Profile-guided memory optimization for deep neural networks

Taro Sekiyama Takashi Imamichi Haruki Imai Rudy Raymond

Recent years have seen deep neural networks (DNNs) becoming wider and deeper to achieve better performance in many AI applications. Such DNNs however require large amount of memory to store weights and intermediate results (e.g., activations, feature maps, etc.) in propagation. This requirement makes it difficult to run the DNNs on devices with limited and hard-to-extend memory, degrades the running time performance, as well as restricts the design of network models. We address this challenge by developing a novel profile-guided memory optimization to efficiently and quickly allocate memory blocks during the propagation in DNNs. The optimization utilizes a simple and fast heuristic algorithm based on the two-dimensional rectangle packing problem. Experimenting with well-known neural network models, we confirm that our method can reduce the memory consumption, and at some cases it can also accelerate training thanks to the ability to use larger mini-batch sizes.

1 Introduction

Since its great success in computer vision [19], deep learning, the machine learning technology based on *deep neural networks* (DNNs), has emerged widely in image processing, machine translation, speech recognition, and many AI applications. The effective use of graphical processing units (GPUs) has made it possible to train sophisticated DNNs on huge datasets [19][20]. Along with the progress of research to obtain better accuracy, DNNs are becoming *deeper* and/or *wider*. For example, in image recognition, AlexNet [19], the winner of ILSVRC 2012, consists of only nine layers and has a sequential structure; GoogLeNet [28] introduces the so-called inception modules, which are a technique to widen DNNs; ResNet [16] consists of more than 50 layers; and the more recent network, Inception-ResNet [27], which extends ResNet with the inception modules, is even larger than ResNet and GoogLeNet.

Although expanding neural networks seems to be a key to obtain better accuracy, it comes with the high memory cost required to store weight parameters and intermediate results in the propagation for training and inference. This gives rise to several undesirable consequences. First, for training DNNs, we can only use smaller mini-batches to avoid running out of memory, which results in the inefficient use of GPUs that can lead to slow convergence, especially, in the distributed training that adopts SGD algorithms for large mini-batches [13][2]. Second, inference in the deployment environment may require many machines equipped with GPUs having a large amount of device memory. Third, the flexibility of the design of neural networks is constrained so that the DNNs can fit in the memory

* プロファイルに基づくニューラルネットワークのためのメモリ最適化

関山 太朗, 国立情報学研究所, National Institute of Informatics. This work was partially done in IBM Research – Tokyo.

今道 貴司，今井 晴基，Rudy Raymond, IBM 東京基礎研究所, IBM Research – Tokyo.

of the underlying devices. The high memory consumption is more serious in the use of GPUs and edge devices that have much smaller and less extendable memory storage than CPUs.†1

We study memory optimization for DNNs. Our approach is based on the observation that propagation of a network model is computed in the same way for different inputs and different learnable parameters; we call such propagation *hot*.†2 This is indeed the case in many neural networks including convolutional neural networks (CNNs). On the basis of this observation, we profile the memory usage (e.g., when a memory block is requested and released) in a sample run and then utilize the profile to find an assignment of memory addresses to requests for memory blocks in the succeeding runs. To minimize the peak memory usage, we reduce the memory assignment problem to *Dynamic Storage Allocation Problem* (DSA), a special case of a packing problem that is known to be NP-hard. Solutions to DSA give the amount of the entire memory and offsets within it for memory blocks requested by the propagation. After solving DSA, we allocate the entire memory and return a memory address for each memory request in the succeeding runs according to the solution to DSA.

Our contributions to overcome the aforementioned consequences are summarized as follows.

- We propose a novel profile-guided memory optimization technique for DNNs. Our approach optimizes memory usage in a hot part of a propagation and never incurs performance overhead once the memory usage is optimized, while preserving the computation of the DNNs. Using the profiling result, we reduce the memory allocation problem to DSA and solve it by a heuristic based on the two-dimensional rectangle packing. We empirically show that the heuristic works well from the perspectives of both computation time and solution quality. Even when the *whole* propagation is not hot, there are cases where some *part* of the propagation is hot. We develop workarounds to optimize the memory usage of propagation that involves non-hot computation so that our method can be also used in a recurrent neural network (RNN) with long short-term memory (LSTM) units [17].
- We implement the memory optimization on two common deep learning frameworks: Chainer [29] and PyTorch [22]. We conduct experiments showing the effectiveness on training and inference using four CNNs (AlexNet, ResNet-50, and Inception-ResNet) and one RNN (seq2seq [26]). We find that our method can reduce the memory consumption during training and inference, and, at some cases, it can also accelerate propagation.

The rest of this paper is organized as follows. We describe related work in Section 2 and introduce our approach in Section 3. Section 4 gives an implementation of our idea and explains how to apply it to DNNs involving non-hot propagation. Section 5 shows experimental results, and Section 6 concludes this paper.

2 Related work

Our proposed method deals with memory management for DNNs by leveraging a heuristics based on packing algorithms. The main characteristic of memory management for deep learning applications is the need to allocate many large and small memory blocks, and for RNNs to deal with variable-length data. A popular technique used in high-performance computing to pre-allocate and hold a

†1 Augmenting devices may mitigate the problem but it introduces another issue about the communication between devices.

†2 This term originates from just-in-time compilation, where repeatedly executed code blocks to be optimized are called hot.

few huge memory blocks for reuse in the entire run is thus not suitable. A better way of memory management for deep learning applications is *dynamic memory allocation* (DMA) (see [31] for a survey). Common deep learning frameworks adopt either or a combination of two major DMA algorithms: the sequential fit, adopted by Theano [4], and the segregated fit, adopted by Chainer, TensorFlow [1], and MXNet [7]; PyTorch is between them. However DMA algorithms have a drawback that is solved by our approach. Namely, they can cause *fragmentation*, which happens when a memory block is split into blocks of smaller sizes than sizes requested by an application program. On the other hand, our method can find contiguous allocation of memory blocks so that the peak memory usage is minimized thanks to its static memory allocation.

Memory reduction for DNNs. There have been extensive literatures on memory reduction for DNNs, such as, memory reuse by analysis of computational graphs [7], memory reduction by recomputation of intermediate outputs from hidden layers in backpropagation [8][21], a new backpropagation algorithm that reuses as many memory blocks allocated in forward propagation as possible for backpropagation [25], compression [15][14] and quantization [12] of DNNs. However, to our best knowledge, our approach appears to be complementary with the previous work as our method addresses allocation of memory blocks and is agnostic of how they are used. Some of the advantages of our method are as follows. First, unlike [7] that can only optimize the usage of memory for intermediate outputs, our approach can also optimize the usage of temporary memory to speed up convolution. Second, it does not incur running time overhead, unlike recomputation-based methods [8][21]. Third, it is applicable to both training and inference, whereas the backpropagation developed by [25] is only for training. Fourth, it does not change the computation involved by a DNN model, unlike compression and quantization [15][14][12], and does not need time-consuming retraining, unlike compression [15].

Offloading device memory not used immediately to a slower storage and prefetch it as necessary, is another way to run large models on a device with limited memory [23][21], but it can cause performance degradation due to CPU-GPU communication for data transfer. Although Unified Memory in NVIDIA CUDA allows more fine-grained offloading, but it incurs significant and difficult-to-control overhead [21]. Wang et al. [30] integrate computational graph analysis, out-of-core technology, and recomputation into one system (which has pros and cons of those methods inherently), but their technique seem to focus on CNNs, and not clear how to apply it to other NNs. In contract, our method is simpler and can be applied to RNNs.

Packing problem as memory allocation. Our method uses a heuristic of memory allocation developed to solve the *Dynamic Storage Allocation problem (DSA)*, a typical NP-hard problem [10]. DSA is a special case of *a two dimensional strip packing problem (2SP)*. The 2SP asks for a set of rectangular items to be placed in a container with a fixed width and for the variable height to be minimized. A memory block corresponds to a rectangular item with its allocation time as width and its memory size as height. The DSA deals with a special case where the allocation times of all memory blocks are fixed. Namely, all rectangular items must be placed at predetermined intervals and the choice of allocations are made by choosing the order of stacking the items. DSA has been studied mainly from the theoretical point of view. Gergov [11] proposed a 3-approximation algorithm for DSA whose running time is $O(n \log n)$ where n is the number of items. The algorithm is still considered as one of the state-of-the-art and has become the main foun-

dation for many theoretical analysis of DSA, such as, [5]. Unfortunately, such theoretical algorithms are often not practical due to high computational cost. Our proposed method is based on a heuristic for 2SP of Burk et al. [6] known as the *best-fit algorithm*. It works well for large-size instances even compared to metaheuristics-based algorithms, and thus good for practical purposes. Quite surprisingly, comparing to [11] our method often obtains better solutions (see Section 5 for details). With regards to 2SP, Arahori et al. [3] proposed a branch-and-bound-based exact algorithm, which works well for small and medium-sized instances. Gálvez et al. [9] proposed a pseudo-polynomial-time approximate algorithms, whose approximation ratio is $4/3 + \epsilon$.

3 Profile-guided memory allocation

We profile the memory usage (e.g., when a memory block is requested and released) in a sample run and then utilize the profile to find an assignment of memory addresses to requests for memory blocks in the succeeding runs by solving DSA. One may raise a concern that a sample run for a profile can be memory-inefficient and needs more memory than the physical capacity. We can obtain the profile even in such a case by utilizing an out-of-core technique [23][21] or Unified Memory in NVIDIA CUDA, which enables us to run the model requiring memory over the capacity with additional performance overhead, and then perform the succeeding runs without the overhead by disabling those techniques.

From a profile of memory usage during hot propagation, we gather the information of the memory blocks requested. Such information allows us to better determine where to allocate the memory blocks in the physical memory. Formally, we list the parameters as follows.

- $n \in \mathbb{Z}$: number of memory blocks.
- $B = \{1, \ldots, n\}$: a set of IDs of memory blocks.
- $W \in \mathbb{N}$: the available maximum memory size.
- $w_i \in \mathbb{N}$ $(i \in B)$: size of memory block i.
- $\underline{y_i} \in \mathbb{N}$ $(i \in B)$: time when i is requested.
- $\overline{y_i} \in \mathbb{N}$ $(i \in B)$: time when i is released.

We assume that these parameters do not change during the entire run (training and inference) of a neural network. This assumption is satisfied if the propagation involved by the neural network is hot. Many commonly used models satisfy this condition. We give workarounds for network models where only a part of the propagation is hot in Section 4. 3. A memory block i is allocated during a time period $[\underline{y_j}, \overline{y_j})$; we call the time period *lifetime* of memory block i.

We next introduce the following decision variables of DSA.

- $u \in \mathbb{Z}$: the peak memory usage.
- $x_i \in \mathbb{Z}$ $(i \in B)$: memory offset (or, starting address) of memory block i within the entire allocated memory.

We call the interval of memory address $[x_i, x_i + w_i)$ of memory block i *address space* of memory block i.

The objective of DSA is to assign memory offsets to memory blocks so that no two memory blocks occupy the same address space at any given time and the peak memory usage is minimized.

Because the number of memory blocks can be more than 1000, we need polynomial time algorithms to DSA and quadratic time or better is preferable in particular. In this paper, we study two methods: a mixed integer programming model by CPLEX and a new heuristic called best-fit heuristic to DSA. We design the new heuristic to DSA on the basis of the best-fit heuristic to 2SP [6] because it is known to be simple, fast, and effective to especially large 2SP instances and it is not complicated to adjust it for DSA. The best-fit heuristic to DSA repeats two operations until all memory

blocks are placed: (1) choosing an offset and (2) searching for a memory block that can be placed at the chosen offset without colliding with memory blocks placed already. The computational time complexity of the heuristic is quadratic in the number of memory blocks. Note that we do not utilize approximate algorithms to 2SP such as [9] because DSA and 2SP are different and we cannot apply them to DSA directly. We also do not take advantage of exact algorithm to 2SP such as [3][18] because they can cope with several tens items at most and there can be much more items in the memory allocation in DNN.

4 Implementation

We incorporate the best-fit heuristic in PyTorch and Chainer to optimize the GPU memory usage. This section describes the details including how to apply our approach to *any* network models.

4. 1 Memory profiling

Since PyTorch and Chainer allocate memory blocks at run time, we profile GPU memory usage by monitoring memory allocation and free operations in a sample run. To obtain memory request time $\underline{y_i}$ and release time $\overline{y_i}$, we use a global integer variable y, which represents the current time and is increased after each allocation and free. We also have a global integer variable λ that denotes the ID of the next requested memory block.

Given a sample input, we initialize the global variables with one and run the model with the input. When receiving a request with memory size s, we extend B (the set of memory block IDs) with λ, set s and y to w_λ and $\underline{y_\lambda}$, respectively, and finally increase λ and y. When memory block i is released, y is set to $\overline{y_i}$ and then increased.

4. 2 Memory allocation

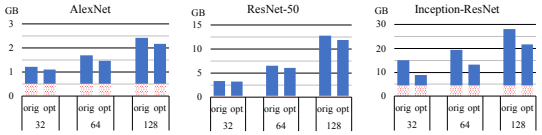
After obtaining the parameters from the sample run, we calculate the peak memory usage u and memory offsets x_i for memory blocks i by solving DSA and then allocate GPU memory of size u; we write p for the address of the memory. In the rest of the running of the model, we return memory address $p + x_i$ for a request of memory block i. We identify memory blocks by maintaining the global variable λ, which is initialized with one before starting each forward propagation. When a memory block is requested, we return address $p+x_\lambda$ and increase λ. This is sound since the propagation should be computed in the same way as in the sample run, where λ is always increased after each allocation. As we explain in Section 3, we implement two methods to DSA and compare the performance.

4. 3 Generalization for non-hot propagation

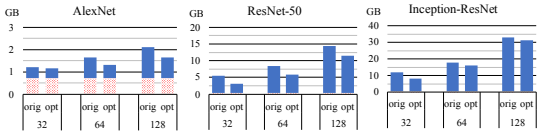
The memory allocation in Section 4. 2 is unsound for models which, for different inputs, (1) perform non-hot propagation (that is, it is computed differently) and (2) request memory of different sizes. This section gives workarounds to avoid them.

A workaround for the first issue is very simple: we do *not* optimize the usage of memory requested in the non-hot part of the propagation. To this end, we provide two operations, `interrupt` and `resume`, which interrupt and resume the monitoring of memory operations, respectively. When entering a non-hot part, we call `interrupt`; and, when leaving that part, we call `resume`. Since our method optimizes only the profiled part of memory usage, the memory requested between calls to `interrupt` and `resume` is out of the scope of the optimization.

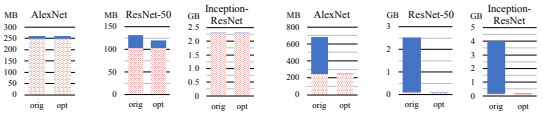
The second issue is resolved by *reoptimization*. In this approach, we continue the monitoring of memory operations after optimizing the memory usage and, when detecting a request for larger memory than expected, we reoptimize the memory allocation by using the new observed parameters—note that we do not need reoptimization for requests of



(a) Training in Chainer

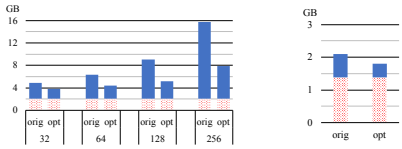


(b) Training in PyTorch



(c) Inference in Chainer. (d) Inference in PyTorch.

図 1: Memory optimization for CNNs in Chainer and PyTorch. Doted red bars show the amounts of memory retained in the entire propagation (thus, they are not optimized by our approach) and solid blue bars show the amounts of memory released until the end of each propagation (thus, they are optimized).



(a) Training. (b) Inference.

図 2: Memory optimization for seq2seq in Chainer.

smaller memory. This workaround may incur an additional performance cost, but it is very low as shown in Section 5. 3.

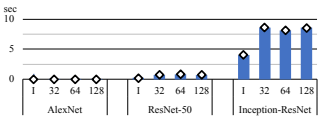
5 Experiments

5. 1 Configurations

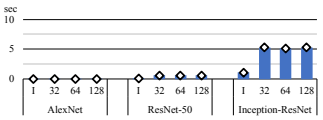
We compare the GPU device memory consumption (Figure 1) in Chainer (version 3 RC 1.0) and PyTorch (0.4.0), which is a baseline and denoted by *orig* in figures for shorthand, and their optimized version by our approach, denoted by *opt*, on

	CPLEX	Best-fit
AlexNet	10169344	10169344
GoogLeNet	12202496	12202496

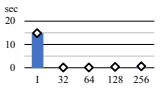
表 1: The required memory sizes calculated by CPLEX and the best-fit heuristic for inference; for configurations not shown here, CPLEX could not finish within 1 hour time limit.



(a) CNNs in Chainer.



(b) CNNs in PyTorch.



(c) Seq2Seq in Chainer.

図 3: The running times of the best-fit heuristic. "I" on the x-axes means that the corresponding numbers are the times for the inference and 32, 64, 128, and 256 denote mini-batch sizes in the training.

three CNNs (AlexNet, ResNet-50, and Inception-ResNet). We furthermore incorporate the both workarounds in Section 4. 3, which make it possible to apply our approach to a variety of DNNs, into Chainer and confirm that the workarounds work well in a RNN (seq2seq); Figure 2 shows comparison of the memory consumption of seq2seq in Chainer and its optimized version.

Training of the CNNs is performed with 32, 64, and 128 mini-batch sizes, and that of seq2seq is with 32, 64, 128, and 256 ones. Inference performs only forward propagation for one input data. We use ImageNet [24] and the English-French cor-

pus from WMT15[†3] as datasets for the CNNs and seq2seq, respectively. We use the first 1000 training mini-batches for the warm-up and next 2000 mini-batches for the evaluation. We turn on Unified Memory of NVIDIA CUDA, which allows us to run models requiring more memory than the physical capacity, in the experiments for memory consumption but turn it off in the measurement of running times since it may incur performance overhead.

We also evaluate the best-fit heuristic implemented in Python in two experiments. We first compare the solutions by the heuristic with the optimal solutions found by CPLEX version 12.8 within one hour. We also compare the computation time of the heuristic for different configurations (Figure 3).

All experiments are run on an IBM POWER8 machine with two 4GHz 10-core POWER8 processors, 512 GB RAM, and NVIDIA Tesla P100 GPUs equipped with 16 GB device memory. Options except mini-batch sizes follow the scripts provided by Chainer and PyTorch scripts.

Finally, we make a few remarks. The first is on the GPU memory management system of our baseline. The original Chainer uses DMA for memory reuse, as described in Section 2, and reduces the memory consumption somewhat compared with naive, network-wise memory allocation, which always allocates a memory block from the physical device memory for each request. For example, we observed that, in the training of AlexNet with 32 mini-batch size, the network-wise memory allocation consumes 1.50 GB device memory whereas the DMA does only 1.21 GB memory. In this section, we show that our approach achieves reduction of more memory than the DMA method. The second remark is on convolution algorithms. There are many algorithms for computing convolution. The most memory-efficient algorithm needs memory only for inputs and outputs, but we can calculate the convolution much faster by allocating additional temporary memory, called *workspace*. Although the optimized version could be accelerated by allocating larger workspace than the original Chainer, the experiments use workspace of the same size (8 MB by default) in both versions for comparing only the effect of the memory optimization.

†3 `http://www.statmt.org/wmt15/`

5.2 CNNs

5.2.1 Training

The total memory consumption during the training of CNNs is shown in Figure 1a for Chainer and Figure 1b for PyTorch. In figures throughout this paper, the amount of memory retained in the entire training (e.g., memory for learnable parameters and gradients) is indicated by doted red bars and the amount of memory released until the end of each propagation is indicated by solid blue bars; our method optimizes usage of only the latter. These figures show that our optimization works well in all models and is the most effective for Inception-ResNet in Chainer (Figure 1a). Specifically, in 64 mini-batch size, the memory consumption in the optimized version fits within the physical memory capacity (16 GB), whereas the required memory in the original Chainer exceeds the capacity considerably. Interestingly, this ability to use a larger mini-batch size enables us to utilize GPUs more fully and improve the running time performance of training in some cases. Actually, we confirmed that the number of images processed per second by the optimized version with 64 mini-batch size is 3.95 times as large as that by the original Chainer with 32 mini-batch size.

5.2.2 Inference

The memory consumption in inference is shown in Figure 1c (Chainer) and Figure 1d (PyTorch).

Since the inference does not need to retain memory for intermediate results, most memory blocks can be reused even in the memory management of Chainer. Nevertheless, we successfully reduce the total memory amounts in ResNet-50 by 10.0%, respectively. The amount of reduced memory is much more significant in PyTorch; 95% of the memory usage in the original is reduced by our optimization. We wonder if this is because PyTorch might retain intermediate results after they become unnecessary, but we need further investigation.

5. 2. 3 Heuristic

CPLEX could obtain the optimal solutions only in two configurations (inference using AlexNet and GoogLeNet), and the objective function values by the heuristic and CPLEX match (10169344 and 12202496, respectively). Our heuristic thus works very well at least in small instances. The execution times of the heuristic are shown in Figures 3a and 3b, which indicate that the heuristic works quickly enough for practical use.

5. 3 Seq2Seq

5. 3. 1 Training and inference

Figure 2a shows the memory consumption immediately after processing 10 mini-batches in the training of seq2seq and demonstrates that our approach significantly reduces the memory consumption. In the original Chainer, since the training of seq2seq requires differently sized memory for different inputs, memory blocks allocated in a training loop may not be used in the succeeding loops, and the whole of such unused blocks finally reaches the device memory capacity. In contrast, we recompute how to allocate memory when necessary, which allows us to keep the memory consumption as low as possible. As for the inference, the amount of consumed memory reduces by 14.6% (Figure 2b).

5. 3. 2 Heuristic

As shown in Figure 3c, the heuristic algorithm takes much longer in the inference, whereas it terminates quickly for the training formulas. This is due to the Chainer script that we use for the evaluation: the script always generates 100 words for inference, whereas it cuts sentences used for the training into up to 50 words. Thus, the inference requests many more memory blocks than the training, and the heuristic takes long in the inference. Fortunately, this should not be problematic in practice, because we can solve DSA with idle CPUs after responding to an inference request. We note that the running time performance of the heuristic can be improved by using faster languages, such as C and C++. CPLEX could not obtain the optimal solutions within the 1-hour time limit.

6 Conclusion

We propose a profile-guided memory optimization for DNNs. We develop a simple heuristic algorithm to DSA to obtain efficient and fast memory allocation, and incorporate the heuristic in Chainer and PyTorch. Differ from the online memory management inherently used in deep learning frameworks which allocates memory blocks on-the-fly during the computation, our method tries to find an optimal allocation by solving by offline optimization. Compared to existing methods, we experimentally confirmed that our method reduces the memory consumption and accelerates propagation in both training and inference using CNNs and seq2seq (RNNs).

There are several directions to extend our work. First, combining offline and memory allocation as well as other techniques in Wang et al. [30] may lead to further optimized allocation. Second, we also observe through experimental results that are omitted due to page limitation, that although our heuristic is based on the the *best-fit* (BF) algorithm for the 2SP of Burk et al. [6], its solutions are mostly better

than those found by the *incremental 2-allocation construction* (IAC). A further research to have a theoretical analysis of the performance of BF algorithm for the Dynamic Memory Allocation (DMA) is an interesting direction.

参 考 文 献

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P. A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X.: TensorFlow: A System for Large-Scale Machine Learning, *Proc. of OSDI*, 2016, pp. 265–283.

[2] Akiba, T., Suzuki, S., and Fukuda, K.: Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes, *CoRR*, Vol. abs/1711.04325(2017).

[3] Arahori, Y., Imamichi, T., and Nagamochi, H.: An exact strip packing algorithm based on canonical forms, *Computers & Operations Research*, Vol. 39, No. 12(2012), pp. 2991–3011.

[4] Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., Warde-Farley, D., and Bengio, Y.: Theano: new features and speed improvements, *CoRR*, Vol. abs/1211.5590(2012).

[5] Buchsbaum, A. L., Karloff, H., Kenyon, C., Reingold, N., and Thorup, M.: OPT Versus LOAD in Dynamic Storage Allocation, *SIAM J. Comput.*, Vol. 33, No. 3(2004), pp. 632–646.

[6] Burke, E. K., Kendall, G., and Whitwell, G.: A New Placement Heuristic for the Orthogonal Stock-Cutting Problem, *Operations Research*, Vol. 52, No. 4(2004), pp. 655–671.

[7] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z.: MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems, *CoRR*, Vol. abs/1512.01274(2015).

[8] Chen, T., Xu, B., Zhang, C., and Guestrin, C.: Training Deep Nets with Sublinear Memory Cost, *CoRR*, Vol. abs/1604.06174(2016).

[9] Gálvez, W., Grandoni, F., Ingala, S., and Khan, A.: Improved Pseudo-Polynomial-Time Approximation for Strip Packing, *CoRR*, Vol. abs/1801.07541(2018).

[10] Garey, M. R. and Johnson, D. S.: *Computers and Intractability : A Guide to the Theory of NP-Completeness*, Series of Books in the Mathematical Sciences, W. H. Freeman, 1979.

[11] Gergov, J.: Algorithms for compile-time memory optimization, *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms (SODA 99)*, SIAM, 1999, pp. 907–908.

[12] Gong, Y., Liu, L., Yang, M., and Bourdev, L. D.: Compressing Deep Convolutional Networks using Vector Quantization, *CoRR*, Vol. abs/1412.6115(2014).

[13] Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K.: Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, *CoRR*, Vol. abs/1706.02677(2017).

[14] Han, S., Mao, H., and Dally, W. J.: Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding, *Proc. of ICLR*, 2016.

[15] Han, S., Pool, J., Tran, J., and Dally, W. J.: Learning both Weights and Connections for Efficient Neural Network, *Proc. of NIPS*, 2015, pp. 1135–1143.

[16] He, K., Zhang, X., Ren, S., and Sun, J.: Deep Residual Learning for Image Recognition, *Proc. of CVPR*, 2016, pp. 770–778.

[17] Hochreiter, S. and Schmidhuber, J.: Long Short-Term Memory, *Neural Computation*, Vol. 9, No. 8(1997), pp. 1735–1780.

[18] Huang, E. and Korf, R. E.: Optimal Rectangle Packing: An Absolute Placement Approach, *Journal of Artificial Intelligence Research*, Vol. 46(2013), pp. 47–87.

[19] Krizhevsky, A., Sutskever, I., and Hinton, G. E.: ImageNet Classification with Deep Convolutional Neural Networks, *Proc. of NIPS*, 2012, pp. 1106–1114.

[20] LeCun, Y., Bengio, Y., and Hinton, G. E.: Deep learning, *Nature*, Vol. 521, No. 7553(2015), pp. 436–444.

[21] Meng, C., Sun, M., Yang, J., Qiu, M., and Gu, Y.: Training Deeper Models by GPU Memory Optimization on TensorFlow, *Proc. of ML Systems Workshop in NIPS*, 2017.

[22] PyTorch: URL: `https://github.com/pytorch/pytorch`. Accessed on 14 May 2018.

[23] Rhu, M., Gimelshein, N., Clemons, J., Zulfiqar, A., and Keckler, S. W.: vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design, *Proc. of MICRO*, 2016, pp. 1–13.

[24] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L.: ImageNet Large Scale Visual Recognition Challenge, *International Journal of Computer Vision*, Vol. 115, No. 3(2015), pp. 211–252.

[25] Shirahata, K., Tomita, Y., and Ike, A.: Memory reduction method for deep neural network training, *Proc. of MLSP*, 2016, pp. 1–6.

[26] Sutskever, I., Vinyals, O., and Le, Q. V.: Sequence to Sequence Learning with Neural Networks, *Proc. of NIPS*, 2014, pp. 3104–3112.

[27] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A.: Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning, *Proc. of AAAI*, 2017, pp. 4278–4284.

[28] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A.: Going deeper with convolutions, *Proc. of CVPR*, 2015, pp. 1–9.

[29] Tokui, S., Oono, K., Hido, S., and Clayton, J.: Chainer: A Next-Generation Open Source Framework for Deep Learning, *Proc. of Workshop on Machine Learning Systems in NIPS*, 2015.

[30] Wang, L., Ye, J., Zhao, Y., Wu, W., Li, A., Song, S. L., Xu, Z., and Kraska, T.: SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks, *CoRR*, Vol. abs/1801.04380(2018).

[31] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D.: Dynamic Storage Allocation: A Survey and Critical Review, *Proc. of IWMM*, 1995, pp. 1–116.