Scheduling conditional task graphs with deep reinforcement learning

Anton Debner*[1], Maximilian Krahn[1], and Vesa Hirvisalo[1]

[1]Aalto University
{anton.debner, maximilian.krahn, vesa.hirvisalo}@aalto.fi

Abstract

Industrial applications often depend on costly computation infrastructures. Well optimised schedulers provide cost efficient utilization of these computational resources, but they can take significant effort to implement. It can also be beneficial to split the application into a hierarchy of tasks represented as a conditional task graph. In such case, the tasks in the hierarchy are conditionally executed, depending on the output of the earlier tasks. While such conditional task graphs can save computational resources, they also add complexity to scheduling.

Recently, there has been research on Deep Reinforcement Learning (DRL) based schedulers, but they mostly do not address conditional task graphs. We design a DRL based scheduler for conditional task graphs in a heterogeneous execution environment. We measure how the probabilities of a conditional task graph affects the scheduler and how these adverse effects can be mitigated. We show that our solution learns to beat traditional baseline schedulers in a fraction of an hour.

1 Introduction

Scheduling of complex computing actions is needed in many modern industrial systems. Industrial systems often have a rich set of sensors and actuators interacting together in a time-sensitive manner. These interactions are often accompanied with computationally heavy tasks, that are mapped to available computational resources. Fine-tuned schedulers can increase the efficient utilization of these computational resources, which in turn can lead to significant cost savings at scale.

While traditional schedulers are well researched and understood [1], efficient scheduling often requires hand-tuned solutions on case-by-case basis. Along with the growing interest towards using machine learning in industrial applications [2, 3], there has been research for automating the creation of schedulers for complex scenarios [4–7].

Due to the introduction of ever larger machine learning models, it can be beneficial to split an application into a hierarchy of models represented as a conditional task graph. In such case, the models in the hierarchy are conditionally executed, depending on the output of the earlier models. Ideally, most of the data runs through the smaller models while the larger models later in the hierarchy are used more sparingly.

We focus on how conditional task graphs affect scheduling. The presence of conditional task graphs is problematic for schedulers, as the scheduler can only make limited long-term planning due to not knowing the exact stages of a job in advance. As a further complication, the transition probabilities between each task may vary depending on external factors, such as changes in the physical condition of sensors, actuators and quality of materials over time. In other words, these external factors may change the probability of, e.g., needing to use computationally heavier models to maintain good accuracy.

As our contribution, we present our deep reinforcement learning (DRL) based scheduler for conditional task graphs and test it in a simulated heterogeneous computation environment[1]. We base our DRL-scheduler in common design decisions used in related work. Our results show that our scheduler improves up to 30 % of mean job completion time (JCT) compared to our baselines with less than 15 minutes of training.

We start the paper by describing conditional task graphs and reinforcement learning in Section 2. Then we describe our problem in Section 3. In Section 4 we describe common design choices for DRL-based schedulers, followed by our experiment (Section 5) and results (Section 6). We end our paper in conclusions (Section 7).

2 Using machine learning in cluster scheduling

In the context of parallel computing in clusters, the scheduling of the *tasks* of an application has a crucial impact on the system performance. A task schedule determines both the mapping of tasks to the processors, and the order in which they are executed. Typical schedule optimization problems, such as minimising the total execution time, are

*Corresponding Author.

[1]https://github.com/Aalto-ESG/drl-scheduler-2024

Proceedings of the 5th Northern Lights Deep Learning Conference (NLDL), PMLR 233, 2024.

challenging optimisation problems (formally strongly NP-hard) and usually heuristics are used instead of optimal solutions [1].

Tasks are typically selected so that they fit the overall execution arrangement including its scheduling aspects. As tasks are usually too small for representing complete algorithms, groups of interdependent tasks are often used to form *jobs*. The typical optimization target is the the Job Completion Time (JCT), i.e., the total execution time of the related tasks (also known as the makespan).

Task interdependencies imply constraints on tasks execution order, and thus, limit the choices that a scheduler has. They are usually expressed by using various forms of Directed Acyclic Graphs (DAG), which include *conditional task graphs* [8]. There exists several variations of conditional task graphs. We use probabilistic conditions in our Task Graph (Fig. 2, denoted as TG in this paper).

Efficient scheduling often requires hand-tuned solutions on case-by-case basis. As machine learning has shown potential for solving complex problems, it has emerged as an way of automating the creation of schedulers for complex scenarios.

Our research concentrates on applying Deep Reinforcement Learning (DRL) [9]. It is a branch of machine learning that aims to learn optimal behavior by interacting with an environment by trial-and-error. Often, the environment is a simulator that implements a gym-interface [10], enabling the use of variety of DRL training frameworks, such as RLlib [11]. A DRL agent interacts with the environment by sending actions and receiving observations and rewards. DRL uses Neural Networks (NN) as part of the agent.

As related work, Chen et al. [2] and Khalil et al. [3] give overview of applying DRL for Internet of Things. As a more concrete example, IRATS [7] is a DRL-based intelligent priority and deadline-aware resource allocation algorithm trained with Proximal Policy Optimization (PPO [12]). They review using Graph Neural Networks (GNNs [13]) for processing graph-like input data. Furthermore, Lin et al. [6] apply reinforcement learning together with heuristics for the scheduling of DAGs. Their approach is based on combining a heuristic-based processor allocation (e.g., HEFT [14]) with DRL-based node selection. They use a Graph Convolutional Network (GCN) as an input layer to process graph-based information, and PPO to train the neural network. Similarly, Duan and Wu [5] consider reducing JCT for DAG-style jobs by adding idle slots. They also apply DRL with GNN to capture the DAG structures.

However, despite the wide research on the scheduling problems, the scheduling of conditional task graphs by means of modern machine learning methods has largely remained not addressed.

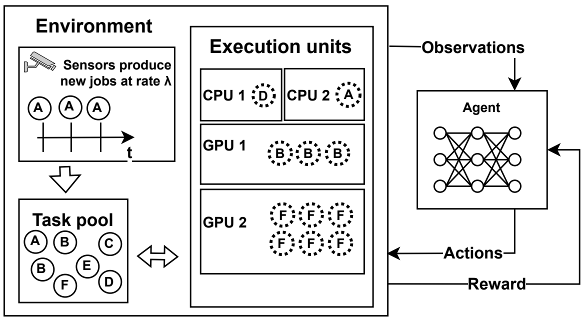

Figure 1. Overview of the system.

3 Problem Description

Our motivation comes from a production line with sensors, where the set of sensors produce a new job at regular intervals. Each job includes a set of classification tasks, where the result of the classification determines the next task. In essence, less interesting data is discarded sooner, while potentially more interesting data is processed through a more computationally intensive set of classification tasks. In other words, each job follows a conditional TG, similar to the TG in Fig. 2.

Our goal is to minimize the average JCT, which for each job J_i is defined as

$$JCT(J_i) = CT(J_i) - AT(J_i) \qquad (1)$$

where $AT(J_i)$ is the time the first task of the i^{th} job arrived and $CT(J_i)$ is the time the last task of the i^{th} job was completed.

3.1 Dynamic DAG properties

We consider two scenarios: In the first scenario we have fixed transition probabilities in the DAG. However, the probabilities are different for each installation of the system. The scheduler should either generalize well to different probabilities or be quick to adapt for a new installation.

In the second scenario, the transition probabilities are changing at runtime. This could happen, if the probabilities are dependent on the physical condition of the equipment and materials. In this case, the scheduler should adapt to different transition probabilities at runtime.

3.2 Heterogeneous environment

We consider a heterogeneous execution environment with multiple accelerators with different properties. The properties are execution time coefficient and maximum batch size. It is natural to assume that different accelerators can run tasks at different speeds. In addition, batched inference is a typical way of increasing accelerator utilization through parallel execution [15].

In our problem, we define task execution time as

$$execution\_time(j, k) = T_j \cdot c_k \tag{2}$$

where c_k is the speed coefficient of the k^{th} accelerator and T_j is the normalized execution time of the j^{th} task type.

The number of tasks executed is defined as

$$batch\_size(j, k) = min(q_j, B_k) \tag{3}$$

where q_j is the number of tasks of j^{th} task type in queue and B_k is the maximum batch size for the k^{th} accelerator.

3.3 Markov Decision Process

DRL can be used to solve problems that are modeled as a Markov Decision Process (MDP). Therefore, we formalize this setup as an MDP. The MDP is a tuple (S, A, P_a, R_a). Here S represents the full state space, including the tasks waiting to be processed along with the state of the cluster. The size of the state space depends on n and m, where n is the number of nodes in the TG and m is the amount of execution units in the cluster.

Since we have multiple execution units that each can process one task type at a time, our action space is vector $A = [E_0, ..., E_m]$, where E_k is the action for the k^{th} execution unit. Each element E_k has $n + 1$ possible values, and the value represents the task type that should be retrieved from a FIFO queue for execution. The last value is an idle action.

$P_a(s, s')$ is the probability of transitioning between two states when choosing action $a \ \epsilon \ A$. $R_a(s, s')$ is the immediate reward received for executing action a in state s. The reward should reflect our goal of minimizing the average JCT. As $P_a(s, s')$ may change over time, the setup can also be seen as a Partially Observable MDP (PO-MDP).

4 Scheduler design

To design a DRL-agent, we need to consider the action space, observation space and the reward signal. Additionally, we need to choose the structure of the NN and the training algorithm. [9]

Action space depends directly on how the problem is modeled as an MDP. The size of the observation space has a direct effect on learning efficiency. Using too large observation vectors can slow down learning as there are more parameters to learn from. Therefore, instead of giving the whole state $s \ \epsilon \ S$ of the MDP as an input to the DRL agent, it is common to use a state abstraction that maps s to s_ϕ, where $|s_\phi| << |s|$ and s_ϕ is the observation vector given to the agent. In practice this can be done, for example, by leaving out potentially less useful information or by computing an arithmetic mean to aggregate multiple values together. However, reducing the observation space too much can also cause adverse effects, as there might not be enough information to select an optimal action.

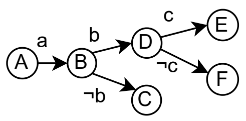

Figure 2. Conditional task graph.

The reward signal should be chosen depending on the desired goal of the system. The quality of the reward signal has direct effect on learning behavior. A poorly designed reward signal can lead the agent to optimize towards a sub-optimal solution.

4.1 Action space

The action space of the agent is the same A that was defined in Section 3.3. To make learning easier, we use action masking [16] to mask out task types that are currently not queued in the task pool. We also mask out execution units that are already occupied. The action mask is a Boolean vector with $m(n + 1)$ elements. The last element $(n+1)$ for each execution unit is always true, as it corresponds to the idle-action. The idle-action is considered as no-op, if the execution unit is already occupied.

4.2 State space

We consider two different observation vectors $s_{\phi 0}$ and $s_{\phi 1}$. The first vector is $s_{\phi 0} = [t_f, t_a, q]$, where t_f is the waiting time for the first num_{tf} queued tasks of each task type. Likewise, t_a is the average waiting time for all queued tasks of each task type. Element q is the number of queued tasks for each task type. The idea behind num_{tf} is to balance between having enough information to make optimal decisions, while also having a small enough observation size to make learning faster.

In order to deal with probabilistic transitions in the TG, we add est_p to the second observation vector $s_{\phi 1}$. The est_p is a running average estimation for each transition probability of the TG. Therefore, the second vector becomes $s_{\phi 1} = [t_f, t_a, q, est_p]$.

4.3 Neural network

As our neural network, we are using a small Feedforward Neural Network (FNN) with fully connected layers. This kind of a network is easy to implement, and fast to compute when using small enough layers. This is suitable in our case, as we are scheduling a single relatively small sized DAG. The weights of the network are optimized with PPO.

Table 1. Environment variables.

Property	Value	Property	Value
CPU speed coef	0.8	p(a)	1
GPU speed coef	1	p(b)	0.5
CPU max batch size	1	p(c)	dynamic
GPU-1 max batch size	5	exec time (A, ..., E)	10 ms
GPU-2 max batch size	10	exec time (F)	100 ms

Table 2. Training and evaluation hyperparameters.

num training jobs	5000	learning rate	0.005
num evaluation jobs	10000	batch size	80000
num eval iterations	40	sgd minibatch size	1000
job arrival interval	8 ms	sgd iterations	5
training steps	5000000	hidden layers	[64, 64]
(observation vector) num_{tf}	3		

If there is need for a more scalable policy that can adapt to multiple DAGs with different sizes, a GNN can be added as an additional input layer to process variable amounts of graph-based data before feeding it to the fully connected layers. The use of GNNs for variable sized inputs is shown in many of the earlier mentioned related work.

4.4 Reward signal

Similar to [4], we define the reward signal as

$$r_p = -(t_p - t_{p-1})num_J(p) \tag{4}$$

where $num_J(p)$ is the number of jobs in the system during interval $[t_{p-1}, t_p)$ after the p^{th} action. This reward signal attempts to minimize average JCT by minimizing the average number of jobs in the system. The advantage of this reward signal is that it is very easy to compute and it matches the goal of minimizing average JCT well.

Many of the alternatives that we considered slow down learning by being computationally more complex, being harder to learn from or causes the policy to skew towards a sub-optimal solution. For example, giving a reward based on the JCT of the most recently completed job causes the reward signal to become sparse, as jobs are not completed on every step. Likewise, computing the average waiting time for all currently queued jobs as a negative reward signal adds additional computational complexity.

5 Experiment setup

In this section, we describe our experiment setup. The goal of the experiment is to find out how conditional transitions affect scheduling in a simple scenario and how our DRL agent can learn to minimize the average JCT with conditional task graphs with minimal information about the system.

5.1 Simulator

Fig. 1 shows the overview of the experiment setup. For the experiment, we designed and implemented a discrete-event simulator to model the operation of a system that processes jobs in a computing cluster. The jobs arrive to the system at regular intervals, and their tasks have to be mapped to accelerators.

Our simulated cluster has a simple heterogeneous accelerator setup with four different accelerators, as shown in Fig. 1. We could think of them as having two identical CPU cores and two distinct GPUs available. The CPUs are slightly faster at executing a singular task, but the GPUs can run a batch of the same task in parallel. The exact properties are shown in Table 1.

As a workload, we designed a TG to represent a simple but reasonably realistic scenario, where longer tasks occur rarely. The Fig. 2 shows the shape of the TG. The Table 1 shows the execution time for each task type and the probabilities of taking the transition to the next task.

Since there are three leaf nodes, we can think of the TG as having three distinct jobs with some shared task types. However, due to the conditional nature of the TG, it is not possible to know in advance which of the three sequences a job will take.

5.2 Training and evaluation

We measure the behavior on two degrees of freedom. First, to investigate how training and design details affect the DRL-based schedulers, we train five separate policies π_0, π_1, π_2, π_3 and π_4.

Second, we measure the scheduling performance for each policy with all values of transition probability $p(c)$ between 0 and 1 at small intervals.

Policies π_0, π_1, π_2 are trained with a fixed value for $p(c)$. The values are $p_0(c) = 0.1$, $p_1(c) = 0.5$ and $p_2(c) = 0.9$ respectively. Policy π_3 is trained by randomizing $p(c)$ at start of each training episode to encourage generalisation. We sample the randomized value for $p(c)$ in the range of [0, 1] from a uniform distribution. In addition to this randomization, π_4 includes a running average estimation of the $p(c)$ value in its observation.

Most of the policies use the observation vector $s_{\phi 0}$. The total size of $s_{\phi 0}$ is 30, with $num_{tf} = 3$. Likewise, policy π_4 uses $s_{\phi 1}$, which has 31 elements. The extra element comes from est_p, that is an estimate for the value of $p(c)$. The action vector for each policy has four elements with seven possible values for each element, as we have four execution units, six task types and an idle action.

For training the models we use Ray [17] along with its reinforcement learning library RLlib [11]. We use the default RLlib implementation of PPO [12] with action masking [16]. Software versions and hard-

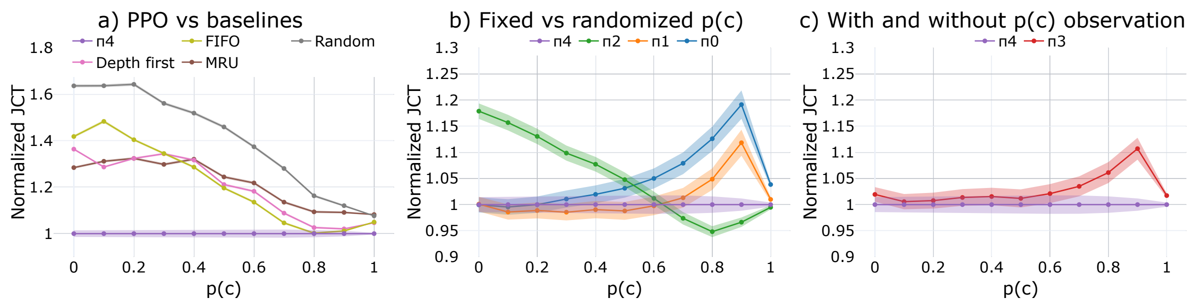

Figure 3. Relative average JCT for different policies at different values for $p(c)$. Values are normalized against π_4. Note the different y-axes.

Table 3. Software and hardware specifications.

CPU	RAM	ML framework	ML backend
Intel Core 13900k	64 GB	Ray 2.5.1	PyTorch 2.0.1

ware specifications are shown in Table 3. Training hyperparameters can be seen in Table 2.

Evaluation is done separately with longer episodes after training is finished. For each policy, we repeat the evaluation with 40 different fixed seeds for each measurement point. The rest of the evaluation parameters can be seen in Table 2.

5.3 Baselines

As baselines, we use multiple traditional algorithms, such as First-In-First-Out (FIFO) and Most-Recently-Used (MRU) and Depth-First, which prioritizes tasks further along the TG. These baseline algorithms allocate tasks to multiple execution units one-by-one following their policy until all execution units have received an action. The baseline algorithms are not aware of the properties of each execution unit.

Heuristic methods such as HEFT [14] could provide a more advanced and resource-aware comparison, but would need to be modified for conditional TGs.

6 Results and discussion

In this section, we present the results of our experiment. We start by describing the overall performance of our policies against each other and the baselines. Then, we analyze the effectiveness of our training setup and discuss further improvements.

6.1 General performance

Fig. 3 shows the average JCT for all policies at different $p(c)$ values. In Fig. 3(a) we can see that our policy π_4 performs better than baselines with all $p(c)$ values, ranging from 0 % to 30 % improvement over the best baseline. The comparison is not entirely fair, as the baselines are not optimized for the heterogeneous execution environment.

Despite the unfairness, we can further analyze the behavior of the DRL-based policies. In Fig. 3(b) we can see that by randomizing the $p(c)$ and having an estimate of $p(c)$ in the observation vector, our policy π_4 is able to generalize well to all values of $p(c)$. We can also see that π_4 has room for improvement, as models trained with fixed values perform better in some cases. Most notably, policy π_2 is 5 % better when $p(c) = 0.8$. As a clarification, with high values of $p(c)$ the computationally heavy task F occurs rarely. However, as one might expect, the policies π_0, π_1, π_2 perform significantly worse when the $p(c)$ gets further from their training target.

Finally, Fig. 3(c) shows the effect of having an estimate of $p(c)$ in the observation vector. As expected, the additional observation helps π_4 to be consistently better than π_3, reaching more than 10 % difference at some $p(c)$ values. On the other hand, even without the $p(c)$ observation, π_3 still beats all baselines by a significant margin in most cases. Additionally, with many values of $p(c)$ (≤ 0.6) the difference between π_3 and π_4 is only roughly 2 %.

The overhead of the schedulers is also rather small, as the inference time for the NN measures at less than 20 microseconds on our test hardware.

6.2 Training behavior

Training five million samples took roughly 15 minutes for each DRL policy. This is rather quick, especially considering that the training hyperparameters are not finely tuned, the hardware is underutilized and our simulator is unoptimized.

Fig. 4 shows the typical behavior for training and test performance for both DRL policies. Both mean episode length and mean episode reward show steady increase, until converging reasonably well at three million samples for all policies.

Fig. 4(c) shows the relative performance compared to the best baseline during training. Values less than 1 means that the policy achieves lower

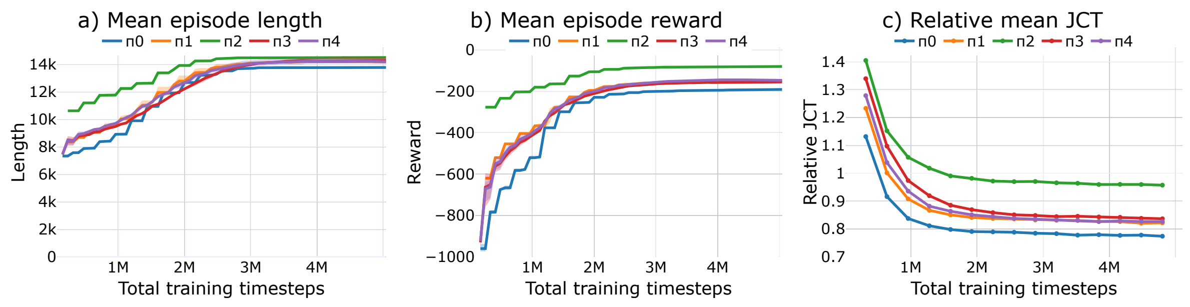

Figure 4. Learning curves for each DRL policy.

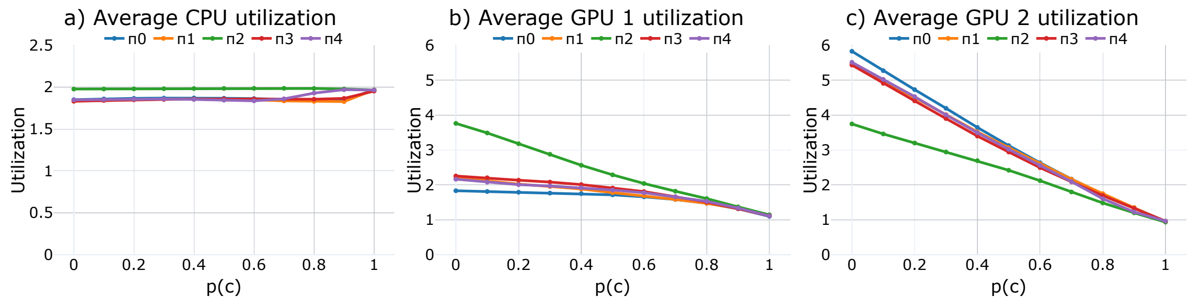

Figure 5. Average task execution unit utilization rates for each DRL policy. Both CPUs are combined in one graph, as they have identical properties and similar behavior.

mean JCT than the best baseline. We can see that better-than-best-baseline performance is achieved after roughly 500 thousand to 2 million samples.

Fig. 5 shows how the DRL-policies learn to utilize different task execution units. In this figure, maximum achievable utilization value depends on the maximum batch size of a given execution unit.

Fig. 5(a) indicates that π_2 outperforms other policies at $p(c) = 0.8$ by prioritizing the faster CPUs and utilizing less of the larger batch sizes of the GPUs. As a result, it spends less time computing the rarely occurring and computationally expensive task F. However, there might also be other differences that are not visible in the figures. As far as the authors see, there is no fundamental reason for why π_4 cannot also learn this behavior, indicating that there is room for more finetuning. Further hyperparameter tuning, or better training strategies such as importance sampling the relatively more difficult values of $p(c)$ could result in a more optimal policy.

7 Conclusions

In this paper, we described our design and results for a DRL-based scheduler for conditional task graphs in a heterogeneous computing cluster.

Our experiments show that runtime changes in the conditional task graph transition probabilities can significantly affect application performance. We show that these effects can be reduced by two simple methods. First, training with randomized properties helps the model to generalize, but is not sufficient on its own. Second, keeping a running average of the properties is enough to help the model perform better in edge cases. With and without these modifications, our model learns to beat the baselines in a fraction of an hour.

The time it takes to train a scheduler affects its adaptability. Adaptability of the scheduler can be important, if the system of sensors, actuators and computational cluster faces frequent changes. With hand-tuned schedulers, it can take a lot of effort to tune it after each change, often leading to the use of more generalized, less optimal solutions. However, with a DRL-scheduler that is fast to train, it could be possible to create an optimized solution for each variation with low effort.

We think that the use of conditional TGs can become more common with the use of larger machine learning models. As in our use case, we can save a significant amount of computational resources by having a conditional DAG, where most of the data runs through the smaller models and the larger models later in the hierarchy are used more sparingly.

As future work, we suggest evaluating the design in more complex environments with more dynamic transition probabilities and against more complex, hand-tuned baselines. If necessary, adding a GNN as the input layer could make the policy scale better to a larger DAG or multiple different DAGs.

References

[1] Y.-C. Tian and D. C. Levy, eds. *Handbook of Real-Time Computing.* Springer, 2022. DOI: 10.1007/978-981-287-251-7.

[2] W. Chen, X. Qiu, T. Cai, H.-N. Dai, Z. Zheng, and Y. Zhang. “Deep Reinforcement Learning for Internet of Things: A Comprehensive Survey”. In: *IEEE Communications Surveys and Tutorials* 23.3 (2021). DOI: 10.1109/COMST.2021.3073036.

[3] R. A. Khalil, N. Saeed, M. Masood, Y. M. Fard, M.-S. Alouini, and T. Y. Al-Naffouri. “Deep Learning in the Industrial Internet of Things: Potentials, Challenges, and Emerging Applications”. In: *IEEE Internet of Things Journal* 8.14 (2021). DOI: 10.1109/JIOT.2021.3051414.

[4] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. “Learning scheduling algorithms for data processing clusters”. In: *ACM SIGCOMM.* Beijing, China, 2019. DOI: 10.1145/3341302.3342080.

[5] Y. Duan and J. Wu. “Reducing average job completion time for DAG-style jobs by adding idle slots”. In: *IEEE Global Communications Conference (GLOBECOM).* 2022. DOI: 10.1109/globecom48099.2022.10001196.

[6] Z. Lin, C. Li, L. Tian, and B. Zhang. “A scheduling algorithm based on reinforcement learning for heterogeneous environments”. In: *Appl. Soft Comput.* 130 (Nov. 2022), p. 109707. DOI: 10.1016/j.asoc.2022.109707.

[7] B. Jamil, H. Ijaz, M. Shojafar, and K. Munir. “IRATS: A DRL-based intelligent priority and deadline-aware online resource allocation and task scheduling algorithm in a vehicular fog network”. In: *Ad Hoc Networks* 141 (2023), p. 103090. DOI: 10.1016/j.adhoc.2023.103090.

[8] M. Lombardi and M. Milano. “Allocation and scheduling of Conditional Task Graphs”. In: *Artificial Intelligence* 174.7 (2010), pp. 500–529. DOI: 10.1016/j.artint.2010.02.004.

[9] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* en. MIT Press, 2018.

[10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. “OpenAI Gym”. In: (2016). DOI: 10.48550/arXiv.1606.01540. eprint: arXiv:1606.01540.

[11] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, and I. Stoica. “RLlib: Abstractions for Distributed Reinforcement Learning”. In: *Proceedings of the 35th International Conference on Machine Learning.* Vol. 80. Proceedings of Machine Learning Research. PMLR, July 2018, pp. 3053–3062. URL: https://proceedings.mlr.press/v80/liang18b.html.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017). DOI: https://doi.org/10.48550/arXiv.1707.06347.

[13] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. “A Comprehensive Survey on Graph Neural Networks”. In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (2021), pp. 4–24. DOI: 10.1109/TNNLS.2020.2978386.

[14] H. Topcuoglu, S. Hariri, and M.-Y. Wu. “Performance-effective and low-complexity task scheduling for heterogeneous computing”. In: *IEEE Transactions on Parallel and Distributed Systems* 13.3 (2002), pp. 260–274. DOI: 10.1109/71.993206.

[15] J. Kosaian, A. Phanishayee, M. Philipose, D. Dey, and R. Vinayak. “Boosting the throughput and accelerator utilization of specialized cnn inference beyond increasing batch size”. In: *International Conference on Machine Learning.* PMLR. 2021, pp. 5731–5741. URL: https://proceedings.mlr.press/v139/kosaian21a.html.

[16] S. Huang and S. Ontañón. “A Closer Look at Invalid Action Masking in Policy Gradient Algorithms”. In: *The International FLAIRS Conference Proceedings* 35 (May 2022). DOI: 10.32473/flairs.v35i.130584.

[17] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. “Ray: A Distributed Framework for Emerging AI Applications”. In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation.* OSDI’18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 561–577. DOI: 10.5555/3291168.3291210.