Job Shop Scheduling via Deep Reinforcement Learning: a Sequence to Sequence approach

Giovanni Bonetta*[0000−0003−4498−1026], Davide Zago*[0000−0003−1112−3543], Rossella Cancelliere[0000−0002−9120−3799], and Andrea Grosso[0000−0002−9926−2443]

Department of Computer Science, University of Turin, 10149 Turin
{giovanni.bonetta,rossella.cancelliere,andrea.grosso}@unito.it
zago@di.unito.it

Abstract. Job scheduling is a well-known Combinatorial Optimization problem with endless applications. Well planned schedules bring many benefits in the context of automated systems: among others, they limit production costs and waste. Nevertheless, the NP-hardness of this problem makes it essential to use heuristics whose design is difficult, requires specialized knowledge and often produces methods tailored to the specific task. This paper presents an original end-to-end Deep Reinforcement Learning approach to scheduling that automatically learns dispatching rules. Our technique is inspired by natural language encoder-decoder models for sequence processing and has never been used, to the best of our knowledge, for scheduling purposes. We applied and tested our method in particular to some benchmark instances of Job Shop Problem, but this technique is general enough to be potentially used to tackle other different optimal job scheduling tasks with minimal intervention. Results demonstrate that we outperform many classical approaches exploiting priority dispatching rules and show competitive results on state-of-the-art Deep Reinforcement Learning ones.

Keywords: Optimal Job Scheduling · Deep Reinforcement Learning · Combinatorial Optimization · Sequence to Sequence.

1 Introduction

Job Shop Problem (JSP) is a well-known Combinatorial Optimization problem fundamental in various automated systems applications such as manufacturing, logistics, vehicle routing, telecommunication industry, etc... In short, some jobs with predefined processing constraints have to be assigned to a set of heterogeneous machines, to achieve the desired objective (e.g. minimizing the flowtime). Due to its NP-hardness, finding exact solutions to the JSP is often impractical (or impossible, in many real-world scenarios), but many tasks can be effectively addressed through heuristics [7,9] or approximate methods [11], that represent the most suitable choice for large-scale problems, providing near optimal solutions with acceptable computational times.

Heuristic algorithms are classified as constructive or as local search methods. Constructive heuristics assemble the solution with an incremental process: at each

* Equal contribution

step, the choice of the next element in the solution is made by examining some local information of the problem, and once one variable has been fixed it's not reconsidered. *Priority Dispatching Rules* (PDRs) [9] belong to the category of constructive approximate methods: each operation is allocated in a dispatching sequence following a monotonic utility measure.

The use of dispatching rules emerged very early in the scheduling area, and it is well established by now. Most dispatching rules are known to be less than a match for modern, sophisticated heuristic optimization techniques (e.g. simulated annealing, tabu search, etc); despite this, they are still commonly used in many practical contexts because they are considered quick, flexible and adaptable to many situations. Besides, PDRs are widely used in real-world scheduling systems because they are intuitive and easy to implement. As a result, optimization literature is rich of PDR methods for the JSP [16], even if it is well known that designing an effective PDR is time-consuming and requires a substantial domain knowledge.

A possible solution is the automation of the process of designing dispatching rules: recent works on learning algorithms for Combinatorial Optimization (see [3] for a survey) show that Deep Reinforcement Learning (RL) could be an ideal technique for this purpose, and in particular that it can be considered a potential breakthrough in the construction of heuristic methods for the JSP [4]. Reinforcement Learning [18] is a subfield of Machine Learning (ML) that experienced a great development in recent years, mainly thanks to the contribution of Deep Learning.

The main idea of this paper is to treat the JSP as a sequence to sequence process: inspired by deep learning natural language models we propose a Deep Reinforcement Learning approach that, exploiting the encoder-decoder architecture typical of language, automatically learns robust dispatching rules. This leads us to consider PDRs as a reasonable match for deep RL-based optimization techniques that, it should be remembered, despite of the huge amount of works appearing on the subject, are still in their infancy.

Our method is able to learn dispatching rules with higher performance than traditional ones, e.g. *Shortest Processing Time* (SPT), *Most Work Remaining* (MWKR). On top of that, our approach shows competitive results against state-of-the-art Deep RL methods when tested on small and medium sized JSP benchmark instances. Besides, it shows a high degree of flexibility: *Flow Shop Problem* (FSP) instances can also be solved, and minimal modifications to the model would allow solving *Open Shop Problem* (OSP).

Since the model requires sequences as inputs and outputs we design an appropriate, yet compact and easily interpretable encoding for JSP instances and solutions. Besides, thanks to a tailored masking procedure, the model outputs a permutation of job operations (virtually a priority list) that respects precedence constraints and can be mapped to a *schedule*, i.e. the association of each operation to a specific starting time.

The rest of this paper is organized as follows: section 2 contains an overview of related works concerning neural and Deep Reinforcement Learning methods for Combinatorial Optimization (CO). Section 3 provides the definition of Markov Decision Process (MDP) and the theoretical foundations of our model. Section 4 introduces the mathematical notation which formalizes the JSP. Section 5 describes

our technique for sequence encoding, the neural architecture used and the proposed masking mechanism, the experimentation details and the results obtained.

2 Related works

Before Deep RL gained the popularity it has today, many ML-based approaches have been applied to CO (see [17] for an in-depth overview), such as assignment problems, cutting stock and bin packing problems, knapsack problems, graph problems, shortest path problems, scheduling problems, vehicle routing problems and the Travelling Salesman Problem (TSP). In the last decade, Natural Language Processing research inspired the formulation of very effective models such as the *Pointer Networks* [21], a deep architecture which builds upon recurrent neural networks. These innovative models have the ability to tackle problems where the number of output tokens varies with the input, a feature that characterized also many CO problems and, exploited for solving the TSP, shown interesting results and great potential.

One of the first attempts of applying the results of Vinyals et al. [21] is the work by Bello et al. [2], which successfully addresses the TSP and the Knapsack Problem (KP) in the context of Markov Decision Processes. It introduces active search, i.e. an RL-based technique that starting from a random (or pre-trained) policy iteratively optimizes the parameters on a single test instance. Deudon et al. [5] and Kool et al. [12] independently proposed a model inspired by the transformer architecture from Vaswani et al. [20] for solving the TSP. More specifically, the proposed architecture is made of an attention-based encoder in combination with a Pointer Network decoder.

The attempt to apply Deep RL to scheduling, and in particular to the JSP, is a phenomenon of growing research interest in recent years. We remand to section 3 and section 4 for all definitions concerning MDPs and the JSP. Waschneck et al., [22] present one of the first relevant works: in the context of MDPs each machine of the JSP is considered as an agent. The resulting multi-agent system is trained with Deep Q-Network (DQN) and, despite not showing higher performance with respect to other heuristics, this model obtains expert-level results. A similar multi-agent method is proposed by Liu et al. [14], where training is based on Deep Deterministic Policy Gradient (DDPG) algorithm. Their approach succeed in reaching higher performance with respect to some dispatching rules.

The approach from Lin et al. [13] assigns a different dispatching rule to each machine. After the training, done using a multi-class DQN, their method performs better that individual dispatching rules, but is far from being optimal.

An other interesting approach focuses on the disjunctive graph representation of the JSP. Zhang et al. [24] use a Graph Neural Network to map the states into an embedding space, followed by a Multi-Layer Perceptron which provides a probability distribution over the possible actions. This method obtains competitive performance and can be easily scaled to larger instances. We chose to compare our proposed approach to this work since it is, at the best of our knowledge, the best performing Deep RL approach to the JSP.

Han and Yang's work [8] presents a technique which, differently from all other summarised here, utilizes a Convolutional Neural Network on images for encoding the state of the problem and operates as a state-action function approximator. The images, which are produced using the disjunctive graph, have three channels representing the features: processing time, current schedule, and machine availability. The action space corresponds to different dispatching rules, whereas the reward function highlights machine utilization.

3 Mathematical foundations

RL substantially differs from other ML paradigms since it's concerned with how an agent learns to act in an environment: agents' behavior is optimized through a training phase, requiring the definition of a Markov Decision Process [1], focused on the maximization of a cumulative expected reward collected through a sequence of actions.

An MDP is a mathematical framework used to formalize a general decision making process involving a single agent acting in an environment.

An MDP is a tuple $M = (S, A, R, T, \gamma, H)$ where:

- S - *state space.*
  It is the set of all the possible representations s of the environment and of the agent's internal state at a given time.
- A - *action space.*
  It is the set of all the possible actions a the agent can perform.
- R - *reward function* $R : S \times A \times S \to \mathbb{R}$.
  It is the reward given to the agent after doing action a in state s and landing in state s'.
- T - *transition function* $T(s'|s, a)$.
  It is the transition probability from state s to s' given that action a has been performed.
- γ - *discount factor.*
  It weights the rewards of future actions. $\gamma \in [0, 1]$.
- H - *time horizon.*
  It is the maximum number of transition that can occur before the decision process is halted.

The objective of RL is to maximize the expected return of the sequence of actions performed by the agent. Each action is sampled from a *stochastic policy* $\pi(a|s)$, with $a \in A$ and $s \in S$, i.e. a probability distribution over the set of actions given a particular state.

3.1 Policy gradient algorithms

Policy gradient (or *policy optimization*) methods [18] are widely used in Deep RL research and they aim to directly optimize the stochastic policy π_θ, which is approximated by a neural network with parameters θ.

By taking actions in the environment, the agent defines trajectories. A *trajectory* τ (alternatively *episode* or *rollout*) is a sequence of states and actions $(s_0, a_0, s_1, a_1, ..., s_{H-1}, a_{H-1}, s_H)$ and it has a return $R(\tau)$ associated to it:

$$R(\tau) = \sum_{t=0}^{H} R(s_t, a_t, s_{t+1}) \tag{1}$$

$R(\tau)$ is called *finite-horizon undiscounted return* since it's defined with horizon H. Moreover, the probability of a trajectory given the policy is:

$$P_\theta(\tau) = \rho(s_0) \prod_{t=0}^{H} T(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t) \tag{2}$$

where $\rho(s_0)$ is the a priori probability of state s_0.

Given the parameterized stochastic policy π_θ, the learning objective is the maximization of the expected return w.r.t. a set of trajectories:

$$\max_\theta J(\pi_\theta), \text{ where } J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)] \tag{3}$$

Considering a policy optimized with gradient ascent, the quantity $\nabla_\theta J(\pi_\theta)$ is called *policy gradient* and the following equation holds:

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right] \tag{4}$$

This leads to the *REINFORCE* algorithm (algorithm 1), also known as *Vanilla policy gradient*, for optimizing policies, first proposed by Williams in [23].

Algorithm 1: REINFORCE

Input: MDP $M = (S, A, T, R, \gamma, H)$
Output: policy π_{θ_k}
$\theta_0 \leftarrow$ INITIAL-PARAMETERS()
for $k \in (0, 1, 2, ...)$ **do**
  $\mathcal{D} \leftarrow$ COLLECT-TRAJECTORIES()
  $g_k \leftarrow \frac{1}{|\mathcal{D}|} \sum_{\tau\in\mathcal{D}} \sum_{t=0}^{H} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)$ *{policy gradient}
  $\theta_{k+1} \leftarrow \theta_k + \alpha g_k$ **{gradient ascent step}
return π_{θ_k}

Equation $*$ is the estimation of the policy gradient over the set of trajectories $\mathcal{D}$. Statement $**$ – i.e. the gradient ascent update rule — can be substituted with the update rule of a different optimization algorithm, e.g. Adam.

Unfortunately the *unbiased policy gradient* g_k suffers from high variance which hinders performance and learning stability. This can be addressed through the use of baselines, terms that only depend on the current state and are subtracted from the reward. Equation 5 is the policy gradient updated with a generic baseline term.

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{H}\nabla_\theta \log \pi_\theta(a_t|s_t)\left(\sum_{t=0}^{H} R(s_t,a_t,s_{t+1}) - b(s_t)\right)\right] \tag{5}$$

4 The Job Shop optimization problem: notation

Scheduling is a decision-making process consisting in the allocation of resources to tasks over a given time period, with the additional constraint of optimizing one (or more) objective functions. The JSP is one of the most studied scheduling problems, along with the Open Shop and the Flow Shop Problems. A $n \times m$ JSP instance is characterized by:

- n *jobs* J_i, with $i \in \{0, ..., n-1\}$, each one consisting of m operations (or tasks) O_{ij}, with $j \in \{0, ..., m-1\}$.
- m *machines* M_{ij}, with $j \in \{0, ..., m-1\}$. M_{ij} identifies the machine required to execute the j-th operation of job i.

We denote the execution time of an operation O_{ij} with p_{ij}; an operation execution cannot be interrupted and each operation of a given job must be executed on a different machine. A JSP solution is represented by a schedule.

As an example, let us consider the JSP instance represented in Table 1. In this case there are three jobs J_i, with $i \in \{0, 1, 2\}$, and four operations O_{ij} for the i-th job, with $j \in \{0, 1, 2, 3\}$. Operation O_{ij} must be executed on machine $M_{ij} \in \{0, 1, 2, 3\}$ and has processing time p_{ij}.

M_{ij}, p_{ij}	O_{*0}	O_{*1}	O_{*2}	O_{*3}
J_0	(0, 4)	(2, 2)	(1, 6)	(3, 2)
J_1	(0, 4)	(3, 5)	(2, 7)	(1, 8)
J_2	(2, 6)	(0, 4)	(1, 3)	(3, 1)

Table 1. Example of a 3×4 JSP instance.

A useful tool for visualizing a schedule is Gantt charts [6]. Figure 1 represents the Gantt chart for a possible schedule of the JSP instance represented in Table 1.

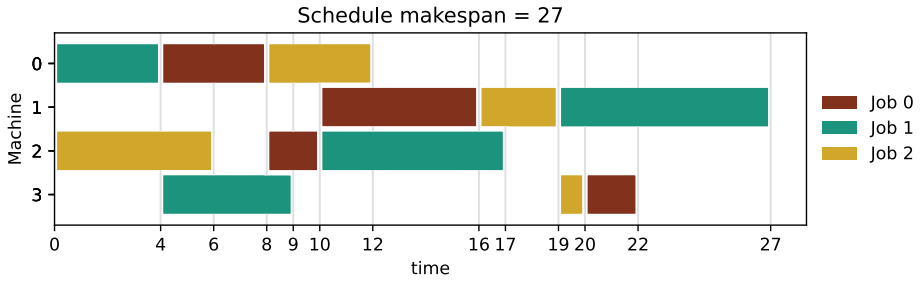

Fig. 1. One possible schedule for the JSP instance in Table 1.

The optimal solution of a JSP is the schedule that minimizes the makespan C_{max}, where $C_{max} = \max_i C_i$, and C_i is the completion time of the i-th job.

5 Our Sequence to Sequence approach to the JSP

The main novelty we present is a sequence-based Deep RL approach applied to the JSP. Inspired by [2] and [12] we make use of a deep neural network used for NLG applications and we train it in a RL setting. Such model (see Figure 2) combines a self-attention based encoder and a Pointer-Network decoder [21]. In order to apply it to the JSP, we formulate a sequence-based encoding of input and output, and design an appropriate masking mechanism to generate feasible solutions.

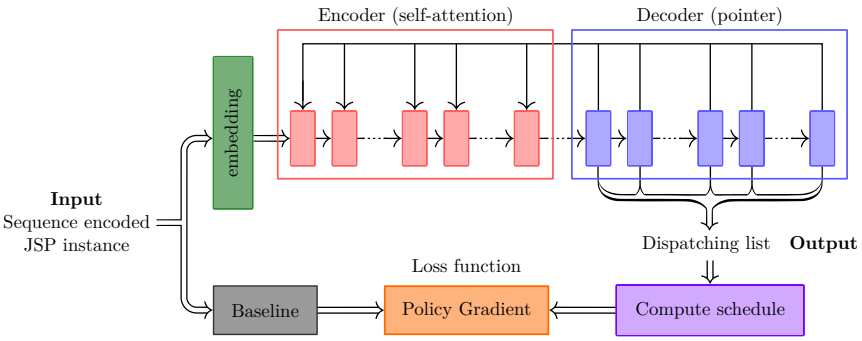

Fig. 2. Our encoder-decoder architecture for scheduling problems.

5.1 Sequence encoding

The input (i.e. problem instance) and the model's output (i.e. solution) need to be encoded as sequences in order for the model to process them correctly.

We consider both the input and the output as sequences of operations and we define a 4-dimensional feature vector $\mathbf{o}_k$ for each operation O_{ij} as follows:

$$\mathbf{o}_k = [i \;\; j \;\; M_{ij} \;\; p_{ij}] \qquad \text{with } k = m \cdot i + j \tag{6}$$

where i is the index of the i-th job and j the index of its j-th operation. Consider a JSP instance S with n jobs J_i ($i \in \{0, ..., n-1\}$) and m operations O_{ij} for job J_i ($j \in \{0, ..., m-1\}$) with required machine $M_{ij} \in \{0, ..., m-1\}$ and execution time p_{ij}. S can be expressed with the following sequence encoding S^{seq}:

$$S^{\text{seq}} = \begin{bmatrix} \mathbf{o}_0 \\ \mathbf{o}_1 \\ \vdots \\ \mathbf{o}_{m-1} \\ \mathbf{o}_m \\ \vdots \\ \mathbf{o}_{(m-1)(n-1)} \end{bmatrix} = \overbrace{\begin{bmatrix} 0 & 0 & M_{00} & p_{00} \\ 0 & 1 & M_{01} & p_{01} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & m-1 & M_{0m-1} & p_{0m-1} \\ 1 & 0 & M_{10} & p_{10} \\ \vdots & \vdots & \vdots & \vdots \\ n-1 & m-1 & M_{n-1m-1} & p_{n-1m-1} \end{bmatrix}}^{i \quad j \quad M_{ij} \quad p_{ij}} \tag{7}$$

The matrix S^{seq} just defined determines which jobs/operations have to be handled: establishing an order in which to execute them allows to identify a schedule.

Besides, a correct encoding of the model's output sequence implies that if for job J_i operation O_{ij} must be executed before O_{ik}, then the vectors of the operations in the output sequence must occur in the same order (not consecutive, in general). In order to comply with these requests, the sequence encoding for the output L^{seq} of the model has the following form:

$$L^{\text{seq}} = PS^{\text{seq}} = \left[\mathbf{o}'_0 \dots \mathbf{o}'_{(n-1)(m-1)}\right]^T \tag{8}$$

where P is a permutation matrix suitable for obtaining a matrix L^{seq} which encodes a feasible JSP solution.

The condition under which this occurs is explained in the following definition:

Definition 1 (Feasible sequence encoded JSP Solution).
Let O_{ij} and O_{ik} be the operations of the i-th job with $j < k$,
and $\mathbf{o}'_s = [i \;\; j \;\; M_{ij} \;\; p_{ij}]$, $\mathbf{o}'_r = [i \;\; k \;\; M_{ik} \;\; p_{ik}]$.
The matrix L^{seq} is the sequence encoding of a feasible schedule iff the permutation P is such that $s < r$, for all s and r in $\{0, ..., (n-1)(m-1)\}$ (i.e. the order of operations for job i defined in S^{seq} is preserved).

As an example the sequence encodings of the JSP instance in Table 1 are the following:

$$S^{\text{seq}} = \overbrace{\begin{bmatrix} 0\,0\,0\,4 \\ 0\,1\,2\,2 \\ 0\,2\,1\,6 \\ 0\,3\,3\,2 \\ 1\,0\,0\,4 \\ 1\,1\,3\,5 \\ 1\,2\,2\,7 \\ 1\,3\,1\,8 \\ 2\,0\,2\,6 \\ 2\,1\,0\,4 \\ 2\,2\,1\,3 \\ 2\,3\,3\,1 \end{bmatrix}}^{i \;\; j \;\; M_{ij} \;\; p_{ij}} \qquad L^{\text{seq}} = \overbrace{\begin{bmatrix} 1\,0\,0\,4 \\ 0\,0\,0\,4 \\ 2\,0\,2\,6 \\ 1\,1\,3\,5 \\ 0\,1\,2\,2 \\ 2\,1\,0\,4 \\ 0\,2\,1\,6 \\ 2\,2\,1\,3 \\ 1\,2\,2\,7 \\ 2\,3\,3\,1 \\ 0\,3\,3\,2 \\ 1\,3\,1\,8 \end{bmatrix}}^{i \;\; j \;\; M_{ij} \;\; p_{ij}}$$

The output L^{seq} can be effectively interpreted as a dispatching list, which can be directly mapped to a schedule as follows:

1. Considering $\mathbf{o}'_p = [i \;\; l \; M_{il} \; p_{il}]$ (p-th row of L^{seq}), schedule operation O_{il} to the earliest time such that machine M_{il} is available and, if $l > 0$ (i.e. O_{il} isn't the first operation of i-th job) the previous operation O_{il-1} has been executed.
2. Repeat for all the rows of L^{seq}.

Mapping L^{seq} to a JSP solution results in the schedule in Figure 1.

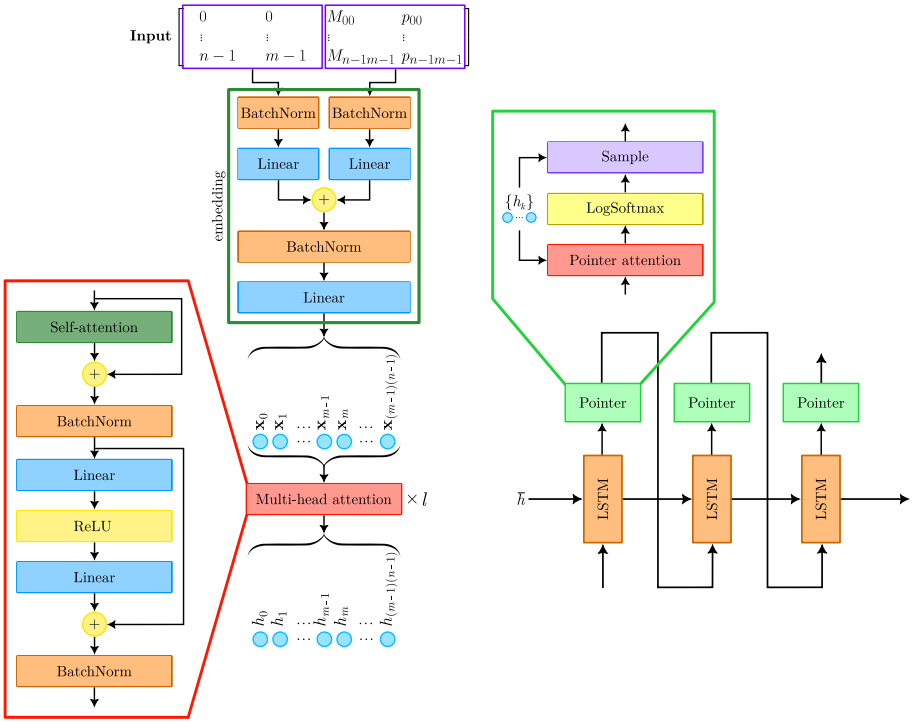

Fig. 3. *Left:* Encoder of our model. *Right:* Decoder with pointer mechanism.

5.2 Model architecture

Our model is composed of a self-attention-based encoder and a Pointer Network used as decoder (shown in Figure 3).

Encoder Represented in (Figure 3). The encoder's input is a 3-dimensional tensor $U \in \mathbb{R}^{N\times(nm)\times 4}$ that represents a batch of sequence-encoded instances. As defined in section 4, n and m are respectively the number of jobs and machines, and N indicates the batch size.

The first portion of the encoder computes two separate embeddings of each input row, respectively for features (i, j) and (M_{ij}, p_{ij}), by batch-normalizing and projecting to the embedding dimension d_h. After that, the sum of the two vectors is batch-normalized and passed through a linear layer resulting in $X \in \mathbb{R}^{N\times(nm)\times d_h}$. X is then fed into l multi-head attention layers (we consider $l = 3$).

The output of the encoder is a tensor $H \in \mathbb{R}^{N\times(nm)\times d_h}$ of embeddings $h_k \in \mathbb{R}^{d_h}$, later used as input in the decoder. $\overline{h}$, the average of the these embeddings, is used to initialize the decoder.

Decoder Represented in (Figure 3), the decoder is a Pointer Network which generates the policy π_θ, a distribution of probability over the rows of the input S^{seq},

via the attention mechanism; during training, the next selected row $\mathbf{o}'_t$ is sampled from it. During evaluation instead, the row with highest probability is selected in a greedy fashion. π_θ is defined as follows:

$$\pi_\theta(\mathbf{o}'_t|\mathbf{o}'_0, ..., \mathbf{o}'_{t-1}, S^{\text{seq}}) = \text{softmax}\left(\text{mask}(u^t|\mathbf{o}'_0, ..., \mathbf{o}'_{t-1})\right) \tag{9}$$

where u^t is the score computed by the Pointer Network's attention mechanism over S^{seq} input rows. $\text{mask}(u^t|\mathbf{o}'_0, ..., \mathbf{o}'_{t-1})$ is a masking mechanism which depends on the sequence partially generated and enforces the constraint in Definition 1.

Masking In order to implement the masking mechanism we use two boolean matrices M^{sched} and M^{mask} defined as follows:

Definition 2 (Boolean matrix M^{sched}). *Given the k-th instance in the batch and the j-th operation of the i-th job, the element M^{sched}_{kp} (which refers to $\mathbf{o}_p$, with $p = m \cdot i + j$) is true iff the j-th operation has already been scheduled.*

Definition 3 (Boolean matrix M^{mask}). *Given the k-th instance in the batch and the index l of an operation of the i-th job, the element M^{mask}_{kp} (which refers to $\mathbf{o}_p$, with $p = m \cdot i + l$) is true iff $l > j$, where j is the index of the next operation of the i-th job (i.e. scheduling the l-th operation would violate Definition 1).*

Given k-th instance in the batch and $\mathbf{o}_p$ feature vector of the operation scheduled at current time-step, we update M^{sched} and M^{mask} as follows.

$$M^{\text{sched}}_{kp} \leftarrow true, \qquad M^{\text{mask}}_{kp+1} \leftarrow false$$

At current step t, the resulting masking procedure of the score associated to input row index $p \in \{0, 1, ..., (m-1)(n-1)\}$ is the following:

$$\text{mask}(u^t_p|\mathbf{o}'_0, ..., \mathbf{o}'_{t-1}) = \begin{cases} -\infty, & \text{if } M^{\text{sched}}_{kp} \text{ OR } M^{\text{mask}}_{kp} \\ u^t_p, & \text{otherwise} \end{cases} \tag{10}$$

Masked scores result in a probability close to zero for operations that are already scheduled or cannot be scheduled. Figure 4 shows a possible generation procedure with the masking mechanism just described in order to solve the JSP instance represented in Table 2.

M_{ij}, p_{ij}	O_{*0}	O_{*1}	O_{*2}
J_0	(1, 4)	(2, 7)	(0, 5)
J_1	(0, 7)	(1, 3)	(2, 7)

Table 2. Example of a 2×3 JSP instance.

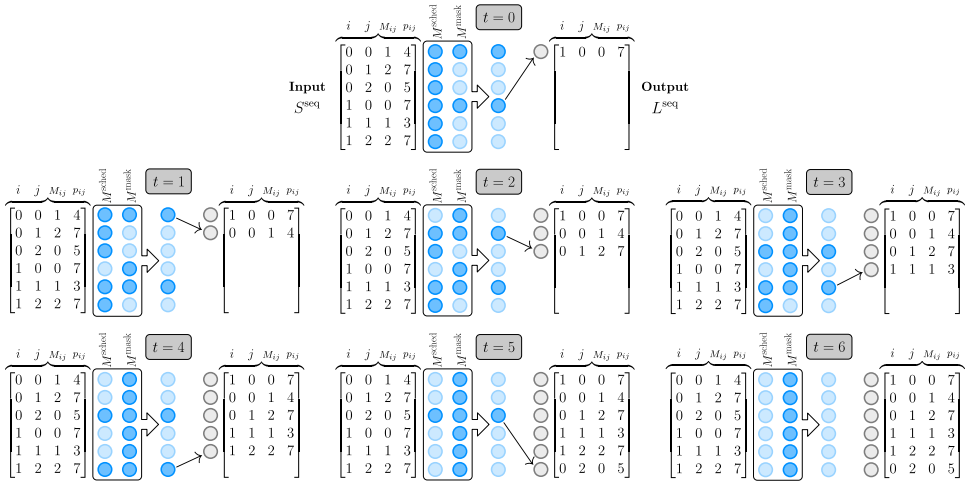

Fig. 4. Sequence generation with masking mechanism for the JSP. Light blue circles indicate masked rows and the arrows represent the agent's choices.

Training algorithm The network is trained with REINFORCE [23] described in subsection 3.1 using the Adam optimizer. We use the following form of the policy gradient:

$$\nabla_\theta L(\pi_\theta) = \mathbb{E}\left[(C_{max}(L^{\text{seq}}) - b(S^{\text{seq}}))\,\nabla_\theta \log P_\theta(L^{\text{seq}}|S^{\text{seq}})\right] \tag{11}$$

where $P_\theta(L^{\text{seq}}|S^{\text{seq}}) = \prod_{t=0}^{nm-1} \pi_\theta(\mathbf{o}'_t|\mathbf{o}'_0, ..., \mathbf{o}'_{t-1}, S^{\text{seq}})$ is the probability of the solution L^{seq} and $b(S^{\text{seq}})$ is the greedy rollout baseline. After each epoch, the algorithm updates the baseline with the optimized policy's weights if the latter is statistically better. This is determined by evaluating both policies on a 10000 samples dataset and running a paired t-test with $\alpha = 0.05$ (see [12] for the detailed explanation). The periodic update ensures that the policy is always challenged by the best model, hence the reinforcement of actions is effective. From a RL perspective, $-C_{max}(L^{\text{seq}})$ is the reward of the solution — lower makespan implies higher reward. After training, the active search approach [2] is applied.

Solving related scheduling problems Our method represents a general approach to scheduling problems and, once trained on JSP instances, it can also solve the Flow Shop Problem. The Open Shop Problem can also be solved with a small modification of the masking mechanism. Since the order constraint between operations is dropped in the OSP, the feasible outputs of the model are all the permutations of the input sequence. This simplifies the masking mechanism, which can be done just by keeping track of the scheduled operations with matrix M^{sched}. In Figure 5 we show three steps of the modified masking mechanism for solving the instance in Table 2, interpreted as an OSP.

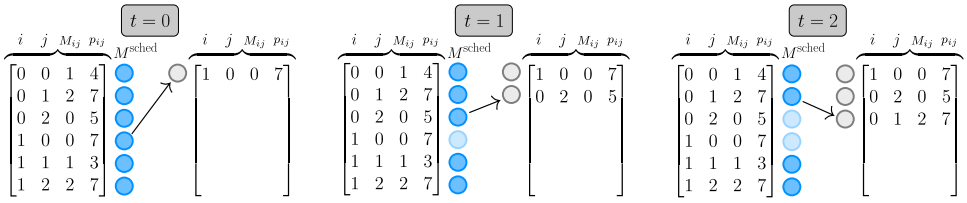

Fig. 5. Sequence generation with modified masking mechanism for the OSP.

5.3 Experiments and results

In this section we present our experiments and results. We consider four JSP settings: 6×6, 10×10, 15×15 and 30×20. After hyperparameter tuning, we set the learning rate to 10^{-5} and gradient clipping to 0.5 in order to stabilize training. At each epoch, the model processes a dataset generated with the well-known Taillard's method [19]. Table 3 sums up training configurations for every experiment.

During training we note the average cost every 50 batches and the validation performance at the end of every epoch. Validation rollouts are done in a greedy fashion, i.e. by choosing actions with maximum likelihood. Training and validation curves are represented in Figure 6.

Size	Epoch size	N° epochs	Batch size	GPU(s)*	Duration
6×6	640000	10	512	Titan RTX	30m
10×10	640000	10	512	RTX A6000	1h 30m
15×15	160000	10	256	RTX A6000	1h 30m
30×20	16000	10	32	Titan RTX	1h 45m

Table 3. Training configurations for all the experiments. * Nvidia GPUs have been used.

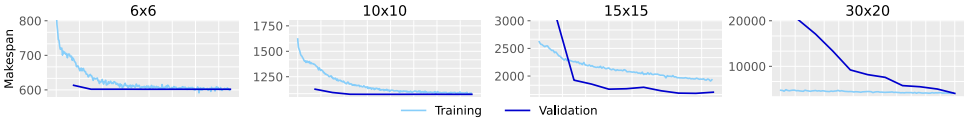

Fig. 6. Training and Validation curves for different JSPs.

Comparison with concurrent work As already said in the Introduction, we compare our results with the work from Zhang et al. [24], and with a set of largely used dispatching rules: *Shortest Processing Time* (SPT), *Most Work Remaining* (MWKR), *Most Operations Remaining* (MOPNR), *minimum ratio of Flow Due Date to most work remaining* (FDD).

Table 4 shows the testing results obtained applying our technique on 100 instances generated by Zhang et al. with the Taillard's method.

We compare each solution with the optimal one obtained with Google OR-Tools' [15] solver; in the last column we report the percentage of instances for which OR-Tools' returns optimal solutions in a limited computation time of 3600 seconds.

The column JSP settings shows the average makespan over the entire test dataset and the gap between $\overline{C}_{max}$ (the average makespan of heuristic solutions) and $\overline{C}^*_{max}$ (the average makespan of the optimal ones), defined as $\overline{C}_{max}/\overline{C}^*_{max} - 1$.

JSP settings		SPT	MWKR	FDD	MOPNR	Zhang [24]	Ours	Opt. Rate(%)
6×6	$\overline{C}_{max}$	691.95	656.95	604.64	630.19	574.09	**495.92**	100%
	Gap	42.0%	34.6%	24.0%	29.2%	17.7%	**1.7%**	
10×10	$\overline{C}_{max}$	1210.98	1151.41	1102.95	1101.08	988.58	**945.27**	100%
	Gap	50.0%	42.6%	36.6%	36.5%	22.3%	**16.9%**	
15×15	$\overline{C}_{max}$	1890.91	1812.13	1722.73	1693.33	**1504.79**	1535.14	99%
	Gap	59.2%	52.6%	45.1%	42.6%	**26.7%**	29.3%	
30×20	$\overline{C}_{max}$	3208.69	3080.11	2883.88	2809.62	**2508.27**	2683.05	12%
	Gap	65.3%	58.7%	48.6%	44.7%	**29.2%**	38.2%	

Table 4. Results over different JSP settings.

From Table 4 we can see that our model greatly outperforms the traditional dispatching rules even by a margin of 71% with respect to SPT. When compared to [24] our model is superior in performance in the 6×6 and 10×10 cases, while having similar results in the 15×15 JSPs, and sligthly underperforming in the 30×20. Speculating about the drop in performance of our solution in the biggest settings (i.e. 30×20 JSPs) we think it could be due to the following reasons:

- Larger JSP instances are encoded by longer sequences: like traditional RNNs and transformers, our model tends to have a suboptimal representation of the input if the sequence is exceedingly long.
- As mentioned before, for execution time reasons we reduce the number of instances and examples in each batch: this implies a gradient estimate with higher variance, hence a potentially unstable and longer learning.

Improving Active Search through Efficient Active Search Efficient Active Search (EAS) is a technique introduced in a recent work by Hottung et al. [10] that extends and substantially improves active search, achieving state-of-the-art performance on the TSP, CVRP and JSP. The authors proposed three different techniques, EAS-Emb, EAS-Lay and EAS-Tab, all based on the idea of performing active search while adjusting only a small subset of model parameters. EAS-Emb achieves the best performance and works by keeping all model parameters frozen while optimizing the embeddings. As pointed out in [10], this technique can be applied in parallel to a batch of instances, greatly reducing the computing time. In

this section we present a preliminary attempt to extend our method applying EAS-Emb and we test it on the 10x10 JSP. Table 5 shows that our model greatly benefits from the use of EAS-Emb, although underperforming Hottung et al.'s approach.

JSP settings		Hottung et al. [10]	Ours+EAS-Emb
10×10	$\overline{C}_{max}$	837.0	864.9
	Gap	3.7%	7.2%

Table 5. Results of EAS-Emb on the 10x10 JSP.

6 Conclusions

In this work we designed a Sequence to Sequence model to tackle the JSP, a famous Combinatorial Optimization problem, and we demonstrated that it is possible to train such architecture with a simple yet effective RL algorithm. Our system automatically learns dispatching rules and relies on a specific masking mechanism in order to generate valid schedulings. Furthermore, it is easy to generalize this mechanism for the Flow Shop Problem and the Open Shop Problem with none or slight modifications. Our solution beats all the main traditional dispatching rules by great margins and achieve better or state of the art performance on small JSP instances. For future works we plan to improve the performance of our method on larger JSP instances with EAS-based approaches. Although this work is mostly concerned with evaluating a Deep RL-based paradigm for combinatorial optimization, the idea of hybridizing these techniques with more classical heuristics remain viable.

Acknowledgements The activity has been partially carried on in the context of the Visiting Professor Program of the Gruppo Nazionale per il Calcolo Scientifico (GNCS) of the Italian Istituto Nazionale di Alta Matematica (INdAM).

References

1. Bellman, R.: A markovian decision process. Journal of mathematics and mechanics pp. 679–684 (1957)
2. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. In: International Conference on Learning Representations (2017)
3. Bengio, Y., Lodi, A., Prouvost, A.: Machine learning for combinatorial optimization: A methodological tour d'horizon. European Journal of Operational Research **290**(2), 405–421 (2021)
4. Cunha, B., Madureira, A.M., Fonseca, B., Coelho, D.: Deep reinforcement learning as a job shop scheduling solver: A literature review. In: International Conference on Hybrid Intelligent Systems. pp. 350–359. Springer (2018)
5. Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., Rousseau, L.M.: Learning heuristics for the tsp by policy gradient. In: International conference on the integration of constraint programming, artificial intelligence, and operations research. pp. 170–181. Springer (2018)
6. Gantt, H.: A Graphical Daily Balance in Manufacture. ASME (1903)

7. Glover, F., Laguna, M.: Tabu Search. Springer New York, NY (1998)
8. Han, B.A., Yang, J.J.: Research on adaptive job shop scheduling problems based on dueling double dqn. IEEE Access **8**, 186474–186495 (2020)
9. Haupt, R.: A survey of priority rule-based scheduling. Operations-Research-Spektrum **11**, 3–16 (1989)
10. Hottung, A., Kwon, Y.D., Tierney, K.: Efficient active search for combinatorial optimization problems. In: International Conference on Learning Representations
11. Jansen, K., Mastrolilli, M., Solis-Oba, R.: Approximation algorithms for flexible job shop problems. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000: Theoretical Informatics. pp. 68–77. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
12. Kool, W., van Hoof, H., Welling, M.: Attention, learn to solve routing problems! In: International Conference on Learning Representations (2019)
13. Lin, C.C., Deng, D.J., Chih, Y.L., Chiu, H.T.: Smart manufacturing scheduling with edge computing using multiclass deep q network. IEEE Transactions on Industrial Informatics **15**(7), 4276–4284 (2019)
14. Liu, C.L., Chang, C.C., Tseng, C.J.: Actor-critic deep reinforcement learning for solving job shop scheduling problems. Ieee Access **8**, 71752–71762 (2020)
15. Perron, L.: Operations research and constraint programming at google. In: International Conference on Principles and Practice of Constraint Programming. pp. 2–2. Springer (2011)
16. Sels, V., Gheysen, N., Vanhoucke, M.: A comparison of priority rules for the job shop scheduling problem under different flow time- and tardiness-related objective functions. International Journal of Production Research **50**(15), 4255–4270 (2012)
17. Smith, K.A.: Neural networks for combinatorial optimization: a review of more than a decade of research. Informs journal on Computing **11**(1), 15–34 (1999)
18. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
19. Taillard, E.: Benchmarks for basic scheduling problems. european journal of operational research **64**(2), 278–285 (1993)
20. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)
21. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. Advances in neural information processing systems **28** (2015)
22. Waschneck, B., Reichstaller, A., Belzner, L., Altenmüller, T., Bauernhansl, T., Knapp, A., Kyek, A.: Optimization of global production scheduling with deep reinforcement learning. Procedia Cirp **72**, 1264–1269 (2018)
23. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning **8**(3), 229–256 (1992)
24. Zhang, C., Song, W., Cao, Z., Zhang, J., Tan, P.S., Chi, X.: Learning to dispatch for job shop scheduling via deep reinforcement learning. Advances in Neural Information Processing Systems **33**, 1621–1632 (2020)