Machine Learning-based Scheduling and Resources Allocation in Distributed Computing

Victor Toporkov [0000–0002–1484–2255], Dmitry Yemelyanov [0000–0002–9359–8245]

and Artem Bulkhak

National Research University "MPEI", Russia
ToporkovVV@mpei.ru, YemelyanovDM@mpei.ru, BulkhakAN@mpei.ru

Abstract. In this work we study a promising approach for efficient online scheduling of job-flows in high performance and distributed parallel computing. The majority of job-flow optimization approaches, including backfilling and microscheduling, require apriori knowledge of a full job queue to make the optimization decisions. In a more general scenario when user jobs are submitted individually, the resources selection and allocation should be performed immediately in the online mode. In this work we consider a neural network prototype model trained to perform online optimization decisions based on a known optimal solution. For this purpose, we designed MLAK algorithm which implements 0-1 knapsack problem based on the apriori unknown utility function. In a dedicated simulation experiments with different utility functions MLAK provides resources selection efficiency comparable to a classical greedy algorithm.

Keywords: Resource, Scheduling, Online, Knapsack, Optimization, Neural Network, Machine Learning.

1 Introduction and Related Works

Modern high-performance distributed computing systems (HPCS), including Grid, cloud and hybrid infrastructures provide access to large amounts of resources [1, 2]. These resources typically include computing nodes, network channels, software tools and data storages, required to execute parallel jobs submitted by HPCS users.

Most HPCS and cloud solutions have requirements to provide a certain quality of services (QoS) for users' applications scheduling, execution and monitoring. Correspondingly, QoS constraints usually include a set of requirements for a coordinated resources co-allocation [3-5], as well as a number of time and cost criteria and restrictions, such as deadline, response time, total execution cost, etc. [2-7].

Some of the most important efficiency indicators of a distributed computational environment include both system resources utilization level and users' jobs time and cost execution criteria [2-4].

HPCS organization and support bring certain economical expenses: purchase and installation of machinery equipment, power supplies, user support, maintenance works, security, etc. Thus, HPCS users and service providers usually interact in economic terms, and the resources are provided for a certain payment. In such conditions, resource management and job scheduling based on the economic models is considered as an efficient way to coordinate contradictory preferences of computing system participants and stakeholders [3-7].

A metascheduler or a metabroker are considered as intermediate links between the users, local resource management and job batch processing systems [3, 4, 7, 8]. They define uniform rules for resources distribution and ensure the overall scheduling efficiency.

The most straightforward way to schedule a job-flow is by using the First-Come-First-Served (FCFS) procedure. FCFS executes jobs one by one in an order of arrival. Backfilling procedure [4, 9] makes use of advanced resources reservations in order to prevent starvation of jobs with a relatively large resource request requirements. Microscheduling [4-5, 10] approach may be added to backfilling to affect global scheduling efficiency by choosing the appropriate secondary optimization criteria.

Online scheduling, on the other hand, requires HPCS scheduler to make resources allocation and optimization decisions immediately when jobs are submitted. One possible online scheduling strategy is to perform locally efficient resources selection for each job. However, in this case the global scheduling efficiency may be degraded. CoP microscheduling strategy [4] implements a set of heuristic rules to optimize job-flow execution time based on the resource's properties: performance, cost, utilization level, etc.

The main contribution of this paper is a machine learning-based approach which can be trained on efficient scheduling results to perform online scheduling based on secondary properties of the resources. To achieve this goal, an artificial neural network was designed in combination with a dynamic programming method. We consider a general 0-1 knapsack scheduling model and evaluate algorithms efficiency in a dedicated simulation experiment.

The paper is organized as follows. Section 2 presents a general problem statement and the corresponding machine learning model. Section 3 contains description of the proposed algorithms and neural network training details. Section 4 provides simulation details, results, and analysis. Finally, section 5 summarizes the paper results.

2 Problem Statement

2.1 Online Resources Selection and Knapsack Problem

The 0-1 knapsack problem is fundamental for optimization of resources selection and allocation. The classic 0-1 knapsack problem operates with a set of N items having two properties: weight w_i and utility u_i. The general problem is to select a subset of items which maximizes total utility with a restriction C on the total weight:

$$\sum_{i=1}^{N} x_i u_i \rightarrow \max, \tag{1.1}$$

$$\sum_{i=1}^{N} x_i w_i \leq C, \tag{1.2}$$

where x_i - is a decision variable determining whether to select item i (x_i = 1) or not (x_i = 0) for the knapsack.

This problem definition (1.1), (1.2) fits the economic scheduling model with available computing resources having cost c_i (weight) and performance p_i (utility) properties. Many scheduling algorithms and approaches implement exact or approximate *knapsack* solutions for the resources' selection step [4, 11-14]. Sometimes the job scheduling problem may require additional constraints, for example, to limit the number n of items in the knapsack [12, 13] or to select items of different subtypes [14].

The most straightforward exact solution for the knapsack problem can be achieved with a brute force algorithm. However, with increasing N and C in (1.1), (1.2) its application eventually requires inadequately large computational costs. *Dynamic programming (DP)* algorithms can provide exact integer solution with a pseudo-polynomial computational complexity of $O(N * C)$ or $O(n * N * C)$ depending on the problem constraints. Dynamic programming algorithms usually rely on recurrent calculation schemes optimizing additive criteria (1.1) when iterating through the available items. For example, the following recurrent scheme can be used to solve the problem (1.1), (1.2):

$$f_i(c) = \max\{f_{i-1}(c), f_{i-1}(c - w_i) + u_i\}, \tag{1.3}$$

$$i = 1,..,N, c = 1,..,C,$$

where $f_i(c)$ defines the maximum criterion (1.1) value allocated out of first i items with a total weight limit c.

When recurrent calculation (1.3) is finished, $f_N(C)$ will contain the problem solution.

Approximate solution can be obtained with more computationally efficient *greedy algorithms*. Greedy algorithms for the knapsack problem usually use a single heuristic function to estimate the items' importance for the knapsack in terms of their weight w_i and utility u_i ratio. Thus, the most common greedy solution for problem (1.1), (1.2) decreasingly arranges items by their u_i/w_i ratio and successively selects them into the knapsack up to the weight limit.

This greedy solution usually provides a satisfactory (1.1) optimization for an adequate computational complexity estimated as $O(N * Log\ N)$.

Most modern scheduling solutions in one way or another implement these algorithms or their modifications. For example, backfilling scheduling procedure defines additional rules for the *job queue* execution order and is able to minimize the overall queue completion time (a *makespan*). Once the execution priority is defined, each parallel job is scheduled independently based on the problem similar to (1.1), (1.2). One important requirement for the backfilling makespan optimization efficiency is that the job queue composition must be known in advance. The *backfilling* core idea implies execution of relatively small jobs from the back of the queue on the currently idle and waiting resources.

However, in a more general scenario the user jobs are submitted individually, and the resources selection and allocation should be performed immediately in the *online* mode. Thus, our main goal is to schedule user jobs independently in a way to optimize global scheduling criteria, for example average jobs' finish time or a makespan.

Similar ideas underlie the so-called microsheduling approaches, including CoP and PeST [4, 10]. They implement *heuristic* rules of how the resources should be selected for a job based on their meta-parameters and properties: utilization level, performance, local schedules, etc.

2.2 Machine Learning Model

Currently relevant is the topic of using machine learning methods to perform combinatorial optimization tasks, including the knapsack problem (1.1), (1.2) [15-17]. For example, [16] introduces a detailed research of a heuristic knapsack solver based on neural networks and deep learning. The neural solver was successfully tested on instances with up to 200 items and provided near optimal solutions (generally better compared to the greedy algorithm) in scenarios with a correlation between the items' utilities and weights.

In [17], a new class of recurrent neural networks is proposed to compute an optimal or provably good solutions for the knapsack problem. The paper considers a question of a network size theoretically sufficient to find solutions of provable quality for the Knapsack Problem. Additionally, the proposed approach can be generalized to other combinatorial optimization problems, including various Shortest Path problems, the Longest Common Subsequence problem, and the Traveling Salesperson problem.

In the current work we consider a more specific job *scheduling* problem based on a machine learning model. An efficient scheduling plan which minimizes makespan of a whole job-flow can be used to train an artificial neural network (ANN) to schedule each job individually (online) with a similar result. However, the job-flow scheduling plan provides only the efficient resources selections for each job (knapsack result), but not the corresponding utility values of the selected resources. Thus, for the training procedure we can use only secondary meta-parameters and properties of the resources. These typically include resources' cost, utilization level, performance attributes, average downtime, time distance to the neighbor reservations, etc. [4]

The more factors and properties of the efficient *reference* solution are considered the more accurate solution could be achieved *online*. Besides, online scheduling imposes additional restrictions on a priori knowledge of the computing environment composition and condition. The exact values of the resources' properties and utility function may be inaccurate or unknown.

In a more general and formal way, *the main task is to design a model, which will solve (predict solution of) 0-1 knapsack problem with a priori unknown utility u_i values based only on a set of secondary resource's properties*. Thus, to generalize this task we will use more complex knapsack model interpretation with items having four numeric properties a_i, b_i, d_i, g_i in an addition to the weight w_i. Utility values u_i will be calculated for each resource as a function F_{val} of properties a_i, b_i, d_i, g_i. This function will be used to calculate the optimal knapsack solution (by using a dynamic programming algorithm). Based on this solution the machine learning model will be trained to select resources based only on the input properties a_i, b_i, d_i, g_i, thus, simulating the online scheduling procedure.

In this paper, the following utility functions F_{val} will serve as examples of hidden conditions for selecting items in a knapsack:

$$F_{val} = a + b + d - g, \tag{2.1}$$

$$F_{val} = a * b + d * g^2, \tag{2.2}$$

$$F_{val} = \sin(a + b) + \cos d + g^2, \tag{2.3}$$

$$F_{val} = a + \lg(b + d) * g, \tag{2.4}$$

$$F_{val} = a * \lg b + d * e^{\frac{g}{10}}, \tag{2.5}$$

where a, b, d, g are knapsack item's properties in addition to the weight. The given functions contain almost the entire mathematical complexity spectrum in order to investigate at the testing stage how the function complexity affects the algorithm's accuracy and efficiency.

3 Algorithms Implementation

3.1 Artificial Neural Network Design and Training

An artificial neural network (ANN) can be represented as a sequence of layers that can compute multiple transformations to return a result. As the design of the network structure is mostly based on an empirical approach, we performed a consistent design and research of neural network architectures for the knapsack problem.

Firstly, we are faced with the task of classifying an action x_i with a certain item: whether to put it in a knapsack or not. Generally, classification tasks are solved with the decision tree models. However, unlike in a classic problem of individual elements classification, the items in a knapsack invest into a common property: their total weight should not exceed the constraint (1.2). Thus, it is infeasible to classify the elements separately, the model should accept and process everything at once. So, most suitable topology for such a classification problem is a fully connected multilayer neural network (multilayer perceptron).

To implement this model, the Python programming language was used with the Tensorflow framework and the Keras library [18]. Keras has a wide functionality for design artificial neural networks of diverse types.

After selecting the general structure, it is necessary to experimentally select the network parameters. These include: the number of layers, the number of neurons in layers, the neurons activation function, the quality criterion, the optimization algorithm.

First, we used binary cross-entropy as the most suitable loss function for predicting a set of dependent output values.

Next, we designed and tested a set of small candidate models to decide on other meta-parameters (see Table 1).

Table 1. ANN Training Results for 5-elements Knapsack

Configuration Number	**Activation Function**	**Optimizer**	**Number of Layers**	**Neurons in Hidden Layers**	**Training Set**	**Train/Test Accuracy**
1	sigmoid	SGD	1	35	10000	0.79/0.80
2	**sigmoid**	**Adam**	1	35	10000	0.87/0.86
3	relu	SGD	1	35	10000	0.51/0.49
4	relu	Adam	1	35	10000	0.77/0.78
5	sigmoid	Adam	5	35	10000	0.89/0.89
6	sigmoid	Adam	5	35	100000	0.93/0.94
7	**sigmoid**	**Adam**	**5**	**70**	**500000**	**0.96/0.96**
8	sigmoid	Adam	5	200	100000	0.97/0.94
9	sigmoid	Adam	9	35	100000	0.89/0.89
10	sigmoid	Adam	9	90	100000	0.94/0.93
11	relu	Adam	9	90	100000	0.69/0.69

From the initial training results (see Table 1), we can make the following conclusions:

1) a pair of Sigmoid activation function and Adam optimizer showed the best result in terms of the accuracy criteria;

2) increase in a number of ANN layers requires a larger size of the training set to achieve a higher accuracy;

3) the achieved 0.96 accuracy shows that ANN is able to solve knapsack problem fairly well given the right number of layers and the size of the training set.

Fig. 1 shows how accuracy and loss values were improved on the validation set during the training of the best ANN configuration (number 7) from Table 1. The smoothness and linearity of the graphs indicates the adequacy of the selected parameters and the possibility of stopping at using 150-200 training epochs.

Next, we consistently increased the dimension of the knapsack problem and estimated how different hidden utility functions affect the ANN accuracy.

The training set was obtained as a dynamic programming-based exact solution for a randomly generated knapsack problem. The items' properties and the weight constraint were generated randomly to achieve the required features: 1) representativeness – a data set selected from a larger statistical population should adequately reproduce a large group according to any studied characteristic or property; 2) consistency – contradictory data in the training sample will lead to a low quality of network training.

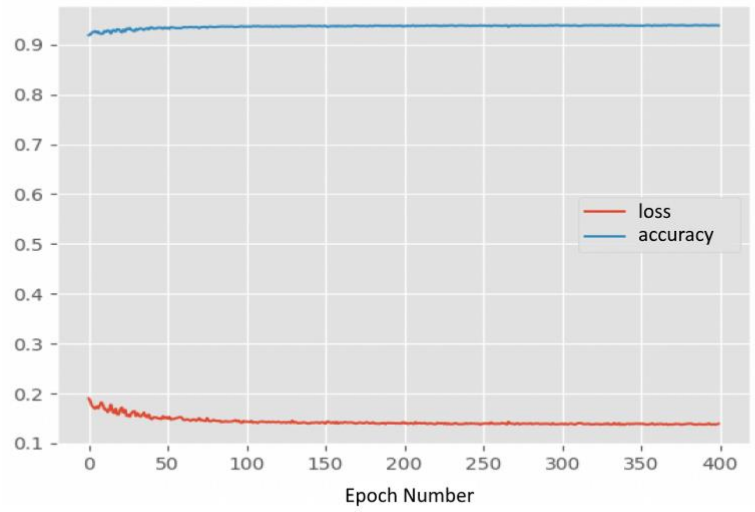


Fig. 1. ANN validation loss and accuracy for 5-element knapsack problem (configuration 7 from Table 1)

The ANN input data is a training sample consisting of the knapsack element properties vectors a_i, b_i, d_i, g_i and the normalized weights vector $w_i^{'} = w_i/C$. Weights normalization allows us to generalize the weight constraint in (1.2) to C=*1* for any input sample. Vector y_answer_i of the correct selection is used for the loss function calculation and backpropagation step. The correct solution was obtained using a dynamic programming algorithm with explicit use of the hidden utility function.

The training and testing results for a 20-elements knapsack are presented in Table 2. As a main result, ANN was able to solve knapsack problem equally successfully for all the considered hidden functions (2.1) - (2.5) by using only the properties a_i, b_i, d_i, g_i of the knapsack items.

Table 2. Training Results for 20-elements Knapsack

Hidden Utility Function	**Number of Layers**	**Train/Test Accuracy**
$a + b + d - g$	14	0.94/0.93
$a * b + d * g^2$	17	0.92/0.91
$\sin(a + b) + \cos d + g^2$	14	0.93/0.92
$a + \lg(b + d) * g$	14	0.93/0.92
$a * \lg b + d * e^{\frac{g}{10}}$	14	0.92/0.91

3.2 MLAK Algorithm

While training a neural network, it is impossible to operate with formal mathematical concepts, in particular those defined for the knapsack problem (1.1), (1.2). The training relies on a set of pre-prepared examples of an optimal selection of the knapsack items.

The main problem with the pure ANN knapsack prediction is that even with a high accuracy we cannot be sure that the condition for the knapsack total weight is fulfilled.

To consider the restriction on the total knapsack weight, we propose to use the ANN classification result as a predicted utility vector h_i which can be used in a separate algorithmic knapsack solution. That is, the input data for the problem (1.1), (1.2) will contain weight w_i and utility $u_i = u_i'$ vectors, where u_i' values are predicted for each element based on the item's properties a_i, b_i, d_i, g_i. In this way, the ANN will operate as a conversion module to identify mutual relationships between the knapsack items' properties and map them to the predicted utility values u_i'.

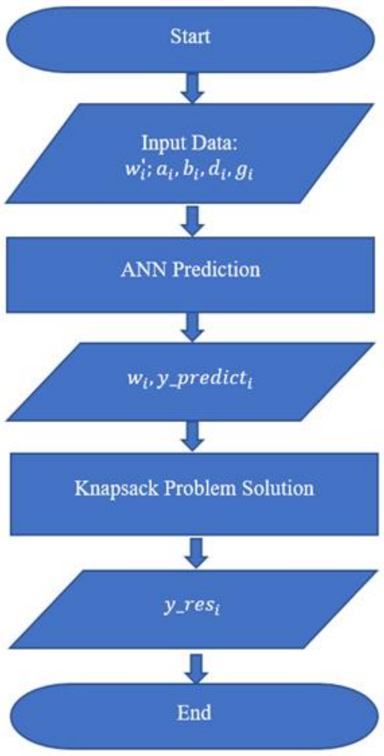
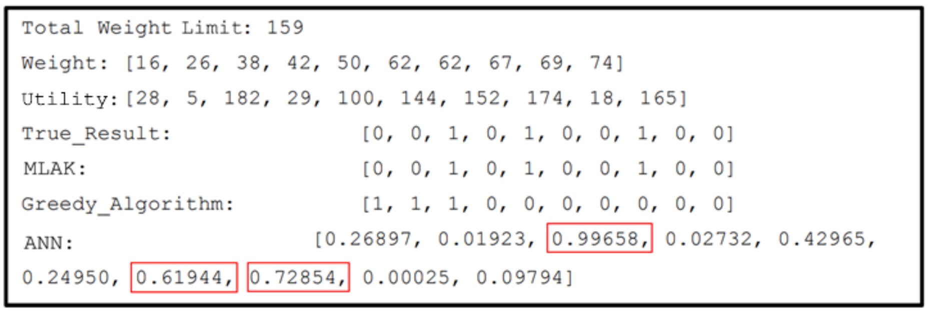

Start
Input Data: $w_i'; a_i, b_i, d_i, g_i$
ANN Prediction
$w_i, y\_predict_i$
Knapsack Problem Solution
y_res_i
End

Fig. 2. MLAK algorithm flowchart

Fig. 2 shows the flowchart of the proposed composite Machine Learning-based Algorithm for the Knapsack problem (hereinafter MLAK). It consists of ANN conversion module and a dynamic programming-based algorithm to provide the final solution for problem (1.1), (1.2) with an unknown, but predicted utility values and a constraint on the total weight.

The artificial neural network input for items $i = 1 \ldots n$:

- a_i, b_i, d_i, g_i – vectors of the properties;
- w_i – vector of the items' weights;
- $w_i' = w_i / C$ – normalized vector of the weights;
- y_answer_i – an optimal selection result calculated by the dynamic programming method with use of a hidden utility function;
- $y\_predict_i$ – items selection predicted by the neural network;
- y_res_i – the final MLAK items selection.



```
Total Weight Limit: 159
Weight: [16, 26, 38, 42, 50, 62, 62, 67, 69, 74]
Utility:[28, 5, 182, 29, 100, 144, 152, 174, 18, 165]
True_Result:                [0, 0, 1, 0, 1, 0, 0, 1, 0, 0]
MLAK:                       [0, 0, 1, 0, 1, 0, 0, 1, 0, 0]
Greedy_Algorithm:           [1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
ANN:                    [0.26897, 0.01923, 0.99658, 0.02732, 0.42965,
0.24950, 0.61944, 0.72854, 0.00025, 0.09794]
```

Fig. 3. ANN and MLAK prediction comparison

Fig. 3 presents sample data to demonstrate ANN and MLAK prediction differences with only a single utility property. In this example of a problem (1.1), (1.2) weight limit $C = 159$ is stated in the first line, while weight and utility vectors are presented on the following lines of Fig. 3. No hidden function is used, utility values are directly used as input data for ANN.

The exact solution (*True_result)* of x_i values was obtained with a dynamic programming procedure (1.3); approximate solution was provided by a greedy algorithm.

ANN result is a prediction vector of the x_i values. Based on this prediction, items (3, 7, 8) should be selected for the knapsack (indicated in red squares in Fig. 3), which is not the exact solution.

MLAK algorithm used ANN prediction vector as new utility values u_i' for the knapsack items. The dynamic programming procedure (1.3) was performed over u_i' and w_i values in (1.1), (1.2) with the result equal to the exact solution.

4 Simulation Experiment

4.1 Simulation Environment

We evaluate MLAK efficiency based on a comparison with classical knapsack algorithms (including dynamic programming and greedy implementation), as well as with a pure ANN knapsack implementation. ANN results were additionally modified to comply with the weight restriction: selected items with the smallest prediction confidence were removed one by one until the restriction is satisfied.

For this comparison we used the same (2.1) - (2.5) hidden utility functions F_{val}. Thus, MLAK and ANN received a_i, b_i, d_i, g_i properties as an input, while Dynamic programming (DP) and Greedy implementations used $u_i = F_{val}$ calculated utility functions to solve the knapsack problem.

Additionally, we implemented a Random selection algorithm to evaluate MLAK and ANN efficiency in the interval between the optimal solution provided by DP and a completely random result.

All the considered algorithms were given a set of the same 1000 knapsack problems as input. We consider two main efficiency indicators for each algorithm's solution:

- the resuting knapsack total utility and its relation to the DP result (average utility);
- the resulting accuracy as element-wise comparison with the DP result for all the experiments.

Both efficiency criteria are based on DP algorithm as it provides an optimal integer solution based on the known utility function.

Accuracy criterion will have 100% value when knapsack solution is identical to the DP in all 1000 experiments. Thus, low accuracy value does not necessarily mean low algorithm efficiency, as the same or comparable knapsack utilities sometimes may be achieved by selecting different combinations of items. Accuracy parameter shows how often the resulting solution matches the optimal one by performing similar optimization operations.

Therefore, an average total utility should serve as a main comparison criterion for problem (1.1), (1.2).

Besides, we measure and compare average working times. All the considered algorithms were implemented using Python language. Working time was observed on desktop PC with Core i5 and 8Gb RAM. MLAK working time includes both internal ANN and DP algorithms execution (see Fig. 2).

4.2 Simulation Results and Analysis

Simulation results collected over 1000 independent knapsack problems with 20 items are presented in Tables 3-7. Each table corresponds to a single hidden function (2.1) - (2.5).

Firstly, the results show that Greedy algorithm provided almost optimal average utility: nearly 99% compared to DP. This result is expected for 20 elements with randomly and uniformly generated properties and utility values. 50-85% accuracy shows that it is usually possible to achieve comparable optimization results with different items selected. Different utility functions are mostly affecting the accuracy difference (50-85% interval) as they provide varying diversity in the resulting utility values of the knapsack items. Low diversity usually leads to a higher accuracy values, as there are less options to achieve an efficient solution.

MLAK and ANN optimization efficiency is generally comparable to the Greedy implementation. Relative difference by the average utility between Greedy and MLAK is less than 1% for functions (2.1), (2.3), (2.4), and (2.6). ANN provides similar results with less than 1% lower utility compared to MLAK.

For functions (2.2) and (2.5) the relative difference with DP reaches 3%, which may be explained by much larger absolute values of the utility functions obtained from the same set of the randomly generated input properties (see column Average Utility in Tables 3-7). ANN prediction works less efficient when relations between the properties include multiplication and exponentiation operations.

However, even this less than 3% optimization loss (in the worst case observed scenarios) is rather small and reasonable when compared to the random selection result with more than 40% difference from DP solution. Besides, in this comparison DP and Greedy performed knapsack optimization using the actual utilities calculated from the hidden functions, while Random shows average results with no optimization.

When compared to each other, MLAK provides a slightly better average resulting utility and noticeably higher accuracy than ANN. Pure ANN usually generates quite efficient solutions which degrade when the weight constraint is applied. So, MLAK is one of the efficient ways to apply weight constraint over the pure ANN knapsack prediction.

In terms of the actual working time MLAK is inferior to all the other considered algorithms. Obviously, the strong difference in execution time between ANN-based and traditional algorithms is due to the nature and complexity of artificial neural networks, which are required to replicate hidden utility functions. For the considered 20-items knapsack problem MLAK prediction time of 0.05 seconds may seem quite insignificant, but larger problems will require increase in the ANN structure, training sample size, time and calculation efforts for the training.

Table 3. Function 2.1 Optimization Results

Algorithm	Average Utility	Average Utility, %	Accuracy, %	Average working time, s
Greedy	2175	98,7	48,0	0.00004
MLAK	2163	98,2	39,9	0.04206
ANN	2117	96,1	25,9	0.02363
Random	1270	57,6	0,6	0.00005
Greedy	2175	98,7	48,0	0.00004

Table 4. Function 2.2 Optimization Results

Algorithm	Average Utility	Average Utility, %	Accuracy, %	Average working time, s
DP	25.2*10^5	100,0	100	0.00609
Greedy	25*10^5	99,2	59,2	0.00004
MLAK	24.6*10^5	97,7	36,6	0.04145
ANN	23.9*10^5	94,9	21,6	0.02385
Random	13.5*10^3	53,5	1,4	0.00005

Table 5. Function 2.3 Optimization Results

Algorithm	Average Utility	Average Utility, %	Accuracy, %	Average working time, s
DP	154057	100,0 %	100 %	0.00743
Greedy	153127	99,4 %	64,1 %	0.00004
MLAK	151601	98,4 %	42,5 %	0.04531
ANN	148806	96,6 %	22,6 %	0.02380
Random	79732	51,7%	1,2 %	0.00005

Table 6. Function 2.4 Optimization Results

Algorithm	Average Utility	Average Utility, %	Accuracy, %	Average working time, s
DP	3371	100	100	0.00734
Greedy	3339	99,0	49,6	0.00004
MLAK	3326	98,7	40,8	0.04556
ANN	3245	96,3	22,6	0.02400
Random	2024	60,0	1,4	0.00005

Table 7. Function 2.5 Optimization Results

Algorithm	Average Utility	Average Utility, %	Accuracy, %	Average working time, s
DP	6049*10^6	100,0	100	0.00823
Greedy	6036*10^6	99,8	85,9	0.00004
MLAK	5849*10^6	96,7	41,2	0.04784
ANN	5864*10^6	96,9	24,1	0.02391
Random	2407*10^6	39,8	1,0	0.00005

5 Conclusion

The paper introduced a promising machine learning-based approach for online scheduling and resources allocation. A generalized model for knapsack problem solution based on hidden (unknown) utility functions was proposed and simulated. The main design and practical development stages of the artificial neural network were presented and considered. Additional optimization step was proposed to apply weight constraint over the neural network prediction.

As a main result, the proposed algorithm MLAK showed the knapsack optimization efficiency comparable to a classical greedy implementation for five different hidden utility functions covering a wide spectrum of mathematical complexity. The importance of this result is that MLAK, unlike greedy algorithm, did not directly used the hidden utility functions of the elements. Instead, it was pre-trained on a set of optimal solutions for randomly generated knapsack problems.

Future work will concern problems of the algorithm scalability and more practical online job-flow scheduling implementations.

Acknowledgments. This work was supported by the Russian Science Foundation (project no. 22-21-00372).

References

1. Bharathi, S., Chervenak, A.L., Deelman, E., Mehta, G., Su, M., Vahi, K.: Characterization of Scientific Workflows. In: 2008 Third Workshop on Workflows in Support of Large-Scale Science, pp. 1–10 (2008)
2. Rodriguez, M.A., Buyya, R.: Scheduling Dynamic Workloads in Multi-tenant Scientific Workflow as a Service Platforms. Future Generation Computer Systems, 79 (P2), 739–750 (2018)
3. Kurowski, K., Nabrzyski, J., Oleksiak, A., Weglarz, J.: Multicriteria Aspects of Grid Resource Management. In: Nabrzyski, J., Schopf, J.M., Weglarz J. (eds.) Grid resource management. State of the art and future trends, pp. 271-293. Kluwer Academic Publishers. (2003)
4. Toporkov, V. and Yemelyanov, D.: Heuristic Rules for Coordinated Resources Allocation and Optimization in Distributed Computing. In: J. M. F. Rodrigues et al. (Eds.): ICCS 2019, LNCS 11538, Springer Nature Switzerland AG, pp. 395–408 (2019)

5. Toporkov, V., Yemelyanov D. and Toporkova, A.: Coordinated Global and Private Job-Flow Scheduling in Grid Virtual Organizations. J. Simulation Modelling Practice and Theory, Vol. 107, Elsevier. (2021)
6. Sukhoroslov, O., Nazarenko, A. and Aleksandrov, R.: An Experimental Study of Scheduling Algorithms for Many-Task Applications. Journal of Supercomputing, 75, 7857–7871 (2019)
7. Samimi, P., Teimouri, Y., Mukhtar M.: A Combinatorial Double Auction Resource Allocation Model in Cloud Computing. J. Information Sciences, 357(C), 201-216 (2016)
8. Rodero, I., Villegas, D., Bobroff, N., Liu, Y., Fong, L., Sadjadi, S.: Enabling Interoperability Among Grid Meta-schedulers. Journal of Grid Computing, 11(2), 311–336 (2013)
9. Shmueli, E., Feitelson, D.G.: Backfilling with Lookahead to Optimize the Packing of Parallel Jobs. Journal of Parallel and Distributed Computing, 65(9), 1090–1107 (2005)
10. Khemka, B., Machovec, D., Blandin, C., Siegel, H.J., Hariri, S., Louri, A., Tunc, C., Fargo, F., Maciejewski, A.A.: Resource Management in Heterogeneous Parallel Computing Environments with Soft and Hard Deadlines. In: Proceedings of 11th Metaheuristics International Conference (MIC'15) (2015)
11. Netto, M. A. S., Buyya, R.: A Flexible Resource Co-Allocation Model based on Advance Reservations with Rescheduling Support. In: Technical Report, GRIDSTR-2007-17, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, (2007)
12. Toporkov, V., Toporkova, A., Yemelyanov, D. Slot Co-Allocation Optimization in Distributed Computing with Heterogeneous Resources. Studies in Computational Intelligence, Volume 798, Pages 40-49, Springer Nature Switzerland AG (2018)
13. Toporkov, V., Yemelyanov, D. Optimization of Resources Selection for Jobs Scheduling in Heterogeneous Distributed Computing Environments // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 10861 LNCS, 2018, Springer Verlag, pp. 574-583 (2018)
14. Toporkov, V., Yemelyanov, D. (2021). Scheduling Optimization in Heterogeneous Computing Environments with Resources of Different Types. In: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (eds) Theory and Engineering of Dependable Computer Systems and Networks. DepCoS-RELCOMEX 2021. Advances in Intelligent Systems and Computing, vol 1389. Springer, Cham. (2021)
15. Xu, S.; Panwar, S. S.; Kodialam, M. S.; and Lakshman, T. V.: Deep Neural Network Approximated Dynamic Programming for Combinatorial Optimization. In: AAAI Conference on Artificial Intelligence, 1684–1691 (2020)
16. Nomer, H. A. A., Alnowibet, K. A., Elsayed, A. and Mohamed, A. W.: Neural Knapsack: A Neural Network Based Solver for the Knapsack Problem. In: IEEE Access, vol. 8, pp. 224200-224210 (2020)
17. Hertrich, C. and Skutella, M.: Provably Good Solutions to the Knapsack Problem via Neural Networks of Bounded Size. In: Proceedings of the AAAI Conference on Artificial Intelligence, 35(9), 7685-7693 (2021)
18. Chollet, F.: Xception: Deep Learning with Depthwise Separable Convolutions, 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017, pp. 1800-1807 (2017)