IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 5 July 2022, accepted 3 September 2022, date of publication 16 September 2022, date of current version 26 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3207559

RESEARCH ARTICLE

Learning the Quality of Machine Permutations in Job Shop Scheduling

ANDREA CORSINI[1], SIMONE CALDERARA[2], (Member, IEEE), AND MAURO DELL'AMICO[1]
[1]Department of Sciences and Methods for Engineering, University of Modena and Reggio Emilia, 41121 Modena, Italy
[2]Department of Engineering Enzo Ferrari, University of Modena and Reggio Emilia, 41121 Modena, Italy

Corresponding author: Andrea Corsini (andrea.corsini@unimore.it)

ABSTRACT In recent years, the power demonstrated by Machine Learning (ML) has increasingly attracted the interest of the optimization community that is starting to leverage ML for enhancing and automating the design of algorithms. One combinatorial optimization problem recently tackled with ML is the Job Shop scheduling Problem (JSP). Most of the works on the JSP using ML focus on Deep Reinforcement Learning (DRL), and only a few of them leverage supervised learning techniques. The recurrent reasons for avoiding supervised learning seem to be the difficulty in casting the right learning task, i.e., what is meaningful to predict, and how to obtain labels. Therefore, we first propose a novel supervised learning task that aims at predicting the quality of machine permutations. Then, we design an original methodology to estimate this quality, and we use these estimations to create an accurate sequential deep learning model (binary accuracy above 95%). Finally, we empirically demonstrate the value of predicting the quality of machine permutations by enhancing the performance of a simple Tabu Search algorithm inspired by the works in the literature.

INDEX TERMS Deep learning, job shop scheduling, metaheuristic, recurrent neural network, scheduling.

I. INTRODUCTION

Nowadays, manufacturing and service industries are becoming larger, more interconnected, and generate every day a large volume of data. This increase in industrial complexity and the shift towards a 4.0 environment pose new challenges in scheduling and demands new personalized algorithms to maximize production while minimizing costs and processing times [1].

In recent years, there has been a surge of new techniques that take advantage of data generated by smart devices, sensors, and industrial systems. The discipline encompassing much of these techniques is machine learning. Machine learning demonstrated how data can be fruitfully used to achieve astonishing results in fields like computer vision and natural language processing. Based on this premise, ML constitutes a concrete opportunity to answer the new industrial demands.

However, ML is not yet mature and ubiquitous in all fields. One of these fields is combinatorial optimization, where only recent works achieve superior performance compared to few non-ML algorithms in problems like the travelling salesman problem [2], [3], the vehicle routing problem [4], scheduling [5], [6], [7], and others [8], [9]. These pioneering works demonstrated how ML can be applied to combinatorial problems, but, due to the limitations of these works and the partial coverage of the many ML paradigms, much more has to be discovered.

In this work, we focus on the *Job Shop scheduling Problem* [10], a notorious NP-hard combinatorial problem with many practical applications in industry. Simply put, the JSP is to schedule a set of jobs onto a set of machines by minimizing an objective function. The distinctive characteristic of the JSP is that each job consists of a strict chain of operations, each of which has to be processed on one and only one machine without interruptions (see Section I-A for the formal definition).

Mixed Integer Linear Programming (MILP) and Constraint Programming are two exact optimization methods to solve the JSP [11]. Although these methods are becoming everyday faster, they do not scale well on medium and large instances [11], and they become rapidly useless even in small but complex industrial environments [1]. For these reasons, approximation methods are still largely employed and constitute an active area of research, besides being one of the subjects of this work.

The associate editor coordinating the review of this manuscript and approving it for publication was Wei Wei.

Most of the recent ML-based works tackling the JSP rely on Deep Reinforcement Learning (DRL) techniques [5], [6], [7]. What makes (deep) reinforcement learning particularly appealing in the context of the JSP is its ability to learn from past decisions, without the need of labels and by correctly formulating the Markov Decision Process [12]. However, training effective DRL agents is a difficult optimization task, it is not easy to reproduce [13], and takes a lot of time [6], especially for Monte Carlo-based methods [12]. Therefore, we investigate herein whether it is possible to use a supervised learning approach to solve the JSP.

Our work has been guided by two fundamental questions: (i) what type of information might be used or might help solve a JSP instance? (ii) is it possible to learn this information in a supervised manner? These questions arise from the fact that not all the solutions to a JSP instance are feasible, i.e., respect the problem constraints, and for those feasible, the objective value (e.g., the makespan or the total tardiness) is not trivially derivable. For these reasons, the application of supervised learning to the JSP requires a learning task tightly related to the objective function, and that may fit in the back-propagation algorithm. We thus propose as a novel supervised learning task to *learn the quality of a machine permutation, i.e., how good is the sequence of operations on a machine*.

Understanding whether a sequence of operations on a machine is of "high quality" is a difficult task in the JSP [10]. Since having a method to judge machines is important, either for speeding up existing algorithms or even in machine-based decomposition, we present an original methodology to learn the quality of machines by means of *sequential deep learning* and a *MILP solver*. There already exists in literature approaches to evaluate a machine, most notably [14], but they frequently estimate the criticality of machines, a related but different concept. Contrary, we define the quality of a machine permutation as the *likelihood of finding this permutation in an optimal or near-optimal solution*.

We evaluate the impact of our proposal by comparing the results obtained with one of the best metaheuristics for the JSP, namely the Tabu Search (TS), with and without these quality estimations. In addition, we compare the results of the TS with some of the DRL approaches to justify our proposal for enhancing existing approximation algorithms.

Summarizing, the contributions of this work are: (i) we propose a novel supervised learning task for the JSP; (ii) we propose an original methodology to evaluate the quality of machine permutations by means of a MILP solver; (iii) we create a supervised dataset on which we train a sequential deep learning model; (iv) we test the advantages of our learning task in a Tabu Search algorithm. In the remainder of this work, we start by describing in Section II some of the start-of-the-art algorithms to solve the JSP and recent trends leveraging ML. In Section III, we present the mathematical intuition behind our learning task, the methodology to estimate the quality of machine permutations, the sequential deep learning model, and the Tabu Search. Finally, in Section IV we present the dataset, the performance of the learning model, the advantages of our proposal when used in a TS, and the comparison with the DRL approaches. In Section V we conclude with few considerations and future works.

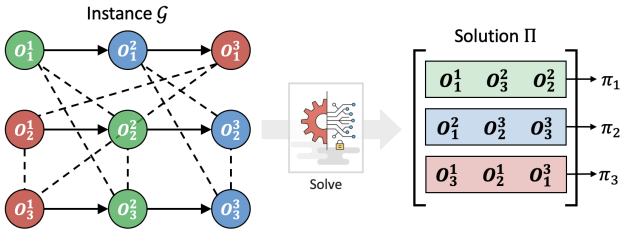

FIGURE 1. On the left, an example of a disjunctive graph that represents a JSP instance with 3 jobs (solid arrows) and 3 machines (dashed lines). On the right, a feasible solution that gives the sequence of operations for each machine.

A. JOB SHOP FORMULATION

The Job Shop Problem is as follows: we are given a set of n jobs $J = \{1, \ldots, n\}$, and a set of m machines $M = \{1, \ldots, m\}$. Each job $j \in J$ is composed by a sequence of $m_j \in \mathbb{N}$ operations $O_j = (o_j^1, \ldots, o_j^{m_j})$ that specifies in which order the jobs must be processed on the machines. Thus, an operation o_j^i belongs to job j, needs to be processed on machine $\mu_j^i \in M$ and has processing time $p_j^i \in \mathbb{R}_{>0}$. In this work, we consider as the objective of the JSP the *minimization of the makespan*, that is the total length in time required to complete all the jobs. Preemption is not allowed, and machines can handle one operation at a time. In scheduling theory, this problem is identified as $Jm \,||\, C_{max}$.

Solving a JSP instance means finding a permutation of operations on each machine such that the makespan is minimized, the precedence among the operations are respected, and the operations do not overlap on each machine. Let $\Pi = \{\pi_1, \ldots, \pi_m\}$ be a solution of a JSP instance, and $\pi_i = (s_1^i, s_2^i, \ldots, s_{n_i}^i)$ the permutation or sequence of the $n_i \in \mathbb{N}$ operations on machine $i \in M$. The permutation π_i fixes the order of operations on machine i, and s_k^i, with $k \in \{1, \ldots n_i\}$, gives the operation of some job j that is processed in the k^{th} position.

It is common to represent a JSP instance as a *disjunctive graph* $\mathcal{G} = (V, A, E)$ (see Figure 1); where V is the set of operations, A is the set of arcs connecting consecutive operations of the same job, and E is the set of disjunctive edges connecting operations to be processed on the same machine. In this representation, the problem of minimizing the makespan is reduced to finding an orientation to the edges in E such that the weighted longest path (a.k.a. the *critical path*) is minimized, where weights are the processing times of operations. We will refer to this set of oriented edges with $\hat{E}$ and the corresponding digraph with $\hat{\mathcal{G}} = (V, A, \hat{E})$. Finally, note there is a unique one-to-one correspondence between a generic solution Π_g and a digraph $\hat{\mathcal{G}}_g$ (orienting the edges of E is equivalent to creating permutations π_i and vice versa), and if this digraph is acyclic the solution is guaranteed to be feasible [10].

II. RELATED WORKS

In this section, we review two popular approximation methods for the JSP, namely *Priority Dispatching Rules* (PDR) and *Metaheuristics*, and some recent trends that leverage machine learning in such methods.

A. PRIORITY DISPATCHING RULES

A priority dispatching rule [15] is a heuristic method that assigns operations to machines based on priorities. In general, priorities are assigned with hard-coded rules that consider the status of the schedule or characteristics about jobs, machines, and operations. Designing an effective PDR is difficult and requires substantial domain knowledge, especially on complex problems like the JSP. Moreover, the performance of a PDR often varies drastically in different instances. Therefore, in the last decade, many researchers tried to automate the design of PDRs with the help of machine learning.

One of the first applications of ML to PDRs is presented in [16], where a neural network selects the most suited PDR among a pool of rules. The decisions of the neural network are based on the current system state and the training phase is done through simulations. In [17], an imitation learning method is proposed to learn PDRs by using the supervision of a MILP solver. This work demonstrated how learning from optimal solutions is not enough to produce robust PDRs.

Most of the recent research efforts focus on adapting DRL to learn PDRs. After a correct formulation of the Markov Decision Process, a policy to schedule operations is learnt from the experience derived by resolving the same instances many times. In [5], an actor-critic architecture [18] is proposed, where the critic evaluates the value of decisions in the partial schedule, whereas the actor learns to make decisions based on the schedule and the critic estimations. In [6], an actor-critic is also proposed, but with a Graph Neural Network [19] (GNN) to construct an adaptive representation of the partial schedule. One interesting aspect of this work is that the authors underline how GNNs seem to have poor performance when applied to disjunctive graphs. Another example of an actor-critic architecture coupled with a GNN is in [7]. This work applies GNNs to the disjunctive graphs of JSP instances by specifically designing a GNN architecture and by using a rich set of features to describe operations. From these works, it is not possible to draw any conclusion on the benefits of applying GNNs to disjunctive graphs, therefore, we prefer to avoid GNNs.

Although these promising works showed how to create superior PDRs, the performance of these proposals is still much worse than the performance of metaheuristics. Due to their lower performance and the lack of guarantees of producing high-quality solutions, PDRs still remain a valid alternative as generators of initial solutions for metaheuristics.

B. METAHEURISTICS

The general idea of a metaheuristic [20], [21] is to describe trajectories in the solution space starting from initial solutions and visiting neighbor solutions according to some criteria. Each trajectory generally stops either when no improving solution exists in the neighborhood, i.e., the current solution is a local optimum, or when a predefined criterion is met. The effectiveness of metaheuristics depends on a brittle and complex balance of its elements that governs the creation of successful trajectories. This balance is achieved by designing elements like the neighborhood structure, the searching procedure, and other mechanisms such that the algorithm can intensify promising regions while escaping from local optima. Therefore, selecting, designing, and assembling the right elements is extremely important and requires domain and algorithm-design expertise.

The breakthrough work in the field of metaheuristics for the JSP is [22]. This work adapted the Simulated Annealing (SA) [23] and proposed one of the most studied and effective neighborhood structures for the JSP, called N1. N1 was the first to show how it is possible to construct the neighborhood of a solution without incurring in unfeasible solutions. In addition, it guarantees the existence of a trajectory that leads to global minima, the so-called *convergence property*.

After this work, many variations and extensions of N1 were proposed in [24], [25], and [26], mostly in the context of a Tabu Search [27]. The most successful application of the TS to the JSP is [25], where the authors proposed a reduced variation of N1 in which some of the neighbor solutions were removed since they cannot immediately improve the current solution. In [25] is also proposed the best implementation of the TS for the JSP, successively refined in [28] by incorporating elements of path relinking in the generator of initial solutions.

Besides the TS and SA, there are other metaheuristics proposed to tackle the JSP [29], [30], [31]. In these regards, we just want to stress that regardless of the metaheuristics, e.g., Single-Source or Population-Based [20], an ad-hoc searching procedure or a local search is often required to enhance performance [29], [30], [31].

As reviewed in [32], ML can be fruitfully integrated in the most common metaheuristics and constitutes an opportunity to enhance, simplify, and automate the creation of effective algorithms. Some examples of how ML techniques can be integrated into metaheuristics for scheduling problems are [33], [34].

In [33], it is proposed a DRL-based rewriting method in which a region-picking policy selects regions of solutions that are rewritten with rules selected by a rule-picking policy. Picking the right regions and selecting the best rewriting rule are non-trivial operations, and learning to perform them from experience outperformed heuristic rules. In [34], a Variable Neighborhood Search is enhanced with a mechanism that favors the creation of solutions having promising attributes during the shaking step. Although this work does not use any ML techniques, learning to construct these solutions might be a viable and better approach. For other examples of how to combine ML with metaheuristics, we refer the reader to [32].

Despite these premises, metaheuristics did not receive the same attention as PDRs in hybridization with ML for the JSP, and we believe there is much to gain from such a combination.

III. PROPOSED METHODOLOGY

This section starts by outlining the proposed learning task and the mathematical intuition behind it. Then, we describe our methodology to evaluate the quality of machine permutations and the learning model to tackle the proposed task. Finally, we present the TS algorithm used to validate the advantages brought by our learning task.

A. LEARNING TASK

Our novel supervised learning task about the JSP is to *predict the quality of machine permutations, where the quality of a permutation is the likelihood of finding this permutation in an optimal solution*. We arrived at this formulation after carefully reviewing the abundant literature about the JSP in search of an answer to the first question of Section I: what type of information might be used for solving the JSP. To justify why our learning task should help in solving the JSP, we briefly report the intuition behind the proof of the convergence property of the N1 neighborhood (see [22] for the complete proof).

Let Π_1 and Π_o be respectively a feasible solution and an optimal solution of an instance. The converge property implies that from any Π_1, it is possible to construct a trajectory of solutions through N1 that allows moving from Π_1 to an optimal solution Π_o. The proof starts from the definition of a special set of critical arcs (remember that critical arcs are those arcs on the longest path in $\hat{\mathcal{G}}$):

$$K_1(\Pi_o) = \{(v, w) \in \hat{E}_1 \mid (v, w) \text{ is critical } \wedge\ (w, v) \in \hat{E}_o\} \tag{1}$$

that is the set of critical arcs in $\hat{\mathcal{G}}_1$ that do not belong to the optimal solution $\hat{\mathcal{G}}_o$. When $\Pi_1 \neq \Pi_o$, this set is always non-empty, and it is possible to create a finite trajectory $(\Pi_1, \Pi_2, \ldots, \Pi_o)$ that guarantees to reach an optimal solution, where Π_2 is obtained from Π_1 by reversing an arc in K_1. Clearly, the convergence is a desirable property for a neighborhood structure, but in practice, it is of no help because it requires to know the set of critical arcs to reverse, i.e., it requires K.

Nonetheless, this proof leads us to what might be beneficial for solving the JSP: *an information about which critical arcs are unlikely to be in an optimal solution*. At least in the context of N1, knowing this information allows excluding those solutions that introduce arcs unlikely to be in $\hat{E}_o$, resulting in better exploration and a faster convergence towards optima. However, there is a problem in learning a function that gives the likelihood of finding an arc in an optimal solution: the representation of the arc must encode enough information about the entire solution.

Instead of learning this function, we propose to learn a function that receives in input the machine permutation associated with an arc and outputs the likelihood of finding this permutation in an optimal solution. If a machine permutation resulting from the inversion of a critical arc is of higher quality than the original permutation, the reversed arc has a higher chance of being in $\hat{E}_o$. Therefore, learning such a function still allows to discriminate which critical arcs should be reversed. In addition, it simplifies the learning task since a permutation intrinsically encodes more information about the entire solution than a single arc.

Based on this theoretical intuition, our learning task should help solve the JSP in at least those approximation algorithms based on N1. Note that the proposed learning task might also benefit other approximation methods, for instance, machine-based decomposition and ruin-and-recreate algorithms [35], but proving this is outside the scope of this work.

B. QUALITY OF MACHINE PERMUTATIONS

Up to this point, we presented our novel learning task, and we justified why this task should help solve the JSP. What remains uncovered is how the quality y_k of a machine permutation π_k can be quantified. To define the quality y_k, we rely on the concept of makespan, and we compute:

$$y_k = 1 - \tanh\left(\frac{C_{max}(\pi_k)}{C^{opt}_{max}} - 1\right) \tag{2}$$

where $C_{max}(\pi_k)$ is the best makespan found by imposing π_k as part of the solution, C^{opt}_{max} is the optimal makespan of the instance, and tanh is the hyperbolic tangent function.

Note that Equation 2, beyond giving the mathematical definition of the quality of a machine permutation, also points out the methodology needed to estimate this quality. This methodology includes a method to optimally solve the JSP and a method to find the best solution with an imposed sequence π_k. With these methods, Equation 2 estimates the quality y_k by comparing the best makespan found with the sequence π_k against the optimal makespan, and it scales this comparison with the *tanh* function. When π_k is near-optimal, meaning that $C_{max}(\pi_k)$ is close to the optimal makespan, y_k takes a value close to 1. Contrary, when $C_{max}(\pi_k)$ is far from the optimal value, y_k takes a value close to 0. Due to its definition, the quality of a permutation is always a value in the interval $[0, 1] \subset \mathbb{R}$, thus, it can be interpreted as a kind of probability (or a likelihood parameterized by some parameters) of finding the permutation in an optimal solution.

As the method to optimally solve the JSP, we propose to use a MILP solver by formulating the problem as a disjunctive model [11]. As pointed out in [11], today solvers can solve instances with 10 jobs and 10 machines in few seconds.

Instead, as the method to find the best makespan $C_{max}(\pi_k)$ by imposing a sequence π_k, we propose to use a modified version of the standard disjunctive model, again in a MILP solver. In this modified version, we introduce a set of constraints to prevent the solver from changing the order of the sequence π_k. Note that this modification effectively reduces the solution space and speeds up the solver. The modified disjunctive model is then:

$$\min\ C_{max}(\pi_k) \tag{3}$$

$$\text{s.t. } x_{o_j^h}^i \geq x_{o_j^{h-1}}^i + p_{o_j^{h-1}}^i \quad \forall j \in J, h = 2, \ldots, m_j \tag{4}$$

$$x_j^i \geq x_k^i + p_k^i - Q\, z_{jk}^i \quad \forall j, k \in J, j < k, i \in M \tag{5}$$

$$x_k^i \geq x_j^i + p_j^i - Q\,(1 - z_{jk}^i) \quad \forall j, k \in J, j < k, i \in M \tag{6}$$

$$x_{s_h^i}^i \geq x_{s_{h-1}^i}^i \quad i \in M, h = 2, \ldots, n_i \tag{7}$$

$$C_{max}(\pi_k) \geq x_{o_j^{m_j}}^i + p_{o_j^{m_j}}^i \quad \forall j \in J \tag{8}$$

$$z_{jk}^i \in \{0, 1\} \quad \forall j, k \in J, i \in M \tag{9}$$

$$x_j^i \geq 0 \quad \forall j \in J, i \in M \tag{10}$$

The model has two decision variables: x_j^i gives the starting time of job j on machine i, and, z_{jk}^i takes value 1 if job j precedes job k on machine i. The set of constraints (4) guarantees that for each job, the start time of every operation must be equal to or higher than the completion time of the previous operation. The disjunctive constraints in sets (5) and (6) guarantee that the start time of an operation o_j^i must be higher than the completion time of another operation o_k^i when o_j^i is scheduled before o_k^i and vice versa. Finally, the set of constraints (7) fixes the order of operations on machine i to be equal to $\pi_k = (s_1^i, s_2^i, \ldots, s_{n_i}^i)$, and the set (8) computes the makespan. The value of Q is set to $\sum_{j \in J} \sum_{i \in M} p_j^i$ to ensure the correctness of the disjunctive constraints.

Summarizing, the methodology to obtain the quality of a machine permutation π_k starts by optimally solving the JSP instance, then the best makespan $C_{max}(\pi_k)$ is found with the presented modified disjunctive model, and finally, the quality is computed with Equation 2.

C. THE LEARNING MODEL

In order to predict the quality y_k of a sequence π_k, we designed a sequential deep learning model that is sensitive to the order of the input. We will refer to such a model as the *oracle*.

As a standard in sequential deep learning, each operation of a sequence $\pi_k = (s_1^i, s_2^i, \ldots, s_{n_i}^i)$ is described by a feature vector in $\mathbb{R}^g$. This means that the representation X_k of a sequence π_k is in turn a sequence of feature vectors, or alternatively, a tensor $X_k \in \mathbb{R}^{n_i \times g}$, where the $h \in \{1, \ldots, n_i\}$ element describes the operation s_h^i. More information about the features describing an operation is given in Section IV-A.

Our oracle is composed of two blocks: the *first block* takes in the representation of a sequence X_k and creates a *sequence embedding*; the *second block* uses this embedding to output the probability y_k of the sequence. The entire architecture is depicted in Figure 2.

The first block is realized with two layers of a Gated Recurrent Unit (GRU) [36]. A GRU is a type of Recurrent Neural Network [37] that uses a "memory structure" to let information from prior inputs influence the current output. This "memory structure" needs to be initialized to some initial state, and is updated at each time step by using the input and current state through a gating mechanism.

Our oracle warms start the initial states with X_k, but without considering the order. Specifically, the initial state of each GRU layer is created by first projecting the feature vectors describing operations in a latent space $\mathbb{R}^d$ with a hidden layer ($H_0 \in \mathbb{R}^{g \times d}$ and $H_1 \in \mathbb{R}^{g \times d}$ in Figure 2), and then by taking the mean along each of the d dimensions. This allows modeling the concept of a JSP machine directly in the architecture.

After this initialization, starts the creation of the sequence embedding by considering the order of the sequence. As depicted in Figure 2, the first GRU layer receives in input at each step $t = (1, \ldots, n_i)$ the feature vector of the operation s_t^i and produces in output the state h_t^0. Whereas the second GRU layer receives in input at the step t the state h_t^0 and produces in output h_t^1. The final sequence embedding is the concatenation of the last states, $h_{n_i}^0$ and $h_{n_i}^1$, and is therefore a vector in $\mathbb{R}^{2d}$.

The second block is realized with a Feedforward Neural Network (FNN) [37] composed by 3 hidden layers of decreasing size. This block takes in input the sequence embedding and produces in output the probability y_k.

D. TABU SEARCH

Since Tabu Search empirically demonstrated to be the best metaheuristic for solving the JSP [21], we evaluate the advantages of our novel learning task in this algorithm. To this end, we design two versions of the TS: sTS is a simple TS inspired by the works reviewed in Section II-B, while oTS is identical to sTS but uses the oracle. We borrow part of the structure of sTS and oTS from the TS proposed in [25]. Since our algorithms are almost identical, they differ only in the procedure to select the next solution, we first describe the structure of sTS and afterwards the modification to the searching procedure.

The most important blocks of sTS are: (i) the generator of the initial solution, (ii) the neighborhood structure, (iii) the tabu list for avoiding revisiting recent solutions, (iv) the neighborhood searching procedure to select the next solution, and (v) the restart list used to intensify promising regions of the solution space.

sTS begins by generating a random solution that constitutes both the starting point of the exploration and the initial best solution. This solution is generated with a random PDR that gives priority to jobs by sampling from a uniform distribution. We decided to use a random starting point to test the capability of our algorithms to converge to global optima in different runs of the same instance. This allows a better comparison between the algorithms.

After this initialization, sTS enters the cyclic phase where the following steps are repeated:

Step 1: *Create the neighborhood* of the current solution.
Step 2: Select the new current solution through the *neighborhood searching procedure*.
Step 3: *Update the best solution* if the new solution improves the best one.

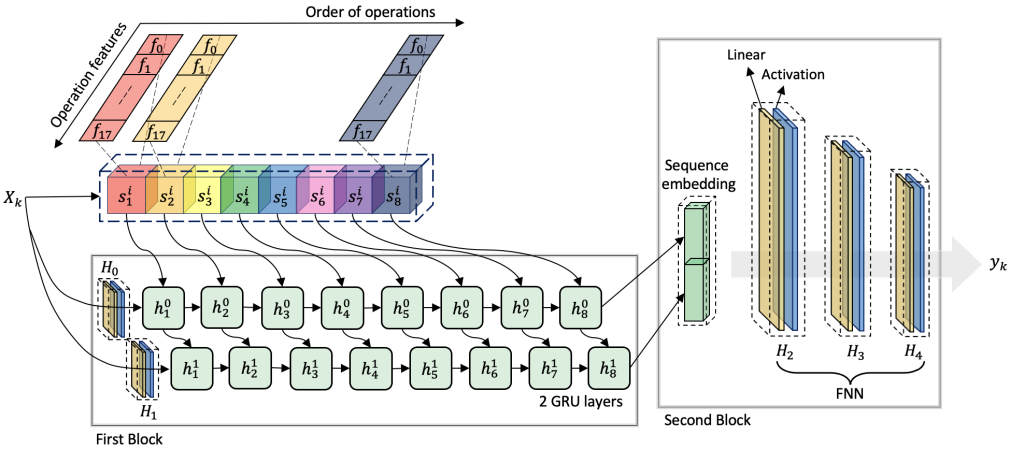

FIGURE 2. The architecture of the oracle. In the left, the 2-dimensional representation X_k of a sequence is transformed into a sequence embedding through the first block. In the right, the sequence embedding is fed into the second block to compute the quality y_k.

Step 4: Save a restart point in the restart list if the *region is promising*.

Step 5: Go to *Step 1:* if the *iteration condition* is met.

Step 6: Restart from the latest promising region and go to *Step 1:* if the *restart condition* is met.

At each iteration, the algorithm selects from the N1 neighborhood [22] the solution with the minimum makespan that is not forbidden by the tabu list (*Step 2:*). Once sTS finds a solution that improves the best one (*Step 3:*), it records this point in the restart list (*Step 4:*). Based on [25], [28], a *promising region* of the JSP solution space is a point in which there is an update of the best solution, and such regions must be intensified by trying to explore the entire neighborhood.

This cyclic exploration is repeated until a maximum number of non-improving iterations is reached (*iteration condition* of *Step 5:*), where a non-improving iteration is an iteration that does not improve the best solution. If the iteration condition is not met, the algorithm tries to resume the exploration from the last promising region inserted in the restart list. The *restart condition* of *Step 6:* simply checks that the restart list is not empty. If this condition is not met, the algorithm stops. The pseudo-code of the neighborhood searching procedure, the tabu list, and the restart list can be found in [25]

oTS is identical to sTS, but it uses the oracle to further reduce the N1 neighborhood by excluding solutions that lower the quality of machine permutations. This aligns with the discussion of Section III-A. Our oracle predicts the likelihood that a sequence (a permutation on some machine $i \in M$) has of belonging to an optimal solution. In N1, a neighbor solution Π_n differs from the current solution Π_c in only one permutation on a machine. Therefore, we use the oracle to remove all the neighbor solutions that introduce a sequence with a lower likelihood of belonging to an optimal solution. More in detail, if the permutation of Π_n on machine i has a higher likelihood of belonging to an optimal solution than Π_c, we accept this solution in the neighborhood, in the opposite case, we remove Π_n from the neighborhood. The searching procedure for selecting a new solution from this reduced neighborhood remains the same of sTS, that in turn is the same of [25]. There might be situations in which all the neighbor solutions are removed, in these cases, we undo the reduction and use the normal N1 neighborhood. Finally, this reduction is applied only for the first quarter of the maximum number of non-improving iterations (*Step* 1-5), and in the same way after every restart.

IV. EXPERIMENTAL RESULTS

In this section, we present the dataset used to train and test the neural network oracle of Section III-C, the results of the oracle on the test set, and the results of our algorithms on 200 JSP instances.

A. THE DATASET

We created a dataset of sequences from a set of 200 JSP instances with 8 jobs ($n_i = 8, \forall i \in M$) and 8 machines ($m_j = 8, \forall j \in J$). The set of instances has been generated following the guidelines of [38].

Then, for each instance, we generated 136 sequences for each machine, and we computed the quality of these sequences with the methodology introduced in Section III-B. This results for a single instance $q \in \{1, \ldots, 200\}$ in a total of 1088 observations of the form (X_k^q, y_k^q), where $X_k^q \in \mathbb{R}^{n_i \times g}$ is the representation of a machine sequence π_k, and y_k^q is its quality. To ease the notation, in the remainder of this work we omit the index of the instance q; nonetheless, remember that each observation of our dataset refers to one and only one instance.

The 136 sequences for each machine have been generated as follows:

- 128 *random sequences* by trying to place each operation in all positions of a machine (the pseudocode for generating such sequences is given in Appendix A.).
- 1 *optimal sequence* taken from the optimal solution of the instance.

- 7 *suboptimal sequences* obtained from the optimal sequence by swapping consecutive operations (we did not swap the first and last operations).

The rationale behind these different sequences is that we tried to uniformly sample the characteristics of a machine in an instance. The 128 random sequences should reflect the "unbiased" impact of the machine on the instance, the optimal sequence is introduced to model the optimality for a machine, and the suboptimal sequences are used to model the neighborhood of an optimal sequence, and hopefully the *Big Valley* phenomenon [28].

Regarding the representation X_k of a sequence π_k, we defined a set of 18 features to describe operations. Our set of features has been constructed by selecting some of the best features from [39] and from the graph theory. The features selected from [39] describe characteristics about single operations and jobs, some examples are: the processing time of operations and the mean processing time of jobs. The graph theory features are extracted from the disjunctive graph and they express relationships among operations, some examples are: the eigenvector centrality and the closeness centrality. These features depend only on information about the instance, therefore, in our experiments, we computed the feature vector for each operation once and we dynamically concatenated the feature vectors in the order given by π_k to form X_k ($X_k \in \mathbb{R}^{8\times 18}$ in this work). We report in Appendix C the complete set of features.

This dataset has been used to train and validate the neural network introduced in Section III-C.

B. LEARNING MODEL PERFORMANCE

We evaluate the performance of the oracle on two different aspects: (i) we quantify the error in the predictions by measuring how much they differ from labels, (ii) we quantify the performance of the oracle in a binary classification problem. The results of this section refer to a test set composed of 54400 sequences (25% of the dataset) randomly selected by ensuring that the test distribution is similar to the one of the entire dataset, see Figure 3.

Due to the nature of our labels $y_k \in [0, 1]$, we trained our oracle to approximate the distribution of the training set by using the Kullback–Leibler Divergence as the loss function. Using this loss allows to train the model without transforming the problem into a binary classification, and this brings several advantages: (i) our labels have a larger semantic compared to binary ones, giving more freedom in the application of the oracle; (ii) it is not clear which threshold should be set on the continuous labels to transform them into binary ones; (iii) casting the problem as a binary classification brings imbalance issues [40]. The whole set of hyperparameters and additional training details are given in Appendix B.

To quantify the errors of the oracle, we compare its predictions against the labels of the test set by defining the Within

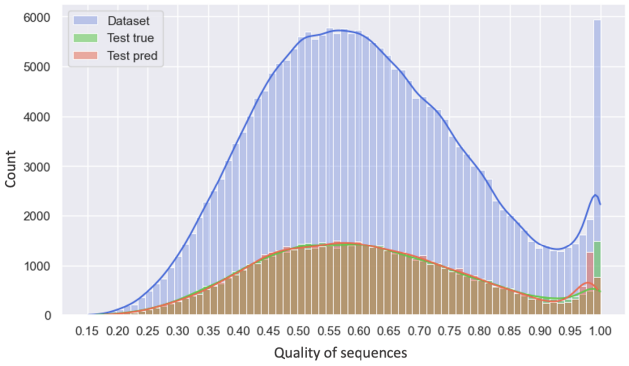

FIGURE 3. The discretized distributions of the dataset (blue), test set (green), and oracle predictions (red).

TABLE 1. The errors and WTA for different quality intervals.

Quality	Num	Abs error		WTA(.05)	WTA(.07)
		avg	max	(%)	(%)
$[0.0, 0.3)$	1117	0.012	0.078	98.0	99.7
$[0.3, 0.4)$	4756	0.015	0.111	95.8	99.3
$[0.4, 0.5)$	9710	0.016	0.158	94.2	98.6
$[0.5, 0.6)$	11775	0.018	0.150	91.7	97.8
$[0.6, 0.7)$	10808	0.020	0.171	89.2	97.1
$[0.7, 0.8)$	7702	0.020	0.158	87.3	96.7
$[0.8, 0.9)$	4343	0.021	0.152	85.9	96.2
$[0.9, 1.0]$	4189	0.018	0.123	91.3	97.9
Test set WTA				91.0	97.7

Tolerance Accuracy (WTA) in Equation 11:

$$\mathrm{WTA}(tol) = \frac{1}{t}\sum_{k=0}^{t} \mathbb{I}(|y_k - \hat{y_k}| < tol) \tag{11}$$

where $\hat{y_k}$ is the predicted quality of a sequence π_k, y_k is the true quality, *tol* is the error tolerance, $\mathbb{I}()$ is the indicator function (it returns 1 when the difference is within the tolerance), and t is the dimension of the test set. As it is clear from the distributions in Figure 3, the predictions of the oracle approximate well the distribution of the test set, with some mistakes in the region around the quality 0.90. This is even more clear from Table 1 which quantifies the errors between true and predicted values in different portions of the test distribution. This table divides the sequences in intervals based on the true quality. The first column points out the quality intervals, the second column gives the number of occurrences in each interval, the third and fourth columns give some statistics about the absolute error between labels and predictions, and the last two columns give the WTA for different tolerances. The last row gives the WTA for the entire test set.

Note how the WTA is almost perfect for a tolerance of 0.07, and still very good for a tighter tolerance of 0.05. As noted above, we can appreciate an increment in the errors in the interval [0.7, 0.9). We believe that this increment is jointly caused by the lower number of training sequences in this

TABLE 2. The results on 5 binary classification problems obtained by setting 5 different thresholds on the labels of the test set.

Threshold	Num positive	Num negative	Imbalance ratio	Accuracy (%)	Balanced accuracy (%)	Precision (%)	Recall (%)
0.5	38817	15583	0.4	95.5	94.3	96.6	97.0
0.6	27042	27358	1.01	94.7	94.7	94.6	94.8
0.7	16234	38166	2.35	95.4	94.5	92.3	92.2
0.8	8532	45868	5.38	97.1	94.0	91.9	89.4
0.9	4189	50211	11.99	98.6	94.4	92.2	89.4

interval and by the fact that these sequences are more difficult to discriminate from optimal ones because they are mostly suboptimal, i.e., they only differ from optima only in one consecutive pair of operations.

To better understand the quality of the oracle, we also report in Table 2 its performance in a binary classification task. In this evaluation, the true quality y_k are transformed into binary labels by setting a threshold and marking with a 1 (positive) all the sequences having a quality higher than the threshold, and with a 0 (negative) all the remaining sequences. The class predicted by the oracle is given by the *argmax* function. We report the results for 5 different thresholds, each producing a binary test set with a different imbalance ratio. Despite these different imbalance ratios, the performance of the oracle on standard imbalanced metrics [40] remains good in all cases. This is possible because we trained the model to match the quality of the training sequences.

With these evaluations, we want to stress how the oracle can be effectively used either for predicting or classifying sequences, allowing great flexibility in its usage within our TS and potentially in other approximation methods. In addition, the results of this section suggest that with our dataset, and hence with the methodology of Section III-B, it is possible to learn which sequences are likely to be in an optimal solution.

C. TABU SEARCH PERFORMANCE

We analyze the impact of the proposed learning task by comparing the results of the TS described in Section III-D with (oTS) and without (sTS) the oracle. This comparison is done on the 200 instances used to create our dataset, where for each instance we repeated the execution of the algorithms 5 times, from the same 5 initial solutions (this is done by seeding the random PDR with 5 different seeds). The results of the algorithms are compared in terms of the number of optimal solutions, the average optimality gap of suboptimal solutions ($\text{gap} = (C_{max}/C^{opt}_{max}) - 1$), and the average execution time. Note that comparing the results of the algorithms on the same instances of our dataset is fair since the sequences visited by sTS and oTS are independent from those used to train the oracle.

Both the algorithms have been written in C++, compiled with g++ 9.3.0, and executed on an Ubuntu machine equipped with an Intel Core i9-11900K and a NVIDIA GeForce RTX 3090. Our oracle has been ported from Python by using the tracing functionality of PyTorch [41], and it has been integrated in the oTS with LibTorch on the GPU.

In Table 3, we report the results of the algorithms for different configurations of parameters. In these configurations, we omit the length of the tabu list that is always set to 10. The first column of the table assigns an identifier to every configuration. The second and third column specifies respectively the maximum number of non-improving iterations and the length of the restart list. The "*Num opt*" and the "*Avg opt gap*" columns compare the number of optimal solutions and the average optimality gap of each algorithm. Whereas the "*Worse*" ("*Better*") columns compare respectively the number of solutions and the average scaled difference ($\text{diff} = (C^{oTS}_{max} - C^{sTS}_{max})/C^{opt}_{max}$) in which oTS worsens (improves) with respects to sTS. The last two columns give the average execution time of each algorithm.

First, we want to underline that oTS finds a higher number of optimal solutions than sTS regardless of the parameter configurations. This is important for empirically confirming that the proposed learning task, our methodology, and the learning model indeed enhance the performance of the TS.

This increment in performance is also supported by the lower average optimality gaps obtained by oTS in suboptimal solutions ("*Avg opt gap*" columns). For all the tested configurations, we only see one case, row with ID 7, in which oTS does slightly worse than sTS in terms of optimality gap. However, note how in this case the overall performance of both the algorithms is almost perfect, and how oTS is still able to find a larger number of optimal solutions.

Regarding the "*Worse*" and the "*Better*" columns, we just highlight how the number of solutions in which oTS does better than sTS is almost twice the number of solutions in which it does worse.

Finally, as it is obvious from the average times, using a deep learning model will likely increase the running time. This trend has already been observed for instance in [6], where the execution of their DRL proposal takes 2x up to 5x the time of traditional PDRs. A similar increment is also observed in [5]. In line with these works, we observe a comparable increment between sTS and oTS. However, our algorithms have been written by keeping the implementation as simple as possible. Therefore, there is space for engineering the code and producing better average execution times, especially in the case of the oTS. For instance, it is possible to reduce the calls to the oracle by batching or keeping a

TABLE 3. The results of the two TS algorithms for different configurations of parameters.

ID	Parameters		Num opt		Avg opt gap		Worse		Better		Avg time	
	Max iter	Restarts	sTS	oTS	sTS (%)	oTS (%)	Num	Avg diff (%)	Num	Avg diff (%)	sTS (ms)	oTS (ms)
0	500		335	465	2.24	2.12	277	1.56	408	1.93	29	142
1	1000		449	620	1.81	1.77	192	1.32	372	1.56	41	210
2	1500	0	511	673	1.66	1.41	167	1.06	341	1.55	56	318
3	2000		559	741	1.55	1.29	135	1.11	320	1.49	72	409
4	2500		593	763	1.51	1.34	116	1.10	293	1.46	86	473
5		0	382	548	1.99	1.75	223	1.22	406	1.75	33	153
6	700	1	741	849	1.02	0.92	80	0.93	195	1.02	138	526
7		2	807	892	0.89	0.92	65	0.97	150	0.89	199	690
8		0	409	572	1.89	1.71	211	1.28	395	1.65	35	165
9	800	1	754	867	1.01	0.84	71	0.79	185	1.04	181	592
10		2	818	905	0.89	0.85	58	0.80	140	0.91	245	795
11		0	425	594	1.84	1.69	218	1.24	377	1.70	38	191
12	900	1	766	879	1.00	0.97	66	0.96	173	1.04	204	714
13		2	833	918	0.88	0.82	48	0.93	128	0.97	263	961

TABLE 4. The comparison between oTS and the DRL approaches on benchmark instances.

	Instance	OPT	SPT	[5]	[6]	oTS-1	oTS-2
10×10	Orb01	1059	1478 (39.6%)	1211 (14.4%)	-	1106 (4.4%)	1106 (4.4%)
	Orb02	888	1175 (32.3%)	1002 (12.8%)	-	902 (1.6%)	902 (1.6%)
	Orb03	1005	1179 (17.3%)	1150 (14.4%)	-	1048 (4.3%)	1044 (3.9%)
	Orb04	1005	1236 (23.0%)	1132 (12.6%)	-	1032 (2.7%)	1032 (2.7%)
	Orb05	887	1152 (29.9%)	1045 (17.8%)	-	902 (1.7%)	896 (1.0%)
	Orb06	1010	1190 (17.8%)	1106 (9.5%)	-	1028 (1.8%)	1028 (1.8%)
	Orb07	397	504 (27.0%)	460 (15.9%)	-	397 (0.0%)	397 (0.0%)
	Orb08	899	1170 (30.1%)	1022 (13.7%)	-	911 (1.3%)	911 (1.3%)
	Orb09	934	1262 (35.1%)	1082 (15.8%)	-	961 (2.9%)	955 (2.2%)
15×15	Ta01	1231	1872 (52.1%)	-	1443 (17.2%)	1281 (4.1%)	1281 (4.1%)
	Ta02	1244	1709 (37.4%)	-	1544 (24.1%)	1283 (3.1%)	1283 (3.1%)
	Ta03	1218	2009 (64.9%)	-	1440 (18.2%)	1292 (6.1%)	1292 (6.1%)
	Ta04	1175	1825 (55.3%)	-	1637 (39.3%)	1248 (6.2%)	1248 (6.2%)
	Ta05	1224	2044 (67.0%)	-	1619 (32.3%)	1280 (4.6%)	1280 (4.6%)
	Ta06	1238	1771 (43.1%)	-	1601 (29.3%)	1272 (2.7%)	1260 (1.8%)
	Ta07	1227	2016 (64.3%)	-	1568 (27.8%)	1250 (1.9%)	1247 (1.6%)
	Ta08	1217	1654 (35.9%)	-	1468 (20.6%)	1240 (1.9%)	1240 (1.9%)
	Ta09	1274	1962 (54.0%)	-	1627 (27.7%)	1307 (2.6%)	1307 (2.6%)
	Ta10	1241	2164 (74.4%)	-	1527 (23.0%)	1290 (3.9%)	1290 (3.9%)

memory of past predictions, and it is possible to reduce the execution time of the oracle by using faster architectures like Transformers [42] and Convolutional Neural Network [37].

Concluding, this comparative analysis shows that it is possible to find better solutions by using the quality predictions in a TS as described in Section III-D. This empirically highlights how the proposed learning task seems to be valuable in the context of the JSP.

D. COMPARISON WITH REINFORCEMENT LEARNING

In this section, we compare the performance of oTS with the proposals relying on DRL. The objective is to justify our efforts by demonstrating the superiority of metaheuristics enhanced with machine learning and the importance of further investigating these hybrid approaches.

For this comparison, we selected instances from the works discussed in Section II-A. Specifically, we selected the instances Orb01-09 [43] and the instances Ta01-10 [38]. We report in Table 4 the instance name, the optimal makespan, and the results in terms of makespan and optimality gap (in round brackets) for the Shortest Processing

TABLE 5. Hyperparameters.

Block	hyperparameter	Value
GRU 0	hidden size	32
	bidirectional	False
	dropout	0.3
	H_0 size	32
	H_0 activation	tanh
	H_0 dropout	0.3
GRU 1	hidden size	32
	bidirectional	False
	H_1 size	32
	H_1 activation	tanh
	H_1 dropout	0.3
FNN	H_2 size	32
	H_2 activation	tanh
	H_3 size	16
	H_3 activation	tanh
	H_4 size	2

TABLE 6. The set of features describing an operation and its relations with others in the JSP instance.

Feature	Name	Equation	Description
f_0	Processing time	$\frac{p_j^i}{\max_{ik} p_k^i}$	The processing time of operation o_j^i normalized by the maximum processing time in the instance.
f_1	Job completion	$\frac{\sum_{k=1}^{i} p_j^k}{\sum_{k=1}^{m_j} p_j^k}$	The completion of job j when its operation o_j^i is scheduled.
f_2	Job mean	$\frac{1}{\text{avg} * m_j} \sum_{i=1}^{m_j} p_j^i$	Mean processing time of job j normalized by the mean processing time of the instance (avg in the equation).
f_3	Job median	$\frac{\text{median}(p_j^1,\ldots,p_j^{m_j})}{\text{avg}}$	Median processing time of job j scaled by the mean processing time of the instance (avg in the equation).
f_4	Job std-mean	$\frac{\text{std}(p_j^1,\ldots,p_j^{m_j})\, m_j}{\sum_{i=1}^{m_j} p_j^i}$	The standard deviation of the processing time in job j normalized by the mean processing time of the job.
f_5	Job std-median	$\frac{\text{std}(p_j^1,\ldots,p_j^{m_j})}{\text{median}(p_j^1,\ldots,p_j^{m_j})}$	The standard deviation of the processing time in job j normalized by the median processing time of the job.
f_6	Job min	$\frac{\min_{i\in\{1,\ldots,m_j\}} p_j^i}{\max_{ik} p_k^i}$	The minimum processing time of job j normalized by the maximum processing time in the instance.
f_7	Job max	$\frac{\max_{i\in\{1,\ldots,m_j\}} p_j^i}{\max_{ik} p_k^i}$	The maximum processing time of job j normalized by the maximum processing time in the instance.
f_8	Source shortest distance	$d^*(\text{src}, o_j^i)$	The shortest weighted distance in the graph from the source (dummy) node to operation o_j^i.
f_9	Destination shortest distance	$d^*(o_j^i, \text{dst})$	The shortest weighted distance in the graph from operation o_j^i to the destination (dummy) node.
f_{10}	Eigenvector Centrality	$Ax = x\lambda$	The eigenvector centrality of an operation o_j^i is the element of the eigenvector x associated with the largest eigenvalue λ that corresponds to o_j^i. A high eigenvector centrality means that an operation connects to other operations having high centrality. A is the adjacency matrix of the graph.
f_{11}	Weighted Eigenvector Centrality	$A^*x = x\lambda$	The same as the eigenvector centrality, but it uses the weighted adjacency matrix A^* where arcs take the weight of the source node.
f_{12}	Closeness Centrality	$\frac{\lvert V\rvert-1}{\sum_{k\in\mathcal{N}(o_j^i)} d(k,o_j^i)}$	The normalized closeness centrality measures the shortest non-weighted distance from the nodes than can reach o_j^i, scaled by the number of nodes in the graph. $\mathcal{N}(o_j^i)$ is the set of nodes that can reach o_j^i, $d(k, o_j^i)$ is the number of arcs on the shortest path from k to o_j^i, and $\lvert V\rvert$ is the number of nodes.
f_{13}	Weighted Closeness Centrality	$\frac{\lvert V\rvert-1}{\sum_{k\in\mathcal{N}(o_j^i)} d^*(k, o_j^i)}$	The same as the closeness centrality, but it uses the weighted shortest path $d^*(k, o_j^i)$.
f_{14}	Betweenness Centrality	$\sum_{v,w\in V} \frac{\Gamma_{v\to w}(o_j^i)}{\Gamma_{v\to w}}$	The betweenness centrality is the fraction of all-pairs shortest paths that pass through operation o_j^i. This measure indicates which operations are "bridges" between others in a graph. $\Gamma_{v\to w}$ is the number of non-weighted shortest paths from v to w, and $\Gamma_{v\to w}(o_j^i)$ is the number of such shortest paths through o_j^i.
f_{15}	Weighted Betweenness Centrality	$\sum_{v,w\in V} \frac{\Gamma^*_{v\to w}(o_j^i)}{\Gamma^*_{v\to w}}$	The same as the betweenness centrality, but it uses the weighted shortest path for computing the number of paths $\Gamma^*_{v\to w}$.
f_{16}	Page Rank	A	The Page Rank.
f_{17}	Weighted Page Rank	A^*	The weighted Page Rank.

Time (SPT), the proposal of [5], the proposal of [6], and oTS. Based on Table 3, we decided to use 2 parameter configurations for oTS: 2 restarts and 700 iterations for oTS-1, and 2 restarts and 800 iterations for oTS-2.

From Table 4, it is immediately clear that oTS outperforms the DRL proposals. This is also true if we qualitatively compare the results of oTS with those reported in [7]. By looking at the average percentage gap reported for Orb01-10 and Ta01-80, we can see that the gap of this other DRL proposal is around 20%, ten times the gap obtained by oTS.

This comparison demonstrates that metaheuristics enhanced with machine learning guarantee to find better solutions. We believe that further research in hybrid approaches as our may give life to simpler and better metaheuristics capable of producing near-optimal solutions in a shorter amount of time.

V. CONCLUSION

In this work, we proposed a novel supervised learning task for the JSP that aims at predicting the quality of machine permutations. We designed an original methodology to estimate this quality by means of a MILP solver. Then, we constructed a dataset with this methodology, and we demonstrated that is possible to learn a flexible and accurate sequential deep learning model to predict the quality of machine permutations. Finally, we justified both theoretically and empirically the benefits of using the proposed learning task in the context of metaheuristics for the JSP.

Our aim was to propose a simple and reasonable methodology that allows evaluating the benefits of applying supervised learning to the JSP. Although DRL seems a more natural and established ML paradigm for this problem, our analysis suggests that also supervised learning is a valuable and viable paradigm, especially if used in tandem with existing approximation methods.

In future works, we will address the main limitation of our hybrid metaheuristic: the increase in the execution time of the algorithm. We will investigate whether our learning task could benefit other methods for solving the JSP, like machine-based decomposition and ruin-and-recreate algorithms, and whether it is possible to develop new ad-hoc methods. In addition, we believe there is a need to extensively compare the benefits and drawbacks of the new ML-based proposals with a wide spectrum of well-established algorithms.

APPENDIX A
SEQUENCE GENERATOR

The sequence generator procedure takes in a sequence of operations on some machine $i \in M$ and generates s random sequences. This procedure is applied on each machine of an instance, and it tries to place each operation in all the n_i positions of a permutation. Note that this procedure may generate repeated sequences. Such repeated sequences must be removed, and the procedure must be called again to ensure that s different sequences are generated. The symbol $\oplus$ indicates that an item is appended to a partial sequence.

function SequenceGenerator($(s^i_1, \ldots, s^i_{n_i}), s$)
  $seq \leftarrow$ Generate s empty sequences
  $w \leftarrow (s^i_1, \ldots, s^i_{n_i})$
  for all $pos \in \{1, \ldots, n_i\}$ **do**
    $idx = 0$
    for all $k \in \{0, \ldots, s-1\}$ **do**
      while w_{idx} in seq_k **do**
        $idx = (idx + 1) \bmod |w|$
      end while
      $seq_k \leftarrow seq_k \oplus w_{idx}$
      $idx = (idx + 1) \bmod |w|$
    end for
    $w \leftarrow w \oplus (s^i_1, \ldots, s^i_n)$ ▷ Increase w's period.
    $w \leftarrow$ shuffle(w)
  end for
  return seq
end function

APPENDIX B
HYPERPARAMETERS AND TRAINING DETAILS

Table 5 reports the hyperparameter of the sequential deep learning model of Section III-C. This model has been trained with the adam optimizer [44], with a batch size of 128, and for a total of 100 epochs divided as follows:

1) 40 epochs with learning rate 0.005.
2) 30 epochs with learning rate 0.002.
3) 20 epochs with learning rate 0.001.
4) 10 epochs with learning rate 0.0005.

APPENDIX C
FEATURES

Table 6 reports the set of 18 features used to describe operations in this work. The first column gives to each feature a unique identifier (in accordance with Figure 2), the second column points out the feature name, the third the equation, and the last column a brief description about the feature. The graph theory features (from row f_8 to row f_{17}) have been computed with the NetworkX package.

REFERENCES

[1] J. Zhang, G. Ding, Y. Zou, S. Qin, and J. Fu, "Review of job shop scheduling research and its new perspectives under industry 4.0," *J. Intell. Manuf.*, vol. 30, no. 4, pp. 1809–1830, Apr. 2019.
[2] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," in *Proc. Int. Conf. Learn. Represent.*, 2017, pp. 1–15.
[3] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.
[4] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác, "Reinforcement learning for solving the vehicle routing problem," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, 2018, pp. 1–11.
[5] C.-L. Liu, C.-C. Chang, and C.-J. Tseng, "Actor-critic deep reinforcement learning for solving job shop scheduling problems," *IEEE Access*, vol. 8, pp. 71752–71762, 2020.
[6] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and X. Chi, "Learning to dispatch for job shop scheduling via deep reinforcement learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 1621–1632.
[7] J. Park, J. Chun, S. Kim, Y. Kim, and J. Park, "Learning to schedule job-shop problems: Representation and policy learning using graph neural network and reinforcement learning," *Int. J. Prod. Res.*, vol. 59, pp. 1–18, Jan. 2021.

[8] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'horizon," *Eur. J. Oper. Res.*, vol. 290, no. 2, pp. 405–421, Apr. 2021.
[9] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev, "Reinforcement learning for combinatorial optimization: A survey," *Comput. Oper. Res.*, vol. 134, Oct. 2021, Art. no. 105400.
[10] M. L. Pinedo, *Scheduling*, vol. 29. New York, NY, USA: Springer, 2012.
[11] W.-Y. Ku and J. C. Beck, "Mixed integer programming models for job shop scheduling: A computational analysis," *Comput. Oper. Res.*, vol. 73, pp. 165–173, Sep. 2016.
[12] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
[13] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proc. AAAI Conf. Artif. Intell.*, Apr. 2018, vol. 32, no. 1, pp. 1–8.
[14] J. Adams, E. Balas, and D. Zawack, "The shifting bottleneck procedure for job shop scheduling," *Manage. Sci.*, vol. 34, no. 3, pp. 391–401, Mar. 1988.
[15] R. Haupt, "A survey of priority rule-based scheduling," *Oper. Res. Spektrum*, vol. 11, no. 1, pp. 3–16, Mar. 1989.
[16] W. Mouelhi-Chibani and H. Pierreval, "Training a neural network to select dispatching rules in real time," *Comput. Ind. Eng.*, vol. 58, no. 2, pp. 249–256, Mar. 2010.
[17] H. Ingimundardottir and T. P. Runarsson, "Discovering dispatching rules from data using imitation learning: A case study for the job-shop problem," *J. Scheduling*, vol. 21, no. 4, pp. 413–428, Aug. 2018.
[18] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 12, 1999, pp. 1–7.
[19] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021.
[20] E.-G. Talbi, *Metaheuristics: From Design to Implementation*, vol. 74. Hoboken, NJ, USA: Wiley, 2009.
[21] E. H. L. Aarts, P. J. M. van Laarhoven, J. K. Lenstra, and N. L. J. Ulder, "A computational study of local search algorithms for job shop scheduling," *ORSA J. Comput.*, vol. 6, no. 2, pp. 118–125, May 1994.
[22] P. J. M. Van Laarhoven, E. H. L. Aarts, and J. K. Lenstra, "Job shop scheduling by simulated annealing," *Oper. Res.*, vol. 40, no. 1, pp. 113–125, Feb. 1992.
[23] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.
[24] M. Dell'Amico and M. Trubian, "Applying Tabu search to the job-shop scheduling problem," *Ann. Oper. Res.*, vol. 41, no. 3, pp. 231–252, Sep. 1993.
[25] E. Nowicki and C. Smutnicki, "A fast taboo search algorithm for the job shop problem," *Manag. Sci.*, vol. 42, no. 6, pp. 797–813, Jun. 1996.
[26] C. Zhang, P. Li, Z. Guan, and Y. Rao, "A Tabu search algorithm with a new neighborhood structure for the job shop scheduling problem," *Comput. Oper. Res.*, vol. 34, no. 11, pp. 3229–3242, Nov. 2007.
[27] F. Glover and M. Laguna, *Tabu Search*. Boston, MA, USA: Springer, 1998, pp. 2093–2229.
[28] E. Nowicki and C. Smutnicki, "An advanced Tabu search algorithm for the job shop problem," *J. Scheduling*, vol. 8, no. 2, pp. 145–159, Apr. 2005.
[29] K.-L. Huang and C.-J. Liao, "Ant colony optimization combined with taboo search for the job shop scheduling problem," *Comput. Oper. Res.*, vol. 35, no. 4, pp. 1030–1046, Apr. 2008.
[30] D. Y. Sha and C.-Y. Hsu, "A hybrid particle swarm optimization for job shop scheduling problem," *Comput. Ind. Eng.*, vol. 51, no. 4, pp. 791–808, Dec. 2006.
[31] R. Cheng, M. Gen, and Y. Tsujimura, "A tutorial survey of job-shop scheduling problems using genetic algorithms. Part II: Hybrid genetic search strategies," *Comput. Ind. Eng.*, vol. 36, no. 2, pp. 343–364, Apr. 1999.
[32] E.-G. Talbi, "Machine learning into metaheuristics: A survey and taxonomy," *ACM Comput. Surv.*, vol. 54, no. 6, pp. 1–32, Jul. 2022.
[33] X. Chen and Y. Tian, "Learning to perform local rewriting for combinatorial optimization," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–12.
[34] S. Thevenin and N. Zufferey, "Learning variable neighborhood search for a scheduling problem with time windows and rejections," *Discrete Appl. Math.*, vol. 261, pp. 344–353, May 2019.
[35] G. Schrimpf, J. Schneider, H. Stamm-Wilbrandt, and G. Dueck, "Record breaking optimization results using the ruin and recreate principle," *J. Comput. Phys.*, vol. 159, no. 2, pp. 139–171, Apr. 2000.
[36] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," in *Proc. NIPS Deep Learn. Represent. Learn. Workshop*, Dec. 2014.
[37] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org
[38] E. Taillard, "Benchmarks for basic scheduling problems," *Eur. J. Oper. Res.*, vol. 64, no. 2, pp. 278–285, 1993.
[39] S. Mirshekarian and D. N. Šormaz, "Correlation of job-shop scheduling problem features with scheduling efficiency," *Expert Syst. Appl.*, vol. 62, pp. 131–147, Nov. 2016.
[40] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009.
[41] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32. Red Hook, NY, USA: Curran Associates, 2019, pp. 8024–8035.
[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30. Red Hook, NY, USA: Curran Associates, 2017, pp. 1–11.
[43] D. Applegate and W. Cook, "A computational study of the job-shop scheduling problem," *ORSA J. Comput.*, vol. 3, pp. 149–156, May 1991.
[44] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent.*, Dec. 2014, pp. 1–15.

ANDREA CORSINI received the B.S. and M.S. degrees in computer engineering from the University of Modena and Reggio Emilia, Modena, Italy, in 2018 and 2020, respectively. He is currently pursuing the Ph.D. degree in industrial innovation engineering with the University of Modena and Reggio Emilia. His current research interests include operations research and machine learning, with a particular focus on how to apply deep learning for solving combinatorial optimization problems.

SIMONE CALDERARA (Member, IEEE) received the master's degree in computer engineering and the Ph.D. degree from the University of Modena and Reggio Emilia, Modena, Italy, in 2005 and 2009, respectively. He is currently an Assistant Professor with the Imagelab Group, University of Modena and Reggio Emilia. His current research interests include computer vision and machine learning applied to human behavior analysis, visual tracking in crowded scenarios, and time series analysis for forensic applications.

MAURO DELL'AMICO is currently a Full Professor of operational research with the University of Modena and Reggio Emilia. He has almost three decades of academic experience in combinatorial optimization and operations research, primarily applied to mobility, logistics, transportation, supply chain management, production scheduling and planning, and network planning. He has participated as a principal investigator in many EU and Italian funded research projects in optimization, logistics, ICT, transportation, and scheduling. He combines the academic activities with consultancy on optimization for private and public companies. He is a member of the scientific board of several conferences and journals in operations research.

Open Access funding provided by 'Università degli Studi di Modena e Reggio Emilia' within the CRUI CARE Agreement