Survey on Efficient Training of Large Neural Networks

Julia Gusak[1,*] , **Daria Cherniuk**[1,*] , **Alena Shilova**[2,*] ,
Alexandr Katrutsa[1,5] , **Daniel Bershatsky**[1] , **Xunyi Zhao**[3] , **Lionel Eyraud-Dubois**[3] ,
Oleh Shliazhko[4] , **Denis Dimitrov**[4] , **Ivan Oseledets**[1,5] , **Olivier Beaumont**[3]
[1]Skolkovo Institute of Science and Technology, Russia
[2]Inria, University of Lille - CRIStAL, France
[3]Inria, University of Bordeaux, France
[4]Sber, Russia
[5]AIRI, Moscow, Russia
{y.gusak, daria.cherniuk, a.katrutsa, d.bershatsky2, i.oseledets}@skoltech.ru,
{alena.shilova, xunyi.zhao, lionel.eyraud-dubois, olivier.beaumont}@inria.fr,
{olehshliazhko, den.dimitrov}@gmail.com

Abstract

Modern Deep Neural Networks (DNNs) require significant memory to store weight, activations, and other intermediate tensors during training. Hence, many models don't fit one GPU device or can be trained using only a small per-GPU batch size. This survey provides a systematic overview of the approaches that enable more efficient DNNs training. We analyze techniques that save memory and make good use of computation and communication resources on architectures with a single or several GPUs. We summarize the main categories of strategies and compare strategies within and across categories. Along with approaches proposed in the literature, we discuss available implementations.

1 Introduction

Modern trends in the development of Deep Learning (DL) and Artificial Intelligence (AI) technologies involve the use of Deep Neural Networks (DNNs) to solve various problems of image, video, audio, natural language processing, content generation in the form of images or text in a given style and subject, etc.

The question we address in this survey is the following: given your model (which you do not want to rewrite) and your computation platform (which you do not want to change), what are the generic approaches that can allow you to perform the training efficiently? For training to be efficient, it must be feasible (the data must fit in memory), it must exploit the computational power of the resources well (the arithmetic intensity of the operations must be sufficient) and, in the parallel case, it must not be stalled by large data exchanges between the nodes. Profiling tools can identify hardware bottlenecks that will determine which strategies described in this survey can be used to solve the problems of arithmetic intensity, memory and large data exchanges.

[*]equal contribution

We consider that the training is performed on nodes, each consisting of a multicore CPU and some GPUs/TPUs. The CPU usually has a large enough memory (typically a few hundreds GB) to store model parameters, optimizer states, and activations, unlike the GPUs, which have a limited memory (typically a few dozens GB) and are connected to the CPU memory through a PCIe bus (typically a few GB/s).

The present survey covers generic techniques to cope with these limitations. If the computation cannot be performed a priori because the model, the optimizer states, and the activations do not fit in GPU/TPU memory, there are techniques to trade memory for computation (re-materialization) or for data movements to CPU(activation and weight offloading), and it is also possible to compress the memory footprint by approximating optimizer states and gradients (compression, pruning, quantization). The use of parallelism (data parallelism, model parallelism, pipelined model parallelism) can also make it possible to distribute the memory requirements over several devices. If the arithmetic intensity of the computations is not sufficient to fully exploit the GPUs and TPUs, it is generally because the size of the mini-batch is too small, and then the above techniques can also enable increasing the size of the mini-batch. Finally, if the communications, typically induced by the use of data parallelism, are too expensive and slow down the computation, then other forms of parallelism can be used (model parallelism, pipelined model parallelism) and the compression of the gradients can allow limiting the volumes of exchanged data.

In this survey, we explain how these different techniques work, we review the literature to evaluate and compare the proposed approaches, and we analyze the frameworks that allow implementation of these techniques (almost) transparently. The different techniques that we consider, and their influence on communications, memory and computing efficiency are depicted in Table 1.

According to our taxonomy, we distinguish the following methods based on their purpose: reducing the memory footprint on a GPU is discussed in Section 2, the use of parallel

Method	# of GPUs	Approx. computations	Communication costs per iteration activation values / weight values / activation grads / weight grads	Batch size per GPU increase?	# of FLOP per iteration
No data parallelism	1	baseline	baseline	baseline	baseline
Rematerialization	≥ 1	✗	= / = = / =	✓	↑
Offloading:					
activations	≥ 1	✗	↑ / = = / =	✓	=
weights	≥ 1	✗	= / ↑ = / ↑ or =	✓	=
tensors in GPU cache	≥ 1	✗	↑ or = / ↑ or = = / ↑ or =	✓	=
Approx. gradients:					
lower-bit activation grad.**	≥ 1	✓	↓ / = = / =	✓	↓ or =
approx. matmul**	≥ 1	✓	↓ / = = / =	✓	↕
lower-bit weight grad**	≥ 1	✓	= / = = / ↓	✓	↓ or =
Data parallelism*	> 1	✗	baseline	✗	=
Partitioning:					
optim. state	> 1	✗	= / ↑ = / =	✓	=
+ gradients	> 1	✗	= / ↑ = / =	✓	=
+ parameters	> 1	✗	= / ↑ = / =	✓	=
Model parallelism*	> 1	✗	↑ / = ↑ / ↓	✓	=
Pipeline parallelism	> 1	✓/✗	↑ / = ↑ / ↕	✓	=

*We assume that updates are performed synchronously. If updates are asynchronous then for both data and model parallelism fewer gradients contribute to epoch update and number of epochs till convergence might be increased.
**Communication channel in this case is a bus between a processing unit and a memory bank.

Table 1: Methods to train large neural networks. ↑ and ↓ correspond to the increase and decrease comparing to the baseline above. FLOP is floating-point operations.

training for models that do not fit on a GPU is considered in Section 3, and the design of optimizers developed to train models stored on multiple devices is addressed in Section 4.

2 Memory Usage Reduction on a Single GPU

During the forward pass, neural networks store the activations necessary to perform backpropagation. In some cases, these activations can consume a substantial amount of memory, making training infeasible. There are two main approaches to reducing that memory footprint: rematerialization (also called checkpointing) and offloading. Tables 2 and 3 depict an overview of these groups of methods.

2.1 Rematerialization of Activations

Rematerialization is a strategy that only stores a fraction of the activations during the forward pass and recomputes the rest during the backward pass. Rematerialization methods can be distinguished by what computational graphs they are dealing with. The first group comes from Automatic Differentiation (AD), they find optimal schedules for homogeneous sequential networks (DNNs whose layers are executed sequentially and have the same computational and memory costs). The second group concentrates on transition model such as heterogeneous sequential networks (DNNs may be any sequential neural networks consisting of arbitrarily complex modules, e.g. CNNs, ResNet, some transformers), which adjust solutions from AD to heterogeneous settings. The final group focuses on general graphs, but this problem becomes NP-complete in the strong sense and thus can be solved optimally only with ILP, otherwise approximately with various heuristics.

For homogeneous sequential networks, the binomial approach was proven to be optimal in [Grimm *et al.*, 1996] and implemented in REVOLVE [Griewank and Walther, 2000]. Compiler-level techniques have been proposed in [Siskind and Pearlmutter, 2018] to make it applicable to fully arbitrary programs. This results in a divide-and-conquer strategy, which, however, assumes that computations can be interrupted at arbitrary points, making it unsuitable for GPU computations. In the case of heterogeneous computation times and homogeneous memory costs the optimal strategy can be found with dynamic programming [Walther and Griewank, 2008]. A direct adaptation of results for homogeneous chains was applied to RNNs in [Gruslys *et al.*, 2016].

[Beaumont *et al.*, 2019] considered the most general case of heterogeneous sequential DNNs. This paper proposed a new modeling of the problem directly inspired by the data dependencies induced by the PyTorch[1] framework. It uses dynamic programming approach to find the optimal strategy for linear or linearized chains.

Some methods can perform rematerialization for general graphs, though the exact approaches are exponentially expensive (see Table 2). For example, [Jain *et al.*, 2020b] proposed an Integer Linear Program (ILP) to find the optimal rematerialization strategy suitable for an arbitrary Directed Acyclic Graph (DAG) structure. [Kirisame *et al.*, 2020] presented a cheap dynamic heuristic called Dynamic Tensor Rematerialization (DTR) that relies on scores that encourage discarding (i) heavy tensors (ii) with a long lifetime and (iii) that can be easily recomputed.

[1]https://pytorch.org

Paper	Approach	Scope	Guarantees	Complexity	Implementation
[Griewank and Walther, 2000] [Grimm *et al.*, 1996]	dynprog closed-form	hom. seq.	optimal	$O(L^2M)$	REVOLVE
[Walther and Griewank, 2008]	dynprog	het. time, hom. memory	optimal	$O(L^3M)$	-
[Siskind and Pearlmutter, 2018]	divide & conquer	compiler	-	-	checkpointVLAD
[Chen *et al.*, 2016]	periodic	het. seq	heuristic	-	PyTorch
[Gruslys *et al.*, 2016]	dynprog	RNN	-	$O(L^2M)$	BPTT
[Beaumont *et al.*, 2019]	dynprog	het. seq	optimal w/ assumptions	$O(L^3M)$	Rotor
[Jain *et al.*, 2020b]	ILP rational LP	any	optimal heuristic	NP-hard $O(EL)$ vars & constraints	CheckMate
[Kusumoto *et al.*, 2019]	dynprog simplified dynprog	any	- heuristic	$O(T2^{2L}))$ $O(TL^2)$	Recompute
[Kumar *et al.*, 2019]	tree-width decomposition	any	bounded	$2^{O(w)}L + O(wL\log L)$	-
[Kirisame *et al.*, 2020]	greedy (priority scores)	any	heuristic	-	DTR

Table 2: Comparison of rematerialization strategies. We use dynprog to refer to dynamic programming solutions, while simplified dynprog corresponds to dynamic programming that solves relaxed problems relying on simplifying assumptions. The complexity is expressed with respect to number of modules (layer or group of layers) L, memory on GPU M, no-checkpoint execution time T, E is the number of dependencies between layers, i.e. number of activations ($E = \Theta(L)$ in linear case, $E = O(L^2)$ in general) and w is a treewidth of the computational graph.

Support in popular open-source frameworks. Machine learning framework PyTorch provides two options for checkpointing: the user explicitly defines which activations to store or uses a periodic strategy based on [Chen *et al.*, 2016]. Checkmate[2], written in TensorFlow[3], accepts user-defined models expressed via the high-level Keras interface. Framework Rotor[4] allows the algorithm from [Beaumont *et al.*, 2019] to be used with any PyTorch DNN implemented with the `nn.Sequential` container.

2.2 Offloading of Activations

Offloading (also called *Memory Swapping*) is a technique that saves GPU memory by offloading activations to CPU memory during forward pass and prefetching them back into GPU memory for the corresponding backward computation.

Due to the limited bandwidth of the PCI bus between the CPU and the GPU, the choice of which activations to transfer and when to do it must be optimized. Authors of vDNN [Rhu *et al.*, 2016] followed a heuristic effective for CNNs by offloading only the inputs of convolutional layers, though it does not generalize well to general DNNs. [Le *et al.*, 2018] considered the activations life-cycle to choose the candidates for offloading and used graph search methods to identify the instants when to insert offload/prefetch operations. AutoSwap [Zhang *et al.*, 2019] decides which activations to offload by assigning each variable a priority score. SwapAdvisor [Huang *et al.*, 2020] used a Genetic Algorithm (GA) to find the best schedule (execution order of the modules) and memory allocation; it relied on Swap Planner to decide which tensors to offload (based on their life cycle) and when to perform offload/prefetch (as soon as possible). Authors in [Beaumont *et al.*, 2020] made a thorough theoretical analysis of the problem. They proposed optimal solutions and extended them in [Beaumont *et al.*, 2021a] to jointly optimize activation offloading and rematerialization. All these techniques are summarized in Table 3.

Support in popular open-source frameworks. Framework vDNN++[5] implemented a technically improved version of vDNN. TFLMS [Le *et al.*, 2018] was initially released as a TensorFlow pull request but later got its own repository[6]. A branch of the Rotor framework[7] provides the implementation of the combined offloading and rematerialization algorithms from [Beaumont *et al.*, 2021a].

2.3 Offloading of Weights

A lot of methods mentioned earlier are also suitable for offloading weights as they rely on universal techniques applicable to any tensors, for example, TFLMS, AutoSwap or SwapAdvisor. L2L (layer-to-layer) [Pudipeddi *et al.*, 2020] keeps a single layer in GPU memory, which results in a significant reduction in the memory cost of the network. Granular CPU offloading [Lin *et al.*, 2021] extends this approach and keeps a part of the network in GPU when there is enough memory. Their experiment shows that offloading only the first half of the network can significantly reduce the training time. ZeRO-Offload [Ren *et al.*, 2021] managed to reduce the high communication overhead by introducing the one-step Delayed Parameter Update (DPU) method. In ZeRO-Offload, only the gradients are offloaded to the CPU to update the weights and

[2]https://github.com/parasj/checkmate
[3]https://www.tensorflow.org
[4]https://gitlab.inria.fr/hiepacs/rotor
[5]https://github.com/shriramsb/vdnn-plus-plus
[6]https://github.com/IBM/tensorflow-large-model-support
[7]https://gitlab.inria.fr/hiepacs/rotor/-/tree/offload-all-rl

Paper	What to offload	Scope	Features
vDNN [Rhu *et al.*, 2016]	all/conv	seq, CNN	choice between memory/performance efficient kernels for conv
TFLMS [Le *et al.*, 2018]	tensors with longer lifetime	any	rewriting graph with swap in/out; treesearch with bounds to schedule transfers
AutoSwap [Zhang *et al.*, 2019]	tensors with highest priority scores	any	uses bayesian optmization to mix priority scores; optimizes memory allocation
SwapAdviser [Huang *et al.*, 2020]	tensors with longer lifetime	any	memory allocation and scheduling operations done with Genetic Algorithm
[Beaumont *et al.*, 2020]	chosen by greedy/dynprog	het. seq.	theoretical analysis, optimality proofs for relaxed models
rotor [Beaumont *et al.*, 2021a]	chosen by dynprog	het. seq.	combination of offloading and rematerialization, optimal w/ assumptions

Table 3: Comparison of offloading strategies.

asynchronous updates are used to overlap communications between CPU and GPU with forward computation on GPU.

Support in popular open-source frameworks. DeepSpeed[8] implements the extremely aggressive memory management strategies proposed in ZeRO-Offload [Ren *et al.*, 2021], which allows a single GPU to train models with more than 10 billion parameters.

3 Parallelism for Models that Don't Fit on a Single GPU

When using Model Parallelism, different layers of a network are allocated onto different resources, so that the storage of DNN weights and activations is shared between the resources. In Model Parallelism (MP), only activations have to be communicated and transfers only take place between successive layers assigned to different processors. Comparison of papers mentioned in this section is presented in Table 4.

The execution within Model Parallelism can be accelerated if several mini-batches are pipelined [Huang *et al.*, 2019], and thus several training iterations are active at the same time. Once forward and backward phases have been computed on all these mini-batches, the weights are then updated. This approach is fairly simple to implement but it leaves computational resources largely idle. The PipeDream approach proposed in [Narayanan *et al.*, 2019] improves this training process, by only enforcing that the forward and backward tasks use the same model weights for a given mini-batch. Such a weakened constraint on the training process allows PipeDream to achieve a much better utilization of the processing resources, but the asynchronous updates affect badly the overall convergence of the training in some cases [Li and Hoefler, 2021].

It has been shown that performing the updates less regularly [Narayanan *et al.*, 2021a] helps limiting weight staleness as well. Alternatively, PipeMare [Yang *et al.*, 2021] proposes to adapt the learning rate and the model weights for backward depending on the pipeline stage. The last method achieves the same convergence rate as GPipe, while having the same resource utilization as PipeDream without storing multiple copies of the weights. Another important issue related to PipeDream is the need to keep many copies of the model parameters, which can potentially cancel the benefit of using Model Parallelism. To address this issue, the methods to limit weight staleness can be used: in [Narayanan *et al.*, 2021a] the updates are done so that it is possible to keep only two versions of the weights (Double-Buffering).

Modeling the storage cost induced by activations in pipeline approaches is a difficult task [Beaumont *et al.*, 2021b]. Some pipelines (DAPPLE [Fan *et al.*, 2021], Chimera [Li and Hoefler, 2021]) use the One-Forward-One-Backward scheduling (1F1B) to reduce memory consumption related to activations. It is a synchronous weight update technique that schedules backward passes of each micro-batch as early as possible to release the memory occupied by activations. Gems [Jain *et al.*, 2020a] and Chimera [Li and Hoefler, 2021] implement bidirectional pipelines, where each GPU is used for two pipeline stages (i and $P - i$, P is the number of stages). The design of Gems is mostly concerned with activations memory: the forward pass of the next micro-batch starts after the first backward stage of the previous micro-batch is computed and activations memory is released. Chimera rather focuses on reducing the computational bubble by starting the forward passes of each pipeline direction simultaneously. A resembling approach was taken in [Narayanan *et al.*, 2021b], where each GPU is assigned more than one pipeline stages (referred to as the Interleaved Pipeline).

Several papers specifically target challenging topologies. To solve the problem in the case of high communication costs and heterogeneous networking capabilities, the authors of Pipe-torch [Zhan and Zhang, 2019] propose an updated dynamic programming strategy which assumes no overlap between computations and communications. HetPipe [Park *et al.*, 2020] addresses the additional problem of heterogeneous GPUs by grouping them into virtual workers and running pipeline parallelism within each virtual worker, while relying on data parallelism between workers. Varuna [Athlur *et al.*, 2021] focuses on "spot" (low-priority) VMs and builds a schedule that is robust to network jitter, by performing a pipelining technique that resembles 1F1B: activation recomputations and respective backward passes are scheduled opportunistically.

[8]https://github.com/microsoft/DeepSpeed

Paper	Parallelism	Pipeline Feature	Partition Optimization
GPipe [Huang *et al.*, 2019]	DP, PP	First introduced pipelining	-
Megatron-LM [Narayanan *et al.*, 2021b]	TP, DP, PP	1F1B, Interleaved Pipeline	Heuristic
PipeDream [Narayanan *et al.*, 2019]	DP, PP	Async Update	DynProg for DP, PP
PipeDream-2BW [Narayanan *et al.*, 2021a]	DP, PP	Async Double-Buffered Update	DynProg for DP, PP, Checkpointing
DAPPLE [Fan *et al.*, 2021]	DP, PP	1F1B	DynProg for DP, PP
PipeMare [Yang *et al.*, 2021]	PP	Async Update, LR Rescheduling, Weight Discrepancy Correction	Splitting weights evenly between model partitions
Piper [Tarnawski *et al.*, 2021]	TP, DP, PP	Async Update	DynProg for TP, DP, PP, Checkpointing
HetPipe [Park *et al.*, 2020]	DP, PP	Parameter Server	LinProg for PP
Pipe-torch [Zhan and Zhang, 2019]	DP, PP	Async Update	DynProg for DP, PP, GPU allocation
Varuna [Athlur *et al.*, 2021]	DP, PP	Opportunistic Backward Scheduling	Heuristic PP partition, Bruteforce for DP, PP depth
Gems [Jain *et al.*, 2020a]	DP, PP	Bidirectional Pipeline	-
Chimera [Li and Hoefler, 2021]	DP,PP	1F1B, Bidirectional Pipeline	Greedy mini-batch size, Bruteforce for DP, PP depth

Table 4: Comparison of model parallelism strategies. TP, DP, and PP stand for tensor-, data- and pipeline- parallelism respectively.

4 Optimizers for Cross-Device Training

4.1 Zero Redundancy Optimizer

The authors of [Rajbhandari *et al.*, 2020] propose ZeRO (Zero Redundancy Optimizer) as an implementation of data-parallelism with reduced memory footprint. The algorithm has three versions depending on what tensors are partitioned across devices: Stage 1 (optimizer states), Stage 2 (optimizer states and gradients), and Stage 3 (optimizer states, gradients and model parameters). ZeRO works in a mixed precision regime to reduce the amount of data transferred between devices. Still, Stages 2 and 3 introduce a communication overhead. In [Ren *et al.*, 2021] authors propose to unite ZeRO and CPU-side computation of parameter updates within ZeRO-Offload: gradients are transferred to CPU where copies of parameters are stored; the update is applied to the copies and the updated weights are transferred back to GPU.

Support in popular open-source frameworks. An open source implementation of all ZeRO-* algorithms is available in the DeepSpeed[8] framework.

4.2 Low-Precision Optimizers

To further reduce memory footprint, low-precision optimizers can be used. These methods use low precision formats to represent the optimizers states and auxiliary vectors of states. Also, error compensation techniques are used to preserve the approximation accuracy of the tracking statistics. [Dettmers *et al.*, 2021] proposes a method to store statistics of Adam optimizer in 8-bit while the overall performance remains the same as when the 32-bit format is used. The key technique to achieve such a result is blockwise dynamic quantization that efficiently handles both large and small magnitude elements. More aggressive precision reduction is presented in [Sun *et al.*, 2020], where special routines to deal with 4-bit representation are developed. In particular, the adaptive Gradient Scaling (GradScale) method aims to mitigate the issue with insufficient range and resolution. Moreover, [Li *et al.*, 2021] combines Adam and 1-bit SGD momentum optimizers in the large batch setting and proposes communication-efficient algorithm for layerwise adaptive learning rates.

4.3 Acceleration of Convergence

Another way to accelerate training of large deep learning models is to reduce communication time between nodes and/or number of required epochs to converge at the appropriate local minimum.

Communication costs reduction. Different approaches have been proposed to compress gradients before transferring them between computational nodes. In particular, three classes of such methods are typically discussed: sparsification, quantization and low-rank methods. Sparsification methods only transfer some subset of complete gradient elements and update the corresponding elements in the parameter vector. This approximation significantly reduces the communication costs [Alistarh *et al.*, 2019] while the trained model performance is preserved. Another approach is based on quantization of transferred gradients, which consists of transferring only a limited number of bits, reconstructing the entire gradient vector from them, and updating all elements of the parameter vector. This approach demonstrates promising results for some neural networks architectures and experimental settings [Alistarh *et al.*, 2017]. In particular, recent results in sending only the signs of stochastic gradient elements [Stich *et al.*, 2018] have been extended to more complicated Adam optimizer [Tang *et al.*, 2021], where the non-linear effect of the optimizer states requires additional investigation of error compensation strategies. Another approach for communication costs reduction is the low-rank approach, in which a low-rank approximation of the gradient is constructed, transferred and used to recover the gradient in full format before updating the parameter vector. The low-rank approximation is constructed with the block power method [Vogels *et al.*, 2019] or with the alternating minimization strategy [Cho *et al.*, 2019]. The main difficulty here is to balance the gains from the reduction of communication costs with the additional costs induced by the construction of low-rank approximations. A comprehensive analysis and numerical comparison of many methods for communication overhead reduction from the aforementioned classes are presented in review [Xu *et al.*, 2021].

Large batch training. Another approach to speed up the convergence of the optimizer is to use a large number of samples per batch. This training setting leads to a reduction of the number of iterations in every epoch and a better utilization of GPU. In [Goyal *et al.*, 2017] authors propose to use a linear scaling rule to update the learning rate along with the batch size. This setting stabilizes the optimization process and converges to the same final performance of the model. Note also that large batch training significantly reduces the variance in the stochastic gradient estimate. However, study [Keskar *et al.*, 2016] observes that this feature of the training reduces the generalization ability of the trained model if other hyperparameters remain the same. Therefore, other alternatives to the linear scaling rule have been considered in further works.

In particular, Layer-wise Adaptive Rate Scaling is combined with SGD [You *et al.*, 2017] (LARS scheme) and Adam optimizers [You *et al.*, 2019] (LAMB scheme) to adjust the learning rate in every layer separately. These strategies are based on the observation of a significant difference in the magnitude of parameters and gradients in the different layers, which is natural for deep neural networks. Note that the memory saving techniques considered in Sections 2 typically allow the batch size to be increased with little overhead, even when exceeding the GPU memory capabilities.

5 Conclusion and Further Research

In the survey we have discussed methods that help to train larger models on a single GPU and do more efficient training on multiple GPUs. Such methods optimize both the training of known high-quality large models (e.g., GPT, CLIP, DALLE) from scratch and the fine-tuning of pre-trained models for specific and personalized tasks.

Several main factors influence the training of large DNNs: (i) memory required to store model parameters, activations, optimizer states(ii) time spent on data exchange (communication) between computing nodes and its impact on the computing time on a separate computing node, (iii) parallel efficiency (percentage of time when GPUs are not idle), (iv) the number of floating-point operations required to calculate the forward and backward passes for the given model architecture, dataset, and target functionality, (v) the number of iterations required to achieve the specified accuracy. The strategies discussed in Sections 2, 3, and 4 of this survey are applied to reduce the influence of these factors.

Current research in rematerilization (Table 2) focus on finding the optimal checkpointing strategy if we have a homogeneous or heterogeneous sequential model. However, modern neural networks have a more complex structure - for example, due to a large number of residual connections. Currently, optimal rematerialization strategies for each architecture and input data size are heuristically searched. Further research can find theoretically optimal solutions for more general types of architectures. Rematerialization methods demonstrated their benefits in reducing memory on one GPU. Despite that, it is important to combine it with other methods to achieve significant decrease in memory consumption. For example, in [Beaumont *et al.*, 2021a] considering the optimal combinations of offloading and rematerialization (Section 2) further pushed the performance of both methods. Considering other combinations, e.g. checkpointing and pipelining (Section 3), can be a promising further development of both methods.

In optimization methods (Section 4) there are three main research directions to adapt them for large model training: low-precision storage of states and gradients, batch size increasing with learning rate scheduling, compression of transferred gradients. They demonstrate promising results to train particular models but, at the same time, they are quite far from the complete technology. For example, the low precision approach requires extensive hardware support of the operations with numbers in a low-precision format, and the compression scheme for gradient transmission can be efficient only after the careful setting of the broadcasting environment. Thus, these approaches require additional research to make them robust and widespread.

Among the promising directions, we should mention computations with reduced precision, approximate methods [Novikov *et al.*, 2022], randomized computations [Bershatsky *et al.*, 2022], and structured NN layers [Hrinchuk *et al.*, 2020], including those based on tensor factorizations. We should highlight the importance for these approximate methods to be additive in the sense that they can be combined and still provide sufficient enough performance with reasonable quality degradation.

Finally, it is worth emphasizing the importance of developing new promising computing architectures that can speed up elementary machine learning operations (for example, matrix-vector multiplication) and low-level optimization techniques.

Acknowledgments

This work was supported by the Inria Molière International Associated Team between Skoltech and Inria. The work was supported by the Analytical center under the RF Government (subsidy agreement 000000D730321P5Q0002, Grant No. 70-2021-00145 02.11.2021).

References

[Alistarh *et al.*, 2017] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding. In *NIPS*, 2017.

[Alistarh *et al.*, 2019] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Sarit Kririrat, Nikola Konstantinov, and Cedric Renggli. The Convergence of Sparsified Gradient Methods. In *NeurIPS*, pages 5973–5983, 2019.

[Athlur *et al.*, 2021] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: Scalable, Low-cost Training of Massive Deep Learning Models. *arXiv:2111.04007*, 2021.

[Beaumont *et al.*, 2019] Olivier Beaumont, Lionel Eyraud-Dubois, Julien Herrmann, Alexis Joly, and Alena Shilova. Optimal Checkpointing for Heterogeneous Chains: How to Train Deep Neural Networks with Limited Memory. Research Report RR-9302, INRIA, 2019.

[Beaumont *et al.*, 2020] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training. In *Euro-Par*, 2020.

[Beaumont *et al.*, 2021a] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Efficient Combination of Rematerialization and Offloading for Training DNNs. In *NeurIPS*, 2021.

[Beaumont *et al.*, 2021b] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Pipelined Model Parallelism: Complexity Results and Memory Considerations. In *Euro-Par*, pages 183–198, 2021.

[Bershatsky *et al.*, 2022] Daniel Bershatsky, Aleksandr Mikhalev, Alexandr Katrutsa, Julia Gusak, Daniil Merkulov, and Ivan Oseledets. Memory-Efficient Backpropagation through Large Linear Layers. *arXiv:2201.13195*, 2022.

[Chen *et al.*, 2016] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv:1604.06174*, 2016.

[Cho *et al.*, 2019] Minsik Cho, Vinod Muthusamy, Brad Nemanich, and Ruchir Puri. GradZip: Gradient Compression using Alternating Matrix Factorization for Large-scale Deep Learning. In *Workshop on Systems for ML at NeurIPS'19*, 2019.

[Dettmers *et al.*, 2021] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit Optimizers via Block-wise Quantization. *CoRR*, 2021.

[Fan *et al.*, 2021] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *PPOPP*, 2021.

[Goyal *et al.*, 2017] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD. *arXiv:1706.02677*, 2017.

[Griewank and Walther, 2000] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM TOMS*, 2000.

[Grimm *et al.*, 1996] José Grimm, Loïc Pottier, and Nicole Rostaing-Schmidt. Optimal Time and Minimum Space-Time Product for Reversing a Certain Class of Programs. Technical report, INRIA, 1996.

[Gruslys *et al.*, 2016] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. Memory-Efficient Backpropagation Through Time. In *NIPS*, 2016.

[Hrinchuk *et al.*, 2020] Oleksii Hrinchuk, Valentin Khrulkov, Leyla Mirvakhabova, Elena Orlova, and Ivan Oseledets. Tensorized embedding layers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4847–4860, Online, November 2020. Association for Computational Linguistics.

[Huang *et al.*, 2019] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, Hyouk Joong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*, 2019.

[Huang *et al.*, 2020] Chien-Chin Huang, Gu Jin, and Jinyang Li. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *ASPLOS*, 2020.

[Jain *et al.*, 2020a] Arpan Jain, Ammar Ahmad Awan, Asmaa M. Aljuhani, Jahanzeb Maqbool Hashmi, Quentin G. Anthony, Hari Subramoni, Dhableswar K. Panda, Raghu Machiraju, and Anil Parwani. GEMS: GPU-Enabled Memory-Aware Model-Parallelism System for Distributed DNN Training. In *SC*, 2020.

[Jain *et al.*, 2020b] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *MLSys*, 2020.

[Keskar *et al.*, 2016] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *arXiv:1609.04836*, 2016.

[Kirisame *et al.*, 2020] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic Tensor Rematerialization. *arXiv:2006.09616*, 2020.

[Kumar *et al.*, 2019] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. Efficient Rematerialization for Deep Networks. In *NeurIPS*, 2019.

[Kusumoto *et al.*, 2019] Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation. In *NeurIPS*, 2019.

[Le *et al.*, 2018] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. TFLMS: Large Model Support in TensorFlow by Graph Rewriting. *arXiv:1807.02037*, 2018.

[Li and Hoefler, 2021] Shigang Li and Torsten Hoefler. Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines. In *SC*, pages 1–14, 2021.

[Li *et al.*, 2021] Conglong Li, Ammar Ahmad Awan, Hanlin Tang, Samyam Rajbhandari, and Yuxiong He. 1-bit LAMB: Communication Efficient Large-Scale Large-Batch Training with LAMB's Convergence Speed. *arXiv preprint arXiv:2104.06069*, 2021.

[Lin *et al.*, 2021] Junyang Lin, An Yang, Jinze Bai, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Yong Li, Wei Lin, et al. M6-10T: A Sharing-Delinking Paradigm for Efficient Multi-Trillion Parameter Pretraining. *arXiv:2110.03888*, 2021.

[Narayanan *et al.*, 2019] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized Pipeline Parallelism for DNN Training. In *SOSP*, 2019.

[Narayanan *et al.*, 2021a] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-Efficient Pipeline-Parallel DNN Training. In *ICML*, 2021.

[Narayanan *et al.*, 2021b] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *SC*, pages 1–15, 2021.

[Novikov *et al.*, 2022] Georgii Novikov, Daniel Bershatsky, Julia Gusak, Alex Shonenkov, Denis Dimitrov, and Ivan Oseledets. Few-Bit Backward: Quantized Gradients of Activation Functions for Memory Footprint Reduction. *arXiv:2202.00441*, 2022.

[Park *et al.*, 2020] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young ri Choi. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *USENIX ATC*, 2020.

[Pudipeddi *et al.*, 2020] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. Training Large Neural Networks with Constant Memory using a New Execution Algorithm. *arXiv:2002.05645*, 2020.

[Rajbhandari *et al.*, 2020] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations toward Training Trillion Parameter Models. In *SC*, pages 1–16, 2020.

[Ren *et al.*, 2021] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training. *arXiv:2101.06840*, 1 2021.

[Rhu *et al.*, 2016] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *MICRO*, 2016.

[Siskind and Pearlmutter, 2018] Jeffrey Mark Siskind and Barak A. Pearlmutter. Divide-and-Conquer Checkpointing for Arbitrary Programs with no User Annotation. *Optimization Methods and Software*, 2018.

[Stich *et al.*, 2018] Sebastian U. Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified SGD with Memory. In *NIPS*, pages 4452–4463, 2018.

[Sun *et al.*, 2020] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi (Viji) Srinivasan, and Kailash Gopalakrishnan. Ultra-Low Precision 4-bit Training of Deep Neural Networks. In *NeurIPS*, volume 33, pages 1796–1807, 2020.

[Tang *et al.*, 2021] Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, and Yuxiong He. 1-bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed. In *ICML*, 2021.

[Tarnawski *et al.*, 2021] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional Planner for DNN Parallelization. In *NeurIPS*, volume 34, 2021.

[Vogels *et al.*, 2019] Thijs Vogels, Sai Praneeth Karinireddy, and Martin Jaggi. PowerSGD: Practical low-rank gradient compression for distributed optimization. In *NeurIPS*, 2019.

[Walther and Griewank, 2008] Andrea Walther and Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.

[Xu *et al.*, 2021] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. GRACE: A compressed communication framework for distributed machine learning. In *ICDCS*, pages 561–572, 2021.

[Yang *et al.*, 2021] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Re, Christopher Aberger, and Christopher De Sa. PipeMare: Asynchronous Pipeline Parallel DNN Training. In *MLSys*, 2021.

[You *et al.*, 2017] Yang You, Igor Gitman, and Boris Ginsburg. Large Batch Training of Convolutional Networks. *arXiv:1708.03888*, 2017.

[You *et al.*, 2019] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large Batch Optimization for Deep Learning. *arXiv:1904.00962*, 2019.

[Zhan and Zhang, 2019] Jun Zhan and Jinghui Zhang. Pipe-Torch: Pipeline-Based Distributed Deep Learning in a GPU Cluster with Heterogeneous Networking. In *CBD*, 2019.

[Zhang *et al.*, 2019] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient Memory Management for GPU-based Deep Learning Systems. *arXiv:1903.06631*, 2019.