A Supervised Learning Approach to Robust Reinforcement Learning for Job Shop Scheduling

Christoph Schmidl[1][a], Thiago D. Simão[3][b] and Nils Jansen[1,2][c]
[1]Radboud University, Nijmegen, The Netherlands
[2]Ruhr-University Bochum, Germany
[3]Eindhoven University of Technology, The Netherlands

Keywords: Reinforcement Learning, Job-Shop, Scheduling, Operations Research, Permutation, Robustness.

Abstract: The job shop scheduling problem (JSSP) is an NP-hard combinatorial optimization problem with the objective of minimizing the makespan while adhering to domain-specific constraints. Recent developments cast JSSP as a reinforcement learning (RL) problem, diverging from classical methods like heuristics or constraint programming. However, RL policies, serving as schedulers, often lack permutation invariance for job orderings in JSSP, limiting their generalization capabilities. In this paper, we improve the generalization of RL in the JSSP using a three-step approach that combines RL and supervised learning. Furthermore, we investigate permutation invariance and generalization to unseen JSSP instances. Initially, RL policies are trained on Taillard instances for 1800 seconds using Proximal Policy Optimization (PPO). These policies generate data sets of state-action pairs, augmented with varying permutation percentages to transpose job orders. The final step uses the generated data sets for retraining in a supervised learning setup, focusing on permutation invariance and dropout layers to improve robustness. Our approach (1) improves robustness regarding unseen instances by reducing the mean makespan and standard deviation after outlier removal by -0.43% and -15.31%, respectively, and (2) demonstrates the effect of job order permutations in supervised learning regarding the mean makespan and standard deviation.

1 INTRODUCTION

Scheduling is a task that permeates many areas of our lives, such as healthcare (Pham and Klinkert, 2008), semiconductor manufacturing (El-Khouly et al., 2009), and education in terms of timetable scheduling (Kadam and Yadav, 2016). The allocation of tasks to a restricted set of resources over time makes scheduling a combinatorial optimization problem (COP) that is NP-hard (Coffman, 1976). Traditionally, there are two common scheduling strategies. Heuristics, such as Priority Dispatching Rules (PDRs), can generate a schedule quickly but may suffer from low quality (Syarif et al., 2021; Zahmani et al., 2015). In contrast, exact solvers, such as constrained programming, can find high-quality solutions but take more time and may be unable to solve hard problem instances (Van Hentenryck et al., 1999). A particularly computationally challenging COP is the

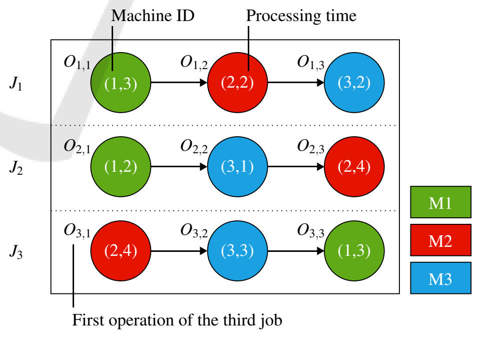

Figure 1: A JSSP instance.

job shop scheduling problem (JSSP) (Applegate and Cook, 1991), which involves determining the optimal sequence of jobs on a set of machines.

Example 1.1. *Figure 1 shows a minimal representation of a JSSP with three jobs distributed over three machines. Every row i represents a job J_i that consists of tuples (μ = machine ID, p = processing time). $O_{i,1}$, $O_{i,2}$, and $O_{i,3}$ are the three operations for each job that must be completed in order.*

An optimal solution to the JSSP (as defined

[a] https://orcid.org/0000-0003-0188-1279
[b] https://orcid.org/0000-0002-3568-9464
[c] https://orcid.org/0000-0003-1318-8973

Schmidl, C., Simão, T. and Jansen, N.
A Supervised Learning Approach to Robust Reinforcement Learning for Job Shop Scheduling.
DOI: 10.5220/0012473600003636
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 16th International Conference on Agents and Artificial Intelligence (ICAART 2024) - Volume 3*, pages 1324-1335
ISBN: 978-989-758-680-4; ISSN: 2184-433X
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

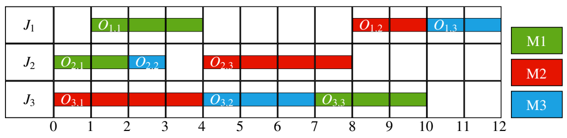

Figure 2: A Gantt chart representing a feasible schedule for a 3x3 JSSP as given in Example 1.1. Every row represents a job that is processed by a sequence of machines. The colors indicate the executing machine for the given operation.

in Section 3.1) consists of a schedule that assigns a start time to every operation of every job and adheres to several domain-specific constraints while minimizing the *makespan*, that is, the length of the schedule from the start of the first job until the end of the last job. These constraints ensure a machine handles at most one job at a time (no-overlap constraint), does not pause and resume jobs (no-preemption constraint), and maintains the order of operations of every job (precedence constraint). A feasible solution to Example 1.1 is the schedule represented as a Gantt chart with a makespan of 12 in Figure 2.

JSSP as a Reinforcement Learning Problem. Recently, Reinforcement Learning (RL) has emerged as a promising approach in the Job Shop Scheduling Problem (JSSP) domain. Its ability to learn and adapt schedules through training, leverage transfer learning across different problem instances, and potentially surpass heuristic-based schedules makes it a compelling choice. RL also minimizes the need for extensive human input in developing effective scheduling heuristics. Typically, RL agents, which can be deep learning-based, generate schedules for JSSP based on a learned policy. These algorithms are often trained and tested on the same JSSP instance. Enhancing the generalization capabilities of deep RL agents in this context is a key area of interest.

Therefore, we study the problem of generalizing RL agents to varying instances of the JSSP. Specifically, in cyber-physical systems, job arrivals, although unpredictable, may often be permutations of jobs the RL agent has previously encountered. However, even minor alterations in job order can adversely affect the makespan, indicating that the learned policy is not inherently permutation-invariant. Handling this issue is critical to making the RL agent more reliable and essential for robust scheduling in these systems.

Problem Statement. We focus on the following concrete research questions. (1) Can we imitate the behavior of an RL policy using supervised learning by using the policy as a labeling oracle? (2) Can we improve the overall robustness of an imitated and learned policy by using (2a) regularization techniques out of the supervised learning domain, such as dropout layers, to generalize better and using (2b) permuted data sets in the supervised learning phase to improve permutation invariance.

Our Approach. First, we provide a series of benchmarks to evaluate the generalization capabilities of RL agents in the JSSP setting. In a similar setup to (Tassel et al., 2021), we consider various instances of the JSSP by using *permutations of jobs*. Our experiments show that the trained RL agent is highly sensitive to these permutations. Consequently, the goal is to render the RL agent more robust against job permutations, referred to as *permutation invariant*.

Then, we propose a method that combines RL with a regularization technique called dropout (Srivastava et al., 2014). Dropout is used in supervised learning to prevent overfitting in deep neural networks. It works by randomly setting a percentage of neurons to zero during each training iteration, controlled by the dropout rate (Drop_p), which ranges from 0 (no dropout) to 1 (dropout applied to all neurons). We use the trained RL agent as a labeling oracle to generate datasets of state-action pairs with varying permutation percentages. We then train new policies using these permuted datasets in a supervised learning setup. This approach is more straightforward than training on the RL task since the target is fixed throughout the supervised learning phase.

Our contributions can be summarized as follows:

- We replicate the results of (Tassel et al., 2021) by modeling the JSSP as a single-agent RL problem, where the agent is a dispatcher that needs to choose a job at each step.
- We design an approach to collect the *experiences* of an RL agent interacting with its scheduling environment to create permuted datasets of experiences. The permutations are based on different permutation strategies.
- We perform an ablation study of different permutation strategies applied to state-action pairs from a job shop scheduling environment. We investigate its utility as a data augmentation step to learn a more robust scheduler that generalizes better to unseen scheduling instances
- We train different supervised learning models on permuted datasets to investigate how far dropout layers influence the generalization of a scheduler.

2 RELATED WORK

The most commonly used approaches for the JSSP are approximation methods such as simulated annealing, Tabu search, and the shifting bottleneck heuristic that decomposes the problem into several single-machine sub-problems that are then solved one after another (Pinedo and Singer, 1999; Pezzella and Merelli, 2000). However, meta- and hybrid heuristics such as genetic algorithms (Fan et al., 2021) and particle swarm optimization (Yu et al., 2020) yield good makespans faster. On the other hand, exact algorithms are designed to find the optimal solution to the JSSP, as opposed to heuristic or metaheuristic algorithms, but fail to find solutions if the problem instance is too hard. Branch and Bound (B&B) efficiently prunes the search space but can be computationally expensive (Brucker et al., 1994). Integer Linear Programming (ILP) benefits from solver advancements but may struggle with scalability (Bülbül and Kaminsky, 2013). Dynamic Programming (DP) leverages overlapping subproblems and memoization but suffers from the "curse of dimensionality" (Wang et al., 1997). Constraint Programming (CP) offers a flexible and expressive approach, yet its efficiency relies on the constraint model and solver (Da Col and Teppan, 2019). Lastly, Enumeration Methods, such as Depth-First Search (Beck and Perron, 2000) and Breadth-First Search (Sabuncuoglu and Bayiz, 1999), guarantee optimality but face exponential time complexity for large instances.

Early applications of RL to the JSSP date back to 1995, when it was used to develop domain-specific heuristics, outperforming existing algorithms in synthetic and NASA payload processing tasks (Zhang and Dietterich, 1995). Various studies have demonstrated the feasibility of deep reinforcement learning (DRL) for JSSP. For example, distributed policy search reinforcement learning (Gabel and Riedmiller, 2012) reduced computation time but did not outperform traditional solvers. Actor-critic DRL models achieved competitive performance in static benchmarks and balanced makespan and execution time in dynamic environments (Liu et al., 2020). However, performance declined with increasing instance size. An adaptive JSSP approach based on Dueling Double Deep Q-Network with prioritized replay was proposed (Han and Yang, 2020), showcasing adaptability, robustness, and comparable performance in dynamic environments. End-to-end DRL agents using Graph Neural Networks (GNNs) have been developed for automatically learning Priority Dispatching Rules (PDRs) (Zhang et al., 2020), with strong performance against existing PDRs but limited generalization to optimal results. GNNs offer permutation invariance by design based on their inherent permutation invariant aggregation functions to create graph embeddings. However, GNNs can be difficult to train in conjunction with reinforcement learning. Permutation invariance can also be achieved by utilizing the transformer architecture (Lee et al., 2019), which is effective in sequence modeling but has high computational demands, especially in their attention mechanisms. Extending transformers to the domain of job shop scheduling in conjunction with reinforcement learning is also a feasible but costly approach to achieve permutation invariance (Chen et al., 2022). A study by (Tassel et al., 2021) proposed a DRL algorithm utilizing a compact state space representation of the environment and a dense reward function. The action space was designed to accommodate n jobs, along with an additional No-Op (No Operation) job providing near-optimum solutions but falling short in generalization. This study serves as the basis for our approach which is a straightforward and effective way to achieve permutation invariance in various RL algorithms, avoiding the complex design and high computational demands of GNNs and transformers.

3 REINFORCEMENT LEARNING FOR JOB SHOP SCHEDULING

Scheduling problems, characterized by a set of jobs, resources, constraints, and objectives, are typically denoted by the triplet $\alpha|\beta|\gamma$ (Pinedo, 2018). Here, α indicates the machine environment (e.g., J_m for a job shop with m machines), β details processing characteristics and constraints, and γ specifies the optimization objective, frequently makespan.

3.1 The Job Shop Scheduling Problem

Job Shop Scheduling Problem. The classical job shop scheduling problem (JSSP) comprises a finite set of *jobs* $\mathcal{J}$, where the *number of jobs* is denoted as $|\mathcal{J}| = n$; and a finite set of *machines* $\mathcal{M}$, where the *number of machines* is denoted as $|\mathcal{M}| = m$. The size of a JSSP instance is specified as $n \times m$.

Every job $J_i \in \mathcal{J}$ must be processed by all m machines in $\mathcal{M}$ in the *order* given by the operation indices of job J_i, which is denoted as $O_{i,1} \rightarrow O_{i,2} \rightarrow \cdots \rightarrow O_{i,m}$. However, the execution order for every job J_i may differ. Each element $O_{i,j} (1 \leq i \leq n, 1 \leq j \leq m)$ is an *operation* of job J_i with a predefined, deterministic *processing time* $p_{i,j} \in \mathbb{N}$ executed by *machine* $\mu_{i,j} \in \mathcal{M}$. The *relation*

$\rightarrow$ denotes the *precedence* between operations, which indicates that operation $O_{i,j+1}$ can only start its execution once its predecessor operation $O_{i,j}$ finishes. Therefore, every job $J_i \in \mathcal{J}$ has a predetermined *sequence of machines* it has to visit, while this sequence is not necessarily the same for all jobs.

Moreover, the classical JSSP comprises three constraints. The *precedence constraint* states that for each job $J_i \in \mathcal{J}$, operation $O_{i,j}$ must be completed before starting operation $O_{i,j+1}$. This constraint can be formally expressed as $C_{i,j} \leq S_{i,j+1}$, where $C_{i,j} = S_{i,j} + p_{i,j}$ is the *completion time* of operation $O_{i,j}$ and $S_{i,j}$ is the *starting time* of operation $O_{i,j}$, i.e., the starting time of operation $O_{i,j+1}$ should not be scheduled before the completion time of operation $O_{i,j}$. The *no-overlap constraint* states that a machine cannot execute two operations simultaneously, but each machine can only execute one operation simultaneously. The *no-preemption constraint* states that an operation, once started, must run to completion, i.e., it is forbidden to interrupt an operation once it is started.

A *feasible solution* to the JSSP is a *schedule* that maps a *starting time* $S_{i,j}$ to every operation $O_{i,j}$ while satisfying all constraints. While there are different objectives available for the JSSP, such as *total tardiness* or *average flow time*, the most common one is minimizing the overall *makespan*. The *makespan* is defined as $C_{max} = \max_{i,j}\{C_{i,j} = S_{i,j} + p_{i,j}\}$, i.e., the *makespan* is the completion time of the last operation $O_{i,j}$ leaving the system. By minimizing the makespan, the length of the overall schedule is minimized as well. Therefore, an *optimal solution* to the JSSP is a schedule that comprises the determination of a start time $S_{i,j}$ for every operation $O_{i,j}$ of every job $J_i \in \mathcal{J}$ that adheres to the constraints and minimizes the makespan.

Disjunctive Graph. The disjunctive graph is the most common representation of a JSSP that incorporates all of its constraints. Scheduling algorithms such as the shifting bottleneck heuristic (Pinedo and Singer, 1999) or meta-heuristics such as Tabu search (Pezzella and Merelli, 2000) use this representation to solve JSSP instances. Recently, Reinforcement Learning (RL) approaches utilized this graph representation of the JSSP as part of their environments (Zhang et al., 2020; Huang et al., 2023).

A disjunctive graph can be defined as a mixed graph $g = (V_g, C_g, D_g)$, where V_g denotes the vertex set of the graph; C_g represents the precedence constraints between operations of the same job as directed edges, i.e., conjunctions; and D_g represents a set of undirected edges, referred to as disjunctions. Each disjunction connects a pair of operations that require the same machine for processing, thereby representing the no-overlap constraint. The vertex set $V_g = \{O_{i,j} | \forall i,j\} \cup \{S,T\}$ represents the set of all operations with additional nodes S and dummy notes T. These dummy nodes have no processing time and denote the start and termination of a schedule.

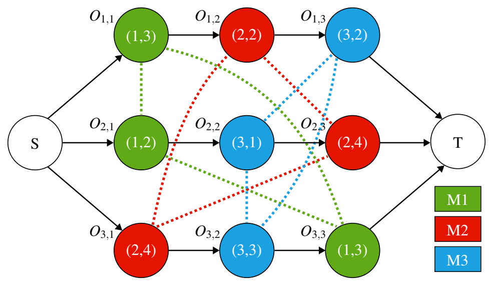

Figure 3: The disjunctive graph representation of a JSSP instance with three jobs and three machines (3x3).

Figure 3 depicts a 3x3 job shop scheduling problem represented by a disjunctive graph. *Operations* are represented as circles with colors indicating the machine executing the operation. The integer value inside the circle represents the *processing time* of that operation. The leftmost circle represents the dummy start node S, while the rightmost circle is the dummy terminal node T, signifying the end of the schedule. Both dummy nodes have no *processing time*. Precedence constraints are illustrated as directed black edges, i.e., conjunctive edges. Undirected colored dashed edges, i.e., disjunctive edges, represent cliques of operations requiring the same machine.

A solution to a JSSP instance is found when each disjunctive edge is directed so that the graph is a Directed Acyclic Graph (DAG). Determining the direction of disjunctions is equivalent to finding a sequence of operations executed by individual machines.

4 BACKGROUND

4.1 Deep Reinforcement Learning

Reinforcement learning (RL) is a computational approach for training intelligent agents to make decisions by interacting with their environment (Figure 4). In RL, an agent learns an optimal policy π that maps states $s \in \mathcal{S}$ to actions $a \in \mathcal{A}$ to maximize the cumulative reward. Deep RL combines RL with deep learning (Plaat, 2022), and was successfully applied

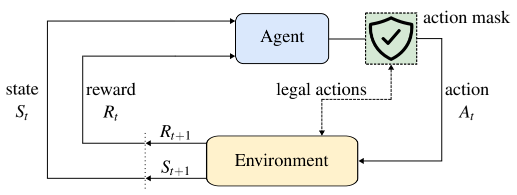

Figure 4: RL consists of an agent who interacts with its environment through actions and retrieves rewards after every step through that environment. Action masks prevent agents from choosing actions that could lead to invalid states.

to high-dimensional problems such as Atari games (Mnih et al., 2015) and the game of Go (Silver et al., 2016). Proximal Policy Optimization (PPO) is a policy gradient algorithm that optimizes the policy using stochastic gradient ascent. PPO has achieved state-of-the-art performance in various tasks, such as robotic manipulation (Rajeswaran et al., 2017) and continuous control problems (Schulman et al., 2017). Moreover, it was successfully applied to the dynamic job shop problem (Luo et al., 2021). In PPO, the objective function is defined as the ratio of probabilities under the current policy $\pi_\theta(a|s)$ and the old policy $\pi_{\theta_{old}}(a|s)$, multiplied by the advantage function $A^{\pi_{\theta_{old}}}(s,a)$:

$$L(\theta) = \mathbb{E}_{s_t,a_t \sim \pi_{\theta_{old}}}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} A^{\pi_{\theta_{old}}}(s_t,a_t)\right] \quad (1)$$

To prevent excessively large policy updates, PPO introduces a trust region constraint by employing a clipped objective function:

$$L_{\text{PPO}}(\theta) = \mathbb{E}_{s_t,a_t \sim \pi_{\theta_{old}}}\Bigg[\min\Big(r_t(\theta)A^{\pi_{\theta_{old}}}(s_t,a_t), \\ \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A^{\pi_{\theta_{old}}}(s_t,a_t)\Big)\Bigg] \quad (2)$$

where $r_t(\theta) = {}^{\pi_\theta(a_t|s_t)}/{}_{\pi_{\theta_{old}}(a_t|s_t)}$ and ε is a hyperparameter controlling the degree of constraint.

PPO optimizes the policy by iteratively updating the policy parameters θ using stochastic gradient ascent on the clipped objective function. This approach balances exploration and exploitation while maintaining stable policy updates.

4.2 Data Augmentation

Data augmentation, commonly used in supervised learning for image classification (Krizhevsky et al., 2017; Simonyan and Zisserman, 2014) and sequence prediction (Vinyals et al., 2015), improves model generalization by expanding the training dataset through

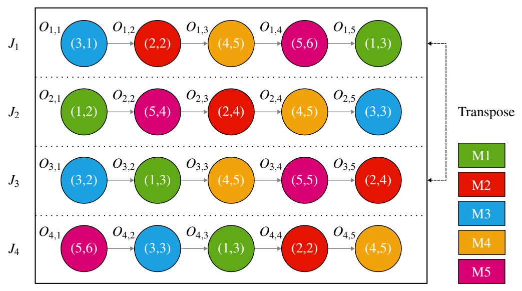

Figure 5: Transposition of jobs for a scheduling instance where Job 1 and Job 3 are swapped for each other. The ordering sequence of operations per job remains the same.).

techniques such as random crops, flips, rotations, and color changes. This concept extends to reinforcement learning (RL), as seen in applications like Deep Q-Network (DQN) for Atari games (Mnih et al., 2015) and AlphaGo (Silver et al., 2016), where state or move permutations help in generalization. In the context of RL for job shop scheduling problems (JSSP), data augmentation through job sequence permutations, e.g., changing a sequence from $[1,2,3,4]$ to $[3,2,1,4]$ as illustrated in Figure 5 introduces diverse training scenarios. This method allows RL agents, such as those using the Proximal Policy Optimization (PPO) algorithm, to learn from a broader range of examples, improving its ability to generalize to new, unseen situations.

In job shop scheduling, sequencing jobs on various machines generates factorial permutations ($n!$ for n jobs). Although these permutations alter job order, the invariant nature of operation sequences and machine assignments for each job means the optimal makespan remains constant. This characteristic makes permutations particularly suitable for JSSP, as they do not affect the optimal makespan but help generalize the RL agent. To evaluate the generalization or robustness of trained RL agents in JSSP, permutations are utilized to generate test instances for model assessment or to augment the training set, providing more diverse scenarios to the learning process of the RL agent.

5 METHOD

This section provides information about our approach and its implementation. The following subsections describe each step in our pipeline but exclude descriptions of the baseline techniques because they are not relevant to the pipeline.

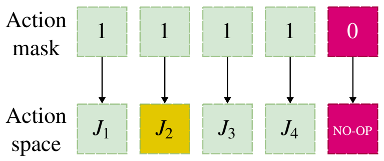

Figure 6: A boolean action mask indicating valid and invalid actions at a certain state. The orange square represents a valid action that the RL agent chose.

5.1 Sequential Decision Making Under Uncertainty

A *Markov Decision Process (MDP)* (Puterman, 1994) is defined by a tuple $M = \langle S, A, P, R, \gamma \rangle$, where S is the finite set of states; A is the finite set of actions; $P\colon S \times A \times S \mapsto [0,1]$ is the transition function, such that $P(s'|s,a)$ denotes the probability of transitioning to $s' \in S$ when $a \in A$ is executed in $s \in S$; $R\colon S \times A \times S \mapsto \mathbb{R}$ is the reward function, such that $R(s,a,s')$ is the reward for executing action $a \in A$ in state $s \in S$ and transitioning to state $s' \in S$; and $\gamma \in [0,1]$ is the discount factor, which indicates the relative importance of immediate versus future rewards (Sutton and Barto, 2018).

Following the approach of (Tassel et al., 2021), we model the JSSP as an MDP, where the *state* is a matrix of dimensions $(\mathcal{J} \times 7)$, with each row representing a job characterized by seven features. Rather than directly using the disjunctive graph for state representation, it is expressed through these features, while the graph is used internally by the environment to transition to the next state upon selecting a valid action. An *action* $a_t \in A$ at any time step t is a valid operation, and with each job having at most one ready operation at t, the action space size is maximized at $|\mathcal{J}|+1$, varying with the instance. As jobs are completed, the size of the actionable set $|A|$ reduces. *Action masking* is implemented to exclude illegal or infeasible actions from consideration.

The definition of an MDP can be extended by a masking function $M_a : S \times A \to \{0,1\}$. Then, for a given state $s \in S$, the set of legal actions $A_{\text{legal}}(s) \subseteq A$ is obtained by applying the masking function $M_a(s,a)$: $A_{\text{legal}}(s) = \{a \in A \mid M(s,a) = 1\}$. The agent can then only consider and select actions from the set $A_{\text{legal}}(s)$ while in state s, avoiding illegal actions and maximizing the expected cumulative discounted reward. Figure 6 depicts an action mask where four jobs represent valid action choices, and the no-op operation represents an invalid action choice. If no job should be available at time step t, then the default action is no-op.

Processing times are deterministic in our case, and there is no stochasticity in the *state transition* that could have been caused by random machine breakdowns or maintenance. If there is no valid job to choose from at time step t, then the no-op action is selected, and the transition continues to time step $t+1$. Notice that the time step t is not synchronized with the actual schedule, i.e., the current time step of the MDP is not always the same as the processing time of the schedule. We used a dense *reward function* based on the scheduled area (Tassel et al., 2021). After each action, we compute the difference between the duration of the allocated operations and the introduced holes, i.e., the idle time of a machine:

$$R(s,a) = p_{aj} - \sum_{\mu \in M} empty_{\mu}(s,s') \tag{3}$$

where s and s' represent the current and next state, respectively; a the j^{th} operation of J_a with processing time p_{aj} scheduled (i.e., the action); s' the next state resulting from applying action a to state s; $empty_{\mu}(s,s')$ a function returning the amount of time a machine μ is IDLE while transitioning from state s to state s'. There is a negative correlation between the reward function and the makespan while training the PPO agent, i.e., the higher the reward, the lower the makespan. A *stochastic policy* $\pi(a_t|s_t)$ outputs a distribution over actions in A for state s_t. When the learned policy is applied as a scheduler, the stochastic policy becomes a deterministic policy where the action with the highest probability is chosen.

5.2 Robust Retraining Approach

This section describes our approach to evaluating the effect of permutations and the usage of a dropout layer to improve the robustness of a previously learned scheduler.

Phase 1 - Reinforcement Learning. The first phase consists of the initial training of RL policies using the PPO algorithm solely on their dedicated Taillard instances for 1800 seconds. RL algorithms besides PPO, such as DNQ or SARSA, could be used in this setup, provided they support discrete action spaces and action masking. The trained RL policies are named π_{Ta41} to π_{Ta50}. Out of time constraints, we only performed hyperparameter tuning based on instance Ta41 following the approach of (Tassel et al., 2021) and applied these hyperparameters to all RL policies. Our RL setup uses a flattened representation of the state as a 30×7 one-dimensional vector, where 30 represents the number of jobs and 7 features.

Phase 2 - Experience Collection. In Phase 2, RL policies trained in the first phase generate state-action pair datasets for supervised learning. We modified the

OpenAI gym JSSP environment from (Tassel et al., 2021) to include a permutation mode, enabling the creation of a permutation matrix that alters job orders according to specified permutation percentages (0%, 20%, 40%, 60%, 80%, 100%). In this enhanced environment, the RL policies label the current state with the highest probability action. While initially presented with the original Taillard instance job orders, the environment applies permutation to the state-action pairs, thereby augmenting the dataset. Each of the ten RL policies produces six uniquely permuted datasets, resulting in 60 data sets for the subsequent supervised learning phase.

Phase 3 - Supervised Learning Retraining. In the final phase, our supervised learning setup, largely mirroring the RL configuration, utilizes stationary targets derived from the collected datasets. Here, a new policy π' is trained to mimic the original trained agent across various permuted datasets. The integration of dropout layers (Srivastava et al., 2014) enhances model generalization, a technique less effective in the RL framework. Utilizing the 60 permuted datasets, we trained 120 models, 60 with active dropout layers and 60 without, to assess the impact of dropout on model performance. The evaluation focuses on robustness, using the mean *makespan* and *standard deviation* as key metrics, as outlined in Section 3.1

Implementation. Our implementation[1] uses the public JSS environment[2] (Tassel et al., 2021) that is based on the OpenAI Gym toolkit[3]. We extended the environment with permutation functionality where the user can choose between different magnitudes of permutations, i.e., the number of pair-swaps for a scheduling instance. Furthermore, our pipeline implementation uses stable-baselines3 (Raffin et al., 2021) and PyTorch (Paszke et al., 2019). We performed Bayesian hyper-parameter optimization using Wandb (Biewald, 2020) on the Taillard Ta41 instance and used the best-performing configuration for all other instances of the class of 30 jobs and 20 machines. All experiments were run on a machine with an Intel Core i9-10980XE CPU (3GHz), 256GB of RAM, and a single Nvidia Geforce RTX 3090.

Models and Configuration. We used the PPO algorithm and two neural networks for our RL setup. The policy network represents the actor, while the value function network represents the critic. Both networks do not share any layers. Both networks have three hidden layers with 256, 256, and 128 neurons, respectively. Rectified linear units (ReLU) (Agarap, 2018) is the activation function. The discount factor γ is 0.99, and the Adam optimizer's learning rate is 1.0802×10^{-3}. The epochs of updating the network were set to 7, and the number of rollouts was set to 731. The clipping parameter is 0.1816, the entropy coefficient is 3.3529×10^{-3}, and the value function coefficient is 0.5. We used a batch size of 64. The neural network structure used for the supervised learning setup differs from the RL setup regarding a dropout layer after the first and second hidden layers. We also used the Adam optimizer with a 1.0802×10^{-3} learning rate but used cross-entropy as the loss function. The dropout value p was set to 0.5 for an activated dropout mode and 0.0 with an inactive dropout mode. The chosen dropout value of 0.5 is a standard often found in supervised learning literature, offering a balanced approach to regularization, but it can be increased for a more aggressive dropout strategy. Our models and configurations are based on the setup described by (Tassel et al., 2021). However, additional hyperparameter tuning was necessary due to our adoption of stable-baselines3, which differs from the implementation in (Tassel et al., 2021). While we retained the original network structure, exploring variations in this structure could offer further insights into its impact on the results.

6 EXPERIMENTS

Data Set. We used the well-known job shop instances provided by Taillard (Taillard, 1993) with varying numbers of jobs and machines. These instances were studied extensively; therefore, the upper and lower-bound solutions are already known for each instance [4]. We used the same subset selection of instances as (Tassel et al., 2021) by using the class of 10 instances with 30 jobs and 20 machines (TA41 to TA50), i.e., 30×20. The problem size does not always correlate with the hardness of the instance; therefore, we also used this instance subset because it is known to be harder than bigger instances.

However, in contrast to (Tassel et al., 2021), we did not evaluate the performance of our approach to another set of instances such as Demirkol, Mehta, and Uzsoy (Demirkol et al., 1998). We are interested in the generalization capabilities over job permutations that originate from the same distribution with simi-

[1]https://github.com/ChristophSchmidl/stable-job-shop

[2]https://github.com/prosysscience/RL-Job-Shop-Scheduling

[3]https://github.com/openai/gym

[4]http://jobshop.jjvh.nl/index.php

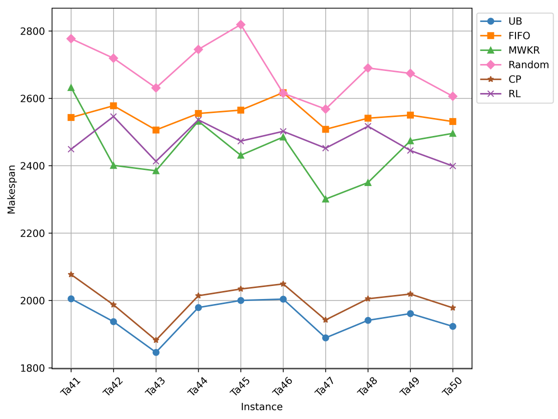

Figure 7: Makespan comparison between the best-known upper bounds (UB), constraint programming (CP), reinforcement learning (RL), and Priority Dispatching Rules such as First-In-First-Out (FIFO), Most-Work-Remaining (MWKR), and Random.

lar processing times, which do not include Demirkol, Mehta, and Uzsoy. The concern was that a significant deviation in processing times would distort the evaluation of permutation robustness.

Baselines. We chose the same computational methods for our baselines as (Tassel et al., 2021) to determine the makespans applied to the original Taillard instances from Ta41 to Ta50. All methods have a time limit of 1800 seconds in contrast to the 3600 seconds used by (Zhang and Dietterich, 1995; Tassel et al., 2021) to compute the makespans. We used the OR-tools library by Google as a *constraint programming (CP)* approach. Moreover, *First-In-First-Out (FIFO)*, *MWKR (Most-Work-Remaining)*, and a *random* heuristic were used as a small subset of commonly used *Priority Dispatching Rules* for the JSSP. Finally, we applied *Proximal Policy Optimization (PPO)* as the reinforcement learning algorithm of choice. These techniques were compared to the theoretical upper bounds (UB) of each Taillard instance that is publicly available. Figure 7 shows the mean makespans for every technique applied to the Taillard instances from Ta41 to Ta50 with a time limit of 1800 seconds and the best-known upper bounds (UB).

Figure 7 clearly shows that constraint programming outperforms all other techniques and often finds optimal solutions in the first 10 minutes that come close to the theoretical upper bounds. As one expected, the random policy performed the worst on average. Moreover, it is interesting that Priority Dispatching Rules perform pretty well compared to our trained RL policies. Especially because Priority Dispatching Rules only need 2 seconds on average

Table 1: Permutation sensitivity of learned RL agents for their dedicated problem instance.

RL policy	0%	20%	40%	60%	80%	100%
π_{Ta41}	2449	2762	2760	2774	2764	**2798**
π_{Ta42}	2546	**2844**	2806	2809	2830	2827
π_{Ta43}	2413	**2798**	2793	2766	2713	2753
π_{Ta44}	2536	2747	2763	2768	2804	**2829**
π_{Ta45}	2473	2763	2738	2756	2761	**2767**
π_{Ta46}	2502	2812	2880	2865	2848	**2888**
π_{Ta47}	2450	2810	2858	2871	2891	**2895**
π_{Ta48}	2517	2783	2826	2848	2867	**2886**
π_{Ta49}	2445	2698	2678	**2725**	2719	2703
π_{Ta50}	2399	2650	2676	2685	**2732**	2707

to output a feasible solution, while our trained RL policies needed 1800 seconds to train and 2 seconds to perform the inference step. Reinforcement learning is a more general technique with many hyperparameters to tune that probably explains the mere average performance of the RL policies. At the same time, Priority Dispatching Rules are specially tailored heuristics to deal with the job shop scheduling problem.

However, our research aim is to improve robustness for RL policies and not RL policies that produce state-of-the-art makespans. These baseline computations should merely serve as first insights into the general performance of different techniques under the time limit constraint of 1800 seconds, a realistic constraint often found in the industry.

Permutation Sensitivity of RL Policies. This section demonstrates the sensitivity of trained RL policies to varying job order permutations. Each RL agent was trained on its dedicated problem instance and evaluated on different permutation percentages, e.g., a 20% permutation indicates that 6 out of 30 jobs have swapped positions. The worst performing makespan for each instance on average is highlighted in bold in Table 1. Makespans, excluding the 0% permutation configuration, were averaged over 100 runs to account for stochastic effects from random job order permutations. The table shows a direct correlation between increasing permutation percentages and makespan growth, suggesting RL agent performance deterioration with more significant disorder in problem instances. The biggest challenge for RL agents is at 100% permutation, marked by the highest average makespan.

Generalization of RL Policies. This section evaluates the generalization of learned RL policies to untrained instances within the Taillard instance set (Ta41 to Ta50). We tested each policy's adaptability

Table 2: Generalization capabilities of trained RL agents applied to instances Ta41 to Ta50.

RL	Evaluation Instance									
policy	Ta41	Ta42	Ta43	Ta44	Ta45	Ta46	Ta47	Ta48	Ta49	Ta50
π_{Ta41}	**2449**	2638	2818	2723	2884	3018	2975	2666	2752	2567
π_{Ta42}	2799	**2546**	2557	3162	2730	2677	2452	2706	2689	2739
π_{Ta43}	2933	2822	**2413**	2811	2857	3025	2808	3051	2776	2591
π_{Ta44}	2694	2815	3102	**2536**	2976	2930	3075	2597	2620	2652
π_{Ta45}	2679	2913	2537	2962	**2473**	2848	2808	2624	2774	2548
π_{Ta46}	2572	2670	2451	2927	2828	**2502**	2656	2820	2594	2703
π_{Ta47}	3002	2638	3007	2857	2747	2781	**2450**	3165	2696	2702
π_{Ta48}	2957	2803	3031	2959	3038	2825	2703	**2517**	2711	2768
π_{Ta49}	2777	2802	2418	2714	2823	3036	2842	2744	**2445**	2609
π_{Ta50}	2843	2950	2428	2726	2846	2897	2762	2953	2630	**2399**

and effectiveness on new problems, where each policy, like π_{Ta41}, was exclusively trained on its corresponding instance, such as Ta41.

Table 2 presents the performance outcomes. The rows correspond to specific RL policies, and the columns represent the instances on which these policies were assessed, including those not seen during training. The table reveals that policies perform best on the instances they were trained on, as indicated by the bold makespan values along the diagonal, highlighting each policy's optimal performance on its dedicated instance.

However, Figure 8 visualizes Table 2 and shows some correlation between instances and policies regarding makespan values. Rl policy π_{Ta49} performs exceptionally well on instance Ta43 with a makespan of 2418, while its dedicated RL policy π_{Ta43} gives a makespan of 2413. On the other hand, we also see big performance drops with high makespan when an RL policy is applied to an instance it has not seen during training, e.g., RL policy π_{Ta47} applied to instance Ta48 produces a schedule with a makespan of 3165. The observation that makespan generally worsens when applying RL policies to new instances indicates that, despite some generalization capability, these policies perform best on instances they were trained on. This shows the challenge in training RL policies to effectively generalize to unseen instances.

Permutation and Dropout Effect. We examined the impact of varying permutation percentages on the datasets for the supervised learning retraining phase and assessed the influence of dropout layer activation. For reference, our baseline *mean* and *standard deviation* of makespan values are 2755.21 and 183.73, respectively, as shown in Table 2. These baseline metrics were compared against the outcomes for different permutation percentages with active and inactive dropout layers, noting that a lower standard deviation indicates improved robustness due to reduced dispersion around the mean.

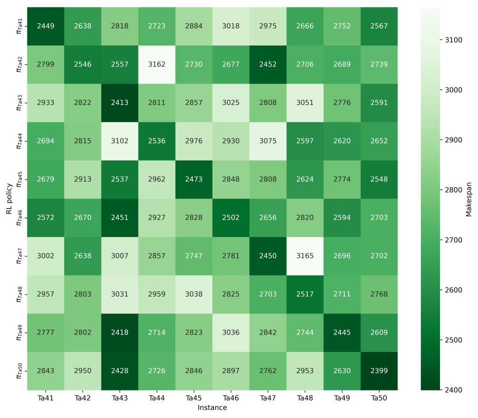

Figure 8: Generalization capabilities of RL policies applied to Taillard instances Ta41 to Ta50. A darker color represents a lower and, therefore, better makespan.

Table 3: Permutation sensitivity measured in makespan means and standard deviation values for Taillard instances Ta41 - Ta50.

Permutation	Mean (inactive)	Std (inactive)	Mean (active)	Std (active)
0%	**2745.37**	**156.68**	2751.55	172.03
20%	2801.43	235.63	2821.25	283.42
40%	2861.01	253.59	2871.92	300.79
60%	2882.25	324.40	2919.67	307.90
80%	2915.89	335.42	2920.89	273.75
100%	2876.07	181.58	2899.29	341.12

(a) Absolute values for makespan means and standard deviations.

Permutation	Mean (inactive)	Std (inactive)	Mean (active)	Std (active)
0%	**-0.36%**	**-14.73%**	-0.13%	-6.37%
20%	+1.68%	+28.24%	+2.40%	+54.25%
40%	+3.84%	+38.02%	+4.24%	+63.71%
60%	+4.61%	+76.55%	+5.97%	+67.58%
80%	+5.83%	+82.55%	+6.01%	+48.99%
100%	+4.39%	-1.18%	+5.23%	+85.66%

(b) Relative values in percentages for makespan means and standard deviations.

Table 3 shows the impact on mean makespan and standard deviation across different permutation percentages, comparing active and inactive dropout layers. While Table 3a presents absolute values, Table 3b highlights changes relative to baseline values of 2755.21 (mean) and 183.73 (standard deviation). The most notable decrease in mean makespan is a modest -0.36% with 0% permutations and an inactive dropout layer, but this configuration yields a significant -14.73% reduction in standard deviation compared to the baseline.

To better visualize the improved robustness in terms of a decreased standard deviation, Figure 9 shows a similar confusion matrix as Figure 8. The more uniform distribution of darker cells across the confusion matrix indicates better generalization capabilities and improved robustness.

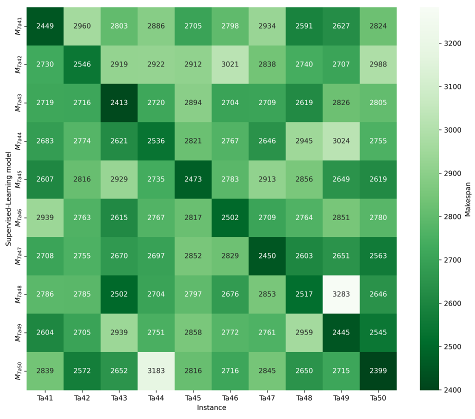

Figure 9: Generalization capabilities of supervised learning models with an inactive dropout layer and 0% permutation applied to Taillard instances Ta41 to Ta50. A darker color represents a lower and, therefore, better makespan.

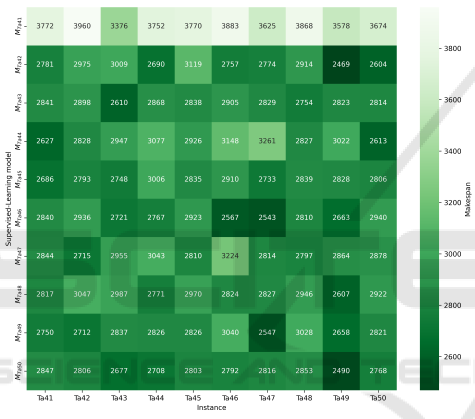

Figure 10: Generalization capabilities of supervised learning models with 60% permutation and active dropout layer applied to Taillard instances Ta42 to Ta50. The first row shows model M_{Ta41} as a clear outlier.

Permutation and Dropout Effect Without Outliers. The relative values in Table 3 show a -1.18% decrease in standard deviation with 100% permutation and inactive dropout, but an 85.66% increase when the dropout layer is active. This unexpected outcome prompted a reevaluation excluding the RL policy π_{Ta41} and instance Ta41, as the hyperparameters, optimized only for π_{Ta41}, might cause overfitting to this specific instance, skewing the results. This bias is evident in the confusion matrix (Figure 10), particularly in the first row. To mitigate this, we recalculated the RL baseline, omitting Taillard instance Ta41 and its dedicated policy.

Excluding outlier Ta41, the adjusted baseline values for mean makespan and standard deviation are 2755.9 and 185.18, respectively, comparable to the original values of 2755.21 and 183.73. As Table 4 illustrates, the greatest reduction in standard deviation without Ta41 is -27.71% achieved with 100% permutation and active dropout, but at the cost of a 1.51% increase in mean makespan. Nevertheless, retraining the supervised learning model with no permutation and dropout remains the most effective strategy, yielding the optimal decrease in mean makespan and standard deviation by -0.43% and -15.31%, respectively.

Table 4: Permutation sensitivity measured in makespan means and standard deviation values for Taillard instances Ta42 to Ta50.

Permutation	Mean (inactive)	Std (inactive)	Mean (active)	Std (active)
0%	2743.99	156.83	2753.63	174.23
20%	2746.37	170.40	2744.89	166.72
40%	2801.6	180.19	2792.23	186.90
60%	2793.9	185.97	2830.1	149.01
80%	2817.71	148.26	2846.77	154.74
100%	2841.11	152.56	2797.49	133.86

(a) Absolute values for makespan means and standard deviations.

Permutation	Mean (inactive)	Std (inactive)	Mean (active)	Std (active)
0%	**-0.43%**	**-15.31%**	-0.08%	-5.91%
20%	-0.35%	-7.98%	-0.40%	-9.96%
40%	+1.66%	-2.69%	+1.32%	+0.93%
60%	+1.38%	+0.43%	+2.69%	-19.53%
80%	+2.24%	-19.94%	+3.30%	-16.44%
100%	+3.09%	-17.61%	+1.51%	-27.71%

(b) Relative values in percentages for makespan means and standard deviations.

7 CONCLUSION AND FUTURE WORK

Our research demonstrates that after 1800 seconds of training, reinforcement learning (RL) can develop a policy that rivals naive Priority Dispatching Rules like FIFO and MWKR in scheduling efficiency. While constraint programming (CP) remains preferred for its near-optimal schedules, its applicability is limited in complex job shop scheduling scenarios. Heuristics or RL, offering approximate solutions or learning-based scheduling, are more suitable in such cases. However, RL policies, as learned schedulers, show limited robustness to new instances and permutations in trained job orders.

Addressing these limitations, we explored the impact of different permutation levels and incorporated a dropout layer during the supervised learning retraining phase. We utilized ten RL policies, each trained for 1800 seconds on specific Taillard instances, to gather state-action pairs. These pairs formed the datasets for supervised learning models and were permuted to various extents.

To measure the effect on the robustness of our approach, we calculated the mean makespan and standard deviation of all ten models over ten instances. The evaluation of all ten models shows that an inactive dropout layer and 0% permutation was the most effective configuration to reduce the mean makespan by -0.36% and the standard deviation by -14.74%. We also evaluated the effect by removing the outlier instance Ta41 and observed similar behavior with the same configuration but a reduction in mean makespan and standard deviation by -0.43% and -15.31%, respectively.

Our approach is particularly beneficial in dynamic environments, where schedules must be robust to unforeseen variations and disruptions. Applied to sectors such as manufacturing, logistics, and healthcare, it significantly enhances the robustness of schedules and, therefore, produces more consistent and predictable schedules. For instance, in manufacturing with large-scale printers and on-demand print shops, our method translates into more predictable and efficient production cycles. In logistics, particularly in airport baggage handling systems, it greatly enhances baggage processing predictability, helping to maintain flight schedules and enhance passenger satisfaction. Similarly, in healthcare management, our approach provides a more stable and efficient scheduling solution, thereby improving patient care.

In the future, we would like to investigate the effect of permutations directly inside the RL loop inside the replay buffer and how to combine it with curriculum learning. Moreover, it would be interesting to see how permutations perform when the original state representation is transformed into a latent state representation using graph neural networks.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This research has been partially funded by the NWO project 17931 (SAM-FMS) in the MasCot program and the ERC Starting Grant 101077178 (DEUCE).

REFERENCES

Agarap, A. F. (2018). Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*.

Applegate, D. and Cook, W. (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on computing*, 3(2):149–156.

Beck, J. C. and Perron, L. (2000). Discrepancy-bounded depth first search. In *Proceedings of CP-AI-OR*, pages 7–17.

Biewald, L. (2020). Experiment tracking with weights and biases, software available from wandb. com (2020). *URL https://www. wandb. com.*

Brucker, P., Jurisch, B., and Sievers, B. (1994). A branch and bound algorithm for the job-shop scheduling problem. *Discrete applied mathematics*, 49(1-3):107–127.

Bülbül, K. and Kaminsky, P. (2013). A linear programming-based method for job shop scheduling. *Journal of Scheduling*, 16:161–183.

Chen, R., Li, W., and Yang, H. (2022). A deep reinforcement learning framework based on an attention mechanism and disjunctive graph embedding for the job-shop scheduling problem. *IEEE Transactions on Industrial Informatics*, 19(2):1322–1331.

Coffman, J. E. (1976). Computer and job-shop scheduling theory, john wiley&sons. *Inc. New York*.

Da Col, G. and Teppan, E. C. (2019). Industrial size job shop scheduling tackled by present day cp solvers. In *Principles and Practice of Constraint Programming: 25th International Conference, CP 2019, Stamford, CT, USA, September 30–October 4, 2019, Proceedings 25*, pages 144–160. Springer.

Demirkol, E., Mehta, S., and Uzsoy, R. (1998). Benchmarks for shop scheduling problems. *European Journal of Operational Research*, 109(1):137–141.

El-Khouly, I. A., El-Kilany, K. S., and El-Sayed, A. E. (2009). Modelling and simulation of re-entrant flow shop scheduling: An application in semiconductor manufacturing. In *2009 International Conference on Computers & Industrial Engineering*, pages 211–216. IEEE.

Fan, H., Xiong, H., and Goh, M. (2021). Genetic programming-based hyper-heuristic approach for solving dynamic job shop scheduling problem with extended technical precedence constraints. *Computers & Operations Research*, 134:105401.

Gabel, T. and Riedmiller, M. (2012). Distributed policy search reinforcement learning for job-shop scheduling tasks. *International Journal of production research*, 50(1):41–61.

Han, B.-A. and Yang, J.-J. (2020). Research on adaptive job shop scheduling problems based on dueling double dqn. *Ieee Access*, 8:186474–186495.

Huang, J.-P., Gao, L., Li, X.-Y., and Zhang, C.-J. (2023). A novel priority dispatch rule generation method based on graph neural network and reinforcement learning for distributed job-shop scheduling. *Journal of Manufacturing Systems*, 69:119–134.

Kadam, V. and Yadav, S. (2016). Academic timetable scheduling: revisited. *International journal of Research in Science & Engineering*, pages 417–423.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90.

Lee, J., Lee, Y., Kim, J., Kosiorek, A., Choi, S., and Teh, Y. W. (2019). Set transformer: A framework for attention-based permutation-invariant neural networks. In *International conference on machine learning*, pages 3744–3753. PMLR.

Liu, C.-L., Chang, C.-C., and Tseng, C.-J. (2020). Actor-critic deep reinforcement learning for solving job shop scheduling problems. *Ieee Access*, 8:71752–71762.

Luo, S., Zhang, L., and Fan, Y. (2021). Real-time scheduling for dynamic partial-no-wait multiobjective flexible job shop by deep reinforcement learning. *IEEE Transactions on Automation Science and Engineering*, 19(4):3020–3038.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nat.*, 518(7540):529–533.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

Pezzella, F. and Merelli, E. (2000). A tabu search method guided by shifting bottleneck for the job shop scheduling problem. *European Journal of Operational Research*, 120(2):297–310.

Pham, D.-N. and Klinkert, A. (2008). Surgical case scheduling as a generalized job shop scheduling problem. *European Journal of Operational Research*, 185(3):1011–1025.

Pinedo, M. and Singer, M. (1999). A shifting bottleneck heuristic for minimizing the total weighted tardiness in a job shop. *Naval Research Logistics (NRL)*, 46(1):1–17.

Pinedo, M. L. (2018). *Scheduling: theory, algorithms, and systems*. Springer.

Plaat, A. (2022). *Deep Reinforcement Learning*. Springer.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1st edition.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *The Journal of Machine Learning Research*, 22(1):12348–12355.

Rajeswaran, A., Kumar, V., Gupta, A., Vezzani, G., Schulman, J., Todorov, E., and Levine, S. (2017). Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *arXiv preprint arXiv:1709.10087*.

Sabuncuoglu, I. and Bayiz, M. (1999). Job shop scheduling with beam search. *European Journal of Operational Research*, 118(2):390–412.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Syarif, A., Pamungkas, A., Kumar, R., and Gen, M. (2021). Performance evaluation of various heuristic algorithms to solve job shop scheduling problem (jssp). *International Journal of Intelligent Engineering & System*, 14(2):334–343.

Taillard, E. (1993). Benchmarks for basic scheduling problems. *european journal of operational research*, 64(2):278–285.

Tassel, P., Gebser, M., and Schekotihin, K. (2021). A reinforcement learning environment for job-shop scheduling. *arXiv preprint arXiv:2104.03760*.

Van Hentenryck, P., Michel, L., Perron, L., and Régin, J.-C. (1999). Constraint programming in opl. In *PPDP*, volume 99, pages 98–116. Springer.

Vinyals, O., Bengio, S., and Kudlur, M. (2015). Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*.

Wang, J., Luh, P. B., Zhao, X., and Wang, J. (1997). An optimization-based algorithm for job shop scheduling. *Sadhana*, 22:241–256.

Yu, H., Gao, Y., Wang, L., and Meng, J. (2020). A hybrid particle swarm optimization algorithm enhanced with nonlinear inertial weight and gaussian mutation for job shop scheduling problems. *Mathematics*, 8(8):1355.

Zahmani, M. H., Atmani, B., Bekrar, A., and Aissani, N. (2015). Multiple priority dispatching rules for the job shop scheduling problem. In *2015 3rd International Conference on Control, Engineering & Information Technology (CEIT)*, pages 1–6. IEEE.

Zhang, C., Song, W., Cao, Z., Zhang, J., Tan, P. S., and Chi, X. (2020). Learning to dispatch for job shop scheduling via deep reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1621–1632.

Zhang, W. and Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *IJCAI*, volume 95, pages 1114–1120. Citeseer.