Optimal Memory-aware Backpropagation of Deep Join Networks

Olivier Beaumont, Julien Herrmann, Guillaume Pallez (Aupy), Alena Shilova

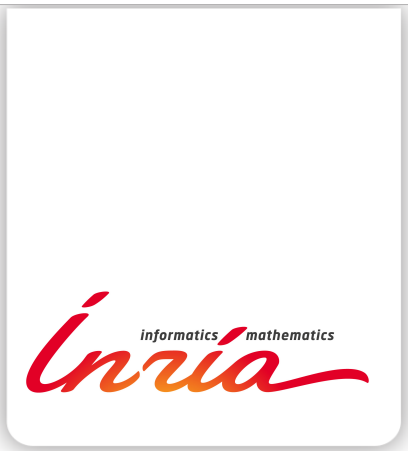
RESEARCH REPORT

N° 9273

May 2019

Project-Teams TADaaM and Hiepacs

ISSN 0249-6399 ISRN INRIA/RR--9273--FR+ENG



Optimal Memory-aware Backpropagation of Deep Join Networks

Olivier Beaumont*, Julien Herrmann*, Guillaume Pallez (Aupy)*, Alena Shilova*

Project-Teams TADaaM and Hiepacs

Research Report n° 9273 — May 2019 — 29 pages

Abstract: In the context of Deep Learning training, memory needs to store activations can prevent the user to consider large models and large batch sizes. A possible solution is to rely on model parallelism to distribute the weights of the model and the activations over distributed memory nodes. In this paper, we consider another purely sequential approach to save memory using checkpointing techniques. Checkpointing techniques have been introduced in the context of Automatic Differentiation. They consist in storing some, but not all activations during the feed-forward network training phase, and then to recompute missing values during the backward phase. Using this approach, it is possible, at the price of re-computations, to use a minimal amount of memory. 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 based on dynamic programming have been proved in the Automatic Differentiation literature. The networks encountered in practice in the context of Deep Learning are much more diverse, both in terms of shape and heterogeneity. The present paper can be seen as an attempt to extend the class of graphs that can be solved optimally. Indeed, we provide an optimal algorithm, based on dynamic programming, for the case of several chains that gathers when computing the loss function. This model typically corresponds to the case of Siamese or Cross Modal Networks.

Key-words: backpropagation, memory, pebble game

* Inria & Labri, Univ. Bordeaux

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Rétropropagation optimale de réseau profonds à forme de Joins

Résumé : L'espace mémoire nécessaire pour l'apprentissage de réseaux profonds peut empêcher l'utilisateur de considérer de grands modèles. Dans ce travail nous discutons l'utilisation des techniques d'ordonnancement sous contraintes mémoire utilisées en différentiation automatique (AD) pour executer des graphes de rétropropagation sous contraintes mémoires.

Le cas d'une chaine simple et homogène est maitrisé et la littérature en différentiation automatique offre de nombreuses techniques et solutions optimales sous diverses contraintes. Dans le cadre de l'apprentissage profond, les réseaux rencontrés sont souvent plus structurés et diverses (forme, hétérogénéité).

Dans ce travail nous définissons la classe des graphes de rétropropagation et nous étendons l'ensemble des grpahes sur lesquels on peut calculer en temps polynomial une solution optimale (en terme de temps d'exécution) sous contraintes mémoire.

En particulier, nous considérons le cas des *join* qui correspondent à des modèles tels que les réseaux siamois ou cross-modaux.

Mots-clés : rétropropagation, mémoire, pebble game

1 Introduction

Training for Deep Learning Networks (DNNs) has become a major compute intensive application [1, 2, 3]. Nevertheless, even if training is still in general performed on small clusters of GPU machines, the use of large HPC infrastructures is becoming popular [4, 5], in particular because they offer high bandwidth and low latency networks [6, 7]. Among the approaches that are currently used for DNN training are hyper-parameter tuning and data parallelism. Hyper-parameter tuning is a simple and efficient way to achieve weak scaling, by trying in parallel many hyper-parameters of the networks, and then deciding which one is the most promising and should therefore be kept [8, 9, 10]. This strategy is very efficient at the beginning of the training phase but it does not offer any parallelism once the parameters of the network have been chosen. Another approach to parallelizing DNNs training is to rely on data parallelism. When using data parallelism [11, 12, 4], the model and all the weights are replicated onto all participating nodes and different mini-batches are trained in parallel on the different nodes. In this setting, all participating nodes run the forward and backward phases in parallel, and thus all compute a gradient for all the weights of the network. The synchronization between nodes takes place at this stage, and all partial gradients are gathered and aggregated into a single one through a collective communication such as `Allreduce` and then broadcast to all participating nodes through an `Broadcast`-like operation. Above approach is very efficient as long as high performance implementations of collective communications are available and as long as the network is able support them efficiently [13]. It can even be combined with compression [14] in order to limit the size of the messages and therefore the size of communications.

Another important limitation of hyper-parameter tuning and data parallelism approaches is that they do not help in considering larger models by solving memory issues. Indeed, in both settings, the whole set of weights has to be stored on all participating resources. In general, the memory consumed by the training phase consists in two main parts [15]. The first part is related to the storage of the parameters of the network and is directly related to the size of the model [16]. For instance, the weights of different ResNet models are given in Table 1. For both hyper-parameters and data parallelism approaches, these weights have to be replicated on every node, and in the case of data parallelism, they even have to be aggregated and broadcast on the network after each parallel mini-batch training phase. The second source of memory consumption is due to the storage on each node of all the forward activations (i.e. all the outputs of the different stages of the network) until the backward phase. This part is directly proportional to the size of the mini-batch itself. The size of the overall forward activation memory requirements for different ResNet models and different image sizes are given in Table 2. Overall, the memory need for a given model $\mathcal{M}$ and a given mini-batch size x is given by

$$\text{ModelWeight}_{\mathcal{M}} + x.\text{ActivationWeight}_{\mathcal{M}},$$

where $\text{ModelWeight}_{\mathcal{M}}$ (resp. $\text{ActivationWeight}_{\mathcal{M}}$) denotes the memory needed to store all the parameters (resp. all the forward activations for a unitary batch size) of the model. In general, the size of the mini-batch is determined using the available memory size M and is given by

$$x_{\max} = \frac{M - \text{ModelWeight}_{\mathcal{M}}}{\text{ActivationWeight}_{\mathcal{M}}}.$$

Given a model and provided that $x_{\max} \geq 1$, it is therefore always possible to choose a mini-batch size such that the model fits into the memory. On the other hand, it is also crucial not to use too small batch-sizes, as vector processing units such as GPUs and TPUs [17] are much more efficient when working on larger batch sizes. Therefore, the use of a minimal batch size

ResNet_x	Weight
$x = 18$	175.05
$x = 34$	329.29
$x = 50$	384.85
$x = 101$	674.65
$x = 152$	913.36

Table 1: Memory requirement for each model to keep all weights; the amount is given in megabytes.

	ResNet_x				
Image Width/Height	$x = 18$	$x = 34$	$x = 50$	$x = 101$	$x = 152$
224	230.05	413.00	620.27	1027.21	1410.62
350	309.83	534.96	964.66	1543.72	2139.75
500	449.21	749.73	1570.93	2472.72	3458.50
650	639.07	1039.08	2387.54	3682.00	5161.76
1100	1496.10	2346.95	6073.06	9208.30	12961.96
1500	2628.70	4075.07	10944.42	16515.11	23277.27

Table 2: Memory requirement for each model to keep all weights and activations for the batch_size = 1; the amount is given in megabytes.

and the memory limitation usually forces data scientists to consider smaller models, that can in turn lead to lower accuracy results.

In order to deal with memory issues, model parallelism approach has been advocated in many papers [18] and it can be combined with data parallelism [19]. Model parallelism consist in splitting the network model into several non-overlapping parts, that are distributed over the different resources. The network model can in general be seen as a Directed Acyclic Graph and the different nodes of the DAG are split across the resources. Each time there is an edge between two nodes allocated onto two different nodes, this will induce the communication of the associated forward and backward activations during the training of each mini-batch. Model parallelism contributes to solve memory issues as the model weights are shared among participating nodes, and the problem becomes a general graph partitioning problem [18] where the goal is to balance the work between the different nodes while minimizing the weights of cut edges (i.e. edges whose two extremities are on different resources).

Another complementary approach is the problematic of scheduling a graph with a shared bounded memory. It is also known in the literature as the Register Allocation problem or Pebble Game. Given a bounded numbers of unit-size registers (or memory slots), can we execute the graph while respecting the constraint that to execute a task, all its inputs need to be in a register? Sethi [20] 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 [21], or recently for Serie-Parallel graphs [22].

In this work we are interested in what we denote by backpropagation graphs: given a Directed Acyclic Graph (DEAG) with a single exit node, we build a dual identical graph where the edges have been reversed, and where every nodes of the initial graph is connected to its dual node. The source node of the dual graph and the sink node of the original graph are merged into a single node called *turn* (see Figure 1 for the case of a chain of nodes). These types of graphs have been

widely studied in the context of Automatic Differentiation (AD) [23]. For a given batch size and a given network model and even on a single node without relying on model parallelism strategies, it enables to save 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 $i-$th backward stage. Checkpointing techniques consist in determining in advance which forward 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 [24]. In the case of homogeneous chains, closed form formulas providing the exact position of checkpoints have even been proposed [25], although the algorithmic ingredient of choice to derive optimal checkpointing strategies is Dynamic Programming [25].

The use of checkpointing strategies has recently been advocated for DNN in several papers [26, 27] and a simple periodic checkpointing strategy, non optimal but still efficient implementation is provided in PyTorch [28, 29] for the restricted case of homogeneous chains, whereas DNN models are in general more complicated. While optimal scheduling and checkpointing are still open in the general case, there are some solutions which in some way benefit from the findings in AD checkpointing strategies [27] [26]. However, they are designed to deal with only sequential models, thus making them inappropriate for more sophisticated cases.

In this paper, our goal is to propose a first attempt to derive optimal checkpointing strategies adapted to more general networks and to show how techniques developed in the context of AD can be adapted to DNN networks. More specifically, we concentrate on the particular context of DNN consisting of several independent chains whose results are gathered through the computation of the loss function. This case corresponds to the case of Siamese and Cross Modal Networks. We show that this specific case is still solvable using dynamic programming, but at the price of more sophisticated techniques and higher computational costs.

The rest of the paper is organized as follows. In Section 2, we review related works on checkpointing for Automatic Differentiation and on the interest of multi-chain networks in the context of DNNs. In Section 3, we present our general model and the notations that will be used throughout the paper, and we present a few basic results of the Automatic Differentiation literature. Then, a characterization of optimal solutions is proposed in Section 4 and is later used in Section 5 to find the optimal checkpointing strategy through Dynamic Programming in the case of multiple chains. At last, we present our implementation and simulation results in Section 6, before providing concluding remarks and perspectives in Section 7.

2 Related Work

2.1 Checkpointing for Automatic Differentiation

Adjoints computation is at the core of many scientific applications, from climate and ocean modeling [30] to oil refinery [31]. In addition, the structure of the underlying dependence graph is also at the basis of the backpropagation step of machine learning [32], as noticed in Section 1.

With that many applications come a wide variety of software (Adifor, Adol-C, Tapenade, etc). Each software for adjoint computations needs to implement different steps: the algorithm library based on parameters of the system; the checkpointing techniques, parallelization of the code and overall front-end. Due to the abundance of algorithmic problems arising from these implementations, many of the recent algorithmic advances for adjoint computations are still not implemented in the latest software and suboptimal strategies are used [33]. The *Devito Project* [34] proposes to implement an API to deal with the implementation of checkpointing

strategies (shared-memory parallelism, vectorization etc). They rely on libraries to provide the algorithms to be used.

Storage has been one of the key issue with the computation of adjoints. The computation of adjoints has always been a trade-off between recomputations and memory requirements [35].

When one type of limited memory is available, Grimm et al. [36] showed the optimality of a binomial approach which was later implement by Griewank and Walther [25] under the name Rev. A variant of the problem has received increasingly attention in the recent years with the introduction of a second level of storage of infinite capacity but with access (write and read) costs [37, 24, 38, 39, 33]. Indeed, with the increase in the sizes of the problem, the memory is not sufficient anymore to solve the problems in reasonable time. Hence solutions have started considering the usage of disks to store some of the intermediary data. Several work have considered this problem. Stumm and Walther [37] provided a first heuristic that uses the schedule provided by Rev, where the checkpoints the least used are executed on disk (level 2 storage). Some implementations such as the one provided by Pringle et al. [33] are based on a two level checkpointing strategy: the first pass (forward mode) of the adjoint graph periodically checkpoints to disk (level 2), then the second pass (reverse mode) reads those disk checkpoints one after the other and uses Rev with only memory (level 1) checkpoints. The main parameter (period used for the forward checkpointing) can be chosen by the user. Aupy and al. [24] designed an algorithm, denoted by Disk-Rev, to solve this problem optimally. In a subsequent work, Aupy and Herrmann [38] showed that the optimal solution returned by Disk-Rev is weakly periodic, meaning that the number of forward computations performed between two consecutive checkpoints into the second level of storage is always the same except for a bounded number of them. More recently, they extended this result for a hierarchical memory architecture with an arbitrary number of storage levels [40].

2.2 Multi-Chains networks in the context of DNN

General Deep Neural Networks (DNN) are described as weighted Directed Acyclic Graph (DAG), but at the moment, optimal checkpointing strategies are known in the case of a single homogeneous chain only, that corresponds to the general context of the Automatic Differentiation literature (see Section 2.1), so that they can be applied directly only to very specific networks such as Recurrent Neural Networks (RNN). In this paper, we generalize the single homogeneous chain model by considering multiple chains. More specifically, we consider the case where the different chains can have different lengths, but where individual nodes are equivalent in terms of computing and storage costs. While some popular neural networks consist of only one chain [41, 42, 43], there are several classes of problems that can be modeled as a multi-chain computational graph.

The first class comprises cross-modal embeddings [44, 45]. Such models are 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 [44], having both a dataset of dish images and a dataset of recipes which represents 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 finally a loss. In practice, training such a model often consists in training individually each sub-model for each data source and then to use 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 can significantly decrease memory consumption.

Siamese Neural Network [46, 47, 48] can also directly benefit from our approach. They are widely used for object 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, either it can either correspond to a two-chains computational graph [46, 48] where the loss is computed based on two images, or to a three-chains computational graph, where the triplet loss is applied [49]. Due to memory constraints, most of the CNNs used in these models are not very deep [47]. 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.

3 Framework

In this work, we consider the Register Allocation Problem [20] (*a.k.a.* 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 2.

We start by motivating the problem with the well studied problem of scheduling the backpropagation graph of a linear chain (Section 3.2), before introducing more generally the problem for a join graph (Section 3.4). Finally, we give results from the literature that we use in this work (Section 3.5).

3.1 Platform model and optimization problem

In this work we consider a sequential platform (at all time, all computing resources are dedicated to the same job) with a finite memory $\mathcal{M}$ of size c. Note that we focus on the memory used for storing input and output data of each job, and do not consider the memory required by the actual execution of the job. Indeed, since we consider a sequential platform, we assume that there is some memory dedicated for this.

To execute a job on this platform, at the beginning of the execution, all its inputs need to be stored in memory.

A job is represented as a Directed Acyclic Graph (DAG) $\mathcal{G} = (V, E)$, where each node 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?

The core of the *makespan problem*, is that, while there can be enough memory to execute the graph, the memory may not suffice to execute it in one go. Hence we need to choose which data to store and which nodes should be recomputed.

3.2 The Adjoint Chain problem

Figure 1 depicts the typical task graph that arises in the context of the training phase of DNN when the underlying graph is a single chain. The general framework for the training phase in the context of supervised learning is the following: each input consists of an example x_0 and its classification y_0. x_0 is passed forward through the chain (of length l) using F computations that are called *forward* steps. In the case of a homogeneous chain, all F operations have the execution cost $u_f \in \mathbb{R}^+$. F operations typically consist in a sequence of a linear function (matrix or tensor

operation) followed by a non linear function (typically ReLU). The output of stage i is denoted by x_{i+1} and the output of the DNN is denoted as x_l. Then, the loss function P takes as input (x_k, y_0) and evaluates the quality of the classification x_k with respect to the known expected output y_0. Then, the gradient of the loss function is propagated back through the chain in oder to estimate the sensibility of each parameter to the loss, so as to update the different parameters. During this backward propagation phase, in order to compute $\bar{x}_i$, both the values of $\bar{x}_{i+1}$ and x_i are needed. This creates long term data dependences. For instance, x_1 has to be kept in memory until the end of the whole process, since it will be used to compute $\bar{x}_1$.

If we do not have enough memory to store all the x_i values (named activations), then we can use checkpointing techniques as an alternative. For instance, after having computed x_2, we can remove x_1 from the memory. Then, during the backward propagation phase, at the time to compute $\bar{x}_1$, x_1 will be recomputed from x_0 and then used immediately to compute $\bar{x}_1$. In this context, an optimal checkpointing strategy is a strategy that, given a chain and a memory size (expressed in terms of free memory slots to hold x_i values), computes which values should be kept into memory and when in order to minimize the number of recomputations while fitting into the memory constraint.

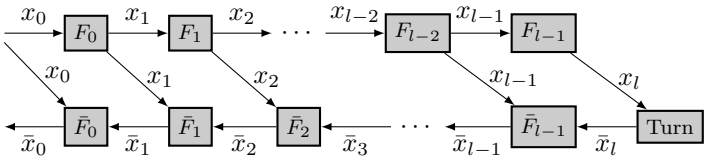


Figure 1: The data dependencies in the Adjoint Computation graph.

Definition 1 (Adjoint Computation [25, 37])**.** An adjoint computation (AC) with l time steps can be described by the following set of equations:

$$F_i(x_i) = x_{i+1} \text{ for } 0 \leq i < l$$
$$\bar{F}_i(x_i, \bar{x}_{i+1}) = \bar{x}_i \text{ for } 0 \leq i < l$$
$$P(x_l) = \bar{x}_l$$

The dependencies between these operations are represented by the graph $\mathcal{G} = (V, E)$ depicted in Figure 1.

Intuitively, the core of the AC problem is the following: after the execution of a forward step F_i, its input is replaced by the corresponding output in the memory, then when it is required to perform the backward step $\bar{F}_i$, if the value x_i is in the memory, then it can be completed immediately, otherwise this value must be recomputed from the closest available checkpointed value. Since F replaces its input value by the corresponding output value, in order to checkpoint a value, it is necessary to duplicate it into memory before applying F.

In accordance to the scheduling literature, we use the term *makespan* to denote the total execution time. Finally, our problem can be written like the following:

Problem 1 ($\textsc{Prob}_{\text{single}}(l, c)$)**.** We want to minimize the makespan of the AC problem with the following parameters:

		Initial state:	Final state:
AC chain:	size l		
Steps:	u_f, u_b, u_t		
Memory:	c	$\mathcal{M} = \{x_0\}$	$\mathcal{M} = \emptyset$

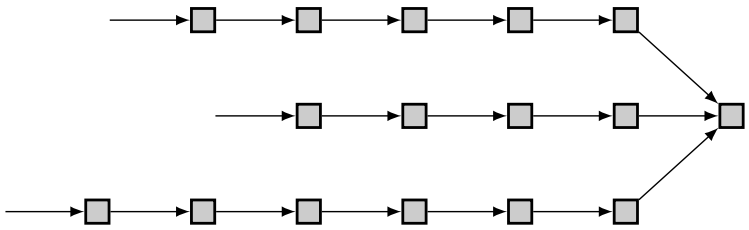


Figure 2: A join graph with 3 branches of respective length 5, 4 and 6.

3.3 Backpropagation graphs

The task graph described in the previous section belongs to a more general class of graphs, *i.e.* the backpropagation graphs. In general, backpropagation is a special sort of transformation of a graph:

Definition 2 (Backpropagation transformation (BP-transform))**.** Given a DAG $\mathcal{G}$ with a single sink node. The *BP-transform* of $\mathcal{G}$ is defined by the following procedure:

1. Build the dual graph $\tilde{\mathcal{G}}$ defined as the same graph where all edges are inversed.
2. For a given node in $\mathcal{G}$, connect its input edges to its dual node in $\tilde{\mathcal{G}}$.
3. Finally, merge the sink node of $\mathcal{G}$ and the source node of $\tilde{\mathcal{G}}$ as a single node, denoted as the *turn*.

Note that the nodes of the initial graph are denoted as *forward steps*, while the nodes 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 $u_b \in \mathbb{R}^+$. 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.

Note that the graph in Figure 1 is the BP-transform of a linear chain.

The key property of the backpropagation graph that motivates this study is

Property 1 (Properties of the BP-graph)**.** *Given a DAG $\mathcal{G}$ with n nodes and a single sink node, without recomputation of nodes, the minimal memory usage to go through the backpropagation graph of $\mathcal{G}$ is $O(n)$.*

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

Another interesting property of backpropagation graphs is that, in machine learning applications, backward steps cannot be recomputed. Indeed, in real life applications, backward steps are performed at the same time as model parameter updates, and some information on the associated forward step is lost. Thus, trying to perform some backward step a second time could yield a different result than it was before.

3.4 The Multiple Adjoint Chains Computation Problem

In this work, we consider transformation of join graphs (Figure 2). Join graphs are used by several deep learning models such as Siamese Neural Network [46, 47, 48] or Cross-modal embeddings [44, 45].

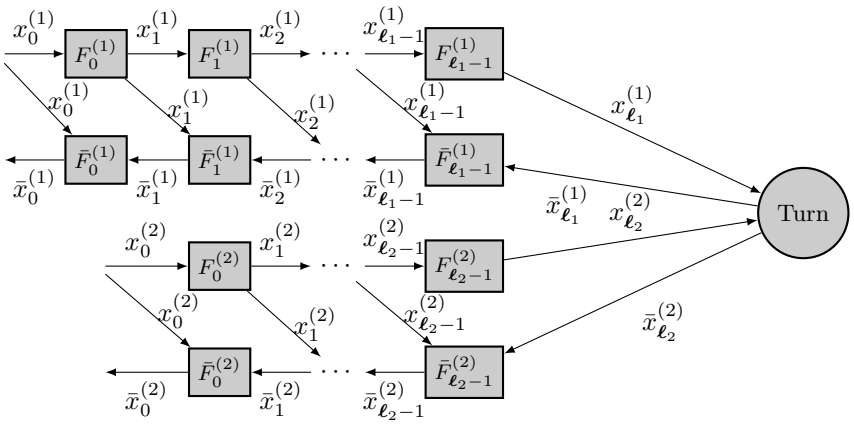


Figure 3: Data dependencies in the BP-transform of a k-Join problem with two chains.

Definition 3 (BP-transform of a k-Join)**.** 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:

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

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.

In the BP-transform of a k-Join model, the last forward value of each chain is required to compute task P (the loss function) and to start the backward propagation phase. Before the computation of the loss function (resp. after the computation of the loss function), all forward (resp. backward) operations on the different chains can be performed independently, except that all chains share the same set of storage slots.

We can now present the optimization problem under consideration:

Problem 2 ($\textsc{Prob}_{\text{multi}}(\boldsymbol{\ell}, c)$)**.** Given $k \in \mathbb{N}$, $c \in \mathbb{N}$, and $\boldsymbol{\ell} \in \mathbb{N}^k$, minimize the makespan of the problem of BP-transform of a k-Join, where the different costs and memory states are given in the following table.

		Initial state	Final state
Join size:	$\boldsymbol{\ell} = (\boldsymbol{\ell}_1, \dots, \boldsymbol{\ell}_k)$		
Steps:	u_f, u_b, u_t		
Memory:	c	$\mathcal{M} = \{x_0^{(1)}, \dots, x_0^{(k)}\}$	$\mathcal{M} = \varnothing$

In order to solve $\textsc{Prob}_{\text{multi}}(\boldsymbol{\ell}, c)$, we introduce the auxiliary problem $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ which takes as additional input a binary vector $\boldsymbol{b}$ such that, $\forall i \in \{1, \dots, k\}$:

$$\boldsymbol{b}_i = \begin{cases} 1 & \text{if the result of the last backward step for the chain } i, \text{ i.e. } x_0^{(i)}, \text{ should be kept,} \\ 0 & \text{otherwise,} \end{cases}$$

Problem 3 ($\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$)**.** Given $k \in \mathbb{N}$, $c \in \mathbb{N}$, $\boldsymbol{\ell} \in \mathbb{N}^k$, and $\boldsymbol{b} \in \{0,1\}^k$, minimize the makespan of the problem of BP-transform of a k-Join with the following constraints:

		Initial state	Final state
Join size:	$\boldsymbol{\ell} = (\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_k)$		
Steps:	u_f, u_b, u_t		
Memory:	c	$\mathcal{M} = \{x_0^{(1)}, \ldots, x_0^{(k)}\}$	$\mathcal{M} = \{\bar{x}_0^{(i)},\ \forall i\ s.t.\ \boldsymbol{b}_i = 1\}$

Note that $\textsc{Prob}_{\text{multi}}(\boldsymbol{\ell}, c) = \textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \vec{0})$.

3.5 Previous Results for Single Adjoint Chain Computation Problem

In this work we review the results from the literature for a slightly different version of $\textsc{Prob}_{\text{single}}(l, c)$. We define $\textsc{Prob}(l, c)$ as the problem consisting of minimizing the makespan of the adjoint chain of size l depicted in in Figure 4 with c memory slots, where x_0 and $\bar{x}_{l+1}$ are initially stored into the memory, and where at the end of the computation $\bar{x}_0$ should be stored in memory.

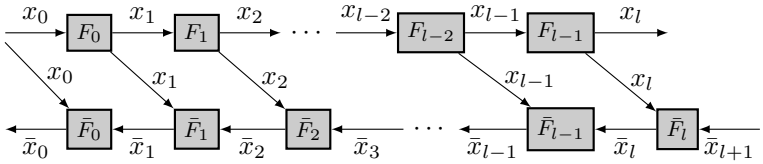


Figure 4: Data dependencies in the AC graph of size l in sub-problem $\textsc{Prob}(l, c)$.

Problem 4 ($\textsc{Prob}(l, c)$)**.** Given $l \in \mathbb{N}$, and $c \in \mathbb{N}$, minimize the makespan of the AC problem with the following parameters:

		Initial state:	Final state:
AC chain:	size l		
Steps:	u_f, u_b		
Memory:	c	$\mathcal{M} = \{x_0, \bar{x}_{l+1}\}$	$\mathcal{M} = \{\bar{x}_0\}$

Definition 4 ($\text{Opt}_0\,(l, c)$)**.**
Given $l \in \mathbb{N}$, and $c \in \mathbb{N}$, $\text{Opt}_0\,(l, c)$ denotes the execution time of an optimal solution to $\textsc{Prob}(l, c)$.

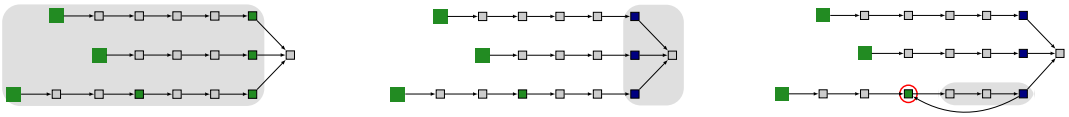
Theorem 1 ([24])**.** *$Opt_0\,(l, c)$ can be computed with the following dynamic programming*

$$\forall c \geq 2, \quad Opt_0\,(0, c) = u_b \tag{1}$$

$$\forall l > 0, \quad Opt_0\,(l, 3) = \frac{l(l+1)}{2} u_f + (l+1) u_b \tag{2}$$

$$\forall l > 0, c > 3, \quad Opt_0\,(l, c) = \min_{1 \leq i < l} \{i u_f + Opt_0\,(l - i, c - 1) + Opt_0\,(i - 1, c)\} \tag{3}$$

The minimal amount of memory slots required to reverse an AC graph of size larger than 1 is equal to 3, in order to keep (i) the initial value x_0, (ii) the current forward value and (iii) the current backward value. Note that the initialization is not exactly the same as the one in our previous work [24] because in our previous work, two of the memory slots were called *buffer* and were not counted in the total memory, but these formulations are equivalent. This problem has been heavily studied in the community of automatic differentiation.



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

Figure 5: The three main phases of the algorithm. Green blocks correspond to the storage of "forward" data, blue blocks to storage of "backward" data.

4 Characterization of optimal solutions

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 (Section 4.2). We call these solutions *canonical* optimal solutions. Then, in Section 5, we show that using this restriction, we can compute a canonical optimal solutions (which is optimal) through a dynamic program.

Let us first notice that an algorithm for $\textsc{Prob}_{\text{multi}}(\boldsymbol{\ell}, c)$ can be decomposed into three different phases (see Figure 5):

- **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.

4.1 Motivating example

Let us start by presenting a toy example and its associated optimal solution in order to introduce the main ingredients of our algorithm and the main lemmas and theorems that prove its correctness. We will consider a network with 3 branches of respective lengths 4, 10 and 12 ($\boldsymbol{\ell} = (4, 10, 12)$). An instance of $\textsc{Prob}_{\text{multi}}(\boldsymbol{\ell}, c)$ is also characterized by the size of the available memory c, expressed in terms of number of forward or backward values that it can host. The other parameters that describe the problem are u_f, u_b, u_t, that are all set to 1 in the toy example.

Note that the above defined instance of $\textsc{Prob}_{\text{multi}}(\boldsymbol{\ell}, c)$ can be transformed into an instance of $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$, if we set $\boldsymbol{b} = (0, 0, 0)$, since in $\textsc{Prob}_{\text{multi}}(\boldsymbol{\ell}, c)$, it is assumed that $\bar{x}_0^{(j)}$ are not kept into memory after the execution. Finding the schedule that achieves minimum makespan for 3 branches of lengths $\boldsymbol{\ell} = (4, 10, 12)$ and unitary computational costs while using at most $c = 9$ memory slots therefore corresponds to solving

$$\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b}) \text{ with } \boldsymbol{\ell} = (4, 10, 12), c = 9, u_f = 1, u_b = 1, u_t = 1, \boldsymbol{b} = (0, 0, 0).$$

In order to solve this instance, we rely on dynamic programming and express the solution of $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ as the minimum between "smaller" instances of $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$. In order to do this, the key ingredient is Theorem 3 and more specifically Equation 5. Equation 5 says that $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ can be obtained by considering all possible positions of the first checkpoint

(denoted as i) taken during the forward phase (before the computation of the turn) on the branch that will finish last its backward propagation (denoted as m). To establish this result, we rely on the characterization of a class of optimal schedules that is defined in Definition 8.

In our specific toy example, the optimal (i, m) pair is $(9, 3)$ so that we checkpoint $x_9^{(3)}$. The solution for the toy example is depicted on Figure 6. In this picture, time is depicted on the $y-$axis and the position on the chain is depicted on the $x-$axis. The chains of respective initial lengths 4, 10 and 12 are depicted using respectively colors blue, green and red. The first 9 time units are used to perform the first 9 forward computations on the red branch, and the red branch will end up the computation with operation REV(8,9).

We are now left with two problems.

1. The first problem is an instance of $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ with parameters $(\boldsymbol{\ell} = (3, 4, 10), c = 8, u_f = 1, u_b = 1, u_t = 1, \boldsymbol{b} = (1, 0, 0))$. Note that throughout the paper, $\boldsymbol{\ell}$ is always a non decreasing vector, so that the rest of the chain of length 12, whose length is now 3, is the first element of $\boldsymbol{\ell}$. Since one memory slot is dedicated to store $x_9^{(3)}$, the number of available memory slots is now $c = 9 - 1 = 8$. At last, $\boldsymbol{b}$ also changed since it is now (1,0,0). This changes expresses the fact that the initial backward value on the first branch must now be kept in memory. Indeed, since the first chain corresponds to the truncated chain, this value is needed in order to continue the backward propagation on the red branch. This observation is the key point to explain the introduction of $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$.

2. The second problem is an instance of the classical single chain problem as previously described in the Automatic Differentiation literature and consists in processing the first 9 elements of the red chain, with all available memory slots since it will be processed last after all other branches.

In Equation 5, we can recognize the different terms corresponding to the first $i = 9$ forward steps, to the first problem $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ with parameters $(\boldsymbol{\ell} = (3, 4, 10), c = 8, u_f = 1, u_b = 1, u_t = 1, \boldsymbol{b} = (1, 0, 0))$ (with $m = 1$) and to the second problem $\text{Opt}_0\left(i - 1, c - \sum_{j \neq m} \boldsymbol{b}_j\right)$ (with all $\boldsymbol{b}_j$s being 0 except on the checkpointed branch $m = 1$).

To solve $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ with parameters $(\boldsymbol{\ell} = (3, 4, 10), c = 8, u_f = 1, u_b = 1, u_t = 1, \boldsymbol{b} = (1, 0, 0))$, we again rely on Equation 5, and we decide to checkpoint the 7th forward value on the green branch. This again leaves us with two problems, $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ with parameters $(\boldsymbol{\ell} = (3, 3, 4), c = 7, u_f = 1, u_b = 1, u_t = 1, \boldsymbol{b} = (1, 1, 0))$ and $\text{Opt}_0(6, 7 = 8 - 1)$. For $\text{Opt}_0(l, c)$, the number of available checkpoints is 7 and not 8, because we need to keep both the initial forward and one backward value for the red chain, as expressed by the fact that its associated b value is 1.

To solve $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ with parameters $(\boldsymbol{\ell} = (3, 3, 4), c = 7, u_f = 1, u_b = 1, u_t = 1, \boldsymbol{b} = (1, 1, 0))$, we rely on a different strategy, based on the observation that 7 is the minimal number of memory slots to process the 3 chains. Indeed, immediately after the computation of the turn, 3 slots will be occupied by the initial values of the different chains and 3 slots will be occupied by the backward values produced by the turn. Another slot is necessary to process any forward value on any chain, that can be seen as a buffer for forward computations. In general, Lemma 4 provides the general formula to find the minimum number of memory slots necessary to process multiple chains defined by $\boldsymbol{\ell}$ and $\boldsymbol{b}$.

In the case where the number of memory slots is minimal, we have less freedom. Indeed, we can observe that we cannot add checkpoints before the turn, so that the question becomes how to schedule (after the turn) a set of single chains with different values of $\boldsymbol{\ell}$ and $\boldsymbol{b}$. After the turn, all single chains are independent, but they still share the memory slots. Once it is processed, the initial value of the chain, that was stored in memory can always be discarded. In addition,

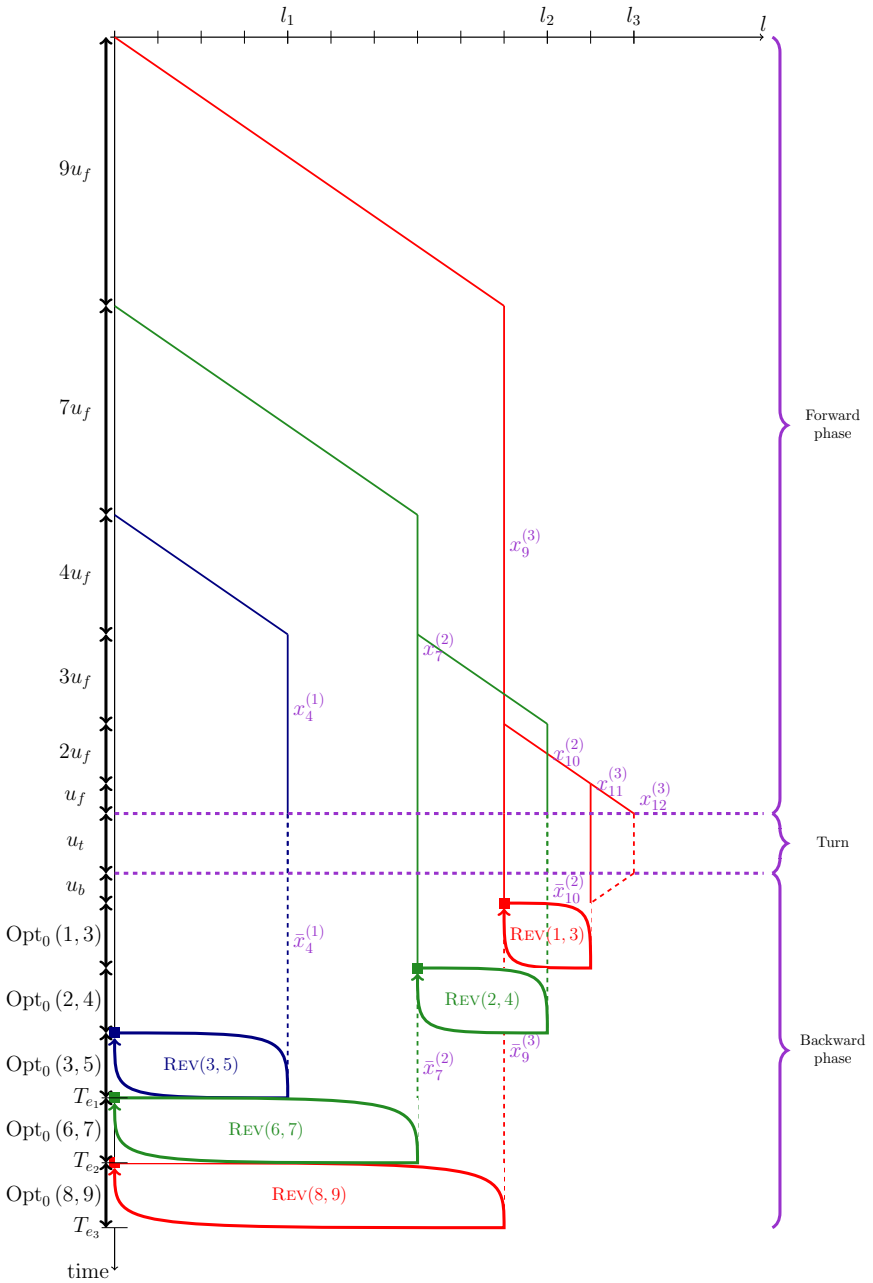


Figure 6: Toy example, an optimal schedule for $\text{Prob}_{\text{b-multi}}(\boldsymbol{\ell} = (4, 10, 12), c = 9, \boldsymbol{b} = (0, 0, 0))$

	Operation	Input	Output	Cost
$F_i^{(j)}$	Executes one forward step for $i \in 1,\dots,k$ and $j \in 0,\dots,\boldsymbol{\ell}_i - 1$	$\{x_i^{(j)}\}$	$\{x_{i+1}^{(j)}\}$	u_f
$\bar{F}_i^{(j)}$	Executes one backward step for $i \in \{1,\dots,k\}$ and $j \in \{0,\dots,\boldsymbol{\ell}_i - 1\}$	$\{\bar{x}_{i+1}^{(j)}, x_i^{(j)}\}$	$\{\bar{x}_i^{(j)}, \varnothing\}$	u_b
Turn	Computation of the loss functions using the last forward value of each chain	$\{x_{\boldsymbol{\ell}_1}^{(1)},\dots,x_{\boldsymbol{\ell}_k}^{(k)}\}$	$\{\bar{x}_{\boldsymbol{\ell}_1}^{(1)},\dots,\bar{x}_{\boldsymbol{\ell}_k}^{(k)}\}$	u_t
$S_i^{(j)}$	Replicates $x_i^{(j)}$ into the memory as a checkpoint for $i \in 1,\dots,k$ and $j \in 0,\dots,\boldsymbol{\ell}_i - 1$	$\{x_i^{(j)}, \varnothing\}$	$\{x_i^{(j)}, x_i^{(j)}\}$	0
$D_i^{(j)}$	Discard $x_i^{(j)}$ from memory	$\{x_i^{(j)}\}$	$\varnothing$	0
$\bar{D}^{(j)}$	Discard $\bar{x}_0^{(j)}$ from memory, so after that there are no any elements of j chain in the memory	$\{\bar{x}_0^{(j)}\}$	$\varnothing$	0

Table 3: Operations performed by a schedule

depending on the value of $\boldsymbol{b}$, the last backward value has to be kept into memory or not. In the case of the toy example, processing the blue chain, where no intermediate forward value has been stored, releases 2 memory slots while the red and green chains (both of length 3), release only one memory slot. On the toy example, the red-green-blue ordering is optimal.

Overall, solving this toy example requires to use all the main results presented in this paper, and whose proofs can be found in the following sections. More specifically, Section 4.2 presents the characterization of the optimal solution that is later used in Theorem 3 proved in Section 5.2 to build the general dynamic programming formulation.

4.2 Canonical form of optimal solutions

In this section, we show that there exist optimal solutions to $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ that follow a canonical form. In Section 5, we later use this characterization to compute an optimal solution. We start by giving a formal description of a valid schedule, before showing our result.

4.2.1 Valid Schedule

Throughout the text, we represent a schedule by a sequence of operations, where the set of operations ($F_i^{(j)}$, $\bar{F}_i^{(j)}$, P, $S_i^{(j)}$, $D_i^{(j)}$, $\bar{D}^{(j)}$) is defined in Table 3. Table 3 defines both the set of operations and the transformation of the memory with each operation. Initially, we assume that all $x_0^{(j)}$ values are stored into memory slots.

Definition 5 (Valid Schedule)**.** A valid schedule is such that

1. for each operation in the sequence, its input values are stored into memory slots when it is processed;
2. at any instant in time, the overall memory capacity c is not exceeded;
3. at the end of the schedule, all $\bar{x}_0^{(j)}$ values must have been computed and all $\bar{x}_0^{(j)}$ values, where $\boldsymbol{b}_j = 1$ must be stored into memory.

Definition 6 (Makespan of a Schedule)**.** We define the makespan of a valid schedule $\mathcal{S}$ as

$$k_F u_f + k_B u_b + k_T u_t,$$

where k_F denotes the number of $F_i^{(j)}$ operations in $\mathcal{S}$, k_B denotes the number of $\bar{F}_i^{(j)}$ operations in $\mathcal{S}$ and k_T denotes the number of Turn operations in $\mathcal{S}$ (we prove in the remainder of this section that in any optimal solution, $k_B = \sum \boldsymbol{\ell}_i$ and $k_T = 1$).

Therefore, our model takes into account computational costs and limited memory capacity, but it assumes that memory accesses, both for reading and writing are cheap and can be neglected.

Finally, we introduce the specific makespan of optimal schedules:

Definition 7 ($\mathrm{Opt}_{\text{multi}}(\boldsymbol{\ell}, c)$, $\mathrm{Opt}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$)**.**
Given $\boldsymbol{\ell} \in \mathbb{N}^k$, $c \in \mathbb{N}$, and $\boldsymbol{b} \in \{0,1\}^k$, let $\mathrm{Opt}_{\text{multi}}(\boldsymbol{\ell}, c)$ denote the makespan of an optimal solution to $\textsc{Prob}_{\text{multi}}(\boldsymbol{\ell}, c)$, and let $\mathrm{Opt}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ denote the makespan of an optimal solution to $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$

4.2.2 Canonical solutions

In the following section, we show that we can restrict the search space of an optimal solution to the space of what we call *Canonical Solutions*.

Notation 1. Given $\boldsymbol{\ell} = (\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_k)$, we denote by $\boldsymbol{\ell}_{[i \leftarrow x]}$ the vector $\boldsymbol{\ell}$ where the i-th element is replaced by the value x:

$$\boldsymbol{\ell}_{[i \leftarrow x]} = (\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_{i-1}, x, \boldsymbol{\ell}_{i+1}, \ldots, \boldsymbol{\ell}_k)$$

Definition 8 (Canonical Solutions)**.** We say that a solution $\mathcal{S}$ to $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ is in a canonical form if there exists j, m, $\mathcal{S}'$ a canonical solution to $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}_{[m \leftarrow \boldsymbol{\ell}_m - j]}, c - 1, \boldsymbol{b}_{[m \leftarrow 1]})$, and $\tilde{\mathcal{S}}$ a solution to $\textsc{Prob}(j-1, c - \sum_{j \neq m} \boldsymbol{b}_j)$, such as $\mathcal{S}$ has the following form:

1. From the input $x_0^{(m)}$ written on memory, j forward steps are performed on the branch m;
2. The output after those steps is replicated into memory;
3. $\mathcal{S}'$ is performed (i.e. a canonical solution for the smaller problem where all branches' size are the same except for branch m which is now smaller by j forward steps with $c-1$ memory slots).
4. From the input $x_0^{(m)}$ written on memory, $\tilde{\mathcal{S}}$ is performed (the j subsequent steps on branch m are *backpropagated* with all available checkpoints).
5. If $\boldsymbol{b}_m = 0$, $\bar{x}_0^{(m)}$ is discarded.

The example provided in Figure 6 is in canonical form.

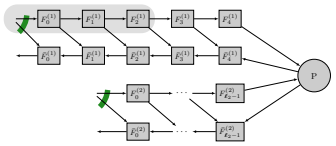
In the following lemmas, we restrict the search space for an optimal solution, by proving that, using basic transformations, it is possible to transform an optimal solution into another optimal solution with stronger structural properties.

Notation 2. We use the following notations to describe the execution of a solution:

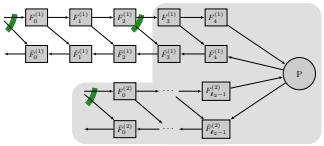
- We define $\mathcal{M}_T$ the state of the memory right before the turn:

$$\mathcal{M}_T = \{x_{\sigma_1(1)}^{(1)}, \ldots, x_{\sigma_1(n_1)}^{(1)}, \ldots, x_{\sigma_k(1)}^{(k)}, \ldots, x_{\sigma_k(n_k)}^{(k)}\} \tag{4}$$

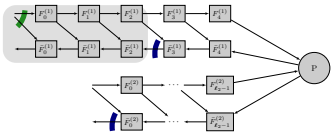
  where n_i is the number of data from branch i in $\mathcal{M}_T$. Without loss of generality, we assume that for all i and j, $\sigma_i(j) < \sigma_i(j+1)$. We have necessarily the following trivial properties:



(a) The solution starts from a state where all inputs are stored; then it performs some forward steps on one of the branches.



(b) After this, it stores the current state and solves a smaller instance of $\textsc{Prob}_{\text{b-multi}}()$



(c) Finally, some backward data are stored (depending on $\boldsymbol{b}$), and an instance of $\textsc{Prob}(l, c)$ is solved.

Figure 7: Graphical execution of a canonical solution

1. $x^{(i)}_{\sigma_i(1)} = x^{(i)}_0$ (those inputs need to be kept until the end of the execution of the branch).
2. $x^{(i)}_{\sigma_i(n_i)} = x^{(i)}_{l_i}$ (those inputs are in memory since they are needed to perform the turn).

- For $i \in \{1, \dots, k\}$ and $j \in \{1, \dots, n_i - 1\}$, we call *the* $(i,j)^{th}$ *Forward Segment*, $FS(i,j)$, the sequence of operations:

$$FS(i,j) = \text{Forward Operations } F^{(i)}_{\sigma_i(j)} \text{ to } F^{(i)}_{\sigma_i(j+1)-1}\text{; Store } x^{(i)}_{\sigma_i(j+1)}$$

- For $i \in \{1, \dots, k\}$ and $j \in \{1, \dots, n_i - 1\}$, we call *the* $(i,j)^{th}$ *Backward Segment*, $BS(i,j)$, the sequence of all operations during the Backward Phase between the first time $x^{(i)}_{\sigma_i(j)}$ is read/used, and the computation of $\bar{F}^{(i)}_{\sigma_i(j)}$

We now provide several results based on the different phases of the execution of an algorithm (forward phase, turn, backward phase).

Forward phase properties We start with a simple lemma on the execution of the forward phase.

Lemma 1 (Optimal Forward Phase to $\mathcal{M}_T$)**.** *Given $\mathcal{M}_T$ as defined in Eq.* (4)*. Any permutation of the forward segments satisfying: for all i, j: $FS(i,j)$ is executed before $FS(i,j+1)$ is a valid forward phase of minimum execution time that leads to the memory state $\mathcal{M}_T$.*

Proof. The result is straightforward: thanks to the constraint on the permutation, the schedule is valid. In addition, during such a forward phase, all forward operations from all branches are executed exactly once. □

Backward phase properties We now provide some properties of the backward phase.

Lemma 2 (Checkpoint Persistence)**.** *Given $\mathcal{M}_T$ as defined in Eq.* (4)*. For all i and j, in an optimal solution, $x^{(i)}_{\sigma_i(j)}$ is not discarded from memory until the end of $BS(i,j)$.*

Proof. This result can be shown by contradiction. Assume $x^{(i)}_{\sigma_i(j)}$ is discarded before the execution of $\bar{F}^{(i)}_{\sigma_i(j)}$. This means that any usage of $x^{(i)}_{\sigma_i(j)}$ during the backward phase is followed by $F^{(i)}_{\sigma_i(j)}$ to obtain $x^{(i)}_{\sigma_i(j)+1}$. Hence storing $x^{(i)}_{\sigma_i(j)} + 1$ instead of $x^{(i)}_{\sigma_i(j)}$ during the forward phase would have reduced the total number of re-executions during the backward phase, contradicting the optimality. □

Lemma 3 (Non-overlap of backward segments)**.** *There exists an optimal solution such that no backward segment overlaps with the last backward segment.*

Proof. Given an optimal solution, let us assume that $BS(i_1, 1)$, the backward segment that finishes last, overlaps with other segments. Let $BS(i_2, j_2)$ the backward segment that finishes one before last. Then $BS(i_1, 1)$ and $BS(i_2, j_2)$ overlap.

If we consider the solution almost identical but where we move all operations of type: (i) $F_j^{(i_1)}$ or (ii) $\bar{F}_j^{(i_1)}$ for $j \leq \sigma(j_1) - 1$ and their corresponding memory movements to after the execution of $\bar{F}_{\sigma(j_2)}^{(i_2)}$, in the same order. Then one can verify that:

- No more segment overlaps with $BS(i_1, j_1)$;
- The schedule for previous segments are still valid, indeed:
  - backward segments on branches other than i_1 are not influenced by this move, except in term of memory available (which can only increase thanks to this move, hence potentially improving the execution time);
  - backward segments on branch i_1 are not influenced by this move thanks to Lemma 2, since until the computation of $\bar{F}_{\sigma_{i_1}(2)}^{(i_1)}$, they can use $x_{\sigma_{i_1}(2)}^{(i_1)}$ which is available.
- And that finally, it can only increase the memory available for $BS(i_1, j_1)$, i.e. the execution is still valid and does not increase in time.

In the end, this shows that either the solution was not optimal, or we can transform it in an optimal solution with identical forward phase and turn, but with strictly less backward segment overlapping. □

Theorem 2. *There exists an optimal solution under canonical form.*

The proof is a corollary of the previous results.

Proof. The result can be shown by recurrence. The initialization when $\boldsymbol{\ell} = \vec{0}$ is trivial.

For $\boldsymbol{\ell} > \vec{0}$, given an optimal solution $\mathcal{S}$ to $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$. Let $\mathcal{M}_T$ the state of its memory before the turn. According to Lemma 3 we can assume w.l.o.g that it is a solution where the last backward segment, $BS(i, 1)$, does not overlap with any other backward segment. In addition, according to Lemma 1, we can also assume that its forward phase starts with $FS(i, 1)$.

Finally, it suffices to see that

- at the beginning of $BS(i, 1)$, the last backward data that has been able to be computed on chain i is exactly $\bar{x}_2^{(i)}$;
- there is no other type of computation (because of the no overlap property during $BS(i, 1)$);
- at the beginning of $BS(i, 1)$ exactly $c - \sum_{j \neq i} \boldsymbol{b}_j$ checkpoints are available.

Hence $BS(i, 1)$ is exactly a schedule that solves $\textsc{Prob}((\sigma_i(2) - \sigma_i(1) - 1, c - \sum_{j \neq i} \boldsymbol{b}_j)$

We now have an optimal solution that has the following shape:

1. from the input $x_0^{(i)}$ written on memory, $\sigma_i(2)$ forward steps are performed on the branch i;
2. the output after those steps is replicated into memory ($BS(i, 1)$);
3. schedule $\mathcal{S}'$ is performed;

4. from the input $x_0^{(i)}$ written on memory, we perform $BS(i,1)$ a solution to $\textsc{Prob}((\sigma_i(2) - \sigma_i(1) - 1, c - \sum_{j \neq i} \boldsymbol{b}_j)$.

5. If $\boldsymbol{b}_m = 0$, $\bar{x}_0^{(m)}$ is discarded.

Note that by construction, $\mathcal{S}'$ is an optimal solution to $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}_{[1 \leftarrow \boldsymbol{\ell}_i - j]}, c-1, \boldsymbol{b}_{[i \leftarrow 1]})$, and by recurrence hypothesis, we can transform it into a canonical optimal solution to the same problem, showing the result. □

5 Optimal Solution for Problem $\textbf{Prob}_{\textbf{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$

5.1 Minimal memory requirement

In this section, we first establish in Lemma 4 the formula to compute the minimal number of memory slots in order to complete multiple chains with parameters $\boldsymbol{\ell}$ et $\boldsymbol{b}$. This result is later used to solve $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ in Theorem 3, since it is used to initialize the dynamic program.

Lemma 4 (Minimal required memory)**.** *The minimal amount of memory slots to solve* $\textsc{Prob}_{b\text{-}multi}(\boldsymbol{\ell}, c, \boldsymbol{b})$*, is*

$$c_{min}(\boldsymbol{\ell}) = \begin{cases} k & \text{if } \boldsymbol{\ell} = \vec{0} \\ L & \text{if } (\exists i : \boldsymbol{\ell}_i = 1) \text{ or } (\exists i : \boldsymbol{\ell}_j = 0 \text{ and } \boldsymbol{b}_i = 0)\,, \\ L+1 & \text{otherwise.} \end{cases}$$

where $L = k + \sum_{i=1}^{k} \mathbb{I}[\boldsymbol{\ell}_i \neq 0]$.

Proof. The case where $\boldsymbol{\ell} = \vec{0}$ is because we simply need to perform the turn which has a memory peak of k: its k inputs are transformed into k outputs.

For the general case, we decompose the memory peak as a function of the phase:

Forward phase: During the forward phase, all initial inputs of all branches need to be stored ($x_0^{(i)}$ values in Figure 3) because they are needed for the computation of $\bar{F}_0^{(i)}$. In addition, at the end of the forward phase, all inputs of the turn ($x_{\boldsymbol{\ell}_i}^{(i)}$ values) are also stored (see Figure 5a). Hence, the forward phase needs at least:

$$L = k + \sum_{i=1}^{k} \mathbb{I}[\boldsymbol{\ell}_i \neq 0]$$

(we use only one slot when $x_0^{(i)} = x_{\boldsymbol{\ell}_i}^{(i)}$).

Note that one can verify that anytime in the forward phase we do not need more storage space than this: during the execution of $F_j^{(i)}$ we can assume that its input uses the storage slot of $x_{\boldsymbol{\ell}_i}^{(i)}$ (unused at first) and replaces it by its output. Hence L is enough for the forward phase.

Turn: The turn does not increase the storage needed from what was used before, indeed it simply replaces each of its input values (already stored by hypothesis) by the corresponding output value. Finally at the end of the turn, we release the checkpoints from branches that are finished, i.e. those where $\boldsymbol{\ell}_i = 0$ and $\boldsymbol{b}_i = 0$. Hence, $\tilde{L} = L - \sum_{i=1}^{k} \mathbb{I}[\boldsymbol{\ell}_i = 0 \ \&\ \boldsymbol{b}_i = 0]$ memory slots are used.

Backward phase: Finally, consider the solution which executed each backward branch one after the other using the strategy of Theorem 1 with $\boldsymbol{\ell}_i - 1$. The peak is when we compute the first backward data: one uses $\tilde{L} - 2$ checkpoints for all branches but the current one, and needs 3 additional memory slots (resp. 2 if $\boldsymbol{\ell}_i - 1 = 0$) to execute the branch under consideration.

Hence if $\tilde{L} < L$ or $\exists\ i,\ \boldsymbol{\ell}_i > 1$, the peak is met during the forward phase with L storage slots.

In the general case ($\tilde{L} = L$ and for all i, $\boldsymbol{\ell}_i > 1$), it is not possible to free memory, and as no backward steps can be performed because of lack of inputs., the minimal required memory is $L + 1$. Hence this shows the result. □

5.2 Optimal solution

Notation 3. We denote by $\mathbf{e}_i$ the vector which every element is null except the i-th one that is equal to 1. Note that, with this notation, the size of $\mathbf{e}_i$ is ambiguous but it will be the same as the other vectors ($\boldsymbol{\ell}$ and $\boldsymbol{b}$) in the rest of this paper

We are now able to establish the following theorem on the execution time of an optimal solution of $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ (and hence $\textsc{Prob}_{\text{multi}}(\boldsymbol{\ell}, c)$):

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

$$\begin{aligned} Opt_{b\text{-}multi}(\boldsymbol{\ell}, c, \boldsymbol{b}) &= \infty && \text{if } c < c_{\min}(\boldsymbol{\ell}), \\ Opt_{b\text{-}multi}\left(\vec{0}, c, \boldsymbol{b}\right) &= u_t && \text{if } c \geq k, \\ Opt_{b\text{-}multi}(\boldsymbol{e}_i, c, \boldsymbol{b}) &= u_f + u_t + u_b && \forall i,\ \text{if } c \geq k+1, \end{aligned}$$

For other cases:

$$Opt_{b\text{-}multi}(\boldsymbol{\ell}, c, \boldsymbol{b}) = \min_{\substack{1 \leq m \leq k \\ 0 < i \leq \boldsymbol{\ell}_m}} \left[i u_f + Opt_{b\text{-}multi}\left(\boldsymbol{\ell}_{[m \leftarrow \boldsymbol{\ell}_m - i]}, c-1, \boldsymbol{b}_{[m \leftarrow 1]}\right) + Opt_0\left(i-1, c - \sum_{j \neq m} \boldsymbol{b}_j\right) \right] \tag{5}$$

The complexity to compute $Opt_{b\text{-}multi}(\boldsymbol{\ell}, c, \boldsymbol{b})$ is $O\left(c \cdot k \cdot \left(2 \max_{i=1}^{k} \boldsymbol{\ell}_i\right)^{k+1}\right)$.

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

Proof. We have shown in the previous section that the optimal canonical solution is an optimal solution. Hence here is suffices to show:

- That there exists a solution with execution time $\text{Opt}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$;
- That $\text{Opt}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ is better than the execution time of any canonical solution for identical parameters.

We remind the recursive shape of a canonical solution $\mathcal{S}$ for $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$:

1. Execute i forward steps on branch $\mathcal{B}_m$;
2. Replicate the output data;
3. A canonical solution $\mathcal{S}'$ for $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}_{[m \leftarrow \boldsymbol{\ell}_m - i]}, c-1, \boldsymbol{b}_{[m \leftarrow 1]})$ (the output of the shorter branch $\mathcal{B}_m$ has to be in memory for the next step);

4. Solve $\textsc{Prob}(i-1, \tilde{c})$ on the last steps of branch $\mathcal{B}_m$ using all available checkpoints $\tilde{c}$.

If we denote by $T(\mathcal{S})$ (resp. $T(\mathcal{S}')$, and $T(B[i-1,\tilde{c}])$ the execution time of $\mathcal{S}$ (resp. $\mathcal{S}'$, and the last step of the procedure), then we have:

$$T(\mathcal{S}) = iu_f + T(\mathcal{S}') + T(B[i-1,\tilde{c}])$$

Existence of a solution of time $\mathbf{Opt_{b\text{-}multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ Based on this, one can see that we can indeed reconstruct recursively a solution based on Equation (5). The initialization when $\boldsymbol{\ell} = \vec{0}$ or $\boldsymbol{\ell} = \mathbf{e}_i$) is trivial.

Then, given i, m such that

$$\mathrm{Opt}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b}) = \left[iu_f + \mathrm{Opt}_{\text{b-multi}}\left(\boldsymbol{\ell}_{[m\leftarrow \boldsymbol{\ell}_m - i]}, c-1, \boldsymbol{b}_{[m\leftarrow 1]}\right) + \mathrm{Opt}_0\left(i-1, c - \sum_{j\neq m} \boldsymbol{b}_j\right)\right]$$

By recurrence hypothesis, $\mathrm{Opt}_{\text{b-multi}}\left(\boldsymbol{\ell}_{[m\leftarrow \boldsymbol{\ell}_m - i]}, c-1, \boldsymbol{b}_{[m\leftarrow 1]}\right)$ is the execution time of a solution $\mathcal{S}'$ to $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}_{[m\leftarrow \boldsymbol{\ell}_m - i]}, c-1, \boldsymbol{b}_{[m\leftarrow 1]})$, and $\mathrm{Opt}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ corresponds to the execution time of the canonical schedule:

- Perform i steps on branch $\mathcal{B}_m$;
- Perform $\mathcal{S}'$;
- Use $\textsc{Rev}$ to backpropagate optimally the last $i-1$ steps of $\mathcal{B}_m$.

Minimality of $\mathbf{Opt_{b\text{-}multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ amongst optimal solutions Similarly we show this result by recurrence. We only consider valid solutions, hence when the memory requirement is met.

The initialization is performed either for $\boldsymbol{\ell} = \vec{0}$ or $\boldsymbol{\ell} = \mathbf{e}_i$. For both those cases there is only one possible canonical solution and its execution time is exactly $\mathrm{Opt}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$. Assume now that for all $\boldsymbol{b}$, and $\boldsymbol{\ell} < \boldsymbol{\ell}'$, $c \geq c_{min}(\boldsymbol{\ell})$ any canonical solution $\mathcal{S}$ to $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$ is such that $T(\mathcal{S}) \geq \mathrm{Opt}_{\text{b-multi}}(\boldsymbol{\ell}, c, \boldsymbol{b})$.

For any given $\boldsymbol{b}'$, $c' \geq c_{min}(\boldsymbol{\ell}')$, we study a solution $\mathcal{S}'$ to $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}', c', \boldsymbol{b}')$. Then by definition there exists m, i', $\tilde{\mathcal{S}}$ a solution to $\textsc{Prob}_{\text{b-multi}}(\boldsymbol{\ell}_{[m\leftarrow \boldsymbol{\ell}'_m - i']}, c'-1, \boldsymbol{b}_{[m\leftarrow 1]})$ and $\mathcal{Z}$ a solution to $\textsc{Prob}(i'-1, c' - \sum_{j\neq m} \boldsymbol{b}_j)$ and:

$$\begin{aligned}
T(\mathcal{S}') &= j'u_f + T(\tilde{\mathcal{S}}) + T(\mathcal{Z}) \\
&\geq j'u_f + \mathrm{Opt}_{\text{b-multi}}\left(\boldsymbol{\ell}_{[m\leftarrow \boldsymbol{\ell}'_m - i']}, c'-1, \boldsymbol{b}_{[m\leftarrow 1]}\right) + \mathrm{Opt}_0\left(i'-1, c' - \sum_{j\neq m} \boldsymbol{b}_j\right) \\
&\geq \min_{\substack{1\leq m\leq k \\ 0<i'\leq \boldsymbol{\ell}_m}} \left[j'u_f + \mathrm{Opt}_{\text{b-multi}}\left(\boldsymbol{\ell}_{[m\leftarrow \boldsymbol{\ell}'_m - i']}, c'-1, \boldsymbol{b}_{[m\leftarrow 1]}\right) + \mathrm{Opt}_0\left(i'-1, c' - \sum_{j\neq m} \boldsymbol{b}_j\right)\right] \\
&= \mathrm{Opt}_{\text{b-multi}}(\boldsymbol{\ell}', c', \boldsymbol{b}')
\end{aligned}$$

Hence showing the result.

□

Remark. Note that for clarity we have used in this proof any vector $\boldsymbol{b}$ (which adds a factor of $O(2^k)$ to the complexity). To even reduce the complexity, we can permute in the dynamic program the branch that has been chosen so that $\boldsymbol{b}$ is always sorted. Hence we simply need to solve $O(k^2)$ different dynamic programs. The complexity would then be $O(ck^3(\max_i \boldsymbol{\ell}_i)^{k+1})$.

In addition, this proof also provides a way to compute the optimal solution based on the computation of its execution time.

6 Simulation Results

In this section, we depict the results of simulations on linearized versions of Cross-Modal embeddings (CM) and Siamese networks with triplet loss. Linearized 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 storage slots. We study the behavior of optimal solutions to the problem $\textsc{Prob}_{\text{multi}}(\boldsymbol{\ell}, c)$.

We propose three observations:

1. The trade-off Makespan vs Memory usage;

2. 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;

3. 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.

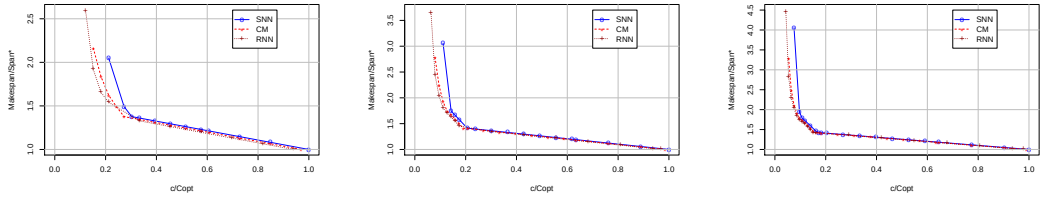
In order to compare the different types of networks in a normalized way, we consider several models with analogous sizes and computational costs. For a length L, we consider three graph structure:

- Cross-Modal (CM) embedding networks with two chains of sizes $(L, 5L)$. The motivation is those 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). Thus, as soon as the memory c is larger than $C_{CM}(L) = 6L + 2$, then the makespan is minimal and equal to $\text{Span}^*_{CM}(L) = 12L + 1$, which corresponds to the situation where all activations are stored during the forward phase of the training.

- 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. Analogously to the case of CMs, as soon as $c \geq C_{SNN}(L) = 6L + 3$, all forward activations can be stored and therefore the makespan is minimal and equal to $\text{Span}^*_{SNN}(L) = 12L + 1$.

- 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 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.



(a) $L = 5$ (b) $L = 10$ (c) $L = 15$

Figure 8: Makespan evolution with respect to different amount of total memory.

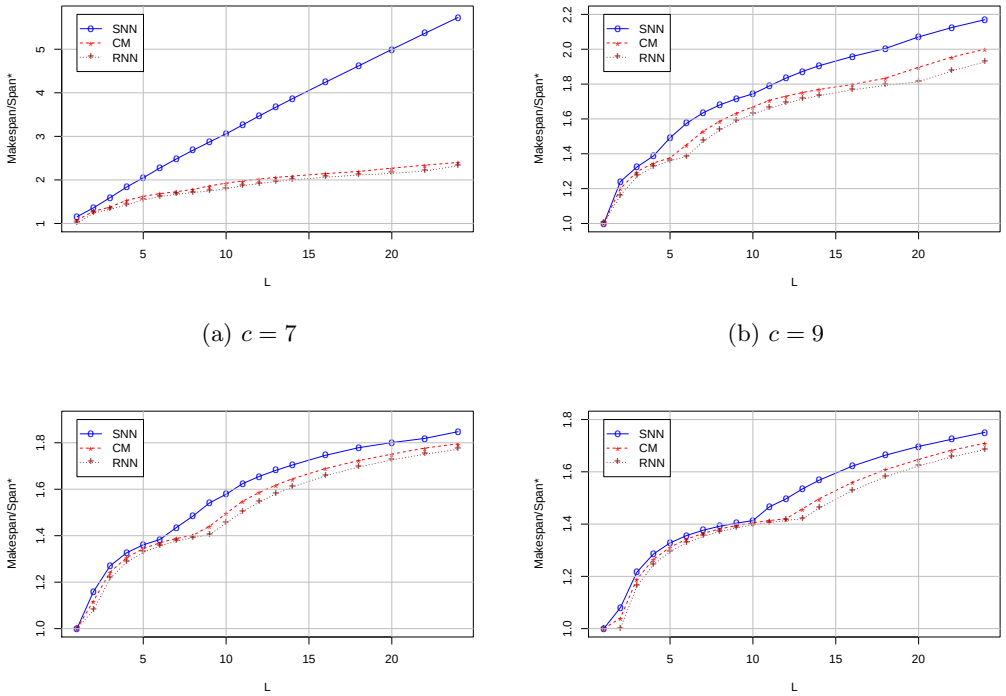
Trade-off Memory – Makespan The first question that we consider is, given a fixed number of forward operations (described by L for all scenarios), how does the makespan evolves as a function of the number of memory slots available.

We run our experiments for $L = 5, 10$ and 15. The plots depicting the evolution of makespan with the amount of memory are gathered in Figure 8. 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 $x \times C_{CM}$ memory slots, the makespan is $y \times \text{Span}^*(L)$.

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 Lemma 4. 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 $c/C_{\text{Opt}} = 0.5$. 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 9, we depict the dual situation, where the number of memory slots is fixed on each plot (either 7, 9, 11 or 13) and the ratio of the achieved makespan over Span^* 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. The exception is when the number of available checkpoints is exactly the minimum number of memory checkpoints ($c = 7$, 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 greater than $c_{\min}$. In addition, it is interesting to observe that for SNN and CNN the ratio follows a pattern similar to that of RNN: different threshold, and between those threshold a performance shaped as $\alpha - \beta/L$. Indeed, for the case of RNN those performance have been proven via a closed-form formula [25]. This can motivate the search for a similar closed-form formula for the problem of chains, and with this an algorithm whose complexity is polynomial in the number of branches. Finally, another observation is that the overall growth for a fixed



(a) $c = 7$ (b) $c = 9$

(c) $c = 11$ (d) $c = 13$

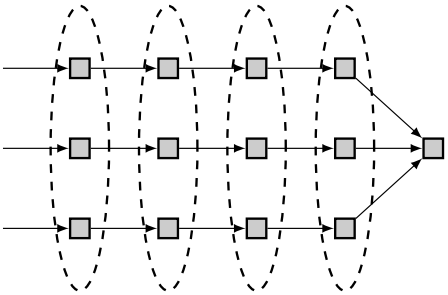
Figure 9: Makespan evolution with respect to different l for fixed memory size c.

number of checkpoints is quite slow even with few checkpoints, hence encouraging the use of these strategies. Indeed, we can observe that for a relatively small number of checkpoints such as $c = 11$, the ratio to the optimal makespan consistently remains below 2.

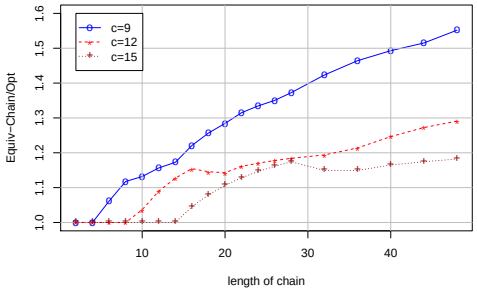
Comparison to linearization strategy While the previous evaluation have shown the gain one can expect by trading space for time, in this final evaluation, we aim at showing the importance of taking the structure of the graph into consideration. In particular we compare the strategy presented in this work versus the strategy that consists in (i) considering the join as a linear chain; (ii) using an existing strategy.

To consider the join as a linear chain, we group together operations that are at the same distance of the turn (see Figure 10). In this case, one can execute any algorithm for linear chains while considering that each forward step has cost ku_f, each backward step ku_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 [25]. 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 10 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 $= 2L + 1$).



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



(b) Performance of Equiv-Chain vs Optimal.

Figure 10: Graph transformation (left) and performance (right).

The results in this optimistic scenario show that Rev is surprisingly robust for small graphs but its performance steadily decreases when the ratio length/number of checkpoints increases. This shows the importance of taking into account the structure of the graph.

7 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 constraints 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 computing time, 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 homogeneous chains on which the literature on Automatic Differentiation has focused.

The goal of the present paper is precisely to extend the graph class by considering multiple parallel 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 and Siamese Neural Networks. In the case of multi-chain, we were able to build a dynamic program that allows us to compute the optimal strategy given a constraint on the size of the memory, i.e. a strategy that fulfills the memory constraint while minimizing the number of recalculations.

This work opens several natural perspectives. The first one obviously concerns the extension to other broader classes of graphs, or even to general DAGs, in order to take into account all possible types of Deep Neural Networks. After working on multiple chains, which have 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.

Acknowledgments

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

[1] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[3] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[4] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. 06 2017.

[5] Yang You, Zhao Zhang, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Imagenet training in 24 minutes. 09 2017.

[6] R. Hemenway. High bandwidth, low latency, burst-mode optical interconnect for high performance computing systems. In *Conference on Lasers and Electro-Optics, 2004. (CLEO).*, volume 1, pages 4 pp. vol.1–, May 2004.

[7] J. Liu, , W. Yu, J. Wu, D. Buntinas, , D. K. Panda, and P. Wyckoff. Microbenchmark performance comparison of high-speed cluster interconnects. *IEEE Micro*, 24(1):42–51, Jan 2004.

[8] Stefan Falkner, Aaron Klein, and Frank Hutter. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*, 2018.

[9] Olivier Bousquet, Sylvain Gelly, Karol Kurach, Marc Schoenauer, Michèle Sebag, Olivier Teytaud, and Damien Vincent. Toward optimal run racing: Application to deep learning calibration. *arXiv preprint arXiv:1706.03199*, 2017.

[10] Olivier Bousquet, Sylvain Gelly, Karol Kurach, Marc Schoenauer, Michèle Sebag, Olivier Teytaud, and Damien Vincent. Toward optimal run racing: Application to deep learning calibration. *arXiv preprint arXiv:1706.03199*, 2017.

[11] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.

[12] Thomas Paine, Hailin Jin, Jianchao Yang, Zhe Lin, and Thomas Huang. Gpu asynchronous stochastic gradient descent to speed up neural network training. *arXiv preprint arXiv:1312.6186*, 2013.

[13] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1509–1519. Curran Associates, Inc., 2017.

[14] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1709–1720. Curran Associates, Inc., 2017.

[15] 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.

[16] Navjot Kukreja, Alena Shilova, Olivier Beaumont, Jan Huckelheim, Nicola Ferrier, Paul Hovland, and Gerard Gorman. Training on the edge: The why and the how. In *1st Workshop on Parallel AI and Systems for the Edge, Rio de Janeiro, Brazil*, 2019.

[17] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 38(3):10–19, 2018.

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] Periodic checkpointing in pytorch, 2018. https://pytorch.org/docs/stable/checkpoint.html.

[30] A Adcroft, JM Campin, S Dutkiewicz, C Evangelinos, D Ferreira, G Forget, B Fox-Kemper, P Heimbach, C Hill, E Hill, et al. Mitgcm user manual, 2008.

[31] Phil Brubaker. *Engineering Design Optimization using Calculus Level Methods*. 2016.

[32] Navjot Kukreja, Jan Hückelheim, and Gerard J Gorman. Backpropagation for long sequences: beyond memory constraints with constant overheads. *arXiv preprint arXiv:1806.01117*, 2018.

[33] GC Pringle, DC Jones, S Goswami, SHK Narayanan, and D Goldberg. Providing the archer community with adjoint modelling tools for high-performance oceanographic and cryospheric computation. 2016.

[34] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman. Devito: an embedded domain-specific language for finite differences and geophysical exploration. *CoRR*, abs/1808.01995, Aug 2018.

[35] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and software*, 1(1):35–54, 1992.

[36] José Grimm, Loïc Pottier, and Nicole Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In Martin Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 95–106. SIAM, Philadelphia, PA, 1996.

[37] Philipp Stumm and Andrea Walther. Multistage approaches for optimal offline checkpointing. *SIAM Journal on Scientific Computing*, 31(3):1946–1967, 2009.

[38] Guillaume Aupy and Julien Herrmann. Periodicity in optimal hierarchical checkpointing schemes for adjoint computations. *Optimization Methods and Software*, 32(3):594–624, 2017.

[39] Michel Schanen, Oana Marin, Hong Zhang, and Mihai Anitescu. Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver nek5000. *Procedia Computer Science*, 80:1147–1158, 2016.

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

[41] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[43] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[44] 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.

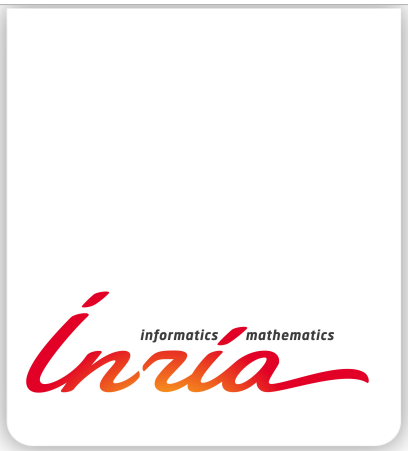
[45] M. Mueller, A. Arzt, S. Balke, M. Dorfer, and G. Widmer. Cross-modal music retrieval and applications: An overview of key methodologies. *IEEE Signal Processing Magazine*, 36(1):52–62, Jan 2019.

[46] 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.

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

[48] 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.

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



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399