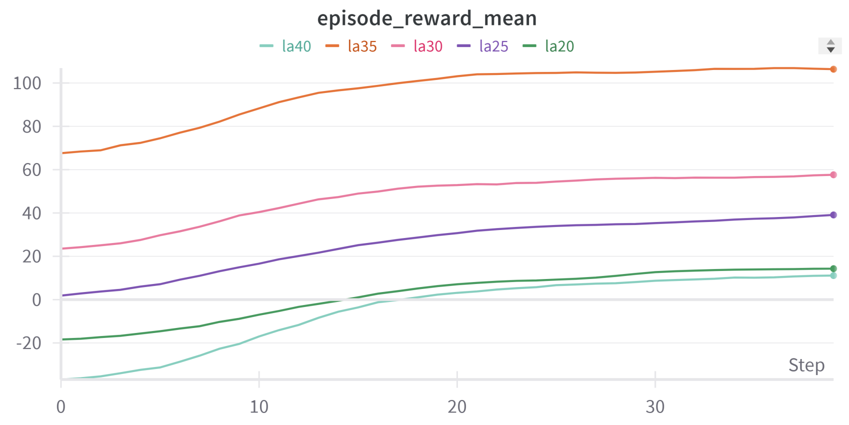

Figure 4.5: Reward of the Lawrence's instances of PG.

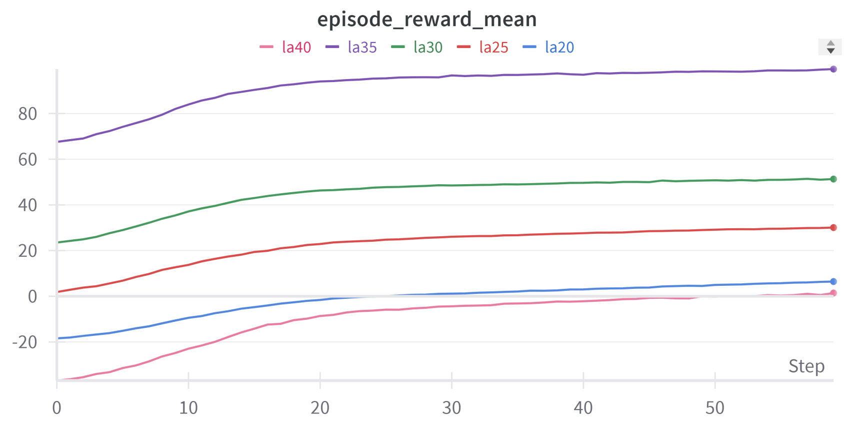

Figure 4.6: Reward of the Lawrence's instances of A2C.

4.4.3 Performance Analysis on Dermikol Dataset

Table 4.4 and Table 4.5 showcase the makespan of Dermikol's instances employing various approaches. Each row corresponds to a specific instance, identified with prefixes such as "dmu16", "dmu21", and so on. The columns represent different RL algorithms, dispatching rules, and other state-of-the-art methods for comparison purposes. The tables show the makespan that each algorithm or method achieves for each instance. The values that are bolded signify the optimal makespan that was attained for that specific algorithm. These tables provide a thorough analysis of how various strategies performed over different Dermikol instances, shedding light on how well various techniques addressed scheduling issues.

Comparison of RL Algorithms Based on Makespan

Table 4.4 shows the comparison of the baseline for RL algorithms, specifically PPO, PG, A2C, and A3C, on Demirkol's dataset. The study was conducted using Dermikol instances listed in Table 3.2, encompassing instances of all sizes. It is observed that the PPO algorithm consistently outperforms the other algorithms in terms of makespan. As expected, the makespan for all considered algorithms increases as the instance size increases. Notably, PG and A2C emerge as tough competitors with only marginal differences, whereas A3C performs the least among all algorithms. Since PPO performs better than other algorithms, it is used as the reference algorithm for the remaining comparisons with other benchmarks.

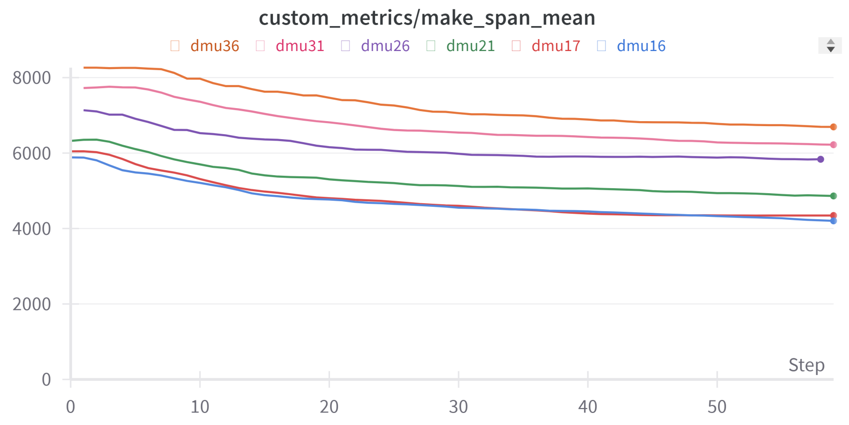

Figure 4.7: Makespan of the Dermikol's instances of PPO.

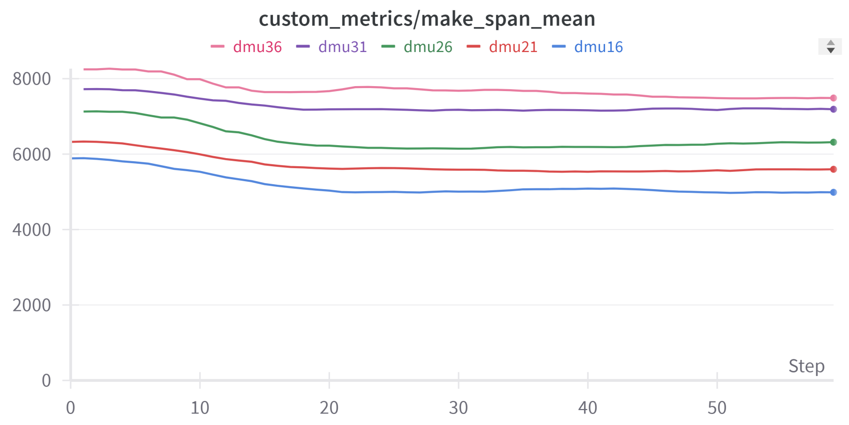

Figure 4.8: Makespan of the Dermikol's instances of PG.

It is shown in Fig. 4.7, Fig. 4.8, and Fig. 4.9 that the makespan for the Dermikol instances for PPO, PG, and A2C, respectively. Each figure displays the makespan on the y-axis and the number of iterations on the x-axis. It is evident that as the number of iterations increases, the average makespan of all instances decreases. This observation suggests that the agent is

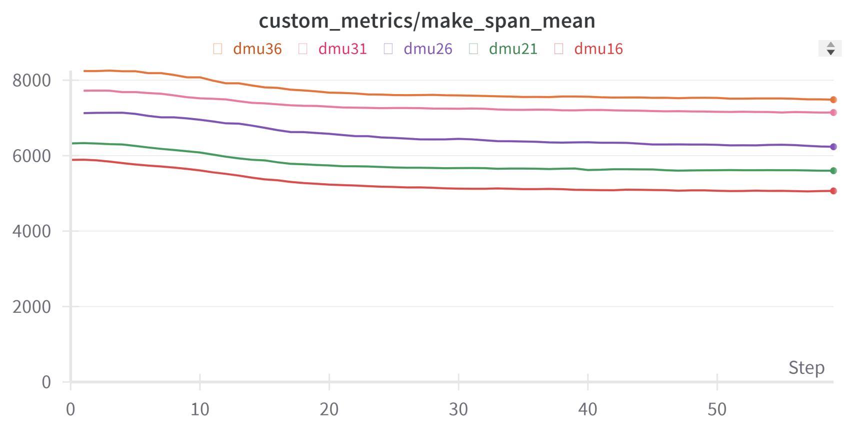

Figure 4.9: Makespan of the Dermikol's instances of A2C.

Table 4.4: Make-span of Demirkol's instances by different approach.

Dataset	PPO	PG	A2C	A3C
dmu16	**4203**	4987	5065	5874
dmu21	**4863**	5598	5600	6363
dmu26	**5835**	6316	6235	7153
dmu31	**6221**	7189	7143	7729
dmu36	**6693**	7487	7482	8250

learning from the environment and improving its performance over time.

From the graphs, it's apparent that the agent using PPO is learning, as evidenced by the decreasing trend in makespan with increasing iterations. However, upon analyzing the graphs for PG and A2C, it's evident that the agent's learning plateaus after 25 iterations, as the makespan remains relatively constant thereafter. This observation suggests that these agents may not benefit significantly from additional training beyond 25 iterations. Therefore, it can be concluded that these agents may be effectively trained up to 25 iterations to achieve optimal performance.

Comparison of Baselines based on Makespan

Table 4.5 presents a comparison between PPO, other approaches described in Sec. 4.3. Here, PPO consistently outperforms dispatching rules except for one instance, dmu26, where the dispatching rule MWKR performs well. PPO exceeded their findings in all but one of these experiments. It's crucial to remember that they used Dueling Double DQN, a separate RL algorithm that uses various settings and functions on off-policy, value-based principles. Despite these variations, this comparison provides insightful information on how well PPO performs in relation to other RL algorithms on the same benchmark instances. FIFO and SPT generally perform poorly compared to the other algorithms.

Table 4.6 displays the optimality gap of Dermikol's instances for PPO and other strategies described in Sec.4.2. The gap is computed by Eq. (4.1). PThe results highlight PPO's superior performance, with an average optimality gap of 12.13, significantly outperforming traditional scheduling algorithms FIFO, MWKR, and SPT, which have average optimality gaps of 25.39, 16.54, and 34.83, respectively. Meanwhile, [37] and [9] exhibit average optimality gaps of 32.97 and 16.32, respectively. Overall, the table underscores PPO's ability to achieve near-optimal solutions, outperforming both traditional and state-of-the-art ap-

Table 4.5: The comparison results on Demirkol's instances with different benchmarks.

Dataset	PPO	FIFO	MWKR	SPT	Zhang et al. (2020)	Han et al. (2020)	Or-Tool
dmu16	**4203**	4934	4550	4990	5876	4414	3811
dmu21	**4863**	5674	5325	6378	5314	5255	4380
dmu26	5835	6125	**5567**	6725	6241	5695	4986
dmu31	**6221**	6817	6523	7666	6639	6588	5642
dmu36	**6693**	7422	6837	7577	7328	6859	5973

Table 4.6: The optimality gap on Demirkol's instances with different benchmarks.

Dataset	PPO	FIFO	MWKR	SPT	Zhang et al. (2020)	Han et al. (2020)
dmu16	10.29	29.47	19.39	30.94	54.19	15.82
dmu21	11.03	29.54	21.58	45.62	21.32	19.98
dmu26	17.03	22.84	11.65	34.88	25.17	14.22
dmu31	10.26	20.83	15.62	35.87	17.67	16.77
dmu36	12.05	24.26	14.47	26.85	46.52	14.83
Average	**12.13**	**25.39**	**16.54**	**34.83**	**32.97**	**16.32**

proaches. Notably, the PPO-based approach achieves a 2-3 times lower optimality gap than the traditional algorithms and state-of-the-art approaches.

Reward of Dermikol Dataset

It is shown in Fig. 4.10 the reward of all instances during the training process for the PPO algorithm. The figure displays the reward on the y-axis and the number of iterations on the x-axis. From the graph, it can be observed that the reward value for each instance increases with the number of training iterations. This trend suggests that the agent is learning from the environment, as it is achieving higher rewards over time, indicating improvement in its performance. Fig. 4.11 and Fig. 4.12 also depict that as the number of iterations increases, the reward also increases for PG and A2C algorithms respectively.

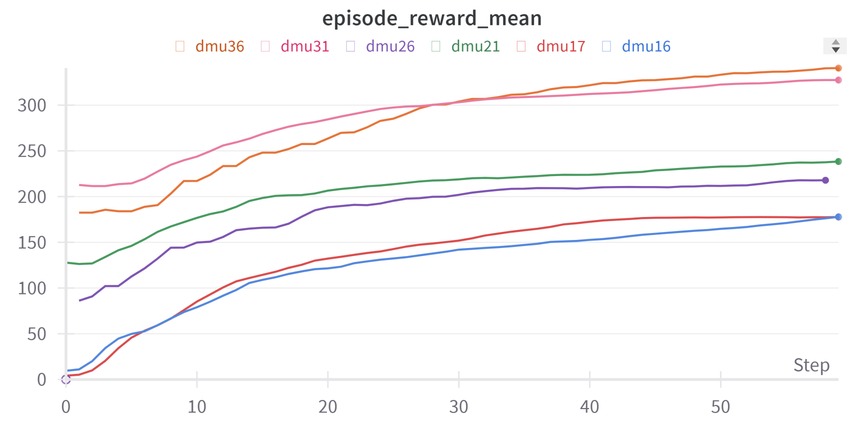

Figure 4.10: Reward of the Dermikol's dataset of PPO.

However, upon analyzing Fig. 4.11 for the PG algorithm and Fig. 4.12 for the A2C algorithm, it is apparent that the reward stagnates after 25 iterations for both algorithms. This ob-

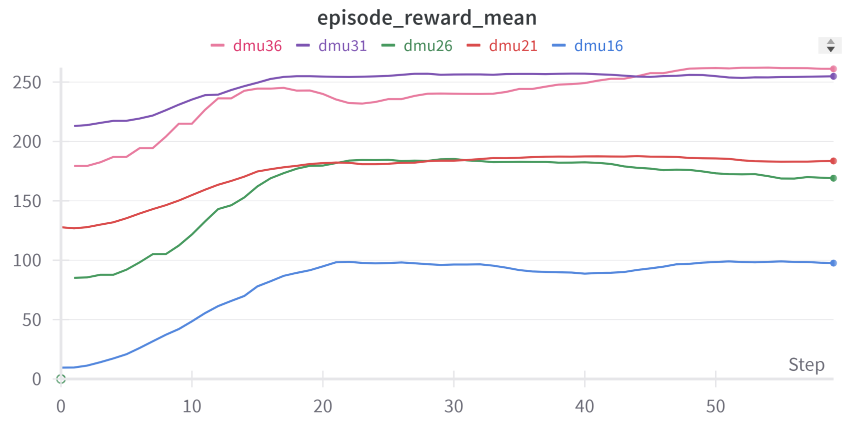

Figure 4.11: Reward of the Dermikol's instances of PG.

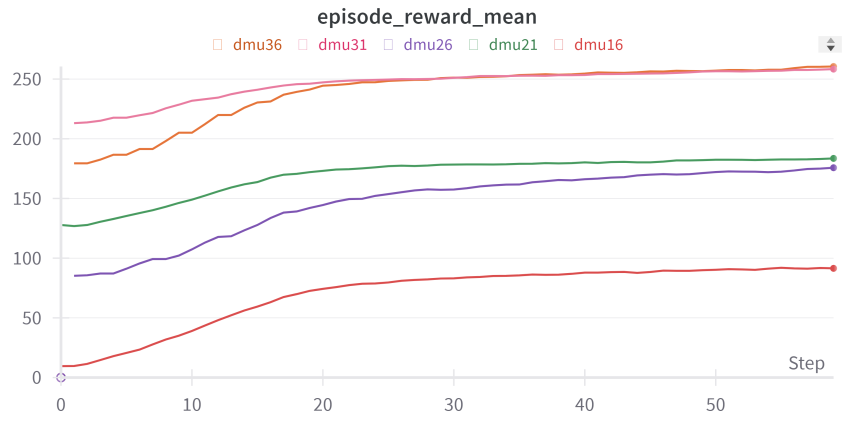

Figure 4.12: Reward of the Dermikol's instances of A2C.

servation implies that the agents may not derive significant benefit from continued training beyond this point. Therefore, it can be inferred that these agents may be effectively trained up to 25 iterations to achieve optimal performance.

4.4.4 Performance Analysis on Taillard Dataset

Table 4.7 and Table 4.8 display the makespan of Taillard's instances using various methods. Each row represents a specific instance, labeled with prefixes like "ta40", "ta41", etc. The columns include different RL algorithms, dispatching rules, and state-of-the-art methods for comparison. The best makespan for each instance is indicated by bold values. These tables offer insights into effective scheduling approaches across Taillard instances.

Comparison of RL Algorithms Based on Makespan

Table 4.7 shows the comparison of the baseline for RL algorithms, specifically PPO, PG, A2C, and A3C, on Tailllard's dataset. The study was conducted using Taillard instances listed in Table 3.3, encompassing instances of all sizes. Based on the data presented in Table 4.7, it is evident that the PPO algorithm consistently demonstrates superior performance in terms of makespan, similarly observed across the Dermikol instances. After PPO, both PG and A2C exhibit comparable performance across various instances, with A2C demonstrating superior results specifically in the 100×20 instances compared to PG. A3C consistently underperforms across all instances except for the 100x20 scenarios, where its performance

Table 4.7: Make-span of Taillard's instances by different approach.

Dataset	**PPO**	**PG**	**A2C**	**A3C**
ta40	**1830**	2222	2243	2600
ta41	**2115**	2697	2711	3096
ta42	**2146**	2625	2631	3046
ta50	**2178**	2575	2599	2982
ta51	**3066**	3355	3454	3728
ta52	**2957**	3402	3425	3724
ta60	**2997**	3359	3370	3660
ta61	**3456**	3754	3779	4154
ta62	**3485**	3776	3821	4211
ta70	**3428**	3937	3854	4225
ta71	**6134**	6527	6441	6678
ta72	**5860**	6385	6082	6263
ta80	**6009**	6328	6247	6409

aligns with that of PG.

Overall, this analysis underscores the importance of algorithm selection in RL tasks. Different algorithms may yield varying performance depending on the problem instance characteristics. Hence, it's crucial to explore multiple algorithms to identify the most suitable one for a given problem domain. Additionally, gaining insights into the reasons behind algorithm performance variations can inform future research and algorithm development endeavors. Since PPO performs better than other algorithms, it is used as the reference algorithm for the remaining comparisons with other benchmarks.

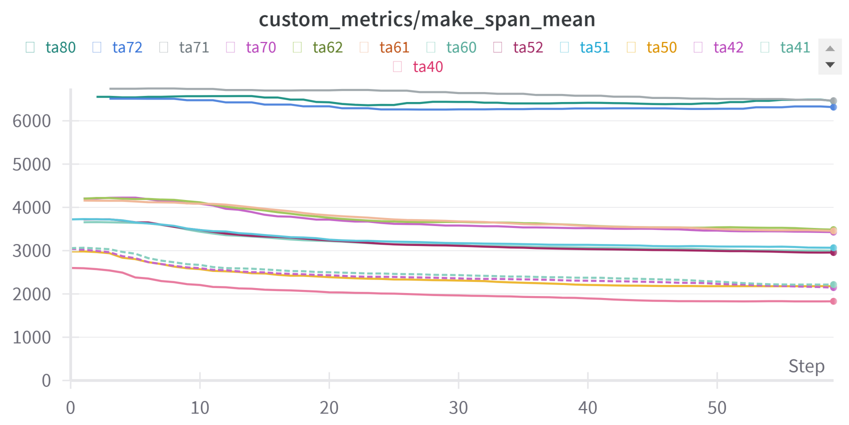

Figure 4.13: Makespan of the Taillard's instances of PPO.

Using PPO, the agent is trained for 60 iterations across all instances of varying sizes. Fig. 4.13 illustrates the makespan of PPO for all instances with corresponding iterations. The figure displays the makespan on the y-axis and the number of iterations on the x-axis. From the figure, it is evident that the makespan decreases for all instances as the number of iterations increases, except for the instance size 100×20, where the makespan remains constant across all iterations. This suggests that the agent struggles to learn this complex environment with only 60 iterations of training. Consequently, the agent undergoes training for 200 iterations, resulting in improved learning, as depicted in Fig. 4.14. The makespan decreases as the number of training iterations increases. Fig. 4.15 and Fig. 4.16 also illustrate that as the

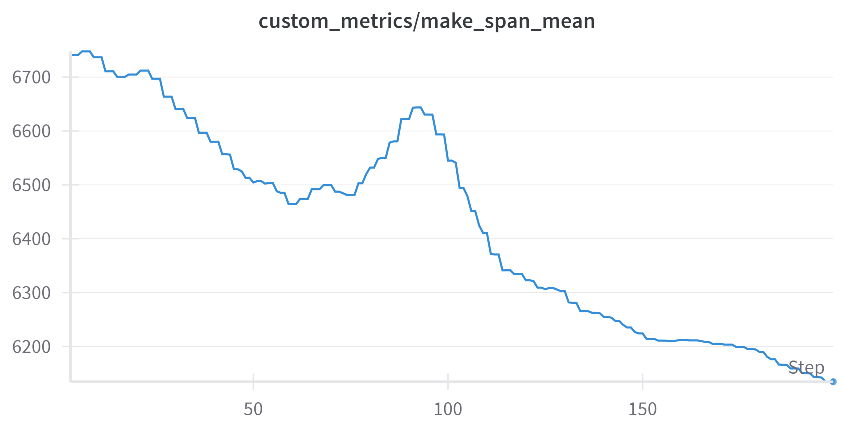

Figure 4.14: Makespan of the Taillard's instances "ta71" of PPO.

number of iterations increases, the makespan decreases for the PG and A2C algorithms, respectively, for large-size instances.

It is distinctly illustrated in Fig. 4.14 that a trend where the makespan initially increases after 60 iterations, followed by a subsequent decrease post the 100th iteration. This observation suggests that the agent requires further training beyond the 100th iteration to achieve a reduced makespan. Consequently, a decision was made to train the agent for an additional 100 iterations, totaling 200 iterations in all. This extended training period aims to optimize the agent's performance and ultimately minimize the makespan, thus enhancing overall efficiency.

From Fig. 4.15, it is seen that for the agent using PG, the makespan is lowest at approximately 22 iterations and then slightly increases. This indicates that the agent's performance peaks early, and additional training beyond 22 iterations may not lead to further improvements in minimizing the makespan. Similarly, from Fig. 4.16, it is observed that for the agent using A2C, the makespan is lowest at approximately 55 iterations and then slightly increases. This pattern indicates that the performance of the PG and A2C agents peaks early, and additional training beyond these points may not lead to further improvements in minimizing the makespan. This trend is particularly true for higher instance sizes.

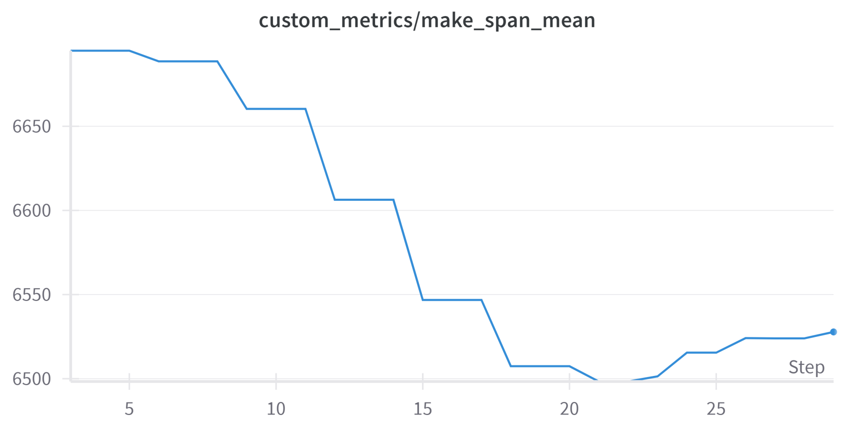

Figure 4.15: Makespan of the Taillard's instances "ta71" of PG.

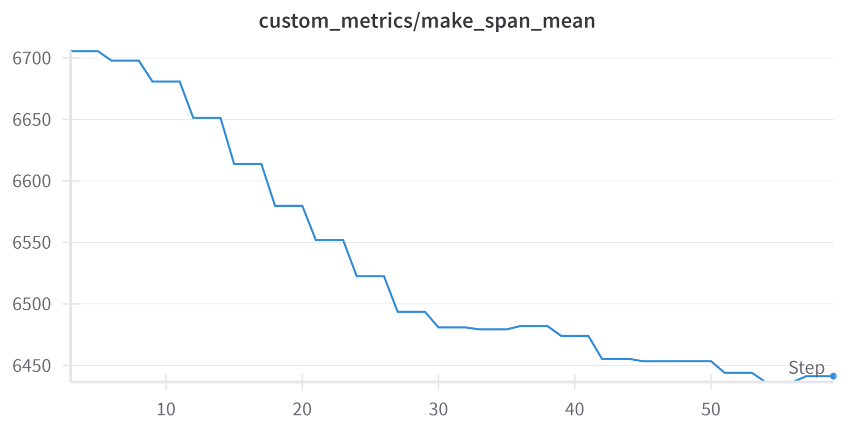

Figure 4.16: Makespan of the Taillard's instances "ta71" of A2C.

Table 4.8: The comparison of makespan on Taillard's instances with different benchmarks.

Dataset	**PPO**	**FIFO**	**MWKR**	**SPT**	**Zhang et al. (2020)**	**Han et al. (2020)**	**Wu et al. (2024)**	**OR Tool**
ta40	**1830**	2069	2093	2398	2140	-	-	1752
ta41	**2115**	2543	2632	2826	2667	2450	2398	2096
ta42	**2146**	2578	2401	2783	2664	2351	2305	2020
ta50	**2174**	2531	2496	2712	2608	-	-	1978
ta51	**3066**	3549	3585	3457	3599	3263	3155	2760
ta52	**2957**	3339	3451	3458	3341	3229	3056	2756
ta60	**2997**	3129	3103	3593	3352	-	-	2726
ta61	**3456**	3625	3471	4030	3654	3367	-	2973
ta62	**3485**	3652	3489	4075	3722	3489	-	3042
ta70	**3428**	3784	3559	3959	3643	-	-	3102
ta71	6134	6161	5948	6651	6452	**5908**	-	5790
ta72	5860	**5610**	5625	6506	5695	5746	-	5531
ta80	6009	5724	**5519**	6472	5892	-	-	5609

Comparison of Baselines based on Makespan

Table 4.8 presents the makespan of all instances, including the makespan by the best PPO than other RL algorithms and other approaches described in Sec. 4.3. The "-" symbol in the table denotes cases where the performance value is not known or not reported in state-of-art. As described in Sec. 4.4.2, comparing algorithms is challenging because their goals differ. However, in the case of JSSP, the primary objective is to complete all tasks within the minimum time. Therefore, we are comparing all the algorithms based on their makespan. Among all the dispatching rules and state-of-art, PPO is outperforming for all the instances except the instances of size 100×20.

In this scenario, PPO demonstrates clear superiority over other DRL-based methods and dispatching rules across all instances, except for the 100×20 instance size. Notably, in the "ta71" instance, the makespan reported in the paper by [9] is the lowest, indicating noteworthy performance. Additionally, for instances "ta72", the FIFO rule outperforms other methods, whereas for "ta80", MWKR is performing the best, suggesting its effectiveness in those specific scenarios.
Table 4.9 shows the optimality gap of Taillard's instances for PPO and other strategies described in Sec. 4.3. For Taillard's instances as well, the average optimality gap of PPO

Table 4.9: The optimality gap on Taillard's instances and different benchmarks.

Dataset	PPO	FIFO	MWKR	SPT	Zhang et al. (2020)	Han et al. (2020)	Wu et al. (2024)
ta40	4.45	18.09	19.46	36.87	30.02	-	-
ta41	0.91	21.33	25.57	34.83	27.24	16.89	14.41
ta42	6.23	27.62	18.86	37.77	31.88	16.39	14.11
ta50	10.11	27.96	26.19	37.11	31.85	-	-
ta51	11.09	28.59	29.89	25.25	30.40	18.22	14.31
ta52	7.29	21.15	25.22	25.47	21.23	17.16	10.89
ta60	9.94	14.78	13.83	31.80	22.96	-	-
ta61	16.24	21.93	16.75	35.55	22.91	13.25	-
ta62	14.56	20.05	14.96	33.96	22.35	14.69	-
ta70	10.51	21.99	14.73	27.63	17.44	-	-
ta71	5.94	6.41	2.73	14.87	11.43	2.04	-
ta72	5.95	1.43	1.70	17.63	2.97	3.89	-
ta80	7.13	2.05	-1.60	15.39	5.05	-	-
Average	**8.49**	**17.95**	**16**	**28.78**	**21.36**	**12.82**	**13.34**

is lower than other approaches, signifying that the results of PPO are close to the optimal solution. The results highlight PPO's superior performance, with an average optimality gap of 8.49, significantly outperforming traditional scheduling algorithms FIFO, MWKR, and SPT, which have average optimality gaps of 17.95, 16, and 28.78, respectively. Notably, the PPO-based approach achieves a 2-3 times lower optimality gap than the traditional algorithms. In comparison to the state-of-the-art approaches, the PPO-based approach significantly outperforms [37], [9], and [34], with an average optimality gap of 8.49, compared to 21.36, 12.82, and 13.34, respectively. Remarkably, the PPO-based approach achieves a 2-3 times lower optimality gap than the state-of-the-art approaches.

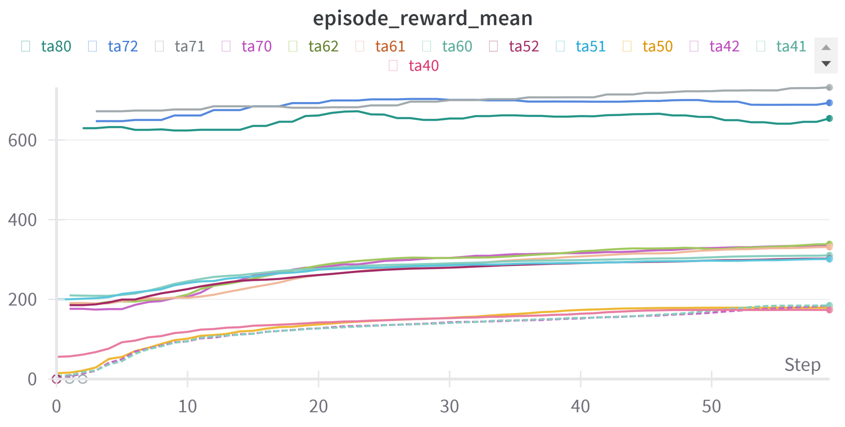

Figure 4.17: Reward of the Taillard's instances of PPO.

Reward of Taillard dataset

Figure 4.17 shows the reward of all instances during the training process for the PPO algorithm. The figure displays the reward on the y-axis and the number of iterations on the x-axis. From the graph, it can be observed that the reward value for each instance increases with the number of training iterations. This trend suggests that the agent is learning from the environment, as it is achieving higher rewards over time, indicating improvement in its performance except in the instances of size 100×20. So, those instances are trained for 200 iterations, and Fig. 4.18 depicts the graph for those instances, showing an increase in reward as the number of iterations increases.

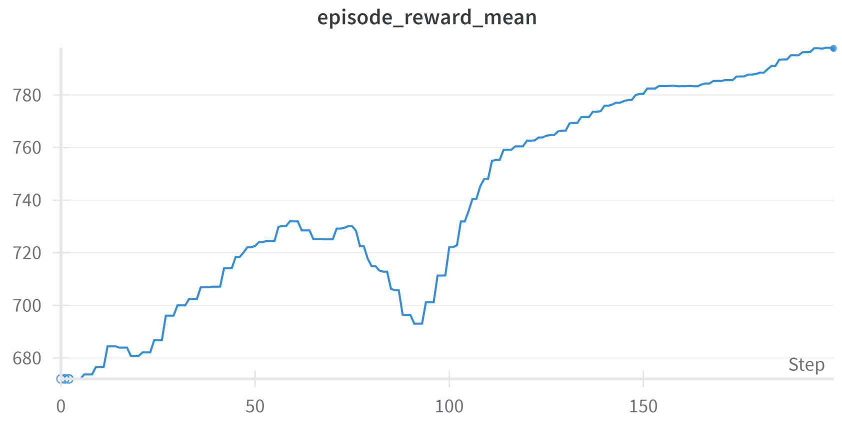

Figure 4.18: Reward of the Taillard's instances "ta71" of PPO.

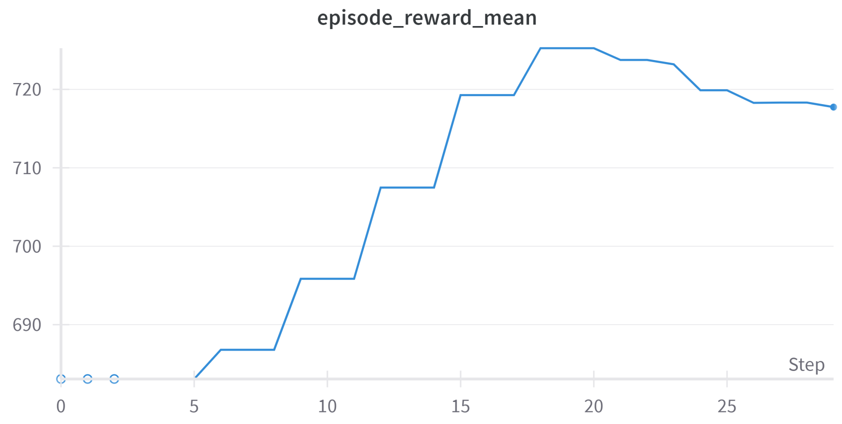

Figure 4.19: Reward of the Taillard's instances "ta71" of PG.

Moreover, upon deeply analyzing Fig. 4.18, it is observed that initially, the reward increases, but after 60 iterations, the reward starts decreasing. However, suddenly after 100 iterations, the reward begins increasing again, indicating that the agent is learning, and after 200 iterations, the reward reaches its maximum. This suggests that for a complex environment, the agent needs to be trained for more iterations. In contrast, for the agent trained using PG, the reward starts decreasing after approximately 20 iterations, indicating that the agent learns quickly.

Similarly, for the agent using A2C, the reward is maximized at 60 iterations, suggesting that the agent needs to be trained for 60 iterations to achieve optimal performance. The reason PPO requires more iterations is due to its reliance on policy optimization methods, which generally involve more gradual updates to the policy. This approach can be more stable but slower to converge, especially in complex environments where the agent needs more time to explore and learn the optimal policy.

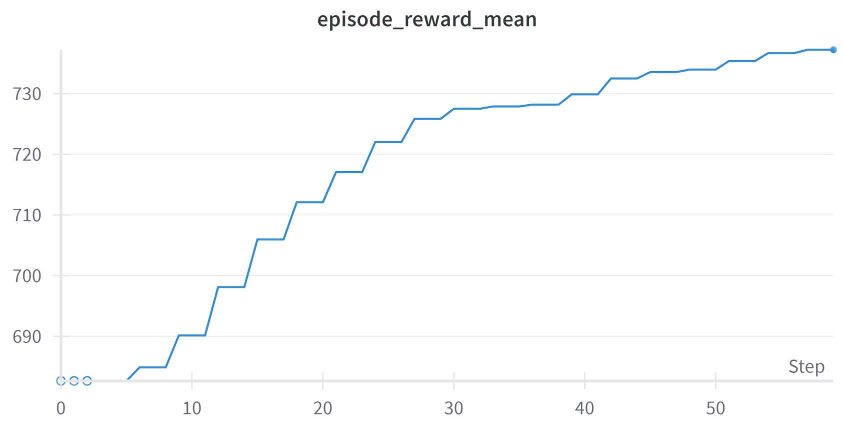

Figure 4.20: Reward of the Taillard's instances "ta71" of A2C.

Chapter 5

Discussion

The findings of this study shed light on the effectiveness of RL algorithms, particularly PPO, in addressing the JSSP. By comparing various RL algorithms with conventional dispatching rules and state-of-the-art methods across Lawrence, Dermikol, and Taillard datasets, we gained valuable insights into their performance in scheduling scenarios.

5.1 Strengths and Effectiveness of PPO

One notable observation is the consistent superiority of PPO over other RL algorithms and dispatching rules in minimizing the makespan across different instance sizes. PPO's robustness and adaptability were evident, suggesting its potential for real-world applications in complex scheduling environments. The ability of PPO to maintain lower makespan values consistently across different problem sizes and datasets highlights its efficiency and generalizability. This supports our hypothesis (H1) that PPO can be effectively implemented to solve the JSSP as a single-agent RL problem, achieving a lower makespan compared to other RL algorithms.

The optimality gap analysis further supported the effectiveness of PPO, particularly in Taillard instances, where it demonstrated closeness to the optimal solution compared to other approaches. This indicates that PPO is capable of producing solutions that are not only effective but also near-optimal, which is crucial for practical applications. While larger instance sizes showed minimal optimality gaps with MWKR, PPO still exhibited competitive performance, emphasizing its reliability across various problem complexities. This validation of H4 underscores PPO's strength in delivering solutions close to the best-known or optimal solutions.

5.2 Training Dynamics and Learning Efficiency

The training dynamics of RL agents also provided valuable insights into their learning capabilities. The increasing rewards and decreasing makespan over training iterations highlighted the agents' ability to learn from the environment and improve their scheduling policies over time. This supports our hypothesis (H2) that the implementation of PPO results in a consistent increase in rewards during the training process, indicating effective learning and adaptation.

5.3 Comparative Performance

In comparison to traditional dispatching rules and other state-of-the-art methods, PPO consistently outperformed these benchmarks in most instances. For example, while FIFO, MWKR, and SPT are well-established rules with specific strengths, they often fell short

when dealing with complex and varied scheduling scenarios. PPO's ability to dynamically adapt and optimize through continuous learning provides it with a significant advantage, validating our hypothesis (H3).
However, it's important to acknowledge that there were instances where specific dispatching rules showed competitive performance. For instance, MWKR occasionally outperformed PPO in certain large-sized instances, indicating the need for careful algorithm selection based on problem characteristics. This suggests that while PPO is generally robust, the nature of the problem instance can influence which algorithm performs best.

5.4 Weaknesses and Limitations

Despite its strengths, PPO has several limitations:

- **Instance-Specific Performance**: Certain dispatching rules like MWKR performed better on specific larger instances, suggesting that PPO may not always be the best choice for all problem types. This indicates that while PPO is robust, it is not universally optimal.
- **Extended Training Requirements**: Instances of size 100×20 required extended training iterations to achieve improved performance, indicating potential scalability issues. This is particularly relevant for large-scale industrial applications where time and computational resources are limited.
- **Computational Resources**: The computational cost associated with training PPO, especially on larger datasets, can be substantial. This limitation impacts the practical applicability of PPO in real-world scenarios where computational resources may be constrained. The need for extensive training to achieve optimal performance on larger instances underscores this challenge.

5.5 Implications for Research Questions and Hypotheses

The results directly address our research questions and hypotheses:

- **RQ1**: The effective implementation of PPO for JSSP demonstrates how DRL models can be used to tackle complex scheduling problems. The superior performance of PPO across various datasets validates its implementation approach.
- **RQ2**: The comparative analysis of different RL algorithms shows that PPO consistently outperforms others in minimizing makespan and achieving closer-to-optimal solutions. This supports hypotheses H1, H3, and H4.
- **H1**: PPO's consistent lower makespan values across different datasets validate its effectiveness in solving JSSP as a single-agent RL problem.
- **H2**: The increasing rewards during training iterations confirm that PPO effectively learns and adapts, validating its learning efficiency.
- **H3**: PPO's performance compared to traditional dispatching rules and state-of-the-art methods shows its superiority in most cases.
- **H4**: The optimality gap analysis confirms that PPO produces solutions closer to the optimal, highlighting its effectiveness.

In summary, this discussion highlights the robustness, efficiency, and learning capabilities of PPO in solving the JSSP, while also acknowledging its limitations and the need for careful algorithm selection based on specific problem characteristics.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis investigated the application of RL algorithms in solving the JSSP, with a specific focus on instances from the Lawrence, Dermikol, and Taillard datasets. The primary aim was to evaluate the performance of popular RL algorithms, namely PPO, PG, A2C, and A3C, against traditional dispatching rules and state-of-the-art methods. The findings from our comprehensive analysis highlight several key insights and contributions.

First, the superior performance of the PPO algorithm was consistently evident across various instance sizes and datasets. This highlights PPO's robustness and adaptability in effectively addressing the complexities of job shop scheduling. PPO outperformed other RL algorithms and traditional dispatching rules, demonstrating its potential as a versatile and powerful solution for dynamic and complex scheduling environments.

Moreover, the training dynamics of RL agents underscored their ability to learn from the environment and enhance their performance over time. The observed increase in reward values and decrease in makespan across all datasets confirmed the agents' capacity to adapt and optimize their scheduling policies effectively. This adaptive learning process was particularly evident in PPO, which consistently achieved higher rewards and lower makespan values compared to other RL algorithms.

The comparative assessment revealed that while certain dispatching rules, such as MWKR and FIFO, showed competitive performance in specific instances, they lacked the generalizability and adaptability of PPO. This underscores the importance of selecting the appropriate algorithm based on the specific characteristics of the scheduling problem at hand.

Furthermore, the optimality gap analysis validated the effectiveness of PPO in producing solutions closer to the optimal or best-known solutions. Remarkably, the PPO-based approach demonstrates a 6-9 times lower optimality gap compared to traditional scheduling algorithms and achieves a 2-3 times lower optimality gap than state-of-the-art approaches in all three datasets, high-lighting its superiority in addressing the JSSP. The consistent performance of PPO across diverse problem sizes and complexities highlights its reliability and efficacy in real-world applications.

In summary, the research questions posed in this thesis were addressed as follows:

- *RQ1: How can DRL models be effectively implemented to solve the JSSP as a single-agent RL problem?*
    - **Answer:** PPO was effectively implemented to solve the JSSP as a single-agent RL problem. The superior performance of PPO across various datasets validates

its implementation approach.

- *RQ2: How do different RL algorithms, including PPO, PG, A2C, and A3C, perform in solving the JSSP across various datasets and instance sizes, particularly in terms of scheduling efficiency and reducing overall makespan?*
  - **Answer:** The comparative analysis of different RL algorithms demonstrated that PPO consistently outperforms others in minimizing makespan and achieving closer-to-optimal solutions. This supports hypotheses **H1**, **H3**, and **H4**.

Additionally, the hypotheses were validated as follows:

- **H1:** PPO's consistent lower makespan values across different datasets validated its effectiveness in solving the JSSP as a single-agent RL problem.
- **H2:** The increasing rewards during training iterations confirmed that PPO effectively learns and adapts, validating its learning efficiency.
- **H3:** PPO's performance compared to traditional dispatching rules and state-of-the-art methods demonstrated its superiority in most cases.
- **H4:** The optimality gap analysis confirmed that PPO produces solutions closer to the optimal, highlighting its effectiveness.

These findings underscore the robustness, efficiency, and learning capabilities of PPO in solving the JSSP, while also acknowledging its limitations and the need for careful algorithm selection based on specific problem characteristics.

6.2 Future Work

Future research could explore combining RL with domain-specific knowledge to enhance scheduling performance. By integrating expertise from the scheduling domain with RL algorithms like PPO, we can potentially develop more effective scheduling solutions. Additionally, addressing scalability issues is crucial for handling larger datasets and complex optimization tasks in real-world settings. Exploring methods such as parallel computing or distributed frameworks could help improve scalability and efficiency. These future directions aim to advance scheduling optimization, offering more effective and scalable solutions for practical applications.

Bibliography

[1] Egon Balas. “An additive algorithm for solving linear programs with zero-one variables.” In: *Operations Research* 13.4 (1965), pp. 517–546.

[2] Adil Baykasoğlu and Fatma S Karaslan. “Solving comprehensive dynamic job shop scheduling problem by using a GRASP-based approach.” In: *International Journal of Production Research* 55.11 (2017), pp. 3308–3325.

[3] Irwan Bello et al. “Neural combinatorial optimization with reinforcement learning.” In: *arXiv preprint arXiv:1611.09940* (2016).

[4] Lukas Biewald et al. “Experiment tracking with weights and biases.” In: *Software available from wandb. com* 2.5 (2020).

[5] Jürgen Branke et al. “Automated design of production scheduling heuristics: A review.” In: *IEEE Transactions on Evolutionary Computation* 20.1 (2015), pp. 110–124.

[6] Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. “Benchmarks for shop scheduling problems.” In: *European Journal of Operational Research* 109.1 (1998), pp. 137–141.

[7] Michel Deudon et al. “Learning heuristics for the tsp by policy gradient.” In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26–29, 2018, Proceedings 15.* Springer. 2018, pp. 170–181.

[8] Christopher D Geiger, Karl G Kempf, and Reha Uzsoy. “A tabu search approach to scheduling an automated wet etch station.” In: *Journal of Manufacturing Systems* 16.2 (1997), pp. 102–116.

[9] Bao-An Han and Jian-Jun Yang. “Research on adaptive job shop scheduling problems based on dueling double DQN.” In: *Ieee Access* 8 (2020), pp. 186474–186495.

[10] Shengyi Huang and Santiago Ontañón. “A closer look at invalid action masking in policy gradient algorithms.” In: *arXiv preprint arXiv:2006.14171* (2020).

[11] Zangir Iklassov et al. “On the study of curriculum learning for inferring dispatching policies on the job shop scheduling.” In: *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI.* 2023, pp. 5350–5358.

[12] Wouter Kool, Herke Van Hoof, and Max Welling. “Attention, learn to solve routing problems!” In: *arXiv preprint arXiv:1803.08475* (2018).

[13] Stephen Lawrence. “Resouce constrained project scheduling: An experimental investigation of heuristic scheduling techniques (Supplement).” In: *Graduate School of Industrial Administration, Carnegie-Mellon University* (1984).

[14] Jaejin Lee et al. “Attention-based Reinforcement Learning for Combinatorial Optimization: Application to Job Shop Scheduling Problem.” In: *arXiv preprint arXiv:2401.16580* (2024).

[15] Matthias Lehmann. “The Definitive Guide to Policy Gradients in Deep Reinforcement Learning: Theory, Algorithms and Implementations.” In: *arXiv preprint arXiv:2401.13662* (2024).

[16] Eric Liang et al. “Ray rllib: A composable and scalable reinforcement learning library.” In: *arXiv preprint arXiv:1712.09381* 85 (2017).

[17] Chun-Cheng Lin et al. “Smart manufacturing scheduling with edge computing using multiclass deep Q network.” In: *IEEE Transactions on Industrial Informatics* 15.7 (2019), pp. 4276–4284.

[18] Chien-Liang Liu, Chuan-Chin Chang, and Chun-Jan Tseng. "Actor-critic deep reinforcement learning for solving job shop scheduling problems." In: *Ieee Access* 8 (2020), pp. 71752–71762.

[19] Alan S Manne. "On the job-shop scheduling problem." In: *Operations research* 8.2 (1960), pp. 219–223.

[20] Hongzi Mao et al. "Learning scheduling algorithms for data processing clusters." In: *Proceedings of the ACM special interest group on data communication.* 2019, pp. 270–288.

[21] Dirk C Mattfeld and Christian Bierwirth. "An efficient genetic algorithm for job shop scheduling with tardiness objectives." In: *European journal of operational research* 155.3 (2004), pp. 616–630.

[22] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning." In: *International conference on machine learning.* PMLR. 2016, pp. 1928–1937.

[23] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning." In: *arXiv preprint arXiv:1312.5602* (2013).

[24] Eiji Morinaga et al. "An improved method of job shop scheduling using machine learning and mathematical optimization." In: *Procedia Computer Science* 217 (2023), pp. 1479–1486.

[25] Su Nguyen, Yi Mei, and Mengjie Zhang. "Genetic programming for production scheduling: a survey with a unified framework." In: *Complex & Intelligent Systems* 3 (2017), pp. 41–66.

[26] Junyoung Park, Sanjar Bakhtiyar, and Jinkyoo Park. "ScheduleNet: Learn to solve multi-agent scheduling problems with reinforcement learning." In: *arXiv preprint arXiv:2106.03051* (2021).

[27] Laurent Perron and Vincent Furnon. "OR-Tools; 2019." In: *URL https://developers. google. com/optimization* 41 ().

[28] Mohit Sewak. *Deep reinforcement learning.* Springer, 2019.

[29] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[30] Eric Taillard. "Benchmarks for basic scheduling problems." In: *european journal of operational research* 64.2 (1993), pp. 278–285.

[31] Pierre Tassel, Martin Gebser, and Konstantin Schekotihin. "A reinforcement learning environment for job-shop scheduling." In: *arXiv preprint arXiv:2104.03760* (2021).

[32] European Union. *Rhinoceros.* https://www.rhinoceros-project.eu/. [Online; accessed 15-May-2024]. 2022.

[33] Bernd Waschneck et al. "Optimization of global production scheduling with deep reinforcement learning." In: *Procedia Cirp* 72 (2018), pp. 1264–1269.

[34] Xinquan Wu et al. "A deep reinforcement learning model for dynamic job-shop scheduling problem with uncertain processing time." In: *Engineering Applications of Artificial Intelligence* 131 (2024), p. 107790.

[35] Amel Yahyaoui, Nader Fnaiech, and Farhat Fnaiech. "A suitable initialization procedure for speeding a neural network job-shop scheduling." In: *IEEE Transactions on industrial electronics* 58.3 (2010), pp. 1052–1060.

[36] Seong Jin Yim and Doo Yong Lee. "Scheduling cluster tools in wafer fabrication using candidate list and simulated annealing." In: *Journal of Intelligent Manufacturing* 10 (1999), pp. 531–540.

[37] Cong Zhang et al. "Learning to dispatch for job shop scheduling via deep reinforcement learning." In: *Advances in neural information processing systems* 33 (2020), pp. 1621–1632.