Self-Supervised Long-Short Term Memory Network for Solving Complex Job Shop Scheduling Problem

Xiaorui Shao[1], Chang Soo Kim[1*]
[1] Department of Information System, Pukyong National University
Busan, 608737 South Korea
[e-mail: shaoxiaorui@pukyong.ac.kr, cskim@pknu.ac.kr]
[*]Corresponding author: Chang Soo Kim

Received March 14, 2021; revised May 13, 2021; accepted June 10, 2021; published August 31, 2021

Abstract

The job shop scheduling problem (JSSP) plays a critical role in smart manufacturing, an effective JSSP scheduler could save time cost and increase productivity. Conventional methods are very time-consumption and cannot deal with complicated JSSP instances as it uses one optimal algorithm to solve JSSP. This paper proposes an effective scheduler based on deep learning technology named self-supervised long-short term memory (SS-LSTM) to handle complex JSSP accurately. First, using the optimal method to generate sufficient training samples in small-scale JSSP. SS-LSTM is then applied to extract rich feature representations from generated training samples and decide the next action. In the proposed SS-LSTM, two channels are employed to reflect the full production statues. Specifically, the detailed-level channel records 18 detailed product information while the system-level channel reflects the type of whole system states identified by the k-means algorithm. Moreover, adopting a self-supervised mechanism with LSTM autoencoder to keep high feature extraction capacity simultaneously ensuring the reliable feature representative ability. The authors implemented, trained, and compared the proposed method with the other leading learning-based methods on some complicated JSSP instances. The experimental results have confirmed the effectiveness and priority of the proposed method for solving complex JSSP instances in terms of make-span.

Keywords: Job Shop Scheduling Problem (JSSP), LSTM, Smart Factory, ICONI 2020

A preliminary version of this paper was presented at ICONI 2020, December 13-16, Jeju Island, South Korea, and was selected as one outstanding paper. This version includes a concrete explanation of SS-LSTM and related works analysis.

http://doi.org/10.3837/tiis.2021.08.016 **ISSN : 1976-7277**

1. Introduction

The job shop scheduling problem (JSSP) plays a vital role in the process of building a smart factory regarding intelligent manufacturing [1][2], resource supplying [3][4], and cost-saving [5]. It is a continuous process-making problem, in which the scheduling arranges each job at each subphase sequentially for finding one optimal solution to minimize the make-span of the whole production. The solution space of one huge JSSP will increase exponentially. Especially for one $m \times n$ JSSP (where m is the number of machines, and n is the number of the jobs), the solution space is $(n!)^m$ [6][7], which is known as NP-hard [8].

Many methods have been proposed to solve this challenging task, which could be divided into three categories: population-based [5], gene-based [9], and learning-based methods. The population-based methods, including particle swarm optimization (PSO) [10], ant colony optimization (ACO), try to find the optimal solution for JSSP but are very time-consumption and resource-consumption, especially when meeting massive JSSP instances. To make a trade-off between solution quality and computational cost, a near-optimal solution: gene-based methods are proposed and employed. E.g., Asadzadeh [10] improved the GA performance for JSSP by using an agent-based local search strategy. Kurdi et al. [11] proposed a new island model genetic algorithm (IMGA) to solve JSSP and verified their model on 52 JSSP instances. However, the current industry production environment is very complex and changing. Population-based, and gene-based methods require to repeat many iterations and update operations frequently, which takes a long time and is hardware-consumption.

Luckily, with the continuous and booming development of artificial intelligence (AI) technology, the learning-based method has been proposed and employed to solve complex JSSP and achieved great success [7][12]. The learning-based methods mainly treated a JSSP as one sub-classification problem, consisting of shallow learning and deep learning methods. The workflow of learning-based methods for solving JSSP, as shown in **Fig. 1**. Step 1 is to obtain the training samples by using other optimal solves such as GA and PSO; Step 3 is the key to ensure the accuracy of solving JSSP; Step 4, 5, and 6 are to solve one upcoming JSSP instance.

Shallow learning-based methods for JSSP mainly include support vector machine (SVM) [13], random forest (RF) [14], and neural network (NN). Among them, NN has attracted various attention and been applied for JSSP in the last decades. E. g., Foo et al. [15] are the first to use NN for the JSSP. They treated the scheduling problem as one integer linear programming problem and utilized an improved Tank and Hopfield neural network model to solve JSSP. Gary et al. [6] used GA to generate the best solution and training samples, then they map those samples into operation, process time, remaining time, and machine load features as the input of NN to train the model. The comparative analysis proved the feasibility and scalability of the NN scheduler on more complex JSSP instances. However, shallow NN only extract the hidden patterns by using two or three hidden layers, which lose some critical information to predict the next action.

Deep learning [16] technology could extract more accurate and robust feature representations by increasing the hidden layers to achieve better performance. One of the most popular deep learning algorithm, convolutional neural network (CNN), has been applied to solve JSSP due it is capable of extracting most hidden representations from one-dimensional (1-D) sequence [17], two-dimensional (2-D) images [18], and three-dimensional video [19]. E. g., Zhao et al. [20] integrated GA and CNN to solve one 7×7 JSSP instance, in which GA is used for the optimization of sequence while CNN is used for extracting the features from the operation start times with a 1-D fixed sequence. Recently, Zang et al. [7] developed one

2-D CNN to extract the features from GA generated training samples, the results confirmed the effectiveness of the proposed hybrid deep neural network scheduler (HDDNS) with various JSSP instances, where they defined ten variables as the inputs of the NN. Moreover, a cartesian product was applied to transform the 1-D regular data into a 2-D tensor to extract the rich features. Although 2-D CNN has obtained excellent results, it requires transforming the 1-D data into a 2-D tensor, which is very time-consumption and hardware-consumption. Moreover, it ignores some critical features with time steps because JSSP is an ongoing process-making problem. Therefore, the accuracy is still not satisfactory.

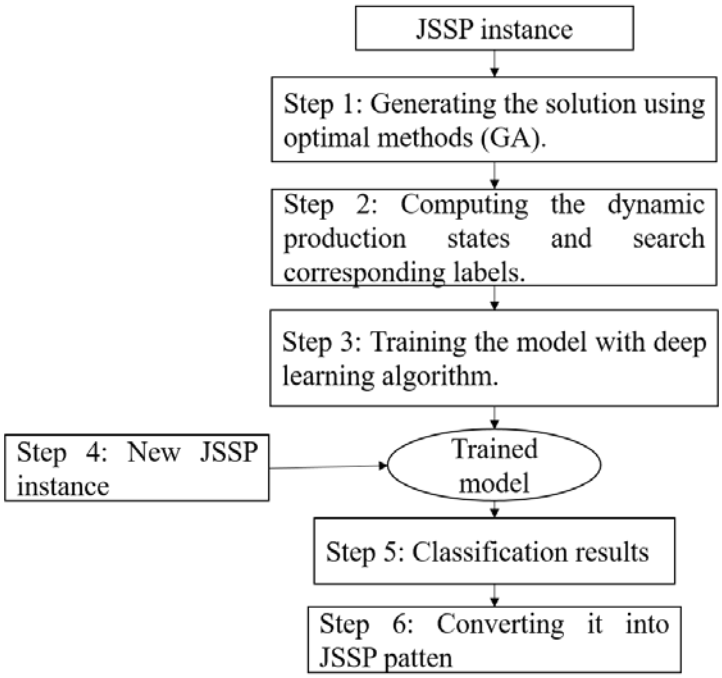

Fig. 1. The learning-based method for solving the JSSP. The black line trains the model while the blue line is to solve the new JSSP instance.

Another branch of learning-based methods for solving JSSP is reinforcement learning (RL), which select the next action with the highest score according to the Q table, a tuple records the states and corresponding action. RL has achieved a big success in the area of JSSP in [21][22][23]. However, RL meets the same issue to the population-based and gene-based methods: Q table is too large with a sizeable JSSP instance, which results in a long time and requires a massive memory. To address this shortcoming, deep reinforcement learning (DRL) [24], also known as deep Q network (DQN), have been proposed and applied for JSSP. In which deep learning technology is to extract the rich feature representations to forecast the next actions' score according to the Q table generated from the RL algorithm, the machine will execute the following action according to the particular policy. E. g., Mao et al. [25] applied DRL with one deep NN (DNN) structure to manage the resource of CPU and memory. Chen et al. [26] used DRL to process multi-resource and multi-job scheduling problems. They adopted CNN to extract feature representations of a dynamic production environment (CNN-DRL1). Moreover, Ye et al. [27] proposed a more complex DRL framework based on CNN for resource scheduling (CNN-DRL2). Lin et al. [28] offered a multi-class DQN with edge computing devices for JSSP. Liu et al. [2] proposed an actor-critic DRL framework for JSSP

based on CNN. However, RL-based methods still need to update the Q table. They may fail to process the complex JSSP instance due to memory limitations in the real world.

The limitations of current methods for JSSP are summarized as follows. Conventional methods such as population-based and gene-based methods are time-consumption and resource-consumption for large JSSP instance. The shallow learning methods such as SVM, RF, and shallow NN cannot extract sufficient features to reflect the production environments. The CNN-based methods did not consider the influence of time steps in the dynamic production environments, so that the performance is still not satisfactory. Besides, it requires some extra transformation. Similar to the population-based methods, the RL-based methods cannot process the complex JSSP.

To overcome the limitations mentioned above, this paper developed a novel learning-based deep model named self-supervised LSTM [29] (SS-LSTM) for solving complex JSSP. In the proposed method, dual channels are employed to reflect the production environment altogether. Primarily, we defined 18 variables to reflect the detailed production states in the detail-level channel, and the K-means algorithm identifies the system-level state to reflect the system-level production states. LSTM extracted detail-level features are merged with the system-level feature for final classification. Furthermore, the self-supervised mechanism makes a trade-off between feature extraction and feature reconstruction. Consequently, the final features are the fusion of detail-level, system-level production states considering the influence of time steps, enabling the forecasting more accurately.

The main contributions of this manuscript are summarized as:

- To our best understanding, this paper is the first to use LSTM for complex JSSP within a self-supervised mechanism.
- A new framework SS-LSTM has been proposed to handle complex JSSP instance accurately and efficiently. Various comparative analysis has confirmed its effectiveness.
- Each component's effect has been analyzed through an ablation study.
- The proposed method has good robustness for solving JSSP.

The rest of the paper is arranged as follows. Section 2 introduces some pre-knowledge, including JSSP, LSTM. Section 3 gives a detailed description of the proposed SS-LSTM for JSSP. In Section 4, we utilized different JSSP data sets to validate the proposed method's effectiveness, and various comparative studies are carried out. We discussed the proposed method for JSSP in Section 5. Section 6 conducted this manuscript and feature work.

2. Methodology

2.1 JSSP

The JSSP is to arrange multi-machines to process multi-jobs with satisfactory make-span and lowest cost. There are n jobs $J = \{J_1, J_2, \dots, J_n\}$ arriving and to be processed by m machines $M = \{M_1, M_2, \dots, M_m\}$. Each job J_i consists of n sub-operations, where O_{ij} denotes j^{th} sub-operation of the job J_i. Any functional machines could process each sub-operation M_k with processing time t_{ijk}, which means that j^{th} sub-operation of the job J_i needs to be processed at the machine M_k in time t. M_k is the corresponding functional machine for sub-operation O_{ij}, selected from the M machines, that is, $M_k = M_{ij} \in M$. The whole process will end until arranging all sub-operations.

An 8×6 JSSP instance ($n = 8, m = 6$) as illustrated in **Fig. 2**. The left is the job order O_{ij} table, which records the requiring machine k to process each sub-operation O_{ij}. The right

is corresponding processing time t_{ijk}, in which j^{th} sub-operation of the job J_i needs to be processed by machine k with the given time. For instance, the red colour content in **Fig. 2** represents sub-operation O_{11} needs to be processed by machine 6 with 15 units. The corresponding solution using GA, as shown in **Fig. 3**. The make-span is 222. Moreover, the example explained above under four constraints as defined follows:

Constrain 1: Each sub-operation only needs to be processed once at one particular machine.
Constrain 2: Each machine only can process one sub-operation at once.
Constrain 3: Each job has one specific order to process n suboperations.
Constrain 4: The set-up times and transmission times are negligible.

Job order table

6	5	4	3	2	1
3	6	2	1	4	5
2	1	4	3	6	5
2	1	3	5	6	4
5	4	2	3	6	1
3	6	2	4	5	1
5	1	3	2	4	6
4	6	3	1	2	5

Processing time table

15	26	18	11	25	12
24	12	23	12	28	13
22	29	12	27	20	15
27	26	13	21	15	29
11	29	21	12	24	18
26	17	19	16	27	28
24	29	18	27	14	23
14	12	18	24	17	22

Fig. 2. An 8×6 JSSP instance ($n = 8, m = 6$). The left is the job order table, while the right is corresponding requiring processing time table.

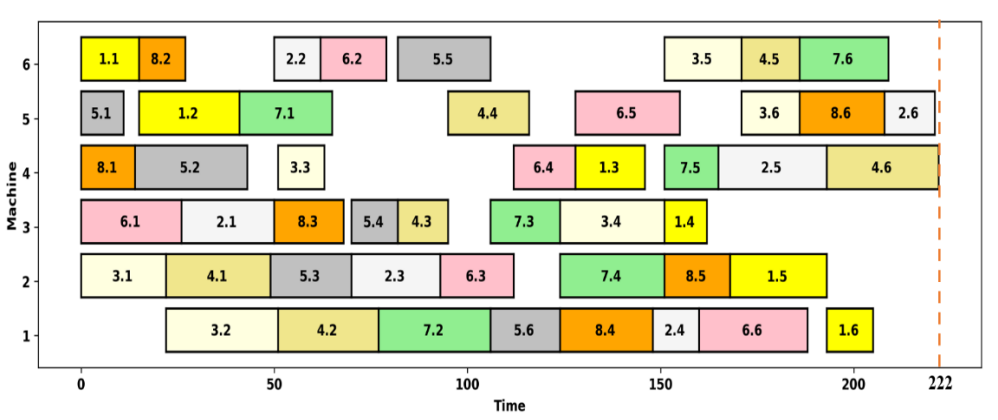

Fig. 3. The solution using GA for 8×6 JSSP instance ($n = 8, m = 6$) with Gantt Chart.

The minimum make-span of a JSSP instance could be calculated by minimizing the maximum ending time C_{max} among all machines. Thus, our objective could be written as (1).

$$Objective = min\ (C_{max}) \tag{1}$$

2.2 LSTM

LSTM is a special kind of recurrent neural network (RNN) [30] developed to solve long-time dependency problems caused by conventional RNN through three gate-like components: input

gate i_t, forgot gate f_t, and output gate o_t, as shown in **Fig. 4**. It has been utilized for speech recognition [31], power forecasting [30], and mixed modulation recognition [32] due to its excellent memory function. Especially, forgot gate decides what information should be deleted from previous hidden output h_{t-1} by using (2). Where σ is the sigmoid function, w_f is weight, x_t is the input value at time t, b_f is the bias vector. On the contrary, the input information depends on the input gate i_t, as described in (3). Same as above, the output gate o_t decides which part could be output for the next step by using (4). $\bar{C}_t$ will control the middle states of the LSTM cell according to the previous hidden output h_{t-1} and current input value x_t through (5). The final states C_t of the LSTM cell is controlled by the blue belt, which integrates the new and useful information and deletes some old and noise meaningless information, as described in (6). Where $*$ is element-wise operation. Moreover, the hidden output could obtain from (7) for the next LSTM cell. In the real application, one LSTM layer consists of several LSTM cells connected in a chain-like mode. Using those LSTM layers, we could extract rich hidden features with long-time dependency for forecasting. It is very suitable for JSSP because it is an ongoing problem and is influenced by the time steps.

$$f_t = \sigma\left(w_f h_{t-1} + w_f x_t + b_f\right) \tag{2}$$

$$i_t = \sigma(w_i h_{t-1} + w_i x_t + b_i) \tag{3}$$

$$o_t = \sigma(w_o h_{t-1} + w_o x_t + b_o) \tag{4}$$

$$\bar{C}_t = tanh\,(w_c h_{t-1} + w_c x_t + b_c) \tag{5}$$

$$C_t = f_t * C_{t-1} + i_t * \bar{C}_t \tag{6}$$

$$h_t = o_t * tanh\,(C_t) \tag{7}$$

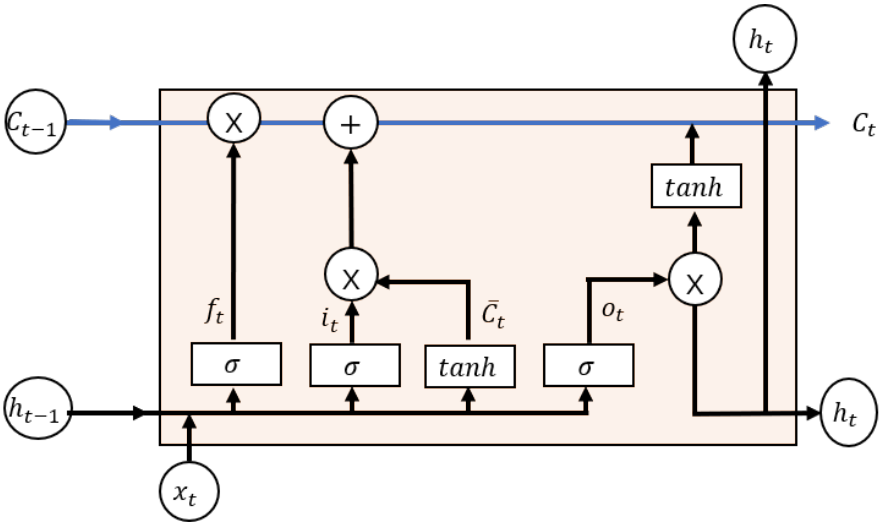

Fig. 4. The structure of LSTM, including three gates: input gate i_t, output gate o_t, and forgot gate f_t. The control belt marked with blue colour controls the whole information flow.

3. The Proposed SS-LSTM for JSSP

The proposed SS-LSTM has dual channels: detailed-level and system-level channels, as shown in **Fig. 5**. The detailed-level channel records 18 detailed production states, while the system-level channel is for system-level states. Furthermore, it consists of five steps: constructing the input sates, detailed-level feature extraction, system-level feature construction, feature fusion and self-supervised learning, output the targe and update the network, respectively. A detailed description of each part is introduced in the following section.

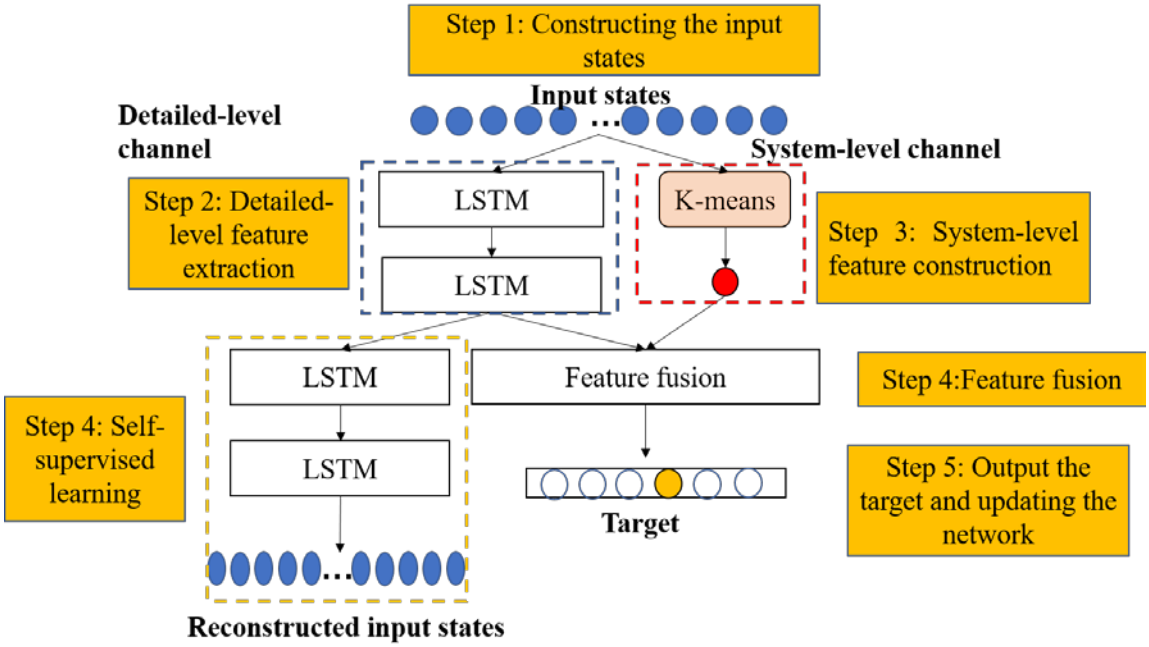

Fig. 5. The proposed SS-LSTM structure for solving JSSP.

3.1 Constructing the Input States

As introduced in **Fig. 1**, the first step for JSSP is using deep learning technology to generate solutions with some optimal solutions. In the proposed framework, we utilized GA to generate the solutions and corresponding training samples, including input features and labels (corresponding to Step 2 in **Fig. 1**). The proposed SS-LSTM defined 18 productions states to fully reflect the detail-level states when generate the solution using GA, as shown in **Table 1**. The corresponding label is generated by using algorithm 1, which selects the machine's priority as the label. By using algorithm 1, one $J \times M$ JSSP instance would be converted into an M-classification problem.

Furthermore, the input is formalized as (8) for continues analysis. Notice, one sample from matrix $Input$ corresponds to one label.

$$Input = [\frac{J_{phase}}{J}, \frac{k}{n}, \dots, O_{i,j}] \tag{8}$$

Table 1. Construting18 detailed-level states

Order	Description
1	The ratio of phase order $\frac{J_{phase}}{J}$, where J is the number of jobs.
2	Position order $\frac{k}{n}$.
3	Order of procedure in machine m.
4	The ratio of the procedure to all machine M: $\frac{m}{M}$.
5	The ratio of processing time to sum: $\frac{T_{im}}{sum(T)}$.
6	The ratio of processing time to the maximum: $\frac{T_{im}}{max(T)}$.
7	The ratio of processing time to minimum: $\frac{T_{im}}{min(T)}$.
8	The ratio of processing time to mean: $\frac{T_{im}}{mean(T)}$.
9	The ratio of processed J_i's processing time $T_{i,:J}$ to the total processing time of job T_i: $\frac{T_{i,:J}}{T_i}$.
10	The ratio of processing time T_{ij} to processed J_i's processing time $T_{i,:J}$: $\frac{T_{i,j}}{T_{i,:J}}$.
11	The ratio of processing time T_{ij} to total J_i's processing time T_i: $\frac{T_{i,j}}{T_i}$.
12	The ratio of processing time T_{ij} to unfinished J_i's processing time $T_{i,j:}$: $\frac{T_{i,j}}{T_{i,j:}}$.
13	The ratio of job index j to job number J: $\frac{j}{J}$.
14	The ratio of machine m to machine number M: $\frac{m}{M}$.
15	The processing time T_{ij} to machine m's processing time $T_{:,m}$: $\frac{T_{i,j}}{T_{:,m}}$.
16	The ratio of J_i's processing time $T_{i,:}$ to machine m's processing time $T_{:,m}$: $\frac{T_{i,:}}{T_{:,m}}$.
17	Processing time $T_{i,j}$.
18	Processing order $O_{i,j}$.

Algorithm 1: Labels generation algorithm

Input: Optimal solution matrix X, which records $J\times M$ elements of $\{Job_{id}, Job_{phase}\}$, where Job_{phase} is the job order.

Output: Sub-classification labels label.

```
for i= 1, 2, 3, …, M do:
    for j=1, 2, 3, …, J do:
        if Job_id == X[i, j, 0] and Job_phase == X[i, j, 1]
            label.append(j)
return label
```

3.2 Detailed-level Feature Extraction

To extract the more abstract and robust hidden patterns for the next action prediction, the proposed SS-LSTM applies two stacked LSTM layer to extract rich feature representations considering the impact on time step in the detailed-level channel, as shown in (9). Where $LSTM()$ is LSTM operation. The reason for using two LSTM layers is to make a trade-off between feature extraction and saving time due to LSTM operation is time-consumption.

$$Features_{detail} = LSTM(LSTM(Input)) \tag{9}$$

3.3 System-level Feature Construction

The previous step has extracted rich detailed-level features. The proposed method adopts the Kmeans algorithm to identify the system-level feature representations to enhance the feature representative capability again, as defined in (10). Where the input samples will be divided into several classes to reflect the system-level production states, $Kmeans(\ )$ is the k-means algorithm.

$$Features_{system} = Kmeans(Input) \tag{10}$$

3.4 Feature Fusion and Self-supervised Learning

To get the fusion features of detailed-level features and system-level features, this manuscript adopts one concatenate layer to combine them, as described in (11). Therefore, the features include the detailed-level and system-level states hidden patterns considering the influence of time steps.

$$Features = Concatenate(Features_{detail}, Features_{system}) \tag{11}$$

Meanwhile, the proposed method adopts self-supervised LSTM to keep high feature extraction capacity simultaneously ensuring the reliable feature representative ability. Two symmetrical LSTM layers are utilized to reconstruct the input states, which is formalized as:

$$Input_{rec} = LSTM(LSTM(Features_{detail})) \tag{12}$$

3.5 Output and Updating

The proposed SS-LSTM has two outputs: reconstructed input states as described in (12) and target machine priority as follows:

$$Target = dense(Features) \tag{13}$$

Where one dense layer with M nodes is employed to predict machine priority, the highest will be selected as the predicting label.

Corresponding to the two outputs we defined in the SS-LSTM, the proposed method calculates two loss functions to update the parameters of hidden layers in the SS-LSTM. One is the mean square error (MSE) for reconstructing input states. Another one is categorical cross-entropy (CCE) for outputting target. The proposed method employs α, β to combine them as follows:

$$loss = \alpha MSE + \beta CCE \tag{14}$$

Where $\alpha + \beta = 1$. By minimizing the loss, the hidden layers of SS-LSTM are updated. Moreover, the influence of α, β, will be discussed in the next section. After obtaining the subclassification results, the converting algorithm will be utilized to convert the classification problem to the JSSP problem, which is similar to the reference [7].

Moreover, solving the JSSP using the proposed SS-LSTM could be written as (15). The new JSSP instance is converted into detailed-level states $Input'$, and system-level states $Kmeans(Input')$; $SS-LSTM()$ are the trained model parameters; The target $JSSP_{target}$ will be converted into JSSP by using converting algorithm.

$$JSSP_{target} = SS - LSTM(Input', Kmeans(Input')) \quad (15)$$

4 Experimental Verification

To validate the effectiveness of the proposed method for complex JSSP. We implement the proposed SS-LSTM based on the operating system of ubuntu 16.04.3 with 23GB memory at a speed rate of 3.6 GHz, TensorFlow backend Keras for several complex JSSP instances.

4.1 Modeling

This section gives one detailed explanation about how to use the proposed SS-LSTM to process the JSSP instance from scratch, including the configuration, data generation, training the model, and system-level states identification.

4.1.1 Configuration

This manuscript adopted LSTM to solve JSSP, which has some hyperparameters that need to be defined, including input nodes, hidden nodes, activation function, loss function, and optimizer. For the loss function, we have defined MSE for target output and CCE for input reconstructing output, as defined in (14). For others, we adopted "Adam" as the optimizer to find the best convergence path. Rectified linear unit (ReLu) is selected as the activation function except for two output layers are "linear" and "softmax." The detailed configurations using an example of ten machines, as shown in **Fig. 6**. The left input layer corresponds to the detailed-level channel, while the right one is the system-level channel. Moreover, the LSTM layers' nodes are 64 and 128, used to extract the hidden patterns from the detailed-level channel for next-action prediction. The "RepeatVector" layer is used to reconstruct the detailed-level states, and the "Concatenate" layer is for fusing the detailed-level and system-level features, respectively. Moreover, the nodes number of the target output (right Dense layer) is set as ten as having ten machines.

4.1.2 Data Generation

The authors randomly generate 246 10×10 JSSP instances whose processing time ranges from 10 to 100. We utilize GA to solve those instances and calculate its detailed-level input states as defined in **Table 1**. Moreover, algorithm 1 is applied to create corresponding labels. At last, it obtained 24600 samples. For other kinds of JSSP instances, by simply changing the job numbers n and machine numbers m.

4.1.3 Training the Model

This paper adopted an early stop strategy to train the model. Specifically, splitting the generated samples into two parts: 80% training samples are used to train the model; The rest 20% validation samples is for finding the best model within given 150 epochs using the early stop strategy with patience 20. If the loss does not decrease for twenty steps in a row, the training process will be the end, and the epoch with the lowest loss will be saved as the trained model; Else will not stop until up to the 150 epochs. Shao et al. [33] also adopted this method to find the best model for power forecasting.

4.1.4 System-level states identification

The proposed method adopts the K-means algorithm to identify system-level production states. This manuscript specifies cluster numbers to be 1-30. **Fig. 7** shows the values of the within-cluster sum of errors (WCSS) [30] on the training set. It is clear to select 13 as cluster numbers due to changing in WCSS begins to level off at cluster 13.

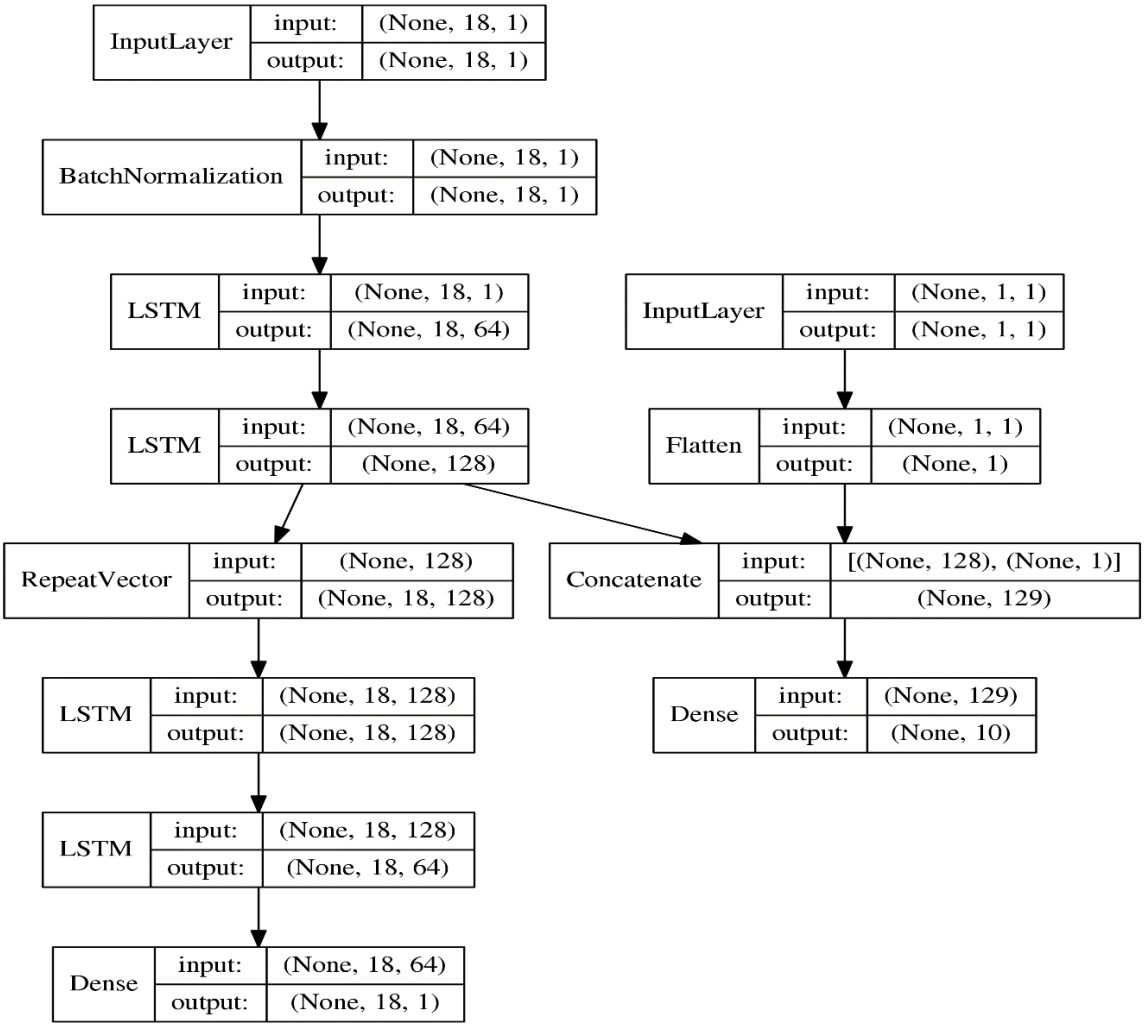

Fig. 6. The detailed configuration of the proposed SS-LSTM for ten machines JSSP instance.

4.2 The influence of α, β

After getting the two-channel inputs, the authors train the model with different α, β on one famous JSSP data set-ft10 [34], a 10×10 JSSP instance to find the best trade-off between them. The results indicate that α 0.4 and β 0.6 performs best with the make-span of 1613, which could be found in **Table 2**. Moreover, the smaller α is better, as conducted from **Fig. 8**. The sequential analysis is based on α 0.4 and β 0.6 as they perform the best.

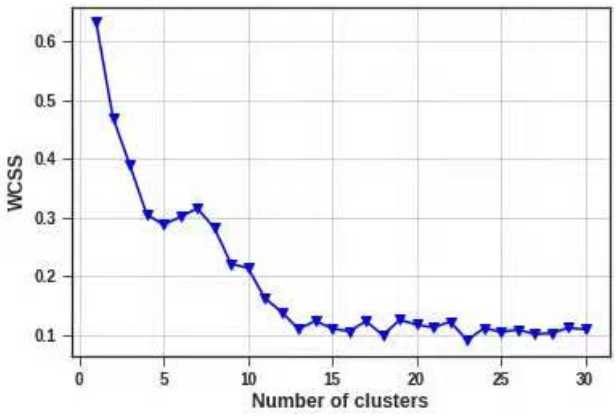

Fig. 7. The WCSS values of different clusters.

Table 2. The results on ft10 using different α, β

α (MSE)	β (CCE)	Make-span
0.1	0.9	1853
0.2	0.8	1788
0.3	0.7	1816
0.4	0.6	1613
0.5	0.5	2037
0.6	0.4	1832
0.7	0.3	1932
0.8	0.2	1901
0.9	0.1	1896

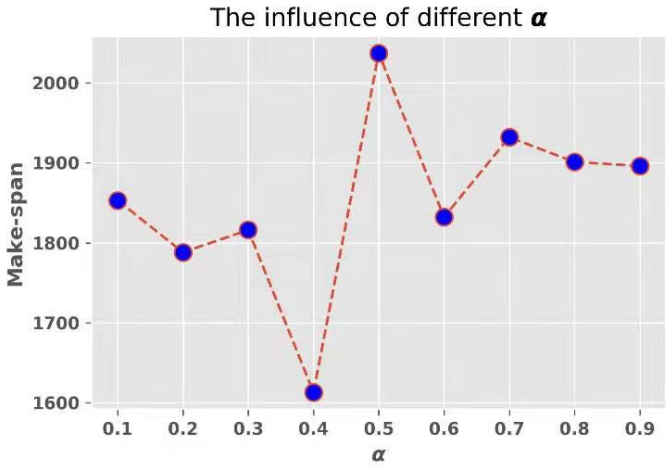

Fig. 8. The influence of different α on ft10.

4.3 Comparative Analysis

To validate the effectiveness of the proposed SS-LSTM for JSSP. The authors compared it with some leading methods, including the deep learning-based method hybrid deep neural network (HDNN) proposed by Zang et al. [7], in which they already proved that HDNN outperforms others such as shallow NN, SVM, and RF. Therefore, this manuscript only compared the proposed SS-LSTM with HDNN. Moreover, we compared the proposed SS-LSTM with some DRL-based methods, including classical DRL [25], CNN-DRL1 [26], more complex CNN-DRL2 [27]. All the above comparative are carried out based on two JSSP instances, including ft10 and ft20 [34], and each DRL-based method run 20,000 epochs to find the best solution. We adopted the original code of HDNN from the website github.com/zangzelin/HDNNMv2.0. Moreover, we adopted their deep learning part for the DRL-based methods as the structure, and reinforcement parts are the same. The deep learning part of each DRL-based method, as described in **Table 3**. Where the term "Conv1D", "Maxpool1D" represent 1-D convolutional and max-pooling operations, n is the job number. The results using make-span, as shown in **Table 4**. The findings indicated that only the proposed method and HDNN could find the solution for both two JSSP instances. However, DNN-DRL cannot find the solutions; CNN-DRL1 and CNN-DRL2 only find the solution for ft10. Those evidence has proven that the deep learning-based method is more effective than the DRL-based method for complex JSSP instance when we consider the limitations of the hardware.

We further compare the proposed SS-LSTM with HDNN on different complicated JSSP instances, including la24 and la36 [35]. The findings indicate that the proposed method wins three times over four JSSP instances, except for HDNN [7] performs a little better on la36, as shown in **Fig. 9**. Moreover, the proposed method does not require two-dimensional transformation, while HDNN needs. Thereby it saves lots of computing resources. The comparative analysis has confirmed that the proposed SS-LSTM could effectively solve complex JSSP; and it has good robustness for different complex JSSP.

Table 3. The configuration information of each comparative method

Method	A detailed description of each method
HDNN	Adopted author's code to run, which could be found from github.com/zangzelin/HDNNMv2.0.
DNN-RL	Input: system-level channel; The structure of deep learning part: Input-Dense(10)-Dense(20)-Dense(30)-Output(n)
CNN-DRL1	Input: system-level channel; The structure of deep learning part: Input-Conv1D(16, (2))-Output(n)
CNN-DRL2	Input: system-level channel; The structure of deep learning part: Input- Conv1D (16,(2))-Maxpool1D(2)-Dense(100)-Output(n)

Table 4. The make-span of each comparative method on ft10, ft20 ("-" means no solution)

Instance	HDNN	DNN-RL	CNN-DRL1	CNN-DRL2	Proposed
ft10 (10 × 10)	1768	-	1985	1860	1613
ft20 (20 × 5)	2015	-	-	-	2198

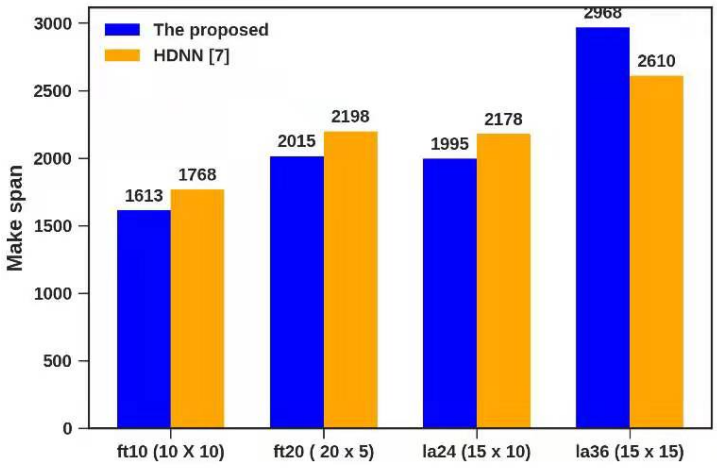

Fig. 9. The comparison results for complex JSSP.

4.4 Ablation Study

To explore each component's influence on the proposed SS-LSTM for solving complex JSSP, the authors designed three experiments. Especially designed LSTM alone to validate the system-level channel's effectiveness. Designed SS-LSTM without a self-supervised mechanism to validate its impact. All configurations are the same as SS-LSTM. The results are tested on the ft10 instance, as shown in **Table 5**. Comparing LSTM alone with SS-LSTM without a self-supervised mechanism, the system-level channel has reduced the make-span of 8.38%. Therefore, it confirmed the effectiveness of the system-level channel in the proposed method. Moreover, the supervised mechanism application has improved 11.18% performance by comparing it with the proposed SS-LSTM. The findings have confirmed the effectiveness of each part in the proposed SS-LSTM.

Table 5. The ablation analysis of the proposed method on ft10 instance

Method	Make-span
LSTM alone	1982
SS-LSTM without the self-supervised mechanism	1816
SS-LSTM	1613

5. Discussion

The JSSP is a very challenging problem with the continues development of the industry. Massive devices increasing the difficulty since it may cause an NP-hard problem. One example is used to explain what is JSSP, as shown in **Fig. 2** and **Fig. 3**. This paper proposed a novel and useful method named self-supervised LSTM for solving complex JSSP, as shown in **Fig. 5**. It treated JSSP as a multiple subclassification problem, the whole workflow, including six steps, as described in **Fig. 1**. The critical step is to build one powerful deep model to accurately extract rich hidden patterns to predict the next action. Moreover, JSSP is one ongoing problem. Therefore, the proposed method applied dual channels based on LSTM to fully extract hidden patterns of production states due to LSTM could address the long-time dependency problem. Significantly, the detailed-level channel is for extracting detailed production states features, while the system-level channel is for system-level feature extraction. Furthermore, to make a

trade-off between classification and feature extraction, the self-supervised mechanism is employed.

The comparative analysis has confirmed the proposed SS-LSTM's effectiveness for solving complex JSSP, which could be conducted from **Table 4** and **Fig. 9**. Moreover, **Table 4** has proven that the deep learning-based methods are more effective than the DRL-based methods for complex JSSP instance when considering the hardware limitations.

To build the model, the authors defined 18 variables to reflect the detail-level states, as shown in **Table 1**. Also, the model adopted the K-means algorithm to identify the system-level states, WCSS is adopted to select the appropriate system-level states, as shown in **Fig. 7**.

To explore the influence of α, β, we trained the model on the ft10 JSSP instance. The results indicate that α 0.4 and β 0.6 performs best, which could be found in **Table 2**. Moreover, the smaller α is better, as conducted from **Fig. 8**.

To explore the inner working mechanism of the proposed SS-LSTM for JSSP. We did an ablation study with three experiments on the ft10 JSSP instance. Significantly, the system-level channel reduced the make-span by 8.38%, and the supervised mechanism improved the performance by 11.18%, respectively. They could be conducted in **Table 5**. In summary, the proposed SS-LSTM could effectively and accurately solve complex JSSP.

However, same to other deep learning models, the proposed SS-LSTM for JSSP needs to set the hyperparameters based on our experience, which limits the accuracy's improvement. Moreover, the proposed method's accuracy depends on the optimal solution, the accuracy still can be improved by using appreciate optimal method.

6. Conclusion

This manuscript has proposed a novel deep learning-based framework named SS-LSTM for solving complex JSSP. The comparative analysis has confirmed its effectiveness and robustness. In the proposed SS-LSTM, dual channels are utilized to extract rich hidden patterns from detail-level and system-level channels to reflect production environments fully. The detail-level channel defined 18 variables to stand by the detail-level states, while the K-means algorithm identifies system-level states. Moreover, adopting a self-supervised mechanism with LSTM autoencoder to keep high feature extraction capacity simultaneously ensuring the reliable feature representative ability. The ablation study has confirmed each component's effectiveness in the proposed method for complex JSSP. Significantly, the system-level channel reduced the make-span by 8.38%, and the supervised mechanism improved the performance by 11.18%, respectively. By combining those technologies properly, the proposed SS-LSTM could accurately process complex JSSP.

As discussed in the discussion part, using proper hyperparameters could improve SS-LSTM's performance. In the future, we will utilize DRL technology to find the best hyperparameters for solving complex JSSP under the proposed framework. Moreover, we will validate its generality on other kinds of time-series data sets.

Acknowledgement

This work was supported by the Technology Innovation Program 20004205, The development of smart collaboration manufacturing innovation service platform in the textile industry by producer-buyer B2B connection funded By the Ministry of Trade, Industry & Energy (MOTIE, Korea).

References

[1] Y. Fu, J. Ding, H. Wang, and J. Wang, "Two-objective stochastic flow-shop scheduling with deteriorating and learning effect in Industry 4.0-based manufacturing system," *Applied Soft Computing Journal*, vol. 68, pp. 847–855, 2019. Article (CrossRef Link)

[2] C. L. Liu, C. C. Chang, and C. J. Tseng, "Actor-critic deep reinforcement learning for solving job shop scheduling problems," *IEEE Access*, vol. 8, pp. 71752–71762, 2020. Article (CrossRef Link)

[3] E. M. Frazzon, A. Albrecht, M. Pires, E. Israel, M. Kück, and M. Freitag, "Hybrid approach for the integrated scheduling of production and transport processes along supply chains," *International Journal of Production Research*, vol. 56, no. 5, pp. 2019–2035, 2018. Article (CrossRef Link)

[4] K. Wang, H. Luo, F. Liu, and X. Yue, "Permutation Flow Shop Scheduling with Batch Delivery to Multiple Customers in Supply Chains," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 48, no. 10, pp. 1826-1837, 2018. Article (CrossRef Link)

[5] H. W. Ge, L. Sun, Y. C. Liang, and F. Qian, "An effective PSO and AIS-based hybrid intelligent algorithm for job-shop scheduling," *IEEE Transactions on Systems, Man, and Cybernetics Part A: Systems and Humans*, vol. 38, no. 2, pp. 358–368, 2008. Article (CrossRef Link)

[6] G. R. Weckman, C. V. Ganduri, and D. A. Koonce, "A neural network job-shop scheduler," *Journal of Intelligent Manufacturing*, vol. 19, no. 2, pp. 191–201, 2008. Article (CrossRef Link)

[7] Z. Zang et al., "Hybrid Deep Neural Network Scheduler for Job-Shop Problem Based on Convolution Two-Dimensional Transformation," *Computational Intelligence and Neuroscience*, vol. 2019, no. ii, 2019. Article (CrossRef Link)

[8] S. Mathematics and N. May, "The Complexity of Flowshop and Jobshop Scheduling," Authors (s): M. R. Garey, D. S. Johnson and Ravi Sethi Published by: INFORMS Stable URL: http://www.jstor.org/stable/3689278 Accessed: 28-03-2016 13: 55 UTC Your use of the JSTOR archive indicat," vol. 1, no. 2, pp. 117–129, 2016. Article (CrossRef Link)

[9] I. González-Rodríguez, C. R. Vela, and J. Puente, "A genetic solution based on lexicographical goal programming for a multiobjective job shop with uncertainty," *Journal of Intelligent Manufacturing*, vol. 21, no. 1, pp. 65–73, 2010. Article (CrossRef Link)

[10] L. Asadzadeh, "A local search genetic algorithm for the job shop scheduling problem with intelligent agents," *Computers and Industrial Engineering*, vol. 85, pp. 376–383, 2015. Article (CrossRef Link)

[11] M. Kurdi, "An effective new island model genetic algorithm for job shop scheduling problem," *Computers and Operations Research*, vol. 67, pp. 132–142, 2016. Article (CrossRef Link)

[12] M. Martí and A. Maki, "A multitask deep learning model for real-time deployment in embedded systems," 2017, [Online]. Available: http://arxiv.org/abs/1711.00146.

[13] Y. Bazi and F. Melgani, "Toward an optimal SVM classification system for hyperspectral remote sensing images," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 44, no. 11, pp. 3374–3385, 2006. Article (CrossRef Link)

[14] Y. L. Pavlov, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, 2019.

[15] Y. P. S. Foo and T. Takefuji, "Integer linear programming neural networks for job-shop scheduling," in *Proc. of IEEE Inernational Conference on Neural Networks*, pp. 341–348, 1988. Article (CrossRef Link)

[16] Y. Lecun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. Article (CrossRef Link)

[17] Xiaorui Shao, C.-S. Kim, and P. Sontakke, "Accurate Deep Model for Electricity Consumption Forecasting Using Multi-Channel and Multi-Scale Feature Fusion CNN–LSTM," *Energies*, vol. 13, no. 8, p. 1881, 2020. Article (CrossRef Link)

[18] K. Kim and J. Lee, "Deep CNN based Pilot Allocation Scheme in Massive MIMO systems," *KSII Transactions on Internet and Information Systems*, vol. 14, no. 10, pp. 4214–4230, 2020. Article (CrossRef Link)

[19] H. Mobahi, R. Collobert, and J. Weston, "Deep learning from temporal coherence in video," in *Proc. of the 26th International Conference On Machine Learning*, ICML 2009, pp. 737–744, 2009. Article (CrossRef Link)
[20] F. Zhao, Y. HongDong, C. Yahong, and M. YuXuhui, "Integration of Artificial Neural Networks and Genetic Algorithm for Job-Shop Scheduling Problem," in *Proc. of International Symposium on Neural Networks-2005*, pp. 770–775, 2005.
[21] W. Zhang and T. G Diettench, "A Reinforcemen t Learnin g Approac h t o Job-shop Scheduling," in *Proc. of 14th Int. Jt Conf. Artif. Intell*, pp. 1114–1120, 1995. [Online]. Available: https://www.ijcai.org/Proceedings/95-2/Papers/013.pdf.
[22] M. E. Aydin and E. Öztemel, "Dynamic job-shop scheduling using reinforcement learning agents," *Robotics and Autonomous Systems*, vol. 33, no. 2, pp. 169–178, 2000. Article (CrossRef Link)
[23] T. Gabel and M. Riedmiller, "Adaptive Reactive Job-Shop Scheduling With Reinforcement Learning Agents," *International Journal of Information Technology and Intelligent Computing*, vol. 2, no. 4, 2008. [Online]. Available: http://ml.informatik.uni-freiburg.de/_media/publications/gr07.pdf.
[24] V. Mnih, D. Silver, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *arXiv:1312.5602 [cs. LG]*, pp. 1–9, 2013. [Online]. Available: https://arxiv.org/abs/1312.5602
[25] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. of HotNets 2016 - Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pp. 50–56, 2016. Article (CrossRef Link)
[26] W. Chen, Y. Xu, and X. Wu, "Deep Reinforcement Learning for Multi-Resource Multi-Machine Job Scheduling," *arXiv:1711.07440 [cs.DC]*, pp. 1–2, 2017. [Online]. Available: http://arxiv.org/abs/1711.07440.
[27] Y. Ye et al., "A New Approach for Resource Scheduling with Deep Reinforcement Learning," *arXiv:1806.08122 [cs.AI]*, pp. 2–6, 2018, [Online]. Available: http://arxiv.org/abs/1806.08122.
[28] C. C. Lin, D. J. Deng, Y. L. Chih, and H. T. Chiu, "Smart Manufacturing Scheduling with Edge Computing Using Multiclass Deep Q Network," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 7, pp. 4276–4284, 2019. Article (CrossRef Link)
[29] S. Hochreiter and J. Urgen Schmidhuber, "Long Short-term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 1997. Article (CrossRef Link)
[30] X. Shao and C. S. O. O. Kim, "Multi-step Short-term Power Consumption Forecasting Using Multi-Channel LSTM With Time Location Considering Customer Behavior," *IEEE Access*, vol. 8, pp. 125263-125273, 2020. Article (CrossRef Link)
[31] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A Critical Review of Recurrent Neural Networks for Sequence Learning," *arXiv:1506.00019 [cs.LG]*, pp. 1–38, 2015. [Online]. Available: https://arxiv.org/abs/1506.00019.
[32] Q. Jing, H. Wang, and L. Yang, "Study on fast-changing mixed-modulation recognition based on neural network algorithms," *KSII Transactions on Internet and Information Systems*, vol. 14, no. 12, pp. 4664–4681, 2020. Article (CrossRef Link)
[33] X. Shao, C. PU, Y. ZHANG, and C. S. KIM, "Domain Fusion CNN-LSTM for Short-Term Power Consumption Forecasting," *IEEE Access*, vol. 8, pp. 188352–188362, 2020. Article (CrossRef Link)
[34] G. L. Thompson, "Probabilistic learning combinations of local job-shop scheduling rules," *Industrial Scheduling*, vol. 3, no. 2, pp. 225–251, 1963.
[35] S. Lawrence, "An Experimental Investigation of heuristic Scheduling Techniques," *Supplement to Resource Constrained Project Scheduling*, 1984.

Xiaorui Shao received a B. S. of engineering degree in computer science and technology and a B. S. of economics degree in the economics department from the Yanbian University, Yanji, Jilin, China 2017. He received an M.S. degree in the Interdisciplinary Program of Information Systems from Pukyong National University (PKNU), Busan, Korea, in 2019. He is pursuing the Ph. D degree in the department of information system from Pukyong National University. He has won the outstanding paper reward of ICONI 2019□ 2020□ and the PKNU early career researcher award. His research interest includes the smart grid, time series forecasting, fault diagnosis, and deep learning.

Chang-Soo Kim received Ph. D. degree in the Department of Computer Engineering from Chung Ang University in 1991, Seoul, Korea. He became a professor in the Department of IT Convergence and Application Engineering, Pukyong National University, Pusan, South Korea, in 1992 and has continued until the present. He has been the Vice President of Korea Multimedia Society since 2011. His current research interests include scheduling of smart factories, big data simulation.