The Transactions of the Korea Information Processing Society
TKIPS Vol.14, No.1, pp.1~8
eISSN: 3022-7011
https://doi.org/10.3745/TKIPS.2025.14.1.1

ADaPT: An Automated Dataloader Parameter Tuning Framework using AVL Tree-based Search Algorithms

MyungHoon Ryu† · XinYu Piao† · JooYoung Park†† · DoangJoo Synn††† · Jong-Kook Kim††††

ABSTRACT

Recently deep learning has become widely used in many research fields and businesses. To improve the performance of deep learning, one of the critical challenges is to determine the optimal values of many parameters that can be adjusted. This paper focuses on the adjustable dataloader parameters that affect the overall training time and proposes an automated dataloader parameter tuning framework, called *ADaPT*, to determine the optimal values of the dataloader parameters. The proposed ADaPT uses the characteristic of the AVL tree and attempts to determine optimal dataloader parameters in a small amount of time to accelerate the data loading. The results show that the proposed method effectively accelerates the data loading speed compared to the default parameter values and values that are recommended by PyTorch and the speed is comparable to the optimal.

Keywords : Deep Learning, AVL-Tree Search Algorithm, Data Loader, Parameter Tuning

ADaPT: AVL 트리 기반의 탐색 알고리즘을 활용한 자동 데이터로더 매개변수 튜닝 프레임워크

유 명 훈† · 박 흠 우† · 박 주 영†† · 신 동 주††† · 김 종 국††††

요 약

최근 딥러닝은 많은 연구 분야와 산업에서 널리 사용되고 있다. 딥러닝의 성능 향상을 위해 중요한 과제 중 하나로는 조정이 가능한 많은 매개변수 중 최적의 값을 결정하는 것이다. 본 논문에서는 전체 학습 시간에 영향을 미치는 조정 가능한 데이터로더 매개변수에 초점을 맞추고 최적값을 결정하는 자동 데이터로더 매개변수 튜닝 프레임워크 ADaPT(Automated Dataloader Parameter Tuning framework)를 제안한다. 본 논문에서 데이터 로딩 속도 가속화를 위해 AVL 트리 기반의 탐색 알고리즘을 사용했으며, 이 알고리즘은 AVL 트리의 짧은 탐색 높이를 활용하여 ADaPT가 짧은 시간 내에 최적의 매개변수를 찾도록 한다. 실험 결과에서 기존의 매개변수 값과 PyTorch에서 권장하는 매개변수 값에 비해 제안한 방법이 더 효과적으로 데이터 로딩 가속했으며, 최적의 매개변수로 구성한 데이터로더와 비슷한 성능을 보여준다.

키워드 : 딥러닝, AVL 트리 기반 탐색 알고리즘, 데이터 로더, 매개변수 튜닝

1. Introduction

In recent years, the variety of datasets collected by many enterprises has been increasing. To use and process this so called Big Data, the need for deep neural networks (DNNs) is also increasing. Also, bigger datasets are required to correctly train more complex DNNs [18], which leads to the need for large-scale computer systems that consist of more computing power and more memory. Thus, modern computer systems for deep learning have begun to use multi-CPUs and multi-GPUs. Many learning techniques are also introduced for the efficient use of such systems, such as data parallelism [1-3], model parallelism [4-7], and pipeline parallelism [8-10].

With datasets becoming larger and larger, data loading time is one of the most crucial challenges to speeding up deep learning. Modern deep learning framework, such as

※ 이 논문은 교육부가 지원하는 한국연구재단의 기초과학연구사업(5년)의 지원을 받아 수행되었음.
† 비 회 원 : 고려대학교 전기전자공학과 석박사통합과정
†† 정 회 원 : 난양기술대학교 컴퓨터공학과 박사과정
††† 비 회 원 : 조지아공과대학 컴퓨터공학과 박사과정
†††† 종신회원 : 고려대학교 전기전자공학부 교수
Manuscript Received : August 26, 2024
First Revision : October 17, 2024
Accepted : October 22, 2024
* Corresponding Author : Jong-Kook Kim(jongkook@korea.ac.kr)

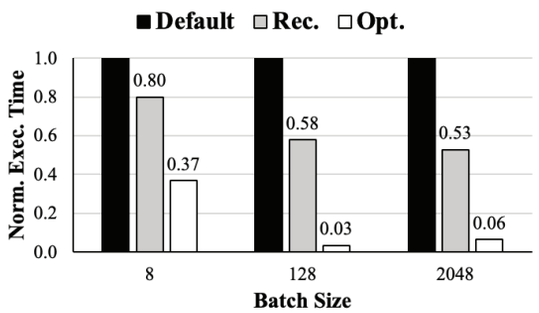


Fig. 1. Normalized execution time for the dataloader when using default, recommended (Rec.), and the optimal values (Opt.), respectively. This result was measured using CIFAR-10 dataset with 32 by 32 image size and 8, 128, 2048 batch sizes.

Tensorflow [11] and PyTorch [12],have many adjustable parameters that effect the overall performance of deep learning (e.g., [13, 14]). A dataloader is used to load data items into main memory and group them as a mini-batch for deep learning models to use. It has a lot of adjustable parameters that may affect the overall execution time of deep learning. In PyTorch, the recommended number of workers and prefetch factor, which are adjustable parameters for the dataloader, are half number of CPU cores and 2, respectively. Due to the variety of types of datasets, however, these recommended parameter values are often not the optimal. The variety of variables in a computer system (e.g., number of CPUs or GPUs, system memory size, system I/O performance, etc.) also make it difficult for users to determine the optimal parameter values for the dataloader. Fig. 1 shows that the normalized execution time of a dataloader with optimal parameters can be accelerated up to 4.73× and 2.76× compared to a dataloader with default and recommended parameters, respectively. This result suggests that tuning dataloader parameters may be a solution to improve the overall speed in deep learning.

Recently, many approaches are proposed such as data sampling [19, 20] and caching [20, 21]. Some research such as DeepLake [22] and Webdataset [24] converts the format of the dataset to stream the data and uses the recommended parameters for the dataloader. The data loading time of these methods (from CPU to GPU) will be the same as the dataloader that uses recommended parameters. Another method is a DALI [23] which avoids the pre-processing bottleneck in the CPU by sending all the data to the GPU and processes on the GPU itself. This makes it difficult to compare the performance of data loading from the CPU to GPU. This paper addresses improving the data loading rate by tuning various parameters of the dataloader on the CPU side.

This paper introduces an Automated Dataloader Parameter Tuning framework, *ADaPT*, that attempts to determine the optimal parameter values for the particular dataloader. ADaPT adjusts four different parameters to maximize and improve the effectiveness of a dataloader. The optimal values can be found using a grid search algorithm but it is too time-consuming. This is because a grid search tries all combinations of adjustable parameters that can be configured to find the optimal parameters in the GPU. To address this problem, the ADaPT uses an AVL tree and its characteristic to achieve speedup comparable to optimal parameters and reduces the time to determine near optimal parameter values.

The contributions of this paper are as follows:

1) To reduce the overall training time of a deep learning model, parameter tuning of the dataloader is investigated and certain parameters are identified that affect the performance.
2) An optimal parameter combination of tunable values are found using grid search.
3) An AVL-tree based search algorithm is proposed to reduce searching time while achieving near optimal performance.

This paper is organized as follows. The concept of the dataloader and the adjustable parameters are described in Section 2. Section 3 introduces the proposed ADaPT and main search algorithm. Experimental results are depicted in Section 4. Finally, Section 5 concludes this paper.

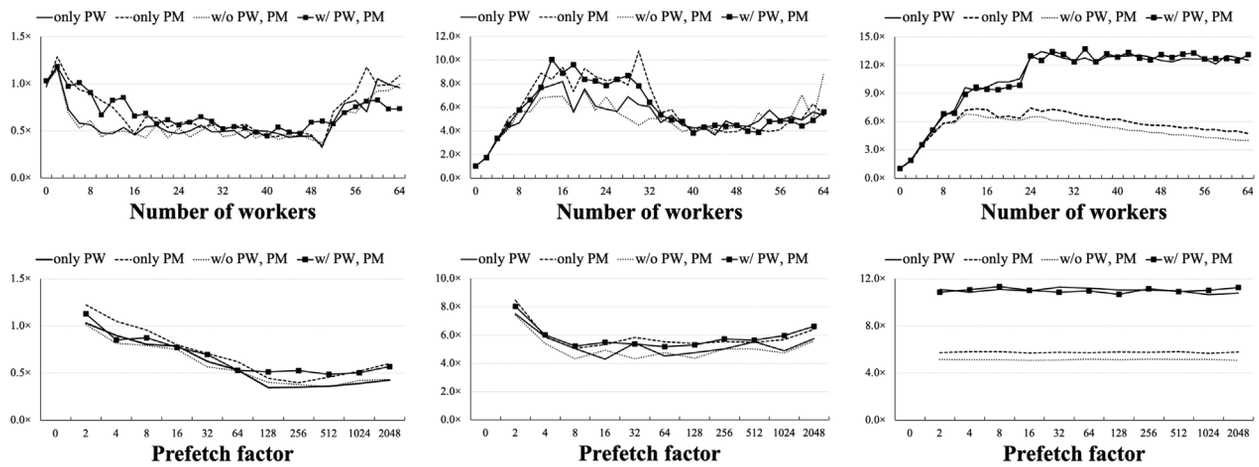
2. Dataloader

2.1 Procedure of a Dataloader

The process of the dataloader usually consists of four steps. First, data items are sequentially loaded from storage into main memory. Second, the loaded data is transformed into a suitable data format for future processes (e.g., padding, tensorization, or normalization). Then, the dataloader groups data loaded into main memory as a mini-batch. Finally, the dataloader transfers a mini-batch to the target device such as GPUs.

2.2 Adjustable Parameters

To improve the performance of dataloader in PyTorch [12], this paper focuses on four adjustable parameters that



(a) 8 batch sizes (b) 128 batch sizes (c) 2048 batch sizes

Fig. 2. Comparison of performance improvements for each adjustable parameter.

are dependent on system configurations for performance. First, *num_workers* (N_W) is an integer-based parameter that indicates the number of subprocesses (i.e., workers). These workers are spawned to load data in parallel. Each worker loads data from storage to main memory and transforms them into a suitable data format (called Tensor) for use in deep learning models. If *num_workers* is 0, the dataloader uses the main process to load and transform data. Second, *prefetch_factor* (N_F) is an integer-based parameter that indicates the total number of mini-batches equally distributed and pre-loaded to all workers. This parameter may be useful for dataloder to pre-load next mini-batches to accelerate data loading speed. Third, *pin_memory* (*PM*) is a boolean-based parameter that allows a dataloader to copy the loaded mini-batch into the fixed host memory (i.e., pinned memory) before transferring the mini-batch to a device. Pinned memory is useful when transfering data, in particular large data, from the host memory to the device memory. The last parameter is *persistent_workers* (*PW*), a boolean-based parameter, which allows workers to be reused by running the worker permanently. As default, all workers for loading data are performed once and terminated when data loading is complete. In the PyTorch framework, *prefetch_factor* (N_F) and *persistent_workers* (*PW*) are not available when *num_workers* (N_W) is 0.

The reason why the above parameters are chosen is because they do not directly affect the performance of deep learning models. Among many adjustable parameters in dataloaders, *batch_size* affects data loading performance (see Table 1). However, as reported in many previous studies [13, 14], adjusting *batch_size* affects performance of deep learning models. Thus, this paper does not investigate *batch_size* and only system parameters related to accelerating dataloader performance.

2.3 Impact of Adjustable Parameters

In Fig. 2, the results show the impact of each parameter for the dataloader. The experiments of these results use CIFAR-10 dataset with 32×32 image sizes and three different batch sizes (8, 128, 2048), and are tested on the system described in Section 4. In 8 batch sizes, small parameter values for *num_workers* and *prefetch_factor* have a significant impact on the performance improvement of the dataloader compared to parameters for *pin_memory* and *persistent_workers* (see Fig. 2a). On the other hand, in 2048 batch sizes, the *num_workers* and *persistent_workers* have a significant impact on the performance improvement compared to *pin_memory* and *prefetch_factor* (see Fig. 2c).

These results suggest that it can be difficult for users to determine optimal parameter values for the dataloader given a dataset. It is also difficult to find the optimal parameters by manually substituting for all combinations of the candidate parameters. Therefore, it is important to determine the optimal dataloader parameters to improve the speedup for deep learning.

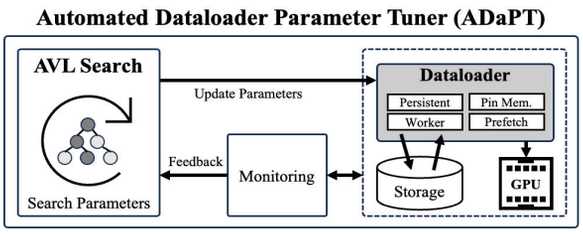


Fig. 3. An overview of the proposed ADaPT.

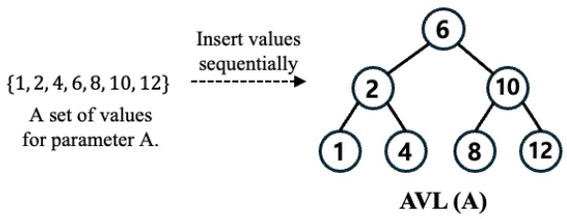


Fig. 4. An example of AVL tree generation.

3. The Proposed Framework

3.1 Overview

The ADaPT is a framework to automatically search for the optimal combination of dataloader parameters using AVL tree-based search in a strictly guaranteed short amount of time. Fig. 3 shows an overview of the ADaPT, which works in two main processes; i) monitoring, and ii) parameter searching. The monitoring process measures the total execution time of dataloader (e.g., data loading, pre-processing, and data transfer), and then feedbacks the measured time to the parameter searching process. In the process of parameter searching, the measured time is used to find the optimal combination of parameters using an AVL tree-based search algorithm. The ADaPT iterates between these two processes until it finds the optimal combination of parameter values or arrives at the final leaf node.

3.2 AVL Tree-based Search

An AVL tree is a binary search tree, with two main characteristics; (i) the maximum height difference between the left and right subtrees is 1, and (ii) the parent node has a value larger than the left node and smaller than the right child node. Using the first characteristic, the best parameters can be found in a short time by reducing the number of parameter combinations to be searched. Based on our observations, in addition, future sub-combinations of current parameter combinations that performed poorly tend to continue to perform poorly. For examples, in 128 batch sizes (see Fig. 2(b)), parameter combinations with batch sizes smaller than 32 perform better than parameter combinations with batch sizes larger than 32. Thus, using the second characteristic of the AVL tree, it is possible to predict that future sub-combinations of current parameter combinations that performed poorly will still perform poorly, dramatically reducing the number of unnecessary searches. This is possible because the AVL tree maintains a height of $\log_2 N$ and guarantees the time complexity of $O(\log_2 N)$ for insertion, search, and deletion. Therefore, the total time complexity of AVL tree search is $O((\log_2 N)^k)$ in the worst case when k parameters need to be found.

Note that the proposed approach does not guarantee to search the optimal parameter combinations, but it is possible to find the best parameter combinations that can perform as well as the optimal parameter combinations.

3.3 The Process of Parameter Searching

In the beginning of this algorithm, ADaPT generates AVL trees for each parameters (is shown in Fig. 4). For example, if the number of adjustable parameters is k and the number of available values for each parameter is N, the ADaPT generates k AVL trees with N nodes, where each node has an adjustable value for each parameter. Then, each AVL tree is given a priority for search. Higher priority is given to more frequently searched trees (see Fig. 5).

In the process of parameter search, attempting to search for optimal combination of parameters, AVL tree with lower priority sends the value of currently selected node to the root node of the AVL tree with higher priority (step ① as shown in Fig. 5). This process is repeated until the AVL tree with the highest priority is reached and continues to send values cumulatively up to the value received from lower priority AVL trees. Then, it passes all of the accumulated parameter values to the dataloader to measure the execution time (see step ①~② in Fig. 5). When the execution time for a node is completed, the execution time of its children nodes are determined in the same way.

When execution time of the current and children nodes are determined, the algorithm selects the node with the lowest execution time (see step ③ in Fig. 5). If the parent node is selected, the algorithm does not search any further and returns the measured time and parameter

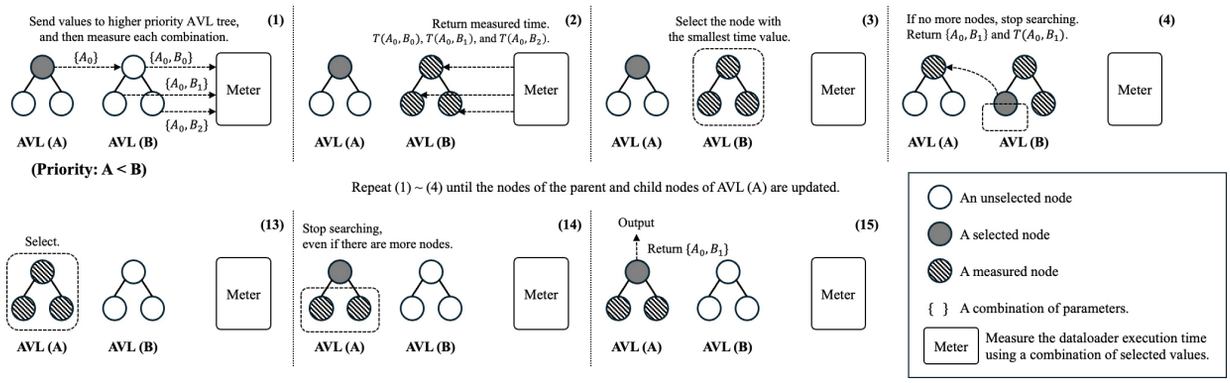


Fig. 5. The overall process of the AVL tree-based search for searching the dataloader parameters combination.

combination to higher priority AVL tree. On the other hand, if one of children nodes is selected, the algorithm measures the children of the child node. If there are no more nodes to search, the algorithm stops searching and returns the measured time and parameter combination to higher prioritized AVL tree (see step ④ in Fig. 5). The algorithm continues to repeat the above processes until the lowest priority AVL tree returns the parameter combination (see step ⑮ in Fig. 5). This returned parameter combination from the lowest priority AVL tree will the best parameter combination found using ADaPT.

This proposed algorithm has two stopping conditions; One is when there are no more searchable nodes, and the other is when the current node is selected (is described in step ④ and ⑭ in Fig. 5). The reason why the algorithm stops searching when current node is selected is because it assumes that the parameter combinations of later nodes will not provide better speedup.

4. Evaluation

4.1 Experimental Setup

System. The experiments tested on a system, which consists of Threadripper 3970 3.7GHz 32-Core 64-Thread processor, Titan X GPU with GDDR5 12GB memory, and 128GB main memory. The ADaPT was implementation by CUDA 11.8 version and Pytorch 2.1.2 version.

Datasets. To evaluate ADaPT performance, CIFAR-10 [15] dataset is used in this paper. CIFAR-10 dataset is widely used in image classification tasks. CIFAR-10 dataset consists of 60,000 32×32 color images in 10 classes. The image size of CIFAR-10 dataset was fixed at 32×32, which is widely used [16, 17].

Parameters. As described in Section 2, the proposed ADaPT aims to find and adjust four different parameters; *num_workers* (N_W), *prefetch_factor* (N_F), *pin_memory* (PM), *persistent_workers* (PW). The *num_workers* (N_W) is starts from 1 and from 2 to 64 it is increased by 2. The *prefetch_factor* (N_F) begins with 1 and it is increased by 2 times up to 2,048. Both *pin_memory* (PM) and *persistent_workers* (PW) are boolean-based parameters (0 (False) or 1 (True)).

Metrics. This paper only evaluates the execution time of the dataloader that includes the time to load data items from storage into main memory, preprocess them, and transfer them to the GPU. This paper also compares the performance between dataloaders using the default, recommended (Rec.), optimal (Opt.), and ADaPT parameter combinations. The recommended parameters use values mentioned on the PyTorch [12], and optimal parameters use values found by the grid search. This work also measures the time to find the optimal parameters for the grid search and the time to find the best values for the ADaPT. All experiments for the ADaPT and grid search are run 10 times and the execution time shown is the averaged value. The search time for the ADaPT and grid search is the time used to find the best combination.

Note that the purpose of the proposed ADaPT is to maximize performance (speed) of the dataloader, not to improve the training performance (e.g., accuracy) of deep learning models.

4.2 Experimental Results

Table 1 shows the results of the dataloader performance for 8, 128, and 2048 batch sizes, respectively. In the table, 'Exec. time' is the total execution time of dataloaders. In 8 batch sizes, the ADaPT shows 1.68× and 1.69× faster than using default and recommended parameters, respectively. For 2048 batch sizes the ADaPT

Table 1. The results show the performance of the dataloader for each batch size.

Batch sizes		8				128				2048			
Methods		Default	Rec.	Opt.	ADaPT	Default	Rec.	Opt.	ADaPT	Default	Rec.	Opt.	ADaPT
Exec. time (ms)		1.14	1.15	0.67	0.68	15.38	1.71	1.30	1.35	252.52	41.54	6.52	6.82
Search time (sec)		-	-	1,688	127	-	-	1,761	91	-	-	3,801	530
Best Params. Found	N_W	0	32	14	10	0	32	14	14	0	32	30	30
	N_F	0	2	2	2	0	2	128	128	0	2	512	128
	PW	False	False	True	False	False	False	False	False	False	False	True	True
	PM	False	False	False	False	False	False	True	True	False	False	True	True
Ranking for Parameter Combinations		359th	365th	1st	16th	1537th	165th	1st	1st	1537th	974th	1st	21st

performed significantly better than using default and recommended parameters, with 37.03× and 6.09× more fast, respectively. Compared to the results of the optimal parameters, the ADaPT shows comparable performance. In small 8 batch sizes, ADaPT found the 16th ranked combination listed by grid search. In large 2048 batch sizes, the ADaPT determined that the best combination of parameters for the 21st rank. The ADaPT selected the 1st best combination of parameters in 128 batch sizes which is the same combination found by grid search method. This result indicates that the ADaPT selects the best combination of parameters where the dataloader execution speed is similar to the optimal combination of parameters. It should be noted that the execution time of the best combination is calculated as an average; therefore, the execution time of ADaPT and the optimal combination may not necessarily be the same even if ADaPT finds the optimal combination.

Table 1 also shows the overhead of searching optimal parameters between the grid search and AVL tree-based search. In 8 batch sizes, the AVL tree-based search shows 13.29× faster in searching time compared to a grid search. This is also 19.35× and 7.17× faster in other batch sizes, respectively. This result means that the AVL tree-based search can find optimal combination of parameters faster than a grid search.

As the above results, the ADaPT does not guarantee that it finds the optimal combination of parameters that the grid finds, but it can find the combination of parameters that performs similar in a short amount of time. In general, the time complexity of the grid search is $O(N^k)$ in the worst case. In this work, the time complexity of a grid search can be represented as $O(N^2)$ because both *pin_memory* (*PM*) and *persistent_workers* (*PW*) are boolean-based adjustable parameters. On the other hand, the time complexity of the proposed searching algorithm is $O((\log_2 N)^2)$ in the worst case.

5. Conclusion

In this paper, an Automated Dataloader Parameter Tuning framework called ADaPT is proposed to find a useful combination of adjustable parameters for a dataloader by using an AVL tree-based search algorithm method. The experimental results show that the proposed approach succeeds in searching the best parameter combinations that performed as well as the optimal parameter combinations in a short amount of time.

In the future, we expect that ADaPT will be helpful in the research of deep learning system optimization for the reduction of data transmission time from the CPU to GPU.

References

[1] Q. Ho et al., "More effective distributed ML via a stale synchronous parallel parameter server," *Advances in Neural Information Processing Systems*, Vol.26, pp.1223-1231, 2013.

[2] Z. Huo, B. Gu, and H. Huang, "Decoupled parallel back-propagation with convergence guarantee," in *International Conference on Machine Learning*, pp.2098-2106, 2018.

[3] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *Proceedings of Machine Learning and Systems*, Vol.1, pp.1-13, 2019.

[4] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp.571-582, 2014.

[5] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, Vol.52, No.4, pp.1-43, 2019.

[6] J. Dean et al., "Large scale distributed deep networks," *Advances in Neural Information Processing Systems*, Vol.25, 2012.

[7] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," *Advances in Neural Information Processing Systems*, Vol.27, 2014.

[8] Y. Huang et al., "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in Neural Information Processing Systems*, Vol.32, 2019.

[9] D. Narayanan et al., "Pipedream: Generalized pipeline parallelism for DNN training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp.1-15, 2019.

[10] X. Piao, D. Synn, J. Park, and J.-K. Kim, "Enabling large batch size training for DNN models beyond the memory limit while maintaining performance," *IEEE Access*, 2023.

[11] M. Abadi et al., "TensorFlow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp.265-283, 2016.

[12] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, Vol.32, 2019.

[13] C. Peng et al., "Megdet: A large mini-batch object detector," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp.6181-6189, 2018.

[14] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *International Conference on Machine Learning*, pp.1597-1607, 2020.

[15] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[16] A. Srinivas, T.-Y. Lin, N. Parmar, J. Shlens, P. Abbeel, and A. Vaswani, "Bottleneck transformers for visual recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp.16519-16529, 2021.

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp.770-778, 2016.

[18] A. Alwosheel, S. van Cranenburgh, and C. G. Chorus, "Is your dataset big enough? Sample size requirements when using artificial neural networks for discrete choice analysis," *Journal of Choice Modelling*, Vol.28, pp.167-182, 2018.

[19] C. C. Yang and G. Cong, "Accelerating data loading in deep neural network training," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp.235-245, 2019.

[20] J. Xu, G. Wang, Y. Yao, Z. Li, C. Cao, and H. Tong, "A deep learning dataloader with shared data preparation," *Advances in Neural Information Processing Systems*, Vol.35, pp.17146-17156, 2022.

[21] G. Leclerc, A. Ilyas, L. Engstrom, S. M. Park, H. Salman, and A. Mądry, "FFCV: Accelerating training by removing data bottlenecks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp.12011-12020, 2023.

[22] S. Hambardzumyan et al., "Deep lake: A lakehouse for deep learning," 2022.

[23] NVIDIA, "DALI: NVIDIA Data Loading Library," 2018. [Online]. Available: https://developer.nvidia.com/dali.

[24] "Webdataset format." [Online]. Available: https://github.com/webdataset/webdataset.

MyungHoon Ryu

https://orcid.org/0009-0000-6872-1830
e-mail : rynmh10@korea.ac.kr

He received the B.S. degree from the School of Electrical Engineering, Soonchunhyang University, in 2024 and is currently pursuing the Ph.D. degree from the Department of Electrical and Computer Engineering, Korea University. His research interests include parallel and distributed processing, system optimization, and systems for machine learning.

XinYu Piao

https://orcid.org/0009-0000-7502-4080
e-mail : xypiao97@korea.ac.kr

He received the B.S. degree from the School of Electrical Engineering, Korea University, in 2020 and is currently pursuing the Ph.D. degree from the Department of Electrical and Computer Engineering, Korea University. From 2019 to 2020, he was an Undergraduate

Student Research Assistant with the Compiler and Microarchitecture Laboratory, Korea University. His research interests include systems for artificial intelligence (AI), algorithms for deep learning, hardware resource management, and cloud computing.

JooYoung Park

https://orcid.org/0000-0002-6979-9362
e-mail : jooyoung.park73@gmail.com

He received the B.S. degree from the School of Electrical Engineering, Korea University, Seoul, South Korea, in 2023. He is currently a Ph.D. student in School of Computer Science and Engineering at Nanyang Technological University, Singapore. From 2020 to 2022, he was an Undergraduate Student Research Assistant with the High Performance and Intelligent Computing Laboratory, Korea University. His research interests include cloud and serverless computing, heterogeneous distributed computing, and systems for machine learning.

DoangJoo Synn

https://orcid.org/0000-0002-8992-4634
e-mail : alansynn@gatech.edu

He received the B.S. and M.S. degrees from the School of Electrical Engineering, Korea University, Seoul, in 2020 and 2023, respectively. He is currently a Ph.D. student in Computer Science at Georgia Tech. From 2012 to 2014, he was a Software Engineer at the KAIST Engineering School. From 2017 to 2020, he was a Software Engineer at the OS and Production Cells, AKA Intelligence inc., and Flysher inc. From 2020 to 2021, he was a AWS Cloud Student Ambassador at the Amazon Web Services. From 2020 to 2022, he was a Research Assistant with the High Performance and Intelligent Computing Laboratory, Korea University. From 2021 to 2022, he was a Research Engineer with the Propia AI inc. Austin, Texas. His research interests include distributed computing, machine learning systems, cloud computing, and hyperscalar computing.

Jong-Kook Kim

https://orcid.org/0000-0003-1828-7807
e-mail : jongkook@korea.ac.kr

He (Senior Member, IEEE and ACM) received the B.S. degree from the Department of Electronic Engineering, Korea University, Seoul, South Korea, in 1998, and the M.S. and Ph.D. degrees from the School of Electrical and Computer Engineering, Purdue University, West Lafayette, USA, in 2000 and 2004, respectively. He was at the Samsung SDS's IT Research and Development Center, from 2005 to 2007. He is currently a Professor with the School of Electrical Engineering, Korea University, where he joined in 2007. His research interests include heterogeneous distributed computing, energy-aware computing, resource management, evolutionary heuristics, distributed mobile computing, artificial neural networks, efficient deep learning, systems for AI and distributed robot systems.