Fast Approximations for Job Shop Scheduling: A Lagrangian Dual Deep Learning Method

James Kotary,[1] **Ferdinando Fioretto,** [1] **Pascal Van Hentenryck** [2]

[1] Syracuse University
[2] Georgia Institute of Technology
jkotary@syr.edu, ffiorett@syr.edu, pvh@isye.gatech.edu

Abstract

The Jobs shop Scheduling Problem (JSP) is a canonical combinatorial optimization problem that is routinely solved for a variety of industrial purposes. It models the optimal scheduling of multiple sequences of tasks, each under a fixed order of operations, in which individual tasks require exclusive access to a predetermined resource for a specified processing time. The problem is NP-hard and computationally challenging even for medium-sized instances. Motivated by the increased stochasticity in production chains, this paper explores a deep learning approach to deliver efficient and accurate approximations to the JSP. In particular, this paper proposes the design of a deep neural network architecture to exploit the problem structure, its integration with Lagrangian duality to capture the problem constraints, and a post-processing optimization to guarantee solution feasibility. The resulting method, called *JSP-DNN*, is evaluated on hard JSP instances from the JSPLIB benchmark library. Computational results show that *JSP-DNN* can produce JSP approximations of high quality at negligible computational costs.

1 Introduction

The Job shop Scheduling Problem (JSP) is defined in terms of a set of jobs, each of which consists of a sequence of tasks. Each task is processed on a predetermined resource and no two tasks can overlap in time on these resources. The goal of the JSP is to sequence the tasks in order to minimize the total duration of the schedule. Although the problem is NP-hard and computationally challenging even for medium-sized instances, it constitutes a fundamental building block for the optimization of many industrial processes and is key to the stability of their operations. Its effects are profound in our society, with applications ranging from supply chains and logistics, to employees rostering, marketing campaigns, and manufacturing to name just a few (Kan 2012).

While the Artificial Intelligence and Operations Research communities have contributed fundamental advances in optimization in recent decades, the complexity of these problems often prevents them from being effectively adopted in contexts where many instances must be solved over a long-term horizon (e.g., multi-year planning studies) or when solutions must be produced under stringent time constraints. For example, when a malfunction occurs or when operating conditions require a new schedule, replanning needs to be executed promptly as machine idle time can be extremely costly. (e.g., on the order of $10,000 per minute for some applications (Gombolay et al. 2018)). To address this issue, system operators typically seek approximate solutions to the original scheduling problems. However, while more efficient computationally, their sub-optimality may induce substantial economical and societal losses, or they may even fail to satisfy important constraints.

Fortunately, in many practical settings, one is interested in solving many instances sharing similar patterns. Therefore, the application of deep learning methods to aid the resolution of these optimization problems is gaining traction in the nascent area at the intersection between constrained optimization and machine learning (Bengio, Lodi, and Prouvost 2020; Kotary et al. 2021; Vesselinova et al. 2020). In particular, supervised learning frameworks can train a model using pre-solved optimization instances and their solutions. However, while much of the recent progress at the intersection of constrained optimization and machine learning has focused on learning good approximations by jointly training prediction and optimization models (Balcan et al. 2018; Khalil et al. 2016; Nair et al. 2020; Nowak et al. 2018; Vinyals, Fortunato, and Jaitly 2015a) and incorporating optimization algorithms into differentiable systems (Amos and Kolter 2017a; Pogančić et al. 2020; Wilder, Dilkina, and Tambe 2019a; Mandi et al. 2020), learning the combinatorial structure of complex optimization problems remains a difficult task. In this context, the JSP is particularly challenging due to the presence of disjunctive constraints, which present some unique challenges for machine learning. Ignoring these constraints produce unreliable and unusable approximations, as illustrated in Section 5.

JSP instances typically vary along two main sets of parameters: **(1)** the continuous *task durations* and **(2)** the combinatorial *machine assignments* associated with each task. This work focuses on the former aspect, addressing the problem of learning to map JSP instances from a distribution over task durations to solution schedules which are close to optimal. Within this scope, the desired mapping is combinatorial in its structure: a marginal increase in one task duration can have cascading effects on the scheduling system, leading to

significant reordering of tasks between respective optimal schedules (Kan 2012).

To this end, this paper integrates Lagrangian duality within a Deep Learning framework to "enforce" constraints when learning job shop schedules. Its key idea is to exploit Lagrangian duality, which is widely used to obtain tight bounds in optimization, during the training cycle of a deep learning model. The paper also proposes a dedicated deep-learning architecture that exploits the structure of JSP problems and an efficient post-processing step to restore feasibility of the predicted schedules.

Contributions The contributions of this paper can be summarized as follows. **(1)** It proposes *JSP-DNN*, an approach that uses a deep neural network to accurately predict the tasks start times for the JSP. **(2)** *JSP-DNN* captures the JSP constraints using a Lagrangian framework, recasting the JSP prediction problem as the Lagrangian dual of the constrained learning task and using a subgradient method to obtain high-quality solutions. **(3)** It further exploits the JSP structure through the design of a bespoke network architecture that uses two dedicated sets of layers: *job layers* and *machine layers*. They reflect the task-precedence and no-overlapping structure of the JSP, respectively, encouraging the predictions to take account of these constraints. **(4)** While the adoption of Lagrangian duals and the dedicated JSP network architecture represent notable improvements to the prediction, the model predictions represent approximate solutions to the JSP and may not feasible. In order to derive feasible solutions from these predictions, this paper proposes an efficient reconstruction technique. **(5)** Finally, experiments against highly optimized industrial solvers show that JSP-DNN provides state-of-the-art JSP approximations, both in terms of accuracy and efficiency, on a variety of standard benchmarks. To the best of the authors' knowledge, this work is the first to tackle the predictions of JSPs using a dedicated supervised learning solution.

2 Related Work

The application of Deep Learning to constrained optimization problems is receiving increasing attention. Approaches which learn solutions to combinatorial optimization using neural networks include (Vinyals, Fortunato, and Jaitly 2015b; Khalil et al. 2017; Kool, Van Hoof, and Welling 2018). These approaches often rely on predicting permutations or combinations as sequences of pointers. Another line of work leverages explicit optimization algorithms as a differentiable layer into neural networks (Amos and Kolter 2017b; Donti, Amos, and Kolter 2017; Wilder, Dilkina, and Tambe 2019b). A further collection of works interpret constrained optimization as a two-player game, in which one player optimizes the objective function and a second player attempts at satisfying the problem constraints (Kearns et al. 2017; Narasimhan 2018; Agarwal et al. 2018). For instance, Agarwal et al. (2018) proposes a best-response algorithm applied to fair classification for a class of linear fairness constraints. To study generalization performance of training algorithms that learn to satisfy the problem constraints, Cotter et al. (2018) propose a two-players game approach in which one player optimizes the model parameters on the training data and the other player optimizes the constraints on a validation set. Arora, Hazan, and Kale (2012) propose the use of a multiplicative rule to maintain some properties by iteratively changing the weights of different distributions; they also discuss the applicability of the approach to a constraint satisfaction domain.

Different from these proposals, this paper proposes a framework that exploits key ideas from Lagrangian duality to encourage the satisfaction of generic constraints within a neural network learning cycle and apply them to solve complex JSP instances. This paper builds on the recent results that were dedicated to learning and optimization in power systems (Fioretto, Mak, and Van Hentenryck 2020).

Model 1: JSP Problem

$$\mathcal{P}(\boldsymbol{d}) = \operatorname{argmin}_{\boldsymbol{s}} \; u \tag{1}$$

$$\text{subject to: } u \geq s_T^j + d_T^j \quad \forall j \in [J] \tag{2a}$$

$$s_{t+1}^j \geq s_t^j + d_t^j \quad \forall j \in [J], \forall t \in [T-1] \tag{2b}$$

$$s_t^j \geq s_{t'}^{j'} + d_{t'}^{j'} \;\vee\; s_{t'}^{j'} \geq s_t^j + d_t^j \tag{2c}$$

$$\forall j, j' \in [J] : j \neq j', t, t' \in [T] \text{ with } \sigma_t^j = \sigma_{t'}^{j'}$$

$$s_t^j \in \mathbb{N} \quad \forall j \in [J], t \in [T] \tag{2d}$$

3 Preliminaries: JSP

The JSP is a combinatorial optimization problem in which J jobs, each composed of T tasks, must be processed on M machines. Each job comprises a sequence of T tasks, each of which is assigned to a different machine. Tasks within a job must be processed in their specified sequential order. Moreover, no two tasks may occupy the same machine at the same time. The objective is to find a schedule which minimizes the time to process all tasks, known as the *makespan*. This paper considers the classical setting in which the number of tasks in each job is equal to the number of machines ($T = M$), so that each job has one task assigned to each machine. This leads to a problem size $n = J \times M$. This is however not a limitation of the proposed work and its implementation generalizes beyond this setting.

The optimization problem associated with a JSP instance is described in Model 1, where σ_t^j denotes the machine that processes task t of job j, and d_t^j denotes the processing time on machine σ_t^j needed to complete task t of job j. The decision variables s_t^j and u represent, respectively, the start times of each task and the makespan. In the following, $\boldsymbol{d}$ and $\boldsymbol{s}$ denote, respectively, the input vector (processing times) and output vector (start times), and $\mathcal{P}(\boldsymbol{d})$ represents the optimal solution of a JSP instance with inputs $\boldsymbol{d}$. For simplicity, we assume that this solution is unique, i.e., there is a rule to break ties when there are multiple optimal solutions.

The *task-precedence* constraints (2b) require that all tasks be processed in the specified order; the *no-overlap* constraints (2c) require that no two tasks using the same machine overlap in time. The difficulty of the problem comes primarily from the disjunctive constraints (2c) defining the no-overlap condition. The JSP is, in general, NP-hard and can be for-

mulated in various ways, including several Mixed Integer Program (MIP), and Constraint Programming (CP) models, each having distinct characteristics (Ku and Beck 2016). In the following sections, $C(\boldsymbol{s}, \boldsymbol{d})$ denotes the set of constraints (2b)–(2d) associated with problem $\mathcal{P}(\boldsymbol{d})$.

4 JSP Learning Goals

Given the set of processing times $\boldsymbol{d} = (d_t^j)_{j\in[J],t\in[T]}$ associated with each problem task (as well as a static assignment of tasks to machines $\boldsymbol{\sigma}$), the paper develops a JSP mapping $\hat{\mathcal{P}} : \mathbb{N}^n \rightarrow \mathbb{N}^n$ to predict the start times $\boldsymbol{s} = (s_t^j)_{j\in[J],t\in[T]}$ for each task. The input of the learning task is a dataset $\mathcal{D} = \{(\boldsymbol{d}_i, \boldsymbol{s}_i)\}_{i=1}^N$, where $\boldsymbol{d}_i$ and $\boldsymbol{s}_i$ represent the i^{th} instance of task processing times and start times that satisfy $\boldsymbol{s}_i = \mathcal{P}(\boldsymbol{d}_i)$. The output is a JSP approximation function $\hat{\mathcal{P}}_{\boldsymbol{\theta}}$, parametrized by vector $\boldsymbol{\theta} \in \mathbb{R}^k$, that ideally would be the result of the following optimization problem

$$\underset{\boldsymbol{\theta}}{\text{minimize}} \quad \sum_{i=1}^{N} \mathcal{L}\left(\boldsymbol{s}_i, \hat{\mathcal{P}}_{\boldsymbol{\theta}}(\boldsymbol{d}_i)\right) \quad \text{subject to:} \quad C\left(\hat{\mathcal{P}}_{\boldsymbol{\theta}}(\boldsymbol{d}_i), \boldsymbol{d}_i\right),$$

whose loss function $\mathcal{L}$ captures the prediction accuracy of model $\hat{\mathcal{P}}_{\boldsymbol{\theta}}$ and $C(\hat{\boldsymbol{s}}, \boldsymbol{d})$ holds if the predicted start times $\hat{\boldsymbol{s}} = \hat{\mathcal{P}}_{\boldsymbol{\theta}}(\boldsymbol{d})$ produce a feasible solution to the JSP constraints.

One of the key difficulties of this learning task is the presence of the combinatorial feasibility constraints in the JSP. The approximation $\hat{\mathcal{P}}_{\boldsymbol{\theta}}$ will typically not satisfy the problem constraints, as shown in the next section. After exposing this challenge, this paper combines three techniques to obtain a feasible solution:

1. it learns predictions which are near-feasible using an augmented loss function;
2. it exploits the JSP structure through the design of a neural network architecture, and
3. it efficiently transforms these predictions into nearby solutions that satisfy the JSP constraints.

Combined, these techniques form a framework for predicting accurate and feasible JSP scheduling approximations.

5 The Baseline Model and its Challenges

This paper first presents the results of a baseline model whose approximation $\hat{\mathcal{P}}_{\boldsymbol{\theta}}$ is learned from a feed-forward ReLU network, named *FC*. The challenges of FC are illustrated in Figure 1, which reports the constraint violations (left) and the performance (right) of this baseline model compared to the proposed *JSP-DNN* model on the *swv11* JSP benchmark instance of size 50×10 (see Section 10 for details about the models and datasets). The left figure reports the non-overlap constraint violations measured as the relative task overlap with respect to the average processing time. While the baseline model often reports predictions whose corresponding makespans are close to the ground truth, the authors have observed that this model learns to "squeeze" schedules in order to reduce their makespans by allowing overlapping tasks. As a result this baseline model converges to solutions that violate the non-overlap constraint, often by very large amounts. The performance plot (right) shed additional light

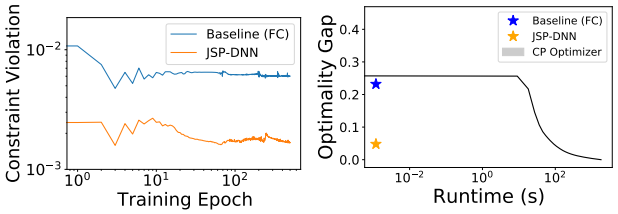

Figure 1: Constraint violations (left) and performance (right) of the baseline model (fully connected network trained with MSE loss) compared to the proposed JSP-DNN model. Benchmark: *swv11*. Constraint violation is the average magnitude of task overlap, measured with respect to the average processing time. The performance (right) reports the relative difference in makespan attained by the baseline and the JSP-DNN models, compared to an optimized version of IBM CP-optimizer over 30 minutes.

on the usability of this model in practice. It reports the time required by this baseline model to obtain a feasible solution (blue star) against the solution quality of a highly optimized solver (IBM CP-optimizer) over time. Obtaining feasible solutions efficiently is discussed in Section 9. The figures also report the constraint violations and solution quality found by the proposed *JSP-DNN* model (orange colors), which show dramatic improvements on both metrics. The next sections present the characteristics of *JSP-DNN*.

6 Capturing the JSP Constraints

To capture the JSP constraints within a deep learning model, a Lagrangian relaxation approach is used. Consider the optimization problem

$$\mathcal{P} = \underset{y}{\text{argmin}}\, f(y) \quad \text{subject to} \quad g_i(y) \leq 0 \quad (\forall i \in [m]). \tag{2}$$

Its *Lagrangian function* is expressed as

$$f_{\boldsymbol{\lambda}}(y) = f(y) + \sum_{i=1}^{m} \lambda_i \max(0, g_i(y)), \tag{3}$$

where the terms $\lambda_i \geq 0$ describe the Lagrangian multipliers, and $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_m)$ denotes the vector of all multipliers associated with the problem constraints. In this formulation, the expressions $\max(0, g_i(y))$ capture a quantification of the constraint violations, which are often exploited in augmented Lagrangian methods (Hestenes 1969) and constraint programming (Fontaine, Laurent, and Van Hentenryck 2014).

When using a Lagrangian function, the optimization problem becomes

$$LR_{\boldsymbol{\lambda}} = \underset{y}{\text{argmin}}\, f_{\boldsymbol{\lambda}}(y), \tag{4}$$

and it satisfies $f(LR_{\boldsymbol{\lambda}}) \leq f(\mathcal{P})$. That is, the Lagrangian function is a lower bound for the original function. Finally, to obtain the strongest Lagrangian relaxation of $\mathcal{P}$, the *Lagrangian dual* can be used to find the best Lagrangian multipliers,

$$LD = \text{argmax}_{\boldsymbol{\lambda}\geq 0}\, f(LR_{\boldsymbol{\lambda}}). \tag{5}$$

For various classes of problems, the Lagrangian dual is a strong approximation of $\mathcal{P}$. Moreover, its optimal solutions

can often be translated into high-quality feasible solutions by a post-processing step, which is the subject of Section 9.

Augmented Lagrangian of the JSP Constraints

Given an enumeration of the JSP constraints, the *violation degree* of constraint i is represented by $\max(0, g_i(\boldsymbol{y}))$. Given the predicted values $\hat{\boldsymbol{s}}$, the violation degrees associated with the JSP constraints are expressed as follows:

$$\nu_{2b}\left(\hat{s}_t^j\right) = \max\left(0, \hat{s}_t^j + d_t^j - \hat{s}_{t+1}^j\right) \tag{6a}$$

$$\nu_{2c}\left(\hat{s}_t^j, \hat{s}_{t'}^{j'}\right) = \min\left(\nu_{2c}^L\left(\hat{s}_t^j, \hat{s}_{t'}^{j'}\right), \nu_{2c}^R\left(\hat{s}_t^j, \hat{s}_{t'}^{j'}\right)\right), \tag{6b}$$

for the same indices as in Constraints (2b) and (2c) respectively, where

$$\nu_{2c}^L\left(\hat{s}_t^j, \hat{s}_{t'}^{j'}\right) = \max\left(0, \hat{s}_t^j + d_j^t - \hat{s}_{t'}^{j'}\right)$$
$$\nu_{2c}^R\left(\hat{s}_t^j, \hat{s}_{t'}^{j'}\right) = \max\left(0, \hat{s}_{t'}^{j'} + d_{j'}^{t'} - \hat{s}_t^j\right).$$

The term ν_{2b} refers to the task-precedence constraint (2b), and the violation degree ν_{2c} refers to the disjunctive non-overlap constraint (2c), with ν_{2c}^L and ν_{2c}^R referring to the two disjunctive components. When two tasks indexed (j,t) and (j',t') are scheduled on the same machine, if both disjunctions are violated, the overall violation degree ν_{2c} is considered to be the smaller of the two degrees, since this is the minimum distance that a task must be moved to restore feasibility.

The Learning Loss Function

The loss function of the learning model used to train the proposed JSP-DNN can now be augmented with the Lagrangian terms and expressed as follows:

$$\mathcal{L}(\boldsymbol{s}, \hat{\boldsymbol{s}}, \boldsymbol{d}) = \mathcal{L}(\boldsymbol{s}, \hat{\boldsymbol{s}}) + \sum_{c \in C} \lambda_c \nu_c(\hat{\boldsymbol{s}}, \boldsymbol{d}). \tag{7}$$

It minimizes the prediction loss–defined as mean squared error between the optimal start times $\boldsymbol{s}$ and the predicted ones $\hat{\boldsymbol{s}}$–and it includes the Lagrangian relaxation based on the violation degrees ν of the JSP constraints $c \in C$.

7 The Learning Model

Let $\hat{\mathcal{P}}_{\boldsymbol{\theta}}$ be the resulting *JSP-DNN* with parameters $\boldsymbol{\theta}$ and let $\mathcal{L}[\boldsymbol{\lambda}]$ be the loss function parametrized by the Lagrangian multipliers $\boldsymbol{\lambda} = \{\lambda_c\}_{c \in C}$. The core training aims at finding the weights $\boldsymbol{\theta}$ that minimize the loss function for a given set of Lagrangian multipliers, i.e., it computes

$$LR_{\boldsymbol{\lambda}} = \min_{\boldsymbol{\theta}} \mathcal{L}[\boldsymbol{\lambda}]\left(\boldsymbol{s}, \hat{\mathcal{P}}_{\boldsymbol{\theta}}(\boldsymbol{d})\right).$$

In addition, *JSP-DNN* exploits Lagrangian duality to obtain the optimal Lagrangian multipliers, i.e., it solves

$$LD = \max_{\boldsymbol{\lambda}} LR_{\boldsymbol{\lambda}}.$$

The Lagrangian dual is solved through a subgradient method that computes a sequence of multipliers $\boldsymbol{\lambda}^1, \ldots, \boldsymbol{\lambda}^k, \ldots$ by solving a sequence of trainings $LR_{\boldsymbol{\lambda}}^0, \ldots, LR_{\boldsymbol{\lambda}}^{k-1}, \ldots$ and adjusting the multipliers using the violations,

$$\boldsymbol{\theta}^{k+1} = \operatorname*{argmin}_{\boldsymbol{\theta}} \mathcal{L}[\boldsymbol{\lambda}^k]\left(\boldsymbol{s}, \hat{\mathcal{P}}_{\boldsymbol{\theta}^k}(\boldsymbol{s})\right) \tag{L1}$$

$$\boldsymbol{\lambda}^{k+1} = \left(\lambda_c^k + \rho\, \nu_c\left(\hat{\mathcal{P}}_{\boldsymbol{\theta}^{k+1}}(\boldsymbol{s}), \boldsymbol{d}\right) \mid c \in C\right), \tag{L2}$$

Algorithm 1: *Learning Step*

input : $\mathcal{D}, \alpha, \rho$: Training data, Optimizer, and Lagrangian step sizes, resp.

1 $\boldsymbol{\lambda}^0 \leftarrow 0 \;\; \forall c \in C$
2 **for** *epoch* $k = 0, 1, \ldots$ **do**
3 **foreach** $(\boldsymbol{d}, \boldsymbol{s}) \leftarrow$ *minibatch*$(\mathcal{D})$ *of size* b **do**
4 $\hat{\boldsymbol{s}} \leftarrow \hat{\mathcal{P}}_{\boldsymbol{\theta}}(\boldsymbol{d})$
5 $\mathcal{L}(\boldsymbol{s}, \hat{\boldsymbol{s}}) \leftarrow \frac{1}{b} \sum_{i \in [b]} \mathcal{L}(\boldsymbol{s}_i, \hat{\boldsymbol{s}}_i) + \sum_{c \in C} \lambda_c^k \nu_c(\boldsymbol{d}_i, \hat{\boldsymbol{s}}_i)$
6 $\theta \leftarrow \theta - \alpha \nabla_\theta (\mathcal{L}(\boldsymbol{s}, \hat{\boldsymbol{s}}))$
7 **foreach** $c \in C$ **do**
8 $\lambda_c^{k+1} \leftarrow \lambda^k + \rho \nu_c(\boldsymbol{d}, \hat{\boldsymbol{s}})$

where $\rho > 0$ is the Lagrangian step size. In the implementation, step (L1) is approximated using a Stochastic Gradient Descent (SGD) method. Importantly, this step does not recompute the training from scratch but uses a warm start for the model parameters $\boldsymbol{\theta}$.

The overall training scheme is presented in Algorithm 1. It takes as input the training dataset $\mathcal{D}$, the optimizer step size $\alpha > 0$ and the Lagrangian step size $\rho > 0$. The Lagrangian multipliers are initialized in line 1. The training is performed for a fixed number of epochs, and each epoch optimizes the weights using a minibatch of size b. After predicting the task start times (line 4), it computes the objective and constraint losses (line 5). The latter uses the Lagrangian multipliers $\boldsymbol{\lambda}^k$ associated with current epoch k. The model weights θ are updated in line 6. Finally, after each epoch, the Lagrangian multipliers are updated following step (L2) described above (lines 7 and 8).

8 The JM-structured Network Architecture

This section describes the bespoke neural network architecture that exploits the JSP structure. Let $\mathcal{I}_k^{(m)}$ and $\mathcal{I}_k^{(j)}$ denote, respectively, the set of task indexes associated with the k^{th} machine and the k^{th} job. Further, denote with $\boldsymbol{d}[\mathcal{I}]$ the set of processing times associated with the tasks identified by index set $\mathcal{I}$. The JSP-structured architecture, called JM as for Jobs-Machines, is outlined in Figure 2. The network differentiates three types of layers: *Job layers*, that process processing times organized by jobs, *Machine layers*, that process processing times organized by machines, and *Shared layers*, that process the outputs of the job layers and the machine layers to return a prediction $\hat{\boldsymbol{s}}$.

The input layers are depicted with white rectangles. Each job layer k takes as input the task processing times $\boldsymbol{d}[\mathcal{I}_k^{(j)}]$ associated with an index set $\mathcal{I}_k^{(j)}$ ($k \in [J]$), and each machine layer k takes as input the task processing times $\boldsymbol{d}[\mathcal{I}_k^{(m)}]$ associated with an index set $\mathcal{I}_k^{(m)}$ ($k \in [J]$). The shared layers combine the latent outputs of the job and machine layers. A decoder-encoder pattern follows for each individual group of layers, which are fully connected and use ReLU activation functions (additional details are reported in Section 10).

The effect of the JM architecture is twofold. First, the patterns created by the job layers and the machine layers

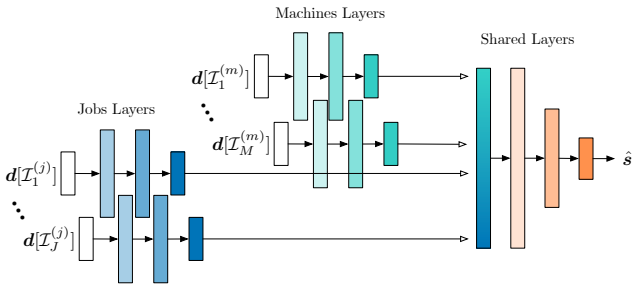

Figure 2: The JM Network Architecture of *JSP-DNN*.

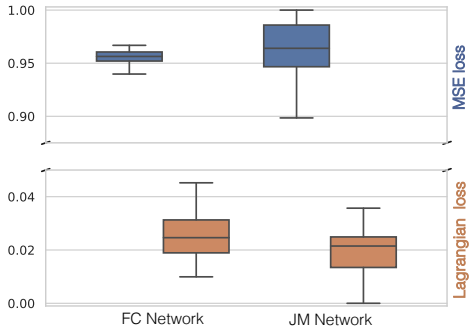

Figure 3: Distribution of No-Overlap Violation results corresponding to the hyperparameter search of Table 1. Benchmark: ta30. Lagrangian dual results (Orange) are compared against a baseline using MSE loss (Blue). JM network architecture is compared against a baseline fully connected network (FC). Scaled between 0 (min) and 1 (max).

reflect the task-precedence and no-overlap structure of the JSP, respectively, encouraging the prediction to account for these constraints. Second, in comparison to an architecture that takes as input the full vector $\boldsymbol{d}$ and processes it using fully connected ReLU layers, the proposed structure reduces the number of trainable parameters, as no connection is created within any two hidden layers processing tasks on different machines (machine layers) or jobs (job layers). This allows for faster convergence to high-quality predictions.

Figure 3 compares the no-overlap constraint violations resulting from each choice of loss function and network architecture over the whole range of hyperparameters searched. The figure clearly illustrates the benefits of using the Lagrangian dual method, and that the best results are obtained when the Lagrangian dual method is used in conjunction with the JM architecture. The results for task precedence constraints are analogous.

9 Constructing Feasible Solutions

It remains to show how to recover a feasible solution from the prediction. The recovery method exploits the fact that, given a task ordering on each machine, it is possible to construct a feasible schedule in low polynomial time. Moreover, a prediction $\hat{\boldsymbol{s}}$ defines implicitly a task ordering. Indeed, using (j,t) to denote task t of job j, a start time prediction vector

Model 2: Recovering a Feasible Solution to the JSP.

$$\Pi(\boldsymbol{s}) = \text{argmin}_{\boldsymbol{s}} \;\; u$$

subject to: (2a), (2b)

$$s_t^j \geq s_{t'}^{j'} + d_{t'}^{j'} \quad \forall j,j' \in [J], t,t' \in [T] \text{ s.t. } (j,t) \preceq_{\hat{\boldsymbol{s}}} (j',t') \tag{8a}$$

$$s_t^j \geq 0 \quad \forall j \in [J], t \in [T] \tag{8b}$$

Algorithm 2: *The JSP Greedy Recovery.*

Input: $\{\hat{s}_t^j\}_{t\in[T],j\in[J]}$: predicted start times

1 $Q \leftarrow$ enqueue$((j,1))$, $\forall j \in [J]$
2 **while** Q *is empty* **do**
3 $(j,t) \leftarrow$ dequeue(Q)
4 Schedule task (j,t) with start time $\hat{s}_t^j$
5 **if** $t < T$ **then**
6 $\hat{s}_{t+1}^j \leftarrow \max(\hat{s}_{t+1}^j, \hat{s}_t^j + d_t^j)$
7 $Q \leftarrow$ enqueue(j,t)
8 **end**
9 **foreach** (t',j') *s.t.* $t \neq t'$ & $\sigma_{t'}^{j'} = \sigma_t^j$ & $\hat{s}_{t'}^{j'} \geq \hat{s}_t^j$ **do**
10 $\hat{s}_{t'}^{j'} \leftarrow \max(\hat{s}_{t'}^{j'}, \hat{s}_t^j + d_t^j)$
11 **end**
12 **end**

$\hat{\boldsymbol{s}}$ can be used to define a task ordering $\preceq$ between tasks executing on the same machine, i.e.,

$$(j,t) \preceq_{\hat{\boldsymbol{s}}} (j',t') \;\text{ iff }\; \hat{\mu}_t^j \leq \hat{\mu}_{t'}^{j'} \;\wedge\; \sigma_t^j = \sigma_{t'}^{j'},$$

where $\hat{\mu}_t^j = \frac{\hat{s}_t^j + d_t^j}{2}$ is the predicted midpoint of a task t of job j.

The Linear Program (LP) described in Model 2 computes the optimal schedule subject to the ordering $\preceq_{\hat{\boldsymbol{s}}}$ associated with prediction $\hat{\boldsymbol{s}}$. This LP has the same objective as the original scheduling problem, along with the upper bound on start times and task-precedence constraints. The disjunctive constraints of the JSP however are replaced by additional precedence constraints (8a) and (8b) for the ordering on each machine. The problem is totally unimodular when the durations are integral.

Note that the above LP may be infeasible: the machine precedence constraints may not be consistent with the job precedence constraints. If this happens, it is possible to use a greedy recovery algorithm that selects the next task to schedule based on its predicted starting time and updates predictions with precedence and machine constraints. The greedy procedure is illustrated in Algorithm 2 and is organized around a priority queue which is initialized with the first task in each job. Each iteration selects the task with the smallest starting time $\hat{s}_t^j$ and updates the starting of its job successor (line 6) and the starting times of the unscheduled task using the same machine (line 10). It is important to note that this greedy procedure was never needed: *all test cases* (e.g., all instances under all hyper-parameter combinations) induced machine orderings that were consistent with the job precedence constraints.

Parameter	Min Value	Max Value
Learning Rate α	0.000125	0.02
Dual Learning Rate ρ	0.001	0.05
# Machine Layers	2	2
# Job Layers	2	2
# Shared Layers	2	2
Machine Layer Size	$2J$	$2J$
Job Layer Size	$2M$	$2M$
Shared Layer Size	$2JM$	$2JM$

Table 1: Model Parameters and Hyper-Parameters.

Remark on Learning Task Orderings *JSP-DNN* learns to predict schedules as assignments of start times to each task. Another option would have been to learn machine orderings directly in the form of permutations since, once machine orderings are available, it is easy to recover start times. However, learning permutations was found to be much more challenging due to its less efficient representation of solutions. For example, for a 50×10 JSP, predicting task ordering (or, equivalently, rankings) requires an output dimension of $50 \times 50 \times 10$ (a one-hot encoding to rank each of the 50 tasks and 10 machines). In contrast, the start time prediction requires only one value for each start-time, in this case, 50×10. Enforcing feasibility also becomes more challenging in the case of rankings. It is nontrivial to design successful violation penalties for task precedence constraints when using the one-hot encoding representation of the solutions.

Start times are easier to predict and indirectly produce good proxies for machine orderings.

10 Experimental Results

The experiments evaluate *JSP-DNN* against the baseline FC network, the state-of-the-art CP commercial solver IBM CP-Optimizer, and several well-known scheduling heuristics.

Data Generation and Model Details A single instance of the JSP with J jobs and M machines is defined by a set of assignments of tasks to machines, along with processing times required by each task. The generation of JSP instances simulates a situation in which a scheduling system experiences an unexpected "slowdown" on some arbitrary machine, inducing an increase in the processing times of each task assigned to the impaired machine. To create each experimental dataset, a root instance is chosen from the JSPLIB repository (JSPLib 2014), and a set of 5000 individual problem instances are generated accordingly, with all processing times on the slowdown machines extended from their original value to a maximum increase of 50 percent. Each such instance is solved using the IBM CP-Optimizer constraint-programming software (Laborie 2009) with a time limit of 1800s. The instance and its solution are included in the training dataset.

Model Configuration To ensure a fair analysis, the learning models have the same number of trainable parameters. The JM-structured neural networks are given two decoder layers per job and machine, whose sizes are twice as large as their corresponding numbers of tasks. These decoders are connected to a single *Shared Layer* (of size $2 \times J \times M$). A final output layer, representing task start times, has size equal to the number of tasks in the JSP instance. The baseline fully connected networks are given 3 hidden layers, whose sizes are chosen to match the size of the corresponding *JSP-DNN* network in terms of the total number of parameters. The experiments apply the recovery operators discussed in the previous section to both the FC and *JSP-DNN* models to obtain feasible solutions from their predictions. Time required for training and for running the models are reported in Appendix A.

The learning models are configured by a hyper-parameter search for the values summarized in Table 1. The search samples evenly spaced learning rates between their minimum and maximum values, along with similarly chosen dual learning rates ρ when the Lagrangian loss function is used. Additionally, each configuration is evaluated with 5 distinct random seeds, which primarily influences the neural network parameter initialization. The model with the best predicted schedules in average is selected for the experimental evaluation.

Model Accuracy and Constraint Violations

Table 2 represents the performance of the selected models. Each reported performance metric is averaged over the entire test set. For each prediction metric, symbols $\downarrow$ and $\uparrow$ indicate whether lower or higher values are better, respectively. The evaluation uses a 80:20 split with the training data.

• **Prediction Error** $\downarrow$ The columns for prediction errors report the error between a predicted schedule and its target solution (before recovering a feasible solution). It is measured as the L1 distance between respective start times. The predictions by *JSP-DNN* are much closer to the targets than those of the FC model. *JSP-DNN* reduces the errors by up to an order of magnitude compared to FC, demonstrating the ability of the *JM* architecture and the Lagrangian dual method to exploit the problem structure to derive good predictions.

• **Constraint Violation** $\downarrow$ The constraint violations are collected before recovering a feasible solution and the columns report the average magnitude of overlap between two tasks as a fraction of the average processing time. The results show again that the violation magnitudes reported by *JSP-DNN* are typically one order of magnitude lower than those reported by FC. They highlight the effectiveness of the Lagrangian dual approach to take into account the problem constraints.

• **Optimality Gap** $\downarrow$ The quality of the predictions is measured as the average relative difference between the makespan of the *feasible solutions* recovered from the predictions of the deep-learning models, and the makespan obtained by the IBM CP-Optimizer with a timeout limit of 1800 seconds. The optimality gap is the primary measure of solution quality, as the goal is to predict solutions to JSP instances that are as close to optimal as possible. The table also reports the optimality gaps achieved by several (fast) heuristics, relative to the same CP-Optimizer baseline. They are Shortest Processing Time (SPT), Least Work Remaining (LWR), Most Work Remaining (MWR), Least Operations Remaining (LOR), and Most Operations Remaining (MOR). Since the CP solver cannot typically find optimal solutions within the given timeout, the results under-approximate the true optimality gaps.

Instance	Size	Prediction Err(×10) ↓		Constraint Viol(×10^2) ↓		Opt. Gap Heuristics (%) ↓					Opt. Gap DNNs (%) ↓		Time SoTA Eq. (s) ↑	
	$J \times M$	FC	JSP-DNN	FC	JSP-DNN	SPT	LWR	MWR	LOR	MOR	FC	JSP-DNN	FC	JSP-DNN
yn02	20×20	2.770	**0.138**	1.134	**0.122**	628	837	40	934	40	12.80 ± 5.4	**-0.045 ± 0.9**	10.20	**1800+**
ta25	20×20	1.607	**0.361**	0.631	**0.244**	593	877	59	787	46	13.61 ± 3.13	**-0.143 ± 0.8**	11.02	**1800+**
ta30	30×15	4.338	**1.196**	1.483	**0.357**	558	910	63	856	46	15.01 ± 2.63	**-0.48 ± 5.18**	9.06	**1800+**
ta40	30×20	7.880	**3.341**	1.863	**0.104**	492	794	57	836	25	23.11 ± 7.33	**3.19 ± 1.88**	8.40	**12.04**
ta50	50×10	4.580	**1.322**	1.223	**0.225**	789	789	53	1116	43	18.30 ± 5.22	**5.85± 2.72**	8.02	**90.30**
swv03	20×15	9.473	**2.683**	2.777	**0.850**	203	212	75	190	50	28.61 ± 14.27	**7.62 ± 2.51**	4.04	**36.36**
swv05	20×10	6.586	**2.950**	2.325	**0.626**	183	192	80	177	66	20.78 ± 10.54	**6.34 ± 1.82**	7.24	**18.18**
swv07	20×10	4.587	**0.681**	1.222	**0.223**	299	295	68	352	43	10.69 ± 6.83	**0.01± 4.75**	26.0	**254.5**
swv09	20×15	5.678	**3.462**	2.132	**0.211**	322	270	69	285	75	22.12 ± 8.52	**5.42 ± 1.21**	6.48	**28.32**
swv11	50×10	7.958	**3.244**	2.711	**0.282**	237	231	94	263	73	23.18 ± 2.27	**4.80 ± 4.47**	7.02	**92.00**
swv13	50×10	23.21	**3.557**	1.615	**0.323**	225	203	114	218	79	22.79 ± 16.21	**8.11 ± 4.20**	7.08	**24.08**

Table 2: Accuracy metrics compared between FC and JSP-DNN (left sub-table) and accuracy of simple heuristics vs CP-Optimizer at 1800s (right sub-table). Best results shown in bold.

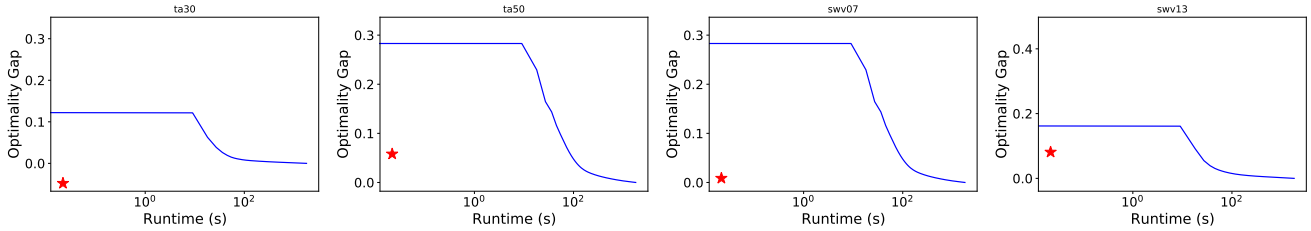

Figure 4: Comparison of mean timing and accuracy of four *JSP-DNN* models (red) against CP-optimizer mean optima and standard deviation (blue).

The results show that the proposed *JSP-DNN* substantially outperforms all other heuristic baselines and the best FC network of equal capacity. On these difficult instances, the most accurate heuristics exhibit results whose relative errors are at least an order of magnitude larger than those of *JSP-DNN*. Similarly, *JSP-DNN* significantly outperforms FC.

It is interesting to observe that, for instances *yn02, ta25*, and *ta30*, the makespan of the recovered *JSP-DNN* solutions outperform, on average, those reported by the CP solver before it reaches its time limit, which is quite remarkable. The CP solver can actually produce these solutions as well if it is allowed to run for several hours. This opens an interesting avenue beyond for further research: the use of *JSP-DNN* to hot-start a CP solver.

Comparison with SoTA Solver ↑

The last results compare *JSP-DNN* and the state-of-the art CP solver in terms of their ability to find high-quality solutions quickly. The runtime of *JSP-DNN* is dominated by the solution recovery process (Model 2 and Algorithm 2): their runtimes depend on the instance size but never exceed 30ms for the test cases.

The results are depicted in the last two columns of Table 2. They report the average runtimes required by the CP solver to produce solutions that match or outperform the feasible (recovered) solutions produced by *JSP-DNN* and FC. The results show that it takes the CP solver less than 12 seconds to outperform FC. In contrast, the CP solver takes at least an order of magnitude longer to outperform *JSP-DNN* and is not able to do so within 30 minutes on the first 3 test cases.

Figure 4 complements these results on three test cases: they depict the evolution of the makespan produced by the CP solver over time and contrast these with the solution recovered by *JSP-DNN*. The thick line depicts the mean makespan, the shaded region captures its standard deviation, and the red star reports the quality of the *JSP-DNN* solution. These results highlight the ability of *JSP-DNN* to generate a high-quality solution quickly.

Overall, these results show that JSP-DNN may be a useful addition to the set of optimization tools for scheduling applications that require high-quality solutions in real time. It may also provide an interesting avenue to seed optimization solvers, as mentioned earlier.

11 Conclusions

This paper proposed *JSP-DNN*, a deep-learning approach to produce high-quality approximations of Job shop Scheduling Problems (JSPs) in milliseconds. The proposed approach combines deep learning and Lagrangian duality to model the combinatorial constraints of the JSP. It further exploits the JSP structure through the design of dedicated neural network architectures to reflect the nature of the task-precedence and no-overlap structure of the JSP, encouraging the predictions to take account of these constraints. The paper also presented efficient recovery techniques to post-process *JSP-DNN* predictions and produce feasible solutions. The experimental analysis showed that *JSP-DNN* produces feasible solutions whose quality is at least an order of magnitude better than commonly used heuristics. Moreover, a state-of-the-art commercial CP solver was shown to take a significant amount of time to obtain solutions of the same quality and may not be able to do so within 30 minutes on some test cases.

References

Agarwal, A.; Beygelzimer, A.; Dudík, M.; Langford, J.; and Wallach, H. 2018. A reductions approach to fair classification. *arXiv preprint arXiv:1803.02453*.

Amos, B.; and Kolter, J. Z. 2017a. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, 136–145. PMLR.

Amos, B.; and Kolter, J. Z. 2017b. Optnet: Differentiable optimization as a layer in neural networks. In *ICML*, 136–145. JMLR. org.

Arora, S.; Hazan, E.; and Kale, S. 2012. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1): 121–164.

Balcan, M.-F.; Dick, T.; Sandholm, T.; and Vitercik, E. 2018. Learning to branch. In *International conference on machine learning*, 344–353. PMLR.

Bengio, Y.; Lodi, A.; and Prouvost, A. 2020. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*.

Cotter, A.; Gupta, M.; Jiang, H.; Srebro, N.; Sridharan, K.; Wang, S.; Woodworth, B.; and You, S. 2018. Training well-generalizing classifiers for fairness metrics and other data-dependent constraints. *arXiv preprint arXiv:1807.00028*.

Donti, P.; Amos, B.; and Kolter, J. Z. 2017. Task-based end-to-end model learning in stochastic optimization. In *NIPS*, 5484–5494.

Fioretto, F.; Mak, T.; and Van Hentenryck, P. 2020. Predicting AC Optimal Power Flows: Combining Deep Learning and Lagrangian Dual Methods. In *AAAI*.

Fontaine, D.; Laurent, M.; and Van Hentenryck, P. 2014. Constraint-Based Lagrangian Relaxation. In *Principles and Practice of Constraint Programming*, 324–339.

Gombolay, M.; Jensen, R.; Stigile, J.; Golen, T.; Shah, N.; Son, S.-H.; and Shah, J. 2018. Human-machine collaborative optimization via apprenticeship scheduling. *Journal of Artificial Intelligence Research*, 63: 1–49.

Hestenes, M. R. 1969. Multiplier and gradient methods. *Journal of optimization theory and applications*, 4(5): 303–320.

JSPLib. 2014. JSPLIB: Benchmark Instances for Job-Shop Scheduling Problem.

Kan, A. R. 2012. *Machine scheduling problems: classification, complexity and computations*. Springer Science & Business Media.

Kearns, M.; Neel, S.; Roth, A.; and Wu, Z. S. 2017. Preventing fairness gerrymandering: Auditing and learning for subgroup fairness. *arXiv:1711.05144*.

Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. In *NIPS*, 6348–6358.

Khalil, E.; Le Bodic, P.; Song, L.; Nemhauser, G.; and Dilkina, B. 2016. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30.

Kool, W.; Van Hoof, H.; and Welling, M. 2018. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*.

Kotary, J.; Fioretto, F.; Van Hentenryck, P.; and Wilder, B. 2021. End-to-End Constrained Optimization Learning: A Survey. *arXiv preprint arXiv:2103.16378*.

Ku, W.-Y.; and Beck, J. C. 2016. Mixed integer programming models for job shop scheduling: A computational analysis. *Computers & Operations Research*, 73: 165–173.

Laborie, P. 2009. IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, 148–162. Springer.

Mandi, J.; Stuckey, P. J.; Guns, T.; et al. 2020. Smart predict-and-optimize for hard combinatorial optimization problems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 34, 1603–1610.

Nair, V.; Bartunov, S.; Gimeno, F.; von Glehn, I.; Lichocki, P.; Lobov, I.; O'Donoghue, B.; Sonnerat, N.; Tjandraatmadja, C.; Wang, P.; et al. 2020. Solving Mixed Integer Programs Using Neural Networks. *arXiv preprint arXiv:2012.13349*.

Narasimhan, H. 2018. Learning with complex loss functions and constraints. In *International Conference on Artificial Intelligence and Statistics*, 1646–1654.

Nowak, A.; Villar, S.; Bandeira, A. S.; and Bruna, J. 2018. Revised Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks.

Pogančić, M. V.; Paulus, A.; Musil, V.; Martius, G.; and Rolinek, M. 2020. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations (ICLR)*.

Vesselinova, N.; Steinert, R.; Perez-Ramirez, D. F.; and Boman, M. 2020. Learning Combinatorial Optimization on Graphs: A Survey With Applications to Networking. *IEEE Access*, 8: 120388–120416.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015a. Pointer Networks. In *Advances in Neural Information Processing Systems (NeurIPS*, 2692–2700.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015b. Pointer networks. In *NIPS*, 2692–2700.

Wilder, B.; Dilkina, B.; and Tambe, M. 2019a. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 33, 1658–1665.

Wilder, B.; Dilkina, B.; and Tambe, M. 2019b. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *AAAI*, volume 33, 1658–1665.