Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory

Olivier Beaumont, Lionel Eyraud-Dubois, Julien Herrmann, Alexis Joly, Alena Shilova

Optimal Re-Materialization Strategies for Heterogeneous Chains: How to Train Deep Neural Networks with Limited Memory

OLIVIER BEAUMONT, Inria Bordeaux Sud-Ouest, France and University of Bordeaux, France
LIONEL EYRAUD-DUBOIS, Inria Bordeaux Sud-Ouest, France and University of Bordeaux, France
JULIEN HERRMANN, CNRS, France and IRIT, France
ALEXIS JOLY, Inria Sophia-Antipolis Méditerranée, France and University of Montpellier, France
ALENA SHILOVA, Inria Bordeaux Sud-Ouest, France and University of Bordeaux, France

Training in Feed Forward Deep Neural Networks is a memory-intensive operation which is usually performed on GPUs with limited memory capacities. This may force data scientists to limit the depth of the models or the resolution of the input data if data does not fit in the GPU memory. The re-materialization technique, whose idea comes from the checkpointing strategies developed in the Automatic Differentiation literature, allows data scientists to limit the memory requirements related to the storage of intermediate data (activations), at the cost of an increase in the computational cost.

This paper introduces a new strategy of re-materialization of activations that significantly reduces memory usage. It consists in selecting which activations are saved and which activations are deleted during the forward phase, and then recomputing the deleted activations when they are needed during the backward phase.

We propose an original computation model that combines two types of activation savings: either only storing the layer inputs, or recording the complete history of operations that produced the outputs. This paper focuses on the fully heterogeneous case, where the computation time and the memory requirement of each layer is different. We prove that finding the optimal solution is NP-hard and that classical techniques from Automatic Differentiation literature do not apply. Moreover, the classical assumption of memory persistence of materialized activations, used to simplify the search of optimal solutions, does not hold anymore. Thus, we propose a weak memory persistence property and provide a Dynamic Program to compute the optimal sequence of computations.

This algorithm is made available through the Rotor software, a PyTorch plug-in dealing with any network consisting of a sequence of layers, each of them having an arbitrarily complex structure. Through extensive experiments, we show that our implementation consistently outperforms existing re-materialization approaches for a large class of networks, image sizes and batch sizes.

Additional Key Words and Phrases: Checkpointing, Re-materialization, Dynamic Programming, Convolutional Neural Networks, Memory

Authors' addresses: Olivier Beaumont, Olivier.Beaumont@inria.fr, Inria Bordeaux Sud-Ouest, Bordeaux, France and University of Bordeaux, Bordeaux, LaBRI, France; Lionel Eyraud-Dubois, Lionel.Eyraud-Dubois@inria.fr, Inria Bordeaux Sud-Ouest, Bordeaux, France and University of Bordeaux, Bordeaux, LaBRI, France; Julien Herrmann, Julien.Herrmann@irit.fr, CNRS, France and IRIT, Toulouse, France; Alexis Joly, Alexis.Joly@inria.fr, Inria Sophia-Antipolis Méditerranée, Montpellier, France and University of Montpellier, Montpellier, France; Alena Shilova, Alena.Shilova@inria.fr, Inria Bordeaux Sud-Ouest, Bordeaux, France and University of Bordeaux, Bordeaux, LaBRI, France.

0098-3500/2024/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Olivier Beaumont, Lionel Eyraud-Dubois, Julien Herrmann, Alexis Joly, and Alena Shilova. 2024. Optimal Re-Materialization Strategies for Heterogeneous Chains: How to Train Deep Neural Networks with Limited Memory. *ACM Trans. Math. Softw.* 1, 1 (January 2024), 38 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep Neural Networks (DNNs) are known to require a huge quantity of memory during the training phase. Consider, for instance, the ResNet101 network with relatively small images of size 224×224 with 3 channels and a batch size of 32. Training this network requires around 5GB of available memory, while the parameters of the model only require 170MB. When we aim to detect small objects within images [Carranza-Rojas et al. 2017], the image resolution must be increased, and the memory required for storing activations increases quadratically with the image resolution. The situation is even worse when moving to 3D object recognition [Feng et al. 2018; Ghadai et al. 2019; Su et al. 2015], or to video based DNNs such as 3D-Resnet [Hara et al. 2018] or CDC [Shou et al. 2017]. In this context, the input consists of a large number of frames (64 for instance), which induces a huge memory usage even for small batch sizes. Memory pressure becomes even more critical in a growing number of IoT applications where learning has to be performed onto a low memory device [Verhelst and Moons 2017].

The reason for such large memory requirements is that the training algorithms of most DNNs require us to store both the model weights and the forward activations in order to perform back-propagation. From a user point of view, the back-propagation of gradient required to perform the training is hidden by the usage of autograd tools, such as `tf.GradientTape` in TensorFlow or `torch.autograd.backward` in PyTorch. The memory limitation of current hardware often prevents data scientists from scaling, i.e. considering larger models, larger image sizes or larger batch sizes [Pleiss et al. 2017; Rota Bulò et al. 2018]. In many cases, reducing the batch size to fit in memory is not an option: it may reduce the GPU efficiency, and degrade batch statistics [Kusumoto et al. 2019] when using batch-normalization [Ioffe and Szegedy 2015]. Furthermore, for very large networks like GPT2, even using a batch size of 1 may not fit in the GPU memory. Training the "XL" version of GPT2 with the recommended sequence length of 1024 requires 12GB memory for the model and 16GB for the activations, which does not fit in a GPU with 16GB memory such as an NVIDIA V100.

In the case of very large Transformer-based models, model weights, optimizer states, and associated activations must be distributed over several GPUs. For example, LLaMA models [Touvron et al. 2023] have from 7B to 65B parameters. The memory requirements of the weights alone are between 28GB to 260GB, greatly exceeding the memory of the largest GPUs. The techniques proposed in the present paper can be used directly once the neural network is partitioned between the GPUs, although not optimally: it might be more efficient to solve both the partitioning and re-materialization problems together rather than one after the other. We focus here on the design of optimal strategies for the single GPU case, and leave the optimization of the multi-GPU case for future research.

In this paper, we consider the so-called *gradient re-materialization* technique, where only some activations are materialized (stored in memory) while others are deleted and then later recomputed from the materialized ones.

For a given batch size and a given neural network model, this technique provides memory savings at the price of recomputation of some operations in the DNN, to re-materialize the forward activations that were discarded in the forward phase. Re-materialization strategies are needed to determine in advance which forward activations should be kept in memory and which should be deleted and then recomputed from materialized activations during the execution of the backward phase.

Re-materialization strategies were first considered under the name of Gradient Checkpointing in the Automatic Differentiation (AD) [Griewank and Walther 2008] literature[1]. In the context of classical AD, the computation can be seen as a (long) *homogeneous* chain, in which all stages have identical memory and computation requirements, and the forward activation corresponding to the $i-$th stage of the chain has to be kept in memory until the associated $i-$th backward stage. Intermediate values that are not checkpointed need to be re-materialized during the computation. The corresponding recomputations induce an overhead in execution time, and the objective is to minimize this overhead given a memory budget.

Several studies have been performed to determine optimal re-materialization strategies for AD in different contexts: in the case of homogeneous chains, closed form formulas have been proposed [Walther and Griewank 2004], and in the case of homogeneous memory requirements but heterogeneous computation times, Dynamic Programming was used to provide optimal solutions [Griewank and Walther 2008]. The proof of optimality of these solutions rely on the *memory persistency* property (discussed in detail in Section 4), and on the fact that there always exists a memory persistent solution with optimal execution time.

Some of these results from the Automatic Differentiation framework have been adapted to the DNN context. Results on homogeneous chains have been translated for specific DNNs such as Recurrent Neural Networks (RNNs) [Gruslys et al. 2016]. Independently, a simple re-materialization approach [Pytorch contributors 2018] has been proposed and is available in PyTorch, based on a (non-optimal) strategy that involves a sub-linear number of re-materialized activations [Chen et al. 2016]. In the present paper, we propose several improvements and generalizations of these approaches.

This paper is organized as follows. We first present the related work on re-materialization in Section 2, in both the Automatic Differentiation and the Deep Learning contexts. In Section 3, we provide a careful modeling of forward/backward operations with different saving modes that are available in DNN frameworks. We show that these frameworks offer more general operations and thus more optimization opportunities than those used in the AD literature. We assume that the DNN is given as a linear sequence of modules, where internal modules can be arbitrarily complex. In practice, this assumption does not hinder the class of models that can be considered, and we propose implementations of classical vision and classification networks (ResNet, Inception, VGG, DenseNet) under this model. Encoder-based and decoder-based networks used in Natural Langage Processing also satisfy this assumption. In Section 3.4, we prove that the optimization problem of finding the best re-materialization sequence is NP-complete in the case of heterogeneous computation times and memory sizes. This result illustrates that the heterogeneity of neural networks introduces an additional fundamental difficulty with respect to the homogeneous context of Automatic Differentiation, in which many closed form formulas for finding the optimal location of materialized values have been proposed in the literature.

In Section 4, we analyze the complexity and structural properties of the underlying optimization problem with heterogeneous activation sizes (in addition to the heterogeneous computation times that have already been considered in the literature). In particular, we prove that the *memory persistency* result on which all previous approaches are based no longer holds: there exist instances where no memory persistent solution is optimal. We propose and prove a weaker property, which allows us to derive an optimal algorithm. We also present a relaxed dynamic program to obtain the optimal memory persistent solution. This relaxed solution may not be optimal in the general

[1]In Machine Learning literature, the term "*checkpointing*" has a different meaning: it refers to the action of saving the current state of the parameters of the model, so that learning can be resumed later. For that reason, we use the less ambiguous term "*re-materialization*".

case, but is very efficient in practice: it provides the same solution as the optimal algorithm on all observed DNNs, in a much shorter time.

In addition, we describe a complete and easy-to-use implementation of this algorithm in the PyTorch framework, called Rotor ([HiePACS team 2019]), in Section 5. This tool automatically measures the memory requirement and computation time of each layer of the DNN, and then computes the forward and the backward phases while enforcing a memory limit, at the cost of a minimal amount of recomputations. The training strategy of Rotor is completely orthogonal to the optimization of the hyper-parameters of the DNN: it will provide exactly the same accuracy after the same number of epochs as a classical PyTorch implementation, but without exceeding the memory limit of a computing device. We show through an extensive experimental evaluation that compared to alternative re-materialization methods, our approach indeed enables us to significantly increase the training throughput, defined as the average number of processed images per second. This evaluation is performed on the training phase of a large variety of vision networks, with different fixed memory limits.

Therefore, we provide on the one hand original theoretical results that generalize the results of AD literature to a much larger class of models and operations, and on the other hand a fully automatic tool [HiePACS team 2019] that implements our algorithms to run a mini-batch training strategy while enforcing a memory constraint.

2 RELATED WORK

Memory usage is an important constraint in Deep Learning today and comprises various aspects. In this paper, we focus on memory requirements during the training process. We first describe related works on memory-efficient neural network training, then present connections of our work with the Automatic Differentiation literature. Finally, we discuss previous works on re-materialization for Deep Neural Networks.

2.1 Other memory-efficient approaches for training

Some approaches design specific memory-efficient network architectures, for which the challenge is to achieve the same performance as standard state-of-the-art networks. Reversible neural Networks [Chang et al. 2018; Gomez et al. 2017] (RevNets), for instance, are designed such that the back-propagation algorithm can be run without storing the forward activations. Quantization [Hubara et al. 2017; Rastegari et al. 2016] and pruning [Han et al. 2015] reduce the memory consumption at inference time by changing some of the network weights to zero or by quantizing them. Finally, other *ad-hoc* architectures such as MobileNet [Howard et al. 2017] or ShuffleNet [Zhang et al. 2018] propose to sparsify the network architecture to reduce the model size.

In this paper, however, we consider methods that reduce the memory footprint of a given fixed model or architecture, while obtaining the exact same output of the training process. Re-materialization strategies belong to this class of research. Layer optimization is another possibility: re-implementations of some commonly used layers like batch normalization have been proposed [Rota Bulò et al. 2018]. Memory usage can indeed be reduced by rewriting the gradient calculation for this layer so that it does not depend on certain activation values (hence, it is no longer necessary to store them).

In the case of GPUs with limited memory connected to a CPU with more memory, another solution [Beaumont et al. 2019a; Rhu et al. 2016; S B et al. 2016] is to offload some of the activations from the GPU memory to the CPU memory, and then fetch them back when needed during the backward phase. This approach is complementary to re-materialization: in order to save memory, some activations can be deleted and then later re-computed, while others are offloaded and fetched

back. The combination of these two approaches has been considered under simplifying assumptions in [Beaumont et al. 2021].

In addition, training is also an extremely expensive task in terms of computation time, especially on modern networks with a very large number of parameters. Hence, many parallel approaches have been proposed, and we discuss here their relationship with re-materialization.

The most classical approach is data parallelism [Das et al. 2016], which consists in (i) distributing a mini-batch of input data over different computational resources, (ii) computing local gradients on each resource and (iii) using an `AllReduce` operation with all local gradients in order to update the weights. This approach is fully compatible with the re-materialization approach considered in the present paper. Re-materialization can indeed be performed on each computational resource independently in order to limit the memory requirements, without modifying the overall parallel scheme.

When using Model Parallelism [Dean et al. 2012; Huang et al. 2019; Narayanan et al. 2019], layers of a network are distributed over different resources, so that the storage of weights and activations is shared between the resources. In model parallelism, only activations are communicated, and transfers only occur between layers assigned to different processors. This results in a smaller amount of data movement compared to data parallelism. Nevertheless, the scalability of the method is poor because of the chain connections in the DNN computational graph, which induce a sequential execution of all tasks. The execution within Model Parallelism can be accelerated by pipelining several mini-batches [Huang et al. 2019], so that several training iterations are active simultaneously, which helps to keep computational resources busy most of the time.

On the one hand, the pipelined model parallelism approach ensures an increased resource utilization. On the other hand, it also requires more memory since several activations are kept simultaneously in memory. In GPIPE [Huang et al. 2019] and DeepSpeed [Rasley et al. 2020], the problem is addressed by using a systematic re-materialization of all the activations between the forward and backward phases. As in the case of data parallelism, the optimization of re-materialization strategies, as proposed in the present paper, directly benefits model parallel approaches by providing a better control on the computational cost of re-materialization for a given memory capacity.

2.2 Connection with Automatic Differentiation

When the network represents a single chain of layers, the computation of the gradient descent in the training phase is similar to Automatic Differentiation (AD). The computation of adjoints has always been a trade-off between re-materialization and memory requirements and the use of checkpointing strategies in the context of AD has been widely studied. In the AD literature, checkpointed values are stored (materialized) in the device memory. Non-checkpointed values are deleted from the memory, and must be re-computed (re-materialized) later. This notion of a checkpoint is very different from the one used in the fault tolerance framework, and does not involve a transfer to another storage device.

Many studies have been performed to determine optimal re-materialization strategies for AD in different contexts, depending on the presence of a single or multi level memory [Aupy et al. 2016]. Closed form formulas providing the exact position of materialized activations have even been proposed for homogeneous chains [Griewank and Walther 2000], where all layers are identical. When computation times are different across the layers of the network but activation sizes are identical, an optimal re-materialization strategy can be obtained with Dynamic Programming [Griewank and Walther 2008]. Due to their application domain, automatic differentiation techniques are developed for chain networks in which the memory requirements (and often also the computation costs) of the tasks are homogeneous, since the layers correspond to iterations of the same algorithm.

Some researchers have attempted to adapt re-materialization strategies to Arbitrary Computational Graphs (ACGs), thus extending the class of target networks beyond what is proposed in this paper. The ideas for homogeneous chains have been adapted into a generic divide-and-conquer approach based on compiler techniques to perform automatic differentiation for arbitrary programs [Siskind and Pearlmutter 2018]. However, it has been shown [Naumann 2008, 2009] that even in the homogeneous case, solving optimally the problem of re-materialization on general graphs is NP-complete in the strong sense. Specific solutions have been proposed when the program is represented as a call tree, and checkpointing can only occur at the call sites of functions [Lotz et al. 2016a]. In the present paper, we focus on a limited context (heterogeneous chain graphs) for which the complexity is open. We show in Section 3.4 that the problem is NP-complete in the weak sense, which opens the door for efficient algorithms based on dynamic programming, as proposed in Section 4.

2.3 Re-materialization in Deep Neural Networks

The use of re-materialization strategies has recently been advocated for DNNs in several papers. Specific work on the popular neural network DenseNet [Pleiss et al. 2017] has shown that it is possible to reduce the memory cost to a linear relationship with network depth (instead of quadratic) by recomputing concatenation and batch normalization operations during back-propagation.

A direct adaptation of the results on homogeneous chains was proposed in the case of Recurrent Neural Networks (RNNs) in [Gruslys et al. 2016], but cannot be extended to other DNNs. The adaptation of automatic differentiation techniques to a completely heterogeneous context, which corresponds to the case of DNNs, has recently been addressed by multiple papers [Beaumont et al. 2019b; Chen et al. 2016; Feng and Huang 2018; Gruslys et al. 2016; Jain et al. 2019; Kirisame et al. 2020; Kumar et al. 2019; Kusumoto et al. 2019], but it is generally non-trivial. An example is the fundamental memory persistency result which states that any activation stored during the forward phase in an optimal solution will be kept until it is used in the backward phase. This is true in the homogeneous context, but we prove in Section 4.1 that it does not generalize to the case of heterogeneous activation sizes. This implies that the generalization proposed in [Gruslys et al. 2016] is not optimal in general for the heterogeneous case. In Section 4.3, we give a dynamic program to solve the fully heterogeneous problem where both computation times and activation sizes of all layers can be different. Another generalization of the results on homogeneous chains has been proposed to derive optimal re-materialization strategies for join networks [Beaumont et al. 2019b], which are made of several homogeneous chains joined together at the end.

The interest for arbitrary computational graphs has led to several heuristic approaches to find good re-materialization strategies. In [Feng and Huang 2018], a polynomial algorithm is provided that finds the re-materialization strategy for the forward propagation that minimizes memory used to process an ACG, under the assumption that activation deletion is not allowed during the backward phase. This assumption is very strong and restrictive in practice, especially in the case of deep networks: this prevents deleting activations from memory during the subsequent forward phases required by the missing activations. We call such solutions *single-pass* re-materialization strategies. In contrast, the process is fully recursive in the AD literature, which allows the complete memory to be used throughout the whole training process, since the memory released shortly after the computation of the loss can be used later.

A similar problem is considered in [Kumar et al. 2019] where activation deletion during the backward propagation is possible. However, the authors only provide an approximate solution through the computation of a tree-width decomposition of the graph. The performance of this solution depends on the quality of the decomposition, which is an NP-complete problem for which

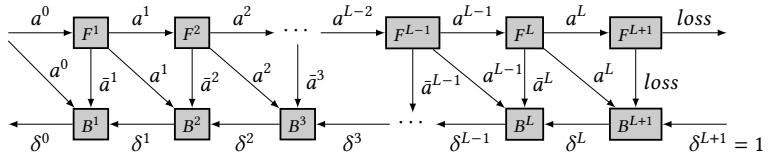
constant-factor approximation algorithms need to be used. In addition, this procedure outputs one solution with logarithmically bounded memory usage and with a performance guarantee on execution time, but it cannot be parametrized to use more or less memory depending on what is available on the device. The problem that we address in the present paper is more general and leads to more efficient solutions, although limited to networks that can be written as sequences of layers. Our algorithms are able to provide a strategy with minimal duration whose memory usage remains below a given budget.

Several authors have also proposed to rely on Integer Linear Programming to find an optimal re-materialization strategy. This was originally restricted to the tree case [Lotz et al. 2016b], and the general problem was addressed in *Checkmate* [Jain et al. 2019]. This Integer Linear Program can handle arbitrary graphs by assuming a fixed ordering of the execution, and can provide a solution of minimum runtime given a memory limit. However, solving this ILP is very computationally intensive and does not converge in a reasonable time as soon as the network exceeds a few dozen layers. We have implemented this approach, originally designed for TensorFlow, by adapting it to the PyTorch framework. Our experimental results (Section 5) show that our improved model for memory cost allows us to obtain re-materialization solutions of better quality, and that relying on a dynamic program rather than an ILP allows us to find solutions faster and on a larger class of networks.

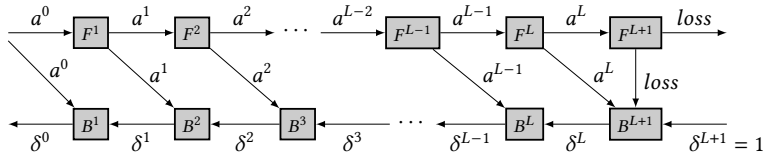
At last, other approaches finely control the tradeoff between memory and computation. A heuristic approach is proposed in [Kirisame et al. 2020] to dynamically decide which activations to delete, based on an estimation of the cost of re-materialization. One benefit of this approach is that, unlike ours, it can be applied to data-dependent networks, where the computation graph depends on the input. However, this approach does not provide any optimality guarantee. In [Kusumoto et al. 2019], the authors also consider a general ACG framework. Their work can be seen as a generalization of [Chen et al. 2016] algorithm to ACGs, in which the ACG is decomposed into groups of nodes. During the forward phase, only the boundaries between groups are materialized. During the backward phase, all the activations of the group are re-materialized from its input boundary to perform all the gradient computations of a group, so that the backward phase is performed without additional re-materialization operations. The advantage of this approach is that it is tractable for ACGs using dynamic programming. However, the search is restricted to single-pass re-materialization strategies as in [Chen et al. 2016] and [Feng and Huang 2018]. As we will prove later in the case of chains, preventing the deletion of activations during the backward phase induces significant memory waste and additional computational costs.

For practical usage, only one implementation of re-materialization exists in the PyTorch library [Pytorch contributors 2018]. It is based upon a simple periodic and single-pass re-materialization strategy which exploits the ideas presented in [Chen et al. 2016]. In this strategy, the chain is divided in equal-length segments, and only the input of each segment is materialized during the forward phase. Such a strategy provides non-optimal solutions in terms of throughput and memory usage, because it does not benefit from the fact that more memory is available when computing the backward phase of the first segment (since values materialized for later segments have already been used). Nevertheless, this implementation is succesfully used to process very large models [Goyal 2018], whose training is infeasible without re-materialization.

To the best of our knowledge, this work presents the first theoretical results in a context where both computation times and activation sizes are heterogeneous. Another specificity of our work is that we model the ability, offered in DNN frameworks, to combine two types of activation savings: by either storing only the layer inputs (as done in most of AD literature), or by recording the complete history of operations that produced the outputs (as available in autograd tools). This



(a) Graph for a general sequential deep neural network.



(b) Graph for an automatic differentiation application.

Fig. 1. Graphs of a general sequential Deep Neural Network and an Automatic Differentiation application.

second possibility has been denoted as "tape" in [Lotz et al. 2016a; Naumann 2008] in the case of programs represented as call trees, and we describe the connection with our model in Section 3.3.

3 MODEL AND PROBLEM FORMULATION

We present here the computation model used throughout the paper to describe different rematerialization strategies that can be used during an iteration of the back-propagation algorithm. Moreover, we highlight how this model differs from the classic Automatic Differentiation model [Griewank 1989].

3.1 Model for Back-Propagation Algorithm

Let us consider a chain of L stages (*i.e.* layers or blocks of layers), numbered from 1 to L. Each stage ℓ is associated both to a forward operation F^ℓ and a backward operation B^ℓ (see Figure 1a). We denote by a^ℓ the activation tensor output of F^ℓ and by $\delta^\ell = \frac{\partial \mathcal{L}}{\partial a^\ell}$ the back-propagated intermediate value provided as the input of the backward operation B^ℓ. For notational convenience, the loss $\mathcal{L}$, which is computed between the forward pass and the backward pass, is represented by two operations F^{L+1} and B^{L+1}. We also use a^0 to denote the input data and δ^{L+1} to denote the first back-propagated value, $\delta^{L+1} = \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$. In this work, we assume that the computation associated with each operation can be an arbitrarily complex operation. For this reason, we include additional dependencies compared to the classical literature on automatic differentiation, as illustrated in Figure 1a. Indeed, since F^ℓ can represent an arbitrary DAG (Directed Acyclic Graph) of elementary operations, performing the back-propagation requires us to keep in memory all intermediate activations computed by these elementary operations. We denote by $\bar{a}^\ell$ the set of all intermediate activation values needed to compute B^ℓ (for simplicity, we consider that $\bar{a}^\ell$ contains a^ℓ, but not $a^{\ell-1}$).

Data dependencies. The dependencies between different operations are described in Figure 1a. The classical dependencies introduced by the gradient back-propagation algorithm [LeCun et al. 2015], depicted in Figure 1b, are a special case where all the intermediate results $\bar{a}^\ell$ are empty. Overall:

- F^ℓ produces a^ℓ from $a^{\ell-1}$;
- B^ℓ produces $\delta^{\ell-1}$ from δ^ℓ, $\bar{a}^\ell$ and $a^{\ell-1}$.

	Operation	Input	Output	Time	Memory overhead
F^ℓ_{all}	Forward and save all	$\{a^{\ell-1}\}$ $\{\bar{a}^{\ell-1}\}$	$\{a^{\ell-1},\ \bar{a}^\ell\}$ $\{\bar{a}^{\ell-1},\ \bar{a}^\ell\}$	u^ℓ_f	o^ℓ_f
F^ℓ_{ck}	Forward and materialize input	$\{a^{\ell-1}\}$ $\{\bar{a}^{\ell-1}\}$	$\{a^{\ell-1},\ a^\ell\}$ $\{\bar{a}^{\ell-1},\ a^\ell\}$	u^ℓ_f	o^ℓ_f
$F^\ell_\emptyset$	Forward without saving	$\{a^{\ell-1}\}$	$\{a^\ell\}$	u^ℓ_f	o^ℓ_f
B^l	Backward step	$\{\delta^\ell,\ \bar{a}^\ell,\ a^{\ell-1}\}$ $\{\delta^\ell,\ \bar{a}^\ell,\ \bar{a}^{\ell-1}\}$	$\{\delta^{\ell-1}\}$ $\{\delta^{\ell-1},\ \bar{a}^{\ell-1}\}$	u^ℓ_b	o^ℓ_b

Table 1. Operations performed by a sequence. The second line shows the behavior when $\bar{a}^{\ell-1}$ is used instead of $a^{\ell-1}$. Performing an operation *replaces* the input values by the output values; if an input value is kept in memory, it explicitly appears as an output value, but its value does not change.

In classical implementations of the back-propagation algorithm, all activation values are materialized in memory during the forward step F^ℓ until the backward step B^ℓ is completed, i.e. in practice, the whole computational graph is stored. The basic principle of re-materialization is to trade memory for computing time by materializing only certain activations in memory and then re-materializing the others when the backward steps require them.

In this work, we introduce three different types of forward operations:

- $F^\ell_\emptyset$ computes F^ℓ without saving any data in memory. It is equivalent to calling a forward operation under `no_grad()` in PyTorch:

```
with torch.no_grad():
    x = F[i](x)
```

- F^ℓ_{ck} computes F^ℓ while *saving the input* $a^{\ell-1}$ of the block of layers ℓ. It can be implemented in PyTorch by calling a forward under `no_grad()`, but without overwriting the input tensor `x`:

```
with torch.no_grad():
    y = F[i](x)
```

- F^ℓ_{all} computes F^ℓ while saving the input $a^{\ell-1}$ and all the intermediate data $\bar{a}^\ell$ required by the backward step B^ℓ (i.e. *saving all*). This involves executing a forward operation under `enable_grad()` context, which tells to PyTorch to *record* this operation, keeping track of each elemental computational step and all intermediate outputs:

```
with torch.enable_grad():
    y = F[i](x)
```

B^ℓ cannot be computed until F^ℓ_{all} has been processed. However, F^ℓ_{all} uses more memory than F^ℓ_{ck}, so it may be more efficient to compute F^ℓ_{ck} first and then compute F^ℓ_{all} from $a^{\ell-1}$ later in the sequence of instructions. The list of available operations is given in Table 1. Each operation requires a certain input and produces a certain output, which *replaces* the input in memory (if an input value is kept in memory, it explicitly appears as an output value, but this does not mean that it is overwritten).

A strategy for computing δ^0 given a^0 is described by a *sequence* of these operations. The processing of a sequence consists in executing all operations one after the other, replacing the input of each operation by its output in the memory. We are interested in *valid* sequences, as defined below:

Index ℓ	a^ℓ	$\bar{a}^\ell$	o_f^ℓ	o_b^ℓ	u_f^ℓ	u_b^ℓ
0	7.63	7.63	N/A	N/A	N/A	N/A
1	9.54	9.54	0.00	20.01	1.60	3.05
2	10.68	10.68	0.00	27.64	2.20	4.48
3	11.06	11.08	0.00	30.99	2.44	5.09
4	10.68	10.66	0.00	30.99	2.51	4.93
5	9.54	9.54	0.00	27.64	2.10	4.21
6	7.63	7.63	0.00	19.08	1.43	3.34
7	N/A	N/A	0.00	0.00	0.00	0.00

Table 2. Measurements for the toy example on Nvidia V100. Memory sizes are in MB, times are in milliseconds.

Definition 3.1 (Valid sequence). A sequence of $F_{all}^\ell \; F_{ck}^\ell \; F_\varnothing^\ell \; B^\ell$ operations is *valid* if it computes δ^0 from a^0 while satisfying all data dependencies defined in Table 1.

Definition 3.2 (Makespan of a valid sequence). The *makespan* of a valid sequence is the sum of the durations of all operations in the sequence.

Memory requirements. In this paper, we only focus on the memory used by activations. We assume that the memory required to store the model and the gradients of the model parameters has already been allocated and removed from the initial available memory. We further assume that the memory needed to store each data item is known.

We denote by ω_a^ℓ the memory required to materialize a^ℓ, $\omega_{\bar{a}}^\ell$ the memory required to materialize $\bar{a}^\ell$ and ω_δ^ℓ the memory required to materialize δ^ℓ (in practice, $\omega_a^\ell = \omega_\delta^\ell$).

Since each stage of the chain can be arbitrarily complex, it may induce a memory peak higher than the size of the sum of its input and output data. This is modeled by introducing memory overheads of operations, as shown in Table 1.

Definition 3.3 (Memory usage of an operation). The memory required to compute an operation is the sum of its input and output data plus a memory overhead o_f^ℓ or o_b^ℓ.

Definition 3.4 (Memory requirement of a sequence). The *maximum memory usage* of a valid sequence is the maximum, over all operations in the sequence, of the memory occupied by stored data (including the output of the operation) plus the overhead of the operation.

Problem formulation. Overall, the problem is to find the optimal sequence of operations that minimizes the computation time while taking into account a memory constraint. We can formalize our optimization problem as below:

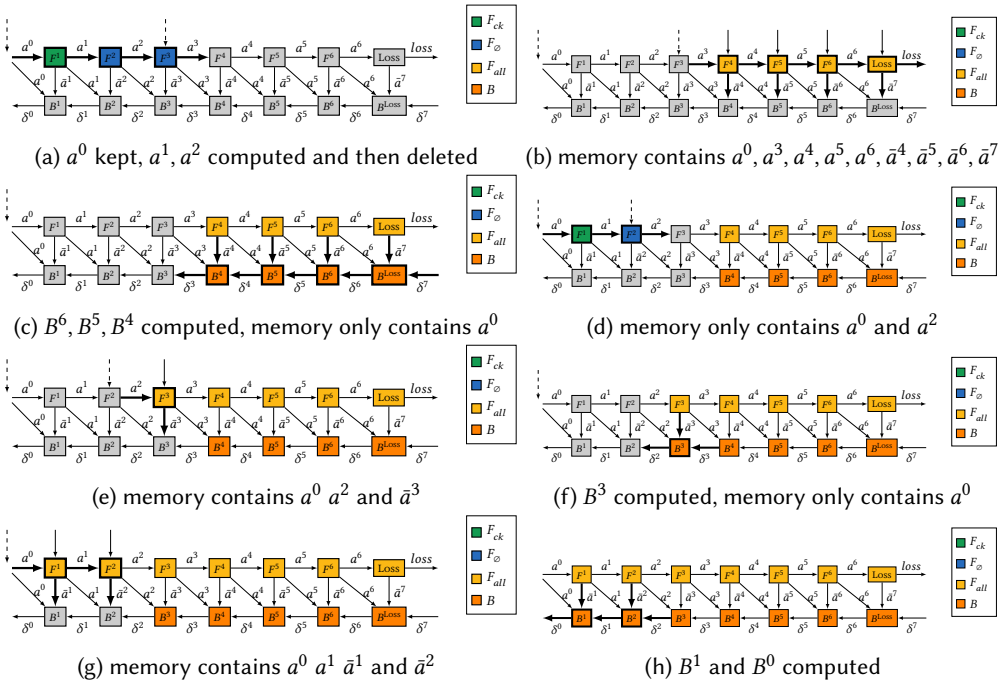
Definition 3.5 (Problem Remat(L, M)). Given

- a chain length L;
- activations a^ℓ, $\bar{a}^\ell$ and gradients $\delta^\ell, \forall 0 \leq \ell \leq L$;
- memory overheads o_f^ℓ, o_b^ℓ and time costs $u_f^\ell, u_b^\ell, \forall 1 \leq \ell \leq L$;
- a memory limit M;

find a valid sequence whose maximal memory usage is at most M and whose makespan is minimal.

3.2 Toy Example

We consider a toy network composed of 6 fully-connected linear layers: each layer corresponds to a matrix of respective sizes (2000×2500), (2500×2800), (2800×2900), (2900×2800), (2800×2500) and (2500×2000). We also consider a batch of size 1000, which yields the measurements provided



(a) a^0 kept, a^1, a^2 computed and then deleted

(b) memory contains $a^0, a^3, a^4, a^5, a^6, \bar{a}^4, \bar{a}^5, \bar{a}^6, \bar{a}^7$

(c) B^6, B^5, B^4 computed, memory only contains a^0

(d) memory only contains a^0 and a^2

(e) memory contains a^0 a^2 and $\bar{a}^3$

(f) B^3 computed, memory only contains a^0

(g) memory contains a^0 a^1 $\bar{a}^1$ and $\bar{a}^2$

(h) B^1 and B^0 computed

Fig. 2. Toy Example, only backward operations and forward activations are specified.

in Table 2, performed on a NVIDIA V100 GPU. For example, the input size a^0 corresponds to a single precision tensor of dimensions $(1000, 2000)$, for a memory usage of $4 \cdot 1000 \cdot 2000 = 8 \cdot 10^6$ bytes = 7.63MB. The strategy with no re-materialization has a makespan of 37.4ms (equal to $\sum_\ell u_f^\ell + u_b^\ell$), and uses 107MB of peak memory. The peak memory is reached when computing B^5, where the memory contains a^0, $\bar{a}^1$ through $\bar{a}^5$, δ^4 and δ^5 for a total memory usage of 79.35MB, added to the temporary memory usage o_b^5 of 27.64MB.

If only $M = 90$MB are available, it is no longer possible to store all the activations. An optimal sequence for this case is:

$$F_{ck}^1 F_\varnothing^2 F_\varnothing^3 F_{all}^4 \cdots F_{all}^7 B^7 \cdots B^4 F_{ck}^1 F_\varnothing^2 F_{all}^3 B^3 F_{all}^1 F_{all}^2 B^2 B^1.$$

Figure 2 describes the execution of this optimal sequence. Its maximum memory usage is also reached when computing B^5, but since fewer activations are stored in memory, the peak usage is only 86.8MB, and the makespan is 47.4ms. We can see the effect of heterogeneous memory requirements: in step (2c), it is possible to compute B^6 with $a^0, \bar{a}^4, \bar{a}^5$ and $\bar{a}^6$ in memory; but in step (2f) it is necessary to compute B^3 with only a^0 and $\bar{a}^3$. Indeed, since this layer operates on a larger matrix size, the temporary memory requirements o_b^3 of B^3 are significantly larger than o_b^6. Thus, the optimal sequence is forced to recompute F^1 and F^2 another time to finish the computation.

3.3 Comparison with alternative approaches

Other approaches for chain networks. Consider an example where $L = 4$. The following sequences of operations are valid:

$$F^1_{ck}, F^2_{\emptyset}, F^3_{ck}, F^4_{all}, F^5_{all}, B^5, B^4, F^3_{all}, B^3, F^1_{all}, F^2_{all}, B^2, B^1$$
$$F^1_{ck}, F^2_{\emptyset}, F^3_{ck}, F^4_{ck}, F^5_{all}, B^5, F^4_{all}, B^4, F^3_{all}, B^3, F^1_{ck}, F^2_{all}, B^2, F^1_{all}, B^1$$
$$F^1_{ck}, F^2_{\emptyset}, F^3_{all}, F^4_{ck}, F^5_{all}, B^5, F^4_{all}, B^4, B^3, F^1_{all}, F^2_{all}, B^2, B^1$$

The first sequence is a typical example of single-pass re-materialization approaches as the one described in [Chen et al. 2016; Feng and Huang 2018; Kusumoto et al. 2019]. The second sequence keeps partial results even during the backward phase (as shown by the usage of F^1_{ck} in the backward phase). However, this remains restrictive: F_{all} operations are only performed right before the corresponding backward operation. This corresponds to using a model based on the dependencies in Figure 1b, as in [Griewank and Walther 2000]: in that paper, the authors argue that their solution can be adapted to complex stages by performing $F^\ell_{all}B^\ell$ when computing the back-propagation of layer ℓ while only having a^{l-1} in memory.

Finally, the third sequence is different from the others: it includes an F_{all} forward operation in the middle of the forward propagation, while not being directly followed by the corresponding backward propagation. This is shown by the fact that the output of F^3_{all} is used to compute F^4_{ck} first, then F^4_{all} and finally B^3. This sequence uses the dependencies described in Figure 1a, which are adapted to what autograd frameworks such as PyTorch and TensorFlow provide. There are additional dependencies in Figure 1a that are not present in Figure 1b (edges from F^ℓ to B^ℓ), which carry all the intermediate activations required to perform the backward step.

By taking these additional dependencies into account, the model considered in this paper is able to account for a larger set of valid strategies than previous approaches for chain networks. The optimal re-materialization strategy for the model of Figure 1a cannot be directly derived from the optimal solution provided in [Griewank and Walther 2000] for the model of Figure 1b. The main contribution of Section 4 is a deeper analysis of the problem resulting in an optimal algorithm for this more precise model.

Approaches based on call tree. Another line of work considers programs described as a tree of function calls, where checkpointing can only take place at the call sites of these functions [Lotz et al. 2016a; Naumann 2008]. This work also considers saving the intermediate activations needed to perform the backward step, but with two main differences compared to the present paper.

First, since this call-tree approach is in the context of a "semantic source code transformation performed by a compiler", intermediate values "cannot be assumed to be persistent due to potential overwriting of memory locations" by the program code. In contrast, our work is in the context of DNN frameworks such as PyTorch, where the intermediate values are complete tensors instead of scalar values, using a programming paradigm where each computation creates a new tensor rather than overwriting the previous one.

As a result, the set of operations used in the call-tree approach contains additional "tape" and "restore" operations for saving and restoring intermediate data in/from another memory location. Instead, our operations contain implicit "delete" operations to remove some intermediate value that is no longer needed, like the "forward without saving" operation. For the same reason, the call-tree approach considers time costs for "storing" or "restoring" intermediate values, whereas we are only concerned with the recomputation cost.

The second difference is in the execution model. In the call-tree model, a function can be reversed either in "split" or "joint" mode. In "split" mode, the intermediate data is saved during the forward phase to be able to perform the backward step later, corresponding to $F_{all} \cdots B$. In "joint" mode, only the arguments to the function are saved, and the function is recomputed during the backward

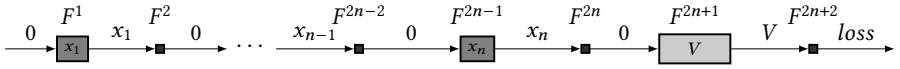


Fig. 3. An instance of Remat(L, M), where the values inside the nodes indicate the durations of the task, and the values above the arrows show the data sizes of the output tensors.

phase, corresponding to $F_{ck} \cdots F_{all} B$. One can express a chain of DNN layers as a shallow call tree with depth 1 and one function per layer to apply this approach in our context. But this would not allow the usage of $F_\varnothing$ operations: all layers would save either their input or all the intermediate data. Our approach allows for solutions where only a small number of layers save their input, which results in significantly lower memory usage.

3.4 Complexity

Theorem 3.6. *The decision problem associated with Remat(L, M), i.e. is there a valid solution to Remat(L, M) whose makespan is not more than T, is NP-complete in the weak sense.*

Proof. It is clear that Remat(L, M) belongs to NP: for a fixed sequence, it is possible to check its validity and to simulate its execution in linear time, verifying that its makespan is at most T and its maximum memory usage at most M. In addition, for a chain of length L, the valid sequence which recomputes all values during the backward phase has length $O(L^2)$.

Let us now show that Remat(L, M) is NP-hard by a reduction from the 2-Partition problem [Garey and Johnson 1979]. We formulate the 2-Partition problem in the following way: for a set of values $S = \{x_i \in \mathbb{N} | 1 \leq i \leq n\}$, such that $\sum_{i=1}^{n} x_i = 2V$, does there exist two disjoint subsets S_1 and S_2 where $S_1 \cup S_2 = S$, such that $\sum_{i \in S_1} x_i = \sum_{i \in S_2} x_i = V$?

We now build a chain whose forward phase is shown in Figure 3. The associated instance of Remat(L, M) can be described as follows:

- $L + 1 = 2n + 2, \quad M = 3V, \quad U_B = \sum_{i=1}^{L+1} u_b^i, \quad T = 4V + U_B$;
- $u_f^{2k+1} = x_k$ and $\omega_a^{2k+1} = x_k$ for $1 \leq k < n$;
- $u_f^{2k} = 0$ and $\omega_a^{2k} = 0$ for $1 \leq k \leq n + 1$;
- $u_f^{2n+1} = V$ and $\omega_a^{2n+1} = V$;
- Input size $\omega_a^0 = 0$;
- u_b^i for any $i \leq L$ can be any positive value (this proof is valid for any backward durations);
- $\omega_{\bar{a}}^i = \omega_\delta^i = \omega_a^i$ for all $0 \leq i \leq L + 1$.

We first show that if there exists a solution to the 2-Partition problem, then this instance has a solution with makespan at most $T = 4V + U_B$. Indeed, given a solution (S_1, S_2) to the 2-Partition instance, it is possible to store only the activations a^{2k-1} with $k \in S_1$, during the processing of B^{2n+1}: this requires at most $\delta^{2n+1} + \bar{a}^{2n+1} + a^{2n} + \delta^{2n} = 2V$ amount of memory. During the backward phase, it is necessary to recompute F^{2k-1} for all $k \in S_2$, thus the total duration of the recomputations is $\sum_{k \in S_2} u_f^{2k-1} = \sum_{k \in S_2} x_k = V$. As the forward phase lasts $\sum_{k=1}^{n} u_f^{2k-1} + V = 3V$, it results in total execution time $T = 4V + U_B$.

Reciprocally, we now show that if there is a sequence with a makespan at most $4V + U_B$ for this problem, then there exists a solution to the 2-Partition problem. For any sequence, the forward execution time is exactly $3V$, the backward execution is U_B, which means that the total recomputation time is at most V. Let us denote S_2 the set of indices k such that the activation a^{2k-1} is discarded during the forward phase. These activations need to be recomputed during the backward phase.

It is not useful to discard the even-indexed activations a^{2k}, since they have zero size. Since the recomputation time is at most V, $\sum_{k\in S_2} u_f^{2k-1} = \sum_{k\in S_2} x_k \leq V$.

We now argue that $\sum_{k\in S_2} x_k \geq V$. Indeed, the memory required during B^{2n+1} is $\sum_{k\notin S_2} a^{2k-1} + \delta^{2n} + \delta^{2n+1} = 2V + \sum_{k\notin S_2} a^{2k-1}$. Since the maximum memory usage of the sequence is at most $M = 3V$, this implies $\sum_{k\in S_2} a^{2k-1} = \sum_{k\in S_2} x_k \geq V$. Combining both inequalities gives $\sum_{k\in S_2} x_k = V$. Finally, by setting $S_1 = S \setminus S_2$, we obtain a solution to the 2-Partition problem, which completes the proof. □

4 OPTIMAL RE-MATERIALIZATION ALGORITHM

In this section, we provide an optimal dynamic programming algorithm to solve Remat(L, M). We first present the memory persistence property in Section 4.1 that is used in Automatic Differentiation literature to prove the optimality of the dynamic programs. The first theoretical contribution of this paper is to provide a counter example which proves that there exist cases where no memory persistent sequence is optimal. This is why the direct adaptation of the dynamic programming algorithm proposed in Section 12.3 of [Griewank and Walther 2008] is non-trivial in the heterogeneous case. This remark applies in particular to the model described in Figure 1a, but also to the AD model depicted in Figure 1b and used in [Gruslys et al. 2016], where memory persistence is implicitly used when deriving the dynamic program, without being explicitly stated.

A second theoretical contribution is described in Section 4.2, where we define a modified version called *weak memory persistence* and prove that even in the heterogeneous case, there always exists an optimal sequence satisfying the weak memory persistence property.

We propose a dynamic program that computes the optimal *weak memory persistent* solution in Section 4.3. Since this algorithm induces extra computational cost, we also describe how to compute the optimal *memory persistent* sequence in Section 4.4. The resulting algorithm, although not theoretically optimal for Remat(L, M), is of crucial practical interest. In Section 5, based on a large set of experiments on neural networks used for computer vision, we observe that the performance of algorithms based on the *memory persistence* and *weak memory persistence* assumptions are identical in practice. This justifies the use of *memory persistence* assumptions to generate efficient re-materialization sequences, as we propose in Rotor [HiePACS team 2019].

4.1 Considerations on Memory Persistence

Definition 4.1 (Memory persistent sequence). A sequence is said to be memory persistent [Griewank and Walther 2008] if any materialized value is kept in memory until it is used in the backward phase. More precisely, memory persistency requires that if F_{ck}^i or F_{all}^i is performed for some layer i, then the subsequence of operations between that forward operation and B^i does not contain $F_\emptyset^i$, nor any forward or backward operation related to layers i' for $i' < i$.

Memory persistency is an important property for AD that facilitates the search of optimal sequences. Indeed, a key observation for homogeneous activation sizes is that all optimal sequences in terms of makespan are memory persistent: if an activation a^i is materialized, but deleted before being used for B^{i+1}, then it would be more efficient to materialize a^{i+1} since it avoids recomputing F^{i+1}, with the exact same memory usage.

However, when activation sizes are heterogeneous, this property no longer holds. We show an example in Figure 4 with a chain of length $L = n + 2$ for any n. All backward computational costs u_b^ℓ are 0, as well as most of the forward computational costs, except $u_f^1 = k$ and $u_f^2 = 2$. Most forward activations sizes ω_a^i are 3, except $\omega_a^0 = \omega_a^{L+1} = 0, \omega_a^1 = 1$ and $\omega_a^L = 4$. Additionally for any i, $\omega_a^i = \omega_{\bar{a}}^i = \omega_\delta^i$. The memory limit is $M = 15$.

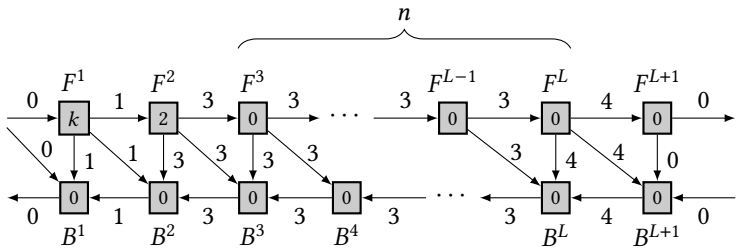


Fig. 4. The counter example where no memory persistent solution is optimal. Values on the edges represent the size of the activations, values inside nodes represent the computing time of the layers. The memory limit is $M = 15$.

Since computing B^L requires a memory of $\omega_a^{L-1} + \omega_{\bar{a}}^L + \omega_\delta^L + \omega_\delta^{L-1} = 3 + 4 + 4 + 3 = 14$, it is not possible to materialize a^2 (whose size is 3) during the forward phase, as our memory budget is only $M = 15$. Thus, we can identify two valid memory persistent sequences, which are potentially optimal: either a^1 is materialized during the forward phase, or it is deleted. In the first case, a^2 is never materialized until B^3, and thus F^2 is processed n times. This results in a makespan $T_1 = k + 2n$. In the second case, the forward phase is performed with only a^0 materialized in memory, until B^L is computed. Then the computation starts from the beginning, and this time it is possible to materialize a^2, which allows us to compute all of the 0-cost F^i without recomputing F^2. At the end, it is necessary to recompute F^1, which results in a makespan $T_2 = 2(k + 2) + k = 3k + 4$.

It is also possible to build the following valid but not memory persistent sequence: a^1 is materialized during the forward phase, and kept in memory until the second time that F^2 is computed. Indeed, at that time, F^L has already been computed. Thus it is possible to materialize a^2 instead of a^1 (but not both at the same time since computing F^{L-1} requires a memory of 12). At the end, it is necessary to recompute F^1, and this results in a makespan $T_0 = k + 2 \times 2 + k = 2k + 4$.

Setting $k = n - 2$ ensures that $T_1 = T_2 = 3n - 2$, while $T_0 = 2n + 2$. In that case, the makespan of the non memory persistent sequence is lower by a factor of $\frac{3}{2}$ than the makespan of any memory persistent sequence.

Nevertheless, the instance of the problem described in Figure 4 is an ad hoc construction and such a counter example is not likely to happen in practice. Actually, in Section 5, we observe that for every classical vision networks we experimentally considered, the computed optimal sequences are memory persistent, even if we do not enforce this property. However, the characterization of the class of neural networks for which there exists a memory persistent optimal sequence is left for future works.

4.2 Properties of Heterogeneous Problem

To find an optimal sequence in the fully heterogeneous case using dynamic programming, we introduce a slightly weaker version of memory persistence. We first divide a valid sequence into *phases*, as below:

Definition 4.2 (Phase k). For a given sequence $\mathcal{S}$, we denote as *phase k* for $k \leq L$ the subsequence of operations that take place between two backward operations B^{k+1} and B^k. In addition, *phase* $L + 1$ corresponds to all operations in the sequence of execution before B^{L+1} (thus to the entire forward propagation).

These phases are ordered by decreasing indices, i.e. for any $0 \leq k \leq L+1$, phase k precedes phase $k-1$. For instance, phase $L+1$ contains all operations between the beginning of the processing and B^{L+1}, while phase L contains all operations between B^{L+1} and B^L. In turn, phase 0 is the last phase in the execution.

Definition 4.3 (Weak memory persistence). A sequence is said to be weakly memory persistent if it satifies the following condition: if activation a^{k-1} is materialized with either F^k_{ck} or F^k_{all} for some k, then no recomputation F^i is performed for $i < k$ until a^{k-1} is discarded with B^k or $F^k_\emptyset$.

This definition adds the possibility to discard an activation a^{k-1} with $F^k_\emptyset$, whereas with the strong memory persistence, only B^k can be used to discard a previously materialized activation. It covers a larger set of possible solutions, which as we show below always contains an optimal solution.

Lemma 4.4. *For any instance of Remat(L, M), there exists an optimal solution which is weakly memory persistent.*

Proof. Suppose that there exists an optimal solution that does not satisfy the weak memory persistence property and let us prove that it is possible to transform it into a sequence with the same cost that is weakly memory persistent. Let phase h (subsequence of executions between B^{h+1} and B^h) corresponds to the earliest phase when some F^i is performed, while both a^k and a^i are materialized in the memory at the beginning of the phase for $h > k > i$. Among all such optimal solutions, let us denote by $\mathcal{S}$ the one with the smallest possible index h.

Since $\mathcal{S}$ performs the F^i operation, the result should be used to produce an activation a^j for $i < j < k$. Indeed, since a^k is available, producing an activation a^j for $j > k$ does not require us to compute F^i. Let us prove that $\mathcal{S}$ can be transformed into a sequence $\mathcal{S}'$ with the same makespan, but where the sequence of operations, denoted as $\mathcal{F}_i^{j-1}$, from F^i to F^{j-1} takes place in the next phase.

If $\mathcal{S}$ does not delete a^i when computing F^i, since a^j is not used in phase h, these recomputations $\mathcal{F}_i^{j-1}$ can be delayed until the beginning of phase $h-1$ without increasing the makespan. In the end, we obtain the sequence $\mathcal{S}'$ where all phases except h and $h-1$ are equivalent to those of $\mathcal{S}$, while the total memory consumption of $\mathcal{S}'$ is not larger. Therefore, this sequence does not overcome the memory limit and is both valid and optimal. As $\mathcal{S}'$ is an optimal solution, it contradicts to the assumption that $\mathcal{S}$ has the minimal index for phase h.

Otherwise, if activation a^i is not kept in the memory at the end of phase h, then under the condition that $\omega_a^j \geq \omega_a^i$, forward operations from i to $j-1$ can be moved to phase $h-1$, leading to sequence $\mathcal{S}'$. As in the previous case, $\mathcal{S}'$ remains valid since it requires at most the same amount of memory. However, when $\omega_a^j < \omega_a^i$, the sequence $\mathcal{S}$ cannot be optimal. Indeed, as phase h is the earliest phase when F^i is recomputed with a^i and a^k both materialized before the beginning of the phase, then storing a^i in the first place has been used just for recomputing a^j and then it has been deleted. Instead, we can consider another sequence $\mathcal{S}'$ where we keep a^j from the very beginning but not a^i, while everything else is the same. This sequence $\mathcal{S}'$ is a valid sequence with smaller overall memory consumption and which performs fewer recomputations, providing a smaller makespan, which contradicts the optimality of $\mathcal{S}$.

Finally, we have shown that the optimal sequence $\mathcal{S}$ where recomputation takes place at phase h can be transformed into a sequence $\mathcal{S}'$ preserving the optimality and where the recomputation takes place at phase $h+1$, which contradicts the initial assumption.

□

Using Lemma 4.4, we can concentrate on finding an optimal weakly memory persistent sequence. During the execution of a weakly memory persistent sequence, the materialized activation with the

largest index is called the *floating activation*. As shown in the proof above, this floating activation can be replaced by another activation with the following restrictions:

Corollary 4.5 (Floating Materialized Activation). *In any optimal weakly memory persistent sequence, at any instant, only the floating activation can be replaced by a new materialized activation in memory. Moreover, this new materialized activation has a larger index and a larger or equal memory size, and thus becomes the new floating activation.*

Lemma 4.6 (Memory persistence for $\bar{a}$). *There exists an optimal solution where no materialized $\bar{a}^i$ activation is deleted before the corresponding backward step B^i.*

Proof. Let us assume that there exists an optimal solution where, for some i, $\bar{a}^i$ is materialized and later deleted before the execution of B^i. Let us consider the solution where a^i is materialized instead of $\bar{a}^i$. It is possible to perform all recomputations using a^i instead of $\bar{a}^i$. Since $\omega_a^i \leq \omega_{\bar{a}}^i$, it also helps to reduce memory consumption, thus producing a solution that is both valid and optimal in terms of makespan. This proves the lemma.

□

4.3 Dynamic Programming with Floating Materialized Activations

We prove that it is possible to compute the optimal re-materialization sequence in the fully heterogeneous case corresponding to the model depicted in Figure 1a, based on the results proved in Section 4.2. For a chain of length L, let us denote by $C_{\text{FL}}(s,t,\ell,m)$ the optimal execution time to perform backward steps from B^ℓ till B^t of the chain starting from position s, where

- a^{s-1} and δ^ℓ are the only materialized values before the processing of the chain,
- δ^{t-1} is the only materialized value after the processing of the chain and
- the peak memory to process the chain is at most m, without taking into account the memory occupied by a^{s-1}.

As only δ^{t-1} stays in the memory after the execution, it means that the floating materialized activation that first appears at position $s-1$ should be fully exploited and deleted by the end of the execution. Let us introduce the following notations

$$m_{\varnothing}^{s,t,\ell} = \max \begin{cases} \omega_\delta^\ell + \omega_a^s + o_f^s, \\ \omega_\delta^\ell + \max\limits_{s+1 \leq j < \ell} \left\{ \omega_a^{j-1} + \omega_a^j + o_f^j \right\} \end{cases} \qquad \text{and } m_{all}^{s,t,\ell} = \max \begin{cases} \omega_\delta^\ell + \omega_{\bar{a}}^s + o_f^s, \\ \omega_\delta^s + \omega_\delta^{s-1} + \omega_{\bar{a}}^s + o_b^s \end{cases}.$$

Thus, $m_{\varnothing}^{s,t,\ell}$ for $1 \leq s \leq t < \ell \leq L+1$ denotes the memory peak to compute all $F_\varnothing$ steps from s to ℓ, and $m_{all}^{s,t,\ell}$ for $1 \leq s \leq \ell \leq L+1$ denotes the memory peak to compute F_{all}^s and B^s.

Theorem 4.7. *$C_{FL}(s,t,\ell,m)$, the optimal makespan to process the chain from stage s with backward steps from B^ℓ till B^t with available memory m, is given by:*

$$C_{FL}(s,s,s,m) = \begin{cases} u_f^s + u_b^s & m \geq m_{all}^{s,s,s} \\ \infty & m < m_{all}^{s,s,s} \end{cases} \tag{1}$$

$$C_{FL}(s,t,\ell,m) = \min\left(C_1(s,t,\ell,m), \quad C_2(s,t,\ell,m)\right) \tag{2}$$

$$C_1(s,t,\ell,m) = \begin{cases} \min\limits_{\substack{s',r,t' \\ s\le r<s'\le t' \\ \omega_a^{s-1}\le\omega_a^{r-1} \\ t<t'\le\ell}} C_{ck}(s,t,r,s',t',\ell,m) & m \ge m_{\emptyset}^{s,t,\ell} \text{ and } t \neq \ell \\ \infty & m < m_{\emptyset}^{s,t,\ell} \text{ or } t = \ell \end{cases}$$

$$C_2(s,t,\ell,m) = \begin{cases} C_{all}(s,t,\ell,m) & m \ge m_{all}^{s,t,\ell} \text{ and } s = t \\ \infty & m < m_{all}^{s,t,\ell} \text{ or } s \neq t \end{cases}$$

where

$$C_{ck}(s,t,r,s',t',\ell,m) = \sum_{k=s}^{s'-1} u_f^k + C_{FL}\left(s',t',\ell,m-\omega_a^{s'-1}-\omega_a^{r-1}+\omega_a^{s-1}\right) \tag{3}$$

$$+ C_{FL}\left(r,t,t'-1,m-\omega_a^{r-1}+\omega_a^{s-1}\right) \tag{4}$$

$$C_{all}(s,t,\ell,m) = u_f^s + C_{FL}\left(s+1,s+1,\ell,m-\omega_{\bar{a}}^s\right) + u_b^s \tag{5}$$

We can interpret these values as follows: $C_{ck}(s,t,r,s',t',\ell,m)$ denotes the makespan for the chain from s with backward steps from B^ℓ to B^t if (i) forward operations from s to $s'-1$ are processed with $F_\emptyset$ except F^r, and (ii) during this forward computations the materialized activation moves from the position s to r, i.e. a^{s-1} is deleted and replaced by a^{r-1} when running F_{ck}^r. $C_{all}(s,t,\ell,m)$ denotes the makespan for the chain from s to ℓ if F^s is processed with F_{all}^s.

Proof. Let us start by proving that Eq. (1) is a valid initialization of the dynamic programming. Indeed, in order to back-propagate one layer, F_{all}^s must be executed to be able to process B^s afterwards. This requires a memory size of $m_{all}^{s,s,s}$. As stated in the definition of $C_{\text{FL}}(s,t,\ell,m)$, we assume that a^{s-1} is not included in memory m, and $m_{all}^{s,s,s}$ represents the highest value of the peak memory usage between forward and backward operations corresponding to layer s.

Let us now provide the proof for the general case. Let us assume that $C_{\text{FL}}(s',t',\ell',m)$ corresponds to the optimal makespan when executing the chain starting at position s' and ending at ℓ' to perform the backward steps from B^ℓ till $B^{s'}$ for any s',t',ℓ' where $s' \ge s$, $\ell' \le \ell$ and $t' \ge s'$, except for the case $\{s'=s,t'=t,\ell'=\ell,m\}$.

Let us prove that in this case, $C_{\text{FL}}(s,t,\ell,m)$ provides the optimal solution. Having only a^{s-1} and δ^ℓ stored in the memory with a^{s-1} not included in m, there are three different possible operations to start the execution: F_{all}^s, F_{ck}^s or $F_\emptyset^s$.

If the first operation is F_{all}^s then $\bar{a}^s$ will also be materialized by definition. As memory persistence holds for $\bar{a}^s$ (see Lemma 4.6), then t should be equal to s, otherwise according to the definition of $C_{\text{FL}}(s,t,\ell,m)$, starting from F_{all}^s makes the sequence invalid. Moreover, due to memory persistence, no other value a^k or $\bar{a}^k$ for $0 \le k \le s-1$ is needed until B^{s+1}, so that computing δ^s can be done in makespan $C_{\text{FL}}\left(s+1,s+1,\ell,m-\omega_{\bar{a}}^s\right)$, where the decrease in memory corresponds to the memory needed to store $\bar{a}^s$. After the completion of this chain, it is possible to perform the last backward step B^s as both $\bar{a}^s$ and a^{s-1} are materialized. Provided that the memory limits are satisfied, we obtain the equation for $C_{all}(s,t,\ell,m)$.

If the first operation is F_{ck}^s, then let us denote by $a^{s'-1}$ the first materialized activation after a^{s-1} (since some F_{all} operation needs to be performed before the first backward, $a^{s'-1}$ necessarily exists). Due to Lemma 4.4 and Corollary 4.5, while the floating materialized activation that first appears at position $s'-1$ is present in memory, there is no need to consider any a^k or $\bar{a}^k$ for $s \le k < s'-1$. Thus, computing $\delta^{t'-1}$ from $a^{s'-1}$ can be done in time $C_{\text{FL}}\left(s',t',\ell,m-\omega_a^{s'-1}\right)$, where $s' \le t'$ (floating materialized activations can only move to the right). Indeed, let us assume that

$a^{s'-1}$ is to be materialized in memory, but count its memory usage outside of the limit $m - \omega_a^{s'-1}$. Once this chain is processed, the remaining part represents another chain which starts at position s and ends at position $t' - 1$, where the new currently stored gradient is $\delta^{t'-1}$, so that floating materialized activation $a^{s'-1}$ is not needed anymore and is finally removed. Putting all this together, we get the equation for $C_{ck}(s, t, s, s', t', \ell, m)$.

Finally, if the first operation is $F_\emptyset^s$ then the materialized activation a^{s-1} will move to position $r - 1$ and according to Corollary 4.5, it must satisfy $\omega_a^{r-1} > \omega_a^{s-1}$. Then, after the materialized value has moved to the next position, we use a similar argument as in the previous case, where the memory usage is adjusted by subtracting $\omega_a^{r-1} - \omega_a^{s-1}$ from m, which leads to the equation for $C_{ck}(s, t, r, s', t', \ell, m)$. In the end, combining the last two cases under the condition that memory constraint is fulfilled brings us to $C_1(s, t, \ell, m)$.

At last, let us prove the validity of the memory limits $m_\emptyset^{s,t,\ell}$ and $m_{all}^{s,t,\ell}$. The first one states that processing the chain from s to ℓ with δ^ℓ stored in memory requires at least enough memory to execute all forward steps without materializing any activation. The second one states that processing the chain from s to ℓ starting with F_{all}^s requires enough memory to perform this operation with δ^ℓ stored, and enough memory to perform the corresponding backward operation. □

Theorem 4.7 establishes the correctness of Algorithm 1 and Algorithm 2 to compute an optimal sequence for any set of input parameters. Indeed, the makespan of the returned sequence is $C_{\text{FL}}(1, 1, L + 1, M)$.

Algorithm 1 Computation of an optimal sequence for a chain of length L with memory M.

```
Initialize table C of size (L+1) × (L+1) × (L+1) × M
for 1 ≤ s ≤ L+1 and 1 ≤ m ≤ M do
    Initialize C[s, s, s, m] with Equation (1)
end for
for m = 1, ..., M do
    for d = 1, ..., L do
        for s = 1, ..., L+1−d do
            ℓ = s + d
            for t = s, ..., ℓ do
                Compute C[s, t, ℓ, m] with Equation (2)
            end for
        end for
    end for
end for
return OptRecFL(C, 1, 1, L+1, M − ω_a^0)                    ▷ Alg. 4
```

The output of this dynamic programming is a table of size $O(M \times L^3)$, and computing any element of this table takes $O(L^3)$ time. Thus, the worst case complexity to obtain $C_{\text{FL}}(1, 1, L + 1, M)$ is $O(ML^6)$. Since the number of bits required to store the input values of Problem Remat(L, M) is $O(L \log M)$, the dependency on M means that Algorithm 1 is a pseudo-polynomial algorithm, which is the best possible result given the NP-completeness result of Section 3.4. A common approach for such dynamic programs is to scale and round up memory values, ensuring that the error is at most $1 + \frac{\epsilon}{L}$ for a given $\epsilon > 0$. This yields an algorithm which produces a sequence using $(1+\epsilon)M$ memory with $O(\frac{1}{\epsilon}L^7)$ time complexity. As explained in Section 5.2, we use in practice a fixed number of bins $S = 500$ to describe the memory values.

Algorithm 2 OptRecFL(C, s, t, ℓ, m) – Obtain optimal sequence from the table C

if $C[s,t,\ell,m] = \infty$ **then**
  return Infeasible
else if $s = t = \ell$ **then**
  return (F^s_{all}, B^s)
else if $C[s,t,\ell,m] = C_{ck}(s,t,s,s',t',\ell m)$ **then**
  $\mathcal{S} \leftarrow (F^s_{ck}, F^{s+1}_{\emptyset}, \ldots, F^{s'-1}_{\emptyset})$
  $\mathcal{S} \leftarrow (\mathcal{S}, \text{OptRecFL}(C, s', t', \ell, m - \omega_a^{s'-1}))$
  return $(\mathcal{S}, \text{OptRecFL}(C, s, t, t'-1, m))$
else if $C[s,t,\ell,m] = C_{ck}(s,t,r,s',t',\ell m)$ **then**
  $\mathcal{S} \leftarrow (F^s_{\emptyset}, F^{s+1}_{\emptyset}, \ldots, F^{r-1}_{\emptyset}, F^r_{ck}, F^{r+1}_{\emptyset}, \ldots F^{s'-1}_{\emptyset})$
  return $(\mathcal{S}, \text{OptRecFL}(C, s', t', \ell, m - \omega_a^{s'-1} - \omega_a^{r-1} + \omega_a^{s-1}), \text{OptRecFL}(C, r, t, t'-1, m))$
else
  return $(F^s_{all}, \text{OptRecFL}(C, s+1, s+1, \ell, m - \omega^s_{\bar{a}}), B^s)$
end if

Despite this scaling, for very deep networks (large L values), this method can still take significant time (see Figure 5). For a fixed network, this optimization has to be performed only once for the whole training process, so that this time can be amortized.

As already mentioned, for many cases the optimal solution is actually a memory persistent solution. Indeed, a direct consequence of Corollary 4.5 is the following: if activation sizes are non-increasing with depth (*i.e.* $\omega_a^l \geq \omega_a^{l'}$ for $l' \geq l$), then there exists an optimal memory persistent solution. The networks used in Section 5.5 do not satisfy this non-increasing property, but we observed experimentally that the optimal solutions for these networks are memory persistent. In addition, it is common for users to perform rapid testing of several variants of a network, and in that case it is worthwhile to provide a faster and less precise optimization method. For these reasons, we introduce an approximate algorithm in Section 4.4. It looks for the optimal solution among memory persistent valid sequences, which turns out to be a good candidate to be used in practice with much smaller complexity.

4.4 Dynamic Programming under Memory Persistence Hypothesis

Let us now relax the problem: instead of looking for general solutions, we want to find the best solution among memory persistent sequences, as introduced in Definition 4.1. Similarly to the general case, we propose a dynamic programming solution for this problem, which extends the result of [Griewank and Walther 2000] to the case of heterogeneous memory costs. Furthermore, our solution is adapted to the neural network data dependencies shown in Figure 1a.

Let us prove that it is possible to compute the optimal memory persistent re-materialization sequence in the fully heterogeneous case corresponding to the model depicted in Figure 1a. For a chain of length L, let us denote by $C_{\text{MP}}(s,t,m)$ the optimal execution time to perform backward steps from B^ℓ till B^t of the chain starting from position s, where (i) input tensors a^{s-1} and δ^t are the only materialized values before the processing of the chain and (iii) the peak memory to process the chain is at most m, without taking into account the memory occupied by a^{s-1}.

The memory limits are modified in the following way:

$$m^{s,t}_{\emptyset} = \max \begin{cases} \omega^t_{\delta} + \omega^s_a + o^s_f, \\ \omega^t_{\delta} + \max\limits_{s+1 \leq j < t} \left\{ \omega^{j-1}_a + \omega^j_a + o^j_f \right\} \end{cases} \qquad m^{s,t}_{all} = \max \begin{cases} \omega^t_{\delta} + \omega^s_{\bar{a}} + o^s_f, \\ \omega^s_{\delta} + \omega^{s-1}_{\delta} + \omega^s_{\bar{a}} + o^s_b \end{cases}$$

$m^{s,t}_{\varnothing}$ for $1 \leq s < t \leq L+1$ denotes the memory peak to compute all $F_{\varnothing}$ steps from s to t, and $m^{s,t}_{all}$ for $1 \leq s \leq t \leq L+1$ denotes the memory peak to run F^s_{all} and B^s.

Theorem 4.8. *$C_{MP}(s,t,m)$, the optimal time for any valid memory persistent sequence to process the chain from stage s to stage $t \geq s$ with available memory m, is given by:*

$$C_{MP}(s,s,m) = \begin{cases} u^s_f + u^s_b & m \geq m^{s,s}_{all} \\ \infty & m < m^{s,s}_{all} \end{cases} \tag{6}$$

$$C_{MP}(s,t,m) = \min\left(C_1(s,t,m), \quad C_2(s,t,m)\right) \tag{7}$$

$$C_1(s,t,m) = \begin{cases} \min\limits_{s'=s+1\ldots t} C_{ck}(s,s',t,m) & m \geq m^{s,t}_{\varnothing} \\ \infty & m < m^{s,t}_{\varnothing} \end{cases}$$

$$C_2(s,t,m) = \begin{cases} C_{all}(s,t,m) & m \geq m^{s,t}_{all} \\ \infty & m < m^{s,t}_{all} \end{cases}, \text{ where}$$

$$C_{ck}(s,s',t,m) = \sum_{k=s}^{s'-1} u^k_f + C_{MP}\left(s',t,m-\omega_a^{s'-1}\right) + C_{MP}(s,s'-1,m)$$

$$C_{all}(s,t,m) = u^s_f + C_{MP}\left(s+1,t,m-\omega^s_{\bar{a}}\right) + u^s_b$$

We can interpret these values as follows: $C_{ck}(s,s',t,m)$ denotes the computing time for the chain from s to t if forward operations from s to $s'-1$ are processed with $F_{\varnothing}$, whereas a^{s-1} is materialized in memory by F^s_{ck}. $C_{all}(s,t,m)$ is the computing time for the chain from s to t if F^s is processed with F^s_{all}.

The proof is based on the same arguments as provided in Theorem 4.7 with the difference that once materialized, an activation is not deleted until after the corresponding backward step, so that this materialized value naturally divides the chain into two subchains, that can be processed one after the other. Therefore, C_{ck} is now defined with only four arguments: s which determines the start of the chain, t for the end of the chain, s' for the separation point between two subchains (the leftmost materialized value) and m, the memory limit.

Proof. Let us start by proving that Eq. (6) is a valid initialization of the dynamic programming. Indeed, in order to back-propagate one layer, F^s_{all} must be performed to execute B^s afterwards. This requires a memory of size $m^{s,s}_{all}$: we consider that a^{s-1} is not included in m, as stated in the definition of $C_{\text{MP}}(s,t,m)$, and $m^{s,s}_{all}$ represents the highest value of the peak memory usage between forward and backward operations corresponding to layer s.

Let us now prove the general case. Since we are looking for a persistent sequence, and the input a^{s-1} is materialized in memory, the optimal sequence has only two possible ways to start: either with F^s_{ck} to materialize a^{s-1} and compute a^s, or with F^s_{all} to compute $\bar{a}^s$ and save a^{s-1} in the memory at the same time (see Table 1).

If the first operation is F^s_{ck}, then we can denote by $a^{s'-1}$ the first value materialized in memory after a^{s-1} (since some F_{all} operation has to be performed before the first backward, $a^{s'-1}$ necessarily exists). Due to memory persistence, while $a^{s'-1}$ is present in memory, there is no need to consider any a^k or $\bar{a}^k$ for $s \leq k < s'-1$, so computing $\delta^{s'-1}$ from input $a^{s'-1}$ can be done in time $C_{\text{MP}}\left(s',t,m-\omega_a^{s'-1}\right)$. Indeed, we assume that $a^{s'-1}$ is materialized in memory, but we count its memory usage outside the limit: $m-\omega_a^{s'-1}$. Once this chain is processed, the remaining part

represents another chain which starts at position s and finishes at $s' - 1$, where the new currently stored gradient is $\delta^{s'-1}$ and $a^{s'-1}$ is not needed anymore and is finally removed. Bringing everything together yields the equation for $C_{ck}(s, s', t, m)$. Choosing s' such that $C_{ck}(s, s', t, m)$ is minimal guarantees the smallest possible cost, which is reflected in $C_1(s, t, m)$.

If the first operation is F^s_{all} then value $\bar{a}^s$ will be materialized together with a^{s-1} by definition. As memory persistence holds and no other value a^k or $\bar{a}^k$ for $0 \leq k \leq s-1$ is needed until B^{s+1}, computing δ^s can be done in time $C_{\text{MP}}\left(s+1, t, m - \omega^s_{\bar{a}}\right)$, where the decrease in memory corresponds to the memory needed to materialize $\bar{a}^s$. After the completion of this chain, it is possible to perform the last backward B^s as both $\bar{a}^s$ and a^{s-1} are already materialized. Provided that the memory constraint is fulfilled, we get the equation for $C_{all}(s, t, m)$.

At last, let us prove that the memory limits $m^{s,t}_{\emptyset}$ and $m^{s,t}_{all}$ are valid. The first one states that executing the chain from s to t with δ^t stored in memory requires at least enough memory to execute all forward steps without saving any activation. The second one states that executing the chain from s to t by starting with F^s_{all} requires enough memory to perform this operation with δ^t stored, and enough memory to perform the corresponding backward operation. □

This theorem proves that Algorithm 3 and Algorithm 4 compute an optimal persistent sequence, for all input parameters. Indeed, the makespan of the returned sequence is exactly $C_{\text{MP}}(1, L+1, M)$. The worst case complexity of Algorithm 3 is $O(ML^3)$, saving a factor of L^3 over Algorithm 1.

Algorithm 3 Optimal persistent sequence for a chain of length L with memory M.

```
Initialize table C of size (L + 1) × (L + 1) × M
for 1 ≤ s ≤ L + 1 and 1 ≤ m ≤ M do
    Initialize C[s, s, m] with Equation (6)
end for
for m = 1, . . . , M do
    for d = 1, . . . , L do
        for s = 1, . . . , L + 1 − d do
            Compute C[s, s + d, m] with Equation (7)
        end for
    end for
end for
return OptRecP(C, 1, L + 1, M − ω^0_a)                    ▷ Alg. 4
```

Algorithm 4 OptRecP(C, s, t, m) – Computation of the optimal persistent sequence from table C

```
if C[s, t, m] = ∞ then
    return Infeasible
else if s = t then
    return (F^s_all, B^s)
else if C[s, t, m] = C_ck(s, s', t, m) then
    return (F^s_ck, F^{s+1}_∅, . . . , F^{s'−1}_∅, OptRecP(C, s', t, m − ω^{s'−1}_a), OptRecP(C, s, s' − 1, m))
else
    return (F^s_all, OptRecP(C, s + 1, t, m − ω^s_ā), B^s)
end if
```

5 IMPLEMENTATION AND VALIDATION

We demonstrate the applicability of our approach by presenting Rotor [HiePACS team 2019], a tool that allows the above algorithms to be used with any Pytorch DNN. The only requirement is that the DNN should be based on the `nn.Sequential` container, which is necessary to ensure that the network has a sequential structure, but allows each block to be arbitrarily complex. Rotor is used in a very similar fashion to the existing `checkpoint_sequential` tool already available in PyTorch [Pytorch contributors 2018]. Nevertheless, `checkpoint_sequential` restricts the optimal solution search to periodic solutions, in which the chain of length L is cut into $\sqrt{L}$ chunks of size $\sqrt{L}$, while Rotor computes more general solutions, which allows us to significantly improve the performance, as we will show in Section 5.5.

From a PyTorch DNN `model` based on the `nn.Sequential` container, using Rotor only requires the user to wrap the model in the `rotor.Checkpointable` class by providing a memory limit (expressed in bytes):

```
import rotor
model = rotor.Checkpointable(model, mem_limit=10*(1024**3))
```

Rotor works in three phases: parameter estimation, optimal sequence computation and sequence processing. The first two phases are performed only once, automatically before the start of the training, while the sequence is used at each iteration.

5.1 Parameter Estimation

In the parameter estimation phase, the goal is to measure the quantities (time and memory sizes) associated with the input DNN. These measures are used to instantiate the parameters of the model described in Figure 1a and used as input values for both Algorithm 1 and Algorithm 3:

- memory sizes $\omega_a^\ell, \omega_{\bar{a}}^\ell, \omega_\delta^\ell$,
- memory overheads o_f^ℓ, o_b^ℓ, and
- execution time of every operation in the sequence u_f^ℓ, u_b^ℓ.

Parameter estimation is done in the following way: given a chain and a sample input data, denoted as a^0_{sample}, forward and backward operations of each stage are processed one after the other. Using a^ℓ_{sample} as input, the forward F^ℓ_{all} is processed to obtain $\bar{a}^{\ell+1}_{\text{sample}}$, and the backward B^ℓ with an arbitrary value $\delta^{\ell+1}$. The execution time of each F and B operation is measured three times, and the median value is kept. This has proven sufficient to avoid any outlier. The memory management interface of PyTorch (which records the memory used by all created tensors) is used to obtain the memory usage of $\bar{a}^{\ell+1}_{\text{sample}}$ and the peak memory usage of both forward and backward operations.

This parameter estimation assumes that the computations performed by the neural network do not depend on the input data (a very similar assumption is made for the `jit.trace()` function of PyTorch), so that the measurement on a sample input is a representative of the actual execution on the training data.

5.2 Computation of the Optimal Sequence

Once all measurements have been performed, the optimal weakly and strongly persistent sequences can be computed using respectively Algorithm 1 and Algorithm 3 for any given memory limit M. In order to limit the computational cost of this phase, we limit the set of possible values for memory occupations. In particular, we divide the memory of size M into S bins (memory slots), each of size $\frac{M}{S}$. Then we scale and round down all other values so that they fit into one of those memory slots. $S = 500$ is a reasonable value that we used for all experiments in this paper. Thus, the complexity

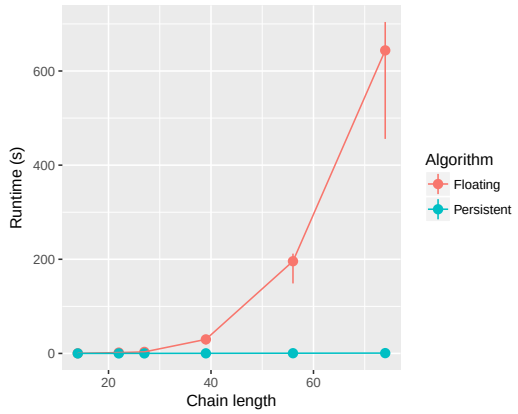


Fig. 5. Running times of Algorithm 1 and Algorithm 3 for different network sizes.

of the resulting algorithm is independent of the actual memory limit, at the cost of at most $1 + \frac{1}{S}$ overestimation of memory sizes.

5.3 Comparison between Algorithm 1 and Algorithm 3

We provide a sequential implementation in the C language of the dynamic programming algorithms (Algorithm 1 and Algorithm 3). These algorithms are executed on one core of the CPU of the computing node. In our experiments, the computing nodes contain 40 Intel Xeon Gold 6148 cores at 2.4GHz. The running time of Algorithm 3 on most of the networks in our experiments is below 1 second. The longest execution time was obtained with the ResNet 1001 network [He et al. 2016], which results in a chain of length 339, and an execution time below 20 seconds. Since this computation is performed only once for the whole training phase, such an execution time is completely acceptable.

For each considered network and for each set of parameters, we computed the optimal sequences produced by Algorithm 1 and Algorithm 3. Although we prove in Section 4.1 that the ratio between the makespan produced by both algorithms can be as high as $\frac{3}{2}$, in practice the computed solutions for all test cases turned out to be exactly the same. We provide in Figure 5 a comparison of the running times of both algorithms as a function of the network length, for several networks and memory limits. The vertical bars on the figure indicate the minimum and maximum running times encountered for different situations for a given network length. This figure shows that the running time of Algorithm 1 is significantly higher than the one of Algorithm 3. Hence, it is a reasonable strategy to use Algorithm 3 to compute the optimal solution.

5.4 Experimental Setting

All experiments presented in this paper have been performed with Python 3.5.9 and PyTorch 1.3.0. The computing node contains an Nvidia Tesla V100-PCIE GPU card with 15.75GB of memory, in addition to the aforementioned 40 Intel Xeon Gold 6148 cores at 2.4GHz. All computations related to layers of the DNNs take place on the GPU card, the CPU is only used as a controller. The experiments are performed with three different kinds of networks, whose implementations are available in the `torchvision` package of PyTorch: ResNet, DenseNet, and Inception v3. All three types of networks have been slightly adapted to be compatible with Rotor [HiePACS team 2019], by using the `nn.Sequential` module where applicable. All available depths are used for ResNet. In

particular, depths 18, 34, 50, 101, 152 are available in `torchvision`, whereas versions with depths 200 and 1001 are available in the previous work [He et al. 2016]. Similarly, for DenseNet, depths 121, 161, 169 and 201 are used.

Three different image sizes are used: small images of shape 224×224 (which is the default and minimal image size for all models of `torchvision`), medium images of shape 500×500, and large images of shape 1000×1000. For each model and image size, we consider different batch sizes that are powers of 2, starting from the smallest batch size that ensures a reasonable throughput. With small batch sizes, we observe that doubling the batch size effectively doubles the throughput, which shows that the GPU is not used efficiently in the former case.

Five strategies to perform a training iteration on those models are compared.

- The **PyTorch** strategy consists in the standard way of computing the forward and backward operations, where all intermediate activations are materialized. On the one hand, since PyTorch never performs recomputations, its running time provides an upper bound on the achievable throughput. On the other hand, PyTorch can only run if enough memory is available to store all activations. Therefore, there is no solution for the PyTorch strategy in some experiments (batch size 8 in Figure 6, all plots in Figure 7, Inception 200 in Figure 8).
- The **sequential** strategy relies on the `checkpoint_sequential` tool of PyTorch [Pytorch contributors 2018]. This strategy splits the chain into s segments and materializes activations at the beginning of each segment during the forward phase only. Thus, every forward computation is performed twice, except those of the last segment. 10 different numbers of segments are used, from 2 (always included) to $2\sqrt{L}$, where L is the length of the chain ($\sqrt{L}$ is the optimal number of segments for this strategy when the chain is homogeneous). The same strategy is used in [Goyal 2018], but the number of segments has to be hand-tuned.
- The **revolve** strategy uses the optimal algorithm adapted to heterogeneous chains of the Automatic Differentiation model [Griewank and Walther 2008], and converts it to a valid solution by saving only activations a to memory, and performing an F_{all} step before each backward step to enforce validity. This is the same strategy as advocated in Appendix C of [Gruslys et al. 2016].
- The **optimal** strategy corresponds to Rotor [HiePACS team 2019] and uses Algorithm 3 (or equivalently Algorithm 1 since they provide the same results for all experiments) for 10 different memory limits, equally spaced between 0 and the memory usage of the PyTorch strategy.
- The **Checkmate** strategy is a reimplementation in our PyTorch framework of the linear programming approach presented in [Jain et al. 2019]. Since we are interested in significantly large networks ($L > 100$), we use the approximate version, based on two-phase rounding of the best fractional solution. This approach does not differentiate between a and $\bar{a}$ values; so for validity we have included a conversion step which uses F_{all} operations when necessary. The linear programs are solved with CPLEX version 12.10 with a one hour time limit. Due to the high computational cost, this strategy is only shown in a selection of plots.

For each model, image size and batch size, we perform enough iterations to ensure that the PyTorch strategy lasts at least 500ms, and the actual peak memory usage and duration over 5 runs are measured. The obtained measurements are very stable, so all plots in the next section present the median duration over 5 runs for each experiment (on average, the difference between the highest and lowest measured throughputs are within 1% of the median). For each run, the memory peak consumption and the throughput of the experiments have been carefully measured, using the same mechanism as the one used to perform the measurement phase.

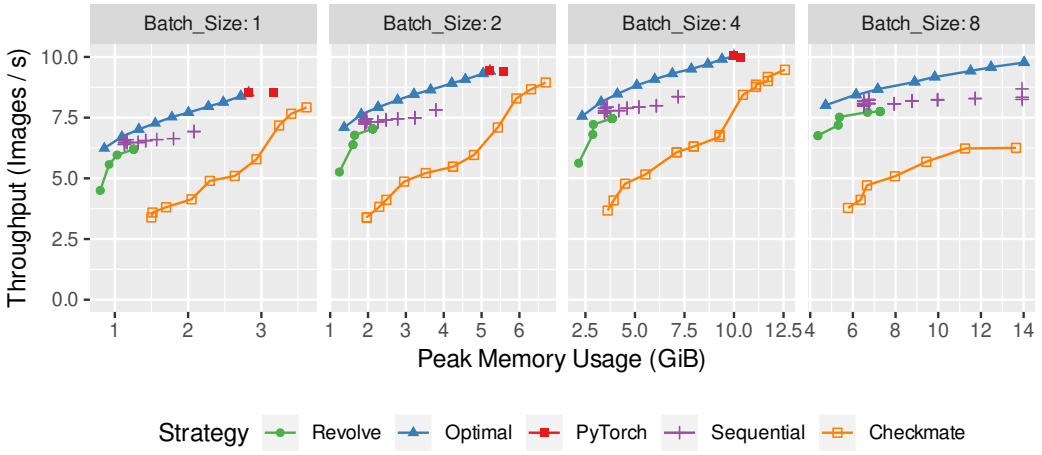


Fig. 6. Experimental results for the ResNet network with depth 101 and image size 1000.

5.5 Experimental Results

All plots corresponding to the above described experiments are available in Appendix A. For the sake of conciseness, we only present here a representative selection of the results; the behavior on other experiments is very similar. All plots have the same structure: for a given set of parameters (network, depth, image size and batch size), we plot for each strategy the achieved throughput (in terms of images per second) against the peak memory usage. The square red dot represents the performance obtained by the standard PyTorch strategy, and its absence from the graph means that a memory overflow error was encountered when attempting to execute it. Purple crosses represent the results obtained with the sequential strategy for different numbers of segments. The blue line with triangles shows the result obtained with our optimal strategy. The green line with circles shows the result obtained with the revolve algorithm. When available, the orange line with crossed squares shows the performance achieved with our Checkmate implementation. The structure of the Inception network makes it more challenging to adapt to the Checkmate algorithm. Since the performance of Checkmate on the other networks is not satisfactory, we did not perform experiments on the Inception network with Checkmate. We draw lines to emphasize the fact that these strategies can be given any memory limit as input, whereas the result of sequential is inherently tied to a discrete number of segments. We provide a representative selection of results in Figures 6 to 8, and the complete results can be found in Figures 9 to 16 in Appendix A.

Figure 6 depicts the results for ResNet with depth of 101. For a batch size of 1, PyTorch strategy has a memory peak consumption of 2.83 GiB, which is enough to fit on this GPU. However, when the batch size is 8, PyTorch fails to compute the back-propagation due to memory limitations. The sequential strategy offers a discrete alternative by dividing the chain into a given number of segments (in this case from 2 to 11). For every batch size, the best throughput is reached when the number of segments is equal to 2. For instance, when the batch size is 8, the throughput of the sequential strategy with 2 segments is on average 8.67 images/s with a memory peak consumption of 13.91 GiB. The optimal strategy offers a continuous alternative by implementing the best re-materialization strategy for any given memory bound. We can see that for a given memory peak, the optimal strategy outperforms the sequential strategy by up to 15%. For instance, when the batch size is 8, the maximum throughput achieved by the optimal strategy is 9.77 images/s. The revolve

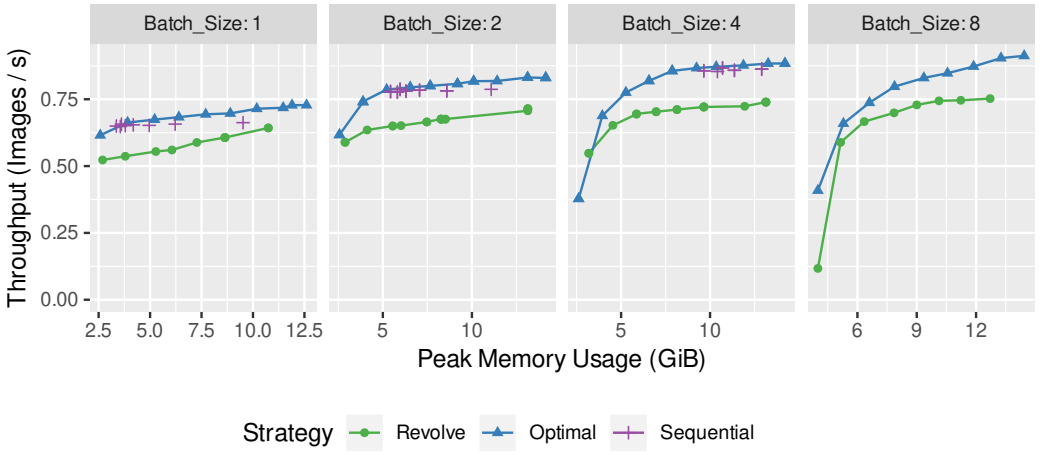


Fig. 7. Experimental results for the ResNet network with depth 1001 and image size 224.

algorithm provides a continuous approach as well. However, it requires that every forward operation is computed at least twice (once in the forward phase, once before the backward operation), which incurs a much lower throughput than both other solutions. Furthermore, since this algorithm does not consider saving the larger $\bar{a}$ values, it is unable to make use of larger memory sizes. Finally, the Checkmate strategy has significant difficulty to optimize these deep networks because of two issues. The first issue is that the resulting linear programs are very large and cannot be solved exactly in reasonable time; thus we use the fractional relaxation proposed in [Jain et al. 2019], which does not correctly estimate the memory usage: the memory used by the solutions of Checkmate is consistently higher than the limit provided to the linear program. The second issue is that Checkmate does not consider the difference between a and $\bar{a}$ values: given a solution produced by the linear program, producing a correct execution sequence for the PyTorch framework requires us to convert the solution using F_{all} operations, which in some cases means inserting additional re-computations, lowering the obtained throughput.

Figure 7 displays the same results for ResNet with depth of 1001. This setup requires much more memory and the PyTorch strategy fails even when the batch size is 1. The sequential strategy requires at least 6 segments for batch size 1, 10 segments for batch size 2, and 18 segments for batch size 4, and cannot perform the back-propagation when the batch size is 8. Not only does the optimal strategy outperform the sequential strategy when it does not fail but it offers a stable solution to train the neural network even with a larger batch size, which allows us to increase the achieved throughput thanks to a better GPU efficiency (0.91 for optimal, whereas the highest throughput achieved by sequential is 0.86). It is interesting to note that based on the parameters estimated by Rotor [HiePACS team 2019], running the setting with batch size 8 with the PyTorch strategy would require 225 GiB of memory, and achieve a throughput of 1.18 images/s. Additional results in Figure 16 also show that Rotor allows us to run this large network even with medium and large image sizes.

All these conclusions hold for every tested neural network and parameters. Figure 8 displays some of them and shows that the behavior of Rotor is stable on various network sizes and image sizes. To summarize, we also compute the ratio between the highest throughput obtained by sequential

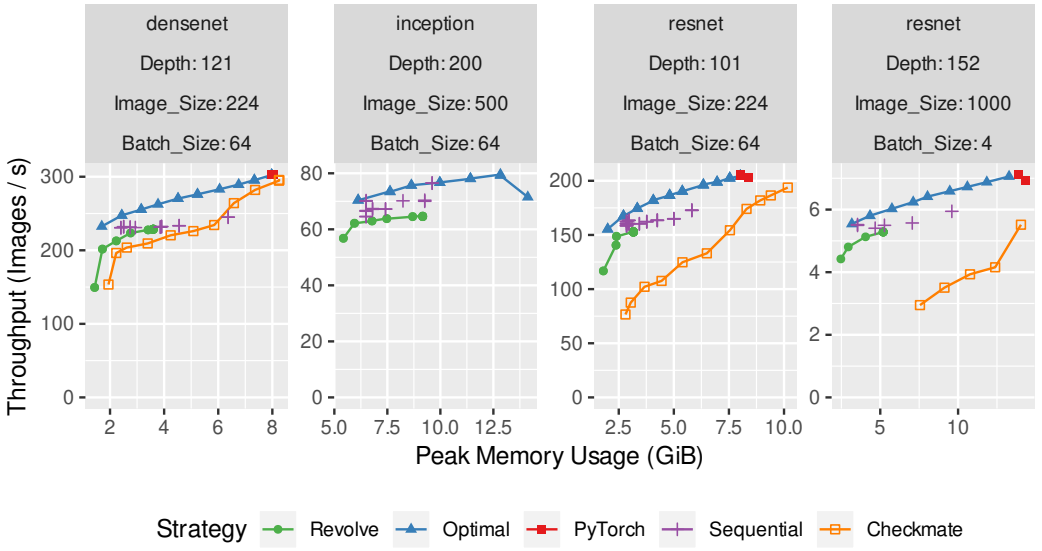


Fig. 8. Experimental results for several situations.

and the throughput achieved by Rotor with the corresponding memory usage. On average over all tested sets of parameters, optimal achieves 12.8% higher throughput than sequential.

6 CONCLUSION

This document describes a new re-materialization strategy that leverages operations available in DNN frameworks with the capabilities of autograd functions. We carefully model back-propagation and we prove that the optimization problem becomes NP-complete in the presence of heterogeneous activation sizes. Furthermore, the memory persistence property used by the Automatic Differentiation community to derive Dynamic Programming optimal solutions holds no longer. Instead, we propose a weak version of memory persistence and prove that it can be used to find optimal re-materialization strategies in the fully heterogeneous case. We also propose a dynamic programming algorithm, which computes the optimal persistent sequence for any sequentialized network. Both algorithms for any sequential PyTorch module are implemented in Rotor [HiePACS team 2019].

On the practical side, we compare achieved results with Rotor against (i) a periodic checkpointing strategy available in PyTorch, (ii) an optimal persistent strategy adapted from the Automatic Differentiation literature to a fully heterogeneous setting and (iii) a re-materialization strategy based on Linear Programming (Checkmate). We show that Rotor consistently outperforms these re-materialization strategies, for a large class of networks, image sizes and batch sizes. The tool Rotor requires a very minor modification of the code, and increases throughput by an average of 12.8% compared to its best competitor sequential. Moreover, Rotor is more flexible as it offers the ability to specify an arbitrary memory limit. Therefore, Rotor allows users to use larger models, larger batches or larger images while adapting to the memory of the training device. In our future work, we want to study the advantages of our approach in combination with other strategies developed to address memory limitations such as model parallelism, activation offloading and domain decomposition.

REFERENCES

Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert. 2016. Optimal Multistage Algorithm for Adjoint Computation. *SIAM Journal on Scientific Computing* 38, 3 (2016), 232–255.

Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. 2019a. Optimal GPU-CPU Offloading Strategies for Deep Neural Network Training. (Oct. 2019). https://hal.inria.fr/hal-02316266 working paper or preprint.

Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. 2021. Efficient Combination of Rematerialization and Offloading for Training DNNs. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 23844–23857. https://proceedings.neurips.cc/paper/2021/file/c8461bf13fca8a2b9912ab2eb1668e4b-Paper.pdf

Olivier Beaumont, Julien Herrmann, Guillaume Pallez, and Alena Shilova. 2019b. *Optimal Memory-aware Backpropagation of Deep Join Networks.* Research Report RR-9273. Inria. https://hal.inria.fr/hal-02131552

Jose Carranza-Rojas, Herve Goeau, Pierre Bonnet, Erick Mata-Montero, and Alexis Joly. 2017. Going deeper in the automated identification of Herbarium specimens. *BMC Evolutionary Biology* 17, 1 (2017), 181.

Bo Chang, Lili Meng, Eldad Haber, Lars Ruthotto, David Begert, and Elliot Holtham. 2018. Reversible architectures for arbitrarily deep residual neural networks. In *Thirty-Second AAAI Conference on Artificial Intelligence.*

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. 2016. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709* (2016).

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems.* 1223–1231.

Jianwei Feng and Dong Huang. 2018. Optimal Gradient Checkpoint Search for Arbitrary Computation Graphs. arXiv:1808.00079 [cs.LG]

Yifan Feng, Zizhao Zhang, Xibin Zhao, Rongrong Ji, and Yue Gao. 2018. GVCNN: Group-view convolutional neural networks for 3D shape recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 264–272.

Michael R Garey and David S Johnson. 1979. *Computers and intractability.* Vol. 174. freeman San Francisco.

Sambit Ghadai, Xian Yeow Lee, Aditya Balu, Soumik Sarkar, and Adarsh Krishnamurthy. 2019. Multi-level 3D CNN for Learning Multi-scale Spatial Features. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops.* 0–0.

Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. 2017. The reversible residual network: Backpropagation without storing activations. In *Advances in neural information processing systems.* 2214–2224.

Priya Goyal. 2018. *PyTorch Memory optimizations via gradient checkpointing.* https://github.com/prigoyal/pytorch_memonger

Andreas Griewank. 1989. On automatic differentiation. *Mathematical Programming: Recent Developments and Applications* 6, 6 (1989), 83–107.

Andreas Griewank and Andrea Walther. 2000. Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)* 26, 1 (2000), 19–45.

Andreas Griewank and Andrea Walther. 2008. *Evaluating derivatives: principles and techniques of algorithmic differentiation.* Vol. 105. Siam.

Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-efficient backpropagation through time. In *Advances in Neural Information Processing Systems.* 4125–4133.

Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. 2018. Can spatiotemporal 3d cnns retrace the history of 2d cnns and imagenet?. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition.* 6546–6555.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. In *Computer Vision – ECCV 2016.* Springer International Publishing, 630–645. https://arxiv.org/tb/1603.05027

HiePACS team. 2019. *Rotor.* https://gitlab.inria.fr/hiepacs/rotor

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems.* 103–112.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research* 18, 1 (2017), 6869–6898.

Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E. Gonzalez. 2019. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. arXiv:1910.02653 [cs.LG]

Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2020. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616* (2020).

Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. 2019. Efficient rematerialization for deep networks. In *Advances in Neural Information Processing Systems*. 15146–15155.

Mitsuru Kusumoto, Takuya Inoue, Gentaro Watanabe, Takuya Akiba, and Masanori Koyama. 2019. A Graph Theoretic Framework of Recomputation Algorithms for Memory-Efficient Backpropagation. *arXiv preprint arXiv:1905.11722* (2019).

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

Johannes Lotz, Uwe Naumann, and Sumit Mitra. 2016a. *Mixed Integer Programming for Call Tree Reversal*. 83–91. https://doi.org/10.1137/1.9781611974690.ch9 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611974690.ch9

Johannes Lotz, Uwe Naumann, and Sumit Mitra. 2016b. Mixed integer programming for call tree reversal. In *2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 83–91.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

Uwe Naumann. 2008. Call Tree Reversal is NP-Complete. In *Advances in Automatic Differentiation*, Christian H. Bischof, H. Martin Bücker, Paul Hovland, Uwe Naumann, and Jean Utke (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 13–22.

Uwe Naumann. 2009. DAG reversal is NP-complete. *Journal of Discrete Algorithms* 7, 4 (2009), 402–410.

Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens van der Maaten, and Kilian Q Weinberger. 2017. Memory-efficient implementation of densenets. *arXiv preprint arXiv:1707.06990* (2017).

Pytorch contributors. 2018. *Periodic Checkpointing in PyTorch*. https://pytorch.org/docs/stable/checkpoint.html

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 18.

Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. 2018. In-place activated batchnorm for memory-optimized training of dnns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5639–5647.

Shriram S B, Anshuj Garg, and Purushottam Kulkarni. 2016. Dynamic Memory Management for GPU-based training of Deep Neural Networks. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Press.

Zheng Shou, Jonathan Chan, Alireza Zareian, Kazuyuki Miyazawa, and Shih-Fu Chang. 2017. Cdc: Convolutional-de-convolutional networks for precise temporal action localization in untrimmed videos. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5734–5743.

Jeffrey Mark Siskind and Barak A. Pearlmutter. 2018. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software* 33, 4-6 (2018), 1288–1330. https://doi.org/10.1080/10556788.2018.1459621 arXiv:https://doi.org/10.1080/10556788.2018.1459621

Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller. 2015. Multi-view convolutional neural networks for 3d shape recognition. In *Proceedings of the IEEE international conference on computer vision*. 945–953.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

Marian Verhelst and Bert Moons. 2017. Embedded deep neural network processing: Algorithmic and processor techniques bring deep learning to iot and edge devices. *IEEE Solid-State Circuits Magazine* 9, 4 (2017), 55–65.

Andrea Walther and Andreas Griewank. 2004. Advantages of binomial checkpointing for memory-reduced adjoint calculations. In *Numerical mathematics and advanced applications*. Springer, 834–843.

Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 6848–6856.

A ADDITIONAL PLOTS

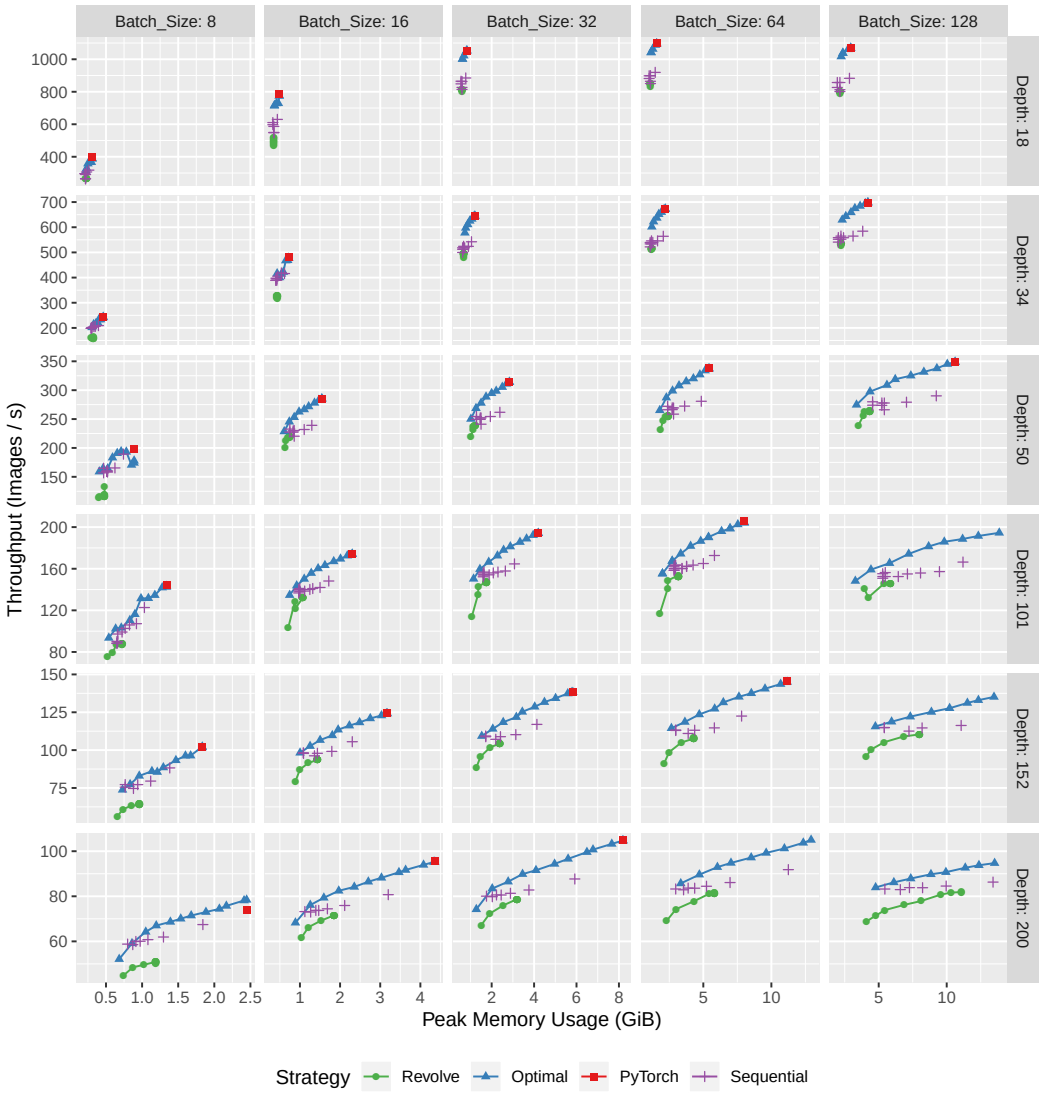


Fig. 9. Results for Resnet with image size 224, for different depths and batch sizes.

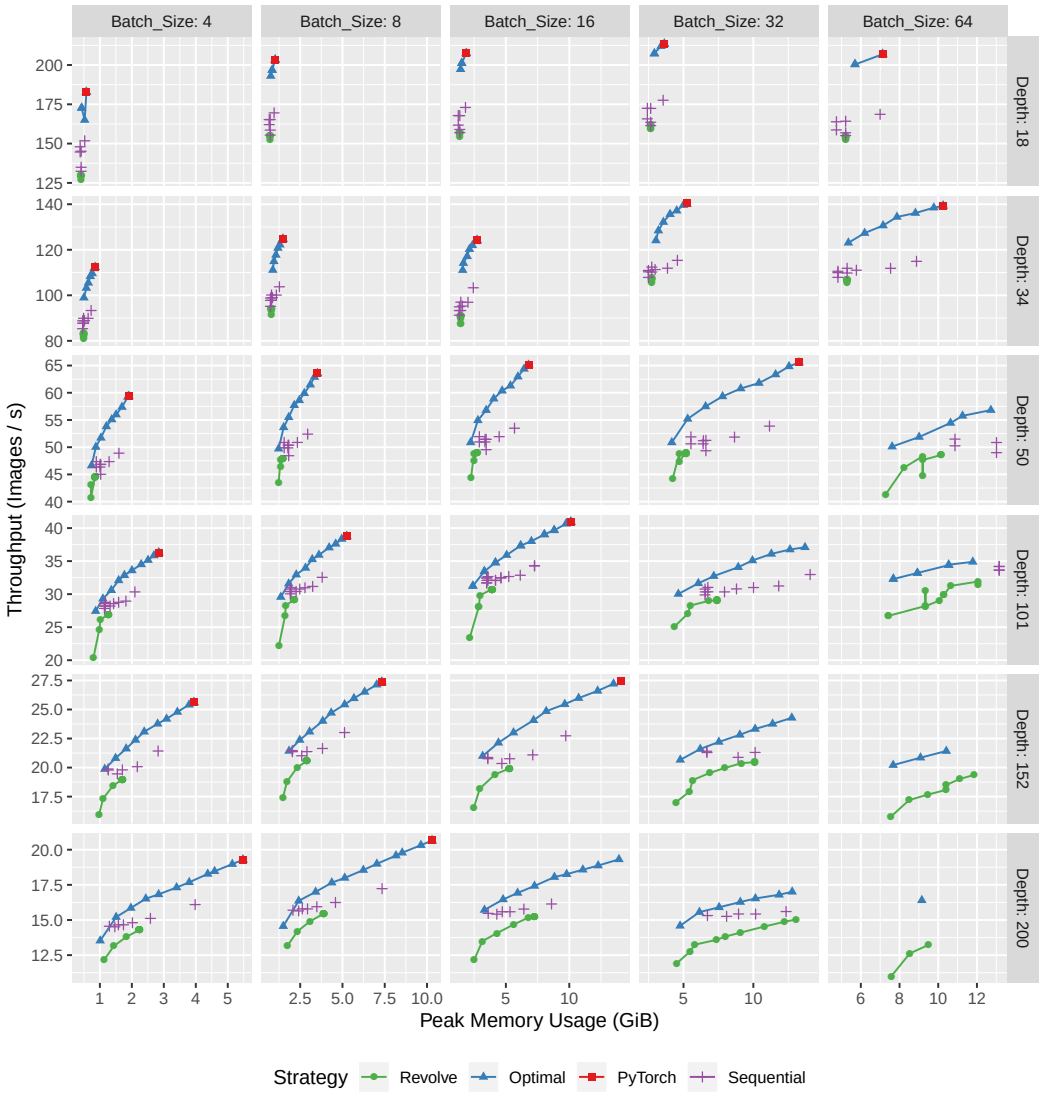


Fig. 10. Results for Resnet with image size 500, for different depths and batch sizes.

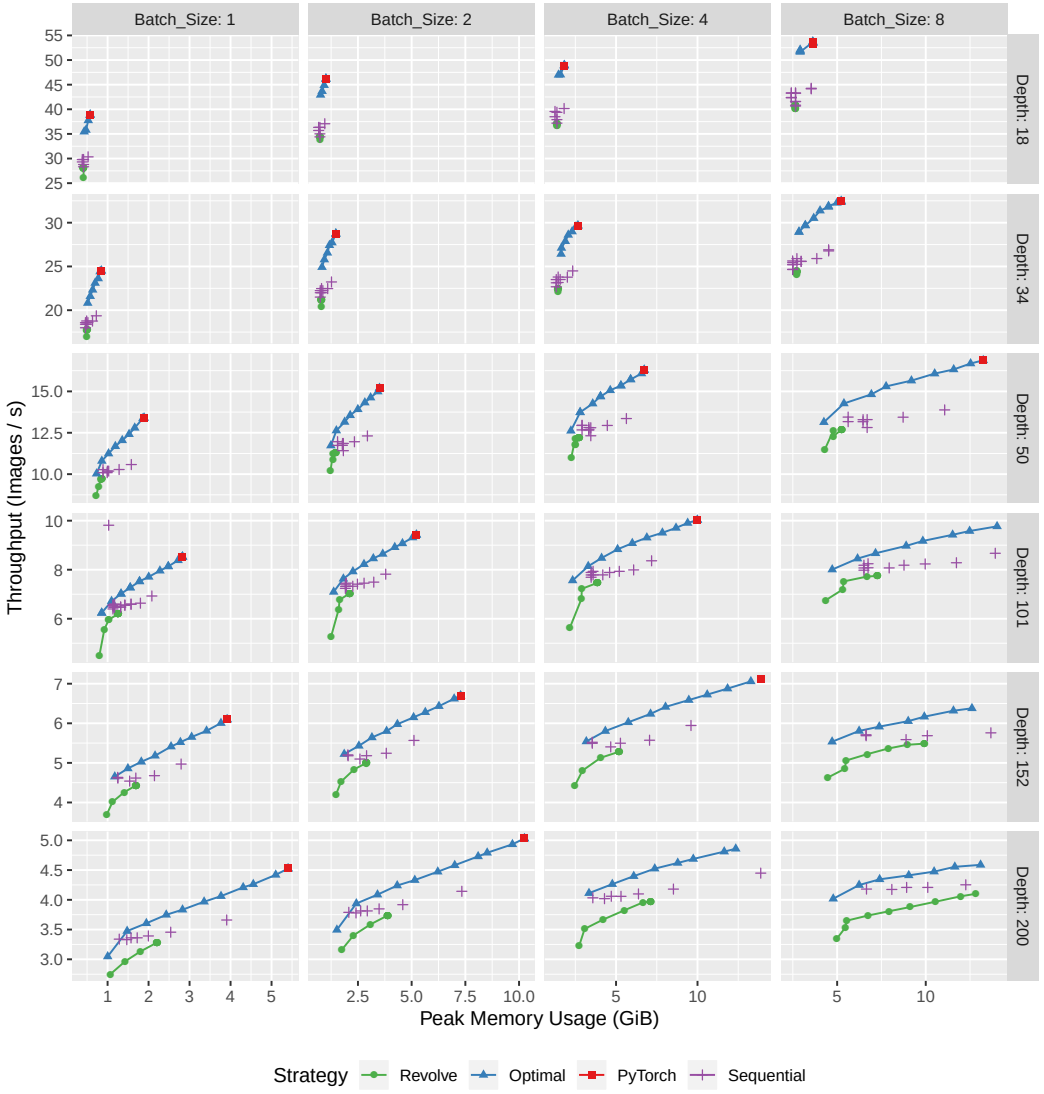


Fig. 11. Results for Resnet with image size 1000, for different depths and batch sizes.

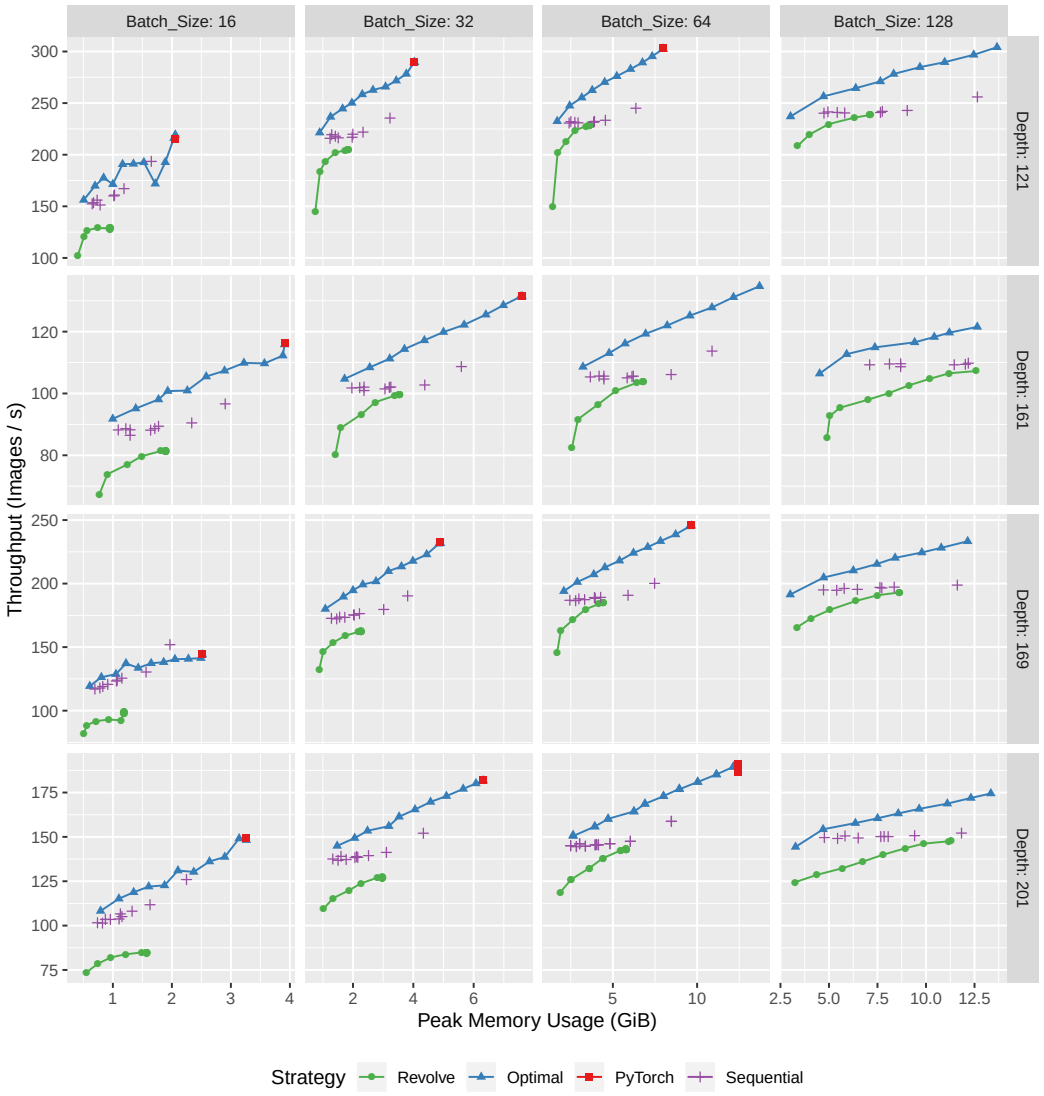


Fig. 12. Results for Densenet with image size 224, for different depths and batch sizes.

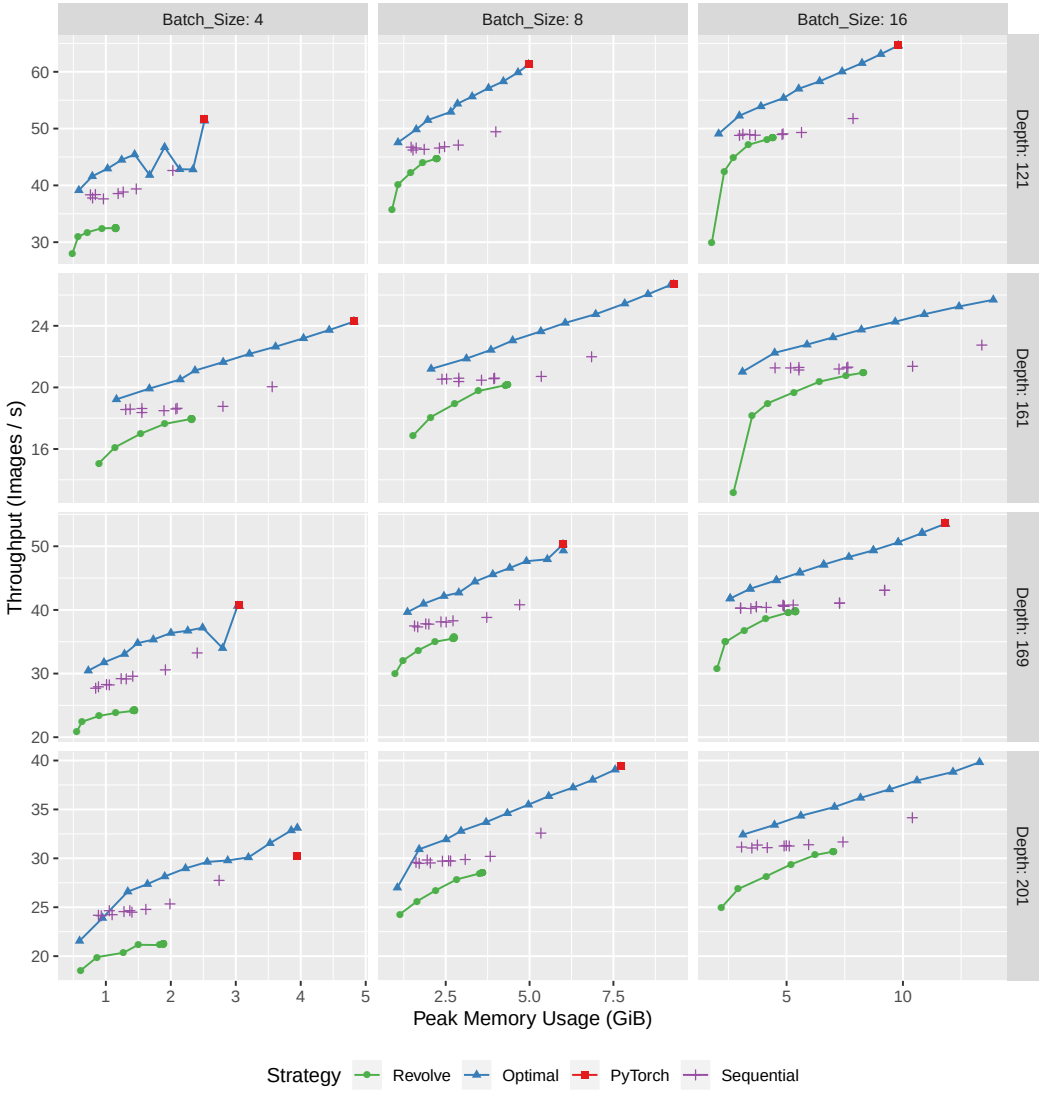


Fig. 13. Results for Densenet with image size 500, for different depths and batch sizes.

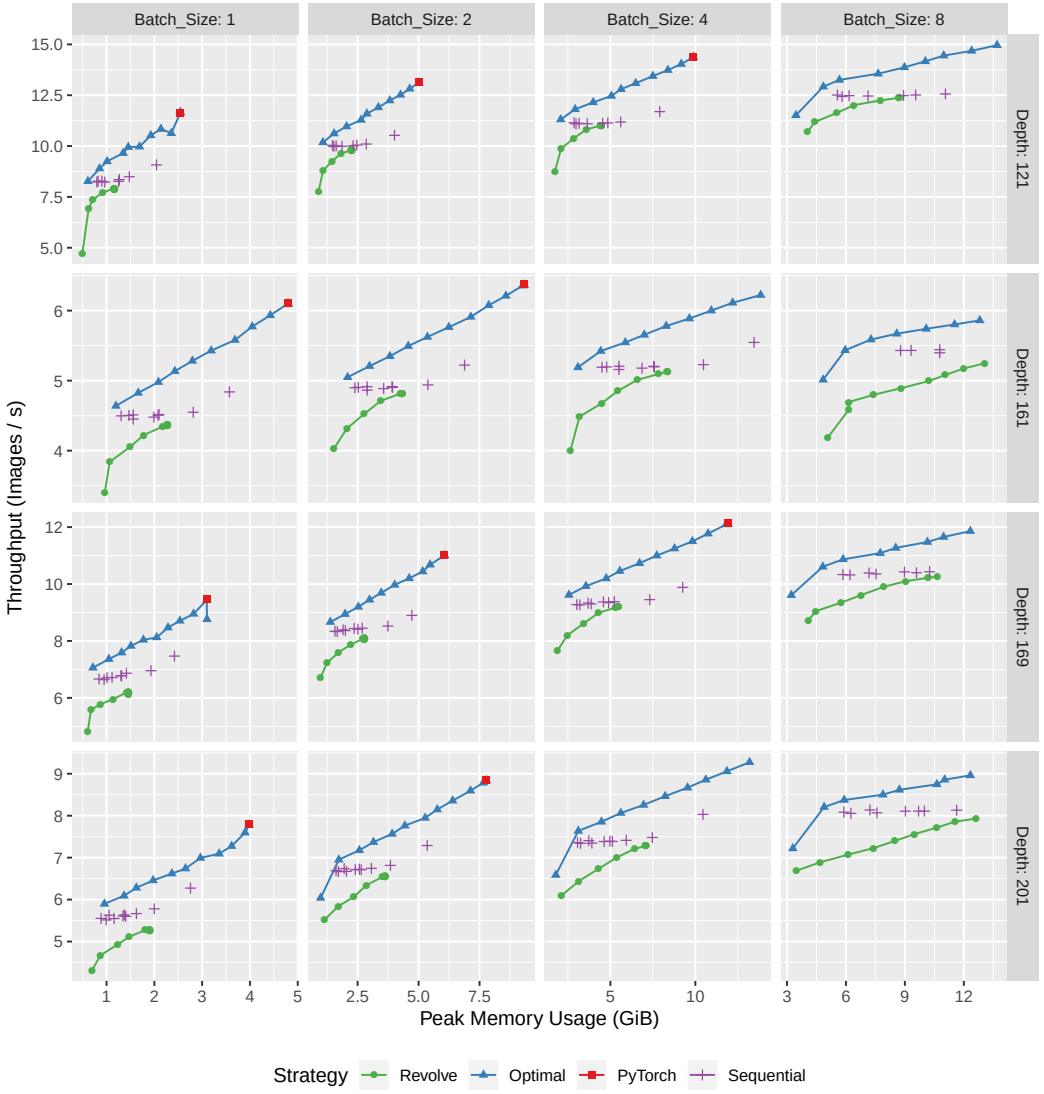


Fig. 14. Results for Densenet with image size 1000, for different depths and batch sizes.

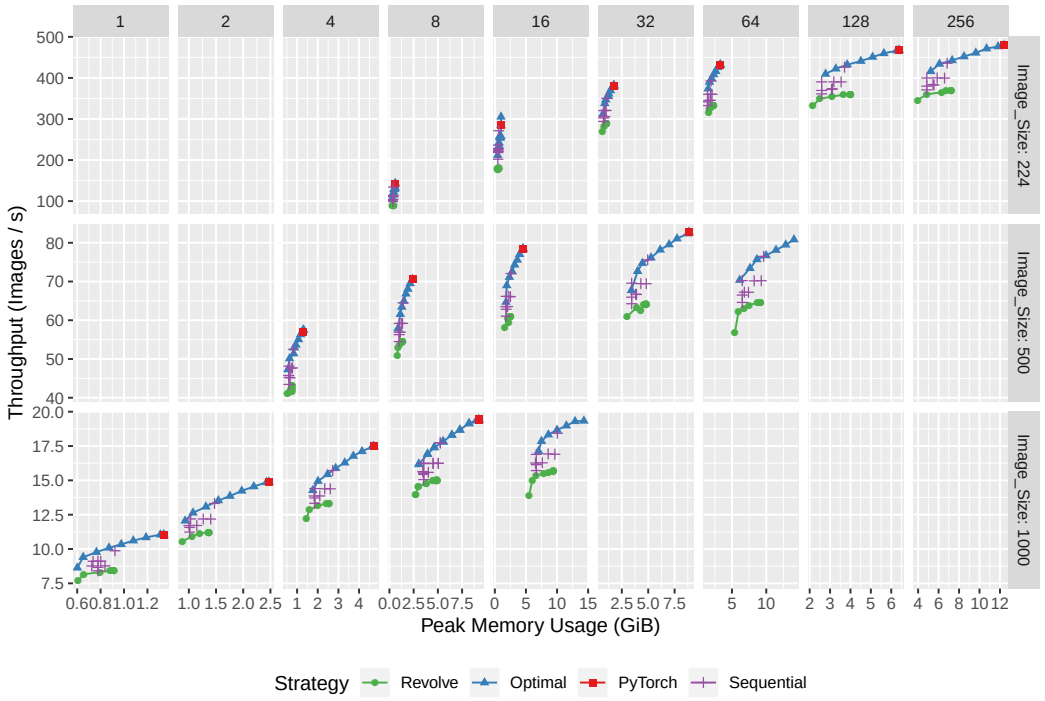


Fig. 15. Results for Inception v3 for different image sizes and batch sizes.

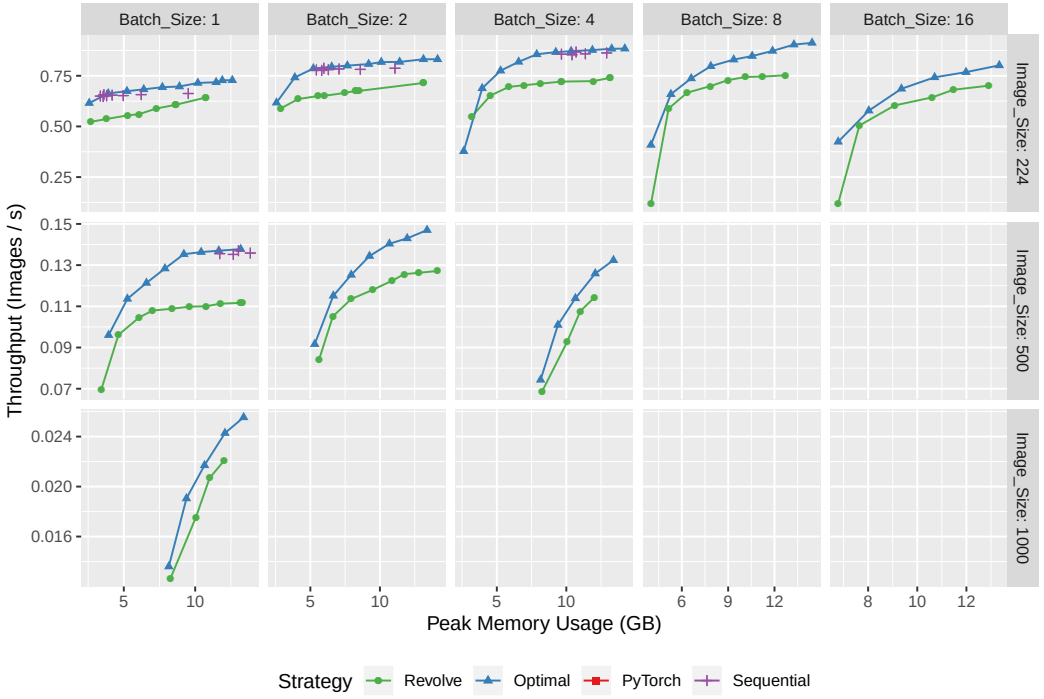


Fig. 16. Results for Resnet 1001, for different image sizes and batch sizes.