Optimal Memory-aware Backpropagation of Deep Join Networks

Olivier Beaumont, Julien Herrmann, Guillaume Pallez, Alena Shilova

PHILOSOPHICAL TRANSACTIONS A

rsta.royalsocietypublishing.org

Research

Article submitted to journal

Subject Areas:
Artificial intelligence [COMPUTER SCIENCE],
Theory of computing [COMPUTER SCIENCE]

Keywords:
backpropagation, memory, pebble game

Author for correspondence:
Guillaume Pallez
e-mail: guillaume.pallez@inria.fr

THE ROYAL SOCIETY PUBLISHING

Optimal Memory-aware Backpropagation of Deep Join Networks

Olivier Beaumont[1], Julien Herrmann[1], Guillaume Pallez (Aupy)[1] and Alena Shilova[1]

[1]Inria, University of Bordeaux, France

Deep Learning training memory needs can prevent the user from considering large models and large batch sizes. In this work, we propose to use techniques from memory-aware scheduling and Automatic Differentiation (AD) to execute a backpropagation graph with a bounded memory requirement at the cost of extra recomputations. The case of a single homogeneous chain, *i.e.* the case of a network whose all stages are identical and form a chain, is well understood and optimal solutions have been proposed in the AD literature. The networks encountered in practice in the context of Deep Learning are much more diverse, both in terms of shape and heterogeneity.

In this work, we define the class of backpropagation graphs, and extend those on which one can compute in polynomial time a solution that minimizes the total number of recomputations. In particular we consider join graphs which correspond to models such as Siamese or Cross Modal Networks.

1. Introduction

Training for Deep Learning Networks (DNN) induces a memory problem. Amongst the different strategies to parallelize and accelerate this training phase come hyper-parameter tuning and data-parallelism [GB10], that both need to replicate the set of weights of the neural network onto all participating resources, thus limiting the size of the model or depth of the graph.

In order to deal with memory issues, several solutions have been advocated. One of them is model parallelism [DCM+12]. It consists of splitting the network model into several non-overlapping parts, that are distributed over the different resources.

The Deep Neural Network (DNN) model can in general be seen as a Directed Acyclic Graph and the different vertices of the DAG are split across the resources. Each time there is an edge between two vertices allocated onto two different resources, this will induce the communication of the associated forward and backward activations during the training phase. As it is in general the case for the training of DNNs, we will assume that the weights are updated after the forward and backward propagation of small groups of training samples that will be called mini-batches, as opposed to full batch learning (where all samples are used before updating the weights) and stochastic gradient descent (where updates are performed after each sample). Model parallelism can be considered as a solution to this problem since model weights are distributed among participating nodes. In this context, the problem becomes a general graph partitioning problem [DCM+12] where the goal is to balance the weights between the different nodes while minimizing the weights of cut edges (i.e. edges whose two extremities are on different resources). This approach has the advantage that it can be combined with data parallelism [DAM+16].

Another complementary approach is the problem of scheduling a graph with a shared bounded memory and to use recomputation of functions. This problem is known in the scheduling literature as the Register Allocation problem or Pebble Game [Set75]. In the Register Allocation problem, in order to execute a task, all its inputs need to be stored in registers. The question is then to decide whether it is possible to process the graph with a bounded number of unit-size registers (or memory slots). Sethi [Set75] showed that this problem is NP-complete for general task graphs. Further study showed that the problem is solvable in polynomial graph for tree-shaped graphs [Liu87], or recently Serie-Parallel graphs [KLMU18].

In this work, we are interested in what we denote by backpropagation graphs: given a Directed Acyclic Graph (DAG) with a single exit vertex, we construct a dual identical graph where all edges are reversed, and where the input of each vertex of the initial graph is connected to its dual vertex. The source vertex of the dual graph and the sink vertex of the original graph are then merged into a single vertex called *turn* (see Figure 1 for the case of a chain of vertices).

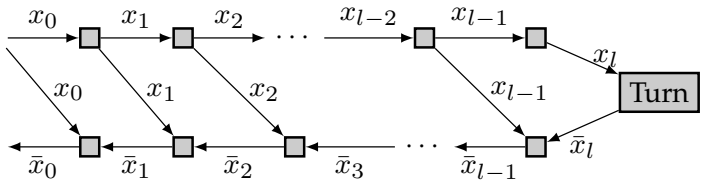


Figure 1: Data dependencies of the backpropagation graph corresponding to the transformation of a linear chain. Functions are modeled by grey vertices, arrows represent data dependencies (input/output of functions) where the data are either x_i and $\bar{x}_i$. The chain is the graph on top, its dual graph is on the bottom.

These types of graphs have been widely studied in the context of Automatic Differentiation (AD) [Gri89]. For a given batch size and a given network model and even on a single node without relying on model parallelism strategies, it saves memory at the price of activation re-computations. In the context of AD, networks can be seen as (long) homogeneous (i.e., all stages are identical) chains and the forward activation corresponding to the $i-$th stage of the chain has to be kept into memory until the associated backward stage. Checkpointing techniques determine in advance which forward activations (checkpoints) should be kept into memory and which one should be recomputed from stored checkpoints when performing the backward phase. Many studies have been performed to determine optimal checkpointing strategies for AD in different contexts, depending on the presence of a single or multi level memory [AHHR16,AH19]. In the case of homogeneous chains, closed form formulas providing

the exact positions of checkpoints have even been proposed [GW00] (algorithm REV in the rest of this paper), although the general algorithmic ingredient is to derive optimal checkpointing strategies through Dynamic Programming [GW00]. This technique has been recently advocated for DNN in several papers [GMD+16,CXZG16]. In the context of homogeneous chains, a periodic checkpointing strategy has recently been implemented in PyTorch [PGC+17,per18]. Nevertheless, DNN models are not restricted to homogeneous chains and there is a need for checkpointing algorithms in more general contexts. While optimal scheduling and checkpointing are still open problems in the general case, there are some solutions which in some way benefit from the findings in AD checkpointing strategies [CXZG16,GMD+16]. However, they are designed to deal with sequential models only, thus making it inappropriate for more sophisticated cases.

In this paper, we propose a first attempt to find optimal checkpointing strategies adapted to more general networks and to show how dynamic programming techniques developed in the context of AD can be adapted to DNN networks. More specifically, we concentrate on the particular context of DNNs consisting of several independent chains whose results are gathered through the computation of the loss function (*Join graph*). We show that this specific case can be solved using dynamic programming, but at the price of more sophisticated techniques and higher computational costs. Note that several popular classes of problems can be modeled as a join graph amongst which are Cross-Modal embeddings or Siamese networks.

Cross-Modal embeddings [MBO+18,MAB+19] are models used when there are multiple sources of data and the goal is to find the connection between those sources. For example, in the image-recipe retrieval task [MBO+18], having both a dataset of dish images and a dataset of recipes represented as a text corpus, the goal is to find a matching image for each recipe. Thus, a Convolution Neural Network (CNN) is applied to process images and extract features while a Long Short-Term Memory (LSTM) network is used for the text part. Then, all feature vectors yielded by both networks are further processed with the help of a small number of fully connected layers before being concatenated to compute an associated loss. In practice, training such a model often consists of training individually each sub-model for each data source and then using them only as feature extractors to train the fully connected layers on top of it. Indeed, training the whole model is not performed due to larger runtime for training and much larger memory requirements. In the latter case, the approach proposed in the current paper can be used as checkpointing strategy and can significantly decrease memory consumption.

Siamese Neural Network [BGL+94,DFS17,MMBS14] can also directly benefit from our approach. They are widely used for objects recognition. The main idea behind these models is to use the same CNN, but for different images, and then finally use all the outputs to estimate a loss that represents a similarity metric. Depending on the choice of the loss function, it can either correspond to a two-chains computational graph [BGL+94,MMBS14] where the loss is computed based on two images, or to a three-chains computational graph, where triplet loss is applied [HA15]. Due to memory constraints, most of the CNNs used in these models are not very deep [DFS17]. However, it is known that deeper neural networks could offer a better quality. Therefore, using checkpointing techniques to decrease memory needs may be used to consider larger and deeper models in the context of Siamese networks.

The rest of the paper is organized as follows. In Section 2, we present the problem and the general framework in which we can derive our scheduling algorithm. Then, a characterization of optimal solutions is proposed in Section 3(a) and is later used to find the optimal checkpointing strategy through dynamic programming in Section 3(b). Finally, we present our implementation and simulation results in Section 4, before providing concluding remarks and perspectives in Section 5.

2. Model

In this work, we consider the Register Allocation Problem [Set75] (*aka* Pebble Game) for special types of graphs, denoted as *backpropagation graphs*. These graphs are obtained by transforming a Directed Acyclic Graph (DAG) as explained in Definition 1.

(a) Platform model and optimization problem

In this work, we consider a sequential platform (at all time, all the compute elements are dedicated to the same job) with a finite memory $\mathcal{M}$ of size c. Activations must be checkpointed into $\mathcal{M}$ until they are used in the backpropagation or they must be discarded from $\mathcal{M}$ and then recomputed from a previous checkpointed value. We do not consider the possibility of offloading activations to another (larger) memory level as in [RGC+16].

A job is represented as a Directed Acyclic Graph (DAG) $\mathcal{G} = (V, E)$, where each vertex of $v \in V$ represents a compute operation (with a given execution time), and each edge of $(v_1, v_2) \in E$ represents a data dependency where an output of v_1 is an input of v_2. Given a graph $\mathcal{G}$, the problems under consideration are (i) can we execute it with a memory of size c (*pebble game problem*) and (ii) if we can, what is the minimal execution time to execute $\mathcal{G}$ (*makespan problem*) with a memory of size c.

In order to process a job on this platform, all its inputs need to be stored in memory at the beginning of the execution. In what follows, we assume that the memory is larger than the minimal amount of memory to perform the training phase, which is formally defined in Theorem 2. It should be noted that the Theorem 2 formula takes into account the possibility of computing some activations several times and therefore does not assume that there is enough memory to run the graph in one go. Then, the core of the *makespan problem*, is to choose which activations to store and which activations should be recomputed.

(b) Backpropagation graphs

In this work we consider specifically the problem of scheduling backpropagation graphs.

Definition 1 (Backpropagation transformation (BP-transform))**.** Given a DAG $\mathcal{G}$ with a single sink vertex. The *BP-transform* of $\mathcal{G}$ is defined by the following procedure:

(i) Build the dual graph $\tilde{\mathcal{G}}$ defined as the same graph where all edges are inversed.
(ii) For a given vertex in $\mathcal{G}$, connect its input edges to its dual vertex in $\tilde{\mathcal{G}}$.
(iii) Finally, merge the sink vertex of $\mathcal{G}$ and the source vertex of $\tilde{\mathcal{G}}$ as a single vertex, denoted as the *turn*.

Note that the vertices of the initial graph are denoted as *forward steps*, while the vertices of the dual graph are denoted as *backward steps*. In the rest of this work, we make the assumption that all forward steps have the same execution cost $u_f \in \mathbb{R}^+$, while all the backward steps have the same execution cost $u_b \in \mathbb{R}^+$[1]. The cost of the Turn operation is denoted as $u_t \in \mathbb{R}^+$. In addition, we assume that all input/output data have the same (unit) size. We give without proof two properties of the backpropagation graph which justify this study:

Property 1 (Properties of the BP-graph)**.** *Given a DAG $\mathcal{G}$ with n verticess and a single sink vertex:*

P1: The backpropagation graph of $\mathcal{G}$ is a DAG;
P2: Without recomputation of vertices, the minimal memory usage to go through this graph is $O(n)$.

Indeed, to scale these types of computations, we are interested in the question of the overhead in computation when one uses much less memory space.

The most simple and most studied backpropagation graph is the transformation of a chain graph (as shown in Figure 1) in Automatic Differentiation. In this case, it is known that for a given volume of memory, one can compute the optimal solution in polynomial time [GW00]. In this work, we consider transformation of join graphs (Figure 2). Join graphs are used by several deep-

[1]For clarity this work is focused on the uniform case. It is possible to extend at a small cost the results to non-uniform time steps as was done for the case of single chains by Walther [Wal99, Section 3.4].

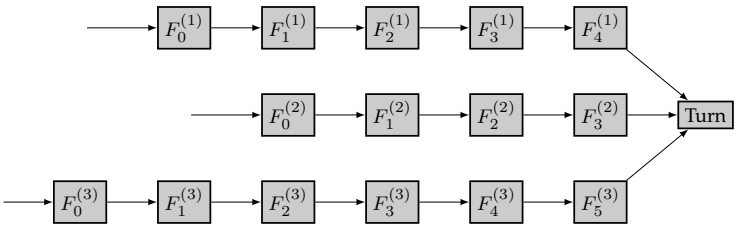


Figure 2: A join graph with 3 branches of respective length 5, 4 and 6.

learning models such as Siamese Neural Network [BGL+94,DFS17,MMBS14] or Cross-modal embeddings [MBO+18,MAB+19].

Definition 2 (BP-transform of join graphs)**.** Given a join graph $\mathcal{G}_k$ with k branches of respective length $\boldsymbol{\ell}_j$, its BP-transform can be described by the following set of equations:

$$F_i^{(j)}(x_i^{(j)}) = x_{i+1}^{(j)} \qquad \text{for } 1 \le j \le k \text{ and } 0 \le i < \boldsymbol{\ell}_j,$$
$$\bar{F}_i^{(j)}(x_i^{(j)}, \bar{x}_{i+1}^{(j)}) = \bar{x}_i^{(j)} \qquad \text{for } 1 \le j \le k \text{ and } 0 \le i < \boldsymbol{\ell}_j,$$
$$\text{Turn}(x_{\boldsymbol{\ell}_1}^{(1)}, \dots, x_{\boldsymbol{\ell}_k}^{(k)}) = (\bar{x}_{\boldsymbol{\ell}_1}^{(1)}, \dots, \bar{x}_{\boldsymbol{\ell}_k}^{(k)}).$$

The dependencies between these operations are represented by the graph $\mathcal{G} = (V, E)$ depicted in Figure 3, in the case of $k = 2$ chains.

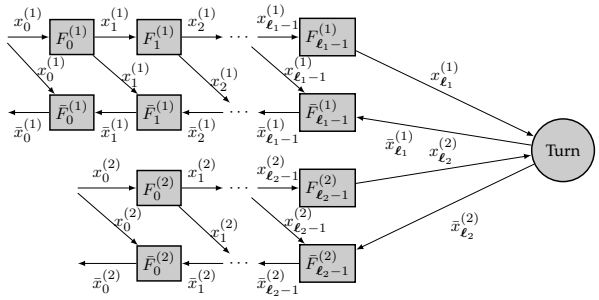
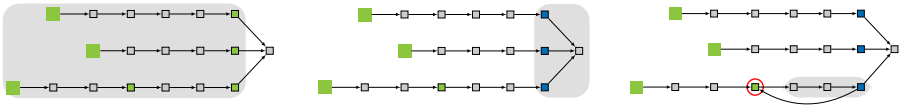


Figure 3: Data dependencies in the multiple adjoint chains problem with two chains. In the following we call *forward* (resp. *backward*) data the output of F (resp. $\bar{F}$) functions.

In the rest of this work, we only use the graph of forward steps to represent the BP-transform. Finally, we are interested in solving the following problem:

Problem 1 ($\text{PROB}_{\text{join}}(\boldsymbol{\ell}, c)$)**.** Given the BP-transform of a join DAG with k branches of respective lengths $\boldsymbol{\ell} \in \mathbb{N}^k$. The respective costs for forward, backward and turn operations are given by u_f, u_b and u_t. Given $c \in \mathbb{N}$ memory slots, minimize the makespan, where the initial memory state is given by $\mathcal{M} = \{x_0^{(1)}, \dots, x_0^{(k)}\}$ and the final memory state is given by $\mathcal{M} = \{\bar{x}_0^{(1)}, \dots, \bar{x}_0^{(k)}\}$.



(a) The forward phase (b) The turn (c) The backward phase

Figure 4: The three main phases of the algorithm. Green blocks correspond to the storage of "forward" data, blue blocks to storage of "backward" data. To process the backward phase, the red-circled value stored during the forward phase is used to recompute subsequent values.

3. Deriving an optimal solution

In this section, we present the core idea to compute an optimal solution. In particular, we show that there are optimal solutions that satisfy properties on their behavior. We call these solutions *canonical* optimal solutions. Then we restrict the search for optimal solutions to the canonical solutions. For simplicity, we leave out all the proofs of this section and focus on giving intuitions. The interested reader can find the proofs in the companion report [BHPS19] for the more general case where not all final outputs need to be kept into memory. The context considered in this paper corresponds to the case where all b_i values are equal to 1 in [BHPS19], where b_i indicates if the results of chain i needs to be kept in memory.

For terminology sake, let us first notice that an algorithm for $\text{PROB}_{\text{join}}(\ell, c)$ can be decomposed into three different phases (see Figure 4):

- **The Forward phase**: we traverse all branches to write all inputs of the turn in memory. During this phase one cannot compute any backward operations, but some of the input data can be stored.
- **The Turn**: at the beginning of this phase, all input data of the turn are stored in memory, and are replaced by all output data at the end in the same memory locations. Indeed, the input data of the turn will never be used anymore.
- **The Backward phase**: we read some input data that were stored earlier to backpropagate a subset of the graph.

(a) Canonical form and optimal solutions

We say that a solution is in a canonical form if it obeys the following recursive structure:

(i) Given c memory slots, from an input x written on memory, j forward steps are performed on the branch m of x;
(ii) The output after those steps is stored into memory;
(iii) A canonical solution for the smaller problem where all branches' size are unchanged except for branch m which is now smaller by j forward steps (recurrence formula) with $c-1$ memory slots.
(iv) From the input x in memory, the j steps following x are *backpropagated*[2] using all available memory slots.

We show in Figure 5 what a canonical solution looks like on a small example. In this section, we show that there exists an optimal solution in this canonical form. In order to prove this result, we show that the backward phase is computed following three properties (represented graphically in Figure 6):

Property 2 (Canonical properties)**.** *Given a join graph $\mathcal{G}_k$ with k branches:*

[2]We use this term to say that we have computed the dual of a vertex of the initial graph

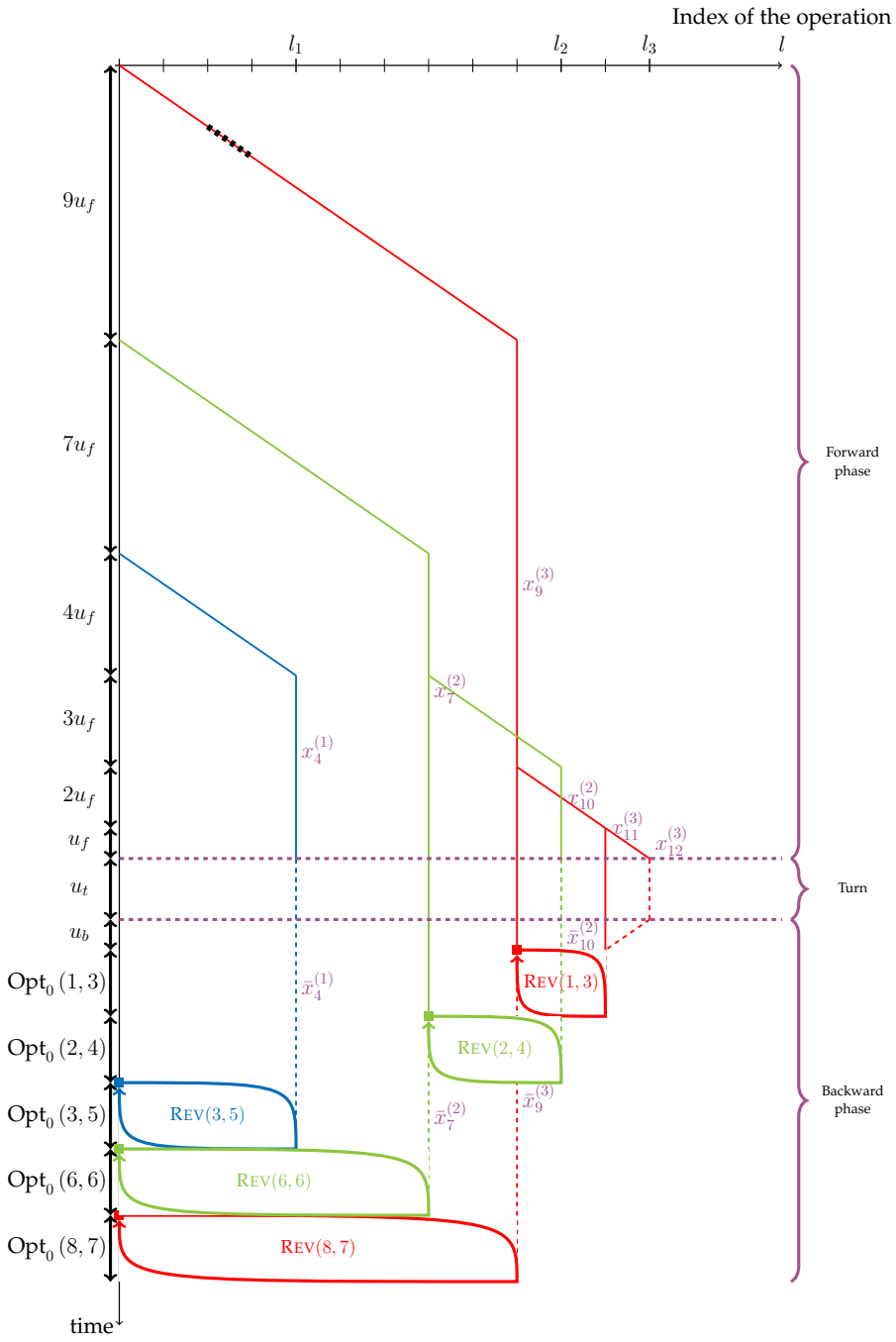


Figure 5: A canonical solution on the join with three branches of respective lengths 4 (blue), 10 (green) and 12 (red), with 9 memory slots. The execution time is represented on the vertical axis, the progress on each of the branches is represented on the horizontal axis. As an example, the dark dashed line at the top of the figure represent the computation of $F_2^{(3)}$. Full (resp. dashed) vertical lines are memory checkpoints of forward data (resp. backward data). $\textsc{Rev}(l, c)$ is the solution [GW00] to the easier problem where there is only one branch of size l and c checkpoints (see Fig 1), $\mathrm{Opt}_0(l, c)$ is its execution time.

(a) Stability 1 (b) Checkpoint Persistence (c) Stability 2

Figure 6: The three key properties of a canonical solution. Green blocks correspond to "forward" data being stored, blue blocks to "backward" data stored. The forbidden operations given by the properties are crossed out in red. For Stability 2, if during the backward phase, we have read the circled checkpoint, then the canonical solution starts by backpropagating (represented by the back arrow) all greyed-out operations until the "backward" data of the circled-out operation is stored.

- **Stability 1:** *If we have* backpropagated *an operation of $\mathcal{G}_k$, then we do not access anymore any of its descendent in $\mathcal{G}_k$.*
- **Checkpoint Persistence** *If the output of an operation of $\mathcal{G}_k$ is stored, then until it is* backpropagated, *we do not access any of its parent in $\mathcal{G}_k$.*
- **Stability 2** *If the output of an operation of $\mathcal{G}_k$ is read, then until it is* backpropagated, *we do not access any part of other branches of the BP-transform of $\mathcal{G}_k$.*

Then there exists an optimal solution for $\textsc{Prob}_{join}(\boldsymbol{\ell}, c)$ *that satisfies these properties.*

With these properties we can state the following result:

Theorem 1. *There exists an optimal solution that follows the canonical form.*

Outline of the proof. Finding an optimal solution reduces to finding what data should be rewritten during the forward phase, and in which order it should be read during the backward phase. Indeed, once this element is read, the backward phase consists of backpropagating the segment of the branch until the location of the backward data already computed (which recursively corresponds to the backpropagation of another element written during the forward phase). In addition, during the backward phase, the backpropagation of each subset of the graph reduces to the problem of a chain whose solution $\textsc{Rev}$ is known [GW00].

Finally, it remains to be shown that the forward phase satisfies the canonical property. This is simple: one can notice that at no cost, one can choose any order for going through the different *forward segments*[3] (except the last one) of the branches. Thereby in particular we can use the reverse order in which they are backpropagated during the backward phase. □

(b) Optimal solution

Using the properties derived in the previous section on canonical solutions, we can see that the problem can be solved with a dynamic programming algorithm. Indeed, given a join graph of length $\boldsymbol{\ell} \in \mathbb{N}^k$, the problem can be seen as:

(i) Performing j forward steps on one of the branches
(ii) Writing the output forward activation after those j steps to memory
(iii) Solving the problem for the smaller problem where all branches have unchanged size except for the branch on which the output has been kept, and which is now smaller by j steps (recurrence formula) and with one fewer memory checkpoint.
(iv) Backpropagating the j forward steps done before.

[3]Defined as the forward steps between two consecutive written elements

We now explain how to derive this dynamic program. Let us first consider the solution to the *pebble game* problem:

Theorem 2 (Minimal memory requirement)**.** *The minimal memory c_{min} needed to execute* $\textsc{Prob}_{join}(\boldsymbol{\ell}, c)$ *is:*

$$c_{min}(\boldsymbol{\ell}) = \begin{cases} k & \text{if } \boldsymbol{\ell} = \vec{0}, \\ k + \sum_{i=1}^{k} \mathbb{I}[\boldsymbol{\ell}_i \neq 0] & \text{if } \exists j,\ \boldsymbol{\ell}_j = 1, \\ k + \sum_{i=1}^{k} \mathbb{I}[\boldsymbol{\ell}_i \neq 0] + 1 & \text{otherwise.} \end{cases} \tag{3.1}$$

Outline of the proof. Intuitively, the case where $\boldsymbol{\ell} = \vec{0}$ corresponds to the situation where we simply need to perform the turn. For the general case, the peak is right after the turn: all initial input for the branch of non zero size need to be stored. In addition, all the outputs of the turn have to be stored as well. Finally, one additional slot is then needed to perform the back and forth between the input data and output data. The only exception to this is when one of the branches has length exactly 1. In that case right after the turn we can backpropagate the input of this branch, hence freeing the memory slot used for this input in order to do the back and forth. □

We are now able to establish the following theorem, that proves that general instances of $\textsc{Prob}_{\text{join}}(\boldsymbol{\ell}, c)$ can be solved using dynamic programming. We denote by $\text{Opt}_0\,(l, c)$ the execution time of the algorithm $\textsc{Rev}(l, c)$ [GW00] (optimal solution when $k = 1$).

Theorem 3. *Given a join DAG $\mathcal{G}_k$ with k branches of lengths $\boldsymbol{\ell}$ and given c memory slots, the execution time $Opt_{join}\,(\boldsymbol{\ell}, c)$ of an optimal solution to* $\textsc{Prob}_{join}(\boldsymbol{\ell}, c)$ *is given by*

$$Opt_{join}\,(\boldsymbol{\ell}, c) = \infty \qquad \text{if } c < c_{\min}(\boldsymbol{\ell}),$$

$$Opt_{join}\left(\vec{0}, c\right) = u_t \qquad \text{if } c \geq k,$$

$$Opt_{join}\,(\boldsymbol{\ell}, c) = u_f + u_t + u_b \qquad \text{if } \exists j \text{ s.t. } \boldsymbol{\ell}_j = 1, \forall i \neq j, \boldsymbol{\ell}_i = 0 \text{ and } c \geq k + 1,$$

For other cases:

$$Opt_{join}\,(\boldsymbol{\ell}, c) = \min_{\substack{1 \leq m \leq k \\ 0 < i \leq \boldsymbol{\ell}_m}} \left[i \cdot u_f + Opt_{join}\left(\boldsymbol{\ell}_{[\boldsymbol{\ell}_m \leftarrow \boldsymbol{\ell}_m - i]}, c - 1\right) + Opt_0\,(i - 1, c - (k - 1)) \right]$$

The complexity to compute $Opt_{join}\,(\boldsymbol{\ell}, c)$ is $O\left(c \cdot k \cdot \left(\max_{i=1}^{k} \boldsymbol{\ell}_i\right)^{k+1}\right)$.

Note that in practice k is small (2 or 3).

Sketch of proof. The dynamic program describes exactly the best makespan of any algorithm with canonical form for all values of $\boldsymbol{\ell}$ and c.

The initialization is the time to perform the turn. Indeed, as we have seen in Theorem 2, there is a subtlety in the manipulation of the checkpoints to perform the turn depending whether there is a chain of length one or not, which would allow to use fewer checkpoints. Finally, when there are not enough memory slots to execute the graph, we set the value to ∞ allowing to discard the cases that use more memory than what is available. □

Note that while this provides an execution time, it is fairly easy to derive an algorithm based on this using standard Dynamic Programming techniques.

4. Evaluation

In this section, we depict the results of simulations on linearized homogeneous versions of Cross-Modal embeddings (CM) and Siamese Networks (SN) with triplet loss. Linearized homogeneous versions of the neural networks are the ones where all computational costs ($u_f = u_b = u_t = 1$) and

storage costs are homogeneous. In this context, a multi chain network is completely defined by the lengths of the different branches and the size of the memory expressed in terms of memory slots.

We propose three observations:

(i) The trade-off Makespan vs Memory usage;
(ii) For a fixed number of storage slots, an observation on the growth of the number of recomputations needed as a function of the number of forward operations;
(iii) A comparison between the optimal solution and an algorithm that does not take into account the join structure but that considers the graphs as an equivalent length linear chain.

In order to compare the different types of networks in a normalized way, we consider several models with analogous sizes and computational costs. For a given value of L, we consider three graph structures:

- Siamese Neural Networks (SNN) with 3 chains of lengths $(2L, 2L, 2L)$. Here also, L can be large as these neural networks are usually applied to images, thus deep neural networks are also possible as they are better in pattern retrieval. As soon as $c \geq C_{SNN}(L) = 6L + 3$, all forward activations can be stored and therefore makespan is minimal and is equal to $\text{Span}^*_{SNN}(L) = 12L + 1$.
- Cross-Modal (CM) embedding networks with two chains of sizes $(L, 5L)$. The motivation is CM with two different types of data sources (images and texts), one of the chains is much longer than the other because more layers are needed to extract useful patterns (e.g. images are usually processed with deep neural networks like ResNet while text can be efficiently encoded into vectors of smaller dimensions with the help of some shallow neural networks instead). Analogously to the case of SNN, as soon as the memory c is larger than $C_{CM}(L) = 6L + 2$, then the makespan is minimal and is equal to $\text{Span}^*_{CM}(L) = 12L + 1$.
- Finally, we also consider the case of a single chain of length $6L$. This corresponds to the case of Recurrent Neural Networks (RNN). Analogously to the case of CMs of SNNs, as soon as $c \geq C_{RNN}(L) = 6L + 1$, all forward activations can be stored and therefore the makespan is minimal and is equal to $\text{Span}^*_{RNN}(L) = 12L + 1$. This case also serves as a lower bound on the makespan that can be reached by the previous models.

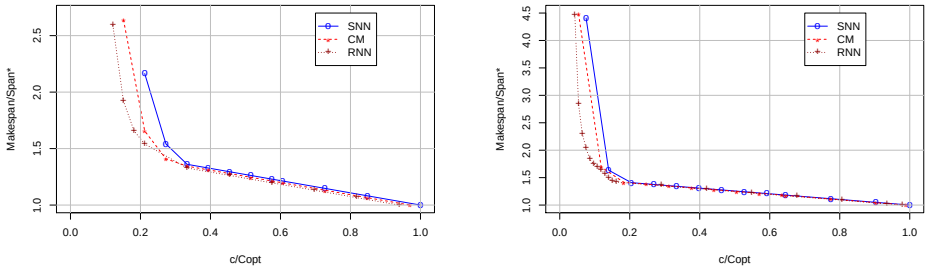
In the following, we denote by $\text{Span}^*(L) = 12L + 1$ the minimal makespan for all models.

Trade-off Memory – Makespan The first question that we consider is, given a fixed number of forward operations (described by L for all scenarios), how does the makespan evolve as a function of the number of memory slots available.

We run our experiments for $L = 5$ and 15. The plots depicting the evolution of makespan with the amount of memory are gathered in Figure 7. Specifically, the x-axis represents the fraction of memory slots available with respect to the minimal number C_{Opt} (which differs slightly depending on the model) to achieve minimal makespan, $\text{Span}^*(L)$. The y-axis represents the ratio between the achieved makespan and $\text{Span}^*(L)$. Thus, point (x, y) on the CM plot means that for a CM network of length $(5L, L)$, with $x \cdot C_{CM}$ memory slots, the makespan is $y \cdot \text{Span}^*(L)$.

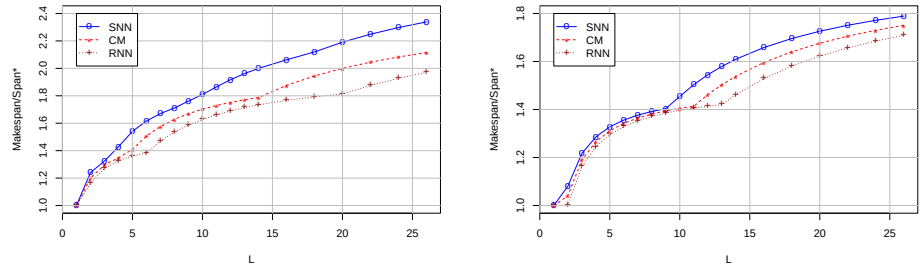
The plots related to CM (resp. SNN) start from memory size 5 (resp. 7), which is exactly the value of $c_{\min}$ for two (resp. three) chains, as proved in Theorem 2. We can notice that the makespan first significantly decreases with the first additional memory slots. In addition, it seems that once it reaches a threshold $k \cdot \text{Span}^*$ with $k \simeq 1.5$, the makespan ratio linearly decreases to Span^* with the number of additional memory slots.

Hence, this shows that this checkpointing strategy is very efficient in decreasing the memory needs while only slightly increasing the makespan. For instance, consider the point where



(a) $L = 5$ (b) $L = 15$

Figure 7: Makespan evolution with respect to different amount of total memory.

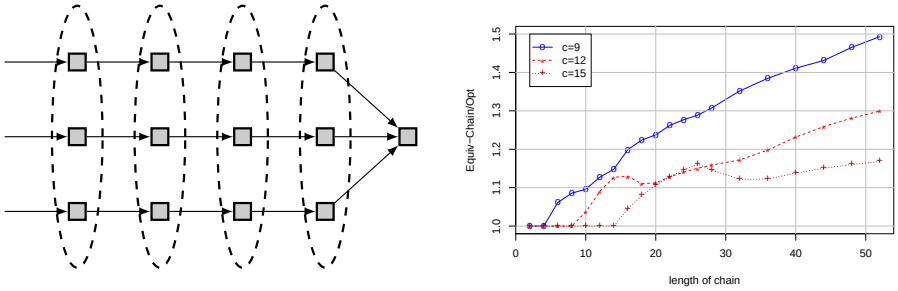


(a) $c = 9$ (b) $c = 13$

Figure 8: Makespan evolution with respect to different L for fixed memory size c.

$c/C_{\text{Opt}} = 0.5$. This corresponds to halving the memory needs with respect to what is needed to achieve minimal makespan. In all cases, halving memory needs only induces an increase of approximatively 25% on the makespan. In addition, we can also observe that even with a very small memory (say $c_{\min} + 2$), the makespan is less than doubled compared to Span*.

Makespan evolution for fixed memory In Figure 8, we depict the dual situation, where the number of memory slots is fixed on each plot (either 9 or 13) and the ratio of the achieved makespan over Span$^*(L)$ is depicted. Several observations can be made. The first one is that the gap to the lower-bound (RNN) is rather small and decreases with the number of available checkpoints. Obviously this gap increases slowly with the size of the model. Note that we have observed an exception when the number of available checkpoints is exactly the minimum number of memory checkpoints (for instance $c = 7$, performance of SNN). This exception is not surprising given the observations of the previous paragraph and the important improvements in performance when the number of available checkpoint is slightly larger than $c_{\min}$. In addition, it is interesting to observe that for SNNs and CNNs, the ratio follows a pattern similar to that of RNNs: different thresholds, and between those thresholds, a performance shaped as $\alpha - \beta/L$. Indeed, for the case of RNN those performance have been proven via a closed-form formula [GW00]. This can motivate the search for a similar closed-form formula for the problem of chains, which could lead to an algorithm of polynomial complexity in the number of branches. Finally, another observation is that the overall growth in makespan for a fixed number of checkpoints is quite slow even with few checkpoints, hence encouraging the use of these strategies.



(a) Transformation of a join with three branches into a chain with four levels

(b) Performance of EQUIV-CHAIN vs Optimal.

Figure 9: Graph transformation (left) and performance (right).

Comparison to linearization strategy The previous evaluation aims at evaluating the gain that can be expected by trading space for time. In this final evaluation, we rather aim at showing the importance of taking the structure of the graph into consideration. In particular we compare the strategy presented in this work to a two-step strategy that consists in (i) transforming the join in a linear chain; (ii) using an existing strategy for linear chains.

To transform the join as a linear chain, we group together operations that are at the same distance of the turn (see Figure 9). In this case, one can execute any algorithm for linear chains while considering that each forward step has cost $k.u_f$, each backward step $k.u_b$ and that the memory needed for a checkpoint is k times the memory needed for a single input/output. For comparison, we consider the best possible algorithm for this structure, REV [GW00]. We call this heuristic: EQUIV-CHAIN.

In order to consider the best possible scenario for EQUIV-CHAIN: (i) we only consider balanced graphs (SNN graphs); (ii) we consider only slot number proportional to the number of branches. We plot in Figure 9 the ratio Makespan of EQUIV-CHAIN over Opt, the performance of our algorithm for different numbers of checkpoints (9, 12 or 15). Because we consider only one type of model, for this plot we use the actual length of the graph as x-axis (length = $2L + 1$).

The results in this optimistic scenario show that EQUIV-CHAIN is surprisingly robust for small graphs. More expected, its performance steadily decreases when the ratio length/number of checkpoints increases. This is not surprising because our algorithm has more freedom in its choice of execution order. This shows the importance of taking into account the structure of the backpropagation graph when studying checkpointing strategies.

5. Conclusion and Perspectives

Being able to perform learning in deep neural networks is a crucial operation, which requires important memory needs in the feed-forward model, because of the storage of activations. These memory needs often limit the size and therefore the accuracy of used models. The automatic differentiation community has implemented checkpointing strategies, which make it possible to significantly reduce memory requirements at the cost of a moderate increase in the computing time, and backpropagation schemes are very similar in the cases of automatic differentiation and deep learning. On the other hand, the diversity of task graphs (ResNet, Inception and their combinations) is much more important in the context of DNNs. It is therefore crucial to design checkpointing strategies for backpropagation on a much broader class of graphs than the homogeneous chains on which the literature on automatic differentiation has focused.

The goal of this paper is precisely to extend the graph class, by considering multiple independent chains that meet when calculating the loss function. This class of graphs allows, from

an application point of view, to also consider neural networks such as Cross Modal Networks (CMNs) and Siamese Neural Networks (SMNs). In the case of multi-chain, we were able to build a dynamic program that allows us to compute the optimal strategy given the size of the available memory, i.e. a strategy that fulfills memory constraints while minimizing the number of recalculations.

This work opens several natural perspectives. The first one obviously concerns the extension to other broader classes of backpropagation graphs, in order to take into account larger classes of DNNs. After working on multiple chains, which has proven to be a much more difficult problem than the single chain problem, we consider that it is probably essential to choose the characteristics of the graph class in question carefully in order to produce algorithms of reasonable complexity with a practical impact in deep learning. Another approach is the design of approximation algorithms for this problem, which would have a low execution time and for which good performance can nevertheless be guaranteed. From the studies we have done, it does seem that there is a fairly high density of optimal and quasi-optimal solutions, so it is possible that there are simpler algorithms to find them.

Authors' Contributions. All authors participated to the model, derivations of theoretical results and writing of the manuscript. JH implemented the main algorithm, AS and GP performed the evaluations. All authors read and approved the manuscript.

Competing Interests. The authors declare that they have no competing interests.

Funding. This work was supported in part by the French National Research Agency (ANR) in the frame of DASH (ANR-17-CE25- 0004) and in part by the Project Région Nouvelle Aquitaine 2018-1R50119 "HPC scalable ecosystem".

References

AH19 Guillaume Aupy and Julien Herrmann.
H-Revolve: A Framework for Adjoint Computation on Synchrone Hierarchical Platforms.
working paper or preprint, March 2019.

AHHR16 Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert.
Optimal multistage algorithm for adjoint computation.
SIAM Journal on Scientific Computing, 38(3):232–255, 2016.

BGL+94 Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah.
Signature verification using a" siamese" time delay neural network.
In *Advances in neural information processing systems*, pages 737–744, 1994.

BHPS19 Olivier Beaumont, Julien Herrmann, Guillaume Pallez, and Alena Shilova.
Optimal Memory-aware Backpropagation of Deep Join Networks.
Research Report RR-9273, INRIA, 2019.

CXZG16 Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin.
Training deep nets with sublinear memory cost.
arXiv preprint arXiv:1604.06174, 2016.

DAM+16 Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey.
Distributed deep learning using synchronous stochastic gradient descent.
arXiv preprint arXiv:1602.06709, 2016.

DCM+12 Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al.
Large scale distributed deep networks.
In *Advances in neural information processing systems*, pages 1223–1231, 2012.

DFS17 William Du, Michael Fang, and Margaret Shen.
Siamese convolutional neural networks for authorship verification.
Proceedings, 2017.

GB10 Xavier Glorot and Yoshua Bengio.

Understanding the difficulty of training deep feedforward neural networks.
In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

GMD+16 Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves.
Memory-efficient backpropagation through time.
In *Advances in Neural Information Processing Systems*, pages 4125–4133, 2016.

Gri89 Andreas Griewank.
On automatic differentiation.
Mathematical Programming: recent developments and applications, 6(6):83–107, 1989.

GW00 Andreas Griewank and Andrea Walther.
Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation.
ACM Transactions on Mathematical Software (TOMS), 26(1):19–45, 2000.

HA15 Elad Hoffer and Nir Ailon.
Deep metric learning using triplet network.
In *International Workshop on Similarity-Based Pattern Recognition*, pages 84–92. Springer, 2015.

KLMU18 Enver Kayaaslan, Thomas Lambert, Loris Marchal, and Bora Uçar.
Scheduling series-parallel task graphs to minimize peak memory.
Theoretical Computer Science, 707:1–23, 2018.

Liu87 Joseph WH Liu.
An application of generalized tree pebbling to sparse matrix factorization.
SIAM Journal on Algebraic Discrete Methods, 8(3):375–395, 1987.

MAB+19 M. Mueller, A. Arzt, S. Balke, M. Dorfer, and G. Widmer.
Cross-modal music retrieval and applications: An overview of key methodologies.
IEEE Signal Processing Magazine, 36(1):52–62, Jan 2019.

MBO+18 Javier Marin, Aritro Biswas, Ferda Ofli, Nicholas Hynes, Amaia Salvador, Yusuf Aytar, Ingmar Weber, and Antonio Torralba.
Recipe1m: A dataset for learning cross-modal embeddings for cooking recipes and food images.
arXiv preprint arXiv:1810.06553, 2018.

MMBS14 Jonathan Masci, Davide Migliore, Michael M Bronstein, and Jürgen Schmidhuber.
Descriptor learning for omnidirectional image matching.
In *Registration and Recognition in Images and Videos*, pages 49–62. Springer, 2014.

per18 Periodic checkpointing in pytorch, 2018.
https://pytorch.org/docs/stable/checkpoint.html.

PGC+17 Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer.
Automatic differentiation in pytorch, 2017.

RGC+16 Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler.
vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design.
In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 18. IEEE Press, 2016.

Set75 Ravi Sethi.
Complete register allocation problems.
SIAM journal on Computing, 4(3):226–248, 1975.

Wal99 Andrea Walther.
Program reversal schedules for single-and multi-processor machines.
PhD thesis, PhD thesis, Institute of Scientific Computing, Technical University Dresden, Germany, 1999.