CONVJSSP: Convolutional Learning for Job-Shop Scheduling Problems

Tianze Wang, Amir H. Payberah, and Vladimir Vlassov
Department of Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden
{tianzew,payberah,vladv}@kth.se

***Abstract*—The Job-Shop Scheduling Problem (JSSP) is a well-known optimization problem with plenty of existing solutions. Although remarkable progress has been made in addressing the problem, most of the solutions require input from human experts. Deep Learning techniques, on the other hand, have proven successful in acquiring knowledge from data without using step-by-step instructions from humans. In this work, we propose a novel solution, called CONVJSSP, by applying Deep Learning to speed up the solving process of JSSPs and to reduce the need for human involvement. In CONVJSSP, we train a Convolutional Neural Network model for predicting the optimal makespan of JSSPs, and use the predicted makespan to accelerate the JSSP solving schema. Through the experiments, we compare several JSSP solving methods based on CONVJSSP approach with a state-of-the-art solution as a baseline, and show that CONVJSSP speeds up the problem solving up to 9% compared to the baseline method.**

***Index Terms*—Deep Learning, Convolutional Neural Networks, Constraint Programming, Job-Shop Scheduling Problem**

I. INTRODUCTION

The *Job-Shop Scheduling Problem (JSSP)* [1], [2] is a well-known optimization problem, which has been studied extensively for a long time, but still attracts incessant interests of researchers. The problem is to find an *optimal schedule* of a number of jobs with varying processing times to be executed in parallel on a number of machines with an *optimal makespan*, which is the minimum time elapsed for executing all operations in all jobs.

Finding the *optimal solution* to a JSSP, that is a schedule with the optimal makespan, is a difficult task. However, once the value of the optimal makespan is known, the search space becomes significantly smaller by giving the extra constraint, that is the makespan of the solution equals the optimal makespan. Thus, we turn the problem of finding the optimal solution into finding the optimal makespan. Nevertheless, finding the optimal makespan is not a trivial task either, due to the combinatorial nature of JSSP [3].

Many solutions have been proposed for optimizing the makespan, such as exact methods [4], estimation methods [5]–[7], heuristics [8], metaheuristics [9], and domain-specific searching schemes on different JSSP variations [10]–[12]. Grimes et al. [13] introduce a new approach for solving JSSP by combining relatively simple inference with generic Constraint Programming (CP) techniques such as restarts, adaptive heuristics, and solution-guided branching, and Mirshekarian et al. [14] develop a Machine Learning (ML) model to predict the optimal makespan by classifying whether the optimal makespan is lower or higher than the average makespan of JSSP instances in the class.

One of the shortcomings of the existing solutions is that most of them require human experts in the loop. On the other hand, over the last few years, Deep Learning (DL) methods have shown their ability in acquiring knowledge from data without using step-by-step instructions from humans, across various domains [15]–[20]. Convolutional Neural Networks (CNN), as a class of the popular DL networks, have been successfully applied to tasks that involve understanding the content of grid-like input, e.g., images. Usually, a CNN is structured as a stack of convolutional layers (that extract the features of the input), followed by one or more fully-connected layers (that perform classification or regression tasks).

We propose CONVJSSP, a novel method of solving JSSPs using Deep Learning, that leverages CNNs to reduce the time of finding optimal solutions of JSSPs without much human effort. In particular, we study whether DL methods (e.g., CNNs) can be used to predict the optimal makespan of a JSSP instance without actually solving a given instance. Furthermore, we assess if the CNN-based predictions can speed up the process of finding the optimal makespan of a JSSP instance. CONVJSSP is a two steps process: (i) first, it takes the specification of a JSSP instance in the form of a 2D-matrix, and uses a CNN model to predict its optimal makespan without solving it, and (ii) second, it gives the specification of each JSSP and its predicted optimal makespan to a search strategy (explained in Section III) to find the optimal solution by prioritizing the sub search space pointed out by the prediction. Through the experiments, we compare several methods that utilize the prediction from our CNN model, with a state-of-the-art approach as a baseline for solving JSSP. The prediction-based methods achieve $1.067\times$ to $1.092\times$ speed up over the baseline method on the test benchmark.

The contributions of our work are as follows.

1) We introduce CONVJSSP, a two-step approach based on CNN, to speed up the process of finding the optimal solution of a JSSP. The implementation of CONVJSSP is available on GitHub [1].

[1] https://github.com/bwhub/CONVJSSP

2) We conduct extensive empirical evaluations on ConvJSSP and compare different prediction-based methods, and show that ConvJSSP speeds up the problem solving to up to 9% compared to the state-of-the-art.

The rest of the paper is organized as follows. Section II provides the preliminaries of JSSP and CNN. Section III describes how to use CNN models to predict the optimal makespan of JSSPs and discusses how to utilize the CNN predictions on the optimal makespan in order to fasten the process of finding the optimal solution for a JSSP. Section IV describes the experiment setup for empirical evaluation and provides the results and short discussions on it. Section V reviews the existing work, and finally, Section VI concludes the paper and points out directions for future work.

II. Preliminaries

In this section, we introduce the preliminaries of this paper.

A. Job-Shop Scheduling Problem

Job-Shop Scheduling Problem (JSSP) is one of the most difficult Constraint Optimization Problems (COPs) [1]. It is NP-hard [3] and many of its variations have been proven to be NP-complete [21]–[23]. There are many variants of JSSPs [10], however, in this work we consider the following version of JSSP: there are a set of m *machines* $M = \{M_1, M_2, \cdots, M_m\}$, and n *jobs* $J = \{J_1, J_2, \cdots, J_n\}$, such that each job J_i has m ordered operations $O_i = \{O_{i_1}, O_{i_2}, \cdots, O_{i_m}\}$. Each of the operations of a job should be processed on one of the m machines exactly once in a given order. Each machine can only process one operation at a time in a non-preemptive manner, meaning that operations cannot be interrupted once they start executing on a machine.

A JSSP *instance* is represented as a 2D-matrix, in which each row shows how the operations of a job should be executed, e.g., $J_i = (M_{i_1}, d_{i_1}, M_{i_2}, d_{i_2}, \cdots, M_{i_m}, d_{i_m})$, as an example of a row of a JSSP instance, indicates job J_i has m operations, and the first operation requires machine M_{i_1} and takes d_{i_1} units of time, and the last operation requires machine M_{i_m} and takes d_{i_m} units of time. A *solution* to a JSSP instance is a feasible schedule that does not violate the constraints, which are (i) preserving the order of the operations of each job, and (ii) executing only one operation at any time on each machine. The *makespan* of a solution is the total elapsed time if the solution is to be executed on real machines, and the *optimal makespan* of a JSSP instance is the smallest makespan of all solutions of that JSSP instance. An *optimal solution* of a JSSP instance is a solution that has the optimal makespan.

B. Finding the optimal solution of a JSSP

To find the optimal solution of a JSSP instance, we build a COP model with the objective function of minimizing the makespan. However, this is not a trivial task, as the number of feasible solutions is $(n!)^m$. Nevertheless, if we know the value of the optimal makespan beforehand, then the COP model becomes much easier to solve by adding an extra constraint, stating that the makespan of the solution should be equal to the optimal makespan of the given JSSP instance. This extra constraint significantly decreases the size of the search space that leads to a shorter time for finding the optimal solution. As it is nearly impossible to get the value of the optimal makespan, we resort to DL models, and in particular CNNs, for making predictions of the optimal makespan. To cope with the inaccuracy that comes with the prediction, instead of adding an extra constraint on the optimal makespan, one can modify the solving process to first look into solutions whose makespan are close to the prediction.

C. Convolutional Neural Network

Convolutional Neural Networks (CNNs) are considered as a common form of neural network for processing data that has a known, grid-like topology [24]. For example, image data can be viewed as a 2D grid of pixels. In a Fully-connected Neural Network (FNN), a neuron in one layer is connected to all the neurons in the previous layer and the next layer. In this way, the calculation in a FNN can be modeled by a sequence of general matrix multiplications. CNNs, on the other hand, replace the matrix multiplication of FNNs with convolutions (a specialized kind of linear operation) in at least one of the layers. CNNs are known for sparse interactions, parameter sharing, and equivalent representations. Each convolutional layer generates a higher-level abstraction of the input data that captures essential features of the input data [25].

Each JSSP instance (which shows how each operation of each job should be carried out) can be represented as a 2D-matrix, therefore it is reasonable to use CNNs to process them. For example, stacked convolutional layers can help to extract important features, e.g., how much operations of different jobs compete with each other on the same machine, or the average, minimum, and maximum duration of the operations in a JSSP instance. These important features are then passed to the fully connected layers to reason about the prediction on the optimal makespan for the given JSSP instance.

III. ConvJSSP

In this section, we present ConvJSSP, our two-step approach for expediting the process of finding the optimal solution of a JSSP instance. In this process, we first build a CNN model for predicting the optimal makespan of a JSSP instance, and then we design different search strategies based on the predicted makespan to speed up the process of finding the optimal solution.

A. Predicting the optimal makespan of a JSSP

In our work, we treat the task of predicting the optimal makespan as a regression task. To this end, we build a CNN model that takes a representation of a JSSP instance (in a form of 2D-matrix) as input, and then without solving the JSSP instance, it directly predicts its optimal makespan in a continuous space. We initially train a CNN model on the training dataset, then we use the trained CNN model for making predictions on JSSP instances on the test set.

1) Motivation for the CNN model: We opt for CNN models for the following reasons: first, feature engineering is not a trivial task and it usually requires a lot of domain expertise. It becomes even more complicated in JSSP, because defining the features, themselves, is challenging. CNN models, on the other hand, can just take JSSPs instances as 2D-matrices, perform the feature selection and combination automatically, and predict the optimal makespan. Second, while the performance of some traditional ML models is not much worse than the CNN models in terms of loss value, they suffer greatly from the problem of overfitting on those manually designed features.

The most straightforward way of defining a DL model is, perhaps, to consider a few fully connected layers. However, a model with only fully connected layers usually has a lot of parameters and tends to overfit on the training dataset. CNN, on the other hand, can share the parameters and reduce overfitting. Thus, for our DL model, we choose to use a CNN model that mainly contains two parts, the first part is a stack of convolutional layers to extract important features from the JSSP instances, and the second part is a couple of fully connected layers to predict the optimal makespan, based on the features extracted by the convolutional layer.

2) Design of the CNN model: To select a CNN model, we start with a simple architecture to show that even a simple model can speed up the solving of JSSP. We expected that with more complex architectures, we can predict the makespan more accurately, and consequently, we could achieve better results. Therefore, we test our solution with different CNN architectures, from simple models (stacked convolutional layers with fully connected layers [26]), to more advanced ones (with parallel convolutions [27]). However, when using advanced (more complex) architectures, the improvements in accuracy were not significant, thus, we decided to go with the simple model in the paper.

To this end, we propose a CNN model based on LeNet-5 [26], shown in Figure 1. Here, Input represents the input layer, Conv2D represents the 2D convolution layer, DepthwiseConv2D represents the 2D depthwise convolution layer, in which each convolution filter is applied to only one input channel at a time, Flatten represents the flatten layer that flattens each input to a 1D-vector, Dense represents the fully connected layer, and Dropout represents the dropout layer.

Our CNN model contains mainly two parts: (i) the convolutional layers that take as input a batch of JSSP instances (which are 9×9 matrices as we will explain in Section IV), and (ii) the fully connected part that takes the feature map extracted by the first part and outputs the prediction of the optimal makespan. Compared to the LeNet-5 model, our network mainly differs in three ways: (i) we remove the pooling layers since the dimension of our input is relatively small, (ii) we add the dropout layer to estimate the uncertainty of predictions, and (iii) we shrink the size of the network due to the relatively small size of the input.

3) Using dropout for modeling the uncertainty: While DL methods have been successfully applied to many tasks, most of the models do not capture the uncertainties of their predictions.

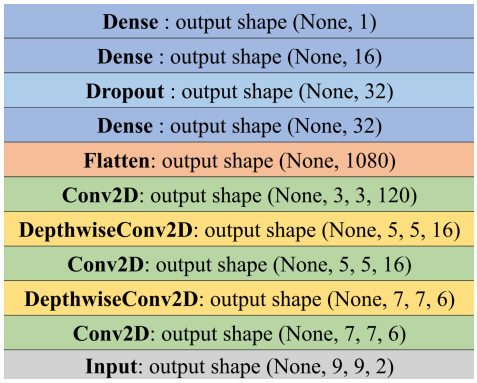

Fig. 1. The CNN model for predicting the optimal makespan of a given JSSP instance. A dropout layer is added to the model to account the uncertainties of the predictions.

Dropout is originally developed as a regularization method in DNNs to prevent overfitting [28], but Gal et al. [29] propose to use dropout during inference as an approximation of the uncertainty of the DL model. During the test phase, if the DL model runs prediction on the same input multiple times with dropout turned on, then we can acquire a group of different predictions for the same input. If these predictions are quite different from each other, then it might be a sign that the model is not very certain for its prediction on the given input.

We use dropout training in our CNN model, and during the test phase, the predictions on the same JSSP instance are run multiple times with dropout turned on with the same dropout rate as in training, which is the fraction of the input units to drop. This leads to a group of different predicted optimal makespan for the same JSSP instance. Then, we use the mean value of this group of predictions as the predicted optimal makespan and the standard deviation as an estimate for the uncertainty associated with the prediction.

B. Developing Search Strategies with the prediction

After predicting the optimal makespan of a JSSP instance, we utilize it to speed up the process of finding the optimal solution for that JSSP instance. Here, we first explain the baseline searching strategy, which is a pure CP approach, and then, we introduce multiple search strategies (i.e., greedy, jumping, and hybrid strategies) that use the predicted optimal makespan from the CNN model to fasten the process of finding the optimal solution.

1) Baseline strategy: The *baseline* strategy [13] represents the state-of-the-art CP approach for finding the optimal solution for a JSSP instance. This process contains three phases: *probing*, *adjusting*, and *solving*.

During the probing phase, the CP program randomly probes into the search space defined together by the set of constraints specific to the given JSSP instance, such as, the order of execution, the duration of each operation, and a lower and upper bound for the optimal makespan. The lower bound can be calculated by assuming that all machines are busy with computation from the beginning to the end and we know that

the optimal makespan cannot be smaller than the lower bound because otherwise we will have machine utilization of more than 100%. The upper bound can be calculated as if all the operations are carried out sequentially.

Through random exploration, if the CP program happens to find a solution, the corresponding makespan is reported and the upper bound is updated to the value of that makespan. The update is based on an important property of the optimal makespan that states: if we know there is a solution whose makespan is of value v, then we know that the value of the optimal makespan should be no larger than v, and if we know there is no solution whose makespan is of value w, then we know that the value of the optimal makespan should be larger than w. The probing phase ends when either the CP programs reach a predefined number of trials or a predefined timeout.

During the adjusting phase, the CP program applies the *interval bisection* on the interval defined by the lower bound l and the upper bound u on the optimal makespan. The CP program repeatedly checks if there is a solution whose makespan is within l and $x = \frac{l+u}{2}$. If there is a solution, then the program updates the upper bound according to $u = x$, if there is no solution, then the program updates the lower bound according to $l = x$. The adjusting phase goes iteratively until either the CP programs reach a predefined timeout, or the lower and upper bound is of the same value, meaning that the optimal solution is found and the optimal makespan is the current value of the lower and upper bound.

During the solving phase, the CP program resorts to exhaustive search to find optimal solutions whose makespan is within the last interval found by the adjusting phase. Since we are interested in finding the optimal solution, the solving phase will only terminate when the CP program finds the optimal solution. The baseline strategy mainly follows the idea from [13]. However, it is important to note that our implementation of the baseline strategy is a sketch and does not include all techniques from the paper [13], that could lead to less competitive performance. For details of our implementation of the baseline, please refer to the following link[2].

2) Greedy strategy: Compared to the baseline strategy that tries to find the optimal solution in three phases using the upper and lower bound, the *greedy* strategy takes a more straightforward approach with an extra input of the predicted optimal makespan of value p, provided by the CNN model. If there is a solution whose makespan is of value p then $p = p-1$, otherwise $p = p + 1$. Then the greedy CP program keeps doing the same on the new value of p iteratively. The stopping criterion would be one of the following two: (i) if there is a solution with the current value of p, but not the previous value of p, then terminate the search and report the current value of p as optimal makespan, and (ii) if there is no solution with the current value of p, but there is a feasible schedule with the previous value of p, then terminate the search and report the previous value of p as the optimal makespan.

3) Jumping strategy: While the greedy strategy represents a direct way to utilize the predicted optimal makespan, it might suffer from performance issues when the prediction is far away from the true optimal makespan, as we are iterating through the search space one step at a time. To overcome this problem, we propose two *jumping* strategies to go faster through the bound for the optimal makespan. Note that the jumping strategies take the predicted optimal makespan of value p as input. The two jumping strategies are as below:

- *Jump-half*: it checks if there is a solution whose makespan is of value p, if so, then $p = \frac{p+l}{2}$, where l represents the value of the lower bound on optimal makespan, otherwise $p = \frac{p+u}{2}$, where u represents the value of the upper bound on optimal makespan. The program will use the same stopping criterion as the greedy version, except it does not necessarily report the optimal solution, but it more often provides us with a better lower and upper bound on the optimal makespan that can be later used by other methods, e.g., exhaustive search to find the optimal solution.
- *Jump-steps*: it is very similar to jump-half, except that instead of taking $p = \frac{p+l}{2}$ or $p = \frac{p+u}{2}$, it uses $p = p - s$ or $p = p+s$ to update p during the iterative search, where s is the step size and is a hyper-parameter.

4) Hybrid strategy: Here, we show four different *hybrid* strategies, each of which uses different techniques to utilize the prediction of the optimal makespan.

- *Hybrid-jump-half-greedy*: the program first applies the probing phase, and if the predicted optimal makespan by CNN is not within the lower and upper bound found after the probing phase, then the program follows the adjusting and solving phases in baseline strategy, otherwise it applies the jump-half and greedy strategies to find the optimal solution. To do so, the program first finds a new lower bound l_{new} and upper bound u_{new} using the jump-half strategy, and then to perform the greedy strategy, it does either of the following: (i) if the predicted optimal makespan is within l_{new} and u_{new}, it starts the greedy search at the predicted optimal makespan, otherwise (ii) it starts the greedy search at the point $\frac{l_{new}+u_{new}}{2}$.
- *Hybrid-jump-half-exhaustive*: the difference between this strategy and the hybrid-jump-half-greedy is that instead of using greedy search, this one uses exhaustive search.
- *Hybrid-jump-steps-greedy*: the difference between this strategy and the hybrid-jump-half-greedy is that instead of using jump-half, this one uses jump-steps.
- *Hybrid-jump-steps-exhaustive*: the difference between this strategy and the hybrid-jump-half-exhaustive is that instead of using jump-half, this one uses jump-steps.

IV. Experimental Results

In this section, we first describe how the datasets for JSSP benchmarking and CNN training are acquired, and also present

[2]https://github.com/Gecode/gecode/blob/master/examples/job-shop.cpp

the settings of hardware, software, and parameters for our experiments. Then, we show the results of the CNN training and the CP benchmarking across different searching strategies.

A. Datasets acquisition

To choose the dataset, initially we test some of the JSSP instances in the MiniZinc benchmark[2], but due to a rather small (insufficient) number of instances for CNN training, we use self-generated random benchmarks of similar formats for the CNN model to learn features from the JSSP instances. To do so, we generate two different datasets in the experiments: (i) `Instance-Q10000` dataset, the dataset for CP benchmarking, and (ii) `DL-Q10000` dataset, the dataset for training and testing the CNN model. The `Instance-Q10000` dataset contains 10000 JSSP instances of nine jobs and nine machines (i.e., 9×9). The duration of operations in each job is randomly sampled from a uniform distribution between 1 to 99, including the boundaries. We selected the number of instances, the number of jobs and machines, and the duration of each operation, such that we have big enough samples to make the models, but small enough to be able to conduct the experiments in a reasonable time. To generate the `DL-Q10000` dataset, we execute the baseline CP program on the `Instance-Q10000` dataset to get the optimal makespan for each JSSP instance. The JSSP instances together with their optimal makespan, then, form the `DL-Q10000` dataset that is used as the training and testing data sets of the CNN model. Within the `DL-Q10000` dataset, 5000 are randomly selected as training data, 1000 as validation data, and 4000 as test data.

B. CP benchmark setting

We implement all the CP programs using Gecode 6.2.0 [30]. Gecode natively provides multi-threading to speed up the solution finding process, however, in all of our benchmark of comparing different strategies to find optimal makespan, we use a single thread, because we use the execution time as a surrogate for the computation cost, thus, the extra speed up brought by multi-threading complicates the measurement of computation of the problem.

To measure the time that each CP program takes, we use the built-in time measuring function provided by Gecode. We dedicate one machine for the execution of testing benchmark instances to minimize other programs' impact in our measurements. To further compensate other factors that might contribute to the measured running time, such as background operating system processes, we run each CP program on each JSSP instances three times and report the relevant statistics in the result.

Timeout is set for some phases so that CP programs move to the next phase when they have spent a certain amount of time in one phase. For the baseline strategy benchmark on the `Instance-Q10000` dataset, the timeout for probing and adjusting phase is set to 1 and 30 seconds, respectively. There is no time out for the solving phase. For the greedy and hybrid strategies, there is no timeout.

For each search strategy, we test it with three different branching options, which are different variable selection criteria that helps to define the shape of the search tree during branching: (i) *Accumulated Failure Count (AFC)* [31] that counts how often propagators have failed during search, (ii) *Action* [32] that captures how often the domain of a variable has been reduced during constraint propagation, and (iii) *Conflict History-based Branching (CHB)* [33] that considers both how often the domain of a variable has been reduced during constraint propagation and how recently the variable has been reduced during failure.

The experiments for solving the JSSP instances in the test set are performed on a machine with the following specification:

- CPU: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
- Memory: 4×4 GB DDR3 1600 MHz
- Operating System: Ubuntu 18.04.4 LTS
- Kernel: Linux 5.3.0-45-generic

Note that Turbo Boost [3] has been turned off on the machine for reproducibility.

C. CNN modeling setting

To train the CNN model, we give as input the JSSP instances in form of 2D-matrices (i.e., 9×9), such that the values of their elements are standardized by removing the mean and scaling to unit variance. As for the optimal makespan, no preprocessing is performed on the optimal makespan. The details of the model are shown in Figure 1. We manually design the model and choose the hyperparameters based on our experience and exploration within this experiment setting. A full study of neural architectural search and hyperparameter optimization might lead to better results, but is outside the scope of this work.

We use the Keras API within TensorFlow [34] 2.1.0 to implement the CNN model and train it. We consider the *Mean Squared Error (MSE)* as the loss function, i.e., $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$, where y_i and $\hat{y}_i$ are the optimal makespan and the predicted optimal makespan of the ith JSSP instance in the batch, respectively, and n is the number of samples in a batch.

D. Results for the CNN model

Figure 2 shows the MSE of the predicted the optimal makespan on the `DL-Q10000` dataset. As we see after 600 epochs of training, the MSE of the model on validation data starts to go up, meaning that the model is performing worse on the validation set. After 500 epochs of training, while the training loss in the experiment decreases significantly, the validation and test loss only decrease for a small amount. Although the model starts to show slight signs of overfitting after 500 epochs, the model performance on the validation set

[2] https://github.com/MiniZinc/minizinc-benchmarks/tree/master/jobshop

[3] https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

peaks at 600 epochs that gives an overall better model than models trained after other numbers of epochs.

Thus, we stop training the model at this point. Then, we use the trained model for making predictions on the test dataset. To do this, we run the prediction for the same JSSP instance, 999 times to form a distribution of the prediction, which is used later to estimate the uncertainty of the prediction. The time spent to train and inference the CNN model is relatively small compared to the time we have saved with different searching strategies. To train the model, each 100 epoch takes around 20 seconds on a single Nvidia GEFORCE GTX1080. Running 999×4000 predictions take about 51 seconds (CPU only) on the same machine with an AMD Ryzen 5 1600 Processor.

E. Result of different searching strategies

After we get the predicted optimal makespan on the test set, we perform the benchmarking by running different search strategies using the three branching strategies, i.e., AFC, Action, and CHB. The average of all the predictions on a single JSSP instance is taken as the predicted optimal makespan for each search strategy. We test the hybrid-jump-steps-greedy and hybrid-jump-steps-exhaustive using three different values for the step size s: one, three, and five standard deviations of all the predictions on a single JSSP instance.

Table I shows the time the search strategies take to finish the benchmark on the 4000 JSSP instances in the test set. It shows the result for each of the branching strategies (i.e., AFC, Action, and CHB), as well as their sum. Table II shows the number of JSSP instances that each of the search strategies outperforms the baseline strategies. As the results show, the greedy strategy outperforms the baseline for a lot of the JSSP instances. However, when the greedy strategy is slower than the baseline method to find the optimal makespan, it is way much slower that is seen from the total running time of the greedy strategy. One explanation for this behavior is that the performance of greedy strategy could be greatly hindered when the predicted optimal makespan is far away from the ground truth, since the greedy strategy is only moving one step at a time within the lower and upper bound for the optimal makespan, while other strategies are using search on intervals.

The behavior of the greedy strategy leads us to design strategies based on a combination of CNN-based search strategies and the baseline method. We have observed through experiments that the probing phase in the baseline strategy, which usually does not take much computation time, can help to significantly speed up the bound on the optimal makespan. Thus, whether the predicted optimal makespan is within this bound found after the probing phase can serve as a reflection on the quality of the prediction. In this way, if the predicted optimal makespan is within this bound, we can use the prediction-based strategies; Otherwise, we just use the baseline strategy.

The plan for hybrid searching strategies, together with the jumping methods, works if we just compare the running time of the greedy and the hybrid-jump-half-greedy, in which the latter requires less time to finish the test benchmark. Nevertheless, as Table II shows, the hybrid-jump-half-greedy is worse than the greedy partly because of the overhead of running the probing phase. Another reason, perhaps the more important one, is that we are still using the greedy methods, which is not very efficient. Moreover, we observe that switching the greedy part to exhaustive search improves the performance, both in time to finish the benchmark, and the number of instances that is faster than the baseline.

Another type of hybrid strategies are the hybrid-jump-steps-greedy and the hybrid-jump-steps-exhaustive that take step size s as input to advance for each jump. The benchmark results for the hybrid-jump-steps-greedy is always worse than its exhaustive search counterparts, and thus is not included due to the space limit. While the jump-steps methods do not always guarantee better performance than the jump-half counterpart, tuning the step size can lead to different performances.

Design choices (when to use what strategy) Finding a proper search strategy is important. While there is no golden rule to choose a search strategy, we can still reason about which one to use based on both the quality of the prediction and the use case. For example, if we believe that the predictions are of high quality, then the greedy method is definitely a good strategy. However, the performance in terms of solving time can greatly be hindered, if the predictions are actually far away no matter if we use the greedy approach or the hybrid approach containing a greedy phase. Thus, if we are not sure about the quality of the predictions, then one of the hybrid approaches could be the way to go, as it offers a fail-safe in case the prediction is far off. When we are certain that the prediction is of low quality, the baseline method is probably the one to go.

V. Related Work

In this section, we describe the related work of the paper. Xu et al. [35] apply CNN to predict the satisfiabilities of Constraint Satisfaction Problems (CSPs). The paper uses a deep CNN that takes the matrix representations of CSPs as input and predicts the satisfiabilities of CSPs. Lin in [36] examines the influence of machine correlation and job correlation on computation time. Empirical evaluation shows that branch-and-bound based algorithms have more difficulty in solving parallel machine scheduling problem with higher correlations, meaning that the correlations can be used to forecast how branch-and-bound based algorithms will perform on certain problems, and thus fasten the solving process by selecting the most appropriate algorithm.

Mirshekarian et al. [14] provide a statistical study of the relationship between JSSP features and the optimal makespan. Their study includes a set of 380 carefully hand-designed features, each representing a certain aspect of the scheduling problem. The 380 features are divided into two categories: (i) the configuration features, which are taken from the JSSP

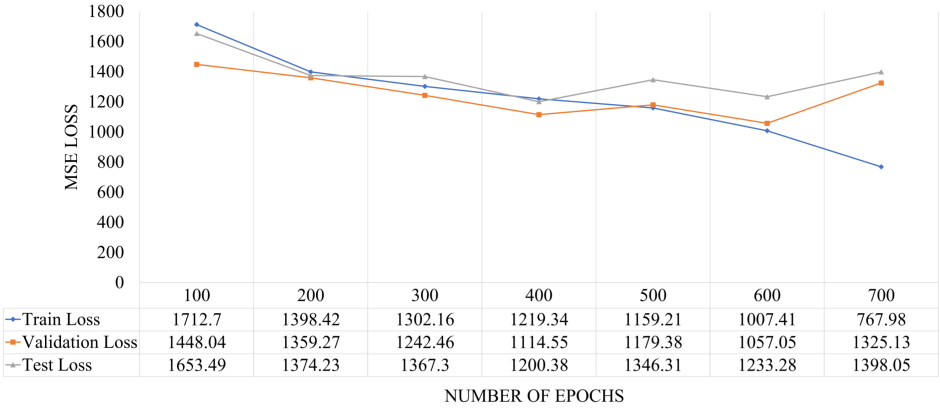

Fig. 2. Loss for the CNN model.

TABLE I
COMPARISON OF RUNNING TIME (SECONDS) TO FINISH THE TESTING BENCHMARK (4000 JSSP INSTANCES) ACROSS DIFFERENT STRATEGIES. EACH DATA POINT IS BASED ON THREE INDEPENDENT RUN AND MEAN OF STANDARD DEVIATION IS REPORTED IN THE BRACKET.

Branching Strategy	Action	AFC	CHB	Total
baseline	5768 (0.013)	11926 (0.035)	9585 (0.050)	27278 (0.033)
greedy	12588 (0.007)	33157 (0.019)	24369 (0.015)	70113 (0.014)
hybrid-jump-half-greedy	9680 (0.004)	26149 (0.011)	15993 (0.007)	51822 (0.007)
hybrid-jump-half-exhaustive	5682 (0.002)	**11056** (0.004)	9932 (0.004)	26670 (0.004)
hybrid-jump-std-exhaustive	7426 (0.003)	16603 (0.009)	13541 (0.006)	37570 (0.006)
hybrid-jump-3std-exhaustive	**5371** (0.003)	11879 (0.005)	9224 (0.004)	26473 (0.004)
hybrid-jump-5std-exhaustive	5414 (0.002)	11362 (0.004)	**8781** (0.004)	**25557** (0.004)

TABLE II
COMPARISON OF DIFFERENT STRATEGIES ON HOW MANY JSSP INSTANCES CAN THEY FIND OPTIMAL MAKESPAN FASTER THAN BASELINE STRATEGY ON THE TESTING BENCHMARK (4000 JSSP INSTANCES). A PERCENTAGE IS ALSO PROVIDED IN THE BRACKET FOR COMPARISON.

Branching Strategy	Action	AFC	CHB	Total
greedy	**2697** (67.4%)	**2463** (61.6%)	2194 (54.9%)	**7354** (61.3%)
hybrid-jump-half-greedy	1233 (30.8%)	1199 (30.0%)	1116 (27.9%)	3548 (29.6%)
hybrid-jump-half-exhaustive	1597 (39.9%)	1804 (45.1%)	1549 (38.7%)	4950 (41.2%)
hybrid-jump-std-exhaustive	1861 (46.5%)	1831 (45.8%)	1755 (43.9%)	5447 (45.4%)
hybrid-jump-3std-exhaustive	2252 (56.3%)	2303 (57.6%)	**2294** (57.4%)	6849 (57.1%)
hybrid-jump-5std-exhaustive	2157 (53.9%)	2253 (56.3%)	2141 (53.5%)	6551 (54.6%)

specification, and (ii) the temporal features, which are concerned with the information about the solving process such as the output of a dispatching rule, heuristic applied to the problem. These features are later used by an ML model to predict whether the optimal makespan of a JSSP instance is higher or lower than the average of the class of instances under study.

While [14] focus on a binary classification task of whether the optimal makespan of a given JSSP instance is smaller or higher than the class average, this kind of prediction is on too coarse a granularity which makes it difficult for CP programs to utilize the prediction. To the best of our knowledge, we are the first one to effectively apply CNN for making predictions on the optimal makespan of JSSP instance in a continuous space without actually solving the instance. Also, our CNN model does not require hand designed features. Furthermore, we empirically demonstrate that the predicted optimal makespan can be used to fasten the process of finding the optimal makespan using CONVJSSP.

VI. CONCLUSION AND FUTURE WORK

In this work, we investigate how Convolutional Neural Networks (CNN) can be used to speed up the process of finding the optimal solution for Job-Shop Scheduling Problems (JSSP). We propose CONVJSSP, a novel method to solve JSSPs by applying CNN models for predicting the optimal makespan given JSSP instances as input. We test our solution on JSSP instances of nine jobs, each of which has nine operations, on nine machines, and show that CONVJSSP outperforms the Constraint Programming (CP) state-of-the-art method, as the baseline, in terms of both the time to finish a test benchmark of 4000 instances and the number of cases that our method is up to 9% faster than the baseline within the test benchmark. An analysis of the experiment results and a discussion on how to choose between CNN-based strategies is also provided.

As with any study, our work also has its limitations that point to some of the directions for future work. How to build a CNN prediction model that can work on various input sizes?

While one can easily fix the problem across nine by nine, and 10 by 10 matrices by adding padding to the smaller matrices, it certainly becomes a problem when sizes of the JSSP instances varies a lot. One potential solution to this is to use a recurrent neural network that can handle input of various sizes by nature.

Moreover, there are several interesting aspects to look into in the future. For example, how can neural architectural search and hyperparameter optimization help to build better Deep Learning (DL) models? How does the dropout rate affect the quality of the uncertainty estimation of each prediction? How to build a DL model that can predict the optimal makespan decently when we do not have access to a large training dataset? Or is there a way to transfer the knowledge the model has learned on the generated dataset to the JSSP benchmarks that are used in the industry? How can we generalize our two-step approach for JSSP to CP problems in other domains? And finally, are there any general strategies for modeling the problem? All the above questions could be subjects to future work on the use of DL in solving JSSPs.

ACKNOWLEDGMENTS

We would like to express our very great appreciation to Christian Schulte, for the initialization and critical contribution at the early stages of this work. We would also like to thank Amir Hossein Akhavan Rahnama, Linnea Ingmar, and Rodothea Myrsini Tsoupidi for their generous discussion.

REFERENCES

[1] E. Lawler et al., "Sequencing and scheduling: Algorithms and complexity," *Handbooks in operations research and management science*, vol. 4, pp. 445–522, 1993.

[2] V. Saraswat et al., *Principles and practice of constraint programming: the Newport papers*. MIT Press, 1995.

[3] J. Lenstra et al., "Computational complexity of discrete optimization problems," in *Annals of discrete mathematics*. Elsevier, 1979, vol. 4, pp. 121–140.

[4] B. Giffler et al., "Algorithms for solving production-scheduling problems," *Operations research*, vol. 8, no. 4, pp. 487–503, 1960.

[5] D. Hochbaum et al., "Using dual approximation algorithms for scheduling problems theoretical and practical results," *Journal of the ACM (JACM)*, vol. 34, no. 1, pp. 144–162, 1987.

[6] Z. Lian et al., "A similar particle swarm optimization algorithm for job-shop scheduling to minimize makespan," *Applied mathematics and computation*, vol. 183, no. 2, pp. 1008–1017, 2006.

[7] K. Mesghouni et al., "Evolutionary algorithms for job-shop scheduling," *International Journal of Applied Mathematics and Computer Science*, vol. 14, no. 1, pp. 91–104, 2004.

[8] E. Balas et al., "The one-machine problem with delayed precedence constraints and its use in job shop scheduling," *Management Science*, vol. 41, no. 1, pp. 94–109, 1995.

[9] K. Tamssaouet et al., "Metaheuristics for the job-shop scheduling problem with machine availability constraints," *Computers & Industrial Engineering*, vol. 125, pp. 1–8, 2018.

[10] J. Błażewicz et al., "The job shop scheduling problem: Conventional and new solution techniques," *European journal of operational research*, vol. 93, no. 1, pp. 1–33, 1996.

[11] A. Jain et al., "Deterministic job-shop scheduling: Past, present and future," *European journal of operational research*, vol. 113, no. 2, pp. 390–434, 1999.

[12] L. P. Michael, *Scheduling: theory, algorithms, and systems*. Springer, 2018.

[13] D. Grimes et al., "Solving variants of the job shop scheduling problem through conflict-directed search," *INFORMS Journal on Computing*, vol. 27, no. 2, pp. 268–284, 2015.

[14] S. Mirshekarian et al., "Correlation of job-shop scheduling problem features with scheduling efficiency," *Expert Systems with Applications*, vol. 62, pp. 131–147, 2016.

[15] G. Hinton et al., "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[16] X. Huang et al., "A historical perspective of speech recognition," *Communications of the ACM*, vol. 57, no. 1, pp. 94–103, 2014.

[17] Y. LeCun et al., "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[18] F. Borisyuk et al., "Rosetta: Large scale system for text detection and recognition in images," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 71–79.

[19] A. Esteva et al., "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, pp. 115–118, 2017.

[20] G. Litjens et al., "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017.

[21] M. Garey et al., "The complexity of flowshop and jobshop scheduling," *Mathematics of operations research*, vol. 1, no. 2, pp. 117–129, 1976.

[22] T. Gonzalez et al., "Flowshop and jobshop schedules: complexity and approximation," *Operations research*, vol. 26, no. 1, pp. 36–52, 1978.

[23] J. Lenstra et al., "Complexity of machine scheduling problems," in *Annals of discrete mathematics*. Elsevier, 1977, vol. 1, pp. 343–362.

[24] I. Goodfellow et al., *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[25] V. Sze et al., "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[26] Y. LeCun et al., "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[27] C. Szegedy et al., "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[28] N. Srivastava et al., "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[29] Y. Gal et al., "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *international conference on machine learning*, 2016, pp. 1050–1059.

[30] C. Schulte et al., "Modeling and programming with gecode," *Schulte et al., C.*, vol. 1, 2010.

[31] F. Boussemart et al., "Boosting systematic search by weighting constraints," in *ECAI*, vol. 16, 2004, p. 146.

[32] L. Michel et al., "Activity-based search for black-box constraint programming solvers," in *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer, 2012, pp. 228–243.

[33] J. Liang et al., "Exponential recency weighted average branching heuristic for sat solvers." in *AAAI*, 2016, pp. 3434–3440.

[34] M. Abadi et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[35] H. Xu et al., "Towards effective deep learning for constraint satisfaction problems," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2018, pp. 588–597.

[36] Y.-K. Lin, "Scheduling efficiency on correlated parallel machine scheduling problems," *Operational Research*, vol. 18, no. 3, pp. 603–624, 2018.