Piotr Świtalski[1]
ORCID: 0000-0003-0747-9766

Karolina Siwiak[2]
ORCID: 0009-0004-2908-9691

University of Siedlce
Faculty of Exact and Natural Sciences
Institute of Computer Science
ul. 3 Maja 54, 08-110 Siedlce, Poland

[1]piotr.switalski@uws.edu.pl
[2]karolina.siwiak@vp.pl

Scheduling using Convolutional Neural Network in GPU environment

DOI: 10.34739/si.2024.30.06

Abstract. Graphics processing units (GPU) have become the foundation of artificial intelligence. Machine learning was slow, inaccurate, and inadequate for many of today's applications. The inclusion and utilization of GPUs made a remarkable difference in large neural networks. The numerous core processors on a GPU allow machine learning engineers to train complex models using many files relatively quickly. The ability to rapidly perform multiple computations in parallel is what makes them so effective; with a powerful processor, the model can make statistical predictions about very large amounts of data. GPUs are widely used in machine learning because they offer more power and speed than CPUs. In this paper, we show the use of GPU for solving a scheduling problem. The results show that this idea is useful, especially for large optimization problems.

Keywords: Job Shop Scheduling Problem, Convolutional Neural Network, optimization, genetic algorithm

1 Introduction

Machine learning on GPUs has become a popular topic in the field of data science. GPUs are specialized hardware that can perform parallel computations much faster than CPUs. This makes them ideal for training machine learning models, which often require a lot of computational power. According to NVIDIA [1], GPUs can help train machine learning models faster and more efficiently. The article also mentions that the RAPIDS ecosystem provides a set of GPU-accelerated machine learning algorithms that can be used to solve various problems in data science. GPUs are ideal for machine learning because they have enhanced mathematical computation capabilities. They can help train more models, larger models, and more complex models in much shorter times than CPUs.

Processing the acquired data has become another challenge to overcome. The amount of data generated globally per day is about 2.5 quintillion bytes (quintillion = 10^{30}) [6] - the above numbers will change significantly if the Internet of Things (IoT) spreads around the world; for example, smart buildings are an example of IoT. "With big data, speed or speed really matters. (...) In many real-world applications, we need to complete a task within a certain time; otherwise, the processing results become less valuable or even worthless, such as earthquake prediction, stock market prediction, agent-based autonomous exchange systems (buying/selling), and so on" [4]. There is a great demand for equipment that can process large data sets in a maximum of a few minutes; personal computers are not able to achieve such results, which is why parallel programming is becoming more and more popular, which can take place, for example, on many processors or graphics cards. Based on the presented problems, this work was created. Processing large amounts of data in a short time is a significant challenge today. The training of the network itself and its use require large hardware resources. Therefore, it becomes realistic to use specialized computing units (e.g. contained in graphics cards) to create extensive neural networks.

Task scheduling is an important aspect of cloud computing systems. Although there are many approaches to improving task scheduling, this is still an open issue. A proposed framework suggests [11] optimizing the utilization of cloud computing resources by using machine learning techniques. The framework uses a dynamic scenario and considers different parameters for scheduling purposes such as Makespan, QoS, energy consumption, execution time, and load balancing. Incoming task requests are classified using supervised machine learning techniques to select the best suitable algorithm for the task request rather than randomly assigning the scheduling algorithm. The outcome of the proposed work leads to the selection of the best task scheduling algorithm for the input task (request).

In this paper, we will show the Job Shop Scheduling Problem (JSSP), which is solved by a Python program using three libraries: TensorFlow, PyTorch, and TensorRT.

The assumptions of this work are as follows.

- a neural network will solve an example of a scheduling problem in a shorter time than a genetic algorithm;

- the neural network, processing calculations on the graphics card, will predict large inputs faster than the main processor;
- GPUs with higher capabilities (number of compute units, threads, and processor clock speed) will not significantly affect the prediction time of small inputs compared to GPUs with lower parameters.

2 Scheduling problem

The Job Shop Scheduling Problem (JSSP) is a well-known optimization problem in computer science [2]. It is a variant of job scheduling. In this problem we consider a set of independent $n \in \mathbb{N}$ jobs $J = \{1, 2, \ldots n\}$, and factory which has $m \in \mathbb{N}$ machines $M = \{1, 2, \ldots m\}$. In the specific variant, each job consists of a set of operations (tasks) $O = \{1, 2, \ldots, m\}$ with p_m processing times. There is a sequence of operations in job j. The single o_{ji} operation of the job j must be executed on the machine i. This assignment needs $t_{ji} \in \mathbb{N}$ time units for completion. We assume that operation is performed on machine without interruption during execution. Operations can be run on m machines in a different order. The main purpose of the problem is to find the best solution, a schedule consisting of assigning all n jobs to m machines fulfilling the optimization criterion(s). Generally, we are trying to minimize the makespan – the total length of the schedule (that is, when all the jobs have finished processing). We accept only feasible solutions which must meet the following conditions:

- no operation for a job can be started until the previous operation for that job is completed; precedence constraints must be respected,
- a machine can only work on one operation at a time,
- an operation, once started, must be executed completely.

This problem is one of the best known NP-hard problems. The size of the search space (a number of schedules) $\mathbb{Z}$ is directly dependent on the number of jobs n and a number of machines m: $\mathbb{Z} = (n!)^m$. For $n = 2$ jobs and $m = 4$ machines we have $(2!)^4 = 16$ possible solutions (schedules), where an instance which consists $n = 5$ jobs and $m = 5$ machines produces 207 360 000 solutions.

Suppose we have a simple job shop problem with three jobs and three machines. Each operation is labeled by a pair of numbers (m, p) where m is the number of the machine the operation must be processed on and p is the processing time of the operation. The jobs are as follows:

- job 0 = [(0, 3), (1, 2), (2, 2)]
- job 1 = [(0, 2), (2, 1), (1, 4)]
- job 2 = [(1, 4), (2, 3)]

In this example, job 0 has three tasks. The first operation (0, 3), must be processed on machine 0 in 3 units of time. The second operation, (1, 2), must be processed on machine 1 in 2 units of time, and so on.

The goal is to schedule the tasks on the machines so as to minimize length of the schedule – the time it takes for all the jobs to be completed. The objective of the JSSP is to find the

correct permutation of all operations where the time is minimized. In this case we minimize the makespan $C_{\max}$. Makespan is defined below:

$$C_{\max} = max_i \left(O \left(S_i \right) \right). \tag{1}$$

Let us denote by S as a schedule. By S_i we denote the schedule on machine M. The completion time of the operations on machine M_m in schedule S_i is denoted by $O(S_i)$. We consider minimizing the maximum completion time on each machine M_m across the system.

3 Solving the scheduling problem by neural networks

Traditionally, optimization problems involve finding the maximum or minimum value of a function subject to certain constraints. Often for this optimization, metaheuristic algorithms are used. There are many types of metaheuristics, such as evolutionary algorithms, swarm intelligence, simulated annealing, tabu search, etc. In this approach, we should define the objective function and the constraints of the problem. The objective function is the function that we want to maximize or minimize, and the constraints are the conditions that limit the feasible solutions. The metaheuristic algorithm can generate a set of solutions that approximate the optimal solution of the problem. We can compare the quality of the solutions using the objective function value and other metrics. Undoubtedly, metaheuristics have many advantages, such as efficiency (can find an optimal or near-optimal solution in a reasonable amount of time), flexibility (they are not problem specific), and ability to escape local minima. However, there are some disadvantages: stochastic nature of the algorithm, need of parameter tuning, or no guarantee of optimality. In such scheduling problems, we have one more disadvantage of this approach. The time to calculate the reasonable solution can be unacceptable, when we need a response from the system in a very short time. To solve this problem we can use trained model of neural network, which could return the solution (schedule) in incomparably shorter time. Our solution is based on this concept. To build the scheduler based on neural network, we used Convolutional Neural Network (CNN). CNN is a type of feedforward neural network that is widely used in image and video recognition, natural language processing, and other applications. CNNs are designed to take input data in the form of images and process the data through multiple layers, each of which applies a different set of filters to the data to extract different features. The filters in each layer are optimized during training to recognize specific patterns in the input data. CNNs are distinguished from other neural networks by their superior performance with input of image, speech, or audio signals. They have three main types of layers: Convolutional layer, Pooling layer, and Fully-connected (FC) layer. The convolutional layer is the first layer of a convolutional network. Convolutional neural networks are variants of multilayer perceptron, designed to emulate the behavior of a visual cortex. These models mitigate the challenges posed by the MLP architecture by exploiting the strong spatially local correlation present in natural images. The design of convolutional neural networks was inspired by the structure of the visual cortex (the part of the brain that receives visual stimuli). It allows images to be processed by many cells in the cerebral cortex, which detect light in areas that overlap the visual field [9]. The mechanism of convolution consists in transforming fragments of a photo using a matrix (kernel) in order to extract information about its specific features. Acquired information allows you to recognize or classify patterns. Compared to the

general assumptions of deep networks, convolutional networks learn through local patterns, not global ones - this allows for greater manipulation of the training and test set data - images can be "twisted" and the model will correctly recognize the object in the photograph anyway, because patterns in the convolutional network are resistant to shifts. There are several variants of CNN architectures. However, the concept of all CNNs is similar. This architecture consists of three types of layers: convolutional, pooling, and fully-connected (FC) layers. The typical architecture of CNN is presented in Fig. 1

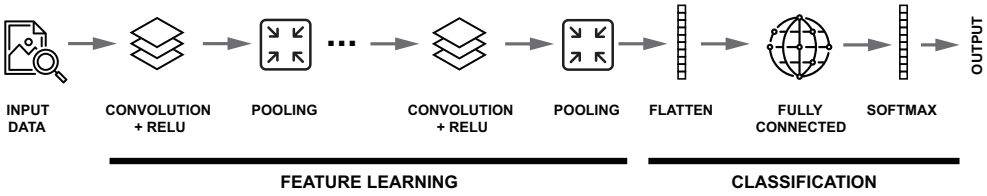

Figure 1. Typical architecture of CNN. Source: Introduction to Convolutional Neural Networks. Stanford University, 2018.

CNNs typically consist of several convolutional layers and subsampling layers. They are also called pooling layers, followed by at least one fully connected layer [11]. The first two layers, convolutional and pooling, perform feature extraction, while the third, fully connected layer, maps the extracted features to a final result, such as classification [10]. The layers thus connected form a multilayer perceptron whose input is tensors: they are matrices with more dimensions than just height and width. Input tensors in the case of convolutional networks are referred to as feature maps, and they are usually three-dimensional: dimensions specify such quantities as height, width and number of colour channels of the examined images; we are talking about shades of grey or RGB channels.

3.1 State of the art

Some papers describing this idea. In the paper [7] authors have proposed a framework to learn to schedule a JSSP using a Graph Neural Network (GNN) and Reinforcement Learning (RL). They formulated the scheduling process of JSSP as a sequential decision-making problem with graph representation of the state to consider the structure of JSSP. The proposed framework employs a GNN to learn node features that embed the spatial structure of the JSSP represented as a graph (representation learning) and derive the optimum scheduling policy that maps the embedded node features to the best scheduling action (policy learning). They used an RL strategy to train these two modules end-to-end. The GNN scheduler, due to its superb generalization capability, outperforms practically favored dispatching rules and RL-based schedulers on various benchmark JSSP.

Another concept is proposed in [8]. The authors show new heuristic algorithms that incorporate techniques from the machine learning field and scheduling theory to solve a hard single machine scheduling problem. These heuristics transform an instance of a hard problem into an instance of a simpler one solved to optimality. Computational experiments show that they are competitive with state-of-the-art heuristics, notably in large instances. An alternative

example is the use of machine learning and optimization for production rescheduling in Industry 4.0 [5]. Scholars have proposed a rescheduling framework that integrates machine learning techniques and optimization algorithms to find a trade-off between the frequency of rescheduling and the growing accumulation of delays. They modeled a flexible job-shop scheduling problem with sequence-dependent setup and limited dual resources (FJSP) inspired by an industrial application and solved it through a hybrid metaheuristic approach. They also trained a machine learning classification model to identify rescheduling patterns and compared its rescheduling performance with periodical rescheduling approaches.

Deep learning is used to solve the scheduling problems. In the paper [13] the authors propose a novel method to solve job-shop scheduling problems (JSSPs) using a deep neural network. The paper introduces two main innovations:

- A two-dimensional convolution transformation (CTDT) that converts the irregular data from the scheduling problem into regular data that can be processed by a convolutional neural network (CNN).
- A hybrid deep neural network structure that combines a CNN and a fully connected network to extract features and learn scheduling rules from the data.

The authors claim that the proposed method, called HDNNS, can achieve better performance and generalization than existing methods, such as genetic algorithms, branch and bound methods, and artificial neural networks.

4 Proposed solution

The idea of JSSP solving is based on the fact that we can collect examples of solutions to the JSSP problem (e.g. generated by a metaheuristic algorithm) and then use them to learn a neural network. At that point, the trained neural network will be able to independently generate solutions to the JSSP problem given at the input. We use CNN as neural network.

The scheduling system is divided into two parts. The first part is responsible (see the left side of Fig. 2) for learning CNN model. We generate the sub-problems using genetic algorithm (GA) and then train the CNN model using data generated by GA. Afterwards, this model is used to generate the solution for provided (real) JSSP (see right side of Fig. 2).

The data generated by GA should consist useful information to the scheduler used in the execution phase. Due to the many variants scheduling instances, we need to provide the following input parameters for GA:

- number of machines m,
- number of tasks n,
- minimum value of the task execution time on the machine in seconds $time\_low$,
- maximum value of the time it takes to execute a task on the machine in seconds $time\_high$.

On the basis of the above parameters, an initial population is defined. Due to the nature of CNN, we need to create the corresponding input dataset of the neural network. GA output (schedules) is converted to a form of matrices [13]:

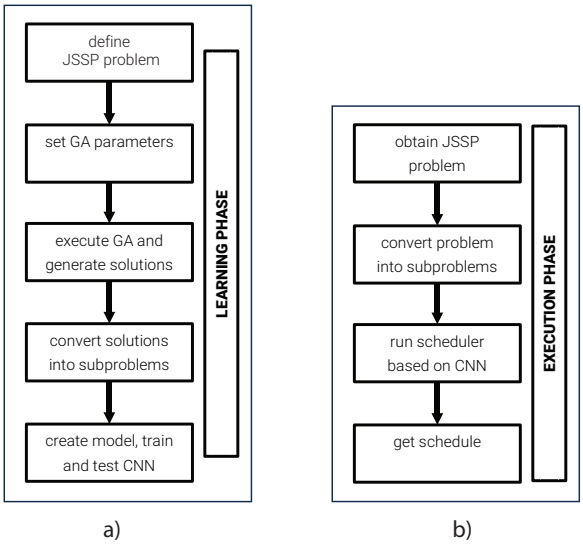

Figure 2. Learning (a) and execution (b) phases in the scheduling system. Source: own study.

- matrix p with consisting of m columns and n rows, represents the processing time of the task on each machine,
- matrix r with consisting of m columns and n rows, is used to determine the order of the task on each machine.

In the next step, we need to determine the parameters of the genetic algorithm: population size, number of generations, probability of crossover, and mutation. As a result of using the genetic algorithm, we obtain the following results:

- matrix h for the start times of each operation on the machine,
- matrix e of completion times of each operation on the machine,
- matrix x containing the order in which operations are performed on each machine.

In the next stage, sub-problems are generated: each JSSP problem is divided into several subproblems, described as operation processing features (see Tab. 1) and priority in the machine. We create a list of subproblems consisting of $m \times n$ subproblems (a subproblem is generated for each operation item on each machine). As a result of these operations, each subproblem receives its own label and two matrices of features (the first one-dimensional, the second has two dimensions). The feature matrices of all subproblems and the priority list (one element from each subproblem) will be saved to the appropriate files – the matrixes will be used as data for training the neural network, and the priority list will be a list of training labels [13].

Each subproblem is generated on the basis of the following parameters:

- label – a number representing the order on the machine in the range $\langle 0;\ n-1 \rangle$;

– list of task characteristics on a particular machine;
– two-dimensional matrix of task features on a specific machine. The list of task features is presented in Tab. 1.

Table 1. List of the task features. Source: own study.

No.	Variable name	Variable description	Equation
1	input001	The ratio of machine number to the number of all tasks	$\frac{i}{n(N)}, \underline{i} \in \underline{M}$
2	input002	The ratio of processing time p jobs on the machine to the sum of all job processing times on all machines	$\frac{p_{ij}}{\sum_{i \in M,\ j \in N} p_{ij}}$
3	input003	The ratio of the sum of processing time p for jobs i_1 on machine j_1 before it is executed to the sum of all processing times for job j_1	$\frac{\sum_{j < j_1} p_{i_1 j}}{\sum_{i \in M} p_{i j_1}}$
4	input004	The ratio of the the sum of processing time p for jobs i_1 on machine j_1 after it is executed to the sum of all processing times for job j_1	$\frac{\sum_{j \geq j_1} p_{i_1 j}}{\sum_{i \in M} p_{i j_1}}$
5	input005	The ratio of the order k on machine-1 to the number of all machines	$\frac{k-1}{n(M)}, \underline{k} \in \{-1, \ldots \underline{n(N)} - 1\}$
6	input006	The ratio of the task number to the number of all tasks	$\frac{i}{n(M)}, \underline{i} \in \underline{M}$
7	input007	The ratio of processing time p job j on machine i_1 to processing time of all jobs on machine i_1	$\frac{p_{i_1 j}}{\sum_{j \in M} p_{i_1 j}}$
8	input008	The ratio of the processing time p of job j_1 to processing time of all jobs on all machines	$\frac{p_{i j_1}}{\sum_{i \in M,\ j \in N} p_{ij}}$
9	input009	The ratio of the processing time of job j_1 on all machines to the processing time of all jobs on machine i_1	$\frac{\sum_{i \in M} p_{i j_1}}{\sum_{j \in M} p_{i_1 j}}$
10	input010	The ratio of the processing time of job j_1 on all machines to the processing time of job j_2 on all machines	$\frac{\sum_{i \in M} p_{i j_1}}{\sum_{i \in M} p_{i j_2}}$
11	input011	The ratio of the $k-1$ position of the task on the machine in order of priority to the number of total tasks $j_1 i_1$	$\frac{k l_{j_1 i_{11}} - 1}{n(N)}, \underline{n} \in \underline{J}$

From this moment on, a two-dimensional feature matrix will be created from three auxiliary matrices (matrix h, matrix e, matrix x). The first $M1$ auxiliary matrix is the sum matrix of the job processing ratio on all machines and the sum of job processing on all machines - these operations can be defined using the following formula:

$$T_{j_1,\ j_2} = \frac{\sum_{i \in M} p_{i j_1}}{\sum_{i \in M} p_{i j_2}} \tag{2}$$

where $j_1, j_2 \in$ set of tasks; M - collection of machines; p_{ij} - the processing time of job j on machine i.

This can be illustrated by the example of the problem of three tasks on three machines - $\{T_{j_0,\ j_0}, T_{j_0,\ j_1}, T_{j_0,\ j_2} \ldots T_{j_2,\ j_0}, T_{j_2,\ j_1}, T_{j_2,\ j_2}\}$. The second $M2$ auxiliary matrix contains the ratio of the processing time of a given task j on the given machine affected by the subproblem to the sum of the task processing times j of each machine. The third $M3$ auxiliary matrix

is a list of features of a task with a changed second dimension - we strive to obtain a one-dimensional matrix. On the basis of the three auxiliary matrices, six matrices will be created, written into a two-dimensional matrix of features:

- the first element - the input12 matrix - will be created by multiplying the auxiliary matrix $M2$ and the transposed matrix $M2$;
- the second element - the input13 matrix - will be created as a result of multiplying the matrix $M2$ and the transposed matrix $M1$;
- the third element - the input14 matrix - will be created as a result of multiplying the matrix $M2$ and the transposed matrix $M3$;
- the fourth element - the input15 matrix - will be formed by multiplying the matrix $M1$ and the transposed matrix $M1$;
- the fifth element - the input16 matrix - will be formed by multiplying the matrix $M1$ and the transposed matrix $M3$;
- the sixth element - the input17 matrix - will be created by multiplying the matrix $M3$ and the transposed matrix $M3$.

The list of features of the task and the six above elements of the two-dimensional matrix will be pass in into the neural network as input data - a simplified scheme of the input data of the neural network is presented in Fig. 3.

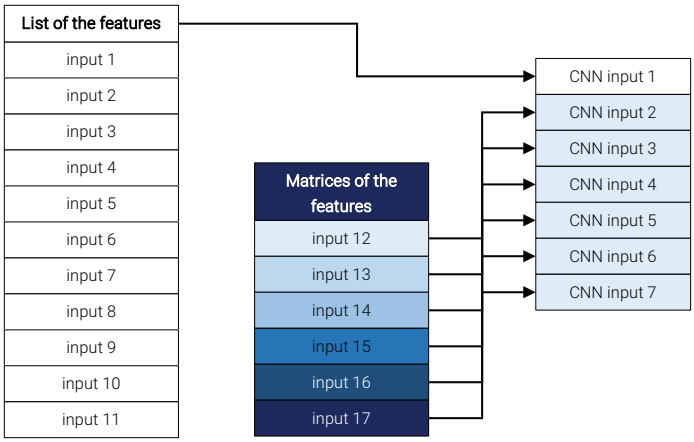

Figure 3. Neural network input. Source: own study.

5 Machine learning on GPU

There are many libraries available for machine learning on GPUs. One popular library is **cuML**, which is part of the RAPIDS ecosystem. According to a NVIDIA, cuML provides a set of GPU-accelerated machine learning algorithms that can be used to solve various problems in data science. NVIDIA also mentions that cuML integrates perfectly with cuDF, the RAPIDS DataFrame framework for processing large amounts of data on an NVIDIA GPU.

Another library that is widely used is **scikit-learn**. Scikit-learn is a popular Python library for machine learning and provides a wide range of algorithms for classification, regression, clustering, and dimensionality reduction. This library can be used with GPUs to accelerate the training of machine learning models. Other libraries that are popular for machine learning on GPUs include:

- **TensorFlow**: TensorFlow is an open source software library for dataflow and differentiable programming on a range of tasks. It was developed by the Google Brain team and is widely used for machine learning applications [14].
- **PyTorch**: PyTorch is an open source machine learning library based on the Torch library. It was developed by Facebook's AI Research lab and is known for its dynamic computational graph and ease of use [15].
- **MXNet**: MXNet is an open source deep learning framework that supports multiple programming languages, including Python, R, Julia, Scala, and C++. It was developed by Apache and is known for its scalability and efficiency [16].
- **TensorRT** is an SDK for high-performance deep learning inference. It includes a deep learning inference optimizer and runtime that delivers low latency and high throughput for inference applications. TensorRT-based applications perform up to 36X faster than CPU-only platforms during inference, allowing you to optimize neural network models trained on all major frameworks, calibrate for lower precision with high accuracy, and deploy to hyperscale data centers, embedded platforms, or automotive product platforms. TensorRT is integrated with PyTorch and TensorFlow so you can achieve 6X faster inference with a single line of code. If you are performing deep learning training in a proprietary or custom framework, you can use the TensorRT C++ API to import and accelerate your models [17].

6 Experimental results

6.1 Implementation details

The scheduling system was implemented in Python language with the use of three libraries: TensorFlow, PyTorch, and TensorRT. The experiments were carried out on two configurations of computers equipped with GPUs (see Tab. 2).

Data for training the neural network will be generated on the results of the genetic algorithm. To achieve the most optimal results of the network model inference, the training data must be of the best quality. In this section, we set an optimal parameters of GA, which will be used in subsequent studies concerning this work:

- number of generations: 200,
- population size: 80,
- probability of crossing: 0.6,
- probability of mutation: 0.05,
- parameter, correcting the number of genes, subjected to mutation: 0.05.

For this experiment we used a JSSP instance with n=10 tasks, m=10 machines, $time\_low$=10, and $time\_high$=50.

Table 2. Specification of the computer configurations. Source: own study.

Configuration name	Processor model	CPU clock speed [GHz]	Number of CPU cores	Number of CPU threads	Operating memory	GPU	Dedicated GPU memory [GB]	Operating system
CF_3090	Intel Xeon Gold 6242	2.80	16	32	64	NVIDIA GeForce RTX 3090	24	Windows 10 Pro, Ubuntu 18.04
CF_1050	Intel Core i5-4670	3.40	4	4	12	NVIDIA GeForce 1050 Ti	4	Windows 10 Pro, Ubuntu 18.04

6.2 Regular experiments

After that, we used JSSP test instances for regular experiments. These instances contain various number of jobs (from 4 to 14 jobs) and number of machines (from 4 to 14 machines). The instances are presented in Tab. 3.

Table 3. JSSP test instances used for regular experiments. Source: own study.

No.	JSSP instance name	Number of machines m	Number of tasks n	Minimum processing time on the machine $time\_low$	Maximum processing time on the machine $time\_high$
1.	SP_4_6	4	6	10	50
2.	SP_6_4	6	4	10	50
3.	SP_6_8	6	8	10	50
4.	SP_8_6	8	6	10	50
5.	SP_8_10	8	10	10	50
6.	SP_10_8	10	8	10	50
7.	SP_10_12	10	12	10	50
8.	SP_12_10	12	10	10	50
9.	SP_12_14	12	14	10	50
10.	SP_14_12	14	12	10	50

Tab. 4 presents the results of training and testing of CNN in TensorFlow and PyTorch libraries. Each model was trained with a number of epochs equal to 20 and a batch size of 64 in each epoch. Each model has been trained for a certain number of samples in one epoch - the number of samples depends on the number of machines and the number of tasks in JSSP. As we can see the times of training and testing for TensorFlow and PyTorch are much different. TensorFlow needed twice as much time as PyTorch for this process.

After that, we researched prediction times using two GPU configuration platforms implemented in TensorRT, TensorFlow, and PyTorch libraries. The results are presented in Tab. 5 and Tab. 6.

Table 4. Results of training and testing of neural network models in the TensorFlow and PyTorch library. Source: own study.

Training dataset name	Test dataset name	Number of samples in training per epoch	**TensorFlow**		**PyTorch**	
			Total training and test time [s]	Trained model file-size [kB]	Total training and test time [s]	Trained model file-size [kB]
Ztr_4_6	Zts_4_6	27	43.08	122	7.32	5503
Ztr_6_4	Zts_6_4	27	17.14	47	3.55	2095
Ztr_6_8	Zts_6_8	53	224.48	321	46.18	13598
Ztr_8_6	Zts_8_6	53	87.79	122	18.72	5503
Ztr_8_10	Zts_8_10	88	743.43	739	357.08	29643
Ztr_10_8	Zts_10_8	88	353.15	321	114.68	13598
Ztr_10_12	Zts_10_12	132	2270.43	1497	1188.68	57801
Ztr_12_10	Zts_12_10	132	1099.27	739	519.04	29643
Ztr_12_14	Zts_12_14	184	5915.52	2744	3993.7	103134
Ztr_14_12	Zts_14_12	184	3127.83	1497	1781.18	57801

Table 5. Results obtained in GPU configuration CF_3090 (NVIDIA GeForce RTX 3090). Source: own study.

JSSP instance name	Average makespan obtained by GA [ms]	**TensorRT**		**TensorFlow**		**PyTorch**	
		Average makespan obtained by CNN [s]	Average prediction time [s]	Average makespan obtained by CNN [s]	Average prediction time [s]	Average makespan obtained by CNN [s]	Average prediction time [s]
SP_4_6	237.11	561.34	0.11	551.7	0.21	446.98	0.06
SP_6_4	242.54	275.93	0.11	272.2	0.17	575.56	0.01
SP_6_8	353.52	419.95	0.11	427.42	0.19	874.61	0.02
SP_8_6	354.65	408.96	0.11	408.64	0.19	1138.35	0.01
SP_8_10	489.75	534.85	0.11	544.11	0.22	1817.25	0.03
SP_10_8	489.65	544.7	0.11	532.44	0.20	1844.77	0.02
SP_10_12	648.2	704.82	0.11	699.03	0.31	1039.18	0.06
SP_12_10	645.49	700.58	0.11	709.14	0.24	2758.43	0.04
SP_12_14	820.01	853.86	0.12	840.17	0.46	3849.3	0.11
SP_14_12	813.52	836.04	0.11	839.23	0.37	3965.91	0.07

The average prediction time for the first configuration (see Tab. 5) are the best for PyTorch. However, the average makespan obtained by this model is very high, contrary to the other libraries. TensorRT gives the best prediction taking into account the average makespan.

Table 6. Results obtained in GPU configuration CF_1050 (NVIDIA GeForce 1050 Ti). Source: own study.

JSSP instance name	Average makespan obtained by GA [ms]	**TensorRT**		**TensorFlow**		**PyTorch**	
		Average makespan obtained by CNN [s]	Average prediction time [s]	Average makespan obtained by CNN [s]	Average prediction time [s]	Average makespan obtained by CNN [s]	Average prediction time [s]
SP_4_6	233.30	562.46	0.005	551.27	0.19	442.33	0.03
SP_6_4	245.61	275.97	0.008	274.50	0.16	559.5	0.01
SP_6_8	358.53	417.83	0.008	426.83	0.23	848.68	0.03
SP_8_6	357.19	415.87	0.006	416.23	0.23	1107.87	0.01
SP_8_10	495.98	550.71	0.016	553.65	0.26	1827.5	0.06
SP_10_8	490.35	545.43	0.012	544.34	0.24	1862.42	0.04
SP_10_12	655.07	698.48	0.051	704.68	0.36	1003.7	0.14
SP_12_10	655.53	696.62	0.034	714.77	0.33	2798.59	0.09
SP_12_14	812.03	865.17	0.101	854.27	0.53	3905.15	0.25
SP_14_12	821.23	835.19	0.067	835.12	0.42	3929.63	0.17

We also researched these models on the second platform with the budget components. The results are similar to those obtained by the previous platform. PyTorch on this platform is also the worst when we get the average values of makespan. Surprisingly, the average prediction times for TensorRT are better than those of a more powerful platform (compare the values on the Tab. 5 and Tab. 6). Fig. 4 concludes the differences between the average prediction times obtained on GPU for the TensorFlow, PyTorch, and TensorRT libraries.

Similarly, we compared the average prediction times obtained by the CPU and GPU environment. Fig 5. presents the values obtained in PyTorch library, whereas Fig. 6. shows the values for TensorFlow.

The correlation between values is very high. As we can see, for more complex JSSP instances, average prediction times are higher for CPU platform.

Finally, we show the differences between the results obtained by GA and CNN. Fig. 7 shows the average values of the processing times for GA and CNN based on the TensorFlow library. Overall, these results suggest that CNN can find the solution in shorter time than GA. It is not surprising, because the trained model of neural network can give the response very shortly, opposite to the other optimization methods.

7 Conclusions

CNN is one of the many known architectures of neural networks. Our task was focused on the implementation of CNN in the scheduling aspect. The task was finished with the success. One of disadvantages of this method is the time to learn the neural network. However, one time

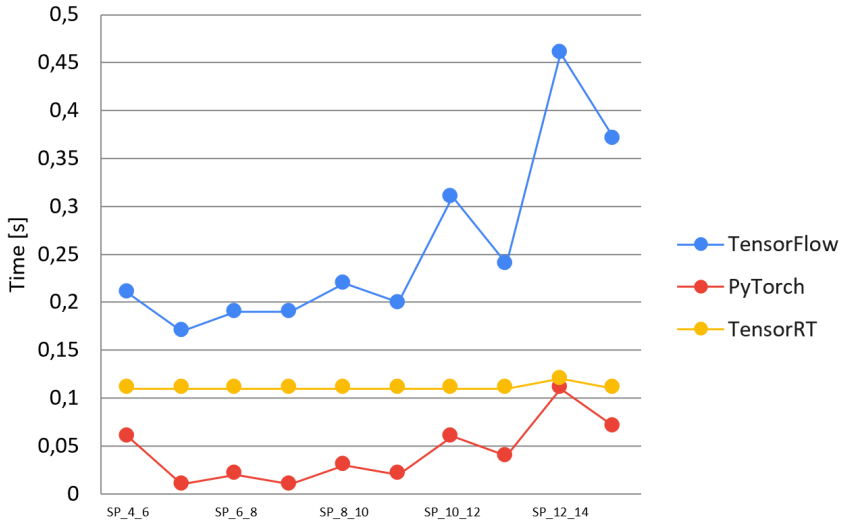

Figure 4. Average prediction times of neural networks on GPU for TensorFlow, PyTorch and TensorRT libraries. Source: own study.

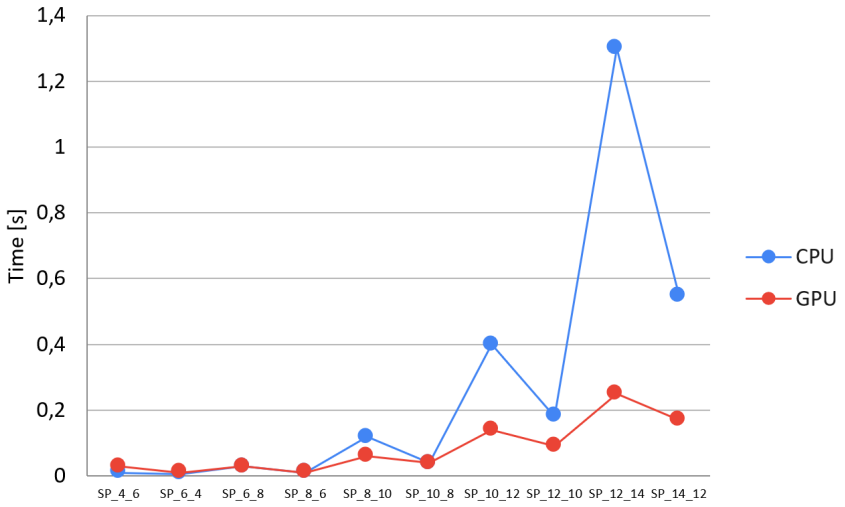

Figure 5. Average prediction times of neural networks in PyTorch library for CPU and GPU. Source: own study.

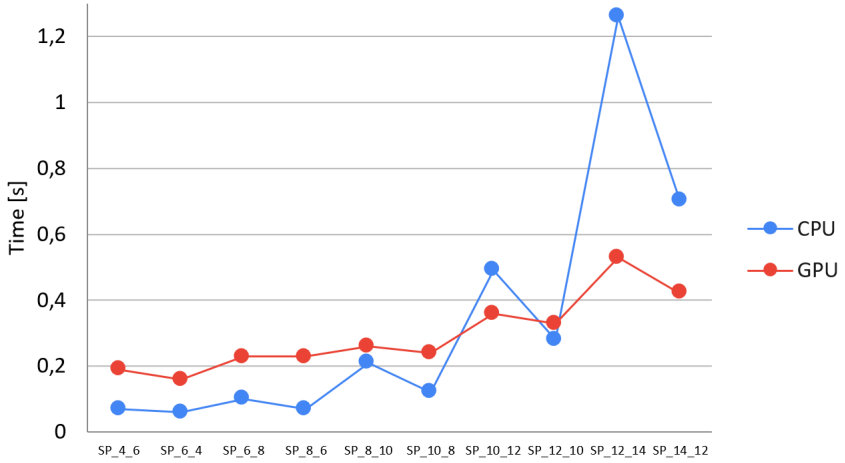

Figure 6. Average prediction times of neural networks in TensorFlow library for CPU and GPU. Source: own study.

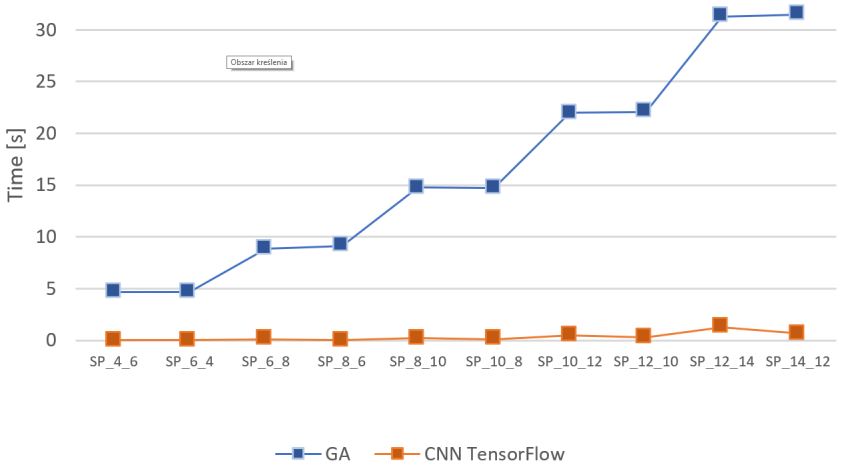

Figure 7. Average processing times for GA and CNN based on TensorFlow. Source: own study.

trained model can be useful for many scheduling problems. The results are promising. This model can be improved. We can use hybrid metaheuristics as input of the neural network. It could give more optimal results on the output of the neural network.

References

1. Drabas T.: Scikit-learn Tutorial – Beginner's Guide to GPU Accelerated ML Pipelines, available at https://developer.nvidia.com/blog/scikit-learn-tutorial-beginners-guide-to-gpu-accelerated-ml-pipelines, access time: 01.09.2023.
2. Graham, R.: Bounds for certain multiprocessing anomalies. Bell System Technical Journal. 45 (9), pp. 1563–1581. DOI: https://doi.org/10.1002/j.1538-7305.1966.tb01709.x.
3. Hearty J.: Zaawansowane uczenie maszynowe z językiem Python, Helion 2017
4. Junfei Q., Qihui W., Guoru D., Yuhua X., Shuo F.: A survey of machine learning for big data processing, EURASIP Journal on Advances in Signal Processing volume 2016.
5. Li Y., Carabelli S., Fadda E. et al. Machine learning and optimization for production rescheduling in Industry 4.0. The International Journal of Advanced Manufacturing Technology 110, 2445–2463 (2020), DOI: https://doi.org/10.1007/s00170-020-05850-5
6. Marr B.: How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read, available at https://bernardmarr.com/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read, access time: 01.09.2023.
7. Park J., Chun J., Hun Kim S., Kim Y., Park J.: Learning to schedule job-shop problems: Representation and policy learning using graph neural network and reinforcement learning, Artificial Intelligence, Multiagent Systems, DOI: https://doi.org/10.1080/00207543.2020.1870013
8. Parmentier A., T'kindt V.: Learning to solve the single machine scheduling problem with release times and sum of completion times. ArXiv 2021.
9. Raschka S., Mirjalili V.: Python : uczenie maszynowe, Helion 2019
10. Rikiya Y., Mizuho N., Gian Do Richard K., Kaori T.: Convolutional neural networks: an overview and application in radiology", Springer Nature 2018, pp. 611-629
11. Shetty C., Sarojadevi H.: Framework for Task scheduling in Cloud using Machine Learning Techniques, Fourth International Conference on Inventive Systems and Control (ICISC), Coimbatore, India, 2020, pp. 727-731, DOI: 10.1109/ICISC47916.2020.9171141.
12. Weise T., An Introduction to Optimization Algorithms. Hefei, Anhui, China: Institute of Applied Optimization (IAO), School of Artificial Intelligence and Big Data, Hefei University, 2018-2019. Available at: http://thomasweise.github.io/aitoa.
13. Zelin Z., Wanliang W., Yuhang S., Linyan L., Weikun L, Yule W., Yanwei Z.: Hybrid Deep Neural Network Scheduler for Job-Shop Problem Based on Convolution Two-Dimensional Transformation, Computational Intelligence and Neuroscience 2019, DOI: https://doi.org/10.1155/2019/7172842
14. https://www.tensorflow.org, access time: 01.09.2023.
15. https://pytorch.org, access time: 01.09.2023.
16. https://mxnet.apache.org, access time: 01.09.2023.
17. https://developer.nvidia.com/tensorrt, access time: 01.09.2023.