Data frugality and computational efficiency in deep learning

Léon Zheng

UNIVERSITÉ DE LYON

ENS DE LYON

THÈSE
en vue de l'obtention du grade de Docteur, délivré par
l'ÉCOLE NORMALE SUPERIEURE DE LYON

École Doctorale N°512
École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 29 mai 2024, par :

Léon ZHENG

Frugalité en données et efficacité computationnelle dans l'apprentissage profond

Data frugality and computational efficiency in deep learning

Devant le jury composé de :

Julien MAIRAL, Directeur de recherche INRIA Grenoble	Rapporteur
François MALGOUYRES, PR Université de Toulouse 3 – Paul Sabatier	Rapporteur
Diane LARLUS, Personnalité scientifique NAVER LABS Europe	Examinatrice
Elisa RICCIETTI, Maître de conférences ENS de Lyon	Examinatrice
Pierre VANDERGHEYNST, Professeur EPFL STI IEL LTS2	Examinateur
Rémi GRIBONVAL, Directeur de recherche ENS de Lyon	Directeur de thèse
Patrick PÉREZ, Personnalité scientifique Kyutai	Co-directeur de thèse

Résumé

La performance des modèles d'apprentissage profond s'appuie aujourd'hui conjointement sur deux passages à l'échelle : une augmentation de la quantité des données d'entraînement d'une part, et une augmentation de la taille des modèles d'autre part. Ces deux passages à l'échelle sont en revanche coûteux. La constitution d'une grande base de données peut demander des efforts considérables en termes de prétraitement et d'annotations, tandis que le déploiement de modèles surparamétrés devient complexe lorsque les ressources computationnelles sont limitées. Ainsi, cette thèse s'intéresse aux enjeux de frugalité en données et d'efficacité en ressources de calcul dans l'apprentissage profond.

Premièrement, nous étudions l'apprentissage auto-supervisé, une approche prometteuse en vision par ordinateur qui ne nécessite pas d'annotations des données pour l'apprentissage de représentations. En particulier, nous proposons d'unifier plusieurs fonctions objectives auto-supervisées dans un cadre de noyaux invariants par rotation, ce qui ouvre des perspectives en termes de réduction de coût de calcul de ces fonctions objectives.

Deuxièmement, étant donné que l'opération prédominante des réseaux de neurones profonds est la multiplication matricielle, nous nous penchons sur la construction d'algorithmes rapides qui permettent d'effectuer la multiplication matrice-vecteur avec une complexité presque linéaire. Plus spécifiquement, nous étudions le problème de factorisation creuse de matrices sous la contrainte de parcimonie dite "butterfly", une structure commune à plusieurs transformées rapides comme la transformée de Fourier discrète. La thèse établit des nouvelles garanties théoriques sur l'algorithme de factorisation butterfly, notamment sur l'identifiabilité des facteurs creux et sur la quasi-optimalité de l'algorithme de factorisation. Nous nous penchons également sur le problème de factorisation butterfly à permutation inconnue des lignes et des colonnes, afin de rendre le modèle de factorisation plus flexible que celui où les supports des facteurs creux sont connus et fixés à l'avance. Enfin, nous explorons l'efficacité des implémentations GPU de la multiplication matricielle avec parcimonie butterfly, dans le but d'accélérer réellement des réseaux de neurones parcimonieux.

Abstract

The performance of deep learning models today relies jointly on two scaling considerations: an increase in the amount of training data, and an increase in the model size. However, both of these scaling strategies are costly. Building a large database can require significant efforts in terms of preprocessing and annotations, while deploying overparameterized models becomes complex when computational resources are limited. Thus, this thesis focuses on the challenges of data frugality and computational resource efficiency in deep learning.

First, we study self-supervised learning, a promising approach in computer vision that does not require data annotations for learning representations. In particular, we propose a unification of several self-supervised objective functions under a framework based on rotation-invariant kernels, which opens up prospects to reduce the computational cost of these objective functions.

Second, given that matrix multiplication is the predominant operation in deep neural networks, we focus on the construction of fast algorithms that enable matrix-vector multiplication with nearly linear complexity. More specifically, we examine the problem of sparse matrix factorization under the constraint of the so-called *butterfly* sparsity, a structure common to several fast transforms like the discrete Fourier transform. The thesis establishes new theoretical guarantees for butterfly factorization algorithms, particularly on the identifiability of the butterfly sparse matrix factorization and the quasi-optimality of the butterfly decomposition algorithm. We also address the problem of approximating a matrix via a butterfly factorization up to some unknown row and column permutations, in order to make the factorization model more flexible than the one where the sparsity patterns of the sparse factors are known and fixed in advance. Finally, we explore the efficiency of GPU implementations for butterfly sparse matrix multiplication, with the goal of truly accelerating sparse neural networks.

Remerciements

Au terme de ces années d'études supérieures, je souhaite remercier les personnes qui m'ont accompagné et soutenu tout au long de ce chemin.

Je remercie tout d'abord Rémi, Elisa, Patrick et Gilles pour leur encadrement tout au long de mon doctorat. Je pense que votre exemplarité dans le métier de la recherche et votre bienveillance m'ont permis de donner le meilleur de moi-même, et je suis content d'avoir pu accomplir mon travail grâce à votre soutien. Vous avez toujours su me conseiller et m'encourager pour progresser, et cela m'a apporté la confiance nécessaire pour mener à bien cette thèse. Vous étiez à l'écoute, et vos retours sur mes travaux m'ont toujours aidé à gagner en maturité. J'ai beaucoup appris à vos côtés durant ces années, que ce soit sur le plan scientifique, sur la démarche de la recherche, sur la communication des idées et des travaux, ou sur le plan humain. C'était un véritable plaisir de travailler avec vous, et j'en garderai de très bons souvenirs.

Je remercie ensuite l'ensemble des membres du jury de thèse d'avoir accepté d'évaluer mes travaux, et en particulier Julien Mairal et François Malgouyres pour leur relecture du manuscrit et leurs retours, en tant que rapporteurs.

Je remercie mes collaborateurs, Quoc-Tung Le, Antoine Gonon, Pascal Carrivain, avec qui j'ai co-écrit des articles qui ont servi à l'élaboration de cette thèse. C'était très agréable de travailler avec vous et j'ai beaucoup appris à vos côtés.

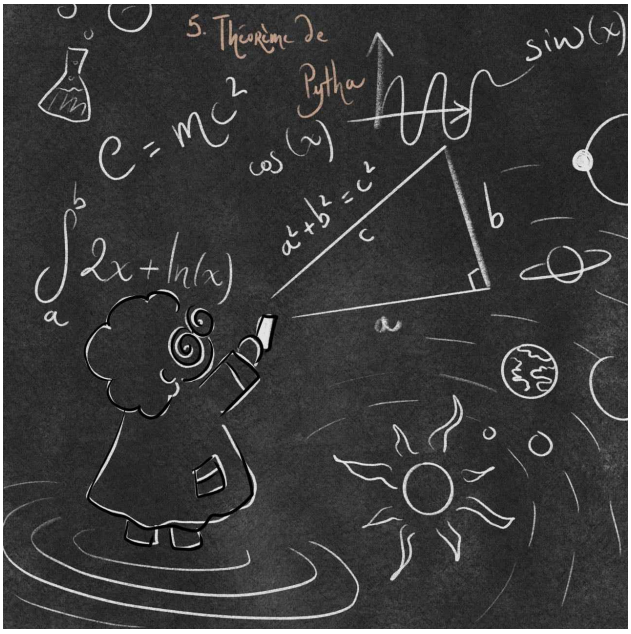
Je remercie l'ensemble de mes collègues de l'équipe OCKHAM, du LIP et de l'IXXI que j'ai côtoyés à Lyon, pour leur animation de la vie scientifique au sein du laboratoire. Merci à chacun lorsque vous étiez disponibles pour me donner des conseils, pour partager des idées et pour votre aide quand j'en avais besoin. Je garderai de bons souvenirs des moments partagés ensemble, que ce soit au bureau, à nos pauses déjeuners, au Ninkasi, à l'escalade, à Lion et Poisson, à l'opéra, au milieu des monts du Lyonnais, à Chambéry ou à tant d'autres moments inoubliables.

I would like to thank all my colleagues at valeo.ai for their support and their

valuable advices during my PhD. It was really nice to have been part of this international team where the working environment is very pleasant and where people are always willing to help each other. I was happy to learn about computer vision from you, and I enjoyed all the moments that we shared together during lunch breaks, workshops, restaurants, secret Santas, afterworks, and so on.

Je remercie tous les professeurs et enseignants que j'ai rencontrés au cours de mes études, qui ont toujours su éveiller ma curiosité d'apprendre davantage, et qui m'ont transmis le goût de l'effort et l'ambition de réussir.

Un grand merci à Stéphane et Yi pour leur gentillesse et leur accueil chaleureux à Lyon. J'ai partagé de très bons moments avec vous, entre nos repas, nos discussions, nos sorties, et j'en garde de merveilleux souvenirs. Je suis vraiment content d'avoir pu partager ces années de thèse avec vous.



Merci à Camille, Didier et Anselme.

Merci à toute ma famille et mes amis, et merci à Tonton David et Tata Léï de m'avoir accueilli quand je suis arrivé en France. Sachez que je me suis senti comme à la maison chez vous. Quel beau chemin nous avons tous parcouru ensemble !

Merci à Monsieur Antoine Zheng, de m'avoir donné des verres d'eau à Pékin. Bonne chance l'année prochaine c'est ton tour, j'assisterai à ton heure de gloire.

感恩父母的无限关怀。

本论文献给婆婆、爷爷、奶奶。

Contents

Notations

We list some notations commonly used throughout the thesis.

Notations for matrices, submatrices, entries. Matrices and vectors are in bold letters. $\mathbf{X}[i,:]$ and $\mathbf{X}[:,j]$ are the i-th row and the j-th column of $\mathbf{X}$, respectively. $\mathbf{X}[i,j]$ is the entry of $\mathbf{X}$ at the i-th row and j-th column. $\mathbf{X}[I,:]$ and $\mathbf{X}[:,J]$ are the submatrices of $\mathbf{X}$ restricted to a subset of row indices I and a subset of column indices J, respectively. $\mathbf{X}[I,J]$ is the submatrix of $\mathbf{X}$ restricted to both I and J. We extend these notations to multidimensional arrays (tensors).

Support of a matrix or a vector. The support of a matrix $\mathbf{M} \in \mathbb{C}^{m\times n}$ is the set $\mathrm{supp}(\mathbf{M}) := \{(i,j) \in [\![m]\!] \times [\![n]\!] \,|\, \mathbf{M}[i,j] \neq 0\}$. By abuse of notation, for any $\mathbf{B} \in \{0,1\}^{m\times n}$, we will sometimes write $\mathrm{supp}(\mathbf{M}) \subseteq \mathbf{S}$ instead of $\mathrm{supp}(\mathbf{M}) \subseteq \mathrm{supp}(\mathbf{S})$.

Concatenation of matrices. The column-wise concatenation of a family of matrices $\{\mathbf{M}_k\}_k$ having the same number of rows is denoted

$$\begin{pmatrix} \cdots & \mathbf{M}_k & \cdots \end{pmatrix}_k .$$

The row-wise concatenation of a family of matrices $\{\mathbf{M}_k\}_k$ having the same number of columns is denoted

$$\begin{pmatrix} \vdots \\ \mathbf{M}_k \\ \vdots \end{pmatrix}_k .$$

The block-diagonal matrix with diagonal blocks $\{\mathbf{M}_k\}_k$ is denoted

$$\begin{pmatrix} \ddots & & 0 \\ & \mathbf{M}_k & \\ 0 & & \ddots \end{pmatrix}_k .$$

For families of matrices $\{\mathbf{M}_k\}_k$ and $\{\mathbf{N}_k\}_k$ of adequate sizes, we have:

$$\begin{pmatrix} \cdots & \mathbf{M}_k\mathbf{N}_k & \cdots \end{pmatrix}_k = \begin{pmatrix} \cdots & \mathbf{M}_k & \cdots \end{pmatrix}_k \begin{pmatrix} \ddots & & 0 \\ & \mathbf{N}_k & \\ 0 & & \ddots \end{pmatrix}_k , \tag{1}$$

$$\text{or} \quad \begin{pmatrix} \vdots \\ \mathbf{M}_k\mathbf{N}_k \\ \vdots \end{pmatrix}_k = \begin{pmatrix} \ddots & & 0 \\ & \mathbf{M}_k & \\ 0 & & \ddots \end{pmatrix}_k \begin{pmatrix} \vdots \\ \mathbf{N}_k \\ \vdots \end{pmatrix}_k . \tag{2}$$

Notations / abbreviations	
$A \times B$	Cartesian product between set A and B
$B \backslash A$	Set difference of B and A
$A \cap B$	Intersection of sets
$A \cup B$	Union of sets
$\lvert A \rvert$	Cardinality of the set A
$\mathbb{R}$	Set of real numbers
$\mathbb{C}$	Set of complex numbers
$\mathbb{N}$	Set of integers
$\mathbb{N}^*$	Set of positive integers
$[\![L]\!]$	Integer set $\{1, \dots, L\}$
$[\![a, b]\!]$	Integer set $\{a, \dots, b\}$
$[a, b]$	Real interval with $a, b \in \mathbb{R}$
$\lvert x \rvert$	Absolute value of $x \in \mathbb{C}$
$\lVert \cdot \rVert_p$	ℓ_p-norm for a vector
$\lVert \cdot \rVert_F$	Frobenius norm
$\lceil \cdot \rceil$	Ceiling function
$\lfloor \cdot \rfloor$	Floor function
$\circ$	Function composition
$f \sim g$	Asymptotic equivalence between two functions
$(\mathbf{X}, \mathbf{Y}) \sim (\bar{\mathbf{X}}, \bar{\mathbf{Y}})$	Equivalent pairs of matrices, see Definition 6.3
$\odot$	Hadamard product
$\otimes$	Kronecker product
$\mathbf{X}^\top$	Matrix transpose
$\mathbf{X}^*$	Matrix conjugate transpose
$\mathbf{0}_{m \times n}$	$m \times n$ matrix full of zeros
$\mathbf{1}_{m \times n}$	$m \times n$ matrix full of ones
$\mathbf{I}_a$	Identity matrix of size a
j	Imaginary unit
$a \equiv b \mod c$	Integers a, b are congruent modulo c
$\binom{n}{k}$	Binomial coefficients
$\mathbf{K} * \mathbf{X}$	Convolution as in (4.5)
$\mathbb{E}$	Expectation of random variable
$x \sim \mathbb{P}$	Random variable with probability distribution $\mathbb{P}$
$\mathcal{N}(0, 1)$	Standard normal distribution
i.i.d.	Independent and identically distributed
resp.	Respectively

Chapter 1

Introduction

The *information revolution* that we have been witnessing over the last decades is likely one of the most significant reorganizations of our societies in history [144]. At the heart of this revolution are digital technologies, which have radically changed our ways of processing, storing, and communicating information. Today, massive amount of data are automatically processed using machine learning *algorithms*, with computers that are able now to perform billions of operations per second. These algorithms are based on the paradigm of *learning from data*, which bypasses the complexity of formally describing the knowledge of the world and the set of logical rules for inference, as proposed in the knowledge-based approach in the early days of artificial intelligence [279]. In machine learning algorithms, such a complexity is rather handled using a statistical approach, where a model acquires knowledge about the world by extracting important patterns from a so-called *training* dataset, in order to make predictions and decisions that generalize well on unseen data during *inference* [27, 160].

The performance of these algorithms then heavily depends on the *representations* of the data they are given [22]. Indeed, well-designed features allow the model to learn the underlying patterns in the data more effectively, leading to better predictions and decisions. In order to avoid the complexity of manually designing the right features for a specific learning task, deep learning algorithms choose to rely on representations that are themselves *learned from data*, by expressing them in terms of "other, simpler representations" [134]. Such a hierarchy of concepts is typically modeled by *deep neural networks*, which are mathematical models originally introduced in the framework of connectionism to study information processing in biological systems [263].

It turns out that that *increasing both the amount of high-quality data and the size of the model yields more effective representations learned by the deep neural network from the dataset*. Data quality refers to how well the data are annotated, curated, well-processed, so that the predictions of a model trained on these data are expected to be accurate. The size of a model can be roughly quantified by the number of its parameters, which is typically characteristic of the *expressiveness* of the model. Hence, with growing computation capabilities becoming available over the years,

the community has been witnessing the scaling up of the training dataset and the model capacity in deep learning algorithms, resulting in their increasing performance across various learning tasks. In other words, the recipe for success in deep learning has been: *training with more data, if possible of good quality, with larger models, and with more computation*. However, this logic pushed to the extreme comes with important challenges, concerning, on the one hand, the cost of collecting a massive amount of high-quality data relevant for the algorithm and the learning task, especially when representation learning requires *data labeling*, and, on the other hand, the computational cost for deploying large-scale neural networks, notably when computational resources are limited.

Overview of the thesis. This thesis focuses on the issue of reducing the dependence on data annotations for learning representations in *vision tasks*, and on the issue of reducing the computational cost of deep neural networks. The challenge is to achieve *data (annotation) frugality*, and *computational efficiency*, in the context of large-scale deep learning. For data frugality, we will study *self-supervised learning*, a promising approach for learning visual representations without any annotation. For computational efficiency, given that the predominant operation of deep neural networks is matrix multiplication, we will focus on the construction of fast algorithms that accelerate matrix-vector multiplication. In particular, we will study fast algorithms associated with some specific sparse matrix factorizations called *butterfly factorizations*.

Content of the chapter. Section 1.1 will start by further detailing how the increase of dataset and model size led to the increase of performance in deep learning models. We will then motivate the thesis by describing the costs of large scale deep learning, and some challenges that the community is facing with respect to self-supervised learning in vision (Section 1.2) and butterfly factorization for neural network compression (Section 1.3). Finally, Section 1.4 presents an overview of the research direction explored in this thesis to address these challenges.

1.1 More data, bigger models, more computation for better performance

Let us first come back on the main reasons that led to our study on data frugality and computational efficiency in deep learning.

The ImageNet database: a large collection of high-quality data. One of the key turning points in the development of deep learning algorithms is the creation of the ImageNet database [78]. This dataset is a vast and diverse collection of images from the Internet, carefully annotated with a hierarchy of concepts, whose quality has been controlled by human annotators through crowd-sourcing. The

scale of such a high-quality database had never been reached before its release. Such an effort in data collection has made the resurgence of deep learning algorithms possible in the field: these algorithms were explored in early pioneering works [219,314], but they were not truly successful at that time due to insufficient computational power to scale the dataset and model size effectively. In 2012, with the use of Graphical Processing Units (GPUs) for hardware efficiency, a deep convolutional neural network with 60 million parameters was able to outperform other approaches for ImageNet classification [203]. This moment marked an acceleration of the scaling of dataset and model size, in the quest for better performance in various learning tasks.

Neural scaling laws. Neural network models can scale in different manners [334, 376], for instance in depth [163, 323, 332] or in width[1] [173, 374], which results in an increasing number of parameters[2]. Mathematically, increasing depth or width enhances the model capacity, which is the ability of the neural network to well approximate functions from certain function classes [97, 303]. The combination of a bigger model with a larger training dataset yields better performance on the given learning task [329], as empirically quantified with *neural scaling laws* [168, 376]. These are studied across several orders of magnitude of dataset sizes, using large-scale databases like ImageNet [78] or JFT [329], which has nearly 3 billion annotated images, and across various orders of magnitude of the number of parameters, up to billions of parameters [376]. Typically, it is possible to use a power law to relate the performance of a neural network, measured in terms of its generalization error, to the dataset volume and the model size [168]. Importantly, the performance increase of deep neural networks cannot be obtained solely by scaling either the dataset volume or the model size: *it is necessary to scale them both simultaneously*. These scaling laws are empirically observed no matter which model architecture is used [21, 334, 336, 376], and they appear across various learning tasks, from language modeling [193], machine translation, speech recognition [168], multimodal image-text tasks [59], image generation, image modeling, video modeling, and mathematical problem solving [166].

The effectiveness of overparameterized neural networks. The term "overparameterization" generally refers to a model that has much more parameters than the number of training samples [6]. At this point, we want to point out that it is not trivial to explain why large-scale deep learning works well in practice. First,

[1] A deep neural network realizes a mathematical function that is essentially the composition of several affine transformations and non-linear activation functions. The depth refers to the number of layers in the network, i.e., the number of affine transformations involved in the composition. The width refers to the number of neurons or the number of channels at each hidden layer, which corresponds to the output dimension of the corresponding affine transformation.

[2] The number of parameters in a neural network usually corresponds to the number of edges in the graph describing the architecture of the neural network.

optimizing the parameters of a large neural network during training is generally regarded as a difficult problem, because it is a large-scale non-convex optimization problem [135]. But in practice, provided that the network is properly initialized, and that instability issues during optimization are properly handled[3], it is possible to reach low training error with stochastic gradient descent. Second, the empirical behavior of overparameterized neural networks challenges previous approaches for understanding generalization in machine learning. Traditionally, the classical bias-variance trade-off [160] suggests that overparameterization should lead to poor generalization, but this does not happen to deep neural networks as observed experimentally in neural scaling laws. This calls for new tools to grasp the generalization capabilities of these overparameterized neural networks [2, 19, 39, 69, 122]. In any case, experiments show that overparameterized deep neural networks are able to learn representations that are powerful enough to make accurate predictions on a considered task, provided that the training dataset is sufficiently large.

Transferring the representations learned at large scale. It happens that these learned representations trained at large scale can be meaningful enough to be *transferred* to other learning tasks that share some similarity to the original learning task, in the sense that it is somehow possible to exploit certain prior knowledge learned during the original task for a more efficient resolution of new tasks, using less computation and fewer data samples specific to these new tasks. For instance, the visual representations learned by *supervised* image classification on ImageNet have led to successful transfers to object detection and image segmentation [48, 127, 311]. In this paradigm of *transfer learning* [293, 378], the pipeline is composed of two steps: the *pretraining* where the deep neural network is optimized to solve a certain learning task, called the *upstream task* or the *pretext task*; and the *transfer* where the representations learned during the upstream task are leveraged in some other learning tasks of interest, called *downstream tasks*. Concretely, the model's parameters optimized during pretraining are directly reused for (hopefully) better initialization in the downstream task, compared to an initialization from scratch that ignores the pretraining. Importantly, even though the scaling laws for downstream and upstream performance may be different [1, 167, 338], it is observed, overall, that downstream performance benefits from the scaling up of the pretraining dataset size and the model capacity [200, 334, 376]. This further motivates the global trend of large-scale deep learning for better downstream performance, under the paradigm of so-called *foundation models* [31].

[3] Using for instance residual connections [163] to mitigate vanishing or exploding gradients [23].

1.2 Learning visual representations without labels

Pretraining on a *supervised* upstream task demonstrates strong results for visual transfer learning [200], but it requires a large amount of *high-quality labeled* images. Yet, collecting annotations becomes infeasible at large scale, because it is expensive in general. Labeling data requires human labor, and is typically performed via crowd-sourcing [201]. Some data present some form of ambiguities, which means that different persons can label the same data in different ways, resulting in noisy labeling. Consequently, ensuring the quality of the annotation might require several rounds of reviews and corrections. Some data require expert knowledge to annotate, as in medical images [396]. Annotation can be also tedious and time consuming when it is complex, for instance in datasets for object detection or image segmentation. As an example, the average time taken to label a single image of urban scene in the Cityscape dataset is 90 minutes [66]. Overall, the cost of labeling in large-scale settings calls for alternative approaches to representation learning that are less dependent on data annotation.

1.2.1 Motivations for self-supervised learning

In *self-supervised learning*, the goal is to design a certain pretext task that *does not require any form of annotation*, with the hope that the learned image representations are useful enough for several downstream tasks, both at image level like image classification, or at pixel level like image segmentation (see Chapter 2). This contrasts with other forms of pretraining where the pretext task relies on a certain degree of supervision, like supervised representation learning where all image are annotated with class labels[4], semi-supervised learning [292] where only a fraction of data is labeled, or weakly-supervised learning where, e.g., images are annotated with some text captions[5] [307]. Self-supervised learning has been successful in natural language processing (NLP). For instance, BERT [83] is trained to predict missing words in a sentence given its surrounding context, and this masked language modeling task has been able to harness a large corpus of non-annotated text to obtain powerful representations. This success in NLP has encouraged researchers in computer vision to adopt self-supervised learning strategies for learning visual representations.

Besides avoiding the cost of annotation or the cost of collecting aligned text-image datasets, self-supervised learning has the benefit of avoiding learning representations with text bias, because annotations can be noisy due to ambiguities that can happen during labeling. Moreover, introducing text supervision could limit "the information that can be retained about the image, since captions only approximate the rich information in images, and complex pixel-level informa-

[4] A class corresponds to a category, and in the supervised setting, we consider that there is a finite number of classes for a given learning task.

[5] Text captions correspond to a short description in natural language that summarizes the semantic content of the image, and are more flexible than class labels for the annotation.

tion may not surface with this supervision" [291]. Under this hypothesis, it is therefore preferable to only rely on visual data as in self-supervised learning, in order to learn stronger image representations that are more suited to visual downstream tasks, like image classification, object detection, object localization, semantic segmentation, depth estimation, video understanding, etc.

Overall, recent works on self-supervised learning in the past years have shown promising results, as their downstream performance can match or even surpass supervised representation learning. Yet, there are still many challenges to be addressed, and we present some of them related to this thesis.

1.2.2 Some challenges in self-supervised learning

When annotations are not available during pretraining, it is necessary to use some *prior knowledge about visual data* to design a pretext task. The idea is to guide the learning process by relying on some sort of regularity assumption about the structure of an image. Such a regularity means, for instance, that it is sometimes possible to predict a missing part of an image from the other parts of the image, or that the semantic of an image is invariant under some deformations of the image, like a horizontal flip of the image or a small change of colors.

Avoiding shortcuts during pretraining. The incorporation of such prior knowledge into the pretext task is then translated into an objective function that is minimized by the network during pretraining. However, the designed optimization problem should be sufficiently well-posed in order to avoid the model learning a so-called *shortcut solution*, which indeed minimizes the objective function, but yields ineffective representations for the resolution of downstream tasks, because too much information about the underlying patterns in the data is lost during pretraining. For instance, in the pretext task where representations are learned to be invariant to some small image transformations (cropping, blurring, color jittering, etc.) via the minimization of their distances in the latent space, we want to avoid the trivial solution where the network learns a constant representation that is independent of the input image. More generally, we want to avoid a *dimensional collapse* [177, 190] where the information encoded in the different components of the learned representations is redundant.

Lack of unification in self-supervised learning methods. As it will be described in Chapter 2, there are a lot of possible ways to incorporate different prior knowledge into different pretext tasks, which led to the design of many self-supervised learning methods in recent years. Yet, the field is currently lacking a unified theoretical view of self-supervised learning, which makes it difficult to explain why representations learned in a self-supervised manner generalize well on downstream tasks [13]. For instance, many works proposed different ways to avoid the collapse issue mentioned above by adding different forms of regularization

during pretraining in order to avoid ill-posedness, but we still lack a unified view of these different forms of regularization.

Computational costs. Being free of annotation does not prevent self-supervised learning methods from being computationally intensive, because they also rely on large-scale (unlabeled) datasets and large models to obtain better downstream performance [138, 291]. Some pretraining losses, such as contrastive loss [290], are based on pairwise comparison in a batch of image representations, which can be costly in terms of memory if the batch size is large. Similarly, some pretraining losses based on the empirical covariance matrix of a batch of representations [15] can be costly if the dimension of the representations is large. Overall, self-supervised learning methods are computationally heavy, and this calls for better efficiency of these methods.

1.3 Computational efficiency of deep neural networks

In general (not only in self-supervised learning), training and deploying neural networks at large scale is not feasible without an appropriate amount of computational resources.

1.3.1 The computational cost of large-scale deep learning

When processing a single data sample, computing the output values of the network, during the so-called *forward pass*, requires a number of operations that is proportional to the number of parameters in the model[6], and computing gradients by the backpropagation algorithm for optimizing the model's parameters during the so-called *backward pass* also requires the same amount of operations. Then, this amount of computation is multiplied by the total number of data samples being processed during training or inference. Given the increase of model size and dataset volume, the amount of computation required for training the largest deep learning models doubled every 3.4 months between 2012 and 2018 [9]. Handling this exponential growth requires large computing infrastructures like cloud computing, through massively parallelizing computation on dedicated hardware such as GPUs or Tensor Processing Units, whose energy consumption per unit can reach up to 700 W, for instance, for the peak power of one NVIDIA H100 GPU. Yet, with the current level of parallelization and hardware performance that we can achieve today, training deep learning algorithms still takes several weeks, up to several months, to achieve.

[6] More precisely, when the neural networks involve certain types of layers, the number of operations also scales with the dimension of the considered data. For instance, the number of operations increases with the image resolution in convolution layers, or with the number of tokens in the sequence in attention layers [345].

Energy footprint of deep learning. Overall, such an amount of computation leads to important energy consumption and carbon emission [300, 320, 328]. To give an order of magnitude, the 176 billion parameter language model BLOOM was trained on 1.6 terabytes of data during 118 days, using 1,082,990 GPU hours of computation performed on NVIDIA A100 GPUs, resulting in a total energy consumption of 433,196 kWh with an estimated carbon emission of 57 gCO2eq/kWh [255], which corresponds to a total carbon emission of 24.7 tCO2eq. The consumption is even larger when multiple experiments are required to achieve highly capable models. For instance, the total carbon emission can reach 284 tCO2eq for training a Transformer big model (213 million parameters) [345] with neural architecture search [328]. Yet, even more computation in deep learning algorithm is dedicated to the inference phase [81]: it is estimated that the total energy required by deep learning algorithms in the industry is split into 10% for training and 90% for inference [300]. Following this trend in energy consumption demand, it is estimated that, by 2027, servers dedicated to deep learning algorithms could consume levels of electricity similar to those of an entire country, such as Argentina, equivalent to around 0.5% of the world's electricity use [76]. Clearly, these trends question the sustainability of deploying large-scale deep learning.

The need for frugality in embedded systems. Reducing the computational footprint of deep learning algorithms is not only important on the server side, but it is crucial when it comes to deploy them on embedded systems for real-time applications, such as on smartphones, drones or vehicles. In some critical scenarios, relying on cloud computing for deep neural network inference on edge devices is not possible, due to issues on communication latency or user's data privacy. This calls for on-device computing, which forbids the deployment of cumbersome large-scale models due to limitations of memory storage, computational power and battery on embedded devices [56, 272]. In this context, it is important to design lightweight neural networks with good performance-efficiency trade-off that fit the constraints on computational resources, to allow real-time inference on the considered hardware.

1.3.2 Parameter redundancy in deep neural networks

It turns out that the parameters in trained overparameterized models present a certain form of *redundancy* that we can exploit to *compress* the model, in the sense that it is possible to reduce empirically the number of parameters in the trained model with minimal degradation of its performance. Parameters can be redundant when they possess a *sparse* structure, meaning that a non-negligeable fraction of its parameters can be *pruned*, i.e., set to zero, so that some operations can be skipped in theory during the forward pass of the network. For instance, it has been shown that 50% of the parameters in the OPT-175B large language model

can be pruned without increasing the perplexity[7] of the model [112]. Parameters can also be redundant in the sense that they admit a *low-rank* structure, which means that the complexity of the model can be reduced via low-rank decompositions of certain weight matrices or weight tensors in the network. For example, in some cases, it has been shown that 95% of a neural network can be predicted by the remaining parameters, in the sense of a certain low-rank decomposition [79].

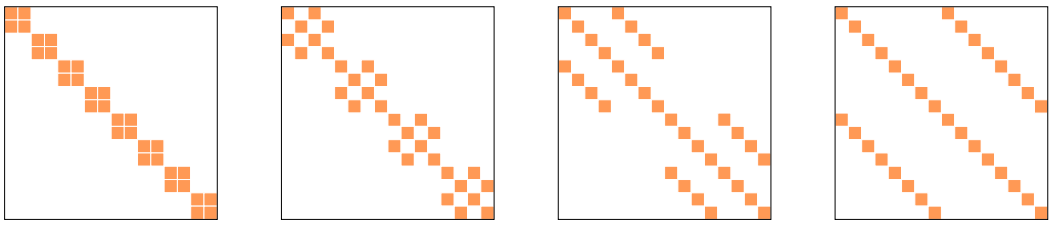
Benefits of reducing the number of parameters. In general, reducing the number of parameters leads to better computational efficiency, because the forward pass would typically require a smaller number of operations to compute. Hence, when the sparse operations are properly implemented, this can lead to time acceleration during training or inference, possibly with less memory footprint or energy consumption. Besides these benefits, it has also been hypothesized that reducing the number of parameters in a model can improve model generalization, out-of-distribution detection, data-efficiency, or robustness to adversarial and privacy attacks (see Chapter 4).

Unstructured *vs.* structured sparsity. Many pruning methods introduce sparsity in neural networks in an *unstructured* manner: the weight matrices at different layers of the neural network have a few nonzero entries, but their supports (i.e., the set of indices corresponding to nonzero entries) are not constrained to follow a specific *structure*. In some learning tasks, this can yield an impressive compression rate, where the number of parameters can be reduced by up to a factor of 10 or 100, without a significant loss of performance compared to the unpruned network [172]. But when a structure is lacking, implementing efficiently the corresponding sparse matrix multiplication can be challenging on some hardware, like on GPUs. This motivates a more structured form of sparsity for neural network compression and the development of efficient implementations that can leverage such a structure for faster sparse matrix multiplication.

Choosing the right model for decomposition. Compression by low-rank matrix decomposition can avoid the difficulty of implementing sparse matrix multiplication, since it results in smaller dense matrices that are more uniformly compatible with various hardware types. However, not all weight matrices in deep neural networks have a clear low-rank structure, such as in some weight matrices in the BERT model [83] where the singular values do not decay fast enough[8] [49], meaning that low-rank compression would not work in this case. In general, the choice of the right model for matrix decomposition is related to the nature of the redundancy in the considered matrix to compress. When low-rank decomposi-

[7] Perplexity measures how well a language model predicts text; lower values mean the model is more accurate.

[8] For instance, the sum of 50% of the singular values in a given weight matrix of the BERT model only accounts for 60% of the sum of all singular values [49].



(a) $\mathbf{S}_{\boldsymbol{\pi}_1}$ (b) $\mathbf{S}_{\boldsymbol{\pi}_2}$ (c) $\mathbf{S}_{\boldsymbol{\pi}_3}$ (d) $\mathbf{S}_{\boldsymbol{\pi}_4}$

Figure 1.1: Square dyadic butterfly supports as defined in (1.1) for the size $n = 16$. Zero entries (white) *vs.* entries that are allowed to be nonzero (colored).

tion is not suited, what are the possible alternative decomposition models to use for compressing weight matrices in overparameterized neural networks?

1.3.3 Butterfly factorization in neural network compression

Recent works in deep learning have been interested in a specific matrix decomposition called *butterfly factorization*. It corresponds to a multi-layer sparse matrix factorization inspired from the fast Fourier transform, which enables fast matrix-vector multiplication in quasi-linear complexity. To illustrate this, let us give one example of a butterfly factorization.

Definition. For a matrix $\mathbf{A}$ of size $n \times n$ where $n := 2^L$ with a certain integer L, the so-called *square dyadic butterfly factorization* corresponds to a multi-layer matrix factorization of the form

$$\mathbf{A} = \mathbf{X}_1 \mathbf{X}_2 \dots \mathbf{X}_L,$$

for some sparse factors $\mathbf{X}_1, \dots, \mathbf{X}_L$ of size $n \times n$ such that

$$\forall \ell \in [\![L]\!], \quad \operatorname{supp}(\mathbf{X}_\ell) \subseteq \operatorname{supp}(\mathbf{S}_{\boldsymbol{\pi}_\ell}), \quad \mathbf{S}_{\boldsymbol{\pi}_\ell} := \mathbf{I}_{2^{\ell-1}} \otimes \mathbf{1}_{2\times 2} \otimes \mathbf{I}_{2^{L-\ell}}, \tag{1.1}$$

Up to certain row or column permutations, this decomposition model can typically express matrices associated with many commonly used fast transforms, such as the discrete Fourier transform, the discrete cosine transform, the discrete sine transform and the Hadamard transform. In fact, the support constraints (1.1) on the factors $\mathbf{X}_1, \dots, \mathbf{X}_L$, illustrated in Figure 1.1, lead to a multi-layer matrix factorization that mimics the radix-2 fast Fourier transform algorithm (see Chapter 3). Consequently, such a butterfly factorization enables *fast* $\mathcal{O}(n \log n)$ matrix-vector multiplication by $\mathbf{A}$, because there are $L = \log_2 n$ sparse factors in the decomposition, where each of them has $2n$ nonzero entries. Importantly, the supports of the butterfly factors have a specific *structure* that can be interesting to enable efficient implementations for sparse matrix multiplication.

Applications to deep neural network compression. In practice, butterfly factorizations have been used in some deep learning applications. In line with recent works [72, 74, 75, 241], several variants of butterfly factorization are used to construct a generic representation for structured matrices that is not only expressive, but also differentiable and thus compatible with machine learning pipelines involving gradient-based optimization of parameters given training samples. For instance:

- The square dyadic butterfly factorization was used to replace hand-crafted structures in speech processing models or channel shuffling in certain convolutional neural networks, or to learn a latent permutation [75].
- The Monarch butterfly factorization [72] of some weight matrices in transformers for vision or language tasks can reduce their number of parameters with a certain drop of accuracy.
- Deformable butterfly factorizations in [241] of kernel weights in convolutional layers (see Section 4.5 for more details), for vision tasks, can reduce the number of the model's parameters, with a certain performance drop compared to the original convolutional neural network.
- Block butterfly factorizations [71] are used for parameter-efficient finetuning of pretrained large vision transformers, large language models, and text-to-image diffusion models [248].

1.3.4 Some challenges related to butterfly factorization

Even though these previous works show promising empirical results about neural network compression via butterfly factorization, this thesis proposes to take a step back by raising some fundamental questions regarding butterfly factorization and their usage for neural network compression.

Lack of baselines for assessing the performance of butterfly networks trained from scratch. One way to obtain a butterfly sparse neural network is to train it from scratch, by directly parameterizing certain weight matrices to admit a butterfly factorization $\mathbf{X}_1 \dots \mathbf{X}_L$, where each sparse factor $\mathbf{X}_\ell$ for $\ell \in [\![L]\!]$ is constrained to satisfy a prescribed support constraint, e.g., like the one given in (1.1). Generally, such a butterfly sparse neural network has fewer parameters, but it comes with a certain drop in performance compared to the dense neural network. Yet, besides not providing practical guidelines on how to select the layers for replacing a dense weight matrix with butterfly parameterization, previous works [72, 241] did not document how the obtained performance-efficiency tradeoff via butterfly factorization compares with other forms of parameter reductions, like pruning methods at initialization, or methods enforcing a low-rank parameterization on the weight matrices during training.

Considering permutation symmetries during decomposition. Another way to obtain a butterfly sparse neural network is to first train a dense neural network, and then decompose weight matrices at different layers of the trained network via butterfly factorization. This has the benefit of not training a butterfly sparse neural network from scratch. However, it is important to note that there exist permutation symmetries in the parameterization of neural networks, in the sense that neurons at the same layer can be shuffled without changing the function realized by the network. Therefore, in general, these permutation symmetries need to be taken into account when decomposing pretrained weight matrices via butterfly factorization. Yet, previous works [72, 241] ignored these symmetries, which can potentially lead to a large approximation error during the decomposition via butterfly factorization.

Lack of analytical description for butterfly factorization. In the context of neural network compression, all variations of butterfly factorization introduced in the previous works are of the form $\mathbf{X}_1 \dots \mathbf{X}_L$ for $L \geq 2$, where the factors satisfy some prescribed fixed-support constraints [71, 72, 74, 241]. However, these works do not characterize analytically the set of matrices admitting a sparse factorization with the considered prescribed sparsity supports. This prevents us from understanding whether or not a given weight matrix obtained after training satisfy the conditions to be well approximated by such a butterfly factorization.

Provable decomposition algorithms for butterfly factorization. The current literature is also lacking provable algorithms to approximate a given matrix by a product of butterfly factors satisfying the prescribed support constraints, like the one in (1.1). In general, deep matrix factorization with sparsity constraints is a difficult problem. Previous work proposed to use gradient-based methods or alternative least squares to perform decomposition via butterfly factorization with fixed-support constraints [74, 241], but these methods have no guarantees of success.

Implementing butterfly sparse matrix multiplication. Finally, previous methods show that they can reduce the number of parameters in neural networks via butterfly factorization, but there have been little numerical reports on the time efficiency of butterfly sparse neural networks, especially for GPU implementations. How can we efficiently implement butterfly sparse matrix multiplication in order to achieve time acceleration?

1.4 Contributions and outline of the thesis

The global objective of the thesis is to provide some answers to the questions raised by the challenges described in Sections 1.2.2 and 1.3.4, related to self-supervised learning for visual representations and butterfly factorization.

1.4.1 Overview of the literature review

The first part of the thesis (Chapters 2 to 4) is devoted to a literature review.

Chapter 2. In order to contextualize our contributions in self-supervised learning, we provide an overview of the different approaches to self-supervision. In particular, we will describe the different ways to introduce a form of regularization to avoid collapse when the pretext task is to learn representations that are invariant to some image transformations.

Chapter 3. To contextualize our work on butterfly factorization, we give an overview of fast algorithms for rapidly evaluating certain linear operators in scientific computing and numerical analysis. Indeed, butterfly factorization has been studied to enable fast matrix-vector multiplication in kernel matrices associated with special function transforms, or integral transforms such as the Fourier integral operator. It is therefore important to describe these previous works in order to better position our contributions.

Chapter 4. We provide an overview of the different methods for removing parameter redundancy in neural networks, such as methods related to pruning or low-rank decomposition. We will in particular detail the different usage of butterfly factorization for deep learning applications.

1.4.2 Overview of the contributions

In the second part, Chapter 5 presents our contribution in self-supervised learning, and Chapters 6 to 9 detail our contributions on butterfly factorization.

Self-supervised learning for image representations

Questions related to *self-supervised learning* are:

1. How can we unify different regularization methods to avoid learning a shortcut solution, when the pretext task is to learn representations to be invariant to some specific image transformations?

2. To what extent can we reduce the computational cost of self-supervised learning methods? In particular, is it possible to reduce the computational complexity of the self-supervised pretraining objective function?

We propose to address these questions using a framework based on *positive-definite kernels*.

Chapter 5. We introduce a regularization loss based on kernel mean embeddings with rotation-invariant kernels on the hypersphere, also known as dot-product kernels. Besides being competitive with the state of the art, the method reduces time and memory complexity for self-supervised training, making it implementable for very large embedding dimensions on existing devices and more easily adjustable than previous methods to settings with limited resources. The considered pretext task is to learn representations to be invariant to some predefined image transformations, while avoiding a degenerate solution by regularizing the embedding distribution. Our particular contribution is to propose a loss family promoting the embedding distribution to be close to the uniform distribution on the hypersphere, with respect to the maximum mean discrepancy pseudometric. We demonstrate that this family encompasses several regularizers of former methods, including uniformity-based and information-maximization methods, which are variants of our flexible regularization loss with different kernels. Beyond its practical consequences for state-of-the-art self-supervised learning with limited resources, the proposed generic regularization approach opens perspectives to leverage more widely the literature on kernel methods in order to improve self-supervised learning methods. This work is a collaboration with Gilles Puy, Elisa Riccietti, Patrick Pérez and Rémi Gribonval.

Butterfly factorization

We first propose a mathematical definition of butterfly factorization that unifies all the different variants introduced for deep learning applications [71,72,74,113, 241,342]. Generally, we say that a matrix admits a butterfly factorization if it can be written in the form $\mathbf{X}_1\mathbf{X}_2\dots\mathbf{X}_L$ for a certain number $L \geq 2$ of sparse factors $\mathbf{X}_\ell$ satisfying some fixed-support constraints:

$$\forall \ell \in [\![L]\!], \quad \operatorname{supp}(\mathbf{X}_\ell) \subseteq \operatorname{supp}(\mathbf{S}_{\boldsymbol{\pi}_\ell}), \quad \mathbf{S}_{\boldsymbol{\pi}_\ell} := \mathbf{I}_{a_\ell} \otimes \mathbf{1}_{b_\ell \times c_\ell} \otimes \mathbf{I}_{d_\ell}, \tag{1.2}$$

where $\boldsymbol{\pi}_\ell := (a_\ell, b_\ell, c_\ell, d_\ell)$ is a tuple of four integers for each $\ell \in [\![L]\!]$. For instance, this general definition includes the square dyadic butterfly factorization defined by the sparsity constraints (1.1). Based on this definition, we would like to provide some answers to the following questions:

1. Can we provide an analytical characterization of this definition of butterfly factorization? What conditions a matrix should satisfy in order to be well-approximated by such a butterfly factorization?

2. How can we approximate a matrix by a product of butterfly factors satisfying the prescribed support constraints (1.2)? Does the corresponding minimization problem always admit a minimizer? Can we design a tractable decomposition algorithm with guarantees on the approximation error?

3. How can we decompose a matrix via butterfly factorization up to some unknown row and column permutations?

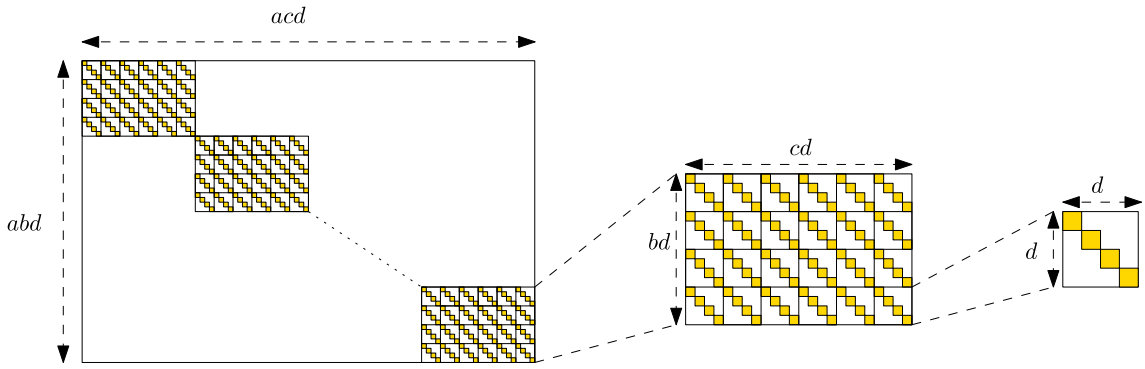


Figure 1.2: The general sparsity pattern of a butterfly factor, under the proposed framework that unifies all the different variants of butterfly factorization previously introduced for compressing deep neural networks.

4. How can we efficiently implement butterfly sparse matrix multiplication on GPUs? How does such an implementation specialized to the butterfly sparsity (1.2) compare to other generic implementations for dense or sparse matrix multiplication?

Chapter 6. In order to analyze the first two questions, we start by studying the specific case of the square dyadic butterfly factorization introduced above. Studying this specific case will illustrate the important ideas that will be reused to tackle the more general case. Our analysis is based on the investigation of the essential uniqueness of the square dyadic butterfly factorization, i.e., uniqueness up to unavoidable scaling ambiguities. We prove that any $n \times n$ matrix having the square dyadic butterfly structure admits an essentially unique factorization into L butterfly factors (where $n = 2^L$), and that the factors can be recovered by a hierarchical factorization method, which consists in recursively factorizing the considered matrix into two factors. This hierarchical identifiability property relies on a simple identifiability condition in the two-layer and fixed-support setting. This approach contrasts with existing ones that fit the product of butterfly factors to a given matrix via gradient descent. The proposed method can be applied in particular to retrieve the square dyadic butterfly factorization of the Hadamard or the discrete Fourier transform matrices of size $n = 2^L$. Computing such factorizations costs $\mathcal{O}(n^2)$ which is of the order of dense matrix-vector multiplication, while the obtained factorizations enable fast $\mathcal{O}(n \log n)$ matrix-vector multiplications. This work is a collaboration with Quoc-Tung Le, Elisa Riccietti and Rémi Gribonval.

Chapter 7. We then extend the results obtained for the square dyadic butterfly factorization to the general case. We investigate the problem of approximating a matrix by a product of sparse and structured factors following the fixed-support

constraints (1.2). Among these sequences of supports, we identify those that ensure the existence of an optimum in the corresponding factorization problem, thanks to a new property called *chainability*. For those supports, we propose an extension of the hierarchical algorithm introduced in the previous chapter. This new algorithm yields an approximate solution to the butterfly factorization problem supported by stronger theoretical guarantees than existing factorization methods. Specifically, we show that the ratio of the approximation error by the minimum value is bounded by a constant, independent of the target matrix. As a consequence of our analysis, we are able to give an analytical characterization of a butterfly factorization satisfying the fixed-support constraints (1.2). This characterization takes the form of a so-called *complementary low-rank property*, which was previously introduced for the decomposition of some kernel matrices in numerical analysis via butterfly factorization. This work is a collaboration with Quoc-Tung Le, Elisa Riccietti and Rémi Gribonval.

Chapter 8. The characterization by the complementary low-rank property will be particularly useful to tackle the third question. We address the more challenging problem of approximating a given matrix $\mathbf{A}$ by a product of butterfly factors $\mathbf{X}_1, \ldots, \mathbf{X}_L$, up to some unknown column and row permutations $\mathbf{P}, \mathbf{Q}$, i.e., we look for an approximation $\mathbf{A} \approx \mathbf{Q}^\top \mathbf{X}_1 \ldots \mathbf{X}_L \mathbf{P}$. We will see that, by the complementary low-rank characterization of the butterfly factorization, a matrix of the form $\hat{\mathbf{A}} := \mathbf{Q}^\top \mathbf{X}_1 \ldots \mathbf{X}_L \mathbf{P}$ admits several partitions of the matrix indices for which the submatrices of $\hat{\mathbf{A}}$ restricted to each element of these partitions are of low-rank. Therefore, our approach for butterfly factorization with unknown permutations is to find partitions of $\mathbf{A}$ into submatrices that can be well approximated by low-rank matrices, using a heuristic based on alternating subspace clustering. This work is a collaboration with Gilles Puy, Elisa Riccietti, Patrick Pérez and Rémi Gribonval.

Chapter 9. For the fourth question, we first assess the state of existing sparse matrix multiplication implementations on GPU for the butterfly sparse matrix multiplication. This is achieved through a comprehensive benchmark, which can be easily modified to add a new implementation. Its goal is to provide a simple tool for users to select the optimal implementation based on their settings. In this first assessment of existing algorithms, we observe that previous implementations perform some permutation operations that necessitate costly memory transfers across different levels of the GPU memory. Our contribution is to propose a novel memory transfer design, reducing the number of inter-level transfers by a factor of three. This is achieved with a new CUDA kernel, as it allows for managing GPU memory at different levels. The proposed implementation improves over previous ones in most of the tested cases in float-precision, with a speed-up factor up to $\times 1.2$. This work is a collaboration with Antoine Gonon, Pascal Carrivain and Quoc-Tung Le.

1.4.3 Publications and code diffusion

The material of this manuscript is based on the following published papers co-written during the thesis:

- Quoc-Tung Le*, Léon Zheng*, Elisa Riccietti, Rémi Gribonval. Fast learning of fast transforms, with guarantee. *Published at IEEE International Conference on Acoustics, Speech and Signal Processing, 2022* [213]. *Equal contribution. Source code has been diffused [385].

- Léon Zheng, Elisa Riccietti, Rémi Gribonval. Efficient identification of butterfly sparse matrix factorizations. *Published at SIAM Journal on Mathematics of Data Science, 2023* [392]. Source code has been diffused [391].

- Léon Zheng, Gilles Puy, Elisa Riccietti, Patrick Pérez, Rémi Gribonval. Self-supervised learning with rotation-invariant kernels. *Published at International Conference of Learning Representations, 2023* [388]. Source code has been diffused [387].

- Léon Zheng, Gilles Puy, Elisa Riccietti, Patrick Pérez, Rémi Gribonval. Butterfly factorization by algorithmic identification of rank-one blocks. *Published at Colloque Francophone de Traitement du Signal et des Images, 2023* [386]. Source code has been diffused [389].

It is also based on two papers under preparation:

- Antoine Gonon*, Léon Zheng*, Pascal Carrivain*, Quoc-Tung Le. Make Inference Faster: Efficient GPU Memory Management for Butterfly Sparse Matrix Multiplication. *Preprint, under review* [132]. *Equal contribution.

- Quoc-Tung Le*, Léon Zheng*, Elisa Riccietti, Rémi Gribonval. Butterfly factorization with error guarantees. *Paper under preparation for submission*. *Equal contribution.

The following paper co-written during the thesis will not be discussed in the manuscript:

- Antoine Gonon*, Léon Zheng*, Clément Lalanne, Quoc-Tung Le, Guillaume Lauga, Can Pouliquen. Sparsity in neural networks can improve their privacy. *Published at Colloque Francophone de Traitement du Signal et des Images, 2023* [133]. *Equal contribution.

Part I

Literature review

Chapter 2

Self-supervised learning for image representations

Scaling the size of the dataset during pretraining is a key ingredient for learning powerful representations that transfer well to various downstream tasks [200]. However, building up such a large dataset is costly when the pretraining method requires considerable effort in terms of annotation. This chapter gives an overview of self-supervised learning, a promising approach that does not require any image annotations for learning visual representations.

2.1 Introduction

In *self-supervised learning* methods, we design a certain learning task, called the pretext task, that does not require any data annotation, so that a neural network trained on this task learns to produce useful generic representations. The quality of these representations is then evaluated not by their performance on the pretext task, but by their performance on several other downstream tasks of interest. Typical visual downstream tasks are image classification, semantic segmentation, object detection, etc. In general, these downstream tasks are supervised, but only a small amount of labeled data is available for training. Hence, the learned representations during pretraining should be powerful enough in order to effectively address these downstream tasks, even when annotated training data are scarce, and they should also be generic enough so that they can be transferred in many different downstream tasks. For that, one way to design a pretext task is, for instance, to assume that some parts of the input data is not observable from the model, so that the model learns to predict these missing parts from the remaining observable parts of the data. In this sense, the data itself provides the supervision signal during the training of the neural network.

This chapter is partly inspired by Céline Hudelot's lecture at the 16th Summer School of Peyresq: https://gretsi.fr/peyresq22/.

Content of the chapter. We present three main families of self-supervised learning methods, namely, methods based on the prediction of image transformation (Section 2.2), invariance-based methods (Section 2.3) and generative methods (Section 2.4). We only focus on methods designed for image datasets: methods based on video datasets or multimodal methods are not covered by this chapter.

2.2 Methods based on the prediction of image transformations

Many methods [3, 85, 125, 275, 281, 282, 379] design a pretext task where the goal is to learn a model that takes a transformed image as input, and predicts which transformation was applied to the original image. The assumption of these methods is that predicting accurately the transformations requires to learn high-level semantic features. When the applied transformation belongs to a predefined parameterized set of transformations, the pretext task can be cast into a supervised learning problem, where transformed images are labeled by the parameter of the applied transformation. Let us give some examples of such pretext tasks, by considering some geometric or spatial transformations of the image.

Predicting rotations. In [125], the pretext task is to predict image rotation, from the observation of a rotated image. Concretely, for each unlabeled image $\mathbf{x}$ in the training set, the method generates four images obtained by the rotation of $\mathbf{x}$ with four angles $0°$, $90°$, $180°$ and $270°$. Each of these rotated images is then labeled by its rotation angle with respect to the original image $\mathbf{x}$. This defines a classification task on this augmented dataset with four classes, and a deep neural network can be trained to solve this task in a supervised manner.

Predicting relative positions. In [85, 281], the pretext task is to predict the relative positions between image patches of a given image. In [85], we construct pairs of image patches where the first one is sampled at a random position in a given image, and the second one is sampled randomly from one of the eight possible neighboring locations around the first patch. Then, the model takes such a pair of patches as input, and has to predict which of the eight spatial configurations corresponds to their relative position. This yields a classification task with eight classes that can be solved in a supervised manner.

An alternative pretext task is to solve a jigsaw puzzle [281]. Given an image partitioned into disjoint image patches, we shuffle the positions of the patches according to a certain permutation from a predefined finite set of permutations, and the model learns to predict from the shuffled patches which of these permutations was applied. This again yields a supervised learning task.

Figure 2.1: Different image transformations applied to the original image. From left to right: original image, color jitter (second and third image), grayscale, Gaussian blur, random cropping (sixth and seventh image). Images are from the documentation of `PyTorch`.

Learning equivariant representations. In order to solve the pretext task in the previous examples, the representation of a transformed image must capture some information about the transformation itself, in order to predict which transformation was applied to the original image. Therefore, the learned representation *varies* with the applied transformation, which contrasts with invariance-based methods presented below where learned representations are *invariant* to some transformations [267]. Variations of image representations with respect to image transformations is related to the mathematical notion of *equivariance* [223]. Formally, an encoder function $f : \mathcal{X} \mapsto \mathcal{Z}$ that maps an image to its representation is equivariant with respect to a group G if $f(T_g^{\mathcal{X}}(\mathbf{x})) = T_g^{\mathcal{Z}}(f(\mathbf{x}))$ for any $\mathbf{x} \in \mathcal{X}$ and $g \in G$, where $(g, \mathbf{x}) \mapsto T_g^{\mathcal{X}}(\mathbf{x})$ and $(g, \mathbf{z}) \mapsto T_g^{\mathcal{Z}}(\mathbf{z})$ are two (left) group actions of G on the image space $\mathcal{X}$ and the latent space $\mathcal{Z}$. However, solving the pretext tasks mentioned in the previous examples does not necessarily yield a truly equivariant encoder function in this mathematical sense. This is because solving the corresponding classification tasks does not guarantee the existence or not of a mapping between the representation of a transformed image and the one of the original image. In order to learn truly equivariant representations, it is necessary to consider an alternative pretext task where a predictor module has to learn such a mapping in the latent space during pretraining, as proposed recently in [82, 121, 296].

2.3 Invariance-based methods

In another main family of self-supervised learning methods, representations are learned to be *invariant* to some image transformations [267]. The underlying assumption of these methods is that high-level semantic features of an image should not depend on some specific image transformation. For instance, flipping horizontally an image containing a certain object should not change the class of this object, so the representation of an image should be invariant to image flipping. Other examples of image transformations used for invariance-based methods are illustrated in Figure 2.1.

The mathematical notion of invariance with respect to a group G is the specific case of equivariance where the encoder $f : \mathcal{X} \to \mathcal{Z}$ satisfies $f(T_g^{\mathcal{X}}(\mathbf{x})) = f(\mathbf{x})$ for

any $\mathbf{x} \in \mathcal{X}$ and $g \in G$. Therefore, the objective function in invariance-based methods includes an invariance criterion which takes low values for a given $g \in G$ if, and only if, $f(T_g^{\mathcal{X}}(\mathbf{x}))$ is close to $f(\mathbf{x})$. The minimization of this invariance criterion is then combined with a certain regularization technique that avoids learning a shortcut solution, where all the learned representations are constant and independent from the input image. Various invariance-based methods of the literature differ fundamentally in the way they enforce such a regularization to avoid collapse solutions. We now review three main strategies of regularization in invariance-based methods: contrastive methods, information-maximization methods and self-distillation methods.

2.3.1 Contrastive methods

In contrastive methods, collapse is avoided using a so-called *contrastive loss*. By construction, minimizing such a loss encourages the representations of two semantically similar images to be close together in the latent space, while pushing apart representations corresponding to dissimilar images. In the terminology of contrastive losses, pairs of images that are supposed to be similar are called *positive*, while pairs of dissimilar images are called *negative*. Therefore, to learn representations invariant to a predefined set of image transformations $\{\mathcal{T}_g^{\mathcal{X}}\}_{g \in G}$, a positive pair in the contrastive loss corresponds to a pair of transformed images obtained from two random transformations sampled from $\mathcal{T}$ of the same original image, while negative pairs are obtained by randomly sampling distinct images in the dataset.

Early contrastive losses. Contrastive losses were initially introduced for labeled images [38, 60, 152], where a positive pair corresponds to images from the same class, while a negative pair corresponds to images from different classes. Given a pair of images $(\mathbf{x}, \mathbf{x}')$ and an encoder function $f : \mathcal{X} \mapsto \mathcal{Z}$, the contrastive loss in [60] is defined as

$$\ell_{\text{cont}}(\mathbf{x}, \mathbf{x}') = \begin{cases} \|f(\mathbf{x}) - f(\mathbf{x}')\|_2 & \text{if } (\mathbf{x}, \mathbf{x}') \text{ is a positive pair} \\ \max(0, m - \|f(\mathbf{x}) - f(\mathbf{x}')\|_2) & \text{otherwise,} \end{cases}$$

where $m > 0$ is a certain margin parameter. Similarly, the triplet loss [47, 319, 355] is defined for three images $(\mathbf{x}, \mathbf{x}^+, \mathbf{x}^-)$ as:

$$\ell_{\text{triplet}}(\mathbf{x}, \mathbf{x}^+, \mathbf{x}^-) = \max\left(0, \|f(\mathbf{x}) - f(\mathbf{x}^+)\|_2 - \|f(\mathbf{x}) - f(\mathbf{x}^-)\|_2 + m\right),$$

where $(\mathbf{x}, \mathbf{x}^+)$ is a positive pair and $(\mathbf{x}, \mathbf{x}^-)$ is a negative pair. This triplet loss was then extended to a multiclass n-pair loss [326], to overcome slow convergence issues of the triplet loss, and its requirement for expensive data sampling methods to provide non-trivial triplet of images for faster training.

Noise contrastive estimation in instance discrimination. The multiclass n-pair loss [326] turns out to be equivalent to the contrastive loss derived from the noise contrastive estimation [148] in the *instance discrimination* approach for representation learning [88,358]. In this approach, the paradigm of learning representations invariant to image transformations is implemented by a classification problem at the *instance* level, where the number of classes k is equal to the size of the dataset. Each class contains only transformed images coming from the *same* original image in the dataset, and the model learns to predict the class of an augmented image, either using a parametric [88] or a non-parametric [358] classifier. A parametric classifier models the probability that an image $\mathbf{x}$ belongs to the class $i \in [\![k]\!]$ as:

$$P_{\text{param}}(i|\mathbf{x}) = \frac{\exp(f(\mathbf{x})^\top \mathbf{w}_i)}{\sum_{j=1}^{k} \exp(f(\mathbf{x})^\top \mathbf{w}_j)}, \tag{2.1}$$

where $\mathbf{w}_i$ is a learned weight vector. In a non-parametric classifier [358], the weight vector $\mathbf{w}_i$ is dropped and replaced simply by $f(\mathbf{x}_i)$, so that the probability becomes:

$$P_{\text{non-param}}(i|\mathbf{x}) = \frac{\exp\left(f(\mathbf{x})^\top f(\mathbf{x}_i)/\tau\right)}{\sum_{j=1}^{k} \exp\left(f(\mathbf{x})^\top f(\mathbf{x}_j)/\tau\right)}, \tag{2.2}$$

where the representations $f(\mathbf{x})$ are assumed to be ℓ_2-normalized for any $\mathbf{x}$, and $\tau > 0$ is a temperature parameter controlling the entropy of the distribution. However, instance-level classification becomes computationally heavy when the number of instances is very large, such as in the ImageNet-1k dataset [78] which contains 1.2 million images. To address this issue, the full softmax can be approximated using the noise contrastive estimator [148], as proposed in [358]. Considering a batch of ℓ_2-normalized representations $(\mathbf{z}, \mathbf{z}^+, \mathbf{z}_1^-, \dots, \mathbf{z}_{n-1}^-)$, where $(\mathbf{z}, \mathbf{z}^+)$ is a positive pair and $(\mathbf{z}, \mathbf{z}_i^-)$ is a negative pair for each $i \in [\![n-1]\!]$, the noise contrastive estimator loss, also called the InfoNCE loss [51, 162, 290, 358], is defined as:

$$\ell_{\text{NCE}}(\mathbf{z}, \mathbf{z}^+, \{\mathbf{z}_i^-\}_{i=1}^{n-1}) = -\log \frac{\exp\left(\mathbf{z}^\top \mathbf{z}^+/\tau\right)}{\exp\left(\mathbf{z}^\top \mathbf{z}^+/\tau\right) + \sum_{i=1}^{n-1} \exp\left(\mathbf{z}^\top \mathbf{z}^-/\tau\right)}.$$

Note that encoding images into ℓ_2-normalized representations helps to stabilize training [249, 298, 319].

Influence of negative pairs. It is overall observed that the downstream performance of contrastive methods increases with the number of negative samples used in the InfoNCE loss [50,162]. This justifies the use of large batch sizes in SimCLR [51], up to 8192, where the contrastive pairs are directly constructed from the transformed images in the current batch. To handle a larger number of negative samples, MoCo [162] proposes to use a dynamic dictionary with a queue that stores up to 65536 image representations computed in previous batches from a

moving-average encoder. Instead of sampling negative pairs of representations at random, contrastive learning methods can be further improved using hard negative mining [43, 63, 184, 192, 357], where the negative pairs are selected in such a way that their representations are close to each other in the latent space, in order to improve convergence during pretraining. Overall, contrastive methods can be computationally heavy due to their dependence on a large number of negative samples.

Asymptotic behavior with infinite negative samples. In the limit of infinite negative samples, it can be shown that the contrastive loss behaves asymptotically like a weighted sum between the invariance criterion and a certain *uniformity* term [354]. Indeed, assume that images are sampled from a data distribution $p_{\text{data}}(\cdot)$ over $\mathcal{X}$, and positive pairs are sampled from $p_{\text{pos}}(\cdot,\cdot)$ over $\mathcal{X}\times\mathcal{X}$, such that p_{pos} is symmetric[1] and the marginal distribution of $p_{\text{pos}}(\cdot,\cdot)$ matches[2] p_{data}. Denoting $\mathcal{S}^{q-1}$ the unit hypersphere in ℓ_2-norm of the Euclidean space of dimension q, the expectation of the InfoNCE loss with an encoder $f : \mathcal{X} \to \mathcal{S}^{q-1}$, n negative samples and a temperature parameter τ is:

$$\mathcal{L}_{\text{NCE}}(f,\tau,n) = \mathbb{E}_{\substack{(\mathbf{x},\mathbf{x}^+)\sim p_{\text{pos}} \\ \{\mathbf{x}_i^-\}_{i=1}^n \sim p_{\text{data}}}} \left[\ell_{\text{NCE}}\left(f(\mathbf{x}), f(\mathbf{x}^+), \{f(\mathbf{x}_i^-)\}_{i=1}^n\right)\right],$$

where $\{\mathbf{x}_i^-\}_{i=1}^n$ are i.i.d. samples from p_{data}. By [354, Theorem 1], as the number of negative samples n tends to infinity, this expectation converges to:

$$\lim_{n\to\infty}\left[\mathcal{L}_{\text{c}}(f,\tau,n) - \log n\right] = \underbrace{-\frac{1}{\tau}\mathbb{E}_{(\mathbf{x},\mathbf{x}^+)\sim p_{\text{pos}}}\left[f(\mathbf{x})^\top f(\mathbf{x}^+)\right]}_{\text{invariance criterion}} + \underbrace{\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\left[\log \mathbb{E}_{\mathbf{x}^-\sim p_{\text{data}}}\left[\exp\left(f(\mathbf{x}^-)^\top f(\mathbf{x})/\tau\right)\right]\right]}_{\text{uniformity term}}.$$

Minimizing the invariance criterion encourages the representations of the positive pairs to be close on the hypersphere, while minimizing the uniformity term encourages the distribution of $f(\mathbf{x})$ when $\mathbf{x} \sim p_{\text{data}}$ to be close to the uniform distribution on the hypersphere. This uniformity term can be interpreted as a regularization loss on the representation distribution that explicitly avoids collapse. In a similar spirit, Chapter 5 designs a generic regularization loss that encourages the representation distribution to be close to the uniform distribution on the hypersphere, in the sense of the maximum mean discrepancy associated with rotation-invariant kernels.

[1] $\forall \mathbf{x},\mathbf{x}^+ \in \mathcal{X}$, $p_{\text{pos}}(\mathbf{x},\mathbf{x}^+) = p_{\text{pos}}(\mathbf{x}^+,\mathbf{x})$
[2] $\forall \mathbf{x} \in \mathcal{X}$, $\int p_{\text{pos}}(\mathbf{x},\mathbf{x}^+)d\mathbf{x}^+ = p_{\text{data}}(\mathbf{x})$

2.3.2 Information-maximization methods

As an alternative to contrastive losses, information-maximization methods seek to prevent collapse without relying on negative samples, by decorrelating the components in the learned representation. This is achieved by encouraging the covariance matrix of $f(\mathbf{x})$ with $\mathbf{x} \sim p_{\text{data}}$, or the cross-correlation matrix of the pair $(f(\mathbf{x}), f(\mathbf{x}^+))$ with $(\mathbf{x}, \mathbf{x}^+) \sim p_{\text{pos}}$, to be close to a diagonal matrix [15, 101, 375].

Feature decorrelation via cross-correlation. For a given batch of representations $\{(\mathbf{z}_i, \mathbf{z}_i^+)\}_{i=1}^n$ encoded from positive pairs of images $(\mathbf{x}_i, \mathbf{x}_i^+) \sim p_{\text{pos}}$ with an encoder $f : \mathcal{X} \to \mathbb{R}^q$, the empirical cross-correlation matrix is defined as

$$\forall (k,l) \in [\![q]\!]^2, \quad \mathbf{C}[k,l] = \frac{\sum_{i=1}^n \mathbf{z}_i[k]\mathbf{z}_i^+[l]}{\sqrt{\sum_{i=1}^n (\mathbf{z}_i[k])^2}\sqrt{\sum_{i=1}^n (\mathbf{z}_i^+[l])^2}}, \tag{2.3}$$

where we assume that $\sum_{i=1}^n \mathbf{z}_i = \sum_{i=1}^n \mathbf{z}_i^+ = \mathbf{0}$ for simplicity. In Barlow twins [375], learning invariant representations while encouraging feature decorrelation can be achieved by minimizing the loss

$$\ell_{bt}(\{(\mathbf{z}_i, \mathbf{z}_i^+)\}_{i=1}^n) = \underbrace{\sum_{k=1}^q (1 - \mathbf{C}[k,k])^2}_{\text{invariance criterion}} + \lambda \underbrace{\sum_{k=1}^q \sum_{\substack{l=1 \\ l \neq k}}^q \mathbf{C}[k,l]^2}_{\text{redundancy reduction term}}, \tag{2.4}$$

where $\lambda > 0$ is a constant trading-off the importance of the two terms in the weighted sum.

Feature decorrelation via covariance. Instead of regularizing the cross-correlation matrix, W-MSE [101] performs feature decorrelation by applying a whitening transform to the current batch of learned representations, so that the whitened representations are centered and have an empirical covariance matrix equal to the identity matrix. Then, W-MSE minimizes the ℓ_2-distance between positive whitened representations. Alternatively, VICReg [15] (presented in more details in Chapter 5) does not perform explicitly a whitening operation, but minimizes a weighted sum between the invariance criterion and a covariance regularization term that encourages off-diagonal entries of the empirical covariance matrix of a batch of representations $\{\mathbf{z}_i\}_{i=1}^n$ to be close to zero.

Influence of large dimensions. In information-maximization methods, it is overall observed that the downstream performance increases with the dimension q of the representations [15, 375]. However, the complexity for computing the covariance or the cross-correlation matrix is quadratic with respect to q, so they become computationally heavy when q is large. We will show in Chapter 5 that this computational cost can be reduced using a generic kernel regularization loss.

Relation with contrastive methods. Under some mild assumption, information-maximization methods such as VICReg [15] and contrastive methods such as SimCLR [50] are shown to be equivalent [120], in the sense that they actually minimize a similar regularization loss during pretraining. Indeed, in the framework of [120], given a batch of n representations $\mathbf{Z} \in \mathbb{R}^{q \times n}$, a method is said to be sample-contrastive (such as SimCLR) if it is minimizes the contrastive criterion $L_c = \|\mathbf{Z}^\top \mathbf{Z} - \mathbf{Z}^\top \mathbf{Z} \odot \mathbf{I}_n\|_F^2$, where $\odot$ is the Hadamard product. It is said to be dimension-contrastive (such as VICReg) if it minimizes the non-contrastive criterion $L_{nc} = \|\mathbf{Z}\mathbf{Z}^\top - \mathbf{Z}\mathbf{Z}^\top \odot \mathbf{I}_q\|_F^2$. Based on the fact that $\|\mathbf{Z}^\top \mathbf{Z}\|_F^2 = \|\mathbf{Z}\mathbf{Z}^\top\|_F^2$, f [120, Theorem 3.3] claims that

$$L_{nc} + \sum_{k=1}^{q} \|\mathbf{Z}[k,:]\|_2^4 = L_c + \sum_{i=1}^{n} \|\mathbf{Z}[:,i]\|_2^4,$$

meaning that the two regularization losses are equivalent up to row and column normalization of the representation matrix $\mathbf{Z}$. In Chapter 5, in the continuity of uniformity-based methods such as [354], we rather propose a unification of contrastive and information-maximization methods under a generic kernel regularization loss.

2.3.3 Self-distillation methods

Both contrastive and information-maximization methods avoid collapse while learning invariant representations by minimizing a regularization loss during pretraining. In contrast, self-distillation methods avoid collapse without an explicit regularization loss, using a teacher-student architecture inspired from knowledge distillation [169]. In this architecture, a positive pair of images $(\mathbf{x}, \mathbf{x}^+)$ is encoded into two representations $(f_\xi(\mathbf{x}), f_\theta(\mathbf{x}^+))$ where $f_\xi : \mathcal{X} \to \mathcal{Z}$ and $f_\theta : \mathcal{X} \to \mathcal{Z}$ are called respectively the teacher and student encoders, parameterized by ξ and θ. Then, the model learns to predict the representation $f_\xi(\mathbf{x})$ from the observation of $f_\theta(\mathbf{x}^+)$. Various architectural tricks are then used to avoid collapse during self-distillation.

Architectural tricks. In BYOL [143], the teacher representation is predicted from the student representation using a predictor module $p_\gamma : \mathcal{Z} \to \mathcal{Z}$ parameterized by γ that is learned to minimize the distance $\|(p_\gamma \circ f_\theta)(\mathbf{x}^+) - f_\xi(\mathbf{x})\|_2$ during training. To avoid collapse, only the parameters γ and θ in the student branch are updated by gradient-descent, and the teacher parameter ξ is updated by an exponential moving average of the student parameter θ, following the update $\xi \leftarrow \tau\xi + (1 - \tau\theta)$ for a given fixed momentum parameter $0 < \tau < 1$. Some works also studied whether or not the batch normalization layers in the encoder plays a role in avoiding collapse during pretraining [313, 339].

In SimSiam [143], it is shown that the predictor module and the stop-gradient mechanism for the teacher encoder are sufficient to avoid collapse. The authors

show that the exponential moving average for the update of the teacher is not necessary, because it is sufficient to set the teacher's parameter ξ to be equal to the student's parameter θ during pretraining.

In DINO [143], instead of using a predictor module, the model learns to minimize the cross-entropy loss between the representations of the student and the teacher encoder, using a temperature softmax. To avoid collapse, a centering operation is applied for each batch of teacher representations so that its empirical mean is zero, and a stop-gradient operator is also used on the teacher encoder so that gradients only propagate through the student encoder. The teacher parameters are updated with an exponential moving average of the student parameters.

Regularization effects on the learned representations. In order to understand which architectural tricks are sufficient or necessary to avoid collapse, some theoretical works [154, 339] proposed to analyze the training dynamics of the learned representations, under some technical assumptions where the predictor module is assumed to be linear, and the network is assumed to be a deep linear network (i.e., a multilayer perceptron without bias and with identity activation function) with Gaussian i.i.d. inputs. They show that the predictor module and the stop-gradient mechanism play an important role in avoiding collapse, since they induce an implicit regularization on the variance of the learned representations. This draws connection with information-maximization methods such as VICReg [15] that minimizes a regularization loss based on feature decorrelation.

2.3.4 Role of image transformations

We end this section by discussing the role of image transformations in invariance-based methods. The success of these methods relies entirely on the good choice of data augmentations [50]. Indeed, a representation invariant to a specific image transformation can be more or less effective depending on the nature of the considered downstream task [32, 100]: some invariances can be beneficial because they help preserving the semantic information of the image, while others may be harmful because they remove necessary information for making good predictions in the downstream task. For instance, invariance to rotation may be beneficial for view-independent aerial image recognition [359], but invariance to color jittering can be harmful for flower classification where color information is important for accurate predictions [221]. In general, in order to learn more general purpose representations, it is important to determine whether the learned representation should be invariant or equivariant to a given image transformation. This has motivated several works [70, 82, 121, 359, 363] to explore the combination of equivariance-based (Section 2.2) and invariance-based methods.

2.4 Generative methods

Generative methods form an alternative family of self-supervised methods that rely less on image transformations. The pretext task is to mask a portion of the image, and to learn a model able to generate this missing part from the remaining observable part of the signal (possibly only in a latent space [291, 395], not necessarily at raw signal level [161, 299, 364]). The underlying assumption of such a pretext task is that the model needs to learn high-level semantic features in order to be able to reconstruct the missing part of the image. This paradigm is mainly inspired from representation learning via denoising auto-encoders [349], which are a class of autoencoders [170] that corrupt an input signal and learn to reconstruct the original uncorrupted signal.

Earlier attempts. The generative paradigm for self-supervision has been implemented in earlier methods [299, 380], where the model learns to recover the color of a grayscale image [380], or to inpaint a masked portion at the center of an image [299]. The task of recovering masked image patches is called *masked image modeling*, by analogy with the paradigm of *masked language modeling* in NLP [83]. However, the initial approach to masked image modeling [299] has not achieved a level of success comparable to that of methods in NLP [83].

Difference between vision and language. This difference of performance between vision [299] and language [83] may be explained by the following hypotheses [161]:

1. The model architecture is not the same. The vision model in [299] is a convolutional neural network, while the language model in [83] is a transformer.

2. The number of possible values for the output of the autoencoder's decoder is significantly larger in vision than in language. Decoders in vision reconstruct image patches that contain many pixels, whereas decoders in language reconstruct word tokens that can take only a small number of values predefined by a vocabulary.

3. Information density is different between language and vision. Images present heavy spatial redundancy, meaning that pixel-level information of neighboring patches may be sufficient to reconstruct a missing patch. Therefore, learning a vision model to reconstruct a small proportion of an image may not be sufficient to learn representations able to capture high-level semantics.

Modern methods for masked image modeling. Based on this analysis, recent methods for masked image modeling [14, 161, 364] proposed some important modifications to achieve competitive results:

1. They adopt the vision transformer architecture [87].
2. During decoding, raw pixel prediction can be replaced by the prediction of discrete visual tokens as in [14], which are extracted by the encoder of a discrete variational autoencoder [309].
3. A more aggressive masking strategy is proposed in [161,364], where a large proportion of patches, up to 75% in [161], is masked at random, so that the model is forced to capture high-level semantic features in order to reconstruct the masked patches.

2.5 Conclusion

In this chapter, we provided an overview of the recent advances in self-supervised learning methods, by distinguishing three families of methods: methods based on the prediction of image transformations, invariance-based methods, and generative methods. We notably describe the different ways to avoid collapse in invariance-based methods, either via a contrastive loss, the maximization of statistical information, or self-distillation. These are the main strategies to regularize the distribution of the learned representations. In Chapter 5, we will focus on invariance-based methods, and present a generic framework to unify certain existing regularization losses, based on a kernel framework using rotation-invariant kernels.

Chapter 3

Fast algorithms associated with butterfly factorization

Algorithms for the rapid evaluation of linear operators are important tools in many domains like scientific computing, signal processing, and machine learning. In such applications, where a very large number of parameters is involved, the *direct* computation of the matrix-vector multiplication hardly scales due to its quadratic complexity in the matrix size. This chapter reviews existing works that rely on some analytical or algebraic assumptions on the considered matrix in order to approximate its matrix-vector multiplication with a subquadratic complexity. We will focus more particularly on fast algorithms associated with the *butterfly factorization*.

3.1 Introduction

Matrix-vector multiplication is at the heart of a wide range of mathematical problems. Given a matrix $\mathbf{A} \in \mathbb{C}^{m \times n}$, it is the operation that maps a vector $\mathbf{x} \in \mathbb{C}^n$ to $\mathbf{y} := \mathbf{A}\mathbf{x} \in \mathbb{C}^m$, defined as:

$$\forall i \in [\![m]\!], \quad \mathbf{y}[i] := \sum_{j=1}^{n} \mathbf{A}[i,j]\mathbf{x}[j]. \tag{3.1}$$

The direct computation requires at least $\mathcal{O}(nm)$ operations, which corresponds to the number of parameters describing the matrix $\mathbf{A}$. This becomes prohibitive in large-scale settings, e.g., where the magnitude of m and n can be up of the order 10^6 or 10^7. Let us give some examples of such large-scale problems.

Example 3.1 (Linear transformation in signal processing)**.** *The matrix multiplication* $\mathbf{y} = \mathbf{A}\mathbf{x}$ *corresponds to a linear transformation of a discretized signal* $\mathbf{x}$ *that repre-*

A part of this chapter takes inspiration from an oral presentation given by Samual Potter at the 2023 SIAM Conference on Computational Science and Engineering.

sents the sampling of a given continuous signal. An example of such a linear operation is the discrete Fourier transform.

Example 3.2 (Discretization of partial differential equations)**.** *Under the variational formulation of partial differential equations, the finite element method uses a mesh to discretize the space domain so that solving numerically a given boundary value problem reduces to the resolution of a given linear system involving a stiffness matrix* $\mathbf{A} \in \mathbb{R}^{n\times n}$*. The obtained numerical solution approximates the real solution, with an error that approaches zero as* n *tends to* ∞*.*

Example 3.3 (n-body problem)**.** *Given a kernel operator* $(x, y) \mapsto k(x, y) \in \mathbb{R}$ *that quantifies the interaction between pairs of particles, evaluating all pairwise interactions in a system of* n *particles implies a multiplication by a matrix* $\mathbf{A} \in \mathbb{R}^{n\times n}$*, which is a discretization of this kernel operator.*

Example 3.4 (Integral transform)**.** *An integral transform* K *of a square integrable function* $f : [a, b] \to \mathbb{C}$*, associated with a kernel operator* $k : [a, b] \times [c, d] \to \mathbb{C}$*, is defined as the function* $(Kf)(y) = \int_a^b k(y, x)f(x)dx$*. This integral transform can be evaluated numerically using a quadrature rule, where* $(Kf)(y)$ *is approximated by* $\sum_{j=1}^n w_j k(y, x_j)f(x_j)$ *with well-chosen nodes* $x_1, \ldots, x_n$ *and weights* $w_1, \ldots, w_n$*. The evaluation of this approximation at several points* $y_1, \ldots, y_m$ *is therefore a matrix-vector multiplication with a kernel matrix* $\mathbf{A} \in \mathbb{C}^{m\times n}$*. An example of such an integral transform is the Fourier integral operator, which typically occurs in the resolution of wave equations like in inverse scattering problems.*

Example 3.5 (Special function transform)**.** *Given an orthogonal basis of functions* $\varphi_1, \varphi_2, \varphi_3, \ldots$ *that are square integrable on some interval* $[a, b]$*, the task of computing scalars* $\beta_1, \ldots, \beta_n$ *such that the truncation* $g : x \mapsto \sum_{j=1}^n \beta_j \varphi_j(x)$ *interpolates a given square integrable function* f *at some nodes* $x_1, \ldots, x_n \in [a, b]$ *can be formulated as a matrix-vector multiplication with a matrix of size* $n \times n$*, given by* $\beta_j = \sum_{i=1}^n w_i \varphi_j(x_i)f(x_i)$ *for* $j \in [\![n]\!]$ *and some scalars* $w_1, \ldots, w_n$*. Examples of such a transform associated with classical special functions include the Fourier-Bessel (Hankel) transforms, orthogonal polynomial transforms, etc.*

In the quest to reduce the complexity of matrix multiplication, when the considered matrix satisfies some specific algebraic or analytical assumptions typically encountered in the previous examples, it is possible to design *fast algorithms* for applying the corresponding linear operator in *quasi-linear* or *linear* complexity. This chapter will review two types of fast algorithms:

- **Fast algorithms associated with structured matrices**: these algorithms allows an *exact* rapid evaluation of the considered matrix, by leveraging some algebraic features of its entries. One example is the Cooley-Tuckey algorithm [65] for the discrete Fourier transform (DFT).
- **Analysis-based**[1] **fast algorithms**: these algorithms do not evaluate exactly the matrix-vector multiplication, but provide an approximate evaluation up

[1]The terminology "analysis-based" is borrowed from [288].

to some approximation error. The considered matrix is usually the discretisation of a certain kernel operator, which has a certain analytical assumption that takes the form of a certain low-rank property for the corresponding kernel matrix. One example of analysis-based fast transforms is the fast multipole method [140].

Butterfly factorization. Among the different assumptions enabling an approximate rapid evaluation of matrix-vector multiplication, previous work has identified the class of *butterfly* matrices [40, 266, 288] that typically satisfy a certain low-rank property, named the *complementary low-rank* property [234]. Under this assumption, the target matrix $\mathbf{A}$, of size $n \times n$, possesses specific submatrices that are numerically low-rank. Consequently, $\mathbf{A}$ can be compressed through successive hierarchical low-rank approximations of these submatrices, in the sense that it can be approximated by a sparse factorization:

$$\hat{\mathbf{A}} = \mathbf{X}_1 \dots \mathbf{X}_L,$$

with $L = \mathcal{O}(\log n)$ factors. Each factor $\mathbf{X}_\ell$ has at most $\mathcal{O}(n)$ nonzero entries for each $\ell \in [\![L]\!]$. This sparse factorization, called in general *butterfly factorization*, would then yield a fast algorithm for the approximate evaluation of the matrix-vector multiplication by $\mathbf{A}$, in $\mathcal{O}(n \log n)$ complexity. Algorithms for the construction of such sparse factors are called *butterfly algorithms* [265, 266, 288, 289] in the literature, and have been applied to many operators appearing in scientific computing problems, like kernel matrices associated with special function transforms (e.g., Fourier-Bessel transform, orthogonal polynomial transforms or spherical harmonic transforms in Example 3.5) [288, 289, 341, 369], time-harmonic scattering operators in electromagnetics [146, 251, 265, 266], or Fourier integral operators [40, 77, 235] (Example 3.4).

Contents of the chapter. The goal of this chapter is to explain the nature of butterfly algorithms in order to better position our contributions in Chapters 6 to 8. As we will see, in contrast to fast algorithms associated with structured matrices [294] that are presented in Section 3.2, butterfly algorithms are *analysis-based* fast algorithms, as is the case with the fast multipole method [140], presented in Section 3.3 and revisited with the framework of $\mathcal{H}$-matrices and $\mathcal{H}^2$-matrices [149–151]. In Section 3.4, we will formally introduce the low-rank property leveraged by butterfly algorithms to enable rapid matrix-vector multiplication, and we will detail some examples of kernel operators that satisfy such a property. Section 3.5 describes how the butterfly algorithm constructs a butterfly factorization that approximates a matrix satisfying this low-rank property. In particular, we will see that the fast algorithm associated with butterfly factorization mimics the fast Fourier transform. The detailed description of the classical butterfly algorithm will allow a precise comparison with the variants proposed in this thesis. Section 3.6 concludes with some important variations of the butterfly algorithms.

3.2 Fast algorithms for structured matrices: the example of the FFT

Probably one of the most famous fast algorithms is the *fast Fourier transform* (FFT), which allows us to compute the discrete Fourier transform in *quasi-linear complexity*. As opposed to analysis-based fast algorithms, the FFT is an *exact* rapid evaluation of the corresponding linear operator. For that, it typically relies on some specific *algebraic features* of the DFT matrix, as we detail with the Cooley-Tukey FFT in the next section. In particular, this leads to an *exact* factorization of the DFT matrix of size $n \times n$ into $\mathcal{O}(\log n)$ sparse factors with $\mathcal{O}(n)$ nonzero entries.

3.2.1 The Cooley-Tukey fast Fourier transform

The goal of the Cooley-Tukey [65] is to compute the DFT defined as follows.

Definition 3.1 (Discrete Fourier transform)**.** *The DFT of a given sequence* $(x_0, \dots, x_{n-1})$ *is defined as the sequence* $(X_0, \dots, X_{n-1}) := \mathbf{DFT}_n(x_0, \dots, x_{n-1})$ *given by*

$$\forall k \in [\![0, n-1]\!], \quad X_k := \sum_{i=0}^{n-1} x_i \omega_n^{ki},$$

where $\omega_n := e^{-\mathrm{j}\frac{2\pi}{n}}$ *denotes the primary n-th complex root of unity for a given integer* n*.*

The main idea is to use a divide-and-conquer strategy that recursively breaks the computation of the DFT of size $n \times n$ into several computations of the DFT at smaller length, assuming that n is composite. This is possible by leveraging some *symmetry and periodicity* structure of the kernel $(k, i) \mapsto \omega_n^{(k-1)(i-1)}$ for $i, k \in \mathbb{N}$.

The presentation of this paragraph is based on [89]. Let n_1, n_2 be two integers such that $n = n_1 n_2$. The Cooley-Tukey FFT [65] considers a partition of $[\![0, n-1]\!]$ into n_1 subsets of n_2 integers that are equally spaced by a distance n_1:

$$\forall i_1 \in [\![0, n_1 - 1]\!], \quad U_{i_1} := \{i_1 + i_2 n_1\}_{i_2 \in [\![0, n_2 - 1]\!]},$$

so that the sum in (3.1) can be written as:

$$\begin{aligned} \forall k \in [\![0, n-1]\!], \quad X_k &= \sum_{i_1=0}^{n_1-1} \sum_{i_2=0}^{n_2-1} x_{i_1+i_2 n_1} \omega_n^{k(i_1+i_2 n_1)} \\ &= \sum_{i_1=0}^{n_1-1} \omega_n^{k i_1} \sum_{i_2=0}^{n_2-1} x_{i_1+i_2 n_1} \omega_n^{k i_2 n_1} \\ &= \sum_{i_1=0}^{n_1-1} \omega_n^{k i_1} \underbrace{\sum_{i_2=0}^{n_2-1} x_{i_1+i_2 n_1} \omega_{n_2}^{k i_2}}_{:= Y_{i_1,k}}, \end{aligned} \tag{3.2}$$

where the last equality comes from the fact that

$$\forall i \in \mathbb{N}, \quad \omega_n^{in_1} = e^{\mathrm{j}\frac{2\pi}{n}n_1 i} = e^{\mathrm{j}\frac{2\pi}{n_2}i} = \omega_{n_2}^{i}. \tag{3.3}$$

This means that $\{Y_{i_1,k}\}_{i_1,k}$ in (3.2) correspond to the outputs of n_1 DFTs of length n_2: indeed, on the one hand, since ω_{n_2} is an n_2-th root of unity, we have

$$\forall k \in [\![n-1]\!], \quad Y_{i_1,k} = Y_{i_1,k_2} \quad \text{where} \quad k_2 \equiv k \mod n_2, \tag{3.4}$$

and on the other hand, we remark by Definition 3.1 that

$$\forall i_1 \in [\![0, n_1 - 1]\!], \quad (Y_{i_1,k_2})_{k_2=0}^{n_2-1} = \mathbf{DFT}_{n_2}(x_{i_1}, x_{i_1+n_1}, \ldots, x_{i_1+(n_2-1)n_1}). \tag{3.5}$$

Define

$$\tilde{Y}_{i_1,k_2} := \omega^{k_2 i_1} Y_{i_1,k_2}, \quad \forall (i_1, k_2) \in [\![0, n_1 - 1]\!] \times [\![0, n_2 - 1]\!]. \tag{3.6}$$

Writing the Euclidean division of $k \in [\![n-1]\!]$ as $k = k_1 n_2 + k_2$, the expression (3.2) becomes

$$\begin{aligned} \forall (k_1, k_2) \in [\![0, n_1 - 1]\!] \times [\![0, n_2 - 1]\!], \quad X_{k_1 n_2 + k_2} &= \sum_{i_1=0}^{n_1-1} \omega_n^{(k_1 n_2 + k_2) i_1} Y_{i_1, k_1 n_2 + k_2} \\ &= \sum_{i_1=0}^{n_1-1} \omega_n^{k_1 n_2 i_1} \underbrace{\omega_n^{k_2 i_1} Y_{i_1,k_2}}_{=\tilde{Y}_{i_1,k_2}} \\ &= \sum_{i_1=0}^{n_1-1} \omega_{n_1}^{k_1 i_1} \tilde{Y}_{i_1,k_2}, \end{aligned} \tag{3.7}$$

where, again, the last equality comes from the fact that

$$\omega_n^{k_1 n_2 i_1} = e^{-\mathrm{j}\frac{2\pi}{n}n_2 i_1} = e^{-\mathrm{j}\frac{2\pi}{n_1}i_1} = \omega_{n_1}^{k_1 i_1}. \tag{3.8}$$

In other words, the expression (3.7) corresponds to n_2 DFTs of length n_1, because by Definition 3.1:

$$\forall k_2 \in [\![n_2 - 1]\!], \quad (X_{k_1 n_2 + k_2})_{k_1=0}^{n_1-1} = \mathbf{DFT}_{n_1}(\tilde{Y}_{0,k_2}, \tilde{Y}_{1,k_2}, \ldots, \tilde{Y}_{n_1-1,k_2}). \tag{3.9}$$

Summary. The divide-and-conquer strategy in the Cooley-Tukey FFT of length $n = n_1 n_2$ is composed of three steps:

1. compute n_1 DFTs of length n_2 as in (3.5);
2. perform $n_1 n_2 = n$ multiplications as in (3.6);
3. compute n_2 DFTs of length n_1 as in (3.9).

Therefore, in general, the total number of multiplications in the three steps is $n_1 n_2^2 + n + n_2 n_1^2 = n(n_2 + 1 + n_1)$, which is smaller than the quadratic complexity n^2 if $n_1, n_2 > 2$. If n_1 or n_2 are also composite, then this step can be repeated recursively to further reduce the complexity, as in the radix-2 FFT algorithm presented below.

Remark 3.1. *Note that there exist other FFT algorithms when n is prime [306].*

3.2.2 Radix-2 FFT algorithm

Assume that n is a power of two. Then, applying recursively the Cooley-Tukey FFT by choosing $n_1 = 2$ and $n_2 = n/2$ yields the so-called *radix-2 algorithm*. In this case, the expression (3.7) becomes:

$$\begin{aligned} \forall k \in [\![0, n/2-1]\!], \quad X_k &= Y_{0,k} + \omega_n^k Y_{1,k}, \\ X_{\frac{n}{2}+k} &= Y_{0,k} - \omega_n^k Y_{1,k}, \end{aligned} \tag{3.10}$$

where

$$\begin{aligned} (Y_{0,k})_{k=0}^{n/2-1} &:= \mathbf{DFT}_{n/2}(x_0, x_2, \dots, x_{n-2}), \\ (Y_{1,k})_{k=0}^{n/2-1} &:= \mathbf{DFT}_{n/2}(x_1, x_3, \dots, x_{n-1}). \end{aligned}$$

Let us rewrite (3.10) in a matrix format.

Definition 3.2 (DFT matrix)**.** *We denote the DFT matrix of size $n \times n$ as*

$$\mathbf{F}_n := \left(\omega_n^{(k-1)(i-1)}\right)_{(k,i)\in[\![n]\!]^2} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \dots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix}.$$

For any even integer n, denote $\mathbf{P}_n$ the permutation matrix of size n that sorts the even then the odd indices, in the convention that the first index of a sequence is zero, i.e., it permutes $(x_0, x_1, \dots, x_{n-2}, x_{n-1})$ to $(x_0, x_2, ..., x_{n-2}, x_1, x_3, \dots, x_{n-1})$. Then, by (3.10):

$$\forall \mathbf{x} \in \mathbb{C}^n, \quad \mathbf{F}_n \mathbf{x} = \underbrace{\begin{pmatrix} \mathbf{I}_{n/2} & \mathbf{W}_{n/2} \\ \mathbf{I}_{n/2} & -\mathbf{W}_{n/2} \end{pmatrix}}_{:=\mathbf{B}_n} \begin{pmatrix} \mathbf{F}_{n/2} & \mathbf{0}_{n/2} \\ \mathbf{0}_{n/2} & \mathbf{F}_{n/2} \end{pmatrix} \mathbf{P}_n \mathbf{x}, \tag{3.11}$$

where $\mathbf{W}_{n/2}$ is the diagonal matrix with diagonal entries $(\omega_n^k)_{k=0}^{n/2-1}$. Unrolling recursively this factorization yields:

$$\begin{aligned} \mathbf{F}_n &= \mathbf{B}_n \begin{pmatrix} \mathbf{F}_{n/2} & \mathbf{0} \\ \mathbf{0} & \mathbf{F}_{n/2} \end{pmatrix} \mathbf{P}_n \\ &= \mathbf{B}_n \begin{pmatrix} \mathbf{B}_{n/2} & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_{n/2} \end{pmatrix} \begin{pmatrix} \mathbf{F}_{n/4} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{F}_{n/4} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{F}_{n/4} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{F}_{n/4} \end{pmatrix} \begin{pmatrix} \mathbf{P}_{n/2} & \mathbf{0} \\ \mathbf{0} & \mathbf{P}_{n/2} \end{pmatrix} \mathbf{P}_n \\ &= \dots \\ &= (\mathbf{I}_1 \otimes \mathbf{B}_n)(\mathbf{I}_2 \otimes \mathbf{B}_{n/2})(\mathbf{I}_4 \otimes \mathbf{B}_{n/4}) \dots (\mathbf{I}_{n/2} \otimes \mathbf{B}_2)\mathbf{Q}_n \end{aligned} \tag{3.12}$$

where $\otimes$ is the Kronecker product, and $\mathbf{Q}_n$ is the so-called bit-reversal permutation matrix defined as

$$\mathbf{Q}_n := (\mathbf{I}_{n/2} \otimes \mathbf{P}_2)(\mathbf{I}_{n/4} \otimes \mathbf{P}_4) \dots (\mathbf{I}_2 \otimes \mathbf{P}_{n/2})\mathbf{P}_n. \tag{3.13}$$

Lemma 3.1. *Denoting $L := \log_2(n) \in \mathbb{N}$, the sparsity patterns on each of the obtained sparse factors $\mathbf{X}_\ell := \mathbf{I}_{2^{\ell-1}} \otimes \mathbf{B}_{2^{L-\ell+1}}$ in (3.12) can be written as a Kronecker product:*

$$\forall \ell \in [\![L]\!], \quad \operatorname{supp}(\mathbf{X}_\ell) \subseteq \operatorname{supp}(\mathbf{I}_{2^{\ell-1}} \otimes \mathbf{1}_{2\times 2} \otimes \mathbf{I}_{2^{L-\ell}}). \tag{3.14}$$

In other words, up to the bit-reversal permutation of its column defined in (3.13), the DFT matrix of size $n \times n$ with $n = 2^L$ admits exactly the square dyadic butterfly factorization, as defined in (1.1).

Proof. By (3.11), we have $\operatorname{supp}(\mathbf{B}_{2^{L-\ell+1}}) \subseteq \operatorname{supp}(\mathbf{1}_{2\times 2} \otimes \mathbf{I}_{2^{L-\ell}})$ for each $\ell \in [\![L]\!]$. □

Remark 3.2. *Another way to visualize the radix-2 FFT is to construct a corresponding* directed acyclic graph *(DAG) that shows the multiplication and the addition operations for computing the DFT. A directed graph $\mathcal{G}$ is defined by a set of vertices $\mathcal{V}$ and edges $\mathcal{E} \subseteq \{(x, y) \in \mathcal{V}^2 \,|\, x \neq y\}$. A directed acyclic graph is a directed graph that does not admit a path[2] for which the starting vertex and the ending vertex is the same. An input vertex is a vertex u for which $(x, u) \notin \mathcal{E}$ for any $x \in \mathcal{V}$. An output vertex is a vertex v for which $(v, y) \notin \mathcal{E}$ for any $y \in \mathcal{V}$. A DAG with m input vertices and n output vertices, associated with a weight function $\lambda : \mathcal{E} \to \mathbb{C}$, can describe a linear operator $\mathbb{C}^m \mapsto \mathbb{C}^n$ as follows. Given some input values $\alpha(u) \in \mathbb{C}$ for each* input *vertex u, we compute recursively the values $\alpha(y)$ for any non-input vertices y as:*

$$\alpha(y) := \sum_{x \in \mathcal{V},\, (x,y) \in \mathcal{E}} \alpha(x)\lambda(x, y). \tag{3.15}$$

Then, the outputs of the corresponding linear operator are the values $\alpha(v)$ for output vertices v. In the case of the DFT of size $n = 8$, the DAG corresponding to the radix-2 FFT is described in Figure 3.1.

Conclusion. The DFT matrix of size $n \times n$ with $n := 2^L$ admits *exactly* a sparse matrix factorization into $L = \log_2(n)$ factors, where each of them has exactly two nonzero entries per row and per column. In particular, this yields the theoretical complexity of the fast Fourier transform in $\mathcal{O}(n \log n)$. The FFT algorithms are based on some algebraic features of the DFT matrix, in the sense that we used Equations (3.3), (3.4) and (3.8) based on the periodicity of $(k, i) \mapsto \omega_n^{(k-1)(i-1)}$.

[2] A path of length k is defined as a finite sequence of edges $\{(x_i, y_i)\}_{i=1}^k$ for which $y_i = x_{i+1}$ for any $i \in [\![k]\!]$. The starting vertex is x_1 and the ending vertex is y_k.

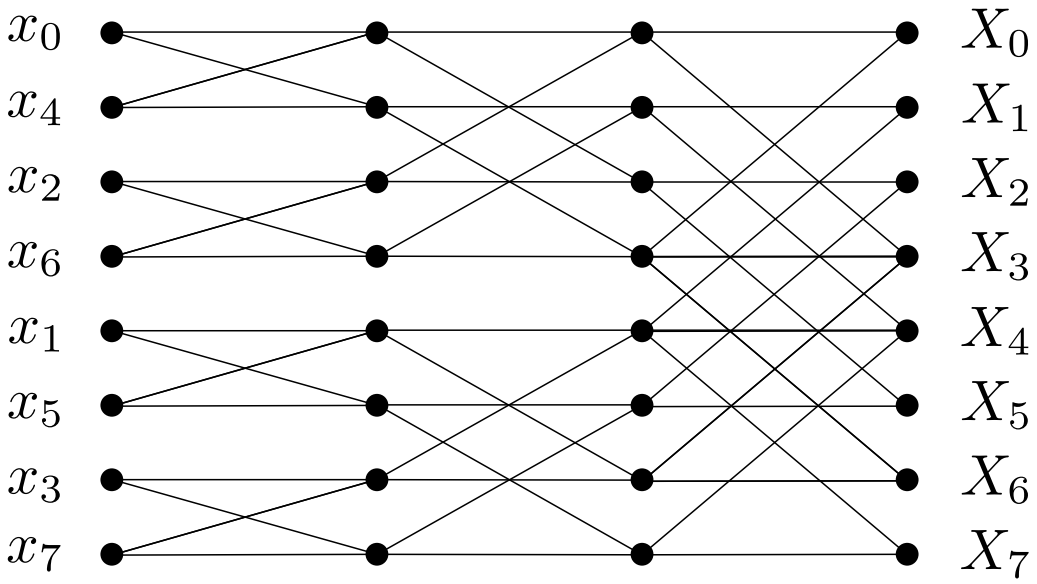


Figure 3.1: DAG associated with the radix-2 FFT for the DFT of size $n = 8$.

3.2.3 A unifying framework for structured matrices

It turns out that, under the *displacement rank* framework [191], the DFT matrix belongs to a more general family of *structured matrices* that covers the following classes of matrices widely used in many contexts.

- **Toeplitz matrices**: they are of the form $(t_{i-j})_{(i,j)\in[\![n]\!]^2}$ for some parameters $t_{-n+1}, t_{-n+2}, \ldots, t_{n-1}$, i.e., they have constant values on their diagonal. This structure corresponds to the property of shift-invariance that can appear for instance in some physical problems.

- **Hankel matrices**: they are of the form $(h_{i+j-2})_{(i,j)\in[\![n]\!]^2}$ for some parameters $h_0, h_1, \ldots, h_{2n-2}$, i.e., they have constant values on their anti-diagonal. Together with Toeplitz matrices, they are related to the one-dimensional discrete convolution operation.

- **Vandermonde matrices**: they are of the form $(x_i^{j-1})_{(i,j)\in[\![m]\!]\times[\![n]\!]}$ for some parameters $x_1, \ldots, x_m$, i.e., there is a geometric progression of the entries in each row. They appear in polynomial interpolation problems. The discrete Fourier transform (DFT) is a Vandermonde matrix of size $n \times n$ associated with the n-th complex roots of unity.

- **Cauchy matrices**: they are of the form $\left(\frac{1}{x_i - y_j}\right)_{(i,j)\in[\![m]\!]\times[\![n]\!]}$ for some parameters $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$. They appear in rational interpolation problems.

These matrices of size $m \times n$ are characterized by the fact that they can be described using far fewer than mn parameters, although they are not sparse matri-

ces in the sense that they admit only a small number of nonzero entries. More precisely, under the displacement rank framework, we say that a matrix is structured in the sense that it can be mapped to a low-rank matrix, called a *generator*, using a certain *linear displacement* operator L, and recovered easily from their image by L [191, 287, 294, 295]. Then, linear algebra operations can be performed rapidly by operating on the generator. In particular, by leveraging such a structure, it is possible to design fast algorithms for the *exact* matrix-vector multiplication in $\mathcal{O}(n \log n)$ with a Toeplitz or Hankel matrix, and $\mathcal{O}(n \log^2 n)$ with Vandermonde or Cauchy matrices [130, 294]. The description of these fast algorithms is beyond the scope of this chapter and we refer the reader to [294] for more details.

3.3 An introduction to analysis-based fast transforms

Is it still possible to build a fast algorithm in the case where we do not know that the matrix admits some algebraic structure as described in the previous section? In this section, we will review some *analysis-based fast transforms* for a rapid *approximate* evaluation of matrix-vector multiplication. The matrix is assumed to have the form $\mathbf{A} = (k(x_i, y_j))_{(i,j)\in[\![m]\!]\times[\![n]\!]}$ for a known kernel operator $k : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ evaluated at some parameters $\{x_i\}_{i=1}^m \subseteq \mathcal{X}$ and $\{y_j\}_{j=1}^n \subseteq \mathcal{Y}$. This is the case of Examples 3.2 to 3.5. We will start by presenting the fast multipole method [140], before showing its relationship to $\mathcal{H}$-matrices [149] and $\mathcal{H}^2$-matrices [151]. The key idea is to use a smoothness property of the kernel operator that translates into a low-rank property for the corresponding kernel matrix, which can then be leveraged to rapidly approximate the matrix-vector multiplication.

3.3.1 An overview of the fast multipole method

The following presentation is based on [18, 260]. Consider a set of locations $\{x_i\}_{i=1}^n \subseteq \mathbb{R}$ in a one-dimensional space[3] verifying $x_1 < x_2 < \ldots < x_n$ for n electrical charges associated with source strength $\mathbf{q} := (q_i)_{i=1}^n \in \mathbb{R}^n$. The evaluation of the potentials $\mathbf{u} := (u_i)_{i=1}^n \in \mathbb{C}^n$ at the locations $x_1, \ldots, x_n$ is given by:

$$\mathbf{u} = \mathbf{A}\mathbf{q} \quad \text{with} \quad \mathbf{A}[i,j] := -\log(x_i - x_j) \quad \forall (i,j) \in [\![n]\!]^2, \tag{3.16}$$

where we use the complex logarithm by convenience. The real potentials are then obtained by taking the real part of $\mathbf{u}$. This is exactly the scenario of Example 3.3. The fast multipole method (FMM) [140] is a fast algorithm to approximate (3.16) in $\mathcal{O}(n)$ operations instead of $\mathcal{O}(n^2)$.

[3] The method is applicable to both two-dimensional and three-dimensional spaces, but we present it in the one-dimensional case for ease of presentation.

Separable expansion and rank deficiency on well-separated domains. The FMM is an analysis-based fast transform in the sense that it exploits some smoothness property of the kernel $(x, y) \mapsto -\log(x-y)$ allowing a separable expansion, as we detail now. Suppose that x belongs to a *target* interval $\Omega_\tau \subseteq \mathbb{R}$ and y to a *source* interval $\Omega_\sigma \subseteq \mathbb{R}$. Denoting c_σ, c_τ the centers of Ω_σ, Ω_τ, we have:

$$\begin{aligned} -\log(x-y) &= -\log(x-c_\sigma) - \log\left(1 - \frac{y-c_\sigma}{x-c_\sigma}\right) \\ &= -\log(x-c_\sigma) + \sum_{p=1}^{\infty} \frac{1}{p}\left(\frac{y-c_\sigma}{x-c_\sigma}\right)^p, \end{aligned} \tag{3.17}$$

provided that $|y-c_\sigma| < |x-c_\sigma|$. This condition is verified if we assume that the intervals Ω_σ and Ω_τ are *well-separated*, e.g., in the sense that there exists $0 < \eta \leq 1$ such that

$$\eta \operatorname{dist}(\Omega_\tau, \Omega_\sigma) \geq r_\tau + r_\sigma, \tag{3.18}$$

where $\operatorname{dist}(\Omega_\tau, \Omega_\sigma) := \min_{x\in\Omega_\tau, y\in\Omega_\sigma} |x-y|$, $r_\tau := \max_{x\in\Omega_\tau} |x-c_\tau|$ and $r_\sigma := \max_{y\in\Omega_\sigma} |y-c_\sigma|$. Indeed, under this condition, we have $|x-c_\sigma| \geq \operatorname{dist}(\Omega_\tau, \Omega_\sigma) \geq \eta \operatorname{dist}(\Omega_\tau, \Omega_\sigma) \geq r_\tau + r_\sigma > r_\sigma \geq |y-c_\sigma|$, since $r_\tau > 0$ because the interval Ω_τ is not reduced to a singleton.

The *separable expansion* (3.17) tells that, up to a given error $\epsilon > 0$, the kernel $(x, y) \mapsto -\log(x-y)$ can be approximated by a truncation of order $r-1$:

$$\forall (x,y) \in \Omega_\tau \times \Omega_\sigma, \quad -\log(x-y) \approx \sum_{p=0}^{r-1} B_p(x) C_p(y), \tag{3.19}$$

where $B_0(x) := -\log(x-c_\sigma)$, $C_0(x) = 1$, $B_p(x) = (x-c_\sigma)^{-p}$ and $C_p(x) = \frac{1}{p}(y-c_\sigma)^p$ for $p \in [\![r-1]\!]$. The truncation error is roughly $\frac{1}{r}(|y-c_\sigma|/|x-c_\sigma|)^r$, which indeed decreases as the separation between the domains increases. Denote now the index subsets I_σ, I_τ of $[\![n]\!]$ defined as:

$$i \in I_\sigma \iff x_i \in \Omega_\sigma, \quad \text{and} \quad i \in I_\tau \iff x_i \in \Omega_\tau. \tag{3.20}$$

Under assumption (3.18), by (3.19), the submatrix $\mathbf{A}[I_\tau, I_\sigma]$ admits a *rank-r approximation*

$$\mathbf{A}[I_\tau, I_\sigma] \approx \mathbf{B}_\tau \mathbf{C}_\sigma^\top, \tag{3.21}$$

where $\mathbf{B}_\tau$, $\mathbf{C}_\sigma$ are matrices with r columns $(B_p(x_i))_{i\in I_\tau}$, $(C_p(x_i))_{i\in I_\sigma}$, respectively, for $p \in [\![0, r-1]\!]$. This means that matrix-vector multiplication by $\mathbf{A}[I_\tau, I_\sigma]$ can be approximated with $\mathcal{O}(r(|I_\tau| + |I_\sigma|))$ operations, which is cheaper than $\mathcal{O}(|I_\tau|\,|I_\sigma|)$ if r is much smaller than $|I_\tau|$, $|I_\sigma|$.

The general idea of the FMM. Going back to the approximation of (3.16), the main idea of the FMM is to partition the interval $\Omega := [\min_{i\in[\![n]\!]} x_i, \max_{i\in[\![n]\!]} x_i]$ containing all the particles $\{x_i\}_{i=1}^n$ into several disjoint subintervals, in such a

way that we can evaluate the interactions between well-separated domains Ω_τ, Ω_σ (*far-field interactions*) using a low-rank approximation of the submatrix $\mathbf{A}[I_\tau, I_\sigma]$, and use direct evaluation only for points that are close (*near-field interactions*). Equivalently, since we assume that $x_1 < x_2 < \ldots < x_n$, the FMM exploits the rank deficiencies in off-diagonal blocks of $\mathbf{A}$ in order to rapidly approximate (3.16).

To achieve the complexity in $\mathcal{O}(n)$ operations, the FMM further considers a *multi-level* strategy. The domain Ω is partitioned hierarchically using a notion of *cluster-tree* [150] (more details are given below), that allows us to consider different scales of interaction distance when evaluating far-field interactions, so that we can reuse computations across different levels.

3.3.2 Introducing the framework of hierarchical matrices

First, let us now detail this multi-level strategy under the framework of *hierarchical matrices*, also called *$\mathcal{H}$-matrices* [149], in order to explain how to reduce the complexity for approximating (3.16). This leads to an approximation in *quasi-linear* complexity. Then, we will see afterward how to obtain a *linear* complexity using the framework of $\mathcal{H}^2$-matrices [151] that builds upon $\mathcal{H}$-matrices. The following presentation is based on [107, 330].

Hierarchical splitting of the domain. For simpler presentation, *we assume the locations $x_1, \ldots, x_n$ are uniformly spread in the interval Ω.* We split the interval Ω into two disjoint subintervals of the same length Ω_{left} and Ω_{right} with $\Omega = \Omega_{\text{left}} \cup \Omega_{\text{right}}$. We repeat recursively this binary splitting on Ω_{left} and Ω_{right} respectively, until the number of particles per subintervals is smaller than a certain threshold ν. This yields a hierarchical splitting of the domain Ω with $L := \lceil \log_2(n/\nu) \rceil$ levels, where there are 2^ℓ subintervals denoted $\{\Omega_i^{(\ell)}\}_{i=1}^{2^\ell}$ at each level $\ell \in [\![L]\!]$ forming a partition of Ω. Then, this hierarchical partitioning of the domain leads to a hierarchical partitioning of the indices, defined by the notion of *cluster tree* for the set of indices $[\![n]\!]$.

Definition 3.3 (Cluster tree [150]). *A cluster tree T of a set of indices $[\![n]\!]$ with depth L is a tree where:*

- *nodes of the tree are subsets of $[\![n]\!]$;*
- *the root is $[\![n]\!]$;*
- *each non-leaf node has non-empty children that form a partition of their parent;*
- *the only leaves are at level L.*

We fix the convention that the root nodes are at level 0. The set of nodes in the same level ℓ of T is denoted $T(\ell)$.

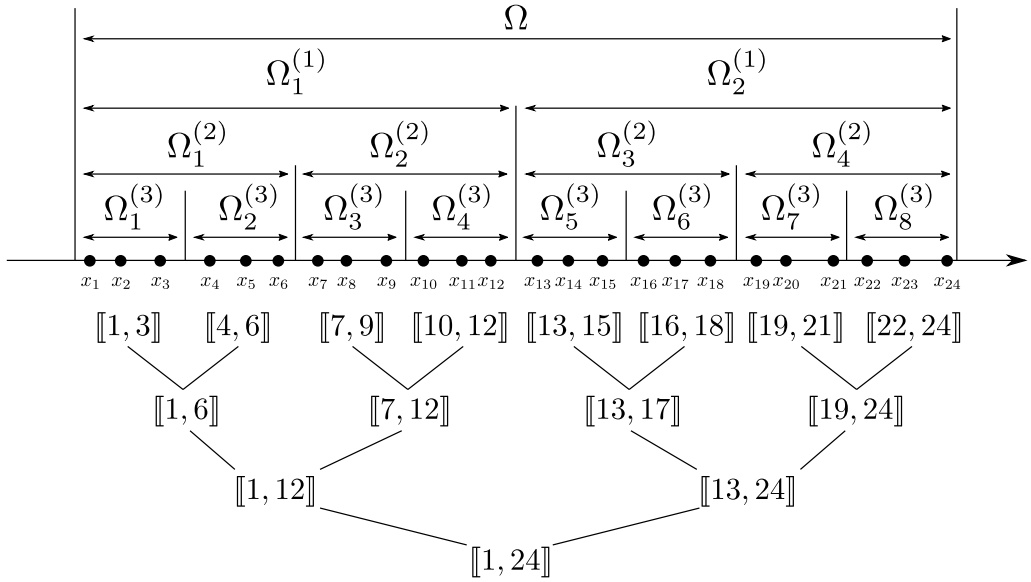


Figure 3.2: Illustration of a hierarchical splitting of an interval $\Omega \subseteq \mathbb{R}$ containing particles $\mathbf{x}_1 < \ldots < \mathbf{x}_n$ with $n = 24$. We show below the corresponding cluster tree of $[\![n]\!]$.

Remark 3.3. *Note that this definition is general and it is not restricted to binary trees only.*

The binary splitting of the domain Ω leads to a *binary cluster tree* T_Ω, i.e., each non-leaf node has exactly two children. Since we assume that the particles are uniformly spread in Ω, each node in $T_\Omega(\ell)$ at level $\ell \in [\![L]\!]$ is of the form

$$I_k^{(\ell)} := \left\{ i \in [\![n]\!] \,|\, x_k \in \Omega_k^{(\ell)} \right\}, \quad |I_k^{(\ell)}| \leq \frac{n}{2^\ell}, \tag{3.22}$$

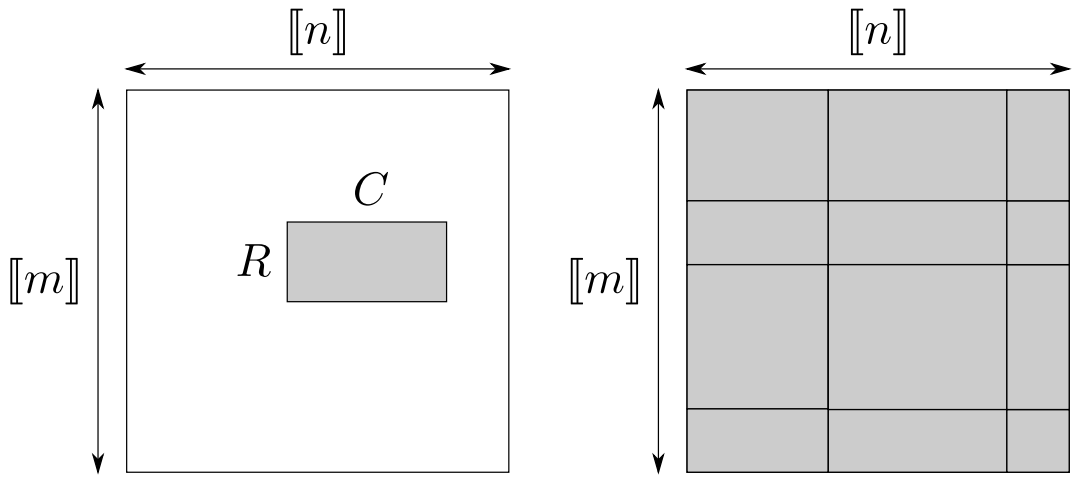
for a certain $k \in [\![2^\ell]\!]$. See Figure 3.2 for an illustration. Note that by definition, $T_\Omega(\ell)$ forms a partition of the root $[\![n]\!]$. More precisely, the cluster tree defines a hierarchy of partitions of $[\![n]\!]$, in the following sense.

Definition 3.4 (Finer and coarser partitions [150, Definition 1.11])**.** *Consider two partitions P and $\tilde{P}$ of $[\![n]\!]$. We say that P is* finer *than $\tilde{P}$ (or $\tilde{P}$ is* coarser *than P) if for all $I \in P$ there exists $\tilde{I} \in \tilde{P}$ such that $I \subseteq \tilde{I}$.*

Matrix partition. At each level $\ell \in [\![L]\!]$, since $T_\Omega(\ell)$ is a partition of $[\![n]\!]$, the family

$$T_{\Omega^2}(\ell) := \{ R \times C \,|\, R, C \in T_\Omega(\ell) \} \tag{3.23}$$

defines a partition of the matrix indices $[\![n]\!] \times [\![n]\!]$ into *index blocks*. Such a partition is called a *block partition*, and more precisely a *product block partition*. These concepts are illustrated in Figure 3.3.



(a) Index block (b) Product block partition

Figure 3.3: Illustration of an index block and a product block partition, for a size $m \times n$.

Definition 3.5 (Index block). *An* index block *(or briefly a block) B to approximate a matrix size $m \times n$ is a subset $R \times C$ where $R \subseteq [\![m]\!]$ is a subset of* row *indices and $C \subseteq [\![n]\!]$ is a subset of* column *indices.*

Definition 3.6 (Block partition). *A family of blocks $P := \{B_k\}_{k=1}^K$ forms a* block partition *of $[\![m]\!] \times [\![n]\!]$ if the blocks of the family are pairwise disjoint, and the union $\bigcup_{k=1}^K B_k$ is equal to $[\![m]\!] \times [\![n]\!]$.*

Example 3.6 (Product block partition). *The* product block partition *associated with a partition P^{row} of $[\![m]\!]$ and a partition P^{col} of $[\![n]\!]$ is the block partition $P := \{R \times C \mid R \in P^{row}, C \in P^{col}\}$ of $[\![m]\!] \times [\![n]\!]$.*

Admissible blocks. Two intervals $\Omega_k^{(\ell)}, \Omega_{k'}^{(\ell)}$ for $k, k' \in [\![2^\ell]\!]$ at a certain level $\ell \in [\![L]\!]$ are well-separated in the sense of (3.18) if, and only if, they do not overlap[4]. In this case, due to the smoothness property of the kernel $(x, y) \mapsto -\log(x - y)$, the submatrix $\mathbf{A}[I_k^{(\ell)}, I_{k'}^{(\ell)}]$ can be well-approximated by a low-rank matrix, and we will say that the index block $I_k^{(\ell)} \times I_{k'}^{(\ell)}$ is *admissible*. In our case, assuming that the intervals $\Omega_1^{(\ell)}, \ldots, \Omega_{2^\ell}^{(\ell)}$ are such that their centers are ordered with increasing values in $\mathbb{R}$, we have, as illustrated in Figure 3.4:

$$\forall k, k' \in [\![2^\ell]\!], \quad I_k^{(\ell)} \times I_{k'}^{(\ell)} \text{ is admissible } \iff |k - k'| > 1. \tag{3.24}$$

[4]Indeed, $\Omega_k^{(\ell)}, \Omega_{k'}^{(\ell)}$ are not well-separated if they are the same, or if they are neighbors in the sense that they touch each other. But otherwise, they are well-separated because in this case the distance between $\Omega_k^{(\ell)}, \Omega_{k'}^{(\ell)}$ is at least $(\max \Omega - \min \Omega)/2^\ell$, by construction of the binary hierarchical splitting of Ω.

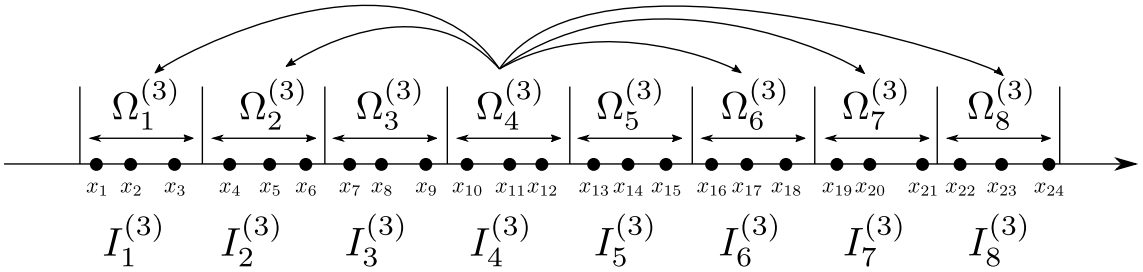


Figure 3.4: Illustration of admissible blocks from the example given in Figure 3.5. In this example, $I_4^{(3)} \times I_{k'}^{(3)}$ is admissible for $k' \in \{1, 2, 6, 7, 8\}$.

Splitting into admissible blocks. Let us now construct a specific block partition of $[\![n]\!] \times [\![n]\!]$ (Definition 3.6) that contains a minimal number of admissible blocks [150, Proposition 5.33]. We define

$$P := \text{MINIMAL\_ADMISSIBLE\_PARTITION}([\![n]\!] \times [\![n]\!], T_\Omega), \tag{3.25}$$

$$P^+ := \{B \in P \mid B \text{ is admissible}\}, \tag{3.26}$$

using the procedure described in Algorithm 3.1. The main idea is to partition each index block into admissible and non-admissible blocks, and recursively repeat the procedure only on the non-admissible blocks until we reach the last level of the cluster tree. By (3.24), one can check that $P \cap T_{\Omega^2}(\ell) = \emptyset$ for $\ell \in \{0, 1\}$. We can then perform a hierarchical splitting of the kernel matrix $\mathbf{A}$ in (3.16) as follows:

$$\mathbf{A} = \sum_{\ell=2}^{L} \mathbf{A}_\ell + \mathbf{N}_L, \tag{3.27}$$

where $\mathbf{A}_\ell$ for $\ell \in [\![2, L]\!]$ is defined as the following block matrix:

$$\forall R \times C \in T_{\Omega^2}(\ell), \quad \mathbf{A}_\ell[R, C] = \begin{cases} \mathbf{A}[R, C] & \text{if } R \times C \in P^+ \\ \mathbf{0} & \text{otherwise} \end{cases}, \tag{3.28}$$

and $\mathbf{N}_L := \mathbf{A} - \sum_{\ell=2}^{L} \mathbf{A}_\ell$. In other words, $\mathbf{A}_\ell$ contains the submatrices of $\mathbf{A}$ corresponding to admissible blocks at level ℓ in the hierarchical splitting, and $\mathbf{N}_L$ contains the submatrices of $\mathbf{A}$ corresponding to non-admissible blocks in P.

Approximation by a hierarchical matrix. As per (3.21), $\mathbf{A}_\ell[R, C]$ with $R \times C \in P^+ \cap T_{\Omega^2}(\ell)$ can be well-approximated by a rank-r matrix:

$$\mathbf{A}_\ell[R, C] \approx \hat{\mathbf{A}}_{R,C}^{(\ell)}, \quad \text{rank}\left(\hat{\mathbf{A}}_{R,C}^{(\ell)}\right) \leq r. \tag{3.29}$$

Define the block matrix $\hat{\mathbf{A}}_\ell$ as:

$$\forall R \times C \in T_{\Omega^2}(\ell), \quad \hat{\mathbf{A}}_\ell[R, C] = \begin{cases} \hat{\mathbf{A}}_{R,C}^{(\ell)} & \text{if } R \times C \in P^+ \\ \mathbf{0} & \text{otherwise} \end{cases}. \tag{3.30}$$

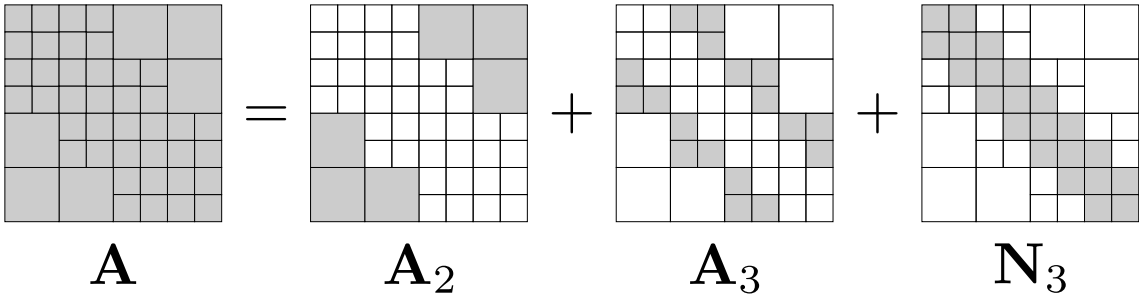


Figure 3.5: Illustration of the minimal admissible partition for the example given in Figure 3.5, and illustration of the support of $\mathbf{A}_\ell$ for $\ell \in [\![2, L]\!]$ and $\mathbf{N}_L$, in the case $L = 3$. Index blocks corresponding to nonzero submatrices are in gray, and those corresponding to zero submatrices are in white.

Algorithm 3.1 Minimal admissible partition [150, (5.44)].

```
procedure MINIMAL_ADMISSIBLE_PARTITION(R × C, cluster tree T)
    Assume that R and C are nodes of T at a same level
    if R × C is admissible or if R, C are leaves in T then
        return {R × C}
    else
        P ← ∅
        for R′, C′ children of R, C in T do
            P ← P ∪ MINIMAL_ADMISSIBLE_PARTITION(R′ × C′, T)
        end for
        return P
    end if
end procedure
```

Therefore:

$$\mathbf{A} \approx \hat{\mathbf{A}} := \sum_{\ell=2}^{L} \hat{\mathbf{A}}_\ell + \mathbf{N}_L. \tag{3.31}$$

The matrix $\hat{\mathbf{A}}$ in the right-hand side of the equation is precisely what is called in the literature a *hierarchical matrix* [149, 150], associated with the block partition P defined in (3.25), the admissibility condition (3.24) and the rank parameter r. It is a matrix with low-rank submatrices at index blocks P^+.

Matrix-vector multiplication by a hierarchical matrix. The operation $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$ is then approximated by $\mathbf{x} \mapsto \hat{\mathbf{A}}\mathbf{x}$, which can be evaluated in $\mathcal{O}(n \log n)$ operations. Indeed, the computation breaks down to:

$$\hat{\mathbf{A}}\mathbf{x} = \underbrace{\sum_{\ell=2}^{L} \hat{\mathbf{A}}_\ell \mathbf{x}}_{\text{far-field interactions}} + \underbrace{\mathbf{N}_L \mathbf{x}}_{\text{near-field interactions}}. \tag{3.32}$$

For $\ell \in [\![2, L]\!]$, the multiplication $\hat{\mathbf{A}}_\ell[R, C]\mathbf{x}[C]$ for $R \times C \in P^+ \cap T_{\Omega^2}(\ell)$ requires $r(|R| + |C|)$ operations. By (3.22), it requires at most $2rn/2^\ell$. But by (3.24), one can check that the cardinality of $P^+ \cap T_{\Omega^2}(\ell)$ is bounded by $\gamma 2^\ell$ with $\gamma = 3$. Therefore, the far-field evaluation $\hat{\mathbf{A}}_\ell\mathbf{x}$ requires at most $(\gamma 2^\ell)(2rn/2^\ell) = 2r\gamma n$ operations. Similarly, by (3.24), the only non-admissible blocks in P are blocks in $T_{\Omega^2}(L)$, and there are at most $\gamma 2^L$ non-admissible blocks with $\gamma = 3$. Therefore, the near-field evaluation $\mathbf{N}_L\mathbf{x}$ requires at most $(\gamma 2^L)(n/2^L)^2 = \gamma n^2/2^L \leq \gamma \nu n$ by definition of $L := \lceil \log_2(n/\nu) \rceil$. In conclusion, the total number of operations for $\hat{\mathbf{A}}\mathbf{x}$ is at most

$$(L-1)(2r\gamma n) + \gamma \nu n = \mathcal{O}(n \log n), \tag{3.33}$$

where γ, r and ν are assumed to be constant.

3.3.3 Revisiting the FMM with $\mathcal{H}^2$-matrices

The FMM, however, requires only $\mathcal{O}(n)$ operations for approximating (3.16), instead of $\mathcal{O}(n \log n)$. This can be interpreted by the fact that $\mathbf{A}$ in (3.16) turns out to be well-approximated by a so-called *$\mathcal{H}^2$-matrix* [151]. The $\mathcal{H}^2$-matrices form a subclass of *uniform $\mathcal{H}$-matrices*, which itself is a subclass of $\mathcal{H}$-matrices. We now explain what it means for $\mathbf{A}$ to be well-approximated by a uniform $\mathcal{H}$-matrix and an $\mathcal{H}^2$-matrix.

Approximation by a uniform $\mathcal{H}$-matrix. At level $\ell \in [\![2, L]\!]$, it can be shown [260, (14)] that there exist row bases $\mathbf{U}_R \in \mathbb{C}^{|R| \times r}$ for each $R \in T_\Omega(\ell)$ and column bases $\mathbf{V}_C \in \mathbb{C}^{|C| \times r}$ for each $C \in T_\Omega(\ell)$ such that the rank-r matrix $\hat{\mathbf{A}}^{(\ell)}_{R,C}$ approximating $\mathbf{A}[R, C]$ for $R \times C \in T_{\Omega^2}(\ell) \cap P^+$ in (3.29) can be chosen as:

$$\hat{\mathbf{A}}^{(\ell)}_{R,C} := \mathbf{U}_R \mathbf{B}_{R,C} \mathbf{V}_C^\top, \quad \text{for a certain} \quad \mathbf{B}_{R,C} \in \mathbb{C}^{r \times r}. \tag{3.34}$$

Note that this is stronger than merely applying SVD to each index block $R \times C \in T_{\Omega^2}(\ell) \cap P^+$ for $\ell \in [\![2, L]\!]$. Then, the obtained approximation $\hat{\mathbf{A}} := \sum_{\ell=2}^L \hat{\mathbf{A}}_\ell + \mathbf{N}_L$ is what is called a *uniform $\mathcal{H}$-matrix* [151]. The row and column bases spanning the low-rank off-diagonal submatrices of $\hat{\mathbf{A}}$ are *shared* across different admissible blocks, with the hope of reducing the computational work when multiplying by $\hat{\mathbf{A}}$. However, this still requires a total of $\mathcal{O}(n \log n)$ operations [151].

Approximation by an $\mathcal{H}^2$-matrix. To further reduce the computational work, it turns out [260, Equations (15), (16)] that the row bases $\{\mathbf{U}_R \mid R \in T_\Omega(\ell)\}$ and the column bases $\{\mathbf{V}_C \mid C \in T_\Omega(\ell)\}$ at each level $\ell \in [\![2, L]\!]$ can be chosen in such a way that they satisfy the following so-called *consistency condition* [151]:

1. for any non-leaf node R of T_Ω with children R_1, R_2,

$$\forall i \in \{1, 2\}, \quad \exists \mathbf{T}_{R_i,R} \in \mathbb{C}^{r \times r}, \quad \mathbf{U}_R[R_i, :] = \mathbf{U}_{R_i} \mathbf{T}_{R_i,R}; \tag{3.35}$$

2. for any non-leaf node C of T_Ω with children C_1, C_2,

$$\forall i \in \{1,2\}, \quad \exists \mathbf{T}_{C_i,C} \in \mathbb{C}^{r\times r}, \quad \mathbf{V}_C[C_i,:] = \mathbf{V}_{C_i}\mathbf{T}_{C_i,C}. \tag{3.36}$$

The obtained approximation $\hat{\mathbf{A}} := \sum_{\ell=2}^{L} \hat{\mathbf{A}}_\ell + \mathbf{N}_L$ is then called an *$\mathcal{H}^2$-matrix* [151]. The row and column bases spanning submatrices of $\hat{\mathbf{A}}$ corresponding to admissible blocks are *nested* across the different levels of the hierarchy, in the sense of (3.35) and (3.35), so that data can be reused to reduce the computational work. The multiplication algorithm by an $\mathcal{H}^2$-matrix [151] coincides in fact with the idea of the FMM [140, 260], and has a complexity in $\mathcal{O}(n)$. As we will see below, nested bases are also used in butterfly algorithms but in a different manner.

3.4 The low-rank property in the butterfly algorithm

Butterfly algorithms form an alternative class of analysis-based fast-transforms that rely on another form of smoothness property than the FMM. Typically, the considered kernel operator leads to a kernel matrix of size $n \times n$ for which *the numerical rank of any $p \times q$ contiguous submatrix is proportional to pq/n* [40, 288, 289]. This contrasts with $\mathcal{H}$-matrices, for which only the off-diagonal submatrices are rank-deficient. In this section, we start by listing some kernel operators associated with such a low-rank property, such as the kernel in the Fourier transform. Then, we will see that these kernel matrices satisfy in fact the so-called *complementary low-rank property* for some specific cluster trees [234], which is the central property in the butterfly algorithm.

3.4.1 When the rank of a submatrix scales with its area

It turns out that the DFT matrix obeys the following special low-rank property [16, 92, 288, 398].

Theorem 3.1 (Rank of submatrices in the DFT matrix [288])**.** *Let $0 < \epsilon < 0.1$ and n be an integer. Then, for any $p, q \in [\![n]\!]$ such that $pq/n > 10/\pi$, the submatrix of the DFT matrix of size $n \times n$ restricted to its first p rows and q columns can be well approximated up to precision ϵ by a low-rank matrix of rank $C(\epsilon)\frac{pq}{n}$, where $C(\epsilon) := \pi(1+\alpha 10^{-2/3})^3$ and $\alpha = \frac{3^{-1/3}}{2}\log^{2/3}(1/\epsilon)$.*

Remark 3.4. *This theorem is true even if n is a prime integer.*

Note that any contiguous $p \times q$ submatrix of the DFT matrix of size $n \times n$ is proportional to the one containing the first p rows and q columns of the DFT matrix, so they have the same rank. In other words, provided that pq/n is sufficiently large, the numerical ranks of all contiguous $p \times q$ submatrices of the DFT matrix are roughly the same and proportional to pq/n. This low-rank property

of the DFT matrix comes from the fact that the kernel $(\rho, \tau) \mapsto e^{\mathrm{j}\rho\tau}$ admits the following separable expansion.

Lemma 3.2 (From [288, Lemma 3.2]). *Denoting $\{J_n\}_n$ the Bessel function of the first kind and $\{T_n\}_n$ the Chebyshev polynomials of the first kind, we have:*

$$\forall (\rho, \tau) \in \mathbb{R} \times [-1, 1], \quad e^{\mathrm{j}\rho\tau} = J_0(\rho) + \sum_{n=1}^{\infty} 2\mathrm{j}^n J_n(\rho) T_n(\tau). \tag{3.37}$$

The proof of Theorem 3.1 relies on some rescaling of the complex exponential in order to leverage the expansion of Lemma 3.2, as we now detail.

Proof of Theorem 3.1. The entries of the submatrix restricted to the first p rows and q columns in the DFT matrix of size n are given by:

$$\forall (k, i) \in [\![p]\!] \times [\![q]\!], \quad e^{-\mathrm{j}\frac{2\pi}{n}(k-1)(i-1)} = e^{\mathrm{j}\omega_k t_i},$$

where $\omega_k := (k-1) \in [0, p]$ and $t_i := -\frac{2\pi}{n}(i-1) \in [-2\pi q/n, 0]$. Consider $\tilde{\omega}_k := \frac{\omega_k}{p} \in [0, 1]$ and $\tilde{t}_i := 1 + \frac{n}{\pi q} t_i \in [-1, 1]$. Then:

$$e^{\mathrm{j}\omega_k t_i} = e^{\mathrm{j}p\tilde{\omega}_k \frac{\pi q}{n}(\tilde{t}_i - 1)} = e^{-\mathrm{j}c\tilde{\omega}_k} e^{\mathrm{j}c\tilde{\omega}_k \tilde{t}_i}, \quad \text{with} \quad c := \pi \frac{pq}{n}. \tag{3.38}$$

By Lemma 3.2:

$$e^{\mathrm{j}\omega_k t_i} = e^{-\mathrm{j}c\tilde{\omega}_k} \left(J_0(c\tilde{\omega}_k) + \sum_{n=1}^{\infty} 2\mathrm{j}^n J_n(c\tilde{\omega}_k) T_n(\tilde{t}_i) \right). \tag{3.39}$$

This means that the submatrix $(e^{j\omega_k t_i})_{(k,i) \in [\![p]\!] \times [\![q]\!]}$ can be approximated by a rank-r matrix, when truncating the series in (3.38) at order $r - 1$. The truncation error is roughly:

$$\left| e^{-\mathrm{j}p\tilde{\omega}_k} 2\mathrm{j}^r J_r(c\tilde{\omega}_k) T_r(\tilde{t}_i) \right| \leq 2 \left| J_r(c\tilde{\omega}_k) \right|. \tag{3.40}$$

By assumption, $c > 10$, and by construction, $0 \leq \tilde{\omega}_k \leq 1$ for any $k \in [\![p]\!]$. Applying [288, Lemma 2.6] yields:

$$r \geq (c^{1/3} + \alpha c^{-1/3})^3 \implies \forall k \in [\![p]\!], \, |J_r(c\tilde{\omega}_k)| \leq \epsilon. \tag{3.41}$$

Since $(c^{1/3} + \alpha c^{-1/3})^3 = c(1 + \alpha c^{-2/3})^3 \leq c(1 + \alpha 10^{-2/3})^3 = C(\epsilon)\frac{pq}{n}$, it is sufficient to consider $r \geq C(\epsilon)\frac{pq}{n}$ in order to obtain $|J_r(c\tilde{\omega}_k)| \leq \epsilon$ for any $k \in [\![p]\!]$. In conclusion, the numerical rank up to precision roughly ϵ of $(e^{j\omega_k t_i})_{(k,i) \in [\![p]\!] \times [\![q]\!]}$ is $C(\epsilon)\frac{pq}{n}$. □

It has been shown that several other kernel matrices satisfy a similar low-rank property as in Theorem 3.1, like:

- the kernel matrix of $(x,t) \mapsto xJ_m(xt)$ associated with the Fourier-Bessel transform for an integer m [289], where J_m denotes the Bessel function of order m;
- or the kernel matrix of $(x,p) \mapsto e^{2\pi \mathrm{j} n \Psi(x,p)}$ associated with the Fourier integral operator [40], where Ψ is some smooth function.

3.4.2 The complementary low-rank property

Consider a matrix $\mathbf{A}$ of size $n \times n$ for which any contiguous $p \times q$ submatrix have the same numerical rank $C(\epsilon)pq/n$ up to precision ϵ for some given $C(\cdot)$. Under the framework of [234], it also satisfies the *complementary low-rank property* in the following sense.

Definition 3.7 (Classical complementary low-rank matrix [234])**.** *Consider two cluster trees (Definition 3.3) T^{row}, T^{col} with the same depth $L \geq 1$. A matrix $\mathbf{A}$ satisfies the* complementary low-rank property *for (T^{row}, T^{col}) if the numerical rank of $\mathbf{A}[R,C]$ is low for all $(R,C) \in \bigcup_{\ell=1}^{L} T^{row}(\ell) \times T^{col}(L-\ell+1)$.*

Remark 3.5. *This definition does not necessarily assume that the cluster trees T^{row}, T^{col} are dyadic or quadtrees, as it was assumed in [234, 236]. In this definition, a low numerical rank means that the submatrix can be approximated up to a certain precision error $\epsilon > 0$ by a rank-r matrix, where r is small with respect to the size of the matrix.*

Indeed, consider for instance a binary cluster tree T (Definition 3.3) on the set of indices $[\![n]\!]$ of depth $L := \lceil \log_2(n/\nu) \rceil$ for an integer ν, such that the two children of a non-leaf node are subsets contain consecutive integers and have roughly the same cardinality[5]. By construction, the cardinality of each node at level ℓ is roughly $n/2^\ell$. Hence, the area of any $(R,C) \in T(\ell) \times T(L-\ell+1)$ is

$$\frac{|R|\,|C|}{n} \approx \frac{1}{n}\frac{n}{2^\ell}\frac{n}{2^{L-\ell+1}} = \frac{n}{2^{L+1}} \leq \frac{\nu}{2}. \tag{3.42}$$

This means that the numerical rank of $\mathbf{A}[R,C]$ is at most $C(\epsilon)\nu/2$. In other words, $\mathbf{A}$ satisfies the complementary low-rank property for the rank $C(\epsilon)\nu/2$ and the cluster trees $T^{\mathrm{row}} = T$ and $T^{\mathrm{col}} = T$.

Remark 3.6. *Definition 3.7 is more general than the definition of the complementary low-rank originally given in [234] or [236], as it does not assume the cluster trees T^{row}, T^{col} to be dyadic or quadtrees.*

In the next section, we explain how the butterfly algorithm leverages this complementary low-rank property in order to construct a butterfly factorization to enable rapid matrix-vector multiplication.

[5] More precisely, we require that the cardinality of a child I of a parent node $\tilde{I}$ is at most $\lceil \tilde{I}/2 \rceil$.

3.5 Description of the classical butterfly algorithm

Constructing a fast algorithm for approximate matrix-vector multiplication associated with butterfly factorization involves two steps.

1. **The precomputation step**: we compress an $n \times n$ matrix $\mathbf{A}$ by building an approximation $\hat{\mathbf{A}}$ with a small approximation error $\|\mathbf{A} - \hat{\mathbf{A}}\|$, such that $\hat{\mathbf{A}}$ admits a factorization into several sparse factors $\mathbf{X}_1, \ldots, \mathbf{X}_L$ with $L = \mathcal{O}(\log n)$. This precomputation step presumes that the target matrix $\mathbf{A}$ satisfies the complementary low-rank property (Definition 3.7), and follows a procedure, called *butterfly algorithm*, that returns the sparse factors $\mathbf{X}_1, \ldots, \mathbf{X}_L$ that we want.

2. **The application step**: it uses the precomputed $\mathbf{X}_1, \ldots, \mathbf{X}_L$ in order to approximate the matrix-vector multiplication $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$, by performing successive matrix multiplication by the sparse factor $\mathbf{X}_\ell$ for $\ell \in [\![L]\!]$.

Some assumptions for easier presentation. To simplify the description of the butterfly algorithm, let us assume that the target matrix $\mathbf{A}$ is a square matrix of size $n \times n$, and satisfies the complementary low-rank property for two cluster trees T^{row} and T^{col} of depth $L := \lfloor \log_2(n/\nu) \rfloor$ for a certain constant ν much smaller than n. We also assume for simplicity that the two trees are *dyadic*, in the sense that each non-leaf node has exactly two children. Moreover, up to some row and column permutations of $\mathbf{A}$, we can assume without loss of generality that the indices in the left child of a given non-leaf node are smaller than the one of the right child, or vice versa. We assume a *noiseless* setting: there exists a certain integer r such that the rank of $\mathbf{A}[R, C]$ is *at most* r for all $(R, C) \in \bigcup_{\ell=1}^{L} T^{\text{row}}(\ell) \times T^{\text{col}}(L - \ell + 1)$. Finally, we introduce the following notation:

$$\forall \ell \in [\![L]\!], \quad P_\ell^{\text{row}} := T^{\text{row}}(\ell), \quad P_\ell^{\text{col}} := T^{\text{col}}(\ell). \tag{3.43}$$

General idea. Essentially, the butterfly algorithm leverages the complementary low-rank property to build low-rank approximations of submatrices in the target matrix $\mathbf{A}$ in a hierarchical way. The column and row bases spanning these low-rank submatrices are *nested* across different levels of the hierarchy, in a way that is reminiscent of (3.35) and (3.36). First, we explain how these nested bases are constructed. Then, we explain how we use these nested bases to build a butterfly factorization approximating the target matrix.

3.5.1 Construction of low-rank approximations

The construction of the bases is performed in an inductive manner with L steps in total [289].

Step $\ell = 1$. By the *exact* complementary low-rank property, for each $(R, C) \in P_1^{\text{row}} \times P_L^{\text{row}}$, the rank of the submatrix $\mathbf{A}[R, C]$ is at most r. This means that there exist a row basis $\mathbf{V}_{R,C} \in \mathbb{C}^{|C| \times r}$ and a column basis $\mathbf{U}_{R,C} \in \mathbb{C}^{|R| \times r}$ with r columns such that

$$\mathbf{A}[R, C] = \mathbf{U}_{R,C} \mathbf{V}_{R,C}{}^*, \quad \text{range}(\mathbf{U}_{R,C}) \subseteq \text{range}(\mathbf{A}[R, C]). \tag{3.44}$$

One possible choice is to compute the truncated singular value decomposition (SVD) at order r as $\mathbf{A}[R, C] = \mathbf{U}\mathbf{D}\mathbf{V}^*$, and set $\mathbf{V}_{R,C} := \mathbf{V}$, $\mathbf{U}_{R,C} := \mathbf{A}[R, C]\mathbf{V}_{R,C}$.

Step $\ell \in [\![2, L]\!]$. Suppose that we have constructed bases $\mathbf{U}_{R,C} \in \mathbb{C}^{|R| \times r}$ for each $(R, C) \in P_{\ell-1}^{\text{row}} \times P_{L-\ell+2}^{\text{col}}$ such that $\text{range}(\mathbf{U}_{R,C}) \subseteq \text{range}(\mathbf{A}[R, C])$. Let us explain how we construct new bases $\mathbf{U}_{R,C} \in \mathbb{C}^{|R| \times r}$ for $(R, C) \in P_\ell^{\text{row}} \times P_{L-\ell+1}^{\text{col}}$ such that $\text{range}(\mathbf{U}_{R,C}) \subseteq \text{range}(\mathbf{A}[R, C])$, with some corresponding row bases $\mathbf{V}_{R,C} \in \mathbb{C}^{2r \times r}$.

Let $(R, C) \in P_{\ell-1}^{\text{row}} \times P_{L-\ell+1}^{\text{col}}$. Since the trees T^{row}, T^{col} of depth L are assumed to be *dyadic*, R and C both have *two* children, denoted $R_1, R_2 \in P_\ell^{\text{row}}$ for R and $C_1, C_2 \in P_{L-\ell+2}^{\text{col}}$ for C.

$$\begin{pmatrix} \mathbf{U}_{R,C_1} & \mathbf{U}_{R,C_2} \end{pmatrix} = \begin{pmatrix} \mathbf{U}_{R,C_1}[R_1, :] & \mathbf{U}_{R,C_2}[R_1, :] \\ \mathbf{U}_{R,C_1}[R_2, :] & \mathbf{U}_{R,C_2}[R_2, :] \end{pmatrix} \in \mathbb{C}^{|R| \times 2r}, \tag{3.45}$$

where the left-hand side is a concatenation of two matrices, and the right-hand side is a block matrix with four blocks.

But for $j \in \{1, 2\}$, since $(R, C_j) \in P_{\ell-1}^{\text{row}} \times P_{L-\ell+2}^{\text{col}}$, we have, by assumption, $\text{range}(\mathbf{U}_{R,C_j}) \subseteq \text{range}(\mathbf{A}[R, C_j])$. A fortiori, since $R_i \subseteq R$ for $i \in \{1, 2\}$, we have $\text{range}(\mathbf{U}_{R,C_j}[R_i, :]) \subseteq \text{range}(\mathbf{A}[R_i, C_j])$. Therefore:

$$\text{range}(\begin{pmatrix} \mathbf{U}_{R,C_1}[R_i, :] & \mathbf{U}_{R,C_2}[R_i, :] \end{pmatrix}) \subseteq \text{range}(\mathbf{A}[R_i, C]), \tag{3.46}$$

because $C = C_1 \cup C_2$. By definition of the complementary low-rank property, since $(R_i, C) \in P_\ell^{\text{row}} \times P_{L-\ell+1}^{\text{col}}$, the submatrix $\mathbf{A}[R_i, C]$ is of rank at most r. Consequently, the rank of $\begin{pmatrix} \mathbf{U}_{R,C_1}[R_i, :] & \mathbf{U}_{R,C_2}[R_i, :] \end{pmatrix} \in \mathbb{C}^{|R_i| \times 2r}$ is also smaller than r, so there exist a row basis $\mathbf{V}_{R_i,C} \in \mathbb{C}^{2r \times r}$ and a column basis $\mathbf{U}_{R_i,C} \in \mathbb{C}^{|R_i| \times r}$ with r columns such that

$$\begin{gathered} \begin{pmatrix} \mathbf{U}_{R,C_1}[R_i, :] & \mathbf{U}_{R,C_2}[R_i, :] \end{pmatrix} = \mathbf{U}_{R_i,C} \mathbf{V}_{R_i,C}{}^*, \\ \text{with } \text{range}(\mathbf{U}_{R_i,C}) \subseteq \text{range}(\mathbf{A}[R_i, C]). \end{gathered} \tag{3.47}$$

In conclusion, we obtain the crucial step in the butterfly algorithm:

$$\forall (R, C) \in P_{\ell-1}^{\text{row}} \times P_{L-\ell+1}^{\text{col}}, \quad \begin{pmatrix} \mathbf{U}_{R,C_1} & \mathbf{U}_{R,C_2} \end{pmatrix} = \begin{pmatrix} \mathbf{U}_{R_1,C} \mathbf{V}_{R_1,C}^* \\ \mathbf{U}_{R_2,C} \mathbf{V}_{R_2,C}^* \end{pmatrix}, \tag{3.48}$$

where R_1, R_2 are the children of R, and C_1, C_2 are the children of C. In other words, the considered bases are nested, in the sense that the restriction of the two bases $\mathbf{U}_{R,C_1}, \mathbf{U}_{R,C_2}$ on the rows R_i are expressed with a shared basis $\mathbf{U}_{R_i,C}$, for each $i \in \{1, 2\}$.

This ends the inductive construction of the column bases $\mathbf{U}_{R,C}$ for all $(R, C) \in P_\ell^{\text{row}} \times P_{L-\ell+1}^{\text{col}}$ and all $\ell \in [\![L]\!]$, with the corresponding row bases $\mathbf{V}_{R,C}$.

3.5.2 Construction of the butterfly factorization

We now explain how these bases can be used to construct a sparse factorization of the target matrix $\mathbf{A}$, yielding the so-called *butterfly factorization* of $\mathbf{A}$. At step $\ell = 1$, we construct two sparse factors $\mathbf{X}_1, \mathbf{Y}_1$ such that $\mathbf{A} = \mathbf{X}_1 \mathbf{Y}_1$. Then, for $\ell > 1$, we construct $\mathbf{X}_\ell, \mathbf{Y}_\ell$ such that $\mathbf{X}_{\ell-1} = \mathbf{X}_\ell \mathbf{Y}_\ell$. Unrolling recursively the constructions gives:

$$\begin{aligned} \mathbf{A} &= \mathbf{X}_1 \mathbf{Y}_1 \\ &= \mathbf{X}_2 \mathbf{Y}_2 \mathbf{Y}_1 \\ &= \dots \\ &= \mathbf{X}_L \mathbf{Y}_L \dots \mathbf{Y}_1. \end{aligned}$$

Let us now explain how we build these sparse factors from the constructed bases $\mathbf{U}_{R,C}$ and $\mathbf{V}_{R,C}$ above, with $(R, C) \in P_\ell^{\text{row}} \times P_{L-\ell+1}^{\text{col}}$ for $\ell \in [\![L]\!]$. This part is technical and can be skipped on a first reading.

Step $\ell = 1$. Define for each $R \in P_1^{\text{row}}$:

$$\begin{aligned} \mathbf{U}_{R,P_L^{\text{col}}} &:= \begin{pmatrix} \cdots & \mathbf{U}_{R,C} & \cdots \end{pmatrix}_{C \in P_L^{\text{col}}}, \\ \mathbf{V}_{R,P_L^{\text{col}}} &:= \begin{pmatrix} \ddots & & 0 \\ & \mathbf{V}_{R,C}{}^* & \\ 0 & & \ddots \end{pmatrix}_{C \in P_L^{\text{col}}}. \end{aligned} \tag{3.49}$$

Then, the matrix $\mathbf{A}$ is equal to

$$\begin{aligned}\begin{pmatrix} \vdots \\ (\cdots \quad \mathbf{A}[R,C] \quad \cdots)_{C\in P_L^{\mathrm{col}}} \\ \vdots \end{pmatrix}_{R\in P_1^{\mathrm{row}}} &\overset{(3.44)}{=} \begin{pmatrix} \vdots \\ (\cdots \quad \mathbf{U}_{R,C}\mathbf{V}_{R,C}{}^* \quad \cdots)_{C\in P_L^{\mathrm{col}}} \\ \vdots \end{pmatrix}_{R\in P_1^{\mathrm{row}}} \\ &\overset{(1)+(3.49)}{=} \begin{pmatrix} \vdots \\ \mathbf{U}_{R,P_L^{\mathrm{col}}}\mathbf{V}_{R,P_L^{\mathrm{col}}} \\ \vdots \end{pmatrix}_{R\in P_1^{\mathrm{row}}} \\ &\overset{(2)}{=} \mathbf{X}_1\mathbf{Y}_1,\end{aligned}$$

where we defined

$$\mathbf{X}_1 := \begin{pmatrix} \ddots & & 0 \\ & \mathbf{U}_{R,P_L^{\mathrm{col}}} & \\ 0 & & \ddots \end{pmatrix}_{R\in P_1^{\mathrm{row}}}, \quad \mathbf{Y}_1 := \begin{pmatrix} \vdots \\ \mathbf{V}_{R,P_L^{\mathrm{col}}} \\ \vdots \end{pmatrix}_{R\in P_1^{\mathrm{row}}}. \tag{3.50}$$

Step $\ell \in [\![2, L]\!]$**.** Suppose that we have constructed $\mathbf{X}_{\ell-1}$ of the form

$$\mathbf{X}_{\ell-1} = \begin{pmatrix} \ddots & & 0 \\ & \mathbf{U}_{R,P_{L-\ell+2}^{\mathrm{col}}} & \\ 0 & & \ddots \end{pmatrix}_{R\in P_{\ell-1}^{\mathrm{row}}}, \tag{3.51}$$

where we defined for $R \in P_{\ell-1}^{\mathrm{row}}$:

$$\mathbf{U}_{R,P_{L-\ell+2}^{\mathrm{col}}} := (\cdots \quad \mathbf{U}_{R,C} \quad \cdots)_{C\in P_{L-\ell+2}^{\mathrm{col}}}. \tag{3.52}$$

Let $R \in P_{\ell-1}^{\mathrm{row}}$, and denote R_1, R_2 the two children of R in the dyadic tree T^{row}, so that $R = R_1 \cup R_2$. Then, without loss of generality, we can rewrite $\mathbf{U}_{R,P_{L-\ell+2}^{\mathrm{col}}}$ as:

$$\mathbf{U}_{R,P_{L-\ell+2}^{\mathrm{col}}} = (\cdots \quad (\mathbf{U}_{R,C_1} \quad \mathbf{U}_{R,C_2}) \quad \cdots)_{C=C_1\cup C_2,\, C\in P_{L-\ell+1}^{\mathrm{col}}}, \tag{3.53}$$

where C_1, C_2 are the two children of $C \in P_{L-\ell+1}^{\mathrm{col}}$. Define for $i \in \{1, 2\}$:

$$\begin{aligned}\mathbf{U}_{R_i,P_{L-\ell+1}^{\mathrm{col}}} &:= (\cdots \quad \mathbf{U}_{R_i,C} \quad \cdots)_{C\in P_{L-\ell+1}^{\mathrm{col}}}, \\ \mathbf{V}_{R_i,P_{L-\ell+1}^{\mathrm{col}}} &:= \begin{pmatrix} \ddots & & 0 \\ & \mathbf{V}_{R_i,C}{}^* & \\ 0 & & \ddots \end{pmatrix}_{C\in P_{L-\ell+1}^{\mathrm{col}}}.\end{aligned} \tag{3.54}$$

Using the key step of the butterfly algorithm (3.48), we have:

$$\begin{aligned}
\mathbf{U}_{R,P_{L-\ell+2}^{\mathrm{col}}} &\overset{(3.48)}{=} \begin{pmatrix} \cdots & \begin{pmatrix} \mathbf{U}_{R_1,C}\mathbf{V}_{R_1,C}{}^{*} \\ \mathbf{U}_{R_2,C}\mathbf{V}_{R_2,C}{}^{*} \end{pmatrix} & \cdots \end{pmatrix}_{C\in P_{L-\ell+1}^{\mathrm{col}}} \\
&= \begin{pmatrix} \begin{pmatrix} \cdots & \mathbf{U}_{R_1,C}\mathbf{V}_{R_1,C}{}^{*} & \cdots \end{pmatrix}_{C\in P_{L-\ell+1}^{\mathrm{col}}} \\ \begin{pmatrix} \cdots & \mathbf{U}_{R_2,C}\mathbf{V}_{R_2,C}{}^{*} & \cdots \end{pmatrix}_{C\in P_{L-\ell+1}^{\mathrm{col}}} \end{pmatrix} \\
&\overset{(1)+(3.54)}{=} \begin{pmatrix} \mathbf{U}_{R_1,P_{L-\ell+1}^{\mathrm{col}}} & \mathbf{V}_{R_1,P_{L-\ell+1}^{\mathrm{col}}} \\ \mathbf{U}_{R_2,P_{L-\ell+1}^{\mathrm{col}}} & \mathbf{V}_{R_2,P_{L-\ell+1}^{\mathrm{col}}} \end{pmatrix} \\
&\overset{(2)}{=} \begin{pmatrix} \mathbf{U}_{R_1,P_{L-\ell+1}^{\mathrm{col}}} & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_{R_2,P_{L-\ell+1}^{\mathrm{col}}} \end{pmatrix} \begin{pmatrix} \mathbf{V}_{R_1,P_{L-\ell+1}^{\mathrm{col}}} \\ \mathbf{V}_{R_2,P_{L-\ell+1}^{\mathrm{col}}} \end{pmatrix},
\end{aligned} \tag{3.55}$$

In conclusion:

$$\mathbf{X}_{\ell-1} \overset{(3.51)}{=} \begin{pmatrix} \ddots & & 0 \\ & \mathbf{U}_{R,P_{L-\ell+2}^{\mathrm{col}}} & \\ 0 & & \ddots \end{pmatrix}_{R\in P_{\ell-1}^{\mathrm{row}}} \overset{(3.55)}{=} \mathbf{X}_\ell \mathbf{Y}_\ell,$$

where we defined

$$\begin{aligned}
\mathbf{X}_\ell &:= \begin{pmatrix} \ddots & & & 0 \\ & \begin{pmatrix} \mathbf{U}_{R_1,P_{L-\ell+1}^{\mathrm{col}}} & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_{R_2,P_{L-\ell+1}^{\mathrm{col}}} \end{pmatrix} & & \\ 0 & & & \ddots \end{pmatrix}_{\substack{R=R_1\cup R_2 \\ R\in P_{\ell-1}^{\mathrm{row}}}}, \\
\mathbf{Y}_\ell &:= \begin{pmatrix} \ddots & & 0 \\ & \begin{pmatrix} \mathbf{V}_{R_1,P_{L-\ell+1}^{\mathrm{col}}} \\ \mathbf{V}_{R_2,P_{L-\ell+1}^{\mathrm{col}}} \end{pmatrix} & \\ 0 & & \ddots \end{pmatrix}_{\substack{R=R_1\cup R_2 \\ R\in P_{\ell-1}^{\mathrm{row}}}}.
\end{aligned} \tag{3.56}$$

We indeed have:

$$\mathbf{X}_\ell = \begin{pmatrix} \ddots & & 0 \\ & \mathbf{U}_{R,P_{L-\ell+1}^{\mathrm{col}}} & \\ 0 & & \ddots \end{pmatrix}_{R\in P_\ell^{\mathrm{row}}}, \tag{3.57}$$

with

$$\forall R \in P_\ell^{\mathrm{row}}, \quad \mathbf{U}_{R,P_{L-\ell+1}^{\mathrm{col}}} = \begin{pmatrix} \cdots & \mathbf{U}_{R,C} & \cdots \end{pmatrix}_{C\in P_{L-\ell+1}^{\mathrm{col}}}. \tag{3.58}$$

Therefore, by induction, we obtain the butterfly factorization

$$\mathbf{A} = \mathbf{X}_L \mathbf{Y}_L \dots \mathbf{Y}_1. \tag{3.59}$$

3.5.3 Complexity analysis

The time complexity for the precomputation step (construction of the sparse factors) is $\mathcal{O}(n^2)$, while the time for the application step (multiplication by the product of sparse factors) is $\mathcal{O}(n \log n)$.

We detail how we derive these complexity bounds.

Application step. Since $\mathbf{A} = \mathbf{X}_L \mathbf{Y}_L \dots \mathbf{Y}_1$, the multiplication $\mathbf{x} \mapsto \mathbf{A}\mathbf{x}$ can be computed by successive multiplication by $\mathbf{x} \mapsto \mathbf{Y}_\ell \mathbf{x}$ for $\ell = 1, \dots, L$, and $\mathbf{x} \mapsto \mathbf{X}_L \mathbf{x}$. Therefore, the time complexity is the sum of the nonzero entries in the factors $(\mathbf{Y}_\ell)_{\ell=1}^L$ and $\mathbf{X}_L$.

- **Number of nonzero entries in $\mathbf{X}_L$**: for each $(R, C) \in P_L^{\text{row}} \times P_1^{\text{col}}$, the basis $\mathbf{U}_{R,C}$ is of size $|R| \times r$, so by (3.54) and (3.57), the number of nonzero entries in $\mathbf{X}_L$ is at most
$$\sum_{R \in P_L^{\text{row}}} \sum_{C \in P_1^{\text{col}}} |R|\, r = |P_1^{\text{col}}| nr = 2nr,$$
since P_L^{row} is a partition of $[\![n]\!]$ by construction, and there are two nodes at the first level of the binary cluster tree T^{col} so $|P_1^{\text{col}}| = 2$.

- **Number of nonzero entries in $\mathbf{Y}_1$**: for each $(R, C) \in P_1^{\text{row}} \times P_L^{\text{row}}$, the matrix $\mathbf{V}_{R,C}$ is of size $|C| \times r$, so by (3.50) and (3.49), the number of nonzero entries in $\mathbf{Y}_1$ is at most
$$\sum_{R \in P_1^{\text{row}}} \sum_{C \in P_L^{\text{col}}} |C|\, r = |P_1^{\text{row}}| nr = 2nr,$$
because P_L^{col} is a partition of $[\![n]\!]$ by construction, and there are two nodes at the first level of the binary cluster tree T^{row} so $|P_1^{\text{row}}| = 2$.

- **Number of nonzero entries in $\mathbf{Y}_\ell$ with $\ell \in [\![2, L]\!]$**: for each $(R, C) \in P_{\ell-1}^{\text{row}} \times P_{L-\ell+1}^{\text{col}}$, for each $i \in \{1, 2\}$, the matrix $\mathbf{V}_{R_i,C}$ is of size $2r \times r$. Moreover, since T^{row} and T^{col} are binary trees, we have $|P_{\ell-1}^{\text{row}}| = 2^{\ell-1}$ and $|P_{L-\ell+1}^{\text{col}}| = 2^{L-\ell+1}$. By (3.56) and (3.54), the number of nonzero entries in $\mathbf{Y}_\ell$ is at most
$$\begin{aligned}
\sum_{R \in P_{\ell-1}^{\text{row}}} \sum_{C \in P_{L-\ell+1}^{\text{col}}} \sum_{i=1}^{2} 2r^2 &= 4r^2 |P_{\ell-1}^{\text{row}}||P_{L-\ell+1}^{\text{col}}| \\
&= 4r^2 (2^{\ell-1})(2^{L-\ell+1}) \\
&= 4r^2 2^L \\
&\leq 4r^2 \frac{n}{\nu},
\end{aligned}$$
by definition of $L := \lfloor \log_2(n/\nu) \rfloor$.

In conclusion, assuming that r, ν are constants, the time complexity for the application step is bounded by

$$2nr + 2nr + \sum_{\ell=2}^{L} 4r^2 \frac{n}{\nu} = \mathcal{O}(n \log n). \tag{3.60}$$

Precomputation step. We now analyze the complexity for building the bases $\mathbf{U}_{R,C}$ for all $(R, C) \in P_\ell^{\text{row}} \times P_{L-\ell+1}^{\text{col}}$ and $\ell \in [\![L]\!]$, with the corresponding row bases $\mathbf{V}_{R,C}$. For that, we assume that the complexity of computing the rank-revealing factorization of a $p \times q$ matrix of rank at most r is $\mathcal{O}(rpq)$, using, e.g., the partial SVD at order r [153].

- At step $\ell = 1$, (3.44) performs a rank-revealing factorization at order r on a matrix of size $|R| \times |C|$ for each $(R, C) \in P_1^{\text{row}} \times P_L^{\text{col}}$, so the total cost is

$$\sum_{R \in P_1^{\text{row}}} \sum_{C \in P_L^{\text{col}}} r\,|R|\,|C| = rn^2, \tag{3.61}$$

  since both P_1^{row} and P_L^{col} are partitions of $[\![n]\!]$.

- At step $\ell \in [\![2, L]\!]$, (3.47) performs a rank-revealing factorization at order r on a matrix of size $|R_i| \times 2r$ for each $(R, C) \in P_{\ell-1}^{\text{row}} \times P_{L-\ell+1}^{\text{col}}$, where R_1, R_2 are the children of R, so the total cost is

$$\sum_{R \in P_{\ell-1}^{\text{row}}} \sum_{C \in P_{L-\ell+1}^{\text{col}}} \sum_{i=1}^{2} r\,|R_i|\,2r = 2r^2 n\,|P_{L-\ell+1}^{\text{col}}|, \tag{3.62}$$

  because $|R_1| + |R_2| = |R|$ and $P_{\ell-1}^{\text{row}}$ forms a partition of $[\![n]\!]$.

Therefore, recalling that $|P_\ell^{\text{col}}| = 2^\ell$ for any $\ell \in [\![L]\!]$ because T^{col} is a binary cluster tree, summing all the costs yields a complexity of the order of

$$\begin{aligned} rn^2 + 2r^2 n \sum_{\ell=2}^{L} |P_{L-\ell+1}^{\text{col}}| &= rn^2 + 2r^2 n \sum_{\ell=2}^{L} 2^{L-\ell+1} \\ &= rn^2 + 2r^2 n(2^L - 1) \\ &\leq rn^2 + 2r^2 n(\frac{n}{\nu} - 1) = \mathcal{O}(n^2), \end{aligned}$$

assuming that r, ν are constants.

3.5.4 Discussion about the initial assumptions

We now discuss the initial assumptions that we made at the beginning of the section for easier presentation of the butterfly algorithm.

Beyond square matrices and binary cluster trees. Obviously the butterfly algorithm can be extended to the case where the target matrix is rectangular, and satisfies the complementary low-rank property for cluster trees that are not binary. For instance, when the kernel operator is defined on a two-dimensional or a three-dimensional space (e.g., the kernel in the Fourier integral operator), the hierarchical partitioning of the space and frequency domain leads to a quadtree or an octree [40, 235, 236].

Noiseless setting. We assumed that the rank of $\mathbf{A}[R, C]$ is at most r for each $(R, C) \in \bigcup_{\ell=1}^{L} T^{\text{row}}(\ell) \times T^{\text{col}}(L-\ell+1)$. This led to an exact factorization in (3.44) and (3.47), meaning that $\mathbf{A}$ admits *exactly* a butterfly factorization (3.59). In the noisy setting where $\mathbf{A}[R, C]$ is well-approximated by a rank-r matrix up to a certain precision ϵ, we can still apply the butterfly algorithm, but the matrix $\mathbf{A}$ is not guaranteed anymore to be equal to the butterfly factorization $\mathbf{X}_L \mathbf{Y}_L \dots \mathbf{Y}_1$ where sparse factors are defined in (3.50), (3.56) and (3.57). As detailed in Chapter 7, the current literature provides little guarantees on the corresponding approximation error, and one of the contribution of this thesis is to bridge this gap.

Adaptive rank. Finally, instead of assuming that the numerical rank up to precision ϵ of $\mathbf{A}[R, C]$ is bounded by the same quantity r for all $(R, C) \in \bigcup_{\ell=1}^{L} T^{\text{row}}(\ell) \times T^{\text{col}}(L-\ell+1)$, we can extend the presented butterfly algorithm to the case where the rank can be different for each submatrices, as mentioned in [289] for instance.

3.5.5 The butterfly factorization mimics the FFT

As originally remarked in [265,266], the fast algorithm obtained with the butterfly factorization (3.59) corresponds to a DAG that resembles the one of the radix-2 FFT given in Figure 3.1, see, e.g., the illustration given in [265, Figure 3]. This can be interpreted by the fact that the sparsity patterns of the sparse factors in (3.59) are somehow similar to those obtained in the factorization of the DFT presented in Section 3.2.2, as we now explain.

Sparsity patterns for general ν and r. Consider that we obtained (3.59) in the case where we choose $n = 2^{L-1}\nu$ for some constant ν, with cluster trees T^{row}, T^{col} of depth L constructed in such a way that

$$\forall \ell \in [\![L]\!], \quad P_\ell^{\text{row}} = P_\ell^{\text{col}} = \left\{ [\![(k-1)n/2^{\ell-1} + 1, kn/2^{\ell-1}]\!] \,|\, k \in [\![2^{\ell-1}]\!] \right\}.$$

In particular, this means that $|P_\ell^{\text{row}}| = |P_\ell^{\text{row}}| = 2^{\ell-1}$ for $\ell \in [\![L]\!]$. Given the definitions (3.50), (3.56) and (3.57), this means that the sparse factors $\mathbf{X}_L, \mathbf{Y}_L, \dots, \mathbf{Y}_1$ satisfy:

$$\begin{aligned} \forall \ell \in [\![2, L]\!], \quad \text{supp}(\mathbf{Y}_\ell) &\subseteq \text{supp}(\mathbf{I}_{|P_{\ell-1}^{\text{row}}|} \otimes \mathbf{1}_{2\times 1} \otimes \mathbf{I}_{|P_{L-\ell+1}^{\text{col}}|} \otimes \mathbf{1}_{r\times 2r}) \\ &= \text{supp}(\mathbf{I}_{2^{\ell-2}} \otimes \mathbf{1}_{2\times 1} \otimes \mathbf{I}_{2^{L-\ell}} \otimes \mathbf{1}_{r\times 2r}), \end{aligned} \tag{3.63}$$

and

$$\mathrm{supp}(\mathbf{Y}_1) \subseteq \mathrm{supp}(\mathbf{I}_{|P_L^{\mathrm{col}}|} \otimes \mathbf{1}_{r \times \frac{n}{|P_L^{\mathrm{col}}|}}) = \mathrm{supp}(\mathbf{I}_{2^{L-1}} \otimes \mathbf{1}_{r \times \nu}), \tag{3.64}$$

$$\mathrm{supp}(\mathbf{X}_L) \subseteq \mathrm{supp}(\mathbf{I}_{|P_L^{\mathrm{row}}|} \otimes \mathbf{1}_{\frac{n}{|P_L^{\mathrm{row}}|} \times r}) = \mathrm{supp}(\mathbf{I}_{2^{L-1}} \otimes \mathbf{1}_{\nu \times r}). \tag{3.65}$$

Sparsity patterns for $\nu = 1$ and $r = 1$. When $\nu = 1$ and $r = 1$, the matrices $\mathbf{Y}_1$ and $\mathbf{X}_L$ are diagonal, and the sparsity patterns of $\{\mathbf{Y}_\ell\}_{\ell=2}^{L}$ becomes:

$$\forall \ell \in [\![2, L]\!], \quad \mathrm{supp}(\mathbf{Y}_\ell) \subseteq \mathrm{supp}(\mathbf{I}_{2^{\ell-2}} \otimes \mathbf{1}_{2\times 1} \otimes \mathbf{I}_{2^{L-\ell}} \otimes \mathbf{1}_{1\times 2}). \tag{3.66}$$

Therefore, one can verify that the factorization $\mathbf{Y}_L\mathbf{Y}_{L-1}\dots\mathbf{Y}_2$ with $L-1 = \log_2(n)$ factors of size $n \times n$ is equivalent to the square dyadic butterfly factorization introduced in Lemma 3.1. Indeed, up to some permutation ambiguities, the factorization $\mathbf{Y}_L\mathbf{Y}_{L-1}\dots\mathbf{Y}_2$ admits $L-1 = \log_2(n)$ sparse factors that satisfy the support constraints in (3.14) in Lemma 3.1. In order words, the DAG associated with the linear operator $\mathbf{x} \mapsto \mathbf{Y}_L\mathbf{Y}_{L-1}\dots\mathbf{Y}_2\mathbf{x}$ is equivalent to the DAG associated with the radix-2 FFT illustrated in Figure 3.1.

3.6 Variations of the butterfly algorithm

Before finishing this chapter, we present some important variations of the butterfly algorithm presented in Section 3.5.

Different factorization orders. There are several variants of the butterfly algorithm that differ by the *hierarchical order* under which the low-rank approximations are constructed for compressing the target matrix. The butterfly algorithm presented in Section 3.5 is the so-called *rowwise* butterfly factorization [40, 250, 288]. It corresponds to the hierarchical order where we start by compressing the matrices $\mathbf{A}[R, C]$ for $(R, C) \in T^{\mathrm{row}}(L) \times T^{\mathrm{col}}(1)$, and we construct the bases for the other levels by traversing T^{row} from bottom to top, and T^{col} from top to bottom. Other hierarchical orders exist, such as the *columnwise* butterfly factorization [250], where we start by compressing $\mathbf{A}[R, C]$ for $(R, C) \in T^{\mathrm{row}}(1) \times T^{\mathrm{col}}(L)$, and we traverse T^{row} from top to bottom, T^{col} from bottom to top in order to construct the bases at other levels. Finally, there exists the *hybrid* approach (or middle-out butterfly factorization), where we start by compressing $\mathbf{A}[R, C]$ for $(R, C) \in T^{\mathrm{row}}(L/2) \times T^{\mathrm{col}}(L/2)$, and traverse the cluster trees from the middle level $L/2$ to both the root level and the leaves level, in order to complete the construction of the other bases [234, 250].

In the next chapters of the thesis (Chapters 6 and 7), we will introduce a notion of *factor-bracketing tree* that describes the various hierarchical order for approaching a target matrix by a butterfly factorization. In particular, we will introduce a factorization algorithm that works for any *factor-bracketing tree*.

Choices for the compression of rank deficient matrices. Earlier butterfly algorithms [265,266,288,289] proposed to use *interpolative decomposition* [57,145,261] to build the low-rank approximations in (3.44) and (3.47). The interpolative decomposition approximates a matrix $\mathbf{M} \in \mathbb{C}^{p\times q}$ by $\mathbf{BP}$ where the columns of $\mathbf{B} \in \mathbb{C}^{p\times r}$ constitute a subset of the columns of $\mathbf{M}$, and the norm of $\mathbf{P} \in \mathbb{C}^{r\times q}$ is not too large. Computing the interpolative decomposition is stable and requires $\mathcal{O}(rpq)$ operations and $\mathcal{O}(pq)$ memory, which leads to the quadratic complexity of the butterfly algorithm analyzed in Section 3.5.3.

Other choices for low-rank approximations are possible, such as the *partial QR decomposition* or the *partial SVD*, which also require $\mathcal{O}(rpq)$ operations [145, 153]. The latter is guaranteed to provide the best low-rank approximation in the Frobenius norm, according to the Eckart–Young–Mirsky theorem.

Last but not least, to achieve further acceleration to compress a rank deficient matrix $\mathbf{M}$, we can use randomized techniques such as *randomized SVD* [153], as proposed in [234]. The main idea is to draw a random Gaussian matrix $\mathbf{W} \in \mathbb{C}^{q\times(r+k)}$ with an oversampling parameter k (e.g., $k = 5$), in order to construct $\mathbf{Y} := \mathbf{MW} \in \mathbb{C}^{p\times(r+k)}$ for which the range approximates the one of $\mathbf{M}$, while having less columns than $\mathbf{M}$ if we assume $r + k < q$. This allows to construct $\mathbf{Q} \in \mathbb{C}^{p\times(r+k)}$ whose columns form an orthonormal basis of the range of $\mathbf{Y}$, in a cheaper way than constructing directly an orthonormal basis of the range of $\mathbf{M}$. Then, we obtain the approximate partial SVD of $\mathbf{M} \approx \mathbf{UDV}^*$ by forming $\mathbf{B} := \mathbf{Q}^*\mathbf{M} \in \mathbb{C}^{(r+k)\times q}$, computing the partial SVD of this smaller matrix: $\mathbf{B} = \tilde{\mathbf{U}}\mathbf{D}\mathbf{V}^*$, and setting $\mathbf{U} := \mathbf{Q}\tilde{\mathbf{U}} \in \mathbb{C}^{p\times r}$. This leads to a complexity of $\mathcal{O}(pq\log(r) + (p+q)r^2)$ in general, with guarantees on the approximation error [153].

Improving the complexity of the butterfly algorithm. In general, without further assumption, the complexity of the butterfly algorithm to approximate a matrix $\mathbf{A}$ of size $n \times n$ satisfying the complementary low-rank property is $\mathcal{O}(n^2)$. However, with some additional assumptions on $\mathbf{A}$, it is possible to approximate $\mathbf{A}$ by a butterfly factorization with a subquadratic complexity.

First, for certain specific kernels, such as the one in Fourier integral operators, we can use some specific analytic techniques [40,233] to provide a faster butterfly algorithm in $\mathcal{O}(n\log n)$. This relies on the fact that the considered kernel, restricted to some domains, can be well approximated by Lagrange interpolation on the Chebyshev grid [40, Theorem 3.3].

Second, for more general kernels, it is possible to use randomized techniques to obtain a butterfly algorithm in $\mathcal{O}(n^{3/2}\log n)$ [234,250]. These randomized techniques can be used if we assume that we can evaluate each entry of the kernel matrix in $\mathcal{O}(1)$ operations, or if we assume that the operator (and its adjoint) can be applied to arbitrary vectors in $\mathcal{O}(n\log n)$ complexity.

3.7 Conclusion

In this chapter we saw that the butterfly algorithm is an analysis-based fast transform that enables approximate rapid evaluation of a linear operator for which the associated matrix satisfies a special low-rank property, called the complementary low-rank property. The fast algorithm is obtained in two steps: the precomputation steps that exploits this low-rank property to approximate the target matrix by a product of sparse factors, and the application step which mimics the FFT. As mentioned during this chapter, we identify some limitations of the previous work concerning their restriction to some specific hierarchical order for building the sparse factors, and their lack of error guarantees for the approximation of the target matrix by a butterfly factorization in the noisy setting. These limitations will be addressed in Chapters 6 and 7.

Chapter 4

Neural network compression via sparsity and matrix factorization

Modern deep learning models require more and more computational resources to train and deploy, due to their increasing size. This calls for better *efficiency* of these models to reduce their computational cost and memory footprint during the training and inference stage. This chapter provides an overview of different techniques for reducing the number of parameters in large redundant models.

4.1 Introduction

In general, deep learning algorithms can be made more efficient either by improving optimization algorithms during the training phase, reducing the volume of training data through data pruning or data compression, or developing computationally more efficient models. Generally speaking, efficiency can be achieved either by:

- Designing smaller model architectures that are well-adapted to a specific dataset, using for instance *neural architecture search* [98];
- *Distilling the knowledge* of a pretrained large model into a smaller one that realizes a similar function when restricted to inputs sampled from a certain data distribution [136];
- Improving the efficiency of certain specific operations, such as reducing the cost of the attention mechanism in transformers [337];
- *Quantizing* the values of the parameters [147, 239];
- Skipping some computations during the forward or the backward pass of a model, e.g., using conditional computation such as in sparse mixture-of-experts [106];

- Reducing the effective number of parameters in the model without changing its architecture.

This chapter will focus on the last type of neural network compression, which revolves around the idea that large models admit some form of *parameter redundancy* that can be removed without significant loss of performance on some learning task, either after training or during training. For instance, when some parameters are not important with respect to a certain criteria, they can be *pruned* out, i.e., set to zero, in order to skip their associated operations during the forward or the backward pass of the model. This process is known as *sparsification*. Alternatively, parameter redundancy can be removed by *decomposition* of certain weight matrices or weight tensors, using for instance low-rank decomposition or sparse matrix factorization such as butterfly factorization.

Content of the chapter. Section 4.2 starts by summarizing the potential benefits of removing redundant parameters in the model. Then, we present different techniques to remove redundant parameters, either via sparsification (Section 4.3), low-rank decomposition (Section 4.4) or butterfly factorization (Section 4.5).

4.2 Benefits of reducing the number of parameters

We expect the following benefits when compressing a neural network by reducing its number of parameters.

Reduced computational cost and memory footprint. When sparse operations are properly implemented for a given hardware, reducing the number of parameters in a model can potentially lead to time acceleration of its forward and backward pass, and it can also yield a smaller memory footprint during computation [172].

Reduced communication time. Federated learning is a paradigm where a shared model is trained on decentralized data, so that the privacy of local data on different devices is preserved during training. However, updating the parameters of the shared model requires communication between the devices, which can become costly when the number of parameters is large. Communication time can be reduced by promoting sparsity in the model because of its reduced storage [26, 46, 286]. It is also expected to see a reduction of communication time in standard multi-GPU training with data parallelism.

Improved generalization. While the reduction of the number of parameters is mainly motivated today by the need for efficiency in deep learning models, its original introduction aimed at reducing the model's capacity in order to prevent

overfitting [220]. However, as discussed in Chapter 1, overparameterized neural networks tend to generalize well in practice, which suggests that reducing the number of parameters in the model does not necessarily lead to better generalization. Yet, recent methods show that sparsifying a model via iterative magnitude pruning (see Section 4.3) can sometimes lead to *better* test accuracy compared to the dense model [109]. Some works propose to explain such an improved generalization not by its number of parameters, but rather by interpreting the pruning mechanism as a form of noise-injection regularization [17]. Nonetheless, understanding the generalization properties of deep neural networks remains an area of ongoing research.

Improved out-of-distribution detection. Given a model trained on a certain dataset, the task of out-of-distribution detection is to classify whether or not a given input data is sampled from the data distribution of this training set [367]. Such a detection can be improved by pruning certain weights in the model, as shown in [331]

Improved adversarial training. In order to avoid vulnerability to adversarial attacks, deep neural networks can be trained in an adversarial manner, using for instance min-max robust optimization. However, such an adversarial training might incur severe robust generalization gap, and it might require more data to train [312, 317]. One way to address these issues in adversarial training is to promote sparsity in the model, as proposed in [54].

Improved robustness to privacy attacks. A trained model presents some privacy risk if, assuming only a black-box access to the model, it is possible to predict whether a given input data belongs to the training set of the model [176]. Some recent works studied whether or not promoting sparsity in the model can increase the robustness of the model to such membership inference attacks [133, 373].

Improved data efficiency. Recent works show that promoting sparsity in the model via iterative magnitude pruning yields better data-efficiency, in the sense that they perform better than the dense model in low-data regimes [53, 344].

Improved transferability. In [183], it is shown that pretraining a sparse vision model on ImageNet-1k [78] in a supervised manner can sometimes yield better downstream performance than the dense model.

4.3 Model compression via sparsification

One major paradigm to remove parameter redundancy in a model is to promote sparsity via pruning. The goal is to set a certain number of parameters to zero,

while avoiding as much as possible a drop of performance, e.g., measured as the model's accuracy on a test set. This can be formulated as the minimization of the empirical risk under some sparsity constraints[1]: for instance, in the context of supervised learning, for a given labeled dataset $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n \subseteq \mathcal{X} \times \mathcal{Y}$, a deep network $f_\theta : \mathcal{X} \to \mathcal{Y}$ parameterized by $\theta \in \mathbb{R}^d$, a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ and a given budget $p \in \mathbb{N}$, the general problem of training a sparse neural network can be defined as:

$$\inf_{\theta \in \mathbb{R}} \quad \frac{1}{n} \sum_{i=1}^{n} \ell(f_\theta(\mathbf{x}_i), \mathbf{y}_i) \quad \text{such that} \quad \|\theta\|_0 \leq d. \tag{4.1}$$

Solving this problem requires to select an appropriate *support* for θ (also called a *mask* in the literature), i.e., the set of indices corresponding to nonzero parameters, and to optimize the parameters of the selected support, with respect to the training objective function.

Promoting sparsity in deep neural networks is difficult. While sparse linear recovery is quite well studied [108], promoting sparsity into deep or multilinear models is considerably more challenging. Identifying an appropriate support among all the possible choices is a difficult combinatorial problem, and even when such a support is known and fixed, optimizing the parameters of the selected support can be challenging in general. For instance, the existence of a minimizer in the optimization problem of training a deep ReLU network with sparsity constraints is not always guaranteed in general [211]. More fundamentally, deep matrix factorization with sparsity constraints, which is equivalent to training a sparse deep linear network[2], is difficult in general, even when the supports of the sparse factors are known and fixed in advance [212].

Heuristic algorithms for sparse neural networks. Despite the lack of theoretical understanding, there exist many heuristic algorithms for training sparse neural networks [172] that perform well in practice. Experimentally, promoting sparsity in a large model performs better than the baseline where we train a smaller dense network with the same number of parameters [102, 238, 245, 397]. In some situations, it can even match the performance of the original large dense model [109, 110]. For the rest of this section, we review some of these algorithms, which can be categorized into several families, depending on their training schedule to promote sparsity, and on their pruning criterion to select an appropriate support. We also discuss the importance of different forms of *structured* sparsity, as well as the importance of the evaluation benchmark for comparing methods.

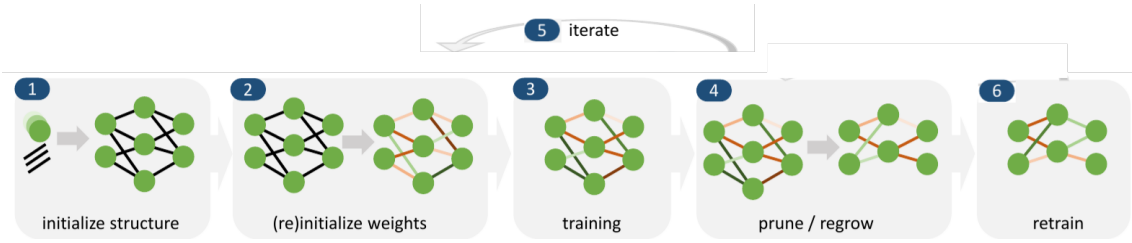


Figure 4.1: Overview of sparsification schedules from [172]. The graph represents the neural network model, where the values of its parameters are indicated by different colors of the edges. Pruning a parameter corresponds to removing the associated edge in the graph.

4.3.1 Sparse deep learning schedules

The general training schedule for promoting sparsity in a neural network includes the following steps summarized in Figure 4.1:

1. Choose an initial support before training, either independently from the data, such as using a random mask [245], or according to a certain criterion that may depend on the training dataset [222, 352].

2. Initialize the parameters for the selected support in a certain way, for instance at random or using some pretrained weights.

3. Optimize the parameters on the selected support for a certain number of iterations. The pruned parameters (i.e., parameters not in the selected support) are not optimized during this phase and their values remain at zero.

4. Remove or add indices in the support, according to a certain criterion. This step is referred to as the pruning or the regrowth step. Pruning promotes sparsity in the model, while regrowth enables some flexibility in the training where previously pruned parameters can be reconsidered if they become relevant according to the predefined criterion [268].

5. Optionally, repeat steps 2, 3, 4 for a certain number of times. The parameters in the new support are either initialized from their values of the previous support after pruning or regrowth, or reset to their values at a certain time of the training, according to the so-called *rewinding* procedure [109, 110]. For instance, they can either be reset to their initial values at the first initialization before training [109], or at their values obtained after a certain number of training iterations since the beginning of the training [109].

6. Retrain the parameters in the final support until convergence. Again, the initial values of the parameters in this step can be either the ones after the

[1]Note that there is an alternative formulation discussed in Section 4.3.3.

[2]More precisely, it is equivalent to training a sparse multilayer perceptron without bias and the activation function being the identity function.

previous pruning step or after rewinding. This final retraining step is usually necessary to improve the model's accuracy [155].

Then, different methods for training sparse neural networks differ on their specific training scheduling.

Dense-to-sparse schedules. The *train-then-sparsify* schedule (also called one-shot pruning) [224] starts with a dense network at step 1, train the dense network during a certain number of iterations at step 3, prune out some parameters at step 4, and does not repeat step 2, 3, 4. This schedule can be effective if the primary goal is to reduce inference cost, but it comes at the cost of requiring a dense training phase which can be costly when the model is large. As detailed in Section 4.3.3 below, it is eventually possible to design a training loss for promoting sparsity (but with no guarantees of success) during the dense training step. The *iterative-sparsification* schedule [109, 155, 271] is the same as the previous schedule, but it repeats steps 2, 3, 4 several times, and step 4 always performs pruning. This promotes sparsity gradually, by pruning out only a small fraction of parameters at each repetition of step 4. Choosing the right amount of parameters to prune each time and the right number of repetitions of step 4 is critical for methods using this schedule [189].

Sparse-to-sparse schedules. In *sparse-to-sparse dynamic schedules* [20, 102, 186, 268, 273], the model is initialized with a sparse structure at step 1, and both pruning and regrowth are allowed in step 4. Compared to the two previous schedules, it does not require to train a dense network which can be prohibitive when computational resources are limited [268]. Finally, the *sparse-to-sparse static schedule* [222, 245, 335, 352] starts with an initial sparse structure in step 1, and never updates the support, i.e., it skips step 4. Since the support is fixed during the whole training, it requires in general a good choice of the prescribed support at initialization.

4.3.2 Criteria for selecting the support

We now discuss the different ways to select a support, either at initialization (step 2) or during pruning (step 4).

Magnitude pruning. Magnitude pruning [116, 155, 186, 224] is a data-free pruning criterion where the selected support given the current parameter $\theta \in \mathbb{R}^d$ of the neural network is defined as

$$\mathbf{s}(\theta) := \{i \in [\![d]\!],\ |\theta[i]| \leq C\}, \tag{4.2}$$

where $C > 0$ is a certain threshold parameter. Note that this pruning criterion is not invariant to some rescaling symmetries [280] of the parameterization of the

neural network[3]. In other words, depending on the chosen rescaling-equivalent parameterization, magnitude pruning can lead to potentially different selected support. Nevertheless, this pruning criterion is quite effective in practice. For instance, in the empirical demonstration of the so-called *lottery ticket hypothesis* [109], using magnitude pruning in the iterative sparsification schedule combined with rewinding, which is the procedure called *iterative magnitude pruning*, can provide a sparse neural network able to match the performance of the dense network[4].

Criteria based on Taylor expansion of the objective function. In order to incorporate some information about the input data, we can design pruning criteria based on the Taylor expansion of the objective function. Consider that the parameters $\theta \in \mathbb{R}^d$ of a neural network f_θ are trained to minimize $L : \mathbb{R}^d \to \mathbb{R}$. Pruning some weights by setting $\theta \leftarrow \theta \odot (\mathbf{1}_d - \mathbf{s})$ with some $\mathbf{s} \in \{0,1\}^d$ is then equivalent to adding the perturbation $\delta\theta := -\theta \odot \mathbf{s}$ to θ. By the Taylor expansion of L around θ, we have:

$$L(\theta + \delta\theta) - L(\theta) \approx \nabla L(\theta)^\top \delta\theta + \frac{1}{2}\delta\theta^\top \mathbf{H}(\theta)\delta\theta := \delta L, \tag{4.3}$$

where $\nabla L(\theta)$ and $\mathbf{H}(\theta)$ are the gradient and the Hessian matrix of L at θ. The importance (or the saliency) of a parameter $\theta[i]$ for $i \in [\![d]\!]$ can then be measured as the approximate variation $|\delta L|$ of the objective function, when $\mathbf{s}$ is the i-th element of the standard basis[5] of $\mathbb{R}^d$. Criteria based only on first-order information, i.e., the gradient term in (4.3), can be useful for pruning before training or during training, but not after training to convergence where the gradient of the objective function is supposed to be close to zero [222, 270, 316]. After convergence, one might instead consider criteria based on second-order information [86, 159, 220, 324], i.e., the term with the Hessian matrix in (4.3). In this case, since the computational cost of the Hessian matrix is quadratic with respect to the number of parameters, it is necessary to approximate it when the number of parameters is very large, for more efficient numerical methods.

Pruning independent from data and parameters. Support selection without any information about the parameters and the input data is an important baseline

[3]To illustrate this, consider a ReLU network with one hidden layer, with N_0 input neurons, N_1 neurons in the hidden layer, and N_2 output neurons. Then, the family of functions realized by this network is the set of functions $f_\theta : \mathbb{R}^{N_0} \to \mathbb{R}^{N_2}$ defined by $\mathbf{x} \mapsto \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$, parameterized by $\theta = (\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1) \in \mathbb{R}^{N_1 \times N_0} \times \mathbb{R}^{N_2 \times N_1} \times \mathbb{R}^{N_1}$. By positive homogeneity of the ReLU function, for a given parameter $\theta = (\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1)$, we have $f_\theta = f_{\theta'}$ for any θ' of the form $(\lambda\mathbf{W}_1, \frac{1}{\lambda}\mathbf{W}_2, \lambda\mathbf{b}_1)$ with $\lambda > 0$. Therefore, there exists $\lambda > 0$ such that $\mathbf{s}(\theta) \neq \mathbf{s}(\theta')$, due to this rescaling symmetry.

[4]More precisely, the lottery ticket hypothesis states that "a randomly-initialized, dense neural network contains a subnetwork that is initialized such that, when trained in isolation, it can match the test accuracy of the original network after training for at most the same number of iterations" [109].

[5]The standard basis of $\mathbb{R}^d$ is the set of vectors, whose components are all zero, except one that equals 1.

to consider in order to evaluate the effectiveness of the above criterion. The most basic support selection is by *uniform random pruning* [116,245,397], where the support is selected uniformly among all the possible supports satisfying a desired global or layerwise sparsity ratio. More elaborated random pruning methods can take into account the topology of the graph representing the neural network. For instance, random Erdős-Rényi pruning can be applied to multilayer perceptrons [268] or convolutional neural networks [102]. In such random topologies, larger layers are allocated with higher sparsity than smaller layers.

4.3.3 Training with a sparsity-promoting loss

An alternative problem formulation to (4.1) is to consider a regularization loss $\mathcal{R} : \mathbb{R}^d \to \mathbb{R}$, in the hope that its minimization promotes sparsity in the network. The problem is then formulated as:

$$\inf_{\theta} \quad \frac{1}{n} \sum_{i=1}^{n} \ell(f_\theta(\mathbf{x}_i), \mathbf{y}_i) + \lambda \mathcal{R}(\theta), \tag{4.4}$$

where λ is some hyperparameter controlling the tradeoff between the data-fitting term and the regularization term. A natural regularization loss is $\mathcal{R}(\theta) = \|\theta\|_0$, but this would lead to a difficult optimization problem due to its discrete nature[6]. The regularization by the ℓ_0-norm can be relaxed either by considering some differentiable gating variables [254], or by adopting its tightest convex relaxation using the ℓ_1-norm [207, 243]. However, as opposed to sparse linear recovery where basis pursuit has some guarantees of success [108], the regularization by ℓ_1-norm is not guaranteed to promote sparsity in deep neural networks. In fact, even ℓ_1-regularization in bilinear models such as in blind deconvolution (which is equivalent to training a linear convolutional neural network with two layers) can be ill-posed, in the sense that the minimizer of the regularized problem is a trivial solution [24].

4.3.4 From unstructured to structured sparsity

We now discuss the different forms of sparsity patterns that can be obtained from the previous algorithm, and their importance when time acceleration of sparse neural networks matters.

Unstructured pruning. The most general form of sparsity is *unstructured*, in the sense that the support of the sparse parameter $\theta \in \mathbb{R}^d$ does not satisfy any additional constraint, apart from the one that it should achieve a certain desired sparsity ratio. This usually reduces the number of operations required to perform a forward or backward pass of the neural network. For instance, for a given sparse linear layer parameterized by a weight matrix $\mathbf{W} \in \mathbb{R}^{M \times N}$, the number of

[6] In particular, sparse linear recovery is shown to be NP-hard in general [108].

multiplication operations required to compute the output $\mathbf{W}\mathbf{x}$ for a given $\mathbf{x} \in \mathbb{R}^N$ is $\|\mathbf{W}\|_0$ instead of MN. Yet, without further assumption about the structure of $\text{supp}(\mathbf{W})$, implementations for sparse matrix multiplication on certain types of hardware might not be able to achieve a time acceleration compared to dense implementations. This is typically the case for GPUs implementations, as illustrated in Chapter 9.

Structured pruning. Introducing a form of regularity in the sparsity pattern can therefore potentially lead to faster implementations of the corresponding sparse matrix multiplication, depending on the considered hardware. In general, structured pruning includes:

- **Neuron pruning**: for a linear layer, this corresponds to setting a whole row or a whole column of entries to zero in the weight matrix [188, 206].
- **Filters / channels pruning**: for a convolutional layer, this corresponds to pruning out a whole filter associated with one input channel and one output channel [164, 224].
- **Block-sparse weights**: a matrix is block sparse if it is a block matrix with equal blocksize and with few nonzero blocks. There exist efficient GPU kernels that can leverage this structure for sparse matrix multiplication [139].
- **N:M sparsity structure**: a weight matrix or a weight tensor satisfies the N:M structure if only N entries are nonzero for every M contiguous entries in memory [112, 178, 393]. For instance, the 4:2 structure can be leveraged by modern GPUs such as the NVIDIA A100 GPUs for faster sparse matrix multiplication.

To promote structured sparsity, it is necessary to adapt the pruning criteria or the regularization losses presented above that were originally designed to promote unstructured sparsity. In general, the performance of neural networks compressed by structured sparsity is usually slightly worse compared to that obtained with unstructured sparsity, due to the additional constraints on the support [172].

4.3.5 On the importance of rigorous experiments

Many algorithms have been proposed for promoting sparsity in neural networks, and experiments indeed show that they can reach a similar performance to dense networks, on some specific experimental settings [172]. However, as we now discuss, it is not always clear which sparsity methods provide the best performance-efficiency tradeoff, due to a lack of fair comparison between them. Moreover, recent works have identified more challenging evaluation benchmarks where the majority of these sparsity methods do not reach the performance levels of dense networks.

Fairness of comparison. It is reported [28, 353] that the current literature lacks a rigorous and solid numerical comparison between different sparsity methods, in a fair and controlled experimental setting. For example, hyperparameters are not always properly tuned when comparing different methods, particularly the learning rate scheduling in the retraining phase (step 6 in Section 4.3.1). Yet, this tuning is crucial for the final performance of the sparse model, as shown in [209].

Importance of comparison to baselines. In particular, proper comparison to baseline methods is important to evaluate the real effectiveness of the proposed methods. For instance, when using a specific weight initialization such as rewinding, it is necessary to compare it with random initialization, which is not always properly done as pointed out by [68, 252]. Similarly, when the proposed method elaborates a specific criteria to select a support, it is necessary to compare it with random pruning at initialization, which turns out to be a strong baseline, as recently demonstrated in [245]. In particular, it is shown in [245] that as "the original dense networks grow wider and deeper, the performance of training a randomly pruned sparse network will quickly grow to match that of its dense equivalent, even at high sparsity ratios".

Issues with current benchmarks. The current literature is currently lacking an evaluation of sparsity methods on more diverse and challenging tasks than traditional ones. Indeed, most of the research papers (79 out of 100 according to [246]) only evaluated their methods on a single or a few tasks, mostly solely on image classification (MNIST [219], CIFAR-10/100 [202], ImageNet [78]) and sometimes on NLP tasks (GLUE [351]). In [246], a new benchmark is proposed with more difficult tasks such as commonsense reasoning, arithmetic reasoning, protein thermostability prediction and multilingual translation. It turns out that many state-of-the-art sparsity methods fails to match the performance of dense networks in this new benchmark, even in the case where only 5% of the model's parameters are pruned out.

4.4 Model compression via low-rank decomposition

Another main paradigm to remove parameter redundancy in a model is to use low-rank matrices or tensors to parameterize the weights of certain layers in the model. Recall that a matrix $\mathbf{W} \in \mathbb{R}^{M \times N}$ admits a low-rank decomposition with rank $R \leq \min(M, N)$ if it can be written as the product $\mathbf{UV}$ for two matrices $(\mathbf{U}, \mathbf{V}) \in \mathbb{R}^{M \times R} \times \mathbb{R}^{R \times N}$. This means that the number of operations required to compute the matrix-vector multiplication is $R(M + N)$ instead of MN. As detailed below, there exist several ways to extend the low-rank decomposition to tensors of higher orders. Overall, the whole pipeline for model compression by low-rank decomposition can be decomposed in the following steps:

1. We first train a dense network, and apply a decomposition algorithm to approximate some of the obtained weight matrices or tensors with a low-rank decomposition, under some rank constraints that control a tradeoff between the compression rate and the approximation error.

2. The obtained decomposition yields an initialization for the compressed network where the considered weight matrices and tensors are parameterized as low-rank ones. The parameters in the decomposition are then optimized in a final retraining step. For instance, when a weight matrix $\mathbf{W}$ is replaced with $\mathbf{UV}$ in the objective function, one trains the compressed network by directly optimizing the parameters $\mathbf{U}$, $\mathbf{V}$ using derivatives of the objective function with respect to $\mathbf{U}$, $\mathbf{V}$, respectively. This retraining step is usually important to recover from the drop of performance due to approximation errors in the decomposition step.

This *train-then-decompose* schedule [80, 185, 196, 218, 333] is analogous to the train-then-sparsify schedule in Section 4.3.1. An alternative is to consider the *train-from-scratch* schedule, analogous to the sparse-to-sparse static schedule in Section 4.3.1, such as in [181, 333], where one ignores the dense training step, randomly initializes the parameters in the compressed network, and optimizes them by gradient descent. In this case, the ranks in the low-rank parameterization are assumed to be known and fixed before training. Note that an appropriate initialization is crucial when training from scratch [181, 195].

4.4.1 Low-rank decomposition in convolutional layers

Neural network compression via low-rank decomposition was initially studied for convolutional neural networks [79], where most of the operations are performed by convolutional layers. The original motivation for such a compression is that the values of the output feature by a convolutional layer tend to be redundant, due to the smoothness of natural images where the value of each pixel is likely to be similar to a weighted average of its neighbors. Several works showed that it is actually possible to compress convolutional neural networks via tensor decomposition, with a small drop of performance compared to the dense network, while enabling time acceleration and reduced energy consumption at inference, e.g., on mobile devices [196].

We give an overview of the different methods by considering the compression of a 2D-convolutional layer with C input channels, N output channels, filters of spatial size $D \times D$. For simplicity, we assume no zero padding and a stride equal to one. This layer is parameterized by a 4D-tensor $\mathbf{K} \in \mathbb{R}^{N \times C \times D \times D}$, and it maps an input tensor $\mathbf{U} \in \mathbb{R}^{C \times X \times Y}$ to an output tensor $\mathbf{V} := \mathbf{K} * \mathbf{X} \in \mathbb{R}^{N \times (X-D+1) \times (Y-D+1)}$ defined as:

$$\forall (n, x, y), \quad \mathbf{V}[n, x, y] = \sum_{c=1}^{C} \sum_{i=1}^{D} \sum_{j=1}^{D} \mathbf{K}[n, c, i, j] \, \mathbf{U}[c, x-1+i, y-1+j]. \tag{4.5}$$

CP-decomposition. The kernel weight tensor can be compressed using the CP-decomposition, based on the idea of separation of variables [199]. We say that $\mathbf{K}$ admits such a decomposition with rank R if

$$\forall (n,c,i,j), \quad \mathbf{K}[n,c,i,j] = \sum_{r=1}^{R} \mathbf{K}_1[n,r]\mathbf{K}_2[c,r]\mathbf{K}_3[i,r]\mathbf{K}_4[j,r], \tag{4.6}$$

for some matrices $\mathbf{K}_1 \in \mathbb{R}^{N\times R}$, $\mathbf{K}_2 \in \mathbb{R}^{C\times R}$, and $\mathbf{K}_3, \mathbf{K}_4 \in \mathbb{R}^{D\times R}$. With such a decomposition, the original convolutional layer is then equivalent to applying a sequence of four one-dimensional convolutions with smaller kernels, where each of them operates on separate variables. This model is typically used in [218] for compressing convolutional layers. Instead of separating all the variables, other works consider a less restrictive decomposition where only the spatial dimensions [185, 333] or only the channel dimensions [80] are separated. Since there is no algorithm for computing the best approximation of a tensor by a CP-decomposition [199] in general, we can only use heuristics to compute an approximate solution, such as linear least square as in [218] or the strategy of greedily finding a best rank-one approximation of the residual [80]. Note however that, when separating only the spatial variables, the optimal decomposition can be computed using singular value decomposition, as remarked in [333].

Tucker decomposition. An alternative tensor decomposition model is the Tucker decomposition [199] with rank (R_1, R_2, R_3, R_4), which takes the form:

$$\mathbf{K}[n,c,i,j] = \sum_{r_1=1}^{R_1}\sum_{r_2=1}^{R_2}\sum_{r_3=1}^{R_3}\sum_{r_4=1}^{R_4} \mathbf{C}[r_1,r_2,r_3,r_4]\,\mathbf{U}_1[n,r_1]\,\mathbf{U}_2[c,r_2]\,\mathbf{U}_3[i,r_3]\mathbf{U}_4[j,r_4], \tag{4.7}$$

for some $\mathbf{C} \in \mathbb{R}^{R_1\times R_2\times R_3\times R_4}$, and $\mathbf{U}_1 \in \mathbb{R}^{N\times R_1}$, $\mathbf{U}_2 \in \mathbb{R}^{C\times R_2}$, $\mathbf{U}_3 \in \mathbb{R}^{D\times R_3}$, $\mathbf{U}_4 \in \mathbb{R}^{D\times R_4}$. In [196], kernel weights in convolutional layers are approximated by a less restrictive decomposition that takes the form:

$$\mathbf{K}[n,c,i,j] = \sum_{r_1=1}^{R_1}\sum_{r_2=1}^{R_2} \tilde{\mathbf{C}}[r_1,r_2,i,j]\,\tilde{\mathbf{U}}_1[n,r_1]\tilde{\mathbf{U}}_2[c,r_2], \tag{4.8}$$

for some $\tilde{\mathbf{C}} \in \mathbb{R}^{R_1\times R_2\times D\times D}$, $\tilde{\mathbf{U}}_1 \in \mathbb{R}^{N\times R_1}$ and $\tilde{\mathbf{U}}_2 \in \mathbb{R}^{C\times R_2}$. With such a decomposition, computing the output of the convolutional layer is then equivalent to apply sequentially a 1×1 convolution, a $D\times D$ convolution and a 1×1 convolution [196].

Regularization loss. In the train-then-decompose schedule, it is possible to add a regularization loss based on the nuclear norm during the dense training phase, as in [8, 180, 365]. The role of such a regularization is to promote a certain low-rank structure during the dense training, so that we can expect a small error when approximating the obtained weight tensors after training by low-rank ones.

Variations of the decomposition model. When the kernel weight tensor admits a low-rank tensor decomposition, the original convolutional layer can be replaced by a sequence of several convolutional layers. The decomposition model in [383] adds a non-linear activation function, such as the ReLU activation, in-between each of these convolutional layers, in order to obtain a non-linear approximation of the convolutional layer. An alternative decomposition model is to consider the sparse low-rank model [372], where the weight tensor is approximated by the sum of a low-rank tensor and a sparse tensor.

4.4.2 Low-rank decomposition in fully-connected layers

In general, as opposed to weight tensors of convolutional layers obtained after training, the weight matrices of fully-connected layers in neural networks do not always admit a low-rank structure. Yet, in some cases, it is still possible to use the idea of low-rank decomposition to compress fully-connected layers, as we now discuss.

Data-aware low-rank decomposition. In [49,371], it is observed that the weight matrices in pretrained transformers (such as BERT [83] in language or DeiT [340] in vision) do not admit a low-rank structure, in the sense that their singular values do not decay very fast. However, it is observed experimentally that the intermediate features computed after each linear layer of the transformer[7] lie in a low-dimensional subspace. Therefore, given a pretrained weight matrix $\mathbf{W} \in \mathbb{R}^{M\times N}$ and a given batch of well-chosen calibration data $\mathbf{X} \in \mathbb{R}^{N\times K}$ of batch size K, the data-aware low-rank decomposition problem with rank R proposed in [49] is

$$\min_{\mathbf{M}\in\mathbb{R}^{N\times N}} \|\mathbf{W}\mathbf{X} - \mathbf{W}\mathbf{M}\mathbf{X}\|_F, \quad \text{such that} \quad \operatorname{rank}(\mathbf{M}) \leq R. \tag{4.9}$$

This problem can be solved using SVD [49, Theorem 1]. After decomposition, we obtain $\mathbf{M} = \mathbf{U}\mathbf{V}$ for some $\mathbf{U} \in \mathbb{R}^{N\times R}$ and $\mathbf{V} \in \mathbb{R}^{R\times N}$, and we can replace $\mathbf{W}$ by $\tilde{\mathbf{U}}\mathbf{V}$ where $\tilde{\mathbf{U}} = \mathbf{W}\mathbf{U} \in \mathbb{R}^{M\times R}$, to initialize the compressed fully-connected layer.

Variants of decomposition model. The low-rank decomposition model for approximating weight matrices in transformers can be made more flexible using other variants such as the sparse low-rank model [237] and the weighted low-rank factorization [174] to achieve more efficient decomposition.

Low-rank adaptation for transfer learning. In the context of transfer learning, when adapting or finetuning a certain dense pretrained backbone on a given downstream task, the weight matrix $\mathbf{W} \in \mathbb{R}^{M\times N}$ for a given fully-connected layer can be parameterized as

$$\mathbf{W} = \mathbf{W}_0 + \mathbf{U}\mathbf{V}, \tag{4.10}$$

[7] more precisely, either after the key, query, value linear projections in the multihead attention module, or at the hidden layer or the output layer of the feed-forward network module

where $\mathbf{W}_0 \in \mathbb{R}^{M\times N}$ is the weight matrix obtained after pretraining that is kept frozen, and $\mathbf{U} \in \mathbb{R}^{M\times R}$, $\mathbf{V} \in \mathbb{R}^{R\times N}$ are the learnable parameters during transfer. In such a low-rank adaptation [175], the matrix $\mathbf{W}$ is not low-rank, so the computation of the forward pass for the corresponding linear layer requires MN operations, but since gradients are computed with respect to $\mathbf{U}$, $\mathbf{V}$ and not to $\mathbf{W}_0$, the backward pass associated with this linear requires only $R(M+N)$ operations. This is a form of parameter-efficient finetuning, especially useful and popular for large language models.

4.5 Model compression via butterfly factorization

Butterfly factorization was initially introduced in the context of numerical analysis to construct fast algorithms for the rapid evaluation of scattering operators [265], special function transforms [289] or Fourier integral operators [40], as discussed in Chapter 3. Some recent works [5, 71, 72, 74, 75, 241, 342] proposed to use an alternative definition of butterfly factorization for neural network compression. As introduced in Chapter 1, butterfly factorization in the context of deep learning refers to a sparse matrix factorization $\mathbf{X}_1 \dots \mathbf{X}_L$ with $L \geq 2$ factors, satisfying some specific *fixed-support constraint*. These supports have a specific structure that can be potentially used for efficient implementations of the corresponding sparse matrix multiplication (see Chapters 9 and 10 for more details). By analogy with the model compression via low-rank matrices, a neural network compressed via butterfly factorization can either be trained with a *train-then-decompose* schedule, or with a *train-from-scratch* schedule. We now detail the different variants of butterfly factorization used for deep neural networks.

4.5.1 Square dyadic butterfly factorization

Recall that the square dyadic butterfly factorization is defined as a factorization $\mathbf{X}_1 \dots \mathbf{X}_L$ with $L \geq 2$ factors $\mathbf{X}_1, \dots, \mathbf{X}_L$ of size $n \times n$ (with $n := 2^L$) satisfying the sparsity constraint (1.1). In [74, 75], the motivation for this factorization in deep learning applications is to avoid manually designing what structured linear maps (like the DFT, the Hadamard matrix, etc.) to use in a given neural network architecture for a given learning task, which was previously the case in [58, 84, 269, 384]. Instead, it is suggested in [74, 75] to use a generic representation for structured matrices based on the square dyadic butterfly factorization, that is not only expressive, but also differentiable so that the parameters can be updated via gradient descent.

Kaleidoscope representation. Let us illustrate the expressivity of the square dyadic butterfly structure using the so-called kaleidoscope matrix representation [75]. Define $\mathcal{B}$ as the class of matrices that admit an exact square dyadic butterfly factorization; $\mathcal{B}\mathcal{B}^*$ as the class of matrices of the form $\mathbf{M}_1\mathbf{M}_2{}^*$, with

$\mathbf{M}_1, \mathbf{M}_2 \in \mathcal{B}$, and where $*$ denotes the conjugate transpose; $(\mathcal{B}\mathcal{B}^*)^W$ as the class of matrices of the form $\mathbf{M}_1 \dots \mathbf{M}_W$ with $\mathbf{M}_w \in \mathcal{B}\mathcal{B}^*$ for $w = 1, \dots, W$. Then, the following structured linear maps have an associated matrix in $(\mathcal{B}\mathcal{B}^*)^W$ with a small W, resulting in a representation of the linear transform with nearly-optimal memory and time complexity [75].

- **Classical fast linear transforms**: this includes the DFT, the discrete cosine transform, the discrete sine transform and the Hadamard transform. For instance, the Hadamard matrix $\mathbf{H}$ of dimension a power of two is in $\mathcal{B}$, and the DFT matrix $\mathbf{F}$ is in $(\mathcal{B}\mathcal{B}^*)^2$, because it can be written as $\mathbf{F} = \mathbf{B}\mathbf{P}$ where $\mathbf{B} \in \mathcal{B} \subset \mathcal{B}\mathcal{B}^*$ and $\mathbf{P}$ is the bit-reversal matrix, which is shown to be also in $\mathcal{B}\mathcal{B}^*$.

- **Circulant matrices**, which are associated with convolutions: any circulant matrix $\mathbf{C}$ can be expressed as $\mathbf{C} = \mathbf{F}^{-1}\mathbf{D}\mathbf{F}$ where $\mathbf{D}$ is a diagonal matrix by [294, Theorem 2.6.4], and $\mathbf{C} \in (\mathcal{B}\mathcal{B}^*)^4$, since the DFT matrix $\mathbf{F}$ as well as its inverse $\mathbf{F}^{-1} = \mathbf{F}^*$, and their scaled versions $\mathbf{D}\mathbf{F}$, $\mathbf{F}^{-1}\mathbf{D}$ are all in $(\mathcal{B}\mathcal{B}^*)^2$. In fact a tighter analysis can show that any circulant matrix $\mathbf{C}$ is in $\mathcal{B}\mathcal{B}^*$ [75, Lemma J.5].

- **Toeplitz matrices**: for any Toeplitz matrix $\mathbf{T}$ of size $N \times N$, there exists a circulant matrix of size $2N \times 2N$ such that $\mathbf{T} = \mathbf{R}\mathbf{C}\mathbf{R}^*$ with $\mathbf{C}$ a circulant matrix of size $2N \times 2N$ and $\mathbf{R} := \begin{bmatrix} \mathbf{I}_N & \mathbf{0}_N \end{bmatrix} \in \mathbb{R}^{N \times 2N}$ a reduced identity matrix. Moreover, any matrix $\mathbf{M}$ of size $N \times N$ can be expressed as a product $\mathbf{M} := \mathbf{T}_1\mathbf{T}_2 \dots \mathbf{T}_{2N+5}$ of (at most) $2N + 5$ Toeplitz matrices [368], hence can be written as $\mathbf{R}\mathbf{N}\mathbf{R}^*$ with $\mathbf{N} \in (\mathcal{B}\mathcal{B}^*)^{2N+5}$. Indeed, writing $\mathbf{T}_i = \mathbf{R}\mathbf{C}_i\mathbf{R}^*$ with $\mathbf{C}_i \in \mathcal{B}\mathcal{B}^*$ a circulant matrix of size $2N \times 2N$, we have $\mathbf{M} := \mathbf{R}\mathbf{N}\mathbf{R}^*$ with $\mathbf{N} := \mathbf{C}_1(\mathbf{R}^*\mathbf{R})\mathbf{C}_2 \dots (\mathbf{R}^*\mathbf{R})\mathbf{C}_{2N+5} \in (\mathcal{B}\mathcal{B}^*)^{2N+5}$, as $(\mathbf{R}^*\mathbf{R})\mathbf{C}_i \in \mathcal{B}\mathcal{B}^*$ for each i due to the fact that $(\mathbf{R}^*\mathbf{R})$ is simply a diagonal matrix.

- **Fastfood transform** [210]: the matrix $\mathbf{V} = \frac{1}{\sigma\sqrt{N}}\mathbf{S}\mathbf{H}\mathbf{G}\mathbf{P}\mathbf{H}\mathbf{B}$, with $\mathbf{S}, \mathbf{G}, \mathbf{B}$ some diagonal matrices, $\mathbf{P}$ a permutation matrix and $\mathbf{H}$ the Hadamard matrix, is used for fast approximation of the Gaussian kernel of scale σ. Since $\mathbf{H} \in \mathcal{B} \subseteq \mathcal{B}\mathcal{B}^*$ and $\mathbf{P} \in \mathcal{B}\mathcal{B}^*$, we get $\mathbf{V} \in (\mathcal{B}\mathcal{B}^*)^3$, but it is also possible to show that $\mathbf{V} \in (\mathcal{B}\mathcal{B}^*)^2$ [75, Lemma J.7].

Applications of the square dyadic butterfly factorization. In practice, representations based on the square dyadic butterfly factorization can be used to replace hand-crafted structures in speech processing models or channel shuffling in certain convolutional neural networks [382], or to learn a latent permutation [75]. It can also be used for efficient channel fusion to replace pointwise convolution [342], or to compress some linear layers using truncated butterfly matrices [5].

4.5.2 Other variants of the butterfly factorization

Other variants of butterfly factorization have been proposed to compress layers in neural network, such as the Monarch factorization [72], the block butterfly factorization [71, 248], or the deformable butterfly factorization [241]. However, the current literature lacks a unifying mathematical description of these different variants. In Chapter 7, we introduce a general framework to study them by considering general sparsity patterns of the form (1.2). Therefore, the detailed description of the sparsity patterns in these variants is deferred to Chapter 7. For now, we describe their applications for the compression of deep learning models.

Compression of fully-connected layers. One application of the Monarch factorization [72] is to train from scratch a compressed transformer architecture in a vision or a language task, where weight matrices in the attention and the feed-forward network modules are replaced by a matrix admitting a Monarch factorization. However, as discussed in Chapter 10, the experiments in [72] do not compare the obtained performance with the low-rank baseline.

Compression of convolutional layers. In the context of image classification, deformable butterfly factorization [241] can be used to compress 2D-convolutional layers in the following way. Given a kernel weight tensor $\mathbf{K} \in \mathbb{R}^{N\times C\times D\times D}$ and an input tensor $\mathbf{U} \in \mathbb{R}^{C\times X\times Y}$, the entries of the output tensor $\mathbf{V} = \mathbf{K} * \mathbf{U}$ as defined in (4.5) can be computed equivalently by the matrix product $\tilde{\mathbf{V}} = \tilde{\mathbf{K}}\tilde{\mathbf{U}} \in \mathbb{R}^{N\times(X-D+1)(Y-D+1)}$, where $\tilde{\mathbf{K}} \in \mathbb{R}^{N\times D^2C}$ is a matrix obtained by reshaping $\mathbf{K}$ in a certain way, and $\tilde{\mathbf{U}} \in \mathbb{R}^{D^2C\times(X-D+1)(Y-D+1)}$ is a matrix generated from $\mathbf{U}$ using the `im2col` command [187]. Then, in [241], the number of parameters in a convolutional layer is reduced by replacing $\tilde{\mathbf{K}}$ with a product of deformable butterfly factors $\mathbf{X}_1 \dots \mathbf{X}_L$. The output $\mathbf{V}$ is then computed as follows:

1. generate $\tilde{\mathbf{U}}$ from $\mathbf{U}$ using the `im2col` command;
2. compute $\tilde{\mathbf{V}} = \mathbf{X}_1 \dots \mathbf{X}_L\tilde{\mathbf{U}}$;
3. reshape the matrix $\tilde{\mathbf{V}}$ in a certain way to obtain the output tensor $\mathbf{V}$.

As detailed in Chapter 9, even though this procedure can reduce the number of parameters with a small drop of performance compared to the dense network, it does not necessarily yield time acceleration, unless there is a proper implementation for the generation of $\tilde{\mathbf{U}}$ from $\mathbf{U}$.

Orthogonal parameter-efficient finetuning. Block butterfly factorization [71] can be used in the context of *orthogonal* parameter-efficient finetuning in transfer learning [248], for various language and vision downstream tasks. In contrast to the parameterization (4.10) where the pretrained matrix $\mathbf{W}_0 \in \mathbb{R}^{M\times N}$ is perturbed *additively* by a low-rank matrix, the weight matrix $\mathbf{W} \in \mathbb{R}^{M\times N}$ of a given

linear layer during the transfer can be parameterized as a *multiplicative* perturbation of $\mathbf{W}_0$, in the sense that $\mathbf{W} = \mathbf{Q}\mathbf{W}_0$ where $\mathbf{Q} \in \mathbb{R}^{M \times M}$ is a learnable parameter during training, and $\mathbf{W}_0$ is frozen. In orthogonal finetuning [305], the matrix $\mathbf{Q}$ is constrained to be orthogonal, and to achieve parameter efficiency, it can be parameterized by a block butterfly factorization $\mathbf{X}_1 \dots \mathbf{X}_L$, with the additional constraint that each factor $\mathbf{X}_\ell$ for $\ell \in [\![L]\!]$ is orthogonal, as proposed in [248]. Note that the idea of constructing an orthogonal matrix with butterfly matrices was previously used in [297] for preconditioning in Gaussian elimination, and in [276] for quadrature rules on the hypersphere.

Butterfly factorization after dense pretraining. In [72, 241], instead of training a butterfly compressed neural network from scratch, it is possible to consider a train-then-decompose schedule, where dense pretrained weights are decomposed as a product of butterfly factors (following either a Monarch factorization [72] or a deformable butterfly factorization [241]), in order to initialize the parameters of the butterfly compressed network. However, the decomposition algorithm used in [241] is based on alternative least squares and does not have guarantee of success. More importantly, the decomposition models in [72, 241] do not take into account permutation symmetries of the parameterization of a neural network, which can yield high approximation error in practice (see Chapter 10 for more details).

4.6 Conclusion

In this chapter, we gave an overview of different techniques for removing parameter redundancy in neural networks, either via sparsity, low-rank decomposition or butterfly factorization. Butterfly factorization can be interesting for neural network compression due to its expressivity as a generic representation of structure linear maps. Moreover, the butterfly sparse factors have a structured sparsity pattern that can potentially lead to an efficient implementation of the corresponding sparse matrix multiplication. However, we currently observe four main limitations in the current studies.

1. Previous works that introduced the square dyadic butterfly factorization [74, 75], the Monarch factorization [72], the deformable butterfly factorization [241] or the block butterfly factorization [71] did not characterize analytically the set of matrices that can be well-approximate by such a factorization. This prevents us from understanding whether or not a pretrained matrix can be well-decomposed by the butterfly factorizations, in the train-then-decompose schedule. Moreover, except for the Monarch factorization, previous work did not propose a decomposition algorithm with guarantees of success.

2. The decomposition models in all previous work did not consider permutation symmetries in the parameterization of neural networks.

3. There are few numerical reports about the current time efficiency of GPU implementations for butterfly sparse matrix multiplication (see Chapter 9 for more details).

4. Previous works did not provide a low-rank baseline when training from scratch a neural network compressed by butterfly factorization (see Chapter 10 for more details).

These limitations will motivate our study in Chapters 6 to 9.

Part II

Contributions

Chapter 5

Self-supervised learning with rotation-invariant kernels

This chapter presents our contribution related to self-supervised learning for visual representations. We propose a novel kernel framework for analyzing previous existing self-supervised learning methods, which opens perspective for reducing their computational complexity.

5.1 Introduction

Self-supervised learning is a promising approach for learning visual representations: recent methods [44, 124, 143, 162] reach the performance of supervised pretraining in terms of quality for transfer learning in many downstream tasks, like classification, object detection, semantic segmentation, etc. These methods rely on some prior knowledge on images: the semantic of an image is *invariant* [267] to some *small* transformations of the image, such as cropping, blurring, color jittering, etc. One way to design an objective function that encodes such an invariance property is to enforce two different augmentations of the same image to have a similar representation (or embedding) when they are encoded by the neural network. However, the main issue with this kind of objective function is to avoid an undesirable loss of information [177, 190] where, e.g., the network learns to represent all images by the same constant representation. Hence, one of the main challenges in self-supervised learning is to propose an efficient way to *regularize* the embedding distribution in order to avoid such a *collapse*.

Introducing a generic regularization loss. Our contribution is to propose a generic regularization loss promoting the embedding distribution to be close to the uniform distribution on the hypersphere, with respect to the *maximum mean*

The material of this chapter is based on [388], in collaboration with Gilles Puy, Elisa Riccietti, Patrick Pérez and Rémi Gribonval.

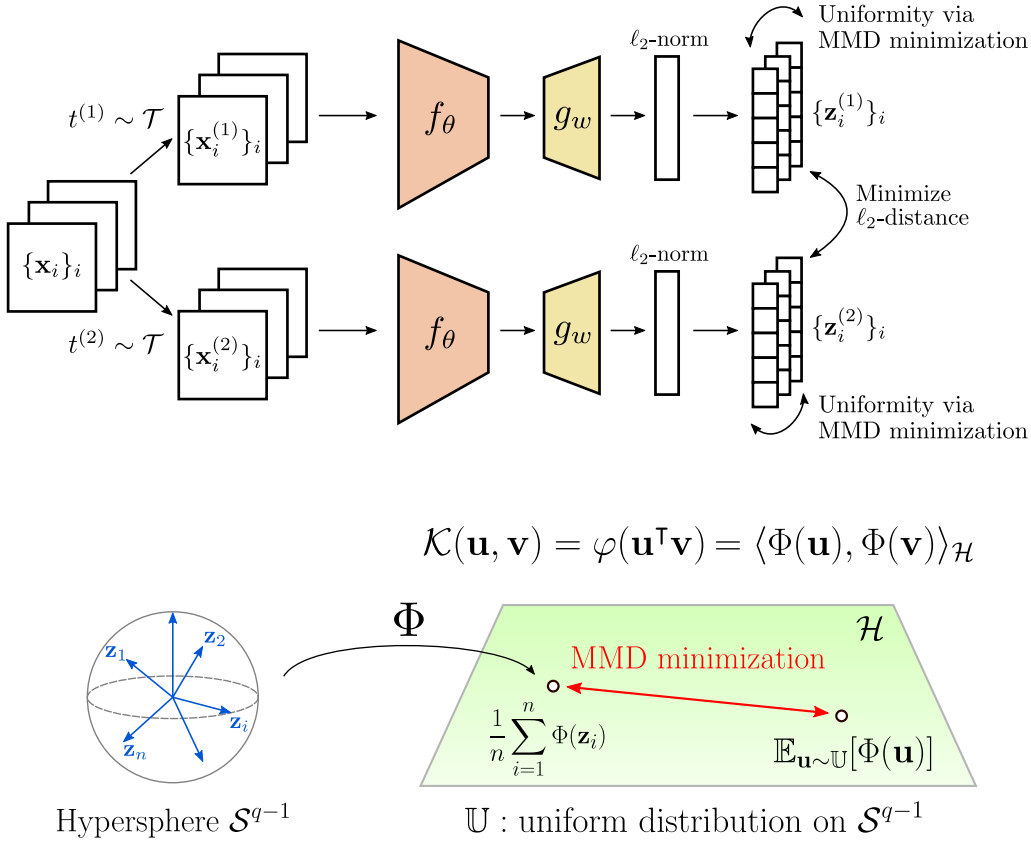


Figure 5.1: **Self-supervised learning with rotation-invariant kernels**. The invariance criterion minimizes the ℓ_2-distance between two normalized embeddings $\{\mathbf{z}_i^{(v)}\}_{v=1,2}$ of two views of the same image $\mathbf{x}_i$ encoded by the backbone f_θ and the projection head g_w. To avoid collapse, the embedding distribution is regularized to be close to the uniform distribution on the hypersphere, in the sense of the MMD associated with a rotation-invariant kernel $\mathcal{K}(\mathbf{u}, \mathbf{v}) = \varphi(\mathbf{u}^\top \mathbf{v})$ defined on the hypersphere; Φ denotes the feature map associated to this kernel.

discrepancy (MMD). The MMD is a probability divergence based on the notion of embedding probabilities in a reproducing kernel Hilbert space (RKHS), using the so-called kernel mean embedding mapping. Inspired by high-dimensional statistical tests for uniformity that are rotation-invariant [118], we choose to embed probability distributions using *rotation-invariant* kernels on the hypersphere, also known as dot-product kernels, i.e., kernels for which the evaluation for two vectors depends only on their inner product [325]. This chapter shows that such an approach leads to important theoretical and practical consequences for self-supervised learning.

Unification under a kernel framework. We demonstrate that our regularization loss family parameterized by such rotation-invariant kernels encompasses several regularizers of former methods. As illustrated in Table 5.1, they are variants of our generic loss with different kernels:

Table 5.1: Correspondence between kernel choices $\mathcal{K}(\cdot,\cdot)$ in our generic regularization loss and regularizers of former methods.

$\mathcal{K}(\mathbf{u},\mathbf{v})$	Method
$(\mathbf{u}\mathbf{v}^\top)^2$	Contrastive
$e^{-t\|\mathbf{u}-\mathbf{v}\|_2^2}$	AUH
$C-\|\mathbf{u}-\mathbf{v}\|_2^{2s-q+1}$	PointContrast
$b_1\mathbf{u}\mathbf{v}^\top + b_2\frac{q(\mathbf{u}\mathbf{v}^\top)^2-1}{q-1}$	Analogous to VICReg (cf. Section 5.3.3)

- the quadratic kernel yields the general sample-contrastive criterion of [120] that englobes many contrastive learning methods like [156], cf. Appendix A.1.2;
- the radial basis function (RBF) kernel yields the uniformity loss of Alignment & Uniformity on the Hypersphere (AUH) [354];
- the generalized distance kernel (cf. Example 2) yields one of the regularizations used in PointContrast [362];
- and a linear combination of the linear kernel and the quadratic kernel yields a regularizer that promotes the covariance matrix of the embedding distribution to be proportional to the identity matrix, similarly to information-maximization methods like VICReg [15].

In other words, these former methods turn out to be particular ways of minimizing the MMD between the embedding distribution and the uniform distribution on the hypersphere during training, with various specific kernel choices. The proposed generic regularization approach opens perspectives to leverage more widely the literature on kernel methods in order to improve self-supervised learning.

Identifying a new competitive kernel. Numerically, we show in a rigorous experimental setting with a separate validation set for hyperparameter tuning that our method yields fully competitive results compared to the state of the art, when choosing truncated kernels of the form $\mathcal{K}(\mathbf{u},\mathbf{v}) = \sum_{\ell=0}^{L} b_\ell P_\ell(q;\mathbf{u}^\top\mathbf{v})$, with $L \in \{2,3\}$, $b_\ell \geq 0$ for $\ell \in \{0,\dots,L\}$, where $P_\ell(q;\cdot)$ denotes the Legendre polynomial of order ℓ, dimension q. To our knowledge, this kernel choice has not been considered in previous self-supervision methods. Therefore, we introduce SFRIK (SelF-supervised learning with Rotation-Invariant Kernels, pronounced like "spheric"), which regularizes the embedding distribution to be close to the uniform distribution with respect to the MMD associated with such a truncated kernel, as summarized in Figure 5.1.

Reducing pretraining complexity. Importantly, our method significantly reduces time and memory complexity for self-supervised training compared to information-maximization methods. Due to the *kernel trick*, the complexity of SFRIK's loss is quadratic in the batch size and *linear* in the embedding dimension, instead of being quadratic as in VICReg. In practice, SFRIK's pretraining time is up to 19% faster than VICReg for an embedding dimension of 16384, and it can scale at dimension 32768, as opposed to VICReg whose memory requirement is too large at this dimension for a machine with 8 GPUs and 32GB of memory per GPU. Hence our work opens perspectives in self-supervised learning on embedded devices with limited memory like in [360].

Summary. We summarize our contributions are as follows:

1. We introduce a generic regularization loss based on kernel mean embeddings with rotation-invariant kernels on the hypersphere for self-supervised learning of image representations.

2. We show that our loss family encompasses several previous self-supervised learning methods, like uniformity-based and information-maximization methods.

3. We numerically show that SFRIK significantly reduces time and memory complexity for self-supervised training, while remaining fully competitive with the state of the art.

5.2 Related work

Instance discrimination methods typically rely on a contrastive loss that behaves asymptotically like an invariance term and uniformity loss on the hypersphere in the limit of infinite samples. Our contribution is to formalize and generalize existing uniformity-based methods by using kernel mean embeddings. To the best of our knowledge, the proposed kernel framework establishes the first connection between uniformity-based methods and information-maximization methods like VICReg.

Instance discrimination. One way of learning image representations that are invariant to predefined image transformations [267] is to rely on an instance classification approach [358]. Typically, contrastive learning [50, 51, 162, 165, 171, 290] discriminates instances within a batch of sampled images using the *noise contrastive estimator* [148], by attracting embeddings of transformed images coming from the same image instance, and repulsing embeddings coming from different image instances. In practice, this estimator needs a large number of image representations in order to achieve good results, which requires a large batch size like

SimCLR [50] or a memory bank [162,358]. In the limit of infinite samples, the contrastive loss is shown to behave asymptotically like the alignment and uniformity loss of AUH.

Uniformity on the hypersphere. Existing uniformity-based methods avoid collapse by regularizing the embedding distribution to be somehow close to the uniform distribution on the hypersphere, which has a high entropy. [30] performs this kind of regularization by aligning the learned representations on a fixed number of vectors sampled uniformly at random on the hypersphere. AUH maximizes the average pairwise distance between embeddings using an RBF kernel, in the spirit of *energy minimization methods* that address the problem of scattering points evenly on the hypersphere [34, 157, 247]. Although alternative high-entropy prior distributions (e.g., the uniform distribution on the hypercube) can be used for regularization [52], encoding images into ℓ_2-normalized representations helps to stabilize training [249, 298, 319].

Kernel mean embedding. As a contribution, our generic loss formalizes and generalizes these previous uniformity losses, by relying on kernel mean embeddings (cf. Appendix A.1.1) to measure the distance between probability distributions on high-dimensional spaces, using the MMD pseudometric [37, 91, 141, 232] with rotation-invariant kernels on the hypersphere [90, 256, 302, 325]. These tools are adapted for high-dimensional problems on the hypersphere whose geometry is different from the one in small dimension, as illustrated by [118]: many statistical tests for uniformity on the hypersphere, i.e., tests for rejecting the null hypothesis where a batch of normalized vectors is sampled from the uniform distribution on the hypersphere, are in fact precisely estimators of the MMD between the embedding distribution and the uniform distribution, for different kernels. Our kernel method for self-supervision is complementary to [231], in which the dependency between image instances and their embedding is maximized with respect to the Hilbert-Schmidt independence criterion (cf. Appendix A.1.3).

Information maximization. Our generic regularization approach has the benefit of connecting uniformity-based and *information-maximization methods* [15, 101, 375]. The latter are alternatives to distillation methods [45, 55, 123, 124, 143] where a student network learns to predict the representations of a teacher network. In such methods, using various architecture tricks (like prediction head, stop-gradient, momentum encoder, batch normalization or centering) is shown empirically to be sufficient to avoid collapse without instance discrimination, even though it is not fully understood how these multiple factors induce a regularization during training [313, 339]. Instead of using these tricks, information-maximization methods use a Siamese architecture and avoid collapse by maximizing the statistical information of a batch of embeddings, using a whitening operation [101], or an explicit regularization term making the covariance [15] or the cross-correlation [375] matrix close to a scaled identity matrix. This chapter

shows that our generic regularization loss with an appropriate kernel also promotes the covariance matrix of the embedding distribution to be proportional to the identity matrix. But in contrast to VICReg which explicitly computes the covariance matrix, our method uses the kernel trick to significantly reduce complexity at large embedding dimensions.

5.3 Method description

Given an unlabeled dataset of images $\mathbf{x}_i \sim \mathbb{P}$, $i \in [\![N]\!]$ for an integer N, sampled independently from a data distribution $\mathbb{P}$, the goal is to learn a backbone network f_θ parameterized by θ (e.g., a convolutional neural network) such that any new image $\mathbf{x} \sim \mathbb{P}$ is encoded by a good representation $f_\theta(\mathbf{x})$ whose quality is evaluated in several downstream tasks (see Section 5.4).

5.3.1 Invariance and uniformity for self-supervision

Our self-supervised learning method (see Figure 5.1) follows the principle of the recent methods like SimCLR or VICReg. During self-supervised training, each image $\mathbf{x}_i$ is augmented using two different random transformations $t^{(1)}$ and $t^{(2)}$ sampled from a distribution $\mathcal{T}$, which yields two views $\mathbf{x}_i{}^{(1)} := t^{(1)}(\mathbf{x}_i)$ and $\mathbf{x}_i{}^{(2)} := t^{(2)}(\mathbf{x}_i)$ of the image $\mathbf{x}_i$. Two representations $\mathbf{z}_i{}^{(v)}$ ($v = 1, 2$) are obtained by encoding each $\mathbf{x}_i{}^{(v)}$ with the backbone f_θ and ℓ_2-normalizing the resulting feature vector. For a given subset of indices $I \subseteq [\![N]\!]$, we write $\mathbf{Z}_I{}^{(v)} := \{\mathbf{z}_i{}^{(v)}\}_{i \in I}$. The backbone f_θ is trained by minimizing the total objective function:

$$\mathcal{L} = \mathbb{E}_{t^{(1)}, t^{(2)} \sim \mathcal{T}} \, \mathbb{E}_{I \subseteq [\![N]\!]} \, \ell(\mathbf{Z}_I{}^{(1)}, \mathbf{Z}_I{}^{(2)}), \tag{5.1}$$

where batches I are drawn at random with a prescribed batch size, and the loss ℓ is a weighted sum involving an *alignment* term ℓ_a and a *uniformity* term ℓ_u, in the spirit of AUH:

$$\ell(\mathbf{Z}_I{}^{(1)}, \mathbf{Z}_I{}^{(2)}) := \lambda \, \ell_a(\mathbf{Z}_I{}^{(1)}, \mathbf{Z}_I{}^{(2)}) + \frac{1}{2} \left(\ell_u(\mathbf{Z}_I{}^{(1)}) + \ell_u(\mathbf{Z}_I{}^{(2)}) \right); \tag{5.2}$$

$\lambda > 0$ is a hyperparameter that tunes the balance between the two terms. The loss ℓ_a enforces the invariance property of the model, and is defined for a batch $I \subseteq [\![N]\!]$ of cardinality $|I|$ as:

$$\ell_a(\mathbf{Z}_I{}^{(1)}, \mathbf{Z}_I{}^{(2)}) := \frac{1}{|I|} \sum_{i \in I} \left\| \mathbf{z}_i{}^{(1)} - \mathbf{z}_i{}^{(2)} \right\|_2^2. \tag{5.3}$$

Our main contribution is in the choice of the uniformity term ℓ_u, detailed in the rest of the section. Note that instead of applying the loss (5.2) to the output of f_θ (called *image representation*), we add a projection head g_w (a multi-layer perceptron) parameterized by w to the output of f_θ and apply (5.2) at the output of

g_w (called *image embedding*). This common practice [44, 143] improves the performance in the downstream tasks. Therefore, denoting $\mathcal{S}^{q-1}$ the unit hypersphere in $\mathbb{R}^q$, the image embedding actually reads

$$\mathbf{z}_i^{(v)} := \frac{(g_w \circ f_\theta)(\mathbf{x}_i^{(v)})}{\left\|(g_w \circ f_\theta)(\mathbf{x}_i^{(v)})\right\|_2} \in \mathcal{S}^{q-1}.$$

Both g_w and f_θ are jointly trained without supervision by minimizing (5.1) using a stochastic mini-batch algorithm. After training, g_w is discarded and only f_θ is kept for the downstream tasks.

5.3.2 Uniformity loss via MMD minimization

We continue by explaining our generic kernel formulation of ℓ_u using the MMD pseudometric and rotation-invariant kernels. Then we provide examples of such kernels and describe our kernel choice.

MMD pseudometric and rotation-invariant kernels

Our uniformity loss relies on a divergence in the space of probability distributions based on a positive definite kernel $\mathcal{K}$ defined on some space $\mathcal{X}$. Denoting $\mathcal{H}$ the corresponding RKHS with norm $\|\cdot\|_{\mathcal{H}}$, the MMD between two probability distributions $\mathbb{Q}_1, \mathbb{Q}_2$ on $\mathcal{X}$ can be expressed as the distance in $\|\cdot\|_{\mathcal{H}}$ between their kernel mean embeddings [33, 274]:

$$\mathrm{MMD}(\mathbb{Q}_1, \mathbb{Q}_2) = \left\| \int_{\mathcal{X}} \mathcal{K}(\mathbf{u}, \cdot) d\mathbb{Q}_1(\mathbf{u}) - \int_{\mathcal{X}} \mathcal{K}(\mathbf{u}, \cdot) d\mathbb{Q}_2(\mathbf{u}) \right\|_{\mathcal{H}}. \tag{5.4}$$

We propose to use this pseudometric[1] to measure the distance between the probability distribution of the embeddings $\mathbf{z}_i^{(v)}$ ($v = 1, 2$) and the uniform probability distribution on the hypersphere $\mathcal{S}^{q-1}$ defined by $\mathbb{U} := \sigma_{q-1} / \left|\mathcal{S}^{q-1}\right|$, where σ_{q-1} denotes the normalized Hausdorff surface measure on $\mathcal{S}^{q-1}$, and $\left|\mathcal{S}^{q-1}\right| := \int_{\mathcal{S}^{q-1}} d\sigma_{q-1} = 2\pi^{\frac{q}{2}} / \Gamma(\frac{q}{2})$ is the surface area of $\mathcal{S}^{q-1}$, with Γ denoting the Gamma function. Intuitively, a good choice of kernel for measuring the distance (5.4) should distinguish any probability distribution from the uniform distribution. Inspired by statistical tests for uniformity that are rotation-invariant [118], we propose to use rotation-invariant kernels on $\mathcal{X} := \mathcal{S}^{q-1}$ of the form $\mathcal{K}(\mathbf{u}, \mathbf{v}) := \varphi(\mathbf{u}^\top \mathbf{v})$ with φ a continuous function defined on the interval $[-1, 1]$ [325]. The following theorem characterizes the form of function φ that ensures positive definiteness of $\mathcal{K}$, and thus that (5.4) is a valid pseudometric.

[1] In general, the MMD is only a pseudometric. It is a metric if, and only if, the considered kernel is characteristic [115].

Theorem 5.1 (from [318, Theorem 1]). *The kernel $\mathcal{K}(\mathbf{u},\mathbf{v}) := \varphi(\mathbf{u}^\top\mathbf{v})$ on $\mathcal{X} := \mathcal{S}^{q-1}$ with φ continuous is positive definite if, and only if, the function φ admits an expansion:*

$$\varphi(t) = \sum_{\ell=0}^{+\infty} b_\ell P_\ell(q;t), \quad \text{with} \quad b_\ell \geq 0, \tag{5.5}$$

where

$$P_\ell(q;t) := \ell!\,\Gamma\left(\frac{q-1}{2}\right)\sum_{k=0}^{\lfloor \frac{\ell}{2} \rfloor}\left(-\frac{1}{4}\right)^k \frac{(1-t^2)^k t^{\ell-2k}}{k!\,(\ell-2k)!\,\Gamma(k+\frac{q-1}{2})}$$

is the Legendre (or Gegenbauer) polynomial of degree ℓ in dimension q [277, (2.32)].

As we are interested in measuring the distance between the embedding distribution and the uniform distribution on the hypersphere $\mathbb{U}$, we compute the kernel mean embedding of $\mathbb{U}$ for a kernel satisfying the condition of Theorem 5.1 using the following known result used, e.g., implicitly in [36]. As we could not locate a formal proof, we provide one in Appendix A.2.1.

Lemma 5.1. *Let $\mathcal{K}(\mathbf{u},\mathbf{v}) := \varphi(\mathbf{u}^\top\mathbf{v})$ be a rotation-invariant kernel on $\mathcal{X} := \mathcal{S}^{q-1}$ where φ admits the expansion (5.5). Then:*

- *The kernel mean embedding of the uniform distribution $\mathbb{U}$ on $\mathcal{S}^{q-1}$ is constant:*

$$\int_{\mathcal{S}^{q-1}} \mathcal{K}(\mathbf{u},\mathbf{v})\,\mathrm{d}\mathbb{U}(\mathbf{u}) = b_0 \in \mathbb{R}, \quad \forall \mathbf{v} \in \mathcal{S}^{q-1}.$$

- *The kernel mean embedding of any probability distribution $\mathbb{Q}$ defined on the hypersphere satisfies:*

$$\int_{\mathcal{S}^{q-1}} \mathcal{K}(\mathbf{u},\cdot)\,\mathrm{d}\mathbb{Q}(\mathbf{u}) = b_0 + \int_{\mathcal{S}^{q-1}} \tilde{\mathcal{K}}(\mathbf{u},\cdot)\,\mathrm{d}\mathbb{Q}(\mathbf{u}),$$

  where $\tilde{\mathcal{K}}(\mathbf{u},\mathbf{v}) := \tilde{\varphi}(\mathbf{u}^\top\mathbf{v})$ for any $\mathbf{u},\mathbf{v} \in \mathcal{S}^{q-1}$ with $\tilde{\varphi} := \sum_{\ell=1}^{+\infty} b_\ell P_\ell(q;\cdot)$.

Using Lemma 5.1 in (5.4) yields

$$\mathrm{MMD}(\mathbb{Q},\mathbb{U}) = \left\| \int_{\mathcal{S}^{q-1}} \tilde{\mathcal{K}}(\mathbf{u},\cdot)\mathrm{d}\mathbb{Q}(\mathbf{u}) \right\|_{\mathcal{H}}$$

for any probability distribution $\mathbb{Q}$ on $\mathcal{S}^{q-1}$. Then, by the reproducing property in the RKHS $\mathcal{H}$, the squared MMD satisfies, for *any* rotation-invariant kernel $\mathcal{K}$ verifying the condition of Theorem 5.1:

$$\mathrm{MMD}^2(\mathbb{Q},\mathbb{U}) = \mathbb{E}_{\mathbf{z},\mathbf{z}'\sim\mathbb{Q}}\left[\tilde{\mathcal{K}}\left(\mathbf{z},\mathbf{z}'\right)\right], \quad \text{with } \mathbf{z},\mathbf{z}' \text{ i.i.d.} \tag{5.6}$$

Estimator of the squared MMD and kernel choices

The proposed uniformity loss ℓ_u for self-supervision is a *biased estimator* [141] of $\mathrm{MMD}^2(\mathbb{Q}, \mathbb{U})$ in (5.6). Given a batch $\mathbf{Z}_I := \{\mathbf{z}_i\}_{i\in I}$ sampled from $\mathbb{Q}$, our uniformity loss is:

$$\ell_u(\mathbf{Z}_I) = \widehat{\mathrm{MMD}}^2(\mathbb{Q}, \mathbb{U}; \{\mathbf{Z}_I\}) := \frac{1}{|I|^2}\sum_{i\in I}\sum_{i'\in I}\tilde{\mathcal{K}}(\mathbf{z}_i, \mathbf{z}_{i'}) = \frac{1}{|I|^2}\sum_{i\in I}\sum_{i'\in I}\tilde{\varphi}(\mathbf{z}_i^\top \mathbf{z}_{i'}). \tag{5.7}$$

In our framework, any rotation-invariant kernel satisfying the condition of Theorem 5.1 can be used to compute (5.7) and train a self-supervised model by minimizing (5.1). The uniformity term (5.7) can be interpreted as an *energy functional* [36]: minimizing the average pairwise energy quantified by $\tilde{\mathcal{K}}$ tends to scatter evenly the embeddings on the hypersphere. We now give examples of kernels that can be used for this uniformity term. This illustrates that our framework offers a unification of several strategies for self-supervision.

Example 5.1 (RBF kernel)**.** *Using $\mathcal{K}(\mathbf{u}, \mathbf{v}) = e^{-t\|\mathbf{u}-\mathbf{v}\|_2^2}$ (with $t > 0$) in the uniformity term ℓ_u (5.7) yields the regularization term from AUH, with the only difference that AUH uses the logarithm of the energy functional as their uniformity loss.*

Example 5.2 (Generalized distance kernel)**.** *It is defined as $\mathcal{K}(\mathbf{u}, \mathbf{v}) := C - \|\mathbf{u} - \mathbf{v}\|_2^{2s-q+1}$ with $\frac{q-1}{2} < s < \frac{q+1}{2}$ and $C > 0$ sufficiently large [36]. A variation of this kernel choice is, e.g., used in the hard-contrastive loss of PointContrast for self-supervision on point clouds.*

Example 5.3 (Truncations of the Laplace-Fourier series)**.** *A truncated kernel up to order L [36] is a kernel $\mathcal{K}(\mathbf{u}, \mathbf{v}) = \sum_{\ell=0}^{L} b_\ell P_\ell(q; \mathbf{u}^\top \mathbf{v})$, with $b_\ell \geq 0$ for $\ell = 0, \ldots, L$. It admits a closed-form expression given by the definition of Legendre polynomials $P_\ell(q, \cdot)$ in Theorem 5.1, e.g., $P_1(q, t) = t$, $P_2(q, t) = \frac{qt^2-1}{q-1}$, $P_3(q, t) = \frac{(q+2)t^3-3t}{q-1}$. We explore numerically this kernel choice in Section 5.4, since it has never been considered in previous self-supervision methods.*

The expansion of φ in Legendre polynomials (5.5) for the RBF (Example 1) and the generalized distance kernel (Example 2) verifies $b_\ell > 0$ for each integer ℓ (see Appendix A.2.2). By [264, Theorem 10], this is a necessary and sufficient condition for a rotation-invariant kernel to be *universal*, and universality is a sufficient condition for injectivity of the corresponding kernel mean embedding mapping, i.e., the kernel is *characteristic* [114]. The benefit of this property is to guarantee that the uniform distribution $\mathbb{U}$ is the unique solution to the minimization problem:

$$\min\left\{\mathrm{MMD}(\mathbb{Q}, \mathbb{U}) \mid \mathbb{Q} \text{ is a probability distribution on } \mathcal{S}^{q-1}\right\}.$$

In contrast, the truncated rotation-invariant kernel up to an order L (Example 3) is not universal. Yet, our experiments in Section 5.4 show that truncated kernels up to order $L \in \{2, 3\}$ provide better results than, e.g., AUH whose uniformity loss is based on the RBF kernel.

Summary. The uniformity loss in our method, called SFRIK, corresponds to (5.7) with a truncated kernel up to order $L = 3$ and satisfies:

$$\ell_u(\{\mathbf{z}_i\}_{i\in I}) = \frac{1}{|I|^2}\sum_{i\in I}\sum_{i'\in I}\left(b_1\mathbf{z}_i^\top\mathbf{z}_{i'} + b_2\frac{q(\mathbf{z}_i^\top\mathbf{z}_{i'})^2 - 1}{q-1} + b_3\frac{(q+2)(\mathbf{z}_i^\top\mathbf{z}_{i'})^3 - 3\mathbf{z}_i^\top\mathbf{z}_{i'}}{q-1}\right), \tag{5.8}$$

where $b_\ell \geq 0$, $\ell \in [\![3]\!]$, are hyperparameters, and q is the dimension of the image embedding $\mathbf{z}_i$.

5.3.3 Connection with information-maximization methods

We now show that an appropriate kernel in the proposed uniformity term (5.7) leads to a regularizer that maximizes a statistical measure of information analog to the one used in VICReg. To the best of our knowledge, this is the first connection made between uniformity-based and information-maximization methods.

Reminders on VICReg. The regularization loss of VICReg is a weighted sum between two terms:

$$\begin{aligned} v(\mathbf{Z}_I) &:= \frac{1}{q}\sum_{j=1}^{q}\max\left(0, \gamma - \sqrt{\mathrm{Var}(\mathbf{Z}_I)[j] + \varepsilon}\right), \\ c(\mathbf{Z}_I) &:= \frac{1}{q}\sum_{1\leq j\neq j'\leq q}[C(\mathbf{Z}_I)]_{j,j'}^2, \end{aligned} \tag{5.9}$$

for a batch of image embeddings $\mathbf{Z}_I := \{\mathbf{z}_i\}_{i\in I}$, where $\mathbf{z}[j]$ denotes the j-th coordinate of a (random) vector $\mathbf{z}$ and ε is a fixed small scalar. The *variance term* $v(\mathbf{Z}_I)$ enforces the empirical variance $\mathrm{Var}(\mathbf{Z}_I)[j] := \frac{1}{|I|-1}\sum_{i\in I}(\mathbf{z}_i[j] - \bar{\mathbf{z}}[j])^2$ in each coordinate $j \in [\![q]\!]$ to be above a certain threshold $\gamma^2 > 0$, where $\bar{\mathbf{z}}$ here is the empirical mean of $\mathbf{Z}_I$. The *covariance term* $c(\mathbf{Z}_I)$ enforces the non-diagonal entries of the empirical covariance matrix $C(\mathbf{Z}_I) := \frac{1}{|I|-1}\sum_{i\in I}(\mathbf{z}_i - \bar{\mathbf{z}})(\mathbf{z}_i - \bar{\mathbf{z}})^\top$ to be zero.

Connection between SFRIK and VICReg. We claim that minimizing $\mathrm{MMD}^2(\mathbb{Q}, \mathbb{U})$ associated with a rotation-invariant kernel promotes the covariance matrix of $\mathbb{Q}$ to be proportional to the identity matrix, in the following sense.

Proposition 5.1. *Consider a kernel admitting an expansion* $\tilde{\mathcal{K}}(\mathbf{u}, \mathbf{v}) = \sum_{\ell=1}^{\infty} b_\ell P_\ell(q; \mathbf{u}^\top\mathbf{v})$ *with* $b_1, b_2 > 0$ *and* $b_\ell \geq 0$ *for any* $\ell \geq 3$. *Then, for any probability distribution* $\mathbb{Q}$ *defined on* $\mathcal{S}^{q-1}$*:*

$$\begin{aligned} \mathrm{MMD}(\mathbb{Q}, \mathbb{U}) = 0 &\implies \mathbb{E}_{\mathbf{z}\sim\mathbb{Q}}[\mathbf{z}] = 0 \quad \textit{and} \quad \mathbb{E}_{\mathbf{z}\sim\mathbb{Q}}[\mathbf{z}\mathbf{z}^\top] = \frac{1}{q}\mathbf{I}_q \\ &\implies \mathbb{E}_{\mathbf{z}\sim\mathbb{Q}}\left[(\mathbf{z} - \mathbb{E}[\mathbf{z}])(\mathbf{z} - \mathbb{E}[\mathbf{z}])^\top\right] = \frac{1}{q}\mathbf{I}_q. \end{aligned}$$

In other words, the regularization both in VICReg and SFRIK induces the embedding distribution to have a covariance matrix with zero non-diagonal entries. The diagonal entries of the covariance matrix are encouraged to be equal to $1/q$ in SFRIK, and greater than γ^2 in VICReg (we recall that the image embeddings $\{\mathbf{z}_i\}_{i\in I}$ are not ℓ_2-normalized in VICReg). However, one difference in terms of regularization behavior is that SFRIK encourages the expectation of the embedding distribution to be zero. This is not the case for VICReg, cf. (5.9).

Proof of Proposition 5.1. Let us consider for simplicity a truncated kernel $\tilde{\mathcal{K}}(\mathbf{u},\mathbf{v}) = \sum_{\ell=1}^{L} b_\ell P_\ell(q;\mathbf{u}^\top\mathbf{v})$ of order $L=2$, and assume $b_1, b_2 > 0$. The reasoning is similar for the general case where the kernel is not truncated, and the complete proof is deferred to Appendix A.2.3.

By the addition theorem [277, Theorem 2, §1], there exists a feature map $\Phi : \mathcal{S}^{q-1} \to \mathbb{R}^m$ involving an orthonormal basis of *spherical harmonics* (homogeneous harmonic polynomials restricted to the hypersphere) of order 1 and 2 such that $\Phi(\mathbf{u})^\top\Phi(\mathbf{v}) = \tilde{\mathcal{K}}(\mathbf{u},\mathbf{v})$. Hence, the kernel mean embedding of a distribution in the associated RKHS contains its first and second-order moments (see Appendix A.2.3). Therefore, denoting $N(q,\ell)$ the dimension of the space of spherical harmonics of order ℓ, dimension q, and defining $\Phi_\ell : \mathcal{S}^{q-1} \to \mathbb{R}^{N(q,\ell)}$, $\mathbf{z} \mapsto (Y_{\ell,k}(\mathbf{z}))_{k=1}^{N(q,\ell)}$ for $\ell \in \{1,2\}$ with

$$\begin{aligned}
\{Y_{1,k}\}_{k=1}^{N(q,1)} &:= \{\mathbf{u} \mapsto \mathbf{u}[j] \mid j \in [\![q]\!]\}\,,\\
\{Y_{2,k}\}_{k=1}^{N(q,2)} &:= \{\mathbf{u} \mapsto \mathbf{u}[j]\mathbf{u}[j'] \mid 1 \le j < j' \le q\} \cup \left\{\mathbf{u} \mapsto (\mathbf{u}[j])^2 - \frac{1}{q} \mid j \in [\![2,q]\!]\right\},
\end{aligned} \tag{5.10}$$

it is possible to show (see Appendix A.2.3) that the squared MMD (5.6) can be written as

$$\mathrm{MMD}^2(\mathbb{Q},\mathbb{U}) = a_1 \left\|\mathbf{M}_1\mathbb{E}_{\mathbf{z}\sim\mathbb{Q}}[\Phi_1(\mathbf{z})]\right\|_2^2 + a_2 \left\|\mathbf{M}_2\mathbb{E}_{\mathbf{z}\sim\mathbb{Q}}[\Phi_2(\mathbf{z})]\right\|_2^2, \tag{5.11}$$

where $a_\ell := b_\ell|\mathcal{S}^{q-1}|/N(q,\ell)$ for $\ell \in \{1,2\}$, and $\mathbf{M}_1$, $\mathbf{M}_2$ are two lower triangular matrices with nonzero diagonal entries. Since $b_1, b_2 > 0$, $\mathrm{MMD}^2(\mathbb{Q},\mathbb{U}) = 0$ implies that

$$\mathbb{E}_{\mathbf{z}\sim\mathbb{Q}}[\mathbf{z}] = 0, \quad \mathbb{E}_{\mathbf{z}\sim\mathbb{Q}}[\mathbf{z}\mathbf{z}^\top] = \frac{1}{q}\mathbf{I}_q,$$

meaning that

$$\mathbb{E}[(\mathbf{z}-\mathbb{E}[\mathbf{z}])(\mathbf{z}-\mathbb{E}[\mathbf{z}])^\top] = \mathbb{E}[\mathbf{z}\mathbf{z}^\top] - \mathbb{E}[\mathbf{z}]\mathbb{E}[\mathbf{z}]^\top = \frac{1}{q}\mathbf{I}_q.$$

□

Complexity. The memory and computational complexities for computing the uniformity term (5.8) in SFRIK are $\mathcal{O}(|I|^2)$ and $\mathcal{O}(q|I|^2)$ respectively, as opposed to $\mathcal{O}(q^2)$ and $\mathcal{O}(q^2|I|)$ for the variance and covariance terms (5.9) in VICReg. In the setting where SFRIK and VICReg work best, i.e., larger dimension q and smaller batch size $|I|$, SFRIK has the lowest time and memory complexities. This computational advantage is due to the kernel trick and it is illustrated in Section 5.4.

5.4 Experiments

We first demonstrate numerically that the regularization loss (5.8) of SFRIK outperforms existing alternatives, in a rigorous experimental setting with a subset of ImageNet-1k [78] for pretraining and a separate validation set for hyperparameter tuning. Then, we pretrain a ResNet-50 backbone [163] with SFRIK on the full ImageNet-1k dataset and show competitive results compared to the state of the art, with significant computational benefits during pretraining. In the interest of reproducible research, we provide our code and our pretrained ResNet-50 backbones with SFRIK on ImageNet-1k at `https://github.com/valeoai/sfrik` [387].

5.4.1 Experimental setting

The backbone f_θ is either ResNet-18 or ResNet-50, depending on the experiment. Following [375], the projection head g_w is a three-layer MLP made of two hidden layers with ReLU activation and batch normalization [182], and a linear output layer. Unless otherwise specified, the size (number of neurons) of the two hidden layers is the same as the one, denoted q, of the output layer and the default value is $q = 8192$. The augmentations used for transforming images into views are the same as the ones used in VICReg. The backbone and the projection head are trained with a LARS optimizer [370]. The weight decay is fixed at 10^{-6}. The learning rate scheduling starts with 10 warm-up epochs [137] with a linear increase from 0 to $initial\_lr = base\_lr * bs/256$, where $base\_lr$ is called the base learning rate [137] and bs is the batch size, followed by a cosine decay [253] with a final learning rate 1000 times smaller than $initial\_lr$. For pretraining, we consider a 20% subset of ImageNet-1k (denoted by IN20%), like in [124], and 100% of ImageNet-1k (denoted by IN100%). In IN20%, we keep all the 1000 classes but only 260 images per class. More details are detailed in Appendix A.3.

5.4.2 Results for ResNet-18 pretrained on IN20%

Many existing self-supervision methods are based on the Siamese architecture and have the same form of training loss $\lambda \ell_a(\mathbf{Z}_I{}^{(1)}, \mathbf{Z}_I{}^{(2)}) + \mu \ell_r(\mathbf{Z}_I{}^{(1)}, \mathbf{Z}_I{}^{(2)})$. This is the case of SimCLR, AUH and VICReg, for which Appendix A.1.4 gives the

expression of the regularization loss ℓ_r. For SFRIK, following (5.2), we have $\mu = 0.5$ and $\ell_r(\mathbf{Z}_I{}^{(1)}, \mathbf{Z}_I{}^{(2)}) = \ell_u(\mathbf{Z}_I{}^{(1)}) + \ell_u(\mathbf{Z}_I{}^{(2)})$ with ℓ_u given by (5.8).

Protocol. To isolate the impact of ℓ_r on the quality of the learned representations, we (re)implement all these four methods in the setting of Section 5.4.1, to get rid of the influence of other design choices, like image augmentations or projection head architecture. We fix the batch size at 2048, and tune the base learning rate and hyperparameters specific to each method's loss. We also compare different embedding dimension $q \in \{1024, 2048, 4096, 8192\}$. In order to perform an extensive hyperparameter tuning by grid search of each method for fair comparisons, we choose a smaller backbone and a reduced dataset for pretraining, i.e., we pretrain a ResNet-18 on IN20% for 100 epochs with all methods.

Number of hyperparameters. Note that in total SFRIK with $L = 2$ has as many hyperparameters to tune as AUH or VICReg, and SFRIK with $L = 3$ has a *single* additional hyperparameter.

Rigorous hyperparameter tuning. In contrast to the common practice in the literature where hyperparameters are directly selected on the *labeled* evaluation dataset, we choose to tune hyperparameters on a *separate labeled validation set* that consists of *another* 20% subset of the ImageNet-1k train set[2]. We select the hyperparameters that yield the highest top-1 accuracy obtained by weighted kNN-classification ($k = 20$) [358] on this validation set, and we finally report the evaluation results by linear probing on the usual ImageNet-1k validation set, which is never seen during hyperparameter tuning. Because the hyperparameter tuning with kNN-classification requires labels for the validation set, the method is not fully unsupervised, even though it does not require labels during pretraining when fixing some given hyperparameters. A protocol without requiring any annotation for hyperparameter tuning (e.g., in the spirit of [119]) can be envisioned in a future work.

Results for linear probing on IN20%. Table 5.2 shows that SFRIK at optimal truncation order $L = 3$ outperforms SimCLR, AUH and VICReg by at least 0.7 points at $q = 8192$, for linear probing trained on IN20% with labels. The gain in top-1 accuracy by linear probing between SFRIK at $L = 1$ and $L = 2$ is important, but is smaller between $L = 2$ and $L = 3$. This suggests that $L > 3$ is likely to marginally improve performance, while requiring more hyperparameter tuning, which is why we did not explore $L > 3$. We also remark that all methods benefit from an increase in embedding dimension q, including SimCLR which was originally introduced with a smaller dimension.

[2]See Appendix A.4.1 for the experimental results that we would obtain if we directly tune the hyperparameters on the labeled evaluation dataset. The obtained top-1 accuracy would be slightly overestimated, but the ranking between methods does not change.

Table 5.2: **Linear probing on IN20% (top-1 accuracy)** at different embedding dimensions q. All methods were pretrained on IN20% with a ResNet-18 for 100 epochs. We tuned all hyperparameters specific to each method and the learning rate. Symbol † indicates models that we retrained ourselves.

	SimCLR†	AUH†	VICReg†	SFRIK		
				$L=1$	$L=2$	$L=3$
$q=1024$	45.2	45.3	40.6	-	45.2	-
$q=2048$	45.8	45.9	44.0	-	45.9	-
$q=4096$	46.0	46.7	44.9	-	46.9	-
$q=8192$	46.1	46.8	46.0	27.7	47.0	**47.5**

Table 5.3: **Linear classification on Places205 and VOC2007 (accuracy and mean average precision).** All methods were pretrained on IN20% with a ResNet-18 for 100 epochs. We tuned all hyperparameters specific to each pretraining method and the learning rate. The symbol † indicates models that we retrained ourselves.

Method	Linear classification		
	Places205		VOC07
	Top-1	Top-5	mAP
VICReg† ($q=8192$)	41.6	71.7	73.3
AUH† ($q=8192$)	42.3	72.8	73.6
SFRIK ($L=3, q=8192$)	**42.7**	**72.9**	**74.1**

Results for transfer learning to Places205 and VOC2007. We also evaluate the quality of the representations learned on other downstream tasks, such as image classification on Places205 [394] (dataset with scene-centric images), or predicting the presence of a class in a given image in VOC2007 [103] (dataset with several objects in the image). Table 5.3 provides extra results for linear probing on Places205 [394] and linear SVM on VOC2007 [103] that further support our findings: SFRIK outperforms AUH while having the same pretraining complexity, and is fully competitive compared to VICReg with a reduced pretraining complexity.

Ablation on the kernel choice. Table 5.4 confirms empirically that a truncated kernel is better than the RBF[3] or the generalized distance kernel for the uniformity term (5.7). During tuning we observed that the truncated kernel performs well when the weights b_2, b_3 in (5.8) are larger than b_1, e.g., $(b_1, b_2, b_3) = (1, 40, 40)$ for $q = 8192$. This contrasts with the RBF and the generalized distance kernel for

[3]The performance gap between AUH and the RBF kernel is only due to the presence of the logarithm in AUH (cf. Example 1). Future work could clarify the role of this logarithm for regularization in self-supervision.

Table 5.4: **Impact of kernel choice in the uniformity term** (5.7). Linear probing on IN20% of ResNet-18 pretrained on IN20% for 100 epochs, at $q = 8192$.

Kernel	Top-1 acc.
RBF	41.3
Generalized distance	27.8
Truncated, $L = 3$, cf. (5.8)	**47.5**

which the weights b_ℓ decay polynomially with respect to ℓ (see Appendix A.2.2). This suggests that it is important to focus more on order 2, 3 than on order 1 in the Legendre expansion (5.5).

5.4.3 Results for ResNet-50 pretrained on IN100%

Protocol. We pretrain a ResNet-50 on IN100% with SFRIK under the setting of Section 5.4.1, with a batch size of 2048. We study the impact of a larger embedding dimension in SFRIK by considering a projection head with two hidden layers of size 8192, and an output layer of size $q \in \{8192, 16384, 32768\}$. Truncation order is either $L = 2$ or $L = 3$. For comparison, we also pretrain a ResNet-50 with VICReg under the same setting with $q = 8192$. Similarly to the original paper [15], the alignment, variance and covariance weights are respectively 25, 25, 1, and the base learning rate is 0.2 for VICReg. All pretrained backbones are evaluated by: linear probing on IN100%; linear classification on Places205 and VOC2007 in order to measure how the learned representations generalize to an unseen dataset; and semi-supervised learning with few labels of IN100% (backbones are fine-tuned for classification using 1% or 10% of labeled images).

Computational complexity. We show under this protocol that SFRIK's time and memory complexity during pretraining is significantly smaller than the one of VICReg for large dimensions. This allows us to scale SFRIK at dimension 16384 and even to 32768 for better results on downstream tasks.[4] We measure the peak memory per GPU during pretraining on IN100% with a batch size of 2048 and the pretraining wall time of both methods on a 8× AMD Radeon Instinct MI50 32GB:

- at $q = 8192$, SFRIK is 8% faster than VICReg and needs 3% less memory per GPU;
- at $q = 16384$, SFRIK is 19% faster than VICReg and needs 8% less memory per GPU;

[4]We recall that the time and memory complexity is identical for all methods on downstream tasks.

Table 5.5: **Linear classification on IN100%, Places205, VOC2007, and semi-supervised learning with few labels of IN100% (top-k accuracy or mean average precision).** Methods are pretrained on IN100% with ResNet-50. We only include methods relying on a Siamese architecture with image augmentations limited to two views (obtained by applying transformations randomly sampled in a set $\mathcal{T}$). The scores of methods marked with * are from [55]. The score of VICReg† was obtained by retraining the model ourselves. For each downstream task, we highlight in bold the best score among all backbones pretrained on 200 epochs.

Method	Epochs	Linear classification					Semi-supervised			
		IN100%		Places205		VOC07	1% labels		10% labels	
		Top-1	Top-5	Top-1	Top-5	mAP	Top-1	Top-5	Top-1	Top-5
SimCLR* [50]	200	68.3	-	-	-	-	-	-	-	-
SwAV* [44] (no multi-crop)	200	69.1	-	-	-	-	-	-	-	-
SimSiam [55]	200	70.0	-	-	-	-	-	-	-	-
VICReg† [15] ($q = 8192$)	200	70.0	89.3	54.1	83.4	84.9	**49.4**	**75.1**	65.9	87.2
SFRIK ($L = 2, q = 8192$)	200	70.1	89.3	53.8	83.0	85.1	46.6	73.3	65.7	87.3
SFRIK ($L = 3, q = 8192$)	200	70.2	89.6	**54.5**	**83.9**	84.6	46.9	73.6	**66.0**	**87.7**
SFRIK ($L = 2, q = 16384$)	200	**70.3**	89.6	54.3	83.4	**85.2**	46.0	73.0	65.3	87.2
SFRIK ($L = 2, q = 32768$)	200	**70.3**	89.6	54.1	83.0	85.0	46.1	73.0	65.4	87.3
SFRIK ($L = 3, q = 32768$)	200	**70.3**	**89.7**	54.4	83.2	85.1	46.6	73.0	65.8	87.5
SFRIK ($L = 2, q = 8192$)	400	70.8	89.9	54.4	83.5	85.7	47.8	74.3	66.4	88.0

Table 5.6: **Peak memory per GPU** during pretraining of ResNet-50 on IN100% at embedding dimension $q = 32768$.

Batch size	VICReg	SFRIK	(ratio)
256	22.5GB	10.3GB	(2.2)
512	25.4GB	13.1GB	(1.9)
1024	31.1GB	18.8GB	(1.7)

- at $q = 32768$, SFRIK is still 2% faster than VICReg *run in the lower dimension 16384*. It only requires 30.9GB per GPU while VICReg at $q = 32768$ needs more than the available memory. Table 5.6 emphasizes this memory advantage at reduced batch sizes.

Results. Table 5.5 compares methods that have the same Siamese architecture and use the same image augmentations described in our protocol. For completeness, this table is completed in Appendix A.4.2 by evaluation results of other existing methods such as BYOL [143], OBoW [123] and SwAV with multi-crop [44], which are not comparable to the methods of Table 5.5 as they use a teacher-student architecture with momentum encoder and/or image augmentations with multi-scale cropping, and are beyond the setting of a Siamese architecture with only two views. Incorporating such designs in SFRIK is possible, and is left as a future work.

Table 5.5 demonstrates the competitiveness of SFRIK: it has the best accuracy

for linear probing on IN100% among SimCLR, SwAV with no multi-crop, SimSiam and VICReg, and it performs better than VICReg for linear classification on Places205, VOC2007, and semi-supervised-learning with 10% of labels. We observe that SFRIK and VICReg offer a different trade-off between performance on linear probing on IN100% and performance on semi-supervised learning with 1% of labels. But as shown in Appendix A.4.2, other methods like BYOL and SwAV with multi-crop similarly have a performance drop compared to VICReg on semi-supervised learning with 1% of labels, even though they perform better on linear probing. Future work will therefore involve understanding what specific ingredients of VICReg make it more robust for semi-supervised learning with few labels. Ideally we could combine these ingredients with our generic kernel framework to design an improved version of SFRIK that can still benefit from its computational advantages over VICReg.

5.5 Conclusion

We proposed a *regularization loss* family based on the MMD and rotation-invariant kernels. We demonstrated that several regularizers of former methods are indeed variants of our flexible loss with different kernels. This generic regularization approach allowed us to leverage degrees of freedom in rotation-invariant kernel design to improve self-supervision methods. In practice, using a truncated kernel, we derived from the proposed framework a fully competitive self-supervised pretraining method, SFRIK, which significantly reduces time and memory complexity during pretraining compared to information-maximization methods. We discuss several perspectives of this work.

Extension to unnormalized embeddings. The proposed generic regularization loss is based on rotation-invariant kernels defined on the hypersphere. One can wonder how to extend this framework to kernels defined in Euclidean space, so that it can cover self-supervised learning methods that do not normalize the embeddings, such as VICReg.

Kernel approximation. In the scenario where it is required to scale the batch size (e.g., up to 32768) for improving the performance of a self-supervised learning method, computing the pretraining loss function becomes costly if its complexity is quadratic with respect to the batch size. In such a scenario, one promising perspective is to combine the proposed kernel approach for self-supervision with kernel approximation techniques such as Nyström method or random feature expansions [256, 302, 308], in order to further enhance the ability to perform self-supervised training with limited computational resources.

Insights on the regularization in self-supervision. The question of choosing an adequate regularizer that avoids collapse [190] during self-supervision is still an

open question in the literature. This work unifies several previous self-supervision methods under the proposed generic kernel loss for regularization. Therefore one possible research direction is to design a good regularizer for self-supervision by identifying a good kernel choice under this kernel framework. A theoretical analysis could start with a simple backbone model (linear layer, multi-layer perceptron). In other words, one promising perspective is to study self-supervision under this new unified kernel point of view, in order to provide novel insights about the behavior of self-supervision algorithms.

Next chapters of the thesis. As the size of the dataset increases [291], further reducing the computational cost of pretraining requires a reduction of the complexity for the forward pass of the backbone. As a complementary research direction to self-supervised learning, the next chapters of the thesis focus on the reduction of the complexity for matrix multiplication, which is the main operation in neural networks.

Chapter 6

Square dyadic butterfly factorization: identifiability and decomposition algorithm

One of the contributions of this thesis is to provide *theoretical guarantees* to decomposition algorithms associated with butterfly factorization, i.e., the problem of approximating a certain matrix by a product of butterfly factors that satisfy a specific sparsity constraint. In order to better illustrate the nature of our results, this chapter starts by studying the specific of *square dyadic butterfly factorization*. In particular, we will focus on the *noiseless setting*, where the target matrix is assumed to admit such an exact butterfly factorization. The results of this chapter will be then generalized in Chapter 7 to more general butterfly factorization, and to the noisy setting.

6.1 Introduction

The *generic sparse matrix factorization* problem with $L \geq 2$ factors is formulated as follows: given a target matrix $\mathbf{A}$, solve

$$\min_{\mathbf{X}_1,\dots,\mathbf{X}_L} \|\mathbf{A} - \mathbf{X}_1\mathbf{X}_2\dots\mathbf{X}_L\|_F, \quad \text{such that } \mathbf{X}_\ell \text{ is sparse for each } \ell \in [\![L]\!]. \tag{6.1}$$

This problem is mainly motivated by the need for a rapid evaluation of the matrix multiplication by $\mathbf{A}$, typically in a large-scale setting where the direct computation of the matrix multiplication hardly scales due to its quadratic complexity with respect to the matrix size. There are typically two kinds of sparsity constraints:

1. **Classical sparsity constraints**: they are encoded by a *family of allowed supports* that force the factors to have some prescribed sparsity patterns, like

The material of this chapter is based on [213, 392], in collaboration with Quoc-Tung Le, Elisa Riccietti and Rémi Gribonval.

k-sparsity by row and/or column (i.e., each factor has at most k nonzero entries per row and/or column).

2. **Fixed-support constraints**: the supports (i.e., the set of indices corresponding to nonzero entries in a matrix) are known a priori and each factor $\mathbf{X}_\ell$ is constrained to have a support included in a given prescribed support $\mathbf{S}_\ell$, $\ell = 1, \dots, L$. Note that classical sparsity constraints mentioned above can be written as a finite union of such fixed-supports constraints.

Sparse matrix factorization is difficult in general. Problem (6.1) is known to be difficult in general, but even specific instances have been shown to be NP-hard. On the one hand, Problem (6.1) with classical sparsity constraints and $L = 2$ factors is the sparse coding problem [93, 108], when the dictionary $\mathbf{X}_1$ is known and the factor $\mathbf{X}_2$ is constrained to be k-sparse by column. This specific problem is shown to be NP-hard in [108, Theorem 2.17]. On the other hand, even in the case with $L = 2$ factors and fixed-support constraints, Problem (6.1) has been recently shown to be NP-hard, without further assumptions on the prescribed fixed supports [212].

Decomposition via square dyadic butterfly factorization. This chapter shows that the particular choice of fixed-support constraint corresponding to the *square dyadic butterfly factorization* makes Problem (6.1) both well-posed and tractable, in the sense that Problem (6.1) in the noiseless setting admits a unique[1] solution that can be computed with an algorithm of polynomial complexity. Let us recall the definition of these fixed-support constraints.

Definition 6.1 (Square dyadic butterfly supports)**.** *The* square dyadic butterfly supports *of size* $n = 2^L$ *are* $\mathsf{S}_\beta := (\mathbf{S}_{\pi_1}, \dots, \mathbf{S}_{\pi_L}) \in (\{0,1\}^{n \times n})^L$ *defined by:*

$$\mathbf{S}_{\pi_\ell} := \mathbf{I}_{2^{\ell-1}} \otimes \mathbf{1}_{2\times 2} \otimes \mathbf{I}_{2^{L-\ell}},\ 1 \leq \ell \leq L. \tag{6.2}$$

Definition 6.2 (Square dyadic butterfly matrix)**.** *A matrix* $\mathbf{A}$ *of size* $n = 2^L$ *is a* square dyadic butterfly matrix *if it can be factorized exactly into* L *factors* $(\mathbf{X}_1, \dots, \mathbf{X}_L)$ *that have a support included in the square dyadic butterfly supports* $\mathsf{S}_\beta := (\mathbf{S}_{\pi_1}, \dots, \mathbf{S}_{\pi_L})$*, in the sense that:*

$$\mathbf{A} = \mathbf{X}_1 \dots \mathbf{X}_L, \quad \textit{with} \quad (\mathbf{X}_1, \dots, \mathbf{X}_L) \in \Sigma^{\pi_1} \times \dots \times \Sigma^{\pi_L}.$$

For this chapter we define:

$$\Sigma^\beta := \Sigma^{\pi_1} \times \dots \times \Sigma^{\pi_L}, \quad \textit{with} \quad \Sigma^{\pi_\ell} := \Sigma^{\mathbf{S}_{\pi_\ell}} \quad \forall \ell \in [\![L]\!]. \tag{6.3}$$

[1] up to unavoidable scaling ambiguities.

Remark 6.1. *The notations β, $\boldsymbol{\pi}$, $\Sigma^{\boldsymbol{\pi}}$, Σ^{β} will be used in Chapter 7 in a more general context (cf. Definitions 7.2 and 7.3).*

As illustrated in Figure 1.1 for the case with $L = 4$ factors, this fixed-support constraint enforces the factors $\mathbf{X}_\ell \in \mathbb{C}^{n \times n}$ ($n = 2^L$, $\ell \in [\![L]\!]$) to have at most two nonzero entries per row and per column, with a *structured* sparsity pattern, in the sense that in the sense that their support can be written as a Kronecker product. Note that, up to some proper row and column permutations, each factor is block-diagonal with 2×2 dense submatrices.

Motivation for square dyadic butterfly factorization. As detailed in Section 4.5, the square dyadic butterfly structure is interesting for machine learning and signal processing applications mainly for two reasons. First, finding the butterfly factors $\mathbf{X}_\ell$ ($\ell \in [\![L]\!]$) from the product $\mathbf{A} := \mathbf{X}_1 \dots \mathbf{X}_L$ enables *fast* $\mathcal{O}(n \log n)$ matrix-vector multiplication by $\mathbf{A}$. Indeed, there are $L = \log_2(n)$ butterfly factors and the complexity of matrix-vector multiplication by each butterfly factor $\mathbf{X}_\ell$ ($\ell \in [\![L]\!]$) is $\mathcal{O}(n)$, since $\mathbf{X}_\ell$ has at most $2n$ nonzero entries. Second, the square dyadic butterfly factorization can be used to construct a generic expressive representation for structured linear maps, in the sense that the matrices associated with many structured matrices (such as the DFT or the Hadamard matrix) admit exactly such a factorization.

Limits of first-order optimization methods. Previous methods [74] to address Problem (6.1) with the constraints $\mathbf{X}_\ell \in \Sigma^{\boldsymbol{\pi}_\ell}$ for each $\ell \in [\![L]\!]$ are based on first-order optimization. However, since this problem is not convex, the previous methods lack theoretical guarantees for finding the optimal solution, and their performance is heavily dependent on initialization and hyperparameter tuning.

Contributions. This chapter shows that every tuple of factors $(\mathbf{X}_1, \dots, \mathbf{X}_L)$ satisfying the square dyadic butterfly constraint can be reconstructed *with guarantee* by a *polynomial* algorithm from $\mathbf{A} := \mathbf{X}_1 \dots \mathbf{X}_L$ (see Algorithm 6.1). This algorithm is associated with our *identifiability* results (see Theorem 6.1), which claims that the factorization $\mathbf{A} := \mathbf{X}_1 \dots \mathbf{X}_L$ for any $\mathbf{X}_1, \dots, \mathbf{X}_L$ satisfying the square dyadic butterfly constraint is *essentially unique* up to natural scaling ambiguities. The algorithm is inspired from previous butterfly algorithms [40, 266, 289], and is based on a hierarchical approach [216] in which the target matrix is iteratively factorized into two factors, until the desired number of sparse factors L is obtained. These successive two-layer factorizations rely on a non-trivial application of the singular value decomposition (SVD) to compute *best rank-one approximations* of specific submatrices, instead of iterative gradient descent steps. The total complexity of the hierarchical algorithm is $\mathcal{O}(n^2)$ where n is the size of $\mathbf{A}$. This is of the same order of magnitude as the complexity of matrix-vector multiplication, and needs to be performed only once to enable $\mathcal{O}(n \log n)$ matrix-vector multiplications with the resulting factored representation of $\mathbf{A}$.

Novelty compared to previous butterfly algorithms. The proposed hierarchical algorithm (Algorithm 6.1) is parameterized by a flexible so-called *factor-bracketing binary tree* (cf. Definition 6.4 below) somehow describing the bracketing of factors associated with their product. This tree defines the order in which the successive two-layers matrix factorizations are performed in the hierarchical approach. Previous butterfly algorithms [250] only consider some specific choices factor-bracketing tree to approximate the target matrix, while the proposed hierarchical algorithm works for *any* factor-bracketing tree for the square dyadic butterfly factorization. This allows us to identify a new balanced factor-bracketing tree that has never been used previously, and has the benefit of allowing a parallelization of the two-layer matrix factorization steps in the hierarchical approach for further acceleration of the hierarchical algorithm.

Summary. The main contributions of this chapter are the following ones:

1. We show in Theorem 6.1 that enforcing the *square dyadic butterfly* structure on the L sparse factors is sufficient to ensure a *hierarchical identifiability* property, meaning that we can recover (up to natural scaling ambiguities) sparse factors $(\mathbf{X}_\ell)_{\ell=1}^L$ from $\mathbf{A} := \mathbf{X}_1 \dots \mathbf{X}_L$, using a *new* hierarchical factorization algorithm described in Algorithm 6.1.

2. We introduce the notion of factor-bracketing bracketing tree and shows that the new hierarchical algorithm can recover successfully the butterfly factors using *any* hierarchical order.

3. We show that this algorithm has a time complexity of only $\mathcal{O}(n^2)$ where n is the size of $\mathbf{A}$, which is of the order of a few dense matrix-vector multiplications, while the obtained factorizations enable fast $\mathcal{O}(n \log n)$ matrix-vector multiplications.

4. We illustrate this complexity by implementing[2] Algorithm 6.1 using *truncated* SVDs. We show numerically that the hierarchical algorithm outperforms gradient-based methods for square dyadic butterfly factorization.

Outline. Section 6.2 analyzes identifiability in the two-layer setting with fixed-support constraint. Section 6.3 describes how the butterfly structure can ensure the hierarchical identifiability of the sparse factors from their product. Section 6.4 discusses related work, and clarifies the difference of the new proposed hierarchical algorithm with previous butterfly algorithms. Section 6.5 presents some numerical experiments about the recovery of the sparse butterfly factors from their product; Section 6.6 discusses perspectives of this work. Appendix B gathers technical proofs.

[2]Implementation available in the FAµST 3.25 toolbox (https://faust.inria.fr/).

6.2 Identifiability in two-layer sparse matrix factorization

In order to show hierarchical identifiability results, we first analyze identifiability in *two-layer sparse matrix factorization*. Given a matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$, and a subset of pairs of factors $\Sigma \subseteq \mathbb{C}^{m\times k} \times \mathbb{C}^{n\times k}$, the so-called *exact matrix factorization* problem with two factors of $\mathbf{A}$ in Σ is:

$$\text{find if possible } (\mathbf{X}, \mathbf{Y}) \in \Sigma \text{ such that } \mathbf{A} = \mathbf{X}\mathbf{Y}^\top. \tag{6.4}$$

Uniqueness properties are studied in the exact setting, hence for the rest of the section, we assume that $\mathbf{A}$ admits such an exact factorization.

We are interested in the particular problem variation where the constraint set Σ encodes some chosen sparsity patterns for the factorization. For a given binary matrix $\mathbf{S} \in \{0,1\}^{m\times k}$ associated with a *sparsity pattern*, denote

$$\Sigma^{\mathbf{S}} := \{\mathbf{M} \in \mathbb{C}^{m\times k} \mid \operatorname{supp}(\mathbf{M}) \subseteq \operatorname{supp}(\mathbf{S})\}, \tag{6.5}$$

which is the set of matrices with a support included in $\mathbf{S}$.

Remark 6.2. *By abuse of notation, throughout the thesis, the fixed-support constraint* $\operatorname{supp}(\mathbf{M}) \subseteq \operatorname{supp}(\mathbf{S})$ *will sometimes simply be written as* $\operatorname{supp}(\mathbf{M}) \subseteq \mathbf{S}$.

A pair of sparsity patterns is written $\mathrm{S} := (\mathbf{S}_{\text{left}}, \mathbf{S}_{\text{right}})$, where $\mathbf{S}_{\text{left}}$ and $\mathbf{S}_{\text{right}}$ are the left and right sparsity patterns respectively, also referred to as a left and a right (allowed) support. Given any pair of allowed supports represented by binary matrices $\mathrm{S} := (\mathbf{S}_{\text{left}}, \mathbf{S}_{\text{right}}) \in \{0,1\}^{m\times k} \times \{0,1\}^{n\times k}$, the set

$$\Sigma^{\mathrm{S}} := \Sigma^{\mathbf{S}_{\text{left}}} \times \Sigma^{\mathbf{S}_{\text{right}}} \subseteq \mathbb{C}^{m\times k} \times \mathbb{C}^{n\times k} \tag{6.6}$$

is a linear subspace. Since the support of a matrix is unchanged under arbitrary rescaling of its columns, Σ^{S} is invariant by column scaling for any pair of supports S. Uniqueness of a solution to (6.4) with such sparsity constraints will always be considered up to unavoidable scaling ambiguities. Using the terminology from the tensor decomposition literature [204], such a uniqueness property will be referred to as *essential uniqueness*.

Definition 6.3 (Essential uniqueness of a two-layer factorization in Σ)**.** *Let* Σ *be a set of pairs of factors, and* $\mathbf{A}$ *be a matrix admitting a factorization* $\mathbf{A} := \mathbf{X}\mathbf{Y}^\top$ *such that* $(\mathbf{X}, \mathbf{Y}) \in \Sigma$*. This factorization is* essentially unique *in* Σ*, if any solution* $(\bar{\mathbf{X}}, \bar{\mathbf{Y}})$ *to* (6.4) *with* $\mathbf{A}$ *and* Σ *is equivalent to* $(\mathbf{X}, \mathbf{Y})$*, written* $(\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \sim (\mathbf{X}, \mathbf{Y})$*, in the sense that there is an invertible diagonal matrix* $\mathbf{D}$ *such that* $(\bar{\mathbf{X}}, \bar{\mathbf{Y}}) = (\mathbf{X}\mathbf{D}, \mathbf{Y}\mathbf{D}^{-1})$.

Remark 6.3. *Another definition of essential uniqueness [204] involves permutation ambiguity in addition to scaling ambiguity. Nevertheless, we only consider scaling ambiguity in our definition of essential uniqueness, because all the fixed-support constraints* Σ_{S} *considered in this chapter remove this permutation ambiguity, as detailed in Remark 6.5 below.*

For any set Σ of pairs of factors, the set of all pairs $(\mathbf{X}, \mathbf{Y}) \in \Sigma$ such that the factorization $\mathbf{A} := \mathbf{X}\mathbf{Y}^\top$ is essentially unique in Σ is denoted $\mathcal{U}(\Sigma)$. In other words, we define:

$$\mathcal{U}(\Sigma) := \left\{ (\mathbf{X}, \mathbf{Y}) \in \Sigma \mid \forall (\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \in \Sigma,\ \bar{\mathbf{X}}\bar{\mathbf{Y}}^\top = \mathbf{X}\mathbf{Y}^\top \implies (\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \sim (\mathbf{X}, \mathbf{Y}) \right\}. \quad (6.7)$$

To characterize $\mathcal{U}(\Sigma^{\mathrm{S}})$ for Σ^{S} defined as in (6.6) with S any pair of supports, we first establish a non-degeneration property, i.e., a necessary condition for identifiability, involving the so-called *column support*. The column support, denoted $\mathrm{colsupp}(M)$, is the subset of indices $i \in [\![n]\!]$ such that the i-th column of M is nonzero. Define the set of pairs of factors with *identical*, resp. *maximal*, column supports in Σ^{S} as

$$\mathtt{IC}^{\mathrm{S}} := \{(\mathbf{X}, \mathbf{Y}) \in \Sigma^{\mathrm{S}} \mid \mathrm{colsupp}(\mathbf{X}) = \mathrm{colsupp}(\mathbf{Y})\}, \quad (6.8)$$

$$\mathtt{MC}^{\mathrm{S}} := \{(\mathbf{X}, \mathbf{Y}) \in \Sigma^{\mathrm{S}} \mid \mathrm{colsupp}(\mathbf{X}) = \mathrm{colsupp}(\mathbf{S}_{\mathrm{left}}) \text{ and } \mathrm{colsupp}(\mathbf{Y}) = \mathrm{colsupp}(\mathbf{S}_{\mathrm{right}})\}. \quad (6.9)$$

Lemma 6.1. *For any pair of supports* S*, we have:* $\mathcal{U}(\Sigma^{\mathrm{S}}) \subseteq \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}}$.

Remark 6.4. *If* $\mathrm{colsupp}(\mathbf{S}_{left}) \neq \mathrm{colsupp}(\mathbf{S}_{right})$*, then the set* $\mathcal{U}(\Sigma^{\mathrm{S}})$ *is empty.*

In other words, if the factorization $\mathbf{A} := \mathbf{X}\mathbf{Y}^\top$ is essentially unique in Σ^{S}, then the left and right supports must have the same column support, and $\mathbf{X}$, $\mathbf{Y}$ do not have a zero column inside this column support. The proof is deferred to Appendix B.1.

As the product $\mathbf{X}\mathbf{Y}^\top$ is the sum of rank-one matrices $\sum_{i=1}^{k} \mathbf{X}[:, i]\mathbf{Y}[:, i]^\top$, the *lifting* procedure [61, 212, 258] suggests to represent the pair $(\mathbf{X}, \mathbf{Y})$ by its k-tuple of so-called *rank-one contributions*

$$\varphi(\mathbf{X}, \mathbf{Y}) := \left(\mathbf{X}[:, i]\mathbf{Y}[:, i]^\top\right)_{i=1}^{k} \in (\mathbb{C}^{m \times n})^k. \quad (6.10)$$

Indeed, one can identify, up to scaling ambiguities, the columns $\mathbf{X}[:, i]$, $\mathbf{Y}[:, i]$ from their outer product $\mathcal{C}_i := \mathbf{X}[:, i]\mathbf{Y}[:, i]^\top$ $(1 \leq i \leq k)$, *as long as* the rank-one contribution $\mathcal{C}_i$ is not zero.

Lemma 6.2 (Reformulation of [214, Chapter 7, Lemma 1])**.** *Consider* $\mathcal{C}$ *the outer product of two vectors* $\mathbf{a}$, $\mathbf{b}$*. If* $\mathcal{C} = \mathbf{0}$*, then* $\mathbf{a} = \mathbf{0}$ *or* $\mathbf{b} = \mathbf{0}$*. If* $\mathcal{C} \neq \mathbf{0}$*, then* $\mathbf{a}$, $\mathbf{b}$ *are nonzero, and for any* $(\mathbf{a}', \mathbf{b}')$ *such that* $\mathbf{a}'\mathbf{b}'^\top = \mathcal{C}$*, there exists a scalar* $\lambda \neq 0$ *such that* $\mathbf{a}' = \lambda\mathbf{a}$ *and* $\mathbf{b}' = \frac{1}{\lambda}\mathbf{b}$.

With this lifting approach, each support constraint $\mathrm{S} = (\mathbf{S}_{\mathrm{left}}, \mathbf{S}_{\mathrm{right}})$ is represented by the k-tuple of rank-one support constraints $\mathcal{S} = \varphi(\mathbf{S}_{\mathrm{left}}, \mathbf{S}_{\mathrm{right}}) =$

$(\boldsymbol{\mathcal{S}}_i)_{i=1}^k$. Thus, if $(\mathbf{X}, \mathbf{Y}) \in \Sigma^{\mathrm{S}} \subseteq \mathbb{C}^{m\times k} \times \mathbb{C}^{n\times k}$, then the k-tuple of rank-one matrices $\varphi(\mathbf{X}, \mathbf{Y}) \in (\mathbb{C}^{m\times n})^k$ belongs to the set:

$$\Gamma^{\mathcal{S}} := \left\{ (\boldsymbol{\mathcal{C}}_i)_{i=1}^k \mid \forall i \in [\![k]\!],\ \mathrm{rank}(\boldsymbol{\mathcal{C}}_i) \leq 1,\ \mathrm{supp}(\boldsymbol{\mathcal{C}}_i) \subseteq \boldsymbol{\mathcal{S}}_i \right\} \subseteq (\mathbb{C}^{m\times n})^k. \quad (6.11)$$

As explained in the general framework of [61] established to analyze identifiability in bilinear inverse problems, the main advantage of this approach is to remove the inherent scaling ambiguity, while preserving a one-to-one correspondence between a pair of factors $(\mathbf{X}, \mathbf{Y})$ and its rank-one contributions representation $\varphi(\mathbf{X}, \mathbf{Y})$. Our work specializes this general framework to matrix factorization problem with two factors and support constraints. Details about the lifting procedure for fixed-support two-layer matrix factorization are in Appendix B.2.

From this lifting procedure, we show that it is possible to derive a simple necessary and sufficient condition for identifiability. Despite its simplicity, this condition is the key to derive identifiability results in square dyadic butterfly factorization using a hierarchical approach (cf. Lemma 6.5 below). The proof is deferred to Appendix B.3. More conditions on fixed-support identifiability are given in [214, 390].

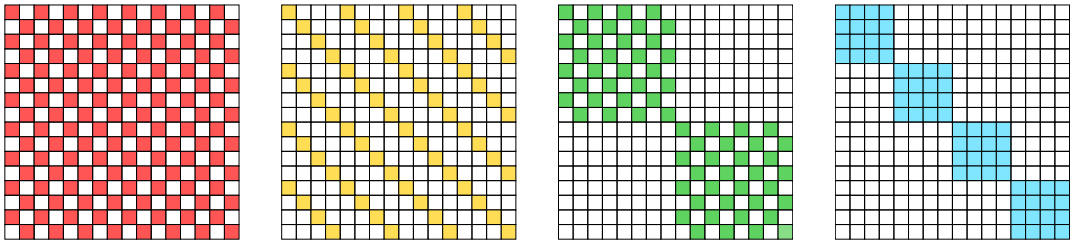
Proposition 6.1. *Assuming* $\mathrm{colsupp}(\mathbf{S}_{left}) = \mathrm{colsupp}(\mathbf{S}_{right})$ *(which is natural by Remark 6.4),* $\mathcal{U}(\Sigma^{\mathrm{S}}) = \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}}$ *if, and only if, the tuple* $\varphi(\mathrm{S}) = (\boldsymbol{\mathcal{S}}_i)_{i=1}^k$ *has disjoint rank-one supports, i.e.,* $\mathrm{supp}(\boldsymbol{\mathcal{S}}_i) \cap \mathrm{supp}(\boldsymbol{\mathcal{S}}_j) = \emptyset$ *for every* $i \neq j$.

Remark 6.5. *Following Remark 6.3, the assumption that* $\varphi(\mathrm{S})$ *has disjoint rank-one supports is sufficient to remove any potential permutation ambiguity in the factorization* $\mathbf{A} := \mathbf{X}\mathbf{Y}^\top$ *in* Σ *for any* $(\mathbf{X}, \mathbf{Y}) \in \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}}$*, in the sense that: for any* $(\mathbf{X}, \mathbf{Y}) \in \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}}$*, any invertible diagonal matrix* $\mathbf{D}$ *and any permutation matrix* $\mathbf{P}$ *such that* $(\mathbf{XDP}, \mathbf{YD}^{-1}\mathbf{P}) \in \Sigma^{\mathrm{S}}$*,* $\mathbf{P}$ *is necessarily the identity matrix. This is true because otherwise, denoting* $(\boldsymbol{\mathcal{S}}_i)_{i=1}^k := \varphi(\mathrm{S})$*,* $(\boldsymbol{\mathcal{C}}_i)_{i=1}^k := \varphi(\mathbf{X}, \mathbf{Y})$ *and noticing that* $\varphi(\mathbf{XDP}, \mathbf{YD}^{-1}\mathbf{P})$ *is equal to* $(\boldsymbol{\mathcal{C}}_i)_{i=1}^k$ *up to a permutation on the index* i*, there would exist two indices* $j \neq j'$ *such that* $\mathrm{supp}(\boldsymbol{\mathcal{C}}_j) \subseteq \mathrm{supp}(\boldsymbol{\mathcal{S}}_j)$ *and* $\mathrm{supp}(\boldsymbol{\mathcal{C}}_j) \subseteq \mathrm{supp}(\boldsymbol{\mathcal{S}}_{j'})$ *with* $\mathrm{supp}(\boldsymbol{\mathcal{C}}_j) \neq \emptyset$ *(because* $(\mathbf{X}, \mathbf{Y}) \in \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}}$*), which would contradict* $\mathrm{supp}(\boldsymbol{\mathcal{S}}_j) \cap \mathrm{supp}(\boldsymbol{\mathcal{S}}_{j'}) = \emptyset$.

6.3 Hierarchical identifiability of square dyadic butterfly factors

The main contribution of the chapter is to establish some identifiability results in the multi-layer sparse matrix factorization when the sparse factors are constrained to have the square dyadic butterfly supports (Definition 6.1). The so-called *square dyadic butterfly sparse matrix factorization* problem is the following special instance of (6.1):

$$\min_{\mathbf{X}_1,\ldots,\mathbf{X}_L} \|\mathbf{A} - \mathbf{X}_1 \ldots \mathbf{X}_L\|_F, \quad \text{such that } (\mathbf{X}_1, \ldots, \mathbf{X}_L) \in \Sigma^{\beta}, \quad (6.12)$$



(a) $\mathbf{S}_{\boldsymbol{\pi}_1 * \boldsymbol{\pi}_2 * \boldsymbol{\pi}_3}$ (b) $\mathbf{S}_{\boldsymbol{\pi}_1 * \boldsymbol{\pi}_2}$ (c) $\mathbf{S}_{\boldsymbol{\pi}_2 * \boldsymbol{\pi}_3}$ (d) $\mathbf{S}_{\boldsymbol{\pi}_3 * \boldsymbol{\pi}_4}$

Figure 6.1: Examples of supports $\mathbf{S}_{\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_t}$ defined by (6.14) ($1 \leq r \leq t \leq 4$) of size $n \times n$ with $n = 16$. Nonzero entries are in color, and zero entries are in white.

with Σ^β defined as in (6.3).

Remark 6.6. *As shown in [212, Remark A.1], there exists a support constraint* $\mathrm{S} = (\mathbf{S}_{left}, \mathbf{S}_{right})$ *and a matrix* $\mathbf{A}$ *such that: (a)* $\mathbf{A}$ cannot be written exactly *as* $\mathbf{A} = \mathbf{X}\mathbf{Y}^\top$ *for any* $(\mathbf{X}, \mathbf{Y}) \in \Sigma^{\mathrm{S}}$*; (b)* $\mathbf{A}$ *can be approximated arbitrarily well by such a product, i.e.,* $0 = \inf_{(\mathbf{X},\mathbf{Y}) \in \Sigma^{\mathrm{S}}} \|\mathbf{A} - \mathbf{X}\mathbf{Y}^\top\|_F$*. This corresponds to a lack of closure of the set* $\{\mathbf{X}\mathbf{Y}^\top : (\mathbf{X}, \mathbf{Y}) \in \Sigma^{\mathrm{S}}\}$ *for general* S*. Fortunately this pathological behavior does not happen here because of the specific choice of the support constraint* Σ^β *corresponding to the square dyadic butterfly factorization: we will show that an optimum for Problem (6.12) always exists for any target matrix* $\mathbf{A}$*, as it will be discussed in Chapter 7 with more details (cf. Corollary 7.3).*

Due to the structure of the sparsity patterns in the square dyadic butterfly factorization, we can show that, for any $(\mathbf{X}_1, \ldots, \mathbf{X}_L) \in \Sigma^\beta$, the partial product of any consecutive factors $\mathbf{X}_r \ldots \mathbf{X}_t$ ($1 \leq r \leq t \leq L$) have a very precise structure encoded by the support $\mathbf{S}_{\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_t}$ as detailed in the following lemma.

Lemma 6.3. *Let* $\mathrm{S}_\beta := (\mathbf{S}_{\boldsymbol{\pi}_1}, \ldots, \mathbf{S}_{\boldsymbol{\pi}_L})$ *be the square dyadic butterfly supports of size* $n = 2^L$*. Then, for* $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$*, for any* $1 \leq r \leq t \leq L$*:*

$$\operatorname{supp}(\mathbf{X}_r \ldots \mathbf{X}_t) \subseteq \operatorname{supp}(\mathbf{S}_{\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_t}), \tag{6.13}$$

$$\text{where} \quad \mathbf{S}_{\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_t} := \mathbf{I}_{2^{r-1}} \otimes \mathbf{1}_{2^{t-r+1} \times 2^{t-r+1}} \otimes \mathbf{I}_{2^{L-t}} \in \{0,1\}^{n \times n}. \tag{6.14}$$

Remark 6.7. *The proof of the lemma is deferred to Chapter 7, because it is a special case of a more general result (cf. Proposition 7.1). The notation* $*$ *will be used for a precise meaning in this more general context. Intuitively, viewing matrix supports as binary matrices, one can verify that* $\mathbf{S}_{\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_t} = \mathbf{S}_{\boldsymbol{\pi}_r} \ldots \mathbf{S}_{\boldsymbol{\pi}_t}$ *for* $1 \leq r \leq t \leq L$*. Figure 6.1 illustrates this structure on some examples of size* $n = 16$*.*

Importantly, this lemma allows for a hierarchical approach to address Problem (6.12), as we now explain.

6.3.1 Hierarchical matrix factorization method

Assume that the target matrix $\mathbf{A}$ in Problem (6.12) admits exactly a square dyadic butterfly factorization, i.e., $\mathbf{A} = \mathbf{X}_1 \dots \mathbf{X}_L$ for some $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$. The goal is to *recover*, up to some unavoidable scaling ambiguities, each butterfly factor $\mathbf{X}_\ell$ for $\ell \in [\![L]\!]$, only from the observation of $\mathbf{A}$. We will show that it is possible via a *hierarchical approach* [216]: instead of directly optimizing Problem (6.12) over the L factors, the hierarchical matrix factorization method is a heuristic approach that performs *successive two-layer matrix factorizations*, until L sparse factors are obtained. Let us illustrate this approach in the following inductive procedure.

Step $\ell = 1$: Recover (up to scaling ambiguities) the factors $(\mathbf{X}_1, \mathbf{X}_2 \dots \mathbf{X}_L)$ by solving the minimization problem

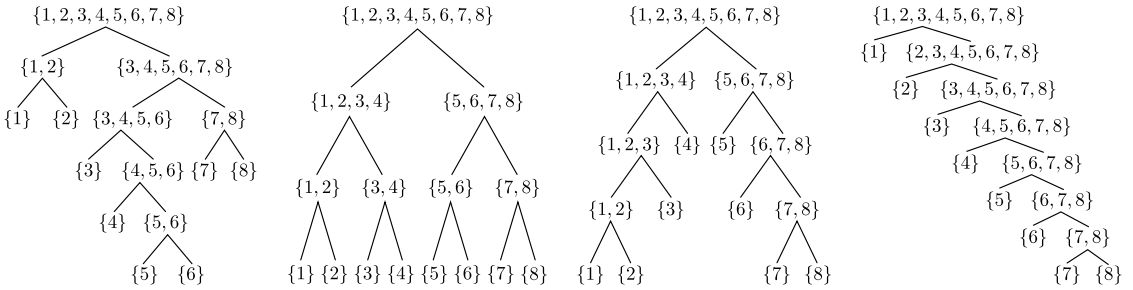
$$\min_{\mathbf{X},\mathbf{Y}} \|\mathbf{A} - \mathbf{X}\mathbf{Y}\|_F \quad \text{such that} \quad \begin{cases} \operatorname{supp}(\mathbf{X}) \subseteq \mathbf{S}_{\boldsymbol{\pi}_1} \\ \operatorname{supp}(\mathbf{Y}) \subseteq \mathbf{S}_{\boldsymbol{\pi}_2 * \dots * \boldsymbol{\pi}_L} \end{cases} . \tag{6.15}$$

Step $\ell \in \{2, \dots, L-1\}$: Assuming that we have recovered (up to scaling ambiguities) the partial product $\mathbf{X}_\ell \dots \mathbf{X}_L$ from the previous step, recover (up to scaling ambiguities) the factors $(\mathbf{X}_\ell, \mathbf{X}_{\ell+1} \dots \mathbf{X}_L)$ by solving the minimization problem

$$\min_{\mathbf{X},\mathbf{Y}} \|\mathbf{X}_\ell \dots \mathbf{X}_L - \mathbf{X}\mathbf{Y}\|_F \quad \text{such that} \quad \begin{cases} \operatorname{supp}(\mathbf{X}) \subseteq \mathbf{S}_{\boldsymbol{\pi}_\ell} \\ \operatorname{supp}(\mathbf{Y}) \subseteq \mathbf{S}_{\boldsymbol{\pi}_{\ell+1} * \dots * \boldsymbol{\pi}_L} \end{cases} . \tag{6.16}$$

The choice of the support constraints on $(\mathbf{X}, \mathbf{Y})$ in the optimization problems (6.15) and (6.16) is motivated by Lemma 6.3, because it is assumed that $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$. If the recovery at each step $\ell \in [\![L-1]\!]$ is successful, then the hierarchical strategy indeed recovers the factors $(\mathbf{X}_\ell)_{\ell=1}^L$ (up to scaling ambiguities) from the observation of $\mathbf{A}$. We will show in the following that the recovery is indeed successful at each step $\ell \in [\![L-1]\!]$, by applying identifiability results from Section 6.2.

Generalization to arbitrary hierarchical order. In the previous paragraph, the recovery of the butterfly factors $(\mathbf{X}_\ell)_{\ell=1}^L$ in a hierarchical manner was performed in the *left-to-right* order, in the sense that it recovers sequentially the pair of factors $(\mathbf{X}_\ell, \mathbf{X}_{\ell+1} \dots \mathbf{X}_L)$ for $\ell = 1, \dots, L-1$. But the the hierarchical factorization method can be generalized to other factorization orders: for instance, in the case of four factors $\mathbf{X}_1, \dots, \mathbf{X}_4$ of size 16×16, one can instead factorize $\mathbf{A} := \mathbf{X}_1\mathbf{X}_2\mathbf{X}_3\mathbf{X}_4$ into $\mathbf{X}_1\mathbf{X}_2$ and $\mathbf{X}_3\mathbf{X}_4$ at the first level, then $\mathbf{X}_1\mathbf{X}_2$ into $\mathbf{X}_1, \mathbf{X}_2$, and finally $\mathbf{X}_3\mathbf{X}_4$ into $\mathbf{X}_3, \mathbf{X}_4$. Let us formally introduce a tree structure that describes the factorization order in the hierarchical method.



(a) Arbitrary (b) Balanced (c) Symmetric (d) Unbalanced

Figure 6.2: Four possible factor-bracketing binary trees of $[\![L]\!]$, $L = 8$, that can serve as input to Algorithm 6.1. NB: the algorithm is applicable to any number of factors L (not only $L = 8$).

Definition 6.4 (Factor-bracketing binary tree)**.** *A* factor-bracketing binary tree *of* $[\![r, t]\!]$*, with* $1 \leq r \leq t$*, is a binary tree, where nodes are non-empty subsets of* $[\![r, t]\!]$*, that satisfies the following axioms:*

1. *each node is a subset of consecutive indices in* $[\![r, t]\!]$*;*
2. *the root is the set* $[\![r, t]\!]$*;*
3. *a node is a singleton if, and only if, it is a leaf;*
4. *for each non-leaf node, the left and right children form a partition of their parent, in such a way that the indices of the left child are* smaller *than those in the right child.*

Such a tree has $t - r$ *non-leaf nodes.*

Examples of factor-bracketing binary trees are illustrated in Figure 6.2. As we will see, the proposed hierarchical algorithm works with *any* factor-bracketing tree.

Description of the hierarchical algorithm (Algorithm 6.1). Given as inputs a factor-bracketing binary tree $\mathcal{T}$ of $[\![L]\!]$ and any target matrix $\mathbf{A}$, Algorithm 6.1 visits the nodes of $\mathcal{T}$ in a breadth-first search order, starting by the root node. At each non-leaf node $[\![r, t]\!] \subseteq [\![L]\!]$ characterized by its so-called *splitting index* s, which is the maximum value of its left child, Algorithm 6.2 approximates an intermediate matrix $\mathbf{M}_{[\![r,t]\!]}$ by a product $\mathbf{M}_{[\![r,s]\!]}\mathbf{M}_{[\![s+1,t]\!]}$, where the left and right factor satisfy the support constraints encoded by $\mathrm{S} := (\mathbf{S}_{\pi_r * \ldots * \pi_s}, \mathbf{S}_{\pi_{s+1} * \ldots * \pi_t}{}^{\top})$, cf. line 7 of Algorithm 6.2. The procedure for computing these two factors is described in Algorithm 6.3, and is proved to be optimal in [212] in the sense that $\|\mathbf{M}_{[\![r,t]\!]} - \mathbf{X}\mathbf{Y}^{\top}\|_F^2$ is minimized among all $\mathbf{X}, \mathbf{Y}$ satisfying the support constraints S. In essence, denoting $(\boldsymbol{\mathcal{S}}_i)_{i=1}^n := \varphi(\mathrm{S})$, the procedure consists of successive best

Algorithm 6.1 Hierarchical algorithm for square dyadic butterfly factorization, size $n = 2^L$.

Require: $\mathbf{A} \in \mathbb{C}^{n \times n}$, $\mathcal{T}$ factor-bracketing tree of $[\![L]\!]$
Ensure: $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$ (cf. Definition 6.2)
1: $(\mathbf{X}_\ell)_{\ell=1}^L \leftarrow$ Algorithm 6.2$(\mathbf{A}, \mathcal{T})$
2: **return** $(\mathbf{X}_\ell)_{\ell=1}^L$

Algorithm 6.2 Hierarchical step for indices $1 \leq r \leq t \leq L$ in the square dyadic butterfly factorization of size $n = 2^L$.

Require: $\mathbf{M}_{[\![r,t]\!]} \in \mathbb{C}^{n \times n}$, factor-bracketing tree $\mathcal{T}_{[\![r,t]\!]}$ of $\{r, \ldots, t\}$
Ensure: $(\bar{\mathbf{X}}_r, \ldots, \bar{\mathbf{X}}_t) \in \Sigma^{\boldsymbol{\pi}_r} \times \cdots \times \Sigma^{\boldsymbol{\pi}_t}$
1: **if** $r = t$ **then**
2: **return** $(\mathbf{M}_{[\![r,t]\!]})$
3: **end if**
4: $s \leftarrow$ maximum value in the left child of the root of $\mathcal{T}_{[\![r,t]\!]}$
5: $\mathcal{T}_{[\![r,s]\!]}, \mathcal{T}_{[\![s+1,t]\!]} \leftarrow$ left and right subtrees of the root of $\mathcal{T}_{[\![r,t]\!]}$
6: $\mathbf{S}_{\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s}, \mathbf{S}_{\boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t} \leftarrow$ supports defined by (6.14)
7: $\mathbf{M}_{[\![r,s]\!]}, \mathbf{M}_{[\![s+1,t]\!]}^\top \leftarrow$ Algorithm 6.3$\left(\mathbf{M}_{[\![r,t]\!]}, \mathbf{S}_{\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s}, \mathbf{S}_{\boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t}^\top\right)$
8: $(\bar{\mathbf{X}}_r, \ldots, \bar{\mathbf{X}}_s) \leftarrow$ Algorithm 6.2 $(\mathbf{M}_{[\![r,s]\!]}, \mathcal{T}_{[\![r,s]\!]})$
9: $(\bar{\mathbf{X}}_{s+1}, \ldots, \bar{\mathbf{X}}_t) \leftarrow$ Algorithm 6.2 $(\mathbf{M}_{[\![s+1,t]\!]}, \mathcal{T}_{[\![s+1,t]\!]})$
10: **return** $(\bar{\mathbf{X}}_r, \ldots, \bar{\mathbf{X}}_s, \bar{\mathbf{X}}_{s+1}, \ldots, \bar{\mathbf{X}}_t)$

rank-one approximations (in the Frobenius norm) of submatrices $\mathbf{M}_{[\![r,t]\!]} \odot \boldsymbol{\mathcal{S}}_i$ for each $i \in [\![n]\!]$ (line 3 of Algorithm 6.3), which can be computed for instance via a *truncated SVD*.

Moreover, we will show that S satisfies the condition of Proposition 6.1, meaning that if the intermediate matrix $\mathbf{M}_{[\![r,t]\!]}$ is *exactly* the product of two factors satisfying the support constraint S, then these two factors are *essentially unique*, and are recovered by Algorithm 6.3 because of its optimality for the corresponding minimization problem.[3] The hierarchical procedure is then repeated recursively on $\mathbf{M}_{[\![r,s]\!]}$ and $\mathbf{M}_{[\![s+1,t]\!]}$, with their respective trees.

Remark 6.8. *One can exploit Algorithm 6.1* beyond the exact setting *to approximate* any *matrix* $\mathbf{A}$ *of size* 2^L *by a matrix having the square dyadic butterfly structure. The procedure in Algorithm 6.3 for two-layer fixed-support matrix factorization is optimal for the corresponding minimization problem [212]. But since this procedure is used in a* recursive greedy *fashion in Algorithm 6.1, global optimality of the resulting multi-layer factorization is not necessarily guaranteed. The stability of the hierarchical algorithm beyond exact recovery is studied in Chapter 7.*

[3] More precisely, we assume $\mathbf{M}_{[\![r,t]\!]} \in \mathtt{IC}^S \cap \mathtt{MC}^S$.

Algorithm 6.3 Fixed-support matrix factorization under assumptions of Proposition 6.1 [212, Algorithm 3.1].

Require: $\mathbf{A} \in \mathbb{C}^{m\times n}, \mathbf{S}_{\text{left}} \in \{0,1\}^{m\times k}, \mathbf{S}_{\text{right}} \in \{0,1\}^{n\times k}$
Ensure: $(\mathbf{X}, \mathbf{Y}) \in \Sigma^{\mathbf{S}_{\text{left}}} \times \Sigma^{\mathbf{S}_{\text{right}}}$
1: $(\boldsymbol{\mathcal{S}}_i)_{i=1}^n \leftarrow \varphi\left(\mathbf{S}_{\text{left}}, \mathbf{S}_{\text{right}}\right)$ as in (6.10) ▷ *pairwise disjoint by assumption*
2: **for** $i \in [\![k]\!]$ **do**
3: $\quad (\hat{\mathbf{x}}_i, \hat{\mathbf{y}}_i) \leftarrow \arg\min_{\mathbf{x}_i,\mathbf{y}_i} \left\{\|\mathbf{A} \odot \boldsymbol{\mathcal{S}}_i - \mathbf{x}_i\mathbf{y}_i{}^\top\|_F, \mid \operatorname{supp}(\mathbf{x}_i\mathbf{y}_i{}^\top) \subseteq \boldsymbol{\mathcal{S}}_i\right\}$
4: **end for**
5: $\mathbf{X} \leftarrow (\hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_r) \in \mathbb{C}^{m\times k}$
6: $\mathbf{Y} \leftarrow (\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_r) \in \mathbb{C}^{n\times k}$
7: **return** $(\mathbf{X}, \mathbf{Y})$

6.3.2 Uniqueness of the square dyadic butterfly factorization

As the main contribution of the chapter, we now show that the exact factorization $\mathbf{A} = \mathbf{X}_1 \ldots \mathbf{X}_L$ into L factors constrained to the square dyadic butterfly supports is essentially unique, and that these factors can be recovered by Algorithm 6.1. Let us first generalize Definition 6.3 to the multi-layer case with $L \geq 3$.

Definition 6.5 (Essential uniqueness of a multi-layer factorization in Σ)**.** *Consider integers $n_0, \ldots, n_L$, a set $\Sigma \subseteq \mathbb{C}^{n_0\times n_1} \times \ldots \times \mathbb{C}^{n_{L-1}\times n_L}$ of L-tuples of factors, and a matrix $\mathbf{A}$ admitting a factorization $\mathbf{A} := \mathbf{X}_1 \ldots \mathbf{X}_L$ such that $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma$.*
We say that this factorization is essentially unique *in Σ, if any $(\bar{\mathbf{X}}_1, \ldots, \bar{\mathbf{X}}_L) \in \Sigma$ such that $\bar{\mathbf{X}}_1 \ldots \bar{\mathbf{X}}_L = \mathbf{A}$ is equivalent to $(\mathbf{X}_1, \ldots, \mathbf{X}_L)$, written $(\mathbf{X}_\ell)_{\ell=1}^L \sim (\bar{\mathbf{X}}_\ell)_{\ell=1}^L$, in the sense that there exist invertible diagonal matrices $\mathbf{D}_1, \ldots, \mathbf{D}_{L-1}$ such that $\bar{\mathbf{X}}_\ell = \mathbf{D}_{\ell-1}{}^{-1}\mathbf{X}_\ell\mathbf{D}_\ell$ for all $\ell \in [\![L]\!]$, with the convention that $\mathbf{D}_0$ and $\mathbf{D}_L$ are identity matrices.*

Theorem 6.1 (Essential uniqueness of square dyadic butterfly factorization)**.** *Consider S_β the square dyadic butterfly supports of size $n = 2^L$ and $(\mathbf{X}_1, \ldots, \mathbf{X}_L) \in \Sigma^\beta$. Assume that $\mathbf{X}_\ell$ does not have a zero column for $1 \leq \ell \leq L-1$, and not a zero row for $2 \leq \ell \leq L$.*
Then, the factorization $\mathbf{A} := \mathbf{X}_1 \ldots \mathbf{X}_L$ is essentially unique in Σ^β. These factors can be recovered *from $\mathbf{A}$, up to scaling ambiguities only, using Algorithm 6.1 with inputs $(\mathbf{A}, \mathcal{T})$, where $\mathcal{T}$ is* any *factor-bracketing tree of $[\![L]\!]$.*

In other words, Algorithm 6.1 is endowed with *exact recovery guarantees*. In particular, Theorem 6.1 can be applied to show identifiability of the square dyadic butterfly factorization of the DFT matrix as suggested in [214, Chapter 7], but also the one of the Hadamard matrix. In both cases, the butterfly factors of the DFT or the Hadamard matrix can be recovered up to scaling ambiguities via Algorithm 6.1 with *any* factor-bracketing binary tree of $[\![L]\!]$ as input. Before proving Theorem 6.1, we show that Algorithm 6.1 has controlled complexity bounds.

6.3.3 Complexity bounds

Existing algorithms for square dyadic butterfly factorization [74, 216] are based on gradient descent, and as such they require to tune several criteria such as the learning rate or the stopping criteria. In contrast, Algorithm 6.1 has a bounded complexity as it essentially consists in a controlled number of truncated SVDs to compute rank-one approximations of submatrices. While the *full SVD* of a matrix of size $m \times n$ would require $\mathcal{O}(mn \min(m,n))$ flops, *partial SVD* with numerical rank k requires only $\mathcal{O}(kmn)$ flops (see e.g. [153] and references therein). Hence, in our complexity analysis of Algorithm 6.1, the theoretical complexity of computing the best rank-one approximation of a matrix of size $m \times n$ will be $\mathcal{O}(mn)$. Below we estimate and compare the complexity for two types of factor-bracketing binary trees.

Unbalanced tree (left-to-right or right-to-left). First we consider running Algorithm 6.1 with a matrix $\mathbf{A}$ of size $n \times n$, $n = 2^L$ ($L \geq 2$), and the *left-to-right unbalanced* factor-bracketing binary tree $\mathcal{T}$ of $[\![L]\!]$ (defined as the factor-bracketing binary tree where the left child of each non-leaf node is a singleton, see Figure 6.2d) as inputs. There are in total $L-1$ non-leaf nodes in this tree. At the non-leaf node of level $\ell \in \{0, \dots, L-2\}$, the algorithm computes the best rank-one approximation of n submatrices of size $2 \times n/2^{\ell+1}$, which yields a cost of the order of $n \times (2 \times n/2^{\ell+1}) = n^2/2^\ell$. Hence, the total cost of Algorithm 6.1 with the unbalanced factor-bracketing binary tree is of the order of:

$$\sum_{\ell=0}^{L-2} \frac{n^2}{2^\ell} = 2\left(1 - 2^{-L+1}\right) n^2 = 2\left(1 - \frac{2}{n}\right) n^2 = \mathcal{O}(n^2).$$

Similarly, the complexity is $\mathcal{O}(n^2)$ when best rank-one approximations are computed with full SVDs (see Appendix B.4). The complexity bounds are the same for the analog *right-to-left unbalanced* tree.

Balanced tree. Consider now Algorithm 6.1 with an $n \times n$ matrix $\mathbf{A}$, $n = 2^L$ where L is also a power of 2, and the *balanced* factor-bracketing binary tree $\mathcal{T}$ of $[\![L]\!]$ (i.e., all children of each non-leaf node have the same cardinality, see Figure 6.2b). At each non-leaf node of level $\ell \in \{0, \dots, \log_2(L) - 1\}$, the best rank-one approximation of n square submatrices of size $\sqrt{n^{1/2^\ell}}$ is computed, at a cost of the order of $n \times (\sqrt{n^{1/2^\ell}} \times \sqrt{n^{1/2^\ell}}) = n \times n^{1/2^\ell}$. At each level $\ell \in [\![0, \log_2(L) - 1]\!]$, there are 2^ℓ nodes. As $2^\ell \leq L/2 = \log_2(n)/2$, the total cost of Algorithm 6.1 is of the order of:

$$\begin{aligned}\sum_{\ell=0}^{\log_2(L)-1} 2^\ell n \times n^{1/2^\ell} &= n^2 + \sum_{\ell=1}^{\log_2(L)-1} 2^\ell n^{1+1/2^\ell} \\ &\leq n^2 + \sum_{\ell}^{\log_2(\log_2(n))-1} \frac{\log_2(n)}{2} n^{3/2} = \mathcal{O}(n^2).\end{aligned}$$

This contrasts with the complexity $\mathcal{O}(n^{5/2})$ when using full SVDs, see Appendix B.4 for more details.

Discussion. The complexity of Algorithm 6.1 is of the same order of magnitude as that of matrix-vector multiplications of size $n \times n$, which is $\mathcal{O}(n^2)$. Assume that we want to compute the product $\mathbf{AB}$, where $\mathbf{A}, \mathbf{B}$ are of size $n \times n$, and $\mathbf{A}$ admits the butterfly structure. The naive computation requires $\mathcal{O}(n^3)$. But the data-sparse representation of $\mathbf{A}$ as a product of butterfly factors can be recovered by Algorithm 6.1 in $\mathcal{O}(n^2)$, in order to enable fast $\mathcal{O}(n \log n)$ matrix-vector multiplication [74]. In other words, this method allows a computation of $\mathbf{AB}$ in $\mathcal{O}(n^2 \log n)$ complexity only, instead of $\mathcal{O}(n^3)$.

This quadratic complexity is the typical complexity of previous butterfly algorithms [289], without considering any *randomized algorithm* for low-rank approximation [99, 153] as proposed in [234]. See Section 3.6 for more details.

Finally, it is possible to implement Algorithm 6.1 in a *distributed* fashion. Indeed, the computation of the best rank-one approximation of each submatrix at line 3 of Algorithm 6.3 can be performed in parallel: in the setting with T threads, each thread computes the best rank-one approximation of $\lfloor n/T \rfloor$ submatrices. Moreover, when running Algorithm 6.1 with a balanced factor-bracketing binary tree, the implementation can be further parallelized, since the factorization at each node of the same level can be performed by independent threads.

6.3.4 Proof of uniqueness (Theorem 6.1)

This subsection is dedicated to the proof of Theorem 6.1. It is technical and the reader can skip this part on a first reading. In order to prove Theorem 6.1, consider $(\mathbf{X}_1, \ldots, \mathbf{X}_L) \in \Sigma^\beta$ satisfying the square dyadic butterfly constraint. These factors are assumed to verify the assumption of Theorem 6.1. For any $1 \leq r \leq t \leq L$, denote $\mathbf{X}_{[\![r,t]\!]} := \mathbf{X}_r \ldots \mathbf{X}_t$, and $\mathbf{A} := \mathbf{X}_{[\![1,L]\!]} = \mathbf{X}_1 \ldots \mathbf{X}_L$. Fix any factor-bracketing binary tree $\mathcal{T}$ of $[\![L]\!]$. Given $\mathbf{A}$ and $\mathcal{T}$ as the inputs of Algorithm 6.1, we write $\mathbf{M}_{[\![r,t]\!]}$ the intermediate matrix obtained at each node $[\![r,t]\!]$ of $\mathcal{T}$ from the hierarchical factorization procedure. The proof is now separated into two steps.

1. We prove that Algorithm 6.1 recovers the butterfly factors $(\mathbf{X}_1, \ldots, \mathbf{X}_L)$ from $\mathbf{A}$, up to scaling ambiguities.

2. We prove that the butterfly factorization $\mathbf{A} = \mathbf{X}_1 \ldots \mathbf{X}_L$ is indeed essentially unique in the sense of Definition 6.5.

(1) The algorithm recovers the butterfly factors

Overview. The proof of the first part consists in conducting an induction over the non-leaf nodes $N_1, \ldots, N_{L-1}$ of the tree, ordered in a breadth-first-search order. The general idea is to show that Algorithm 6.1 reconstructs *recursively* from $\mathbf{A}$ the *partial products* $\mathbf{X}_{[\![r,t]\!]}$ up to scaling ambiguities at each node $[\![r,t]\!]$ of the

tree $\mathcal{T}$. To that end, we need to prove a *crucial lemma* (Lemma 6.4) that essentially reduces the analysis of the multi-layer factorization to the case with only two factors. Then, the proof of Lemma 6.4 itself relies on two key ingredients about *optimality* (Proposition 6.2) and *essential uniqueness* (Proposition 6.3) of the considered two-layer factorization problem.

For each $v \in [\![L-1]\!]$, denote r_v, t_v the minimum and maximum index of node $N_v = [\![r_v, t_v]\!]$, and s_v as the "splitting" index of node N_v, which is the maximum value of its *left* child. In the proof we will use the following consequence of the breadth-first-search order.

Remark 6.9. *For any $u \in \{2, \dots, L-1\}$, the node $N_u = [\![r_u, t_u]\!]$ is a child of some node $N_v = [\![r_v, s_v]\!] \cup [\![s_v+1, \dots, t_v]\!]$ with $v \in [\![u-1]\!]$. If N_u is the left child of N_v, then $(r_u, t_u) = (r_v, s_v)$. If N_u is the right child of N_v, then $(r_u, t_u) = (s_v+1, t_v)$.*

Proof by induction. Define for any $V \in [\![L-1]\!]$, the assertion P_V: "there exist invertible diagonal matrices $\mathbf{D}_{s_1}, \dots, \mathbf{D}_{s_V}$ such that, for each $v \in [\![V]\!]$, we have: $\mathbf{M}_{[\![r_v,s_v]\!]} = \mathbf{D}_{r_v-1}{}^{-1}\mathbf{X}_{[\![r_v,s_v]\!]}\mathbf{D}_{s_v}$ and $\mathbf{M}_{[\![s_v+1,t_v]\!]} = \mathbf{D}_{s_v}{}^{-1}\mathbf{X}_{[\![s_v+1,t_v]\!]}\mathbf{D}_{t_v}$", with the convention[4] $\mathbf{D}_0 = \mathbf{D}_L = \mathbf{I}_n$.

The principle of the proof is to show P_V by *induction* for all $V \in [\![L-1]\!]$. In particular, proving P_{L-1} yields our claim, as we now explain. Indeed, any leaf node $\{\ell\}$ is either a left child or a right child of a non-leaf node $N_v = \{r_v, \dots, t_v\}$ with $v \in [\![L-1]\!]$. In the case of a left child, $\{\ell\} = \{r_v, \dots, s_v\}$ hence $\ell = r_v = s_v$, and in the other case $\{\ell\} = \{s_v+1, \dots, t_v\}$ hence $\ell = s_v+1 = t_v$. In both cases assertion P_{L-1} implies that $\mathbf{M}_{[\![\ell,\ell]\!]} = \mathbf{D}_{\ell-1}{}^{-1}\mathbf{X}_{[\![\ell,\ell]\!]}\mathbf{D}_\ell = \mathbf{D}_{\ell-1}{}^{-1}\mathbf{X}_\ell\mathbf{D}_\ell$, hence $(\mathbf{M}_{[\![\ell,\ell]\!]})_{\ell=1}^L \sim (\mathbf{X}_\ell)_{\ell=1}^L$, meaning that the algorithm recovers the butterfly factors up to scaling ambiguities only.

Crucial lemma. The following lemma is *central* in the proof by induction.

Lemma 6.4. *Under the assumptions of Theorem 6.1, consider $V \in [\![L-1]\!]$ and assume that there are invertible diagonal matrices $\mathbf{D}_{r_V-1}$ and $\mathbf{D}_{t_V}$ such that $\mathbf{M}_{[\![r_V,t_V]\!]} = \mathbf{D}_{r_V-1}{}^{-1}\mathbf{X}_{[\![r_V,t_V]\!]}\mathbf{D}_{t_V}$.*
Then the pair $(\mathbf{M}_{[\![r_V,s_V]\!]}, \mathbf{M}_{[\![s_V+1,t_V]\!]}{}^\top)$ computed at line 7 of Algorithm 6.2 such that the product $\mathbf{M}_{[\![r_V,s_V]\!]}\mathbf{M}_{[\![s_V+1,t_V]\!]}$ approximates $\mathbf{M}_{[\![r_V,t_V]\!]}$ is equal (up to scaling ambiguities) to the pair $(\bar{\mathbf{X}}, \bar{\mathbf{Y}})$ where

$$\bar{\mathbf{X}} := \mathbf{D}_{r_V-1}{}^{-1}\mathbf{X}_{[\![r_V,s_V]\!]}, \quad \bar{\mathbf{Y}}^\top := \mathbf{X}_{[\![s_V+1,t_V]\!]}\mathbf{D}_{t_V}.$$

[4] Remark that for any $v \in [\![V]\!]$ the node $N_v = \{r_v, \dots, t_v\}$ is either the root node (when $v = 1$) or a child of a node n_w with $w < v$. In the latter case, by Remark 6.9, either $(r_v, t_v) = (r_w, s_w)$, or $(r_v, t_v) = (s_w+1, t_w)$, with $w \in [\![v-1]\!] \subseteq [\![V]\!]$. In other words, $r_v - 1, s_v, t_v \in \{0, s_1, \dots, s_V, L\}$ for all $v \in [\![V]\!]$, meaning that the diagonal matrices $\mathbf{D}_{r_v-1}$, $\mathbf{D}_{s_v}$ and $\mathbf{D}_{t_v}$ used in the definition of P_V above are well defined.

Indeed, since $\mathbf{M}_{[\![r_1,t_1]\!]} = \mathbf{A} = \mathbf{X}_{[\![r_1,t_1]\!]}$ and since $\mathbf{D}_{r_1-1} = \mathbf{D}_{t_1} = \mathbf{I}_n$ (recall that $r_1 = 1$ and $t_1 = L$), this lemma applied to $V = 1$ shows that P_1 is true. This starts the induction, and we now show that the lemma can similarly be used to proceed to the induction. Assume that P_{V-1} is true where $V \in [\![2, L-1]\!]$ and consider $\mathbf{D}_{s_1}, \dots, \mathbf{D}_{s_{V-1}}$ the corresponding invertible diagonal matrices. By Remark 6.9, the parent of node N_V is necessarily some node N_v with $v \in [\![V-1]\!]$ and, depending on whether N_V is a left or right child of N_v, we have either $N_V = [\![r_v, s_v]\!]$ or $N_V = [\![s_v + 1, t_v]\!]$. Without loss of generality assume the former (the proof is similar if we suppose the latter) so that $(r_V, t_V) = (r_v, s_v)$. Since P_{V-1} is true we have that $\mathbf{M}_{[\![r_V,t_V]\!]} = \mathbf{M}_{[\![r_v,s_v]\!]} = \mathbf{D}_{r_v-1}{}^{-1}\mathbf{X}_{[\![r_v,s_v]\!]}\mathbf{D}_{s_v} = \mathbf{D}_{r_V-1}{}^{-1}\mathbf{X}_{[\![r_V,t_V]\!]}\mathbf{D}_{t_V}$. By Lemma 6.4, there exists an invertible diagonal matrix $\mathbf{D}_{s_V}$ such that $\mathbf{M}_{[\![r_V,s_V]\!]} = \mathbf{D}_{r_V-1}{}^{-1}\mathbf{X}_{[\![r_V,s_V]\!]}\mathbf{D}_{s_V}$ and $\mathbf{M}_{[\![s_V+1,t_V]\!]} = \mathbf{D}_{s_V}{}^{-1}\mathbf{X}_{[\![s_V+1,t_V]\!]}\mathbf{D}_{t_V}$, which proves P_V.

Proving the crucial lemma. We now focus on the proof of Lemma 6.4. It relies on two key ingredients formulated in Proposition 6.2 and Proposition 6.3 below. They are both derived from the following property of the square dyadic butterfly supports, proved in Appendix B.5.

Lemma 6.5. *Given $1 \leq r \leq s < t \leq L$, denoting $\mathbf{S}_{left} := \mathbf{S}_{\pi_r * \dots * \pi_s}$ and $\mathbf{S}_{right} := \mathbf{S}_{\pi_{s+1} * \dots * \pi_t}{}^\top$ where we recall (6.14), the tuple of rank-one supports $\varphi\left(\mathbf{S}_{left}, \mathbf{S}_{right}\right)$ has disjoint rank-one supports.*

The first consequence of this property is the following *optimality* result.

Proposition 6.2 (Application of [212, Theorem 3.3])**.** *Denote $\mathrm{S} := (\mathbf{S}_{\pi_r * \dots * \pi_s}, \mathbf{S}_{\pi_{s+1} * \dots * \pi_t}{}^\top)$. For arbitrary matrix $\mathbf{M}$, the procedure* PairwiseDisjointFSMF$(\mathbf{M}, \mathbf{S}_{\pi_r * \dots * \pi_s}, \mathbf{S}_{\pi_{s+1} * \dots * \pi_t}{}^\top)$ *described in Algorithm 6.3 computes in polynomial time a pair of factors solving the problem: $\min_{(\mathbf{X},\mathbf{Y}) \in \Sigma^{\mathrm{S}}} \|\mathbf{M} - \mathbf{X}\mathbf{Y}^\top\|_F$.*

The second consequence is that it allows a characterization of the set $\mathcal{U}(\Sigma^{\mathrm{S}})$ when $\mathrm{S} := (\mathbf{S}_{\pi_r * \dots * \pi_s}, \mathbf{S}_{\pi_{s+1} * \dots * \pi_t}{}^\top)$.

Lemma 6.6. *Denote $\mathrm{S} = (\mathbf{S}_{left}, \mathbf{S}_{right}) := (\mathbf{S}_{\pi_r * \dots * \pi_s}, \mathbf{S}_{\pi_{s+1} * \dots * \pi_t}{}^\top)$. We have*

$$\mathcal{U}\left(\Sigma^{\mathrm{S}}\right) = \left\{(\mathbf{X}, \mathbf{Y}) \in \Sigma^{\mathrm{S}} \mid \mathrm{colsupp}(\mathbf{X}) = \mathrm{colsupp}(\mathbf{Y}) = [\![n]\!]\right\}.$$

Proof. By Proposition 6.1, $\mathcal{U}\left(\Sigma^{\mathrm{S}}\right) = \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}}$. Since

$$\mathrm{colsupp}(\mathbf{S}_{\mathrm{left}}) = \mathrm{colsupp}(\mathbf{S}_{\mathrm{right}}) = [\![n]\!],$$

by definition of $\mathtt{IC}^{\mathrm{S}}$ and $\mathtt{MC}^{\mathrm{S}}$, cf. (6.8)-(6.9), we have $(\mathbf{X}, \mathbf{Y}) \in \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}}$ if, and only if, $\mathrm{colsupp}(\mathbf{X}) = \mathrm{colsupp}(\mathbf{Y}) = [\![n]\!]$. □

Recall that the factors $(\mathbf{X}_1, \dots, \mathbf{X}_L)$ verify the assumption of Theorem 6.1, i.e., $\mathbf{X}_\ell$ does not have a zero column for $1 \leq \ell \leq L-1$, and not a zero row for $2 \leq \ell \leq L$. As claimed in the following lemma proved in Appendix B.6, this assumption is in fact a necessary and sufficient condition to ensure that each pair of partial products $(\mathbf{X}_{[\![r,s]\!]}, \mathbf{X}_{[\![s+1,t]\!]}^\top)$ with $1 \leq r \leq s < t \leq L$ is non-degenerate (in the sense that the left and right factor do not have a zero column).

Lemma 6.7. *Let S_β be the square dyadic butterfly supports of size $n = 2^L$, and $(\mathbf{X}_1, \dots, \mathbf{X}_L) \in \Sigma^\beta$. The following are equivalent:*

1. *for each $1 \leq r \leq s < t \leq L$, $\mathbf{X}_{[\![r,s]\!]} := \mathbf{X}_r \dots \mathbf{X}_s$ does not have a zero column, and $\mathbf{X}_{[\![s+1,t]\!]} := \mathbf{X}_{\ell+1} \dots \mathbf{X}_t$ does not have a zero row;*
2. *$\mathbf{X}_\ell$ does not have a zero column for $1 \leq \ell \leq L-1$, and not a zero row for $2 \leq \ell \leq L$.*

Consequently, we obtain the second key ingredient for the proof of Lemma 6.4.

Proposition 6.3. *Assume that $(\mathbf{X}_1, \dots, \mathbf{X}_L) \in \Sigma^\beta$ verifies the hypothesis of Theorem 6.1. Let $1 \leq r \leq s < t \leq L$. Denote $\mathrm{S} := (\mathbf{S}_{\pi_r * \dots * \pi_s}, \mathbf{S}_{\pi_{s+1} * \dots * \pi_t}^\top)$. Then, for any invertible diagonal matrices $\mathbf{D}, \bar{\mathbf{D}}$, denoting $\bar{\mathbf{X}} := \mathbf{D}^{-1}\mathbf{X}_{[\![r,s]\!]}$ and $\bar{\mathbf{Y}} := (\mathbf{X}_{[\![s+1,t]\!]}\bar{\mathbf{D}})^\top$, the factorization $\mathbf{D}^{-1}\mathbf{X}_{[\![r,t]\!]}\bar{\mathbf{D}} = \bar{\mathbf{X}}\bar{\mathbf{Y}}^\top$ into two factors $(\bar{\mathbf{X}}, \bar{\mathbf{Y}})$ is essentially unique in Σ^{S}.*

Proof. By Lemma 6.3, $(\mathbf{X}_{[\![r,s]\!]}, \mathbf{X}_{[\![s+1,t]\!]}^\top) \in \Sigma^{\mathrm{S}}$. By Lemma 6.7 and the assumption on the factors $\mathbf{X}_\ell$ ($\ell \in [\![L]\!]$), the matrices $\mathbf{X}_{[\![r,s]\!]}$ and $\mathbf{X}_{[\![s+1,t]\!]}^\top$ do not have a zero column. The same is true for $\mathbf{D}^{-1}\mathbf{X}_{[\![r,s]\!]}$ and $\bar{\mathbf{D}}\mathbf{X}_{[\![s+1,t]\!]}^\top$, as the multiplication of a matrix by $\mathbf{D}^{-1}$ or $\bar{\mathbf{D}}$ does not change its support. By Lemma 6.6, $(\mathbf{D}^{-1}\mathbf{X}_{[\![r,s]\!]}, \bar{\mathbf{D}}\mathbf{X}_{[\![s+1,t]\!]}^\top) \in \mathcal{U}(\Sigma^{\mathrm{S}})$. □

We now have all the ingredients to prove the crucial Lemma 6.4, which ends the inductive proof of the first part of Theorem 6.1 showing that Algorithm 6.1 *recovers* the butterfly factors $(\mathbf{X}_\ell)_{\ell=1}^L$ from $\mathbf{A} = \mathbf{X}_1 \dots \mathbf{X}_L$, up to scaling ambiguities.

Proof of Lemma 6.4. Since $\mathbf{X}_{[\![r_V,t_V]\!]} = \mathbf{X}_{[\![r_V,s_V]\!]}\mathbf{X}_{[\![s_V+1,t_V]\!]}$, we can factorize the matrix $\mathbf{M}_{[\![r_V,t_V]\!]} = {\mathbf{D}_{r_V-1}}^{-1}\mathbf{X}_{[\![r_V,t_V]\!]}\mathbf{D}_{t_V}$ as $\mathbf{M}_{[\![r_V,t_V]\!]} = \bar{\mathbf{X}}\bar{\mathbf{Y}}^\top$, with $\bar{\mathbf{X}} := {\mathbf{D}_{r_V-1}}^{-1}\mathbf{X}_{[\![r_V,s_V]\!]}$, $\bar{\mathbf{Y}} := (\mathbf{X}_{[\![s_V+1,t_V]\!]}\mathbf{D}_{t_V})^\top$. By Lemma 6.3, $(\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \in \Sigma^{\mathrm{S}}$ where

$$\mathrm{S} := (\mathbf{S}_{\pi_{r_V} * \dots * \pi_{s_V}}, {\mathbf{S}_{\pi_{s_V+1} * \dots * \pi_{t_V}}}^\top).$$

By Proposition 6.2, the factors $(\mathbf{M}_{[\![r_V,s_V]\!]}, \mathbf{M}_{[\![s_V+1,t_V]\!]}^\top) \in \Sigma^{\mathrm{S}}$ computed at line 7 of Algorithm 6.2 minimize the optimization problem: $\min_{(\mathbf{X},\mathbf{Y}) \in \Sigma^{\mathrm{S}}} \|\mathbf{M}_{[\![r_V,t_V]\!]} - \mathbf{X}\mathbf{Y}^\top\|_F$. Since $\mathbf{M}_{[\![r_V,t_V]\!]} = \bar{\mathbf{X}}\bar{\mathbf{Y}}^\top$ with $(\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \in \Sigma^{\mathrm{S}}$, this minimum is zero hence

the computed pair satisfies $\mathbf{M}_{[\![r_V,t_V]\!]} = \mathbf{M}_{[\![r_V,s_V]\!]}\mathbf{M}_{[\![s_V+1,t_V]\!]}$. By Proposition 6.3, the factorization $\mathbf{M}_{[\![r_V,t_V]\!]} = \bar{\mathbf{X}}\bar{\mathbf{Y}}^\top$ into two factors is essentially unique in Σ^{S}, so $(\mathbf{M}_{[\![r_V,s_V]\!]}, \mathbf{M}_{[\![s_V+1,t_V]\!]}{}^\top) \sim (\bar{\mathbf{X}}, \bar{\mathbf{Y}})$. □

(2) The factorization is essentially unique

We now show that the factorization $\mathbf{A} = \mathbf{X}_1 \dots \mathbf{X}_L$ is *essentially unique* in Σ^β (Definition 6.5). To that end, consider the factors $(\mathbf{M}_{[\![\ell,\ell]\!]})_{\ell=1}^L$ computed using Algorithm 6.1 with $\mathbf{A}$ as input, as well as arbitrary factors $(\bar{\mathbf{X}}_\ell)_{\ell=1}^L \in \Sigma^\beta$ such that $\bar{\mathbf{X}}_1 \dots \bar{\mathbf{X}}_L = \mathbf{A}$. We will show that $(\bar{\mathbf{X}}_\ell)_{\ell=1}^L$ verifies the assumptions of Theorem 6.1: this will imply that $(\mathbf{M}_{[\![\ell,\ell]\!]})_{\ell=1}^L$ is rescaling-equivalent *both* to $(\mathbf{X}_\ell)_{\ell=1}^L$ and to $(\bar{\mathbf{X}}_\ell)_{\ell=1}^L$, hence, by transitivity, $(\mathbf{X}_\ell)_{\ell=1}^L \sim (\bar{\mathbf{X}}_\ell)_{\ell=1}^L$ as claimed.

Denote $\bar{\mathbf{X}}_{[\![r,t]\!]} := \bar{\mathbf{X}}_r \dots \bar{\mathbf{X}}_t$ for any $1 \le r \le t \le L$. For any $\ell \in [\![L-1]\!]$, we have $\mathbf{X}_{[\![1,\ell]\!]}\mathbf{X}_{[\![\ell+1,L]\!]} = \mathbf{A} = \bar{\mathbf{X}}_{[\![1,\ell]\!]}\bar{\mathbf{X}}_{[\![\ell+1,L]\!]}$. By Lemma 6.3, $(\bar{\mathbf{X}}_{[\![1,\ell]\!]}, \bar{\mathbf{X}}_{[\![\ell+1,L]\!]}{}^\top) \in \Sigma^{\mathrm{S}}$ where $\mathrm{S} := (\mathbf{S}_{\boldsymbol{\pi}_1 * \dots * \boldsymbol{\pi}_\ell}, \mathbf{S}_{\boldsymbol{\pi}_{\ell+1} * \dots * \boldsymbol{\pi}_L}{}^\top)$. Besides, by assumption and by Lemma 6.7, $\mathbf{X}_{[\![1,\ell]\!]}$ and $\mathbf{X}_{[\![\ell+1,L]\!]}{}^\top$ do not have a zero column, so by Lemma 6.6, $(\mathbf{X}_{[\![1,\ell]\!]}, \mathbf{X}_{[\![\ell+1,L]\!]}{}^\top) \in \mathcal{U}(\Sigma^{\mathrm{S}})$. By definition of the set $\mathcal{U}(\Sigma^{\mathrm{S}})$, $(\bar{\mathbf{X}}_{[\![1,\ell]\!]}, \bar{\mathbf{X}}_{[\![\ell+1,L]\!]}{}^\top) \sim (\mathbf{X}_{[\![1,\ell]\!]}, \mathbf{X}_{[\![\ell+1,L]\!]}{}^\top)$. Since $\mathbf{X}_{[\![1,\ell]\!]}$ and $\mathbf{X}_{[\![\ell+1,L]\!]}{}^\top$ do not have a zero column, this implies that $\bar{\mathbf{X}}_{[\![1,\ell]\!]}$ and $\bar{\mathbf{X}}_{[\![\ell+1,L]\!]}{}^\top$ also do not have a zero column. Consequently, $\bar{\mathbf{X}}_\ell$ does not have a zero column (otherwise $\bar{\mathbf{X}}_{[\![1,\ell]\!]}$ would have a zero column) and similarly $\bar{\mathbf{X}}_{\ell+1}$ does not have a zero row. As this holds for any $\ell \in [\![L-1]\!]$, $(\bar{\mathbf{X}}_\ell)_{\ell=1}^L$ verifies the assumption of Theorem 6.1. This ends the proof of Theorem 6.1.

6.4 Related work

We now discuss the positioning of the proposed hierarchical algorithm with respect to previous butterfly algorithms. We also review some existing work on identifiability in multilinear inverse problems.

Similarities with previous butterfly algorithms. The proposed hierarchical algorithm (Algorithm 6.1) that comes with the identifiability result (Theorem 6.1) can be seen as a variant of *butterfly algorithms* [40, 234, 266, 289] detailed in Chapter 3, because of the following correspondences:

1. The best rank-one approximations performed each step of the hierarchical algorithm (Algorithm 6.1) corresponds to the core step of the butterfly algorithm described in Section 3.5.1 that computes column bases $\{\mathbf{U}_{R,C}\}_{R,C}$ and row bases $\{\mathbf{V}_{R,C}\}_{R,C}$ in order to approximate a family of some specific submatrices by $\mathbf{U}_{R,C}\mathbf{V}_{R,C}{}^\top$ for each subset of row and columns indices R, C.

2. The best rank-one approximations of the hierarchical algorithm are repeated recursively on the left and right factors obtained after each two-layer sparse

matrix factorization (cf. Algorithm 6.1), which corresponds, e.g., to the induction step of the butterfly algorithm that continues to perform low-rank approximation on some specific concatenation of column bases $\begin{pmatrix}\mathbf{U}_{R,C_1} & \mathbf{U}_{R,C_2}\end{pmatrix}$, as described in Section 3.5.1.

One the one hand, previous butterfly algorithms [40, 234, 266, 289] decompose the target matrix assuming that it satisfies the complementary low-rank property [234], as shown in Section 3.5. One the other hand, the proposed hierarchical algorithm (Algorithm 6.1) assumes that the target matrix admits a *sparse factorization with some specific fixed-support constraints on the sparse factors*, as illustrated in Figure 1.1. As detailed in Chapter 7, these two assumptions are in fact *equivalent* (cf. Corollary 7.2), which explains the similarity of the hierarchical algorithm (Algorithm 6.1) to previous butterfly algorithms [234]. However, this characterization is not straightforward, and to the best of our knowledge, such an equivalence has never been formally stated. Therefore, we will give a formal characterization of butterfly matrices using the complementary low-rank property in Chapter 7.

Novelties compared to previous butterfly algorithms. Yet, the proposed hierarchical algorithm (Algorithm 6.1) for square dyadic butterfly factorization possesses several novelties compared to previous butterfly algorithms [234]. First, the hierarchical algorithm comes with *exact recovery guarantees* for identifying the sparse butterfly factors from their product, which allows us to prove *essential uniqueness* of the square dyadic butterfly factorization (Theorem 6.1). Second, the hierarchical algorithm (Algorithm 6.1) works for any factor-bracketing trees. This generalizes existing butterfly algorithms [234, 250] that only consider *specific* hierarchical orders:

- The hierarchical order in the so-called rowwise butterfly factorization [40, 266, 289, 341] corresponds to the *right-to-left* factor-bracketing tree (defined as the binary tree where the right child of each non-leaf node is a singleton).
- The so-called columnwise butterfly factorization [250] corresponds to the analog *left-to-right* factor-bracketing tree (each left child is a singleton, which corresponds to the unbalanced tree illustrated in Figure 6.2d).
- The hybrid butterfly factorization [234, 250] corresponds to the *symmetric* factor-bracketing tree, which is the tree $\mathcal{T}$ illustrated in Figure 6.2c and defined for an even integer L as follows: (a) the children of the root node are $[\![1, L/2]\!]$ and $[\![L/2+1, L]\!]$; (b) in the left (resp. right) subtree of the root of $\mathcal{T}$, every right (resp. left) child of a non-leaf node is a singleton.

To the best of our knowledge, the *balanced* factor-bracketing tree (Figure 6.2b), defined for an integer L that is a power of two and constructed recursively in such a way that the children of the same parent have the same cardinal, has never been proposed in previous butterfly algorithms. Such a hierarchical order allows a parallelization of the proposed hierarchical algorithm.

Identifiability in multilinear inverse problems. The construction of the hierarchical algoritm (Algorithm 6.1) is based on an analysis of identifiability, i.e., essential uniqueness, in square dyadic butterfly factorization. Many identifiability results in multilinear inverse problems are derived from the so-called *lifting* procedure [61, 62, 230, 258, 259]. This procedure was originally used in the PhaseLift method [41, 42, 226] to address the phase retrieval problem. Other bilinear inverse problems, like blind-deconvolution [4, 12, 61, 194, 228–230] or self-calibration [242], have also been addressed using the lifting procedure. A general framework to analyze identifiability for any bilinear inverse problem has been given in [61]. Following the work from [214, Chapter 7], our analysis of identifiability in the case with $L = 2$ factors is a specialization of this general lifting procedure to the matrix factorization problem *with support constraints*. Identifiability results for $L \geq 3$ are more challenging as the usual lifting procedure from bilinear inverse problems cannot be directly leveraged. This multi-layer setting requires extending this procedure using a so-called tensorial lifting [257–259]. However, due to this multi-layer structure, conditions obtained via tensorial lifting might be difficult to verify in practice, although stable recovery conditions for convolutional linear networks have been derived [257, 259]. In contrast, our work proves identifiability results in matrix factorization with $L \geq 3$ factors *without using tensorial lifting*, using a hierarchical approach that reduces the analysis with multiple factors to the case with only two factors.

6.5 Experiments

We now demonstrate empirically the advantages of the proposed hierarchical factorization (Algorithm 6.1). All methods are implemented in Python and available in open source for reproducible research [385, 391]. An implementation of the algorithm in C++ via Python and Matlab wrappers is also provided by the FAµST 3.25 toolbox at `https://faust.inria.fr/`.

6.5.1 Outperforming iterative optimization methods

In [74], an *iterative optimization technique* is introduced to approximate a matrix by a product $\mathbf{BP}$, where $\mathbf{B}$ is a square dyadic butterfly matrix and $\mathbf{P}$ is a permutation matrix that belongs to a subset $\mathcal{P}$ of *eight* pre-chosen permutation matrices, that typically includes the *bit-reversal* permutation matrix (cf. Section 3.2.2). In other words, given a target matrix $\mathbf{A}$, the considered optimization problem is

$$\min_{(\mathbf{X}_\ell)_{\ell=1}^{L} \in \Sigma^\beta, \mathbf{P} \in \mathcal{P}} \|\mathbf{A} - \mathbf{X}_1 \dots \mathbf{X}_L \mathbf{P}\|_F. \tag{6.17}$$

This so-called *BP-model* [74] for addressing Problem (6.17) covers the DFT matrix and the Hadamard matrix, in the sense that the approximation error is null when $\mathbf{A}$ is one of these two matrices. The method from [74] consists of two steps:

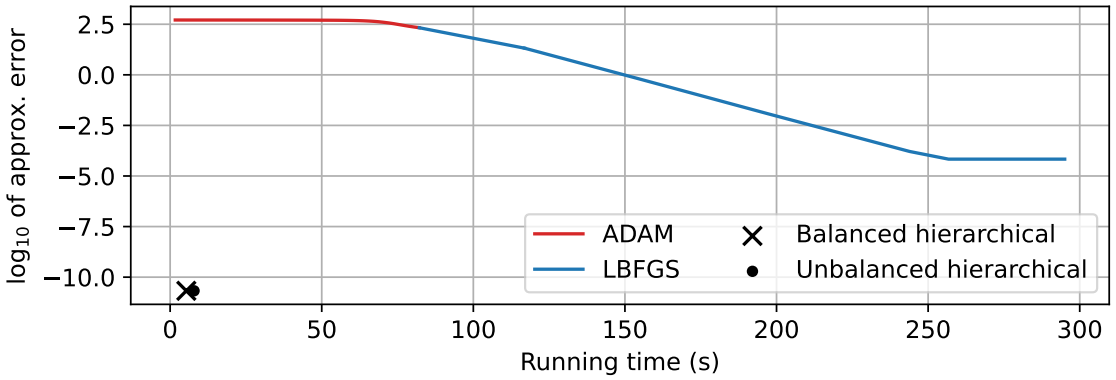


Figure 6.3: Approximation error (in Frobenius norm) of the DFT matrix of size 512 with 9 factors: iterative method [74] vs. hierarchical method (Algorithm 6.1). Cumulated running times at each iteration are shown for the former, total running time for the latter.

1. We optimize with Adam [197] over $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$ and $\tilde{\mathbf{P}}$ that is a relaxed, continuous parameterization of $\mathcal{P}$. After a fixed number of several iterations, we select a permutation $\mathbf{P} \in \mathcal{P}$ based on the values of $\tilde{\mathbf{P}}$. The optimization is re-run several times with different random initializations if it fails to find a good permutation $\mathbf{P} \in \mathcal{P}$.

2. The selected $\mathbf{P} \in \mathcal{P}$ is kept fixed and the LBFGS algorithm [244] is employed to optimize only the butterfly factors $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$ on several iterations.

Protocol. As an alternative, we apply the proposed hierarchical factorization method (Algorithm 6.1) to $\mathbf{A}\mathbf{P}^\top$ by fixing each of these eight permutations $\mathbf{P} \in \mathcal{P}$, since $\|\mathbf{A} - \mathbf{B}\mathbf{P}\|_F^2 = \|\mathbf{A}\mathbf{P}^\top - \mathbf{B}\|_F^2$ if $\mathbf{P}$ is a permutation matrix. Then, we choose the one with the lowest error. In comparison to [74], this leads to a factorization method that is free of hyperparameter tuning (learning rate, stopping criteria, etc.). We compare it to the iterative method [74] described above under the same setting as in [74]: Adam is run with 50 iterations[5], and LBFGS is run with 20 iterations. The target matrix $\mathbf{A}$ is either the DFT matrix of size $n \times n$ with $n = 2^L$ or a noisy version of it. We approximate $\mathbf{A}$ by a product of L butterfly factors.

Results. For the noiseless DFT matrix, Figure 6.3 shows that the hierarchical approach, whether balanced or unbalanced, gives both better precision and much shorter running time, with gains of several orders of magnitude. Similar results were obtained with the Hadamard matrix (not shown here).

We also compare the robustness of the approaches to noise. For the noisy DFT matrix, Figure 6.4 shows the instability of the iterative method [74], as it can give poor approximation of the noisy DFT matrix over several factorization experiments. In contrast, the proposed method is not only faster, but also finds more

[5]We found out that performing more than 50 iterations yields similar precision.

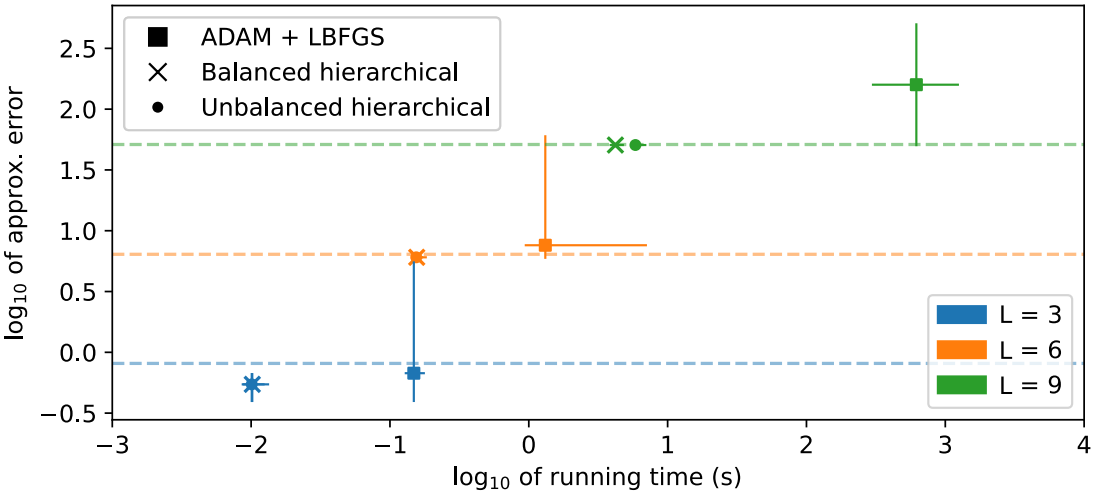


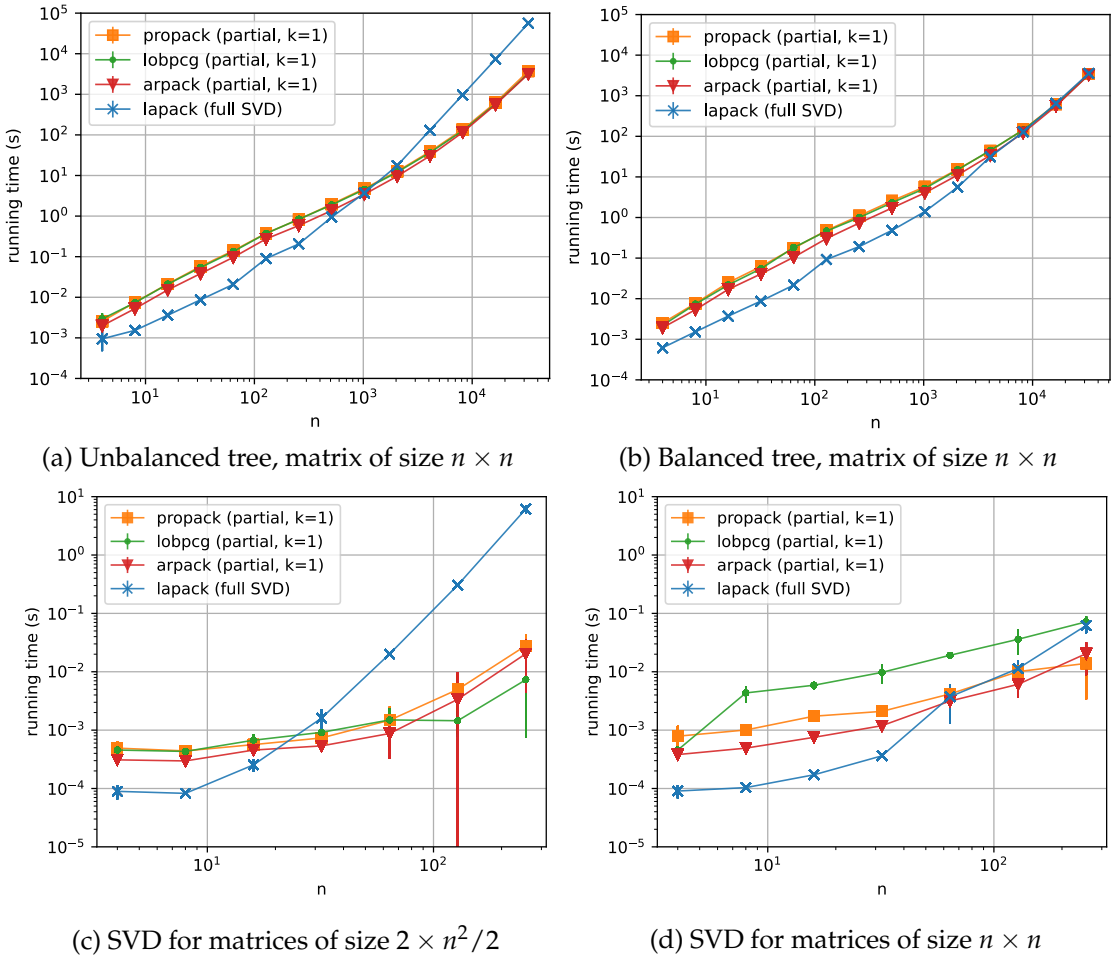
Figure 6.4: Approximation error (in Frobenius norm) of a noisy DFT matrix of size 2^L: iterative method [74] vs. hierarchical method (Algorithm 6.1). The added noise matrix has i.i.d. complex centered Gaussian entries with variance $\sigma^2 = 0.005$ for both the real and imaginary part. Markers show the average running time and approximation error. Error bars (almost invisible for the hierarchical methods) show the extreme values over 10 different realizations of the noise matrix. Horizontal dashed lines show the average of noise matrix norms.

reliably a good approximation of the noisy DFT matrix, with an approximation error of the same order of magnitude as the norm of the noise matrix. This is true for both the balanced and the unbalanced trees. These empirical observations suggest that Algorithm 6.1 has a certain stability with respect to noise. We will study guarantees on approximation error of the hiearchical algorithm in Chapter 7.

6.5.2 Benchmarking SVD solvers in the hierarchical algorithm

The theoretical complexity of the hierarchical algorithm (Algorithm 6.1) is $\mathcal{O}(n^2)$ for a target matrix of size $n \times n$, and the hierarchical method is shown empirically to be faster than gradient-based optimization [74] in the previous paragraph. We now study the *scaling laws* of Algorithm 6.1 with respect to the size n. We will show that it is important to use an appropriate implementation of SVD in order to achieve a fast implementation.

Protocol. We implement Algorithm 6.1 in Python using the `Scipy 1.8.0` package to compute the SVD for the best rank-one approximation. We measure the running time of our implementation of Algorithm 6.1 to approximate $\mathbf{A} = \mathbf{H} + \sigma\mathbf{W}$ as a product of L butterfly factors, where $\mathbf{H}$ is the Hadamard matrix of size $n = 2^L$ with $L \in \{2, \ldots, 15\}$, $\mathbf{W}$ is a random matrix with i.i.d. standard Gaussian entries (zero mean and variance equal to 1), and $\sigma = 0.01$. We use different *SVD solvers* in our experiments, among: LAPACK, ARPACK, PROPACK, LOBPCG.



(a) Unbalanced tree, matrix of size $n \times n$

(b) Balanced tree, matrix of size $n \times n$

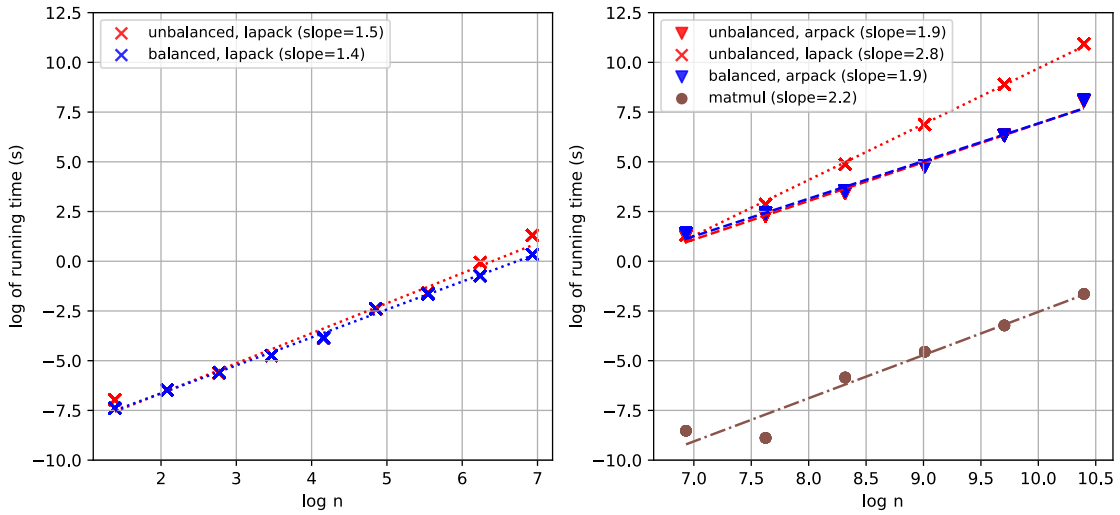
(c) SVD for matrices of size $2 \times n^2/2$

(d) SVD for matrices of size $n \times n$

Figure 6.5: Comparing different solvers - running time of our implementation of Algorithm 6.1. For a given matrix size, each factorization (resp. SVD) is repeated 3 (resp. 10) times in order to plot the mean running time with a vertical error bar showing the standard deviation.

The first one performs a full SVD before truncating it, while the last ones perform partial SVD at a given order k ($k = 1$ in our case). Experiments are run on an Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz.

Comparing different solvers. We compare the running time of Algorithm 6.1 implemented with different SVD solvers. In Figure 6.5a, the unbalanced hierarchical factorization is faster with LAPACK for $n \leq 1024$, while the other solvers are faster for $n \geq 1024$. In Figure 6.5b, the balanced hierarchical factorization is faster with LAPACK for $n \leq 4096$, and all the solvers have a similar running time for $n \geq 4096$. The difference in running time can be empirically explained by our benchmark of these SVD solvers on a rectangular matrix (size $2 \times n^2/2$) in Figure 6.5c, and on a square matrix (size $n \times n$) in Figure 6.5d. These matrix sizes correspond to the size of submatrices on which an SVD is performed in the hier-



(a) Small matrix size $n \times n$

(b) Large matrix size $n \times n$

Figure 6.6: Comparing balanced and unbalanced trees - running time of our implementation of Algorithm 6.1, in logarithmic scale. For better visualization a least-square regression is performed for each set of measurements, and we report the estimated slope a of the regression line $Y = aX + b$. For slope comparison, we measure the running time for matrix-vector multiplication, computed with `numpy.matmul` method from `Numpy 1.22.3`.

archical algorithm. We indeed observe that the full SVD with LAPACK is faster than partial SVDs with ARPACK, PROPACK, LOBPCG for lower matrix size.

Comparing balanced and unbalanced tree. We now compare the running time of the hierarchical factorization algorithm with the unbalanced and balanced tree. In Figure 6.6a, the comparison is performed in a setting with small matrix size with $n \leq 1024$, using LAPACK to compute full SVDs. Hierarchical factorization is slightly faster with a balanced tree compared to an unbalanced tree. In Figure 6.6b, the comparison is performed in a setting with large matrix size with $n \geq 1024$. When using ARPACK, the running time for hierarchical factorization is the same for both trees. This is coherent with our analysis of complexity bounds for Algorithm 6.1, which is $\mathcal{O}(n^2)$ for both trees. For completeness Figure 6.6b also compares the running time of hierarchical factorization with the one of matrix-vector multiplication for a matrix of size $n \times n$, whose complexity is also $\mathcal{O}(n^2)$. Finally we see in Figure 6.6b that using an unbalanced tree with LAPACK for large matrix size is not optimal.

6.6 Conclusion

We established *hierarchical identifiability* in the square dyadic butterfly factorization. We proved that the butterfly factors $(\mathbf{X}_\ell)_{\ell=1}^L$ in the product $\mathbf{A} := \mathbf{X}_1 \dots \mathbf{X}_L$ of size $n \times n$ with $n := 2^L$ can be recovered up to scaling ambiguities from the obser-

vation of $\mathbf{A}$, with a hierarchical factorization method described by Algorithm 6.1 that is endowed with *exact recovery guarantees*, and has *controlled time complexity* of $\mathcal{O}(n^2)$. This chapter opens different perspectives.

Stability results. Our experiments suggests that the hierarchical algorithm (Algorithm 6.1) admit some form of stability or with respect to additive noise on the target matrix. In particular, it would be interesting to determine if the stability of the algorithm to noise is also similar for different factor-bracketing trees (e.g., unbalanced vs. balanced tree). Studying the approximation error of the hierarchical algorithm is a natural continuation of this chapte, which is done in Chapter 7. More specifically, we will construct an new hierarchical algorithm (cf. Algorithm 7.5) that outputs butterfly factors $(\hat{\mathbf{X}}_\ell)_{\ell=1}^L \in \Sigma^\beta$ that are guaranteed to satisfy the bound:

$$\|\mathbf{A} - \hat{\mathbf{X}}_1 \dots \hat{\mathbf{X}}_L\|_F \leq C_\beta \min_{(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta} \|\mathbf{A} - \hat{\mathbf{X}}_1 \dots \hat{\mathbf{X}}_L\|_F, \tag{6.18}$$

where $\mathbf{A}$ is any target matrix of size $n \times n$ with $n := 2^L$, and $C_\beta \geq 1$ is a constant that only depends on L. This implies that new hierarchical algorithm is optimal when the target $\mathbf{A}$ is assumed to admit exactly a square dyadic butterfly factorization (noiseless setting). The result about *essential uniqueness* (Theorem 6.1) is complementary to this stability result: it *also* shows that the optimal butterfly factors $(\hat{\mathbf{X}}_\ell)_{\ell=1}^L \in \Sigma^\beta$ in the noiseless setting are essentially unique.

Implementation of the hierarchical algorithm. Our benchmark of SVD solvers in Figures 6.5c and 6.5d suggests that our implementation of the hierarchical algorithm can be further improved by choosing the optimal SVD solver depending on the block's dimension. One could also envision randomized algorithms for fast low-rank approximation [99, 153] in the spirit of [234].

Identifiability of sparse matrix factorization with $L = 2$ factors. Our analysis of identifiability using the hierarchical approach relies on identifiability results in the case with $L = 2$ factors. This motivates further explorations of identifiability conditions in this setting. Our work focuses on fixed-support constraints, and we considered in Proposition 6.1 the case of disjoint rank-one supports. More relaxed conditions can be envisioned, such as supports satisfying the so-called *complete equivalence class* condition introduced in [212]. Beyond the fixed-support setting, one can also explore essential uniqueness by considering a *family* of sparsity patterns like in [390]. One can then use results from matrix completion literature [67, 94, 198] to establish more elaborate conditions for identifiability in the case with two factors.

Chapter 7

Butterfly factorization with guarantees on approximation error

This chapter extends the results of Chapter 6, by proving new *theoretical guarantees* for decomposition algorithms associated with butterfly factorization, beyond the specific case of the square dyadic butterfly factorization and the noiseless setting.

7.1 Introduction

In Chapter 3, we saw that many matrices appearing in scientific computing problems, like kernel matrices associated with special function transforms [288,369] or Fourier integral operators [40,77,235], satisfy a certain low-rank property, named the *complementary low-rank* property [234]: it has been shown that if specific submatrices of a target matrix $\mathbf{A}$ of size $n \times n$ are numerically low-rank, then $\mathbf{A}$ can be compressed by successive hierarchical low-rank approximations of these submatrices, in the sense that it can be approximated by a sparse factorization

$$\hat{\mathbf{A}} = \mathbf{X}_1 \dots \mathbf{X}_L$$

with $L = \mathcal{O}(\log n)$ factors $\mathbf{X}_\ell$ having at most $\mathcal{O}(n)$ nonzero entries for each $\ell \in [\![L]\!] := \{1, \dots, L\}$. This sparse factorization, called in general *butterfly factorization*, would then yield a fast algorithm for the approximate evaluation of the matrix-vector multiplication by $\mathbf{A}$, in $\mathcal{O}(n \log n)$ complexity.

As detailed in Section 4.5, an alternative definition of the butterfly factorization, typically used in deep learning applications, refers to a sparse matrix factorization with *specific* constraints on the sparse factors. According to [72,74,75,213, 241,392], a matrix $\mathbf{A}$ admits a certain butterfly factorization if, up to some row and column permutations, it can be factorized into a certain number of factors

The material of this chapter is based on an on-going work, in collaboration with Quoc-Tung Le, Elisa Riccietti and Rémi Gribonval.

$\mathbf{X}_1, \ldots, \mathbf{X}_L$ for a prescribed number $L \geq 2$, such that each factor $\mathbf{X}_\ell$ for $\ell \in [\![L]\!]$ satisfies a certain *fixed-support* constraint, i.e., the support of $\mathbf{X}_\ell$, denoted $\mathrm{supp}(\mathbf{X}_\ell)$, is included in the support of a prescribed binary matrix $\mathbf{S}_\ell$. The different existing butterfly factorizations only vary by their number of factors L, and their choice of binary matrices $\mathbf{S}_1, \ldots, \mathbf{S}_L$. Let us recall some examples of such factorizations.

1. **Square dyadic butterfly factorization** [74, 75, 213, 392]. Recall that this was defined in Definition 6.2, for matrices of size $n \times n$ where n is a power of two. The number of factors is $L := \log_2 n$. For $\ell \in [\![L]\!]$, the factor $\mathbf{X}_\ell$ is of size $n \times n$, and satisfies the support constraint $\mathrm{supp}(\mathbf{X}_\ell) \subseteq \mathrm{supp}(\mathbf{S}_\ell)$, where

$$\forall \ell \in [\![L]\!], \quad \mathbf{S}_\ell := \mathbf{I}_{2^{\ell-1}} \otimes \mathbf{1}_{2\times 2} \otimes \mathbf{I}_{n/2^\ell}.$$

2. **Monarch factorization** [72]. A Monarch factorization parameterized by two integers p, q decomposes a matrix $\mathbf{A}$ of size $m \times n$ into $L := 2$ factors $\mathbf{X}_1, \mathbf{X}_2$ such that $\mathrm{supp}(\mathbf{X}_\ell) \subseteq \mathrm{supp}(\mathbf{S}_\ell)$ for $\ell = 1, 2$ where

$$\mathbf{S}_1 := \mathbf{1}_{p\times q} \otimes \mathbf{I}_{\frac{m}{p}}, \quad \mathbf{S}_2 := \mathbf{I}_q \otimes \mathbf{1}_{\frac{m}{p}\times \frac{n}{q}}.$$

   Here, we assume that p, q divides m, n respectively. The DFT matrix of size $n \times n$ admits such a factorization for $p = q$, up to a column permutation. Indeed, according to the Cooley-Tukey algorithm, computing the discrete Fourier transform of size n is equivalent to performing p discrete Fourier transforms of size n/p first, and then n/p discrete Fourier transforms of size p, see, e.g., equations (14) and (21) in [89].

3. **Deformable butterfly factorization** [241]. Previous conventional butterfly factorizations can be generalized as follows. Given an integer $L \geq 2$, a matrix $\mathbf{A}$ admits a deformable butterfly factorization parameterized by a list of tuples $(p_\ell, q_\ell, r_\ell, s_\ell, t_\ell)_{\ell=1}^L$ if $\mathbf{A} = \mathbf{X}_1 \ldots \mathbf{X}_L$ where each factor $\mathbf{X}_\ell$ for $\ell \in [\![L]\!]$ is of size $p_\ell \times q_\ell$ and has a support included in $\mathrm{supp}(\mathbf{S}_\ell)$, defined as:

$$\forall \ell \in [\![L]\!], \quad \mathbf{S}_\ell := \mathbf{I}_{\frac{p_\ell}{r_\ell t_\ell}} \otimes \mathbf{1}_{r_\ell \times s_\ell} \otimes \mathbf{I}_{t_\ell}.$$

   Here, it is assumed that $\frac{p_\ell}{r_\ell t_\ell} = \frac{q_\ell}{s_\ell t_\ell}$ is an integer, for each $\ell \in [\![L]\!]$.

In all these examples the fixed-support constraint on each butterfly factor $\mathbf{X}$ takes the form $\mathrm{supp}(\mathbf{X}) \subseteq \mathrm{supp}(\mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d)$ for some integer parameters (a, b, c, d). Figure 7.1 illustrates the sparsity pattern $\mathbf{S}_{\boldsymbol{\pi}} := \mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d$ of a butterfly factor associated with the tuple $\boldsymbol{\pi} = (a, b, c, d)$, that we call a *pattern*. One of the main benefits of choosing such fixed-support constraints *instead of an arbitrary sparse support* is its *block structure* that could enable efficient implementation on specific hardware like Intelligence Processing Unit (IPU) [321] or GPU [72, 74], with practical speed-up for matrix multiplication. See Chapter 9 for a discussion about GPU implementations for butterfly sparse matrix multiplication.

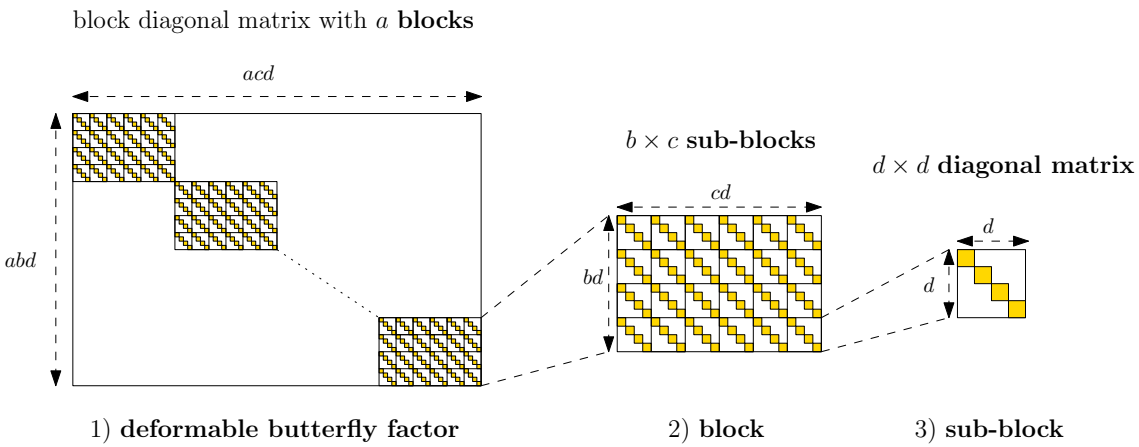


Figure 7.1: Illustration of the support of a butterfly factor with pattern $\boldsymbol{\pi} = (a, b, c, d)$. The yellow squares indicate the indices belonging to the support. The sub-figures (1), (2), (3) illustrate respectively the concepts of factor, block and sub-block .

Problem formulation. This chapter focuses on the problem of approximating a target matrix $\mathbf{A}$ by a product of butterfly factors associated with a given *architecture* $\boldsymbol{\beta} = (\boldsymbol{\pi}_\ell)_{\ell=1}^{L}$:

$$E^{\boldsymbol{\beta}}(\mathbf{A}) := \inf_{(\mathbf{X}_\ell)_{\ell=1}^{L}} \|\mathbf{A} - \mathbf{X}_1 \dots \mathbf{X}_L\|_F^2 = \inf_{\mathbf{B}} \|\mathbf{A} - \mathbf{B}\|_F^2, \tag{7.1}$$

where $\mathbf{B}$ is a butterfly matrix and each $\mathbf{X}_\ell$ is a butterfly factor with sparsity pattern prescribed by $\boldsymbol{\pi}_\ell$, and $\|\cdot\|_F$ is the Frobenius norm. We will call these factors "deformable butterfly factors" because the description of the prescribed supports above is equivalent to that of [241]. Several methods have been proposed to address this butterfly factorization problem, but we argue that they either lack guarantees of success, or only have partial guarantees. We fix this issue here by introducing a new hierarchical algorithm endowed with theoretical guarantees.

Contributions. More precisely, the main contributions of this chapter are:

1. To introduce a formal mathematical description of the "deformable butterfly factors" introduced in [241]. While we owe [241] the original idea of extending previous butterfly factorizations, the mathematical formulation of the prescribed supports as Kronecker products is a novelty that allows a theoretical study of the corresponding butterfly factorization, as done in this chapter. Moreover, our parameterization uses 4 parameters and removes the redundancy in the original 5-parameter description of deformable butterfly factors of [241]. Table 7.1 summarizes the main characteristics of existing butterfly architectures covered by our framework.

2. To define the *chainability* of an architecture $\boldsymbol{\beta}$ (Definition 7.5), and to prove that Problem (7.1) admits an optimum when $\boldsymbol{\beta}$ is chainable (Corollary 7.3).

Table 7.1: Existing architectures β in the literature. (†) Note that [241] did not explicitly state constraints on $(a_\ell, b_\ell, c_\ell, d_\ell)_{\ell=1}^L$ for deformable butterfly factorization, because they use an alternative description of the sparsity patterns of the butterfly factors. In fact, they only consider variants of butterfly factorization corresponding to chainable architectures (cf. Definition 7.5 below) with $\mathbf{q}(\beta) = (1, \dots, 1)$.

Architectures	**Size**	$\lvert\beta\rvert$	**Values of** $\beta = (\pi_\ell)_\ell$	**Chainable?**
Low-rank matrix	$m \times n$	2	$(1, m, r, 1), (1, r, n, 1)$	Yes
Square dyadic butterfly [74]	$2^L \times 2^L$	L	$(2^{\ell-1}, 2, 2, 2^{L-\ell})_{\ell=1}^L$	Yes
Monarch [72]	$m \times n$	2	$(1, p, q, m/p), (q, m/p, n/q, 1)$	Yes
Deformable butterfly$^{(\dagger)}$ [241]	$m \times n$	L	$(a_\ell, b_\ell, c_\ell, d_\ell)_{\ell=1}^L$	Yes
Kaleidoscope [75]	$2^L \times 2^L$	$2L$	$\pi_\ell = \begin{cases} (2^{\ell-1}, 2, 2, 2^{L-\ell}) & \text{if } \ell \leq L \\ (2^{2L-\ell}, 2, 2, 2^{\ell-L-1}) & \text{if } \ell > L \end{cases}$	No

3. To *characterize analytically* the set of butterfly matrices with architecture β,

$$\mathcal{B}^\beta := \{\mathbf{X}_1 \dots \mathbf{X}_L \mid \operatorname{supp}(\mathbf{X}_\ell) \subseteq \operatorname{supp}(\mathbf{S}_{\pi_\ell})\, \ell \in [\![L]\!]\}\,, \tag{7.2}$$

   for a chainable β, in terms of low-rank properties of certain submatrices of $\mathbf{A}$ (Corollary 7.2) which are equivalent to a generalization of the complementary low-rank property (Corollary C.2).

4. To define the *redundancy* of a chainable architecture (cf. Definition 7.6). Intuitively, a chainable architecture β is redundant if one can replace it with a "compressed" (non-redundant) one β' such that $\mathcal{B}^\beta = \mathcal{B}^{\beta'}$ (cf. Proposition 7.2). Thus, from the acceleration of linear operators viewpoint, redundant architectures have no practical interest.

5. To propose a *new hierarchical algorithm* (Algorithm 7.5) able to provide an approximate solution to Problem (7.1) *for non-redundant chainable architectures*. Compared to previous similar algorithms, this algorithm introduces a new orthogonalization step which is key to obtain approximation guarantees. The algorithm can be readily extended to redundant chainable architectures, under the same theoretical guarantee (see Algorithm 7.7).

6. To prove that, for chainable β, Algorithm 7.5 outputs butterfly factors $(\hat{\mathbf{X}}_\ell)_{\ell=1}^L$ such that

$$\|\mathbf{A} - \hat{\mathbf{X}}_1 \dots \hat{\mathbf{X}}_L\|_F \leq C_\beta \cdot \inf_{(\mathbf{X}_\ell)_{\ell=1}^L} \|\mathbf{A} - \mathbf{X}_1 \dots \mathbf{X}_L\|_F, \tag{7.3}$$

   where $C_\beta \geq 1$ depends *only* on β (Corollary 7.1), see Table 7.2 for examples. To the best of our knowledge, this is the first time such a *near-optimality bound* is established for a butterfly approximation algorithm.

Table 7.2: The approximation ratio C_β (see Equation (7.3)) of Algorithm 7.5 with a selection of chainable architectures β from Table 7.1.

Parameterization	**Size**	$\lvert\beta\rvert$	C_β **in (7.17) - Corollary 7.1**	C_β **in (7.18) - Corollary 7.1**
Low rank matrix	$m \times n$	2	1	1
Monarch [72]	$m \times n$	2	1	1
Square dyadic butterfly [74]	$n \times n$	$\log n$	$n/2 - 1 = \mathcal{O}(n)$	$\frac{\sqrt{2}}{3}\sqrt{n^{\log 3} - 1} = \mathcal{O}(n^{0.7925})$

Outline. Section 7.2 discusses related work. Section 7.3 introduces some preliminaries on two-factor matrix factorization with fixed-support constraints. This is also where we setup our general notations. Section 7.4 formalizes the definition of deformable butterfly factorization associated with β, and introduces the *chainability* and *non-redundancy* conditions for an architecture β, that will be at the core of the proof of error guarantees on our proposed hierarchical algorithm. Section 7.5 extends an existing hierarchical algorithm, currently expressed only for dyadic butterfly factorization, to any chainable β. For non-redundant chainable β, Section 7.6 introduces novel orthonormalization operations in the proposed hierarchical algorithm. This allows to establish in Section 7.7 our main results on the control of the approximation error and the low-rank characterization of butterfly matrices associated with chainable β. Section 7.8 proposes some numerical experiments about the proposed hierarchical algorithm. The most technical proofs are deferred to Appendix C.

7.2 Related work

Several methods have been proposed to address the butterfly factorization problem (7.1), but we argue that they either lack guarantees of success, or only have partial guarantees.

First-order methods. Optimization methods based on gradient descent [74] or alternating least squares [241] are not suitable for Problem (7.1) and lack guarantees of success, because of the non-convexity of the objective function. In fact, the problem of approximating a given matrix by the product of factors with fixed-support constraints, as it is the case for (7.1), is generally NP-hard and might even lead to numerical instability even for $L = 2$ factors, as shown in [212]. In contrast, we show that the minimum of (7.1) always exist for chainable β.

Butterfly algorithms and the complementary low-rank property. This paragraph summarizes some material from Chapter 3 related to butterfly factoriza-

tion. Butterfly algorithms [40, 234, 235, 250, 265, 266, 288] look for an approximation of a target matrix $\mathbf{A}$ by a sparse factorization $\hat{\mathbf{A}} = \mathbf{X}_1 \dots \mathbf{X}_L$, assuming that $\mathbf{A}$ satisfies the so-called *complementary low-rank property*, formally introduced in [234]. This low-rank property assumes that the rank of certain submatrices of $\mathbf{A}$ restricted to some specific blocks is numerically low and that these blocks satisfy some conditions described by a hierarchical partitioning of the row and column indices, using the notion of *cluster tree* [150]. Then, the butterfly algorithm leverages this low-rank property to approximate the target matrix by a data-sparse representation, by performing successive low-rank approximation of specific submatrices. The literature in numerical analysis describes many linear operators associated with matrices satisfying the complementary low-rank property, such as kernel matrices encountered in electromagnetic or acoustic scattering problems [146,265,266], special function transforms [289], spherical harmonic transforms [341] or Fourier integral operators [40,233,235,236,369].

The formal definition of the complementary low-rank property currently given in the literature only considers cluster trees that are dyadic [234] or quadtrees [236]. In this work, we give a more general definition of the complementary low-rank property that considers arbitrary cluster trees. To the best of our knowledge, this allows us to give the first formal characterization of the set of matrices admitting a (deformable) butterfly factorization associated with an architecture β, as defined in (7.2), using this extended definition of the complementary low-rank property. In particular, this shows that the definition in (7.2) is more general than the previous definitions of the complementary low-rank property that were restricted to dyadic trees or quadtrees [234,236].

Existing error bounds for butterfly algorithms. Several existing butterfly algorithms [40,234,235,288] are guaranteed to provide an approximation error $\|\mathbf{A} - \hat{\mathbf{A}}\|_F$ equal to zero, when $\mathbf{A}$ satisfies *exactly* the complementary low-rank property, i.e., the best low-rank approximation errors of the submatrices described by the property are *exactly* zero [234, 288]. However, when these submatrices are only approximately low-rank (with a positive best low-rank approximation error), existing butterfly factorization algorithms *are not* guaranteed to provide an approximation $\hat{\mathbf{A}}$ with the *best* approximation error. To the best of our knowledge, the only existing error bound in the literature [250] is based on a butterfly algorithm that performs successive low-rank approximation of blocks $\mathbf{M}$, and takes the form

$$\|\mathbf{A} - \hat{\mathbf{A}}\|_F^2 \leq C_n \epsilon_0^2 \|\mathbf{A}\|_F^2, \quad \text{with } C_n = \mathcal{O}(\log n), \tag{7.4}$$

where $\mathbf{A}$ is an $n \times n$ matrix and ϵ_0 is the maximum relative error $\|\mathbf{M} - \hat{\mathbf{M}}\|_F / \|\mathbf{M}\|_F$ across all blocks $\mathbf{M}$ on which the algorithm performs low-rank approximation, with $\hat{\mathbf{M}}$ a best low-rank approximation of $\mathbf{M}$. Even though the constant C_n in the error bound grows slowly with respect to the matrix size n, the bound (7.4) is not satisfying for the following reasons: (i) the quantity ϵ_0 can only be determined algorithmically *after* applying the butterfly algorithm on the target matrix $\mathbf{A}$; (ii) it does not compare the approximation error $\|\mathbf{A} - \hat{\mathbf{A}}\|_F^2$ to the *best* approximation

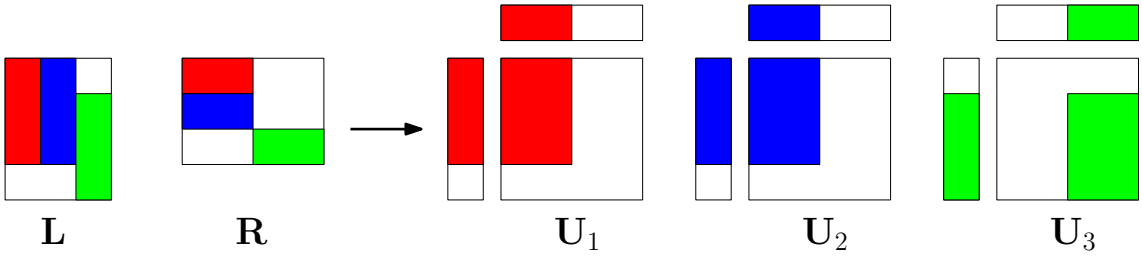


Figure 7.2: An example of support constraints $(\mathbf{L}, \mathbf{R})$ and the supports of the corresponding rank-one contributions. Colored parts indicate indices inside the support constraints $\mathbf{L}, \mathbf{R}$ and $\mathbf{U}_i$ for $i \in [\![3]\!]$. $\{1, 2\}$ and $\{3\}$ are the two equivalence classes (Definition 7.1).

error, that is, the minimal error $\|\mathbf{A} - \mathbf{A}^*\|_F$ with $\mathbf{A}^*$ satisfying *exactly* the complementary low-rank property. In this chapter, we propose the first error bound for butterfly factorization that compares the approximation error to the minimal approximation error, cf. (7.3). We compare our new error bound to (7.4) in Section 7.7.2.

7.3 Two-factor, fixed-support matrix factorization

Following the hierarchical approach of Chapter 6, our analysis of the butterfly factorization problem (7.1) with *multiple* factors in general ($L \geq 2$) relies on the analysis of the simplest setting with only $L = 2$ factors. This setting is studied in [212]. We recall some important results that will be used in the rest of the chapter.

Given two binary matrices $\mathbf{L}, \mathbf{R}$, the problem of *fixed-support matrix factorization* (FSMF) with two factors is formulated as:

$$\inf_{(\mathbf{X},\mathbf{Y})} \|\mathbf{A} - \mathbf{X}\mathbf{Y}\|_F^2, \text{ with } \operatorname{supp}(\mathbf{X}) \subseteq \mathbf{L}, \operatorname{supp}(\mathbf{Y}) \subseteq \mathbf{R}. \tag{7.5}$$

While Problem (7.5) is NP-hard[1] in general [212, Theorem 2.4], it becomes tractable under certain conditions on $(\mathbf{L}, \mathbf{R})$. To describe one of these conditions, we rely on the lifting procedure introduced in Section 6.2. For two binary matrices $\mathbf{L} \in \{0,1\}^{m \times r}, \mathbf{R} \in \{0,1\}^{r \times n}$, denote $(\mathbf{U}_i)_{i=1}^r$ the corresponding rank-one contribution supports (see for Figure 7.2 for an illustration), defined as:

$$(\mathbf{U}_i)_{i=1}^r := \varphi(\mathbf{L}, \mathbf{R}) := (\mathbf{L}[:, i]\mathbf{R}[i, :])_{i=1}^r, \tag{7.6}$$

where φ denotes the lifting operator defined in (6.10).

Remark 7.1. *The binary matrix* $\mathbf{L}[:, i]\mathbf{R}[i, :]$ *for* $i \in [\![r]\!]$ *encodes the* support constraint *of* $\mathbf{X}[:, i]\mathbf{Y}[i, :]$ *for each* $(\mathbf{X}, \mathbf{Y})$ *such that* $\operatorname{supp}(\mathbf{X}) \subseteq \mathbf{L}, \operatorname{supp}(\mathbf{Y}) \subseteq \mathbf{R}$.

[1] and does not always admit an optimum: the infimum may not be achieved [212, Remark A.1].

The rank-one supports $(\mathbf{U}_i)_{i=1}^r$ defines an equivalence relation and its induced equivalence classes on the set of column indices $[\![r]\!]$, as illustrated in Figure 7.2.

Definition 7.1 (Equivalence classes of rank-one supports, representative rank-one supports [212]). *Given* $\mathbf{L} \in \{0,1\}^{m\times r}, \mathbf{R} \in \{0,1\}^{r\times n}$, *denoting* $(\mathbf{U}_i)_{i=1}^r = \varphi(\mathbf{L},\mathbf{R})$, *define an equivalence relation on* $[\![r]\!]$ *as:*

$$i \sim j \iff \mathbf{U}_i = \mathbf{U}_j.$$

This yields a partition of $[\![r]\!]$ *into equivalence classes, denoted* $\mathcal{P}(\mathbf{L},\mathbf{R})$. *For each* $P \in \mathcal{P}(\mathbf{L},\mathbf{R})$, *denote* $\mathbf{U}_P$ *a representative rank-one support,* $R_P \subseteq [\![m]\!]$ *and* $C_P \subseteq [\![n]\!]$ *the supports of rows and columns in* $\mathbf{U}_P$, *respectively, i.e.,* $\operatorname{supp}(\mathbf{U}_P) = R_P \times C_P$, *and denote* $|P|$ *the cardinal of the equivalence class* P.

We now recall a sufficient condition on $(\mathbf{L},\mathbf{R})$ for which corresponding instances of Problem (7.5) can be solved in polynomial time via Algorithm 7.1.

Theorem 7.1 (Tractable support constraints of Problem (7.5) [212, Theorem 3.3]). *If all elements of* $\varphi(\mathbf{L},\mathbf{R})$ *are pairwise disjoint or identical, then Algorithm 7.1 yields an optimal solution of Problem (7.5). In addition, the infimum of Problem (7.5) is given by:*

$$\inf_{\operatorname{supp}(\mathbf{X})\subseteq\mathbf{L},\operatorname{supp}(\mathbf{Y})\subseteq\mathbf{R}} \|\mathbf{A}-\mathbf{X}\mathbf{Y}\|_F^2 = \sum_{P\in\mathcal{P}(\mathbf{L},\mathbf{R})} \min_{\mathbf{B},\operatorname{rank}(\mathbf{B})\le|P|} \|\mathbf{A}[R_P,C_P]-\mathbf{B}\|_F^2 + c, \tag{7.7}$$

where $c := \sum_{(i,j)\notin\operatorname{supp}(\mathbf{L}\mathbf{R})} \mathbf{A}[i,j]^2$ *is a constant depending only on* $(\mathbf{A},\mathbf{L},\mathbf{R})$.[a]

[a] Note that $\mathbf{L}\mathbf{R}$ is a product of two binary matrices.

The equation (7.7) was not proved in [212], so we provide a complete proof of Theorem 7.1 in Appendix C.1. The main idea is the following.

Sketch of proof. For $(\mathbf{X},\mathbf{Y})$ such that $\operatorname{supp}(\mathbf{X}) \subseteq \mathbf{L}$ and $\operatorname{supp}(\mathbf{Y}) \subseteq \mathbf{R}$:

$$\|\mathbf{A}-\mathbf{X}\mathbf{Y}\|_F^2 = \sum_{P\in\mathcal{P}(\mathbf{L},\mathbf{R})} \|\mathbf{A}[R_P,C_P]-\mathbf{X}[R_P,P]\mathbf{Y}[P,C_P]\|_F^2 + c, \tag{7.8}$$

because $\varphi(\mathbf{L},\mathbf{R})$ are pairwise disjoint or identical, by assumption. Thus, minimizing the left-hand-side is equivalent to minimize each summand in the right-hand side, which is equivalent to finding the best rank-$|P|$ approximation of the matrix $\mathbf{A}[R_P,C_P]$ for each $P \in \mathcal{P}(\mathbf{L},\mathbf{R})$. □

Remark 7.2. *Best low-rank approximation in line 3 of Algorithm 7.1 can be computed via truncated SVD. Note that the definition of* $\hat{\mathbf{H}},\hat{\mathbf{K}}$ *in this line is not unique, because, for instance, the product* $\hat{\mathbf{H}}\hat{\mathbf{K}}$ *is invariant to some rescaling of columns and rows.*

Algorithm 7.1 Two-factor fixed-support matrix factorization, under conditions of Theorem 7.1

Require: $\mathbf{A} \in \mathbb{C}^{m\times n}$, $\mathbf{L} \in \{0,1\}^{m\times r}$, $\mathbf{R} \in \{0,1\}^{r\times n}$
Ensure: $(\mathbf{X},\mathbf{Y}) \in \mathbb{C}^{m\times r} \times \mathbb{C}^{r\times n}$ such that $\mathrm{supp}(\mathbf{X}) \subseteq \mathbf{L}$, $\mathrm{supp}(\mathbf{Y}) \subseteq \mathbf{R}$
1: $(\mathbf{X},\mathbf{Y}) \leftarrow (\mathbf{0}_{m\times r}, \mathbf{0}_{r\times n})$
2: **for** $P \in \mathcal{P}(\mathbf{L},\mathbf{R})$ **do**
3: $\quad (\mathbf{X}[R_P,P], \mathbf{Y}[P,C_P]) \leftarrow (\hat{\mathbf{H}},\hat{\mathbf{K}}) \in \underset{\substack{\mathbf{H}\in\mathbb{C}^{|R_P|\times|P|}\\ \mathbf{K}\in\mathbb{C}^{|P|\times|C_P|}}}{\arg\min} \|\mathbf{A}[R_P,C_P] - \mathbf{H}\mathbf{K}\|_F$
4: **end for**
5: **return** $(\mathbf{X},\mathbf{Y})$

7.4 Deformable butterfly factorization

This section presents a mathematical formulation of the deformable butterfly factorization [241] associated with a sequence of patterns $\beta := (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ called an architecture. We then introduce the notions of *chainability* and *non-redundancy* of an architecture, that are crucial conditions for constructing a hierarchical algorithm for Problem (7.1) with error guarantees.

7.4.1 A mathematical formulation for (deformable) butterfly factors

Many butterfly factorizations [72,74,75,213,241,392] take the form $\mathbf{A} = \mathbf{X}_1 \dots \mathbf{X}_L$ with $\mathrm{supp}(\mathbf{X}_\ell) \subseteq \mathbf{I}_{a_\ell} \otimes \mathbf{1}_{b_\ell\times c_\ell} \otimes \mathbf{I}_{d_\ell}$ for $\ell \in [\![L]\!]$, for some parameters $(a_\ell,b_\ell,c_\ell,d_\ell)_{\ell=1}^L$, cf. Section 7.1. We therefore introduce the following definition.

Definition 7.2 (Butterfly factors and their sparsity patterns)**.** *For $a,b,c,d \in \mathbb{N}$, a* (deformable) butterfly factor *of pattern $\boldsymbol{\pi} := (a,b,c,d)$ (or $\boldsymbol{\pi}$-factor) is a matrix in $\mathbb{R}^{m\times n}$ or $\mathbb{C}^{m\times n}$, where $m := abd, n := acd$, such that its support is included in $\mathbf{S}_{\boldsymbol{\pi}} := \mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d \in \{0,1\}^{m\times n}$. The tuple $\boldsymbol{\pi}$ will be called an* elementary deformable butterfly pattern, or simply a *pattern. The set of all $\boldsymbol{\pi}$-factors is denoted by $\Sigma^{\boldsymbol{\pi}}$.*

Figure 7.1 illustrates the support of a $\boldsymbol{\pi}$-factor, for a given pattern $\boldsymbol{\pi} = (a,b,c,d)$. A $\boldsymbol{\pi}$-factor matrix is block diagonal with a blocks in total. By definition, each block in the diagonal has support included in $\mathbf{1}_{b\times c} \otimes \mathbf{I}_d$. Thus, each block is a *block matrix* of size $b \times c$, where each *sub-block* is a diagonal matrix of size $d \times d$.

Example 7.1. *The following matrices are $\boldsymbol{\pi}$-factors for certain choices of $\boldsymbol{\pi}$.*

1. **Dense matrix**: *Any matrix of size $m \times n$ is a $(1,m,n,1)$-factor.*
2. **Diagonal matrix**: *Any diagonal matrix of size $m \times m$ is either a $(m,1,1,1)$-factor or $(1,1,1,m)$-factor.*

3. **Factors in a square dyadic butterfly factorization** *[74, 75, 213, 392]: the pattern of the ℓ-th factor is $\boldsymbol{\pi}_\ell = (2^{\ell-1}, 2, 2, 2^{L-\ell})$ for $\ell \in [\![L]\!]$.*

4. **Factors in a Monarch factorization** *[72]: the patterns of the two factors are $\boldsymbol{\pi}_1 = (1, p, q, m/p)$, $\boldsymbol{\pi}_2 = (q, m/p, n/q, 1)$ for p, q such that $p \mid m$ and $q \mid n$.*

Lemma 7.1 (Sparsity level of a $\boldsymbol{\pi}$-factor)**.** *For $\boldsymbol{\pi} = (a, b, c, d)$, the number of nonzero entries of a $\boldsymbol{\pi}$-factor of size $m \times n$ is at most $\|\boldsymbol{\pi}\|_0 := abcd = mc = nb$.*

Proof. The cardinal of $\operatorname{supp}(\mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d)$ is $abcd = mc = nb = \frac{mn}{ad}$. □

A $\boldsymbol{\pi}$-factor is sparse if it has few nonzero entries compared to its size, i.e., if $\|\boldsymbol{\pi}\|_0 \ll mn$, or equivalently if $ad \gg \mathcal{O}(1)$. Given a number of factors $L \geq 1$, a sequence of patterns $\boldsymbol{\beta} := (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ parameterizes the set

$$\Sigma^{\boldsymbol{\beta}} := \Sigma^{\boldsymbol{\pi}_1} \times \ldots \times \Sigma^{\boldsymbol{\pi}_L} \tag{7.9}$$

of L-tuples of $\boldsymbol{\pi}_\ell$-factors, $\ell = 1, \ldots, L$. Since we are interested in matrix products $\mathbf{X}_1 \ldots \mathbf{X}_L$ for $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^{\boldsymbol{\beta}}$, we will only consider sequences of patterns $\boldsymbol{\beta}$ such that the size of $\mathbf{X}_\ell \in \Sigma^{\boldsymbol{\pi}_\ell}$ and $\mathbf{X}_{\ell+1} \in \Sigma^{\boldsymbol{\pi}_{\ell+1}}$ are compatible for computing the matrix product $\mathbf{X}_\ell \mathbf{X}_{\ell+1}$, for each $\ell \in [\![L-1]\!]$. In other words, we require that the sequence of patterns $\boldsymbol{\beta}$ satisfies:

$$\forall \ell \in [\![L-1]\!], \quad \underbrace{a_\ell c_\ell d_\ell}_{n_\ell} = \underbrace{a_{\ell+1} b_{\ell+1} d_{\ell+1}}_{m_{\ell+1}}. \tag{7.10}$$

Therefore, under assumption (7.10), a sequence $\boldsymbol{\beta}$ can describe a factorization of the type $\mathbf{A} = \mathbf{X}_1 \ldots \mathbf{X}_L$ such that $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^{\boldsymbol{\beta}}$. We introduce the following terminology for such a sequence.

Definition 7.3 (Butterfly architecture)**.** *A sequence of patterns $\boldsymbol{\beta} := (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ is called a* (deformable) butterfly architecture, or simply an *architecture*, when it satisfies (7.10). *By analogy with deep networks, the number of factors is called the depth of the chain and denoted by $|\boldsymbol{\beta}| := L$ and, using the notation $\|\boldsymbol{\pi}\|_0$ from Lemma 7.1, the number of parameters is denoted by*

$$\|\boldsymbol{\beta}\|_0 := \sum_{\ell=1}^{L} \|\boldsymbol{\pi}_\ell\|_0.$$

For any architecture $\boldsymbol{\beta}$, $\mathcal{B}^{\boldsymbol{\beta}}$ is the set of *(deformable) butterfly matrices* associated with $\boldsymbol{\beta}$, as defined in (7.2). We also say that any $\mathbf{A} \in \mathcal{B}^{\boldsymbol{\beta}}$ admits an *exact (deformable) butterfly factorization* associated with the architecture $\boldsymbol{\beta}$. Table 7.1 describes existing architectures fitting our framework.

The rest of this section introduces two important properties of an architecture $\boldsymbol{\beta}$:

- *Chainability* will be shown (Corollary 7.3) to ensure the existence of an optimum in (7.1), so that we can replace "inf" by "min" in (7.1). We also show that for chainable architecture one can exploit a hierarchical algorithm (Algorithm 7.3), which extends an algorithm from [213, 386]) to compute an approximate solution to Problem (7.1) for any chainable architecture.

- *Non-redundancy* is an additional property satisfied by some chainable architectures β, that allows to insert orthonormalization steps in the hierarchical algorithm, in order to control the approximation error for Problem (7.1) in the sense of (7.3). Non-redundancy plays the role of an intermediate tool to design and analyze our algorithms. However, it should not be treated as an additional hypothesis, because we do propose a factorization method (cf. Algorithm 7.7), endowed with error guarantees, for *any chainable architecture, whether redundant or not*.

Both conditions are first defined for the most basic architectures β of depth $|\beta| = 2$, before being generalized to architectures β of arbitrary depth $L \geq 2$.

7.4.2 Chainability

We start by defining this condition in the case of architectures of depth $L = 2$. This definition is primarily introduced to ensure a key "stability" property given next in Proposition 7.1, which will have many nice consequences.

Definition 7.4 (Chainable pair of patterns, operator $*$ on patterns)**.** *Two patterns $\boldsymbol{\pi}_1 := (a_1, b_1, c_1, d_1)$ and $\boldsymbol{\pi}_2 := (a_2, b_2, c_2, d_2)$ are* chainable *if:*

1. $\frac{a_1 c_1}{a_2} = \frac{b_2 d_2}{d_1}$ *and this quantity, denoted $q(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$, is an integer;*
2. $a_1 \mid a_2$ *and* $d_2 \mid d_1$.

We also say that the pair $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ is chainable. We define the operator $*$ on the set of chainable pairs of patterns as follows: if $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ is chainable, then

$$\boldsymbol{\pi}_1 * \boldsymbol{\pi}_2 := \left(a_1, \frac{b_1 d_1}{d_2}, \frac{a_2 c_2}{a_1}, d_2 \right) \in \mathbb{N}^4. \tag{7.11}$$

Note that assumption 2. in Definition 7.4 is indeed that a_1 divides a_2 (and d_2 divides d_1), even though the definition of $q(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ involves the quotient a_1/a_2 (resp. d_2/d_1).

Remark 7.3. *The order $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ in the definition matters,* i.e.*, this property is not symmetric: the chainability of $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ does not imply that of $(\boldsymbol{\pi}_2, \boldsymbol{\pi}_1)$. Moreover, by the first condition of Definition 7.4, a chainable pair is indeed an architecture in the sense of Definition 7.3.*

Definition 7.4 comes with the following two key propositions.

Proposition 7.1. *If (π_1, π_2) is chainable, then:*

$$\mathbf{S}_{\pi_1}\mathbf{S}_{\pi_2} = q(\pi_1, \pi_2)\mathbf{S}_{\pi_1 * \pi_2}. \tag{7.12}$$

The proof is deferred to Appendix C.2.1. The equality (7.12) was proved in [392, Lemma 3.4] for the choice $\pi_1 = (2^{\ell-1}, 2, 2, 2^{L-\ell})$ and $\pi_2 = (2^{\ell}, 2, 2, 2^{L-\ell-1})$, for any integer $L \geq 2$ and $\ell \in [\![L-1]\!]$. Proposition 7.1 extends (7.12) to *all* chainable pairs (π_1, π_2).

Chainability and Definition 7.2 imply that $\forall (\mathbf{X}_1, \mathbf{X}_2) \in \Sigma^{\pi_1} \times \Sigma^{\pi_2}$, $\mathbf{X}_1\mathbf{X}_2 \in \Sigma^{(\pi_1 * \pi_2)}$, i.e., a product of butterfly factors with chainable patterns (π_1, π_2) is still a butterfly factor, with pattern $\pi_1 * \pi_2$. We can characterize the set $\{\mathbf{X}_1\mathbf{X}_2 \,|\, \mathbf{X}_i \in \Sigma^{\pi_i}, i \in [\![2]\!]\}$ more precisely as follows. The proof is deferred to Appendix C.2.2.

Lemma 7.2. *If $\beta := (\pi_1, \pi_2)$ is chainable then (with the notations of Definition 7.1) for each $P \in \mathcal{P}(\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})$ we have $|P| = q(\pi_1, \pi_2)$, $|R_P| = b_1$ and $|C_P| = c_2$ (with $\pi_i = (a_i, b_i, c_i, d_i)$). Moreover, with $\mathcal{A}^\beta$ the set of matrices of the size of matrices in $\mathcal{B}^\beta$ such that* rank $(\mathbf{A}[R_P, C_P]) \leq q(\pi_1, \pi_2)$ *for each $P \in \mathcal{P}(\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})$, we have*

$$\mathcal{B}^\beta = \Sigma^{\pi_1 * \pi_2} \cap \mathcal{A}^\beta. \tag{7.13}$$

Lemma 7.3 (Associativity of $*$)**.** *If (π_1, π_2) and (π_2, π_3) are chainable, then*

1. *$(\pi_1, \pi_2 * \pi_3)$ and $(\pi_1 * \pi_2, \pi_3)$ are chainable;*
2. *$q(\pi_1, \pi_2 * \pi_3) = q(\pi_1, \pi_2)$ and $q(\pi_1 * \pi_2, \pi_3) = q(\pi_2, \pi_3)$;*
3. *$\pi_1 * (\pi_2 * \pi_3) = (\pi_1 * \pi_2) * \pi_3$.*

The proof of Lemma 7.3 is deferred to Appendix C.2.3. We can now extend the definition of chainability to a general architecture β of arbitrary depth $L \geq 1$.

Definition 7.5 (Chainable architecture)**.** *An architecture $\beta := (\pi_\ell)_{\ell=1}^L$, $L \geq 2$, is* chainable *if π_ℓ and $\pi_{\ell+1}$ are chainable for each $\ell \in [\![L-1]\!]$ in the sense of Definition 7.4. We then denote $\mathbf{q}(\beta) = (q(\pi_\ell, \pi_{\ell+1}))_{\ell=1}^{L-1} \in \mathbb{N}^{L-1}$. By convention any architecture of depth $L = 1$ is also chainable.*

Example 7.2. *One can check that the square dyadic butterfly architecture (resp. the Monarch architecture), cf Example 7.1, are chainable, with $\mathbf{q}(\beta) = (1, \ldots, 1)$ (resp. $\mathbf{q}(\beta) = (1)$). They are particular cases of the 5-parameter deformable butterfly architecture of [241], which is chainable with $\mathbf{q}(\beta) = (1, \ldots, 1)$. In contrast, the Kaleidoscope architecture of depth $2L$ with $L \geq 2$ of Table 7.1 is* not *chainable, because for $\ell = L+1$ we have $\pi_\ell = (2^{L-1}, 2, 2, 1)$, $\pi_{\ell+1} = (2^{L-2}, 2, 2, 2)$, and this pair is not chainable since 2^{L-1} does not divide 2^{L-2}.*

We state in the following some useful properties of chainable architectures.

Lemma 7.4. *If $\beta = (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ with $L \geq 2$ is chainable then $\mathcal{B}^\beta \subseteq \Sigma^{(\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_L)}$, with*

$$\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_L = \left(a_1, \frac{b_1 d_1}{d_L}, \frac{a_L c_L}{a_1}, d_L\right). \tag{7.14}$$

Partial proof. The result $\mathcal{B}^\beta \subseteq \Sigma^{(\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_L)}$ follows from an induction on $|\beta|$ where the base case ($L = 2$) is given by Lemma 7.2. We prove (7.14) in Appendix C.2.4. □

Remark 7.4. *As a consequence of this lemma, if the first pattern $\boldsymbol{\pi}_1$ of a chainable architecture β satisfies $a_1 > 1$ then all matrices in $\mathcal{B}^\beta$ have a support included in $\mathbf{S}_{\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_L}$, which has zeroes outside its main block diagonal structure (see Figure 7.1). A similar remark holds when $d_L > 1$, and in both cases we conclude that $\mathcal{B}^\beta$ does not contain any dense matrix where all entries are nonzero. In contrast, when $a_1 = d_L = 1$, it is known for specific architectures that some dense matrices do belong to $\mathcal{B}^\beta$. This is notably the case when β is the square dyadic butterfly architecture or the Monarch architecture (see Example 7.1): then we have $\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_L = (a_1, m, n, d_L) = (1, m, n, 1)$ for some integers m, n, and indeed the Hadamard (or the DFT matrix, up to bit-reversal permutation of its columns, cf. [74]) is a dense matrix belonging to $\mathcal{B}^\beta$.*

Next we state an essential property of chainable architectures. It builds on and extends Lemma 7.3, and corresponds to a form of *stability* under pattern multiplication that will serve as a cornerstone to support the introduction of hierarchical algorithms.

Lemma 7.5. *If $\beta = (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ is chainable then for each $1 \leq r \leq s < t \leq L$, the patterns $(\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s)$ and $(\boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t)$ are well-defined and chainable with $q(\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t) = q(\boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1})$.*

The proof is deferred to Appendix C.2.5.

7.4.3 Non-redundancy

A first version of our proposed hierarchical factorization algorithm (expressed recursively as Algorithm 7.3 in Algorithm 7.3, or non-recursively as Algorithm 7.4 in Section 7.6) will be applicable to any chainable architecture β. However, establishing approximation guarantees of the type (7.3) will require a variant of this algorithm (Algorithm 7.5) involving certain orthonormalization steps, which are only well-defined if the architecture β satisfies an additional *non-redundancy* condition. Fortunately, any redundant architecture β can be transformed (Proposition 7.2) into an expressively equivalent architecture β' (*i.e.*, $\mathcal{B}^{\beta'} = \mathcal{B}^\beta$) with reduced number of parameters ($\|\beta'\|_0 \leq \|\beta'\|_0$) thanks to Algorithm 7.2 below. This will be instrumental in introducing the proving approximation guarantees

of the final hierarchical algorithm Algorithm 7.5 applicable to *any (redundant or not) chainable architecture*.

To define redundancy of an architecture we begin by considering elementary pairs.

Definition 7.6 (Redundant architecture)**.** *A chainable pair of patterns* $\boldsymbol{\pi}_1 = (a_1, b_1, c_1, d_1)$ *and* $\boldsymbol{\pi}_2 = (a_2, b_2, c_2, d_2)$ *is* redundant *if* $q(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2) \geq \min(b_1, c_2)$ *(i.e., if* $a_1 c_1 \geq a_2 c_2$ *or* $b_2 d_2 \geq b_1 d_1$*). A chainable architecture* $\boldsymbol{\beta} = (\boldsymbol{\pi}_\ell)_{\ell=1}^L$, $L = |\boldsymbol{\beta}| \geq 1$ *is* redundant *if there exists* $\ell \in [\![L-1]\!]$ *such that* $(\boldsymbol{\pi}_\ell, \boldsymbol{\pi}_{\ell+1})$ *is redundant. Observe that by definition, any chainable architecture with* $|\boldsymbol{\beta}| = 1$ *is non-redundant.*

Lemma 7.6. *If* $\boldsymbol{\beta} = (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ *is chainable and non-redundant then, for any* $1 \leq r \leq s < t \leq L$*, the pair* $(\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t)$ *is chainable and non-redundant.*

The proof is deferred to Appendix C.2.6.

Example 7.3. *The architecture* $\boldsymbol{\beta} := (\boldsymbol{\pi}_1, \boldsymbol{\pi}_2) := ((1, m, r, 1), (1, r, n, 1))$ *is chainable, with* $q(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2) = r$*. The set* $\mathcal{B}^{\boldsymbol{\beta}}$ *is the set of* $m \times n$ *matrices of rank at most* r*.* $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ *is redundant if* $r \geq \min(m, n)$*. We observe that on this example redundancy corresponds to the case where* $\mathcal{B}^{\boldsymbol{\beta}}$ *is the set of* all $m \times n$ *matrices.*

A (chainable and) redundant architecture is as expressive as a smaller chainable architecture with less parameters. This is first proved for pairs.

Lemma 7.7. *Consider a chainable pair* $\boldsymbol{\beta} = (\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$*. If* $\boldsymbol{\beta}$ *is redundant, then the single-factor architecture* $\boldsymbol{\beta}' = (\boldsymbol{\pi}_1 * \boldsymbol{\pi}_2)$ *satisties:*

1. $\mathcal{B}^{\boldsymbol{\beta}} = \Sigma^{\boldsymbol{\pi}_1 * \boldsymbol{\pi}_2} = \mathcal{B}^{\boldsymbol{\beta}'}$.
2. $\|\boldsymbol{\beta}'\|_0 = \|\boldsymbol{\pi}_1 * \boldsymbol{\pi}_2\|_0 < \|\boldsymbol{\pi}_1\|_0 + \|\boldsymbol{\pi}_2\|_0 = \|\boldsymbol{\beta}\|_0$.

Proof. By Lemma 7.2 we have $\mathcal{B}^{\boldsymbol{\beta}} = \Sigma^{\boldsymbol{\pi}_1 * \boldsymbol{\pi}_2} \cap \mathcal{A}^{\boldsymbol{\beta}}$ and $|R_P| = b_1, |C_P| = c_2$ for each $P \in \mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}_1}, \mathbf{S}_{\boldsymbol{\pi}_2})$. This first claim follows from the fact that $\mathcal{A}^{\boldsymbol{\beta}}$ is the set of all matrices of appropriate size: indeed for any such matrix $\mathbf{A}$, the block $\mathbf{A}[R_P, C_P]$ is of size $b_1 \times c_2$ hence its rank is at most $\min(b_1, c_2)$ which is smaller than $q(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ since $\boldsymbol{\beta}$ is redundant. By definition of $\mathcal{A}^{\boldsymbol{\beta}}$ this shows that $\mathbf{A} \in \mathcal{A}^{\boldsymbol{\beta}}$. For the second claim, by Definition 7.3 of $\|\boldsymbol{\beta}\|_0, \|\boldsymbol{\beta}'\|_0$ we only need to prove the strict inequality. Since $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ is (chainable and) redundant, we have either $a_1 c_1 \geq a_2 c_2$ or $b_2 d_2 \geq b_1 d_1$, hence by Lemma 7.1 and Equation (7.11) we obtain $\|\boldsymbol{\pi}_1 * \boldsymbol{\pi}_2\|_0 = a_2 c_2 b_1 d_1 < a_1 c_1 b_1 d_1 + a_2 c_2 b_2 d_2 = \|\boldsymbol{\pi}_1\|_0 + \|\boldsymbol{\pi}_2\|_0$. □

Lemma 7.7 serves as a basis to define Algorithm 7.2, which replaces any chainable (and possibly redundant) architecture by a "smaller" non-redundant one.

Algorithm 7.2 Architecture redundancy removal algorithm

Require: A chainable $\beta = (\boldsymbol{\pi}_\ell)_{\ell=1}^{L}$
Ensure: A chainable and non-redundant $\beta' = (\boldsymbol{\pi}'_\ell)_{\ell=1}^{L'}$ $(1 \leq L' \leq L)$
1: $\beta' \leftarrow \beta$.
2: **while** β' is redundant (cf. Definition 7.6) **do**
3: $\quad (\boldsymbol{\pi}'_\ell)_{\ell=1}^{L'} \leftarrow \beta'$
4: $\quad \ell \leftarrow$ an integer ℓ such that $(\boldsymbol{\pi}'_\ell, \boldsymbol{\pi}'_{\ell+1})$ is redundant (cf. Definition 7.6)
5: $\quad \beta' \leftarrow (\boldsymbol{\pi}'_1, \ldots, \boldsymbol{\pi}'_{\ell-1}, \boldsymbol{\pi}'_\ell * \boldsymbol{\pi}'_{\ell+1}, \boldsymbol{\pi}'_{\ell+2}, \ldots, \boldsymbol{\pi}'_{L'})$
6: **end while**
7: **return** β'

Proposition 7.2. *For any chainable architecture $\beta = (\boldsymbol{\pi}_\ell)_{\ell=1}^{L}$, Algorithm 7.2 stops in finitely many iterations and returns an architecture β' such that:*

1. *β' is chainable and non-redundant, and either a single factor architecture $\beta' = (\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_L)$, or a multi-factor one $\beta' = (\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_{\ell_1}, \boldsymbol{\pi}_{\ell_1+1} * \ldots * \boldsymbol{\pi}_{\ell_2}, \ldots, \boldsymbol{\pi}_{\ell_p+1} * \ldots * \boldsymbol{\pi}_L)$ for some indices $1 \leq \ell_1 < \ldots < \ell_p < L$ with $p \in [\![1, L-1]\!]$;*
2. *$\mathcal{B}^{\beta'} = \mathcal{B}^{\beta}$;*
3. *$\|\beta'\|_0 \leq \|\beta\|_0$.*

Proof. Algorithm 7.2 terminates since $|\beta'|$ decreases at each iteration. At each iteration, the updated β' is obtained by replacing a chainable redundant pair $(\boldsymbol{\pi}'_\ell, \boldsymbol{\pi}'_{\ell+1})$ by a single pattern $(\boldsymbol{\pi}'_\ell * \boldsymbol{\pi}'_{\ell+1})$. The architecture β' remains chainable by Lemma 7.5 and by chainability of β, hence the algorithm can continue with no error. Due to the condition of the "while" loop, the returned β' is either non-redundant with $|\beta'| > 1$, or $|\beta'| = 1$ in which case it is in fact also non-redundant by Definition 7.6. This yields the first condition (a formal proof of the final form of β' can be done by an easy but tedious induction left to the reader). Moreover, a straightforward consequence of Lemma 7.7 is that the update of β' in line 5 does not change $\mathcal{B}^{\beta'}$, and it strictly decreases $\|\beta'\|_0$ if the condition of the "while" loop is met at least once (otherwise the algorithm outputs $\beta' = \beta$). This yields the two other properties. □

In particular, Algorithm 7.2 applied to a redundant architecture β in Example 7.3 returns $\beta' = ((1, m, r, 1) * (1, r, n, 1)) = ((1, m, n, 1))$.

7.5 Hierarchical algorithm for chainable architectures

We show how Algorithm 6.1, initially introduced for specific (square dyadic) architectures in Chapter 6 can be directly extended to the case where β is *any* chain-

able architecture. The case where $L = |\beta| = 1$ is trivial since Problem (7.1) is then simply solved by setting $\mathbf{X}_1$ to be a copy of $\mathbf{A}$ where all entries off the prescribed support are set to zero. We thus focus on $L \geq 2$ and start with β of depth $L = 2$ before considering arbitrary $L \geq 2$.

7.5.1 Case with $L = 2$ factors

Problem (7.1) with an architecture $\beta = (\pi_1, \pi_2)$ is simply an instance of Problem (7.5) with $(\mathbf{L}, \mathbf{R}) = (\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})$.

Lemma 7.8. *The conditions in Theorem 7.1 are verified by the pair of supports* $(\mathbf{L}, \mathbf{R}) = (\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})$ *for any* (not necessarily chainable) *architecture* (π_1, π_2). *Consequently, for* any *architecture* β *of depth* $|\beta| = 2$, *Algorithm 7.1 returns an optimal solution to the corresponding instance of Problem* (7.1).

Proof. In general, for any pattern π, by Definition 7.2, the column supports $\{\mathbf{S}_\pi[:, j]\}_j$ and the row supports $\{\mathbf{S}_\pi[i, :]\}_i$ are pairwise disjoint or identical, respectively. A fortiori, any pair of elements in the tuple $\varphi(\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})$ is either identical (if their corresponding column and row supports coincide) or disjoint. □

7.5.2 Case with $L \geq 2$ factors

Consider now Problem (7.1) associated with a *chainable* architecture β of depth $L := |\beta| \geq 2$, and a given target matrix $\mathbf{A}$. A first proposition of hierarchical algorithm, introduced in Algorithm 7.3, is a direct adaptation to our framework of Algorithm 6.1. It computes an approximate solution by performing successive two-factor matrix factorization in a certain hierarchical order that is described by a factor-bracketing tree (Definition 6.4). Further refinements of the algorithm will later be added to obtain approximation guarantees.

Before exposing the limitations of Algorithm 7.3 and proposing fixes, let us briefly explain how it works with a focus on its main step in line 7. Consider any factor-bracketing tree $\mathcal{T}$. Algorithm 7.3 computes a matrix $\mathbf{X}_{[\![r,t]\!]} \in \Sigma^{\pi_r * \ldots * \pi_t}$ for each node $[\![r, t]\!]$ in a recursive manner. $\pi_r * \ldots * \pi_t$ is well-defined for any $1 \leq r \leq t \leq L$ because β is chainable. At the root node, we set $\mathbf{X}_{[\![1,L]\!]} := \mathbf{A}$. At each non-leaf node $[\![r, t]\!]$ whose matrix $\mathbf{X}_{[\![r,t]\!]}$ is already computed during the hierarchical procedure, and with children $[\![r, s]\!]$ and $[\![s+1, t]\!]$, at line 7 we compute $(\mathbf{X}_{[\![r,s]\!]}, \mathbf{X}_{[\![s+1,t]\!]}) \in \Sigma^{\pi_r * \ldots * \pi_s} \times \Sigma^{\pi_{s+1} * \ldots * \pi_t}$ that is solution to the following instance of the Fixed Support Matrix Factorization Problem (7.5):

$$\begin{aligned} \text{Minimize} \quad & \|\mathbf{X}_{[\![r,t]\!]} - \mathbf{X}_{[\![r,s]\!]} \mathbf{X}_{[\![s+1,t]\!]}\|_F \\ \text{Subject to} \quad & \operatorname{supp}(\mathbf{X}_{[\![r,s]\!]}) \subseteq \mathbf{S}_{\pi_r * \ldots * \pi_s}, \\ & \operatorname{supp}(\mathbf{X}_{[\![s+1,t]\!]}) \subseteq \mathbf{S}_{\pi_{s+1} * \ldots * \pi_t}. \end{aligned} \tag{7.15}$$

Indeed, by Lemma 7.8, Problem (7.15) is solved by Algorithm 7.1, which yields line 7 in Algorithm 7.3. After computing $(\mathbf{X}_{[\![r,s]\!]}, \mathbf{X}_{[\![s+1,t]\!]})$, we repeat recursively

Algorithm 7.3 Hierarchical algorithm (recursive version)

Require: $\mathbf{A} \in \mathbb{C}^{m \times n}$, chainable $\beta = (\pi_\ell)_{\ell=1}^{L}$, factor-bracketing tree $\mathcal{T}$
Ensure: `factors` $\in \Sigma^\beta$
1: **if** $L = 1$ **then**
2: **return** $(\mathbf{A} \odot \mathbf{S}_{\pi_1})$
3: **end if**
4: $[\![1, s]\!], [\![s+1, L]\!] \leftarrow$ two children of the root $[\![1, L]\!]$ of $\mathcal{T}$
5: $(\mathcal{T}_{\text{left}}, \mathcal{T}_{\text{right}}) \leftarrow$ the corresponding left and right subtrees of $\mathcal{T}$
6: $(\pi_{\text{left}}, \pi_{\text{right}}) \leftarrow (\pi_1 * \ldots * \pi_s, \pi_{s+1} * \ldots * \pi_L)$
7: $(\mathbf{X}_{[\![1,s]\!]}, \mathbf{X}_{[\![s+1,L]\!]}) \leftarrow$ Algorithm 7.1$\left(\mathbf{A}, \mathbf{S}_{\pi_{\text{left}}}, \mathbf{S}_{\pi_{\text{right}}}\right)$
8: `left_factors` $\leftarrow$ Algorithm 7.3 $\left(\mathbf{X}_{[\![1,s]\!]}, (\pi_1, \ldots, \pi_s), \mathcal{T}_{\text{left}}\right)$
9: `right_factors` $\leftarrow$ Algorithm 7.3 $\left(\mathbf{X}_{[\![s+1,L]\!]}, (\pi_{s+1}, \ldots, \pi_L), \mathcal{T}_{\text{right}}\right)$
10: `factors` $\leftarrow$ `left_factors` $\cup$ `right_factors`
11: **return** `factors`

the procedure on these two matrices independently, as per lines 8 and 9, until we obtain the butterfly factors $(\mathbf{X}_{[\![\ell,\ell]\!]})_{\ell=1}^{L} \in \Sigma^\beta$ that yield an approximation $\hat{\mathbf{A}} := \mathbf{X}_{[\![1,1]\!]} \ldots \mathbf{X}_{[\![L,L]\!]} \in \mathcal{B}^\beta$ of $\mathbf{A}$. In conclusion, Algorithm 7.3 is a greedy algorithm that seeks the optimal solution at each two-factor matrix factorization problem during the hierarchical procedure.

7.5.3 Algorithm 7.3 does not satisfy the theoretical guarantee (7.3)

However, the control of the approximation error in the form of (7.3) for Algorithm 7.3 in its current form is *impossible*, as illustrated in the following example.

Example 7.4. *Consider* $\beta = (\pi_1, \pi_2, \pi_3) = \left((2^{\ell-1}, 2, 2, 2^{3-\ell})\right)_{\ell=1}^{3}$, *which is the square dyadic architecture of depth* $L = 3$. *Define* $\mathbf{A} := (\mathbf{D}\mathbf{S}_{\pi_1})\mathbf{S}_{\pi_2}\mathbf{S}_{\pi_3}$ *where* $\mathbf{D}$ *is the diagonal matrix with diagonal entries* $(0, 1, 1, 1, 0, 1, 1, 1)$. *Hence,* $\mathbf{A} \in \mathcal{B}^\beta$, *meaning that any algorithm with a theoretical guarantee* (7.3) *must output butterfly factors whose product is exactly* $\mathbf{A}$. *However, we claim that this is not the case of Algorithm 7.3 with the so-called left-to-right factor-bracketing tree of* $[\![1, 3]\!]$ *(defined as the tree where each left child is a singleton). To see why, let us apply this algorithm to* $\mathbf{A}$.

1. *In the first step, the hierarchical algorithm applies Algorithm 7.1 with the inputs* $(\mathbf{A}, \mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2 * \pi_3})$, *and returns* $(\mathbf{X}_{[\![1,1]\!]}, \mathbf{X}_{[\![2,3]\!]}) \in \Sigma^{\pi_1} \times \Sigma^{\pi_2 * \pi_3}$.

2. *At the second step (which is the last one), Algorithm 7.1 is applied to the input* $(\mathbf{X}_{[\![2,3]\!]}, \mathbf{S}_{\pi_2}, \mathbf{S}_{\pi_3})$, *and returns* $(\mathbf{X}_{[\![2,2]\!]}, \mathbf{X}_{[\![3,3]\!]}) \in \Sigma^{\pi_2} \times \Sigma^{\pi_3}$.

By construction, the first and the fifth row of $\mathbf{A}$ *are null, so the first step can return* many possible optimal solutions $(\mathbf{X}_{[\![1,1]\!]}, \mathbf{X}_{[\![2,3]\!]})$ *for the considered instance of Problem*

(7.5), *such as* $\mathbf{X}_{[\![1,1]\!]} := \mathbf{D}\mathbf{S}_{\pi_1}$ *and*

$$\mathbf{X}_{[\![2,3]\!]} = \begin{pmatrix} \mathbf{B} & \mathbf{0}_{4\times 4} \\ \mathbf{0}_{4\times 4} & \mathbf{B} \end{pmatrix} \quad \textit{with} \quad \mathbf{B} = \begin{pmatrix} \alpha & \beta & \gamma & \delta \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix},$$

where the scalars $\alpha, \beta, \gamma, \delta$ *can be arbitrary. Then, with the choice* $(\alpha, \beta, \gamma, \delta) = (1, 2, 3, 4)$, *one can check that the second step of the procedure will always output* $\mathbf{X}_{[\![2,2]\!]}$ *and* $\mathbf{X}_{[\![3,3]\!]}$ *such that* $\mathbf{X}_{[\![2,2]\!]}\mathbf{X}_{[\![3,3]\!]} \neq \mathbf{X}_{[\![2,3]\!]}$ *and* $\mathbf{X}_{[\![1,1]\!]}\mathbf{X}_{[\![2,2]\!]}\mathbf{X}_{[\![3,3]\!]} \neq \mathbf{A}$. *In conclusion, this example*[2] *shows that the output of Algorithm 7.3 cannot satisfy the theoretical guarantee* (7.3).

The inability to establish an error bound as in (7.3) for Algorithm 7.3 is due to the ambiguity for the choice of optimal factors $(\mathbf{X}_{[\![1,s]\!]}, \mathbf{X}_{[\![s+1,L]\!]})$ returned by Algorithm 7.1 called at line 7 in Algorithm 7.3, cf. Remark 7.2. At each iteration, there are *multiple* optimal pairs of factors, and the choice at line 7 impacts subsequent factorizations in the recursive procedure. To guarantee an error bound of the type (7.3), Section 7.6 proposes a revision of Algorithm 7.3, where, among all the possible choices, the modified algorithm selects *specific* input matrices at lines 8 and 9.

7.6 Hierarchical algorithm with error guarantees

We now propose a modification of Algorithm 7.3 using *orthonormalization operations* that are novel in the context of butterfly factorization. It is based on an unrolled version of Algorithm 7.3 and will be endowed with error guarantees stated and proved in the next section.

7.6.1 A non-recursive version for Algorithm 7.3

The factor-bracketing tree $\mathcal{T}$ in Algorithm 7.3 describes in which order the successive $L-1$ two-factor matrix factorization steps are performed, where $L := |\beta|$. An equivalent way to describe this hierarchical order is to store a permutation $\sigma := (\sigma_\ell)_{\ell=1}^{L-1}$ of $[\![L-1]\!]$, by saving each splitting index $s \in [\![L-1]\!]$ that corresponds to the maximum integer in the left child $[\![r, s]\!]$ of each non-leaf node $[\![r, t]\!]$ of $\mathcal{T}$ (cf. Definition 6.4). We can then reformulate the recursive Algorithm 7.3 as a non-recursive version of the algorithm as in Algorithm 7.4. These two algorithms are equivalent when $\mathcal{T}$ and σ match, and the non-recursive version will ease the incorporation of orthogonalization operations.

[2] At first sight, this seems to contradict the so-called exact recovery property in the case of square dyadic butterfly factorization from Theorem 6.1. This is not the case, since the statement of these exact recovery results includes a technical assumption excluding matrices with zero columns/rows, which is not satisfied by $\mathbf{A}$ here.

Algorithm 7.4 Hierarchical algorithm (non-recursive version)

Require: $\mathbf{A} \in \mathbb{C}^{m \times n}$, chainable $\beta = (\boldsymbol{\pi}_\ell)_{\ell=1}^{L}$, permutation $(\sigma_\ell)_{\ell=1}^{L-1}$ of $[\![L-1]\!]$
Ensure: `factors` $\in \Sigma^\beta$
1: **if** $L = 1$ **then**
2: **return** $(\mathbf{A} \odot \mathbf{S}_{\boldsymbol{\pi}_1})$
3: **end if**
4: `partition` $\leftarrow (P_1)$ where we denote $P_1 = [\![1, L]\!]$
5: `factors` $\leftarrow (\mathbf{A})$
6: **for** $J = 1, \ldots, L-1$ **do**
7: $(P_j)_{j=1}^{J} \leftarrow$ `partition`
8: $(\mathbf{X}_{P_j})_{j=1}^{J} \leftarrow$ `factors`
9: $j \leftarrow$ the unique $j \in [\![J]\!]$ such that $P_j := [\![r, t]\!] \ni s := \sigma_J$
10: $(\boldsymbol{\pi}_{\text{left}}, \boldsymbol{\pi}_{\text{right}}) \leftarrow (\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t)$
11: $(\mathbf{X}_{[\![r,s]\!]}, \mathbf{X}_{[\![s+1,t]\!]}) \leftarrow$ Algorithm 7.1$\left(\mathbf{X}_{P_j}, \mathbf{S}_{\boldsymbol{\pi}_{\text{left}}}, \mathbf{S}_{\boldsymbol{\pi}_{\text{right}}}\right)$
12: `partition` $\leftarrow \left(P_1, \ldots, P_{j-1}, [\![r, s]\!], [\![s+1, t]\!], P_{j+1}, \ldots, P_J\right)$
13: `factors` $\leftarrow \left(\mathbf{X}_{P_1}, \ldots, \mathbf{X}_{P_{j-1}}, \mathbf{X}_{[\![r,s]\!]}, \mathbf{X}_{[\![s+1,t]\!]}, \mathbf{X}_{P_{j+1}}, \ldots, \mathbf{X}_{P_J}\right)$
14: **end for**
15: **return** `factors`

7.6.2 Hierarchical algorithms with orthonormalization steps

Being equivalent to Algorithm 7.3, the non-recursive version Algorithm 7.4 still suffers from the pitfall highlighted in Example 7.4 regarding error guarantees. Algorithm 7.5 improves it by introducing additional operations (lines 10-15, performed using Algorithm 7.6), called *orthonormalization operations*, that are only well-defined if we restrict to a *non-redundant* chainable architecture β. Combining this modification with redundancy removal (Algorithm 7.2) yields Algorithm 7.7. It applies to any chainable architecture and will be shown in Section 7.7 to satisfy the desired approximation guarantees.

Orthonormalization operations under the non-redundancy assumption. The goal of the orthonormalization operations (lines 10-15 of Algorithm 7.5) is to rescale the butterfly factors $(\mathbf{X}_{P_k})_{k=1}^{J}$ without changing their product, in order to make a specific choice of $\mathbf{X}_{[\![r,t]\!]}$ given as input to Algorithm 7.1 at line 17 during subsequence steps of the algorithm, while constructing factors $\mathbf{X}_{P_k}$ for all $k \in [\![J]\!]$ such that $k \neq j$ that are orthonormal in a certain sense detailed in Appendix C.3. Let us highlight that orthonormalization operations are indeed well-defined under the non-redundancy assumption. First, in Algorithm 7.6, the input pair of patterns $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ is assumed to be chainable and non-redundant. By Lemma 7.2 and Definition 7.6, we have $|R_P| \leq |P|$ and $|C_P| \leq |P|$, which makes the operations at lines 10 and 5 in Algorithm 7.6 well-defined. Second, in Algorithm 7.5, the

Algorithm 7.5 Hierarchical algorithm with orthonormalization operations for *non-redundant* chainable architecture

Require: $\mathbf{A} \in \mathbb{C}^{m\times n}$, non-redundant, chainable $\beta = (\pi_\ell)_{\ell=1}^{L}$, permutation $(\sigma_\ell)_{\ell=1}^{L-1}$ of $[\![L-1]\!]$
Ensure: `factors` $\in \Sigma^\beta$

1: **if** L = 1 **then**
2: **return** $(\mathbf{A} \odot \mathbf{S}_{\pi_1})$
3: **end if**
4: `partition` $\leftarrow (P_1)$ where we denote $P_1 = [\![L-1]\!]$
5: `factors` $\leftarrow (\mathbf{A})$
6: **for** $J = 1, \dots, L-1$ **do**
7: $(P_j)_{j=1}^{J} \leftarrow$ `partition`
8: $(\mathbf{X}_{P_j})_{j=1}^{J} \leftarrow$ `factors`
9: $j \leftarrow$ the unique $j \in [\![J]\!]$ such that $P_j := [\![r,t]\!] \ni s := \sigma_J$
10: **for** $k = 1, \dots, j-1$ **do** ▷ Begin orthonormalization
11: $(\mathbf{X}_{P_k}, \mathbf{X}_{P_{k+1}}) \leftarrow$ Algorithm 7.6 $(\pi_{P_k}, \pi_{P_{k+1}}, \mathbf{X}_{P_k}, \mathbf{X}_{P_{k+1}}, \texttt{column})$
12: **end for**
13: **for** $k = J, \dots, j+1$ **do**
14: $(\mathbf{X}_{P_{k-1}}, \mathbf{X}_{P_k}) \leftarrow$ Algorithm 7.6 $(\pi_{P_{k-1}}, \pi_{P_k}, \mathbf{X}_{P_{k-1}}, \mathbf{X}_{P_k}, \texttt{row})$
15: **end for** ▷ End orthonormalization
16: $(\pi_{\text{left}}, \pi_{\text{right}}) \leftarrow (\pi_r * \dots * \pi_s, \pi_{s+1} * \dots * \pi_t)$
17: $(\mathbf{X}_{[\![r,s]\!]}, \mathbf{X}_{[\![s+1,t]\!]}) \leftarrow$ Algorithm 7.1 $\left(\mathbf{X}_{P_j}, \mathbf{S}_{\pi_{\text{left}}}, \mathbf{S}_{\pi_{\text{right}}}\right)$
18: `partition` $\leftarrow \left(P_1, \dots, P_{j-1}, [\![r,s]\!], [\![s+1,t]\!], P_{j+1}, \dots, P_J\right)$
19: `factors` $\leftarrow \left(\mathbf{X}_{P_1}, \dots, \mathbf{X}_{P_{j-1}}, \mathbf{X}_{[\![r,s]\!]}, \mathbf{X}_{[\![s+1,t]\!]}, \mathbf{X}_{P_{j+1}}, \dots, \mathbf{X}_{P_J}\right)$
20: **end for**
21: **return** `factors`

architecture β is assumed to be non-redundant. By Lemma 7.6, this means that the pair $(\pi_{P_k}, \pi_{P_{k+1}})$ at line 11 or the pair $(\pi_{P_{k-1}}, \pi_{P_k})$ at line 14 are chainable and non-redundant. This makes the call to Algorithm 7.6 at these lines well-defined.

Dealing with redundant architectures. When the architecture β is redundant, Algorithm 7.7 provides a flexible method to construct an approximate solution for Problem (7.1). It applies Algorithm 7.5 with a non-redundant architecture β' (cf. Line 1 - Algorithm 7.7) that is expressively equivalent to β (i.e, $\mathcal{B}^{\beta'} = \mathcal{B}^{\beta}$) as input. This yields an approximation $\mathbf{A} \approx \prod_{\ell=1}^{L'} \mathbf{X}'_\ell$ with $L' := |\beta'|$ and $(\mathbf{X}_{\ell'})_{\ell'=1}^{L'} \in \Sigma^{\beta'}$. Then, it is possible to show that we can construct $(\mathbf{X}_\ell)_{\ell=1}^{L} \in \Sigma^{\beta}$ such that $\prod_{\ell=1}^{L'} \mathbf{X}'_\ell = \prod_{\ell=1}^{L} \mathbf{X}_\ell$. Indeed, by Lemma 7.7, for any *redundant* pair of patterns $(\pi, \tilde{\pi})$, Algorithm 7.1 with inputs $\mathbf{A} \in \Sigma^{\pi * \tilde{\pi}}$ and the support constraints $(\mathbf{S}_\pi, \mathbf{S}_{\tilde{\pi}})$ returns $(\mathbf{X}, \mathbf{Y}) \in \Sigma^\pi \times \Sigma^{\tilde{\pi}}$ such that $\mathbf{XY} = \mathbf{A}$. In conclusion, this yields automatically an approximation $\mathbf{A} \approx \prod_{\ell=1}^{L} \mathbf{X}_\ell$ with the same approximation error as

Algorithm 7.6 Column/row-orthonormalization

Require: Non-redundant $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$, $\mathbf{X} \in \Sigma^{\boldsymbol{\pi}_1}$, $\mathbf{Y} \in \Sigma^{\boldsymbol{\pi}_2}$, $u \in \{\texttt{column}, \texttt{row}\}$
Ensure: $(\tilde{\mathbf{X}}, \tilde{\mathbf{Y}}) \in \Sigma^{\boldsymbol{\pi}_1} \times \Sigma^{\boldsymbol{\pi}_2}$ such that $\tilde{\mathbf{X}}\tilde{\mathbf{Y}} = \mathbf{X}\mathbf{Y}$
$(\tilde{\mathbf{X}}, \tilde{\mathbf{Y}}) \leftarrow (\mathbf{0}, \mathbf{0})$
for $P \in \mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}_1}, \mathbf{S}_{\boldsymbol{\pi}_2})$ **do**
  if u is `column` **then**
    $(\mathbf{Q}, \mathbf{R}) \leftarrow$ QR-decomposition of $\mathbf{X}[R_P, P]$
    $\tilde{\mathbf{X}}[R_P, P] \leftarrow \mathbf{Q}$
    $\tilde{\mathbf{Y}}[P, C_P] \leftarrow \mathbf{R}\mathbf{Y}[P, C_P]$
  else if u is `row` **then**
    $(\mathbf{Q}, \mathbf{R}) \leftarrow$ QR-decomposition of $\mathbf{Y}[P, C_P]^\top$
    $\tilde{\mathbf{X}}[R_P, P] \leftarrow \mathbf{X}[R_P, P]\mathbf{R}^\top$
    $\tilde{\mathbf{Y}}[P, C_P] \leftarrow \mathbf{Q}^\top$
  end if
end for
return $(\tilde{\mathbf{X}}, \tilde{\mathbf{Y}})$

Algorithm 7.7 Generic factorization method for all chainable architectures

Require: $\mathbf{A} \in \mathbb{C}^{m \times n}$, chainable $\boldsymbol{\beta} = (\boldsymbol{\pi}_\ell)_{\ell=1}^{L}$
Ensure: $(\mathbf{X}_\ell)_{\ell=1}^{L} \in \Sigma^{\boldsymbol{\beta}}$
$\boldsymbol{\beta}' \leftarrow$ Algorithm 7.2$(\boldsymbol{\beta})$ ▷ Remove redundancy of $\boldsymbol{\beta}$
$L' \leftarrow |\boldsymbol{\beta}'|$
Pick σ' a permutation of $[\![L' - 1]\!]$
$(\mathbf{X}'_\ell)_{\ell=1}^{L'} \leftarrow$ Algorithm 7.5$(\mathbf{A}, \boldsymbol{\beta}', \sigma')$
Construct $(\mathbf{X}_\ell)_{\ell=1}^{L} \in \Sigma^{\boldsymbol{\beta}}$ such that $\mathbf{X}_1 \dots \mathbf{X}_L = \mathbf{X}'_1 \dots \mathbf{X}'_{L'}$, e.g., via successive applications of Algorithm 7.1
return $(\mathbf{X}_\ell)_{\ell=1}^{L}$

$\prod_{\ell=1}^{L'} \mathbf{X}'_\ell$. Therefore, the following section only discusses the theoretical guarantees of Algorithm 7.5 for non-redundant chainable architectures, since they will automatically provide guarantees to Algorithm 7.7.

7.6.3 Complexity analysis

It is not hard to see that all versions of the hierarchical algorithms (Algorithms 7.3 to 7.5 and 7.7) have polynomial complexity with respect to the sizes of the butterfly factors and the target matrix, since they only perform a polynomial number of standard matrix operations such as matrix multiplication, QR and SVD decompositions.

Theorem 7.2 (Complexity analysis). *Consider a chainable architecture* $\beta = (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ *and a target matrix* $\mathbf{A}$ *of size* $m \times n$. *Define* $M_\beta := \max_{\ell \in [\![L]\!]} a_\ell c_\ell$, $N_\beta = \max_{\ell \in [\![L]\!]} b_\ell d_\ell$. *Denote* $\|\mathbf{q}(\beta)\|_1, \|\mathbf{q}(\beta)\|_\infty$ *the* ℓ_1 *and* ℓ_∞ *norm of the vector* $\mathbf{q}(\beta)$, *cf. Definition 7.5. Then, the complexity is bounded by:*

- $\mathcal{O}(\|\mathbf{q}(\beta)\|_1 M_\beta N_\beta)$ *for Algorithms 7.3 and 7.4 in general, and* $\mathcal{O}(\|\mathbf{q}(\beta)\|_1 mn)$ *when* β *is not redundant;*
- $\mathcal{O}\left((\|\mathbf{q}(\beta)\|_1 + |\beta|^2 \|\mathbf{q}(\beta)\|_\infty) mn\right)$ *for Algorithm 7.5 when* β *is not redundant;*
- $\mathcal{O}\left((\|\mathbf{q}(\beta)\|_1 + |\beta|^2 \|\mathbf{q}(\beta)\|_\infty) mn + \|\mathbf{q}(\beta)\|_1 M_\beta N_\beta\right)$ *for Algorithm 7.7.*

The proof of Theorem 7.2 is in Appendix C.4. The complexity bounds in Theorem 7.2 are generic for any matrix size m, n, chainable β and factor-bracketing tree $\mathcal{T}$ (or equivalent permutation σ). They can be improved for specific β. For example, in the case of the square dyadic butterfly, [213, 392] showed that the complexity of Algorithm 7.3 is $\mathcal{O}(n^2)$ where $n = 2^L$ instead of $\mathcal{O}(\|\mathbf{q}(\beta)\|_1 n^2) = \mathcal{O}(n^2 \log n)$. This is optimal in the sense that it already matches the space complexity of the target matrix.

7.7 Guarantees on approximation error

One of the main contributions of this chapter is to show that Algorithm 7.5 outputs an approximation solution to Problem (7.1) that satisfies an error bound of the type (7.3).

7.7.1 Main results

Our error bounds are based on the following relaxed problem.

Definition 7.7 (First level factorization). *Given a chainable* $\beta := (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ *with* $L \geq 2$, *we define for each splitting index* $s \in [\![L-1]\!]$ *the two-factor "split" architecture:*

$$\beta_s := (\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_L).$$

When $L = 2$ *we have* $\beta_1 = \beta$. *For any target matrix* $\mathbf{A}$ *we consider the problem*

$$E^{\beta_s}(\mathbf{A}) := \min_{(\mathbf{X},\mathbf{Y}) \in \Sigma^{\beta_s}} \|\mathbf{A} - \mathbf{X}\mathbf{Y}\|_F = \min_{\mathbf{B} \in \mathcal{B}^{\beta_s}} \|\mathbf{A} - \mathbf{B}\|_F. \tag{7.16}$$

The following two theorems are the central theoretical results of this chapter. The first one bounds the approximation error of Algorithm 7.5 in the general case where σ can be any permutation. The second one is a tighter bound specific to the case where σ is the identity permutation.

Theorem 7.3 (Approximation error, arbitrary permutation σ, Algorithm 7.5)**.** *Let β be a* non-redundant chainable *architecture of depth $L \geq 2$. For any target matrix $\mathbf{A}$ and permutation σ of $[\![L-1]\!]$ with $L = |\beta|$, Algorithm 7.5 with inputs $(\mathbf{A}, \beta, \sigma)$ returns butterfly factors $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$ such that*

$$\|\mathbf{A} - \mathbf{X}_1 \dots \mathbf{X}_L\|_F \leq \sum_{k=1}^{L-1} 2^{L-1-k} E^{\beta_{\sigma_k}}(\mathbf{A}).$$

Theorem 7.4 (Approximation error, identity permutation σ, Algorithm 7.5)**.** *Under the same assumptions and notations of Theorem 7.3, Algorithm 7.5 with inputs $(\mathbf{A}, \beta, \sigma)$ where $\sigma = (1, \dots, L-1)$ is the identity permutation returns butterfly factors $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$ such that:*

$$\|\mathbf{A} - \mathbf{X}_1 \dots \mathbf{X}_L\|_F^2 \leq 3^{L-2} E^{\beta_1}(\mathbf{A})^2 + 2 \sum_{k=2}^{L-1} 3^{L-1-k} E^{\beta_k}(\mathbf{A})^2.$$

For $L = 2$ both results yield $\|\mathbf{A} - \mathbf{X}_1\mathbf{X}_2\|_F \leq E^\beta(\mathbf{A})$, i.e. the algorithm is optimal.

Before proving these theorems in Section 7.7.5 we state and prove their main consequences: the quasi-optimality of Algorithm 7.5, and a "complementary low-rank" characterization of butterfly matrices.

7.7.2 Quasi-optimality of Algorithm 7.5

A consequence of the theorems is that butterfly factors obtained via Algorithm 7.5 satisfy an error bound (7.3).

Corollary 7.1 (Quasi-optimality of Algorithm 7.5)**.** *Let β be* any chainable *architecture of arbitrary depth $L := |\beta| \geq 1$. For any target matrix $\mathbf{A}$, the outputs $(\mathbf{X}_\ell)_{\ell=1}^L$ of Algorithm 7.5 with inputs $(\mathbf{A}, \beta, \sigma)$ for arbitrary permutation σ satisfy:*

$$\|\mathbf{A} - \mathbf{X}_1 \dots \mathbf{X}_L\|_F \leq (2^{\max(L-1,1)} - 1) E^\beta(\mathbf{A}). \tag{7.17}$$

When σ is the identity permutation, the outputs also satisfy the finer bound:

$$\|\mathbf{A} - \mathbf{X}_1 \dots \mathbf{X}_L\|_F \leq \sqrt{2 \times 3^{\max(L-2,0)} - 1}\, E^\beta(\mathbf{A}). \tag{7.18}$$

For $L \in \{1, 2\}$ the output of Algorithm 7.5 is thus indeed optimal.

As opposed to (7.4), the error bounds in this corollary compare the approximation error to the *minimal possible* error. Table 7.2 summarizes the consequences of Corollary 7.1 for some standard examples of chainable β. It is worth emphasizing that although the constant C_β is *exponential* with respect to $L = |\beta|$, it is *linear*

or even *sublinear* with respect to the size of the target matrix in most practical cases.

The proof of Corollary 7.1 is based on the following lemma. First, we use the fact that the errors in (7.16) lower bound the error of (7.3), by Definition 7.7.

Lemma 7.9. *If the architecture β of depth $L := |\beta|$ is chainable then*

$$\forall s \in [\![L-1]\!], \quad \mathcal{B}^{\beta} \subseteq \mathcal{B}^{\beta_s}. \tag{7.19}$$

Consequently, for any matrix $\mathbf{A}$ the quantity $E^{\beta}(\mathbf{A})$ defined in (7.1) satisfies:

$$E^{\beta}(\mathbf{A}) \geq \max_{1 \leq s \leq L-1} E^{\beta_s}(\mathbf{A}). \tag{7.20}$$

Proof. If $\mathbf{B} \in \mathcal{B}^{\beta}$, then there exist $(\mathbf{X}_\ell)_{\ell=1}^{L} \in \Sigma^{\beta}$ such that $\mathbf{B} = \mathbf{X}_1 \dots \mathbf{X}_L$. By Lemma 7.4, $\mathbf{X}_1 \dots \mathbf{X}_s \in \Sigma^{(\pi_1 * \dots \pi_s)}$ and $\mathbf{X}_{s+1} \dots \mathbf{X}_L \in \Sigma^{(\pi_{s+1} * \dots * \pi_L)}$ for any $s \in [\![L-1]\!]$. □

Proof of Corollary 7.1. We start by proving (7.17). We consider two possibilities for the depth $L := |\beta|$ of the non-redundant, chainable architecture β:

- If $L = 1$: we have $\beta = \{\pi\}$ for some pattern π. The projection of $\mathbf{A}$ onto Σ^{π} is simply $\mathbf{A} \odot \mathbf{S}_{\pi} \in \Sigma^{\pi}$, which is exactly the output computed by the algorithm. Hence the obtained factor $\mathbf{X}_1$ satisfies $\|\mathbf{A} - \mathbf{X}_1\|_F = E^{\beta}(\mathbf{A})$.
- If $L \geq 2$: using Lemma 7.9, Theorem 7.3, and the fact that $\sum_{k=1}^{L-1} 2^{L-1-k} = \sum_{k=0}^{L-2} 2^k = 2^{L-1} - 1$, we have: $\|\mathbf{A} - \mathbf{X}_1 \dots \mathbf{X}_L\|_F \leq (2^{L-1} - 1) E^{\beta}(\mathbf{A})$.

In both cases, we have $\|\mathbf{A} - \mathbf{X}_1 \dots \mathbf{X}_L\|_F \leq (2^{\max(L-1,1)} - 1) E^{\beta}(\mathbf{A})$. The proof for (7.18) is similar, the only difference being that we use the equality: $3^{L-2} + 2\sum_{k=2}^{L-1} 3^{L-1-k} = 2 \times 3^{L-2} - 1$. □

Remark 7.5. *Because the generic factorization method - Algorithm 7.7 - uses Algorithm 7.5, it inherits naturally the bounds (7.17) and (7.18), depending on the choice of σ' chosen in Line 3 of Algorithm 7.7. In fact, the bounds in Corollary 7.1 for a redundant architecture can be even tighter since the depth L' of its non-redundant counterpart (β' - output of Algorithm 7.2) is strictly smaller than L.*

7.7.3 Complementary low-rank characterization of butterfly matrices

Another important consequence of Theorem 7.3 is a characterization of matrices admitting an exact butterfly factorization associated with a chainable β. This allows (when β is chainable) to *verify* whether or not a given matrix $\mathbf{A}$ admits a butterfly factorization associated with β, by checking the rank of a polynomial number of specific submatrices of $\mathbf{A}$. This is feasible using SVDs, and contrasts

with the *synthesis* definition of $\mathcal{B}^{\boldsymbol{\beta}}$ given by (7.2), which is a priori harder to verify since it requires checking the *existence* of an exact factorization of $\mathbf{A}$.

Definition 7.8 (Generalized complementary low-rank property)**.** *A matrix* $\mathbf{A}$ *satisfies the* generalized complementary-low rank property *associated with a chainable architecture* $\boldsymbol{\beta} := (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ *if, denoting* $\mathbf{S}_{r,t} := \mathbf{S}_{\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_t}$ *for* $1 \leq r \leq t \leq L$*, it satisfies:*

1. $\operatorname{supp}(\mathbf{A}) \subseteq \mathbf{S}_{1,L}$;
2. $\operatorname{rank}(\mathbf{A}[R_P, C_P]) \leq q(\boldsymbol{\pi}_\ell, \boldsymbol{\pi}_{\ell+1})$ *for each* $P \in \mathcal{P}(\mathbf{S}_{1,\ell}, \mathbf{S}_{\ell+1,L})$ *and* $\ell \in [\![L-1]\!]$ *(with the notations of Definition 7.1, Definition 7.4).*

We show in Corollary C.2 of Appendix C.6 that this generalized definition indeed coincides with the classical definition of a complementary low-rank property (Definition 3.7) from the literature [234], for every architecture $\boldsymbol{\beta}$ with patterns such that $a_1 = d_L = 1$, i.e., architectures such that $\mathcal{B}^{\boldsymbol{\beta}}$ contains some dense matrices, see Remark 7.4. The following results show that a matrix admits an exact butterfly factorization associated with $\boldsymbol{\beta}$ if, and only if, it satisfies the associated generalized complementary low-rank property.

Corollary 7.2 (Characterization of $\mathcal{B}^{\boldsymbol{\beta}}$ for chainable $\boldsymbol{\beta}$)**.** *If* $\boldsymbol{\beta} := (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ *is chainable with* $L \geq 2$ *then, with the notations of Definition 7.7 and Lemma 7.2:*

$$\mathcal{B}^{\boldsymbol{\beta}} = \bigcap_{\ell=1}^{L-1} \mathcal{B}^{\boldsymbol{\beta}_\ell} = \Sigma^{(\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_L)} \cap \bigcap_{\ell=1}^{L-1} \mathcal{A}^{\boldsymbol{\beta}_\ell}. \tag{7.21}$$

Proof. The second equality in (7.21) is a reformulation based on Lemma 7.2, so it only remains to prove the first equality. The inclusion $\mathcal{B}^{\boldsymbol{\beta}} \subseteq \bigcap_{\ell=1}^{L-1} \mathcal{B}^{\boldsymbol{\beta}_\ell}$ is a consequence of Lemma 7.9. We now prove the other inclusion.

First consider the case of a non-redundant $\boldsymbol{\beta}$. If $\mathbf{A} \in \bigcap_{\ell=1}^{L-1} \mathcal{B}^{\boldsymbol{\beta}_\ell}$, then $E^{\boldsymbol{\beta}_\ell}(\mathbf{A}) = 0$ for each $\ell \in [\![L-1]\!]$ by Definition 7.7. By Theorem 7.3, Algorithm 7.5 with inputs $(\mathbf{A}, \boldsymbol{\beta}, \sigma)$ for arbitrary permutation σ returns $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^{\boldsymbol{\beta}}$ such that $\|\mathbf{A} - \mathbf{X}_1 \ldots \mathbf{X}_L\|_F = 0$, thus $\mathbf{A} = \mathbf{X}_1 \ldots \mathbf{X}_L \in \mathcal{B}^{\boldsymbol{\beta}}$. This proves $\bigcap_{\ell=1}^{L-1} \mathcal{B}^{\boldsymbol{\beta}_\ell} \subseteq \mathcal{B}^{\boldsymbol{\beta}}$.

For redundant $\boldsymbol{\beta}$, consider $\boldsymbol{\beta}'$ returned by Algorithm 7.2 with input $\boldsymbol{\beta}$. By Proposition 7.2: $\mathcal{B}^{\boldsymbol{\beta}'} = \mathcal{B}^{\boldsymbol{\beta}}$. Moreover, by the same proposition, $\boldsymbol{\beta}'$ is of the form $(\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_{\ell_1}, \boldsymbol{\pi}_{\ell_1+1} * \ldots * \boldsymbol{\pi}_{\ell_2}, \ldots, \boldsymbol{\pi}_{\ell_p+1} * \ldots * \boldsymbol{\pi}_L)$ for some indices $1 \leq \ell_1 < \ldots < \ell_p < L$ with $p \in [\![L-1]\!]$. Therefore, for any $s' \in [\![L'-1]\!]$, there exists $s \in [\![L-1]\!]$ such that $\boldsymbol{\beta}_s = \boldsymbol{\beta}'_{s'}$, by associativity of the operator $*$ (Lemma 7.3). Thus,

$$\bigcap_{\ell=1}^{L-1} \mathcal{B}^{\boldsymbol{\beta}_\ell} \subseteq \bigcap_{\ell'=1}^{L'-1} \mathcal{B}^{\boldsymbol{\beta}'_{\ell'}} = \mathcal{B}^{\boldsymbol{\beta}'} = \mathcal{B}^{\boldsymbol{\beta}}$$

where in the first equality we used the result proved above for non-redundant β'. □

7.7.4 Existence of an optimum

Corollary 7.2 also allows to prove the existence of optimal solutions for Problem (7.1) when β is chainable.

Corollary 7.3 (Existence of optimum in butterfly approximation)**.** *If β is chainable then, for any target matrix $\mathbf{A}$, Problem (7.1) admits a minimizer.*

Proof. The set of matrices of rank smaller than a fixed constant is closed, and closed sets are stable under finite intersection, so by the characterization of $\mathcal{B}^\beta$ from Corollary 7.2, the set $\mathcal{B}^\beta$ is closed. Therefore, Problem (7.1), which is equivalent to a projection problem on the closed set $\mathcal{B}^\beta$, always admits a minimizer. □

The rest of the section is dedicated to the proofs of Theorems 7.3 and 7.4. Readers more interested in numerical aspects of the proposed hierarchical algorithms can directly jump to Section 7.8.

7.7.5 Proof of Theorems 7.3 and 7.4

Consider an iteration number $J \in [\![L-1]\!]$, and denote $(P_k)_{k=1}^J$ the partition obtained after line 7 and $(\mathbf{X}_{P_k})_{k=1}^J$ the list `factors` obtained *after* the orthonormalization operations in lines 10-15, at the J-th iteration of Algorithm 7.5. With j defined in line 9 and $[\![r,t]\!] := P_j$, $s := \sigma_J$, denote

$$\mathbf{X}_{\text{left}}^{(J)} := \mathbf{X}_{P_1} \dots \mathbf{X}_{P_{j-1}}, \tag{7.22}$$

$$\mathbf{X}_{\text{right}}^{(J)} := \mathbf{X}_{P_{j+1}} \dots \mathbf{X}_{P_J}, \tag{7.23}$$

with the convention that $\mathbf{X}_{\text{left}}^{(J)}$ (resp. $\mathbf{X}_{\text{right}}^{(J)}$) is the identity matrix of size $a_1 b_1 d_1$ if $j = 1$ (resp. $a_L c_L d_L$ if $j = J$). We also denote $(\mathbf{X}_{[\![r,s]\!]}, \mathbf{X}_{[\![s+1,t]\!]})$ the matrices computed in line 17, $\mathbf{B}_J := \mathbf{X}_{\text{left}}^{(J)} \mathbf{X}_{[\![r,s]\!]} \mathbf{X}_{[\![s+1,t]\!]} \mathbf{X}_{\text{right}}^{(J)}$ and $R_J := \|\mathbf{A} - \mathbf{B}_J\|_F$.

Note that $\mathbf{B}_J$ is the product of butterfly factors in the list `factors` at the end of the iteration J (line 19). In particular, $\mathbf{B}_{L-1} \in \mathcal{B}^\beta$ is the product of the butterfly factors returned by the algorithm after $L-1$ iterations. By convention we also define $\mathbf{B}_0 := \mathbf{A}$ and $R_0 := 0$.

When σ is an arbitrary permutation, we will prove below the following relationship between the residual errors R_{J-1} and R_J:

$$\forall J \in [\![L-1]\!], \quad R_J \leq 2R_{J-1} + E^{\beta_{\sigma_J}}(\mathbf{A}). \tag{7.24}$$

By an immediate recursion this will yield the desired bound $R_{L-1} := \|\mathbf{A} - \mathbf{B}_{L-1}\|_F \leq \sum_{J=1}^{L-1} 2^{L-1-J} E^{\beta_{\sigma_J}}(\mathbf{A})$ and therefore prove Theorem 7.3.

When σ is the identity permutation, i.e., $\sigma_J = J$ for $J \in [\![L-1]\!]$, we will prove the following inequality:

$$\forall J \in [\![2, L-1]\!], \quad R_J^2 \leq 3R_{J-1}^2 + 2E^{\beta_J}(\mathbf{A})^2. \tag{7.25}$$

Then, by definition, $R_1 = E^{\beta_1}(\mathbf{A})$, since $\sigma_1 = 1$ (σ_1 is the identity permutation). By induction, this yields $R_{L-1}^2 \leq 3^{L-2}E^{\beta_1}(\mathbf{A})^2 + 2\sum_{k=2}^{L-1} 3^{L-1-J}E^{\beta_J}(\mathbf{A})^2$, and proves Theorem 7.4.

The rest of this section is dedicated to the proof of (7.24) and (7.25).

Proof of (7.24). Let $J \in [\![L-1]\!]$. By the triangle inequality we have

$$R_J = \|\mathbf{A} - \mathbf{B}_J\|_F \leq \|\mathbf{A} - \mathbf{B}_{J-1}\| + \|\mathbf{B}_{J-1} - \mathbf{B}_J\| = R_{J-1} + \|\mathbf{B}_{J-1} - \mathbf{B}_J\|_F. \tag{7.26}$$

Moreover $\mathbf{B}_{J-1} = \mathbf{X}_{\text{left}}^{(J)}\mathbf{X}_{[\![r,t]\!]}\mathbf{X}_{\text{right}}^{(J)}$: this holds by definition when $J = 1$, and for $J \geq 2$ this remains the case by construction of Algorithm 7.5. Indeed, the butterfly factors in the list `factors` at the beginning of iteration J (line 8) are the same as the ones in the list `factors` at the end of the previous iteration $J-1$ (line 19), and their product does not change after the orthonormalization operations of iteration J, because (as formally proved in Lemma C.11 in Appendix C.3.3), each call to Algorithm 7.6 in lines 11 and 14 of Algorithm 7.5 preserves the product of the butterfly factors in the list `factors`.

As shown in Appendices C.3.4 and C.5.1, the following properties hold: for any $J \in [\![L-1]\!]$, we have:

$$\|\mathbf{X}_{\text{left}}^{(J)}(\mathbf{X}_{[\![r,t]\!]} - \mathbf{X}_{[\![r,s]\!]}\mathbf{X}_{[\![s+1,t]\!]})\mathbf{X}_{\text{right}}^{(J)}\|_F = \|\mathbf{X}_{[\![r,t]\!]} - \mathbf{X}_{[\![r,s]\!]}\mathbf{X}_{[\![s+1,t]\!]}\|_F, \tag{7.27}$$

$$E^{\beta_s}(\mathbf{X}_{\text{left}}^{(J)}\mathbf{X}_{[\![r,t]\!]}\mathbf{X}_{\text{right}}^{(J)}) = \|\mathbf{X}_{[\![r,t]\!]} - \mathbf{X}_{[\![r,s]\!]}\mathbf{X}_{[\![s+1,t]\!]}\|_F. \tag{7.28}$$

Therefore, combining (7.27) and (7.28) with the definition of $\mathbf{B}_J$, we obtain

$$\begin{aligned}\|\mathbf{B}_{J-1} - \mathbf{B}_J\|_F &= \|\mathbf{X}_{\text{left}}^{(J)}(\mathbf{X}_{[\![r,t]\!]} - \mathbf{X}_{[\![r,s]\!]}\mathbf{X}_{[\![s+1,t]\!]})\mathbf{X}_{\text{right}}^{(J)}\|_F \\ &\overset{(7.27)}{=} \|\mathbf{X}_{[\![r,t]\!]} - \mathbf{X}_{[\![r,s]\!]}\mathbf{X}_{[\![s+1,t]\!]}\|_F \overset{(7.28)}{=} E^{\beta_s}(\mathbf{B}_{J-1}).\end{aligned}$$

Let $\mathbf{A}_s$ be an optimal solution of the first level factorization for index $s = \sigma_J$ and for the target matrix $\mathbf{A}$ (cf. Definition 7.7). In particular, $\mathbf{A}_s \in \mathcal{B}^{\beta_s}$. Then, by definition of $E^{\beta_s}(\mathbf{B}_{J-1})$:

$$\|\mathbf{B}_{J-1} - \mathbf{B}_J\|_F = E^{\beta_s}(\mathbf{B}_{J-1}) \leq \|\mathbf{B}_{J-1} - \mathbf{A}_s\|_F.$$

By the triangle inequality and by definition of $\mathbf{A}_s$:

$$\begin{aligned}\|\mathbf{B}_{J-1} - \mathbf{B}_J\|_F &\leq \|\mathbf{B}_{J-1} - \mathbf{A}_s\|_F \\ &\leq \|\mathbf{B}_{J-1} - \mathbf{A}\|_F + \|\mathbf{A} - \mathbf{A}_s\|_F = R_{J-1} + E^{\beta_s}(\mathbf{A}).\end{aligned} \tag{7.29}$$

Combining (7.26) and (7.29) yields (7.24), which ends the proof. □

Proof of (7.25). Suppose now that σ is the identity permutation. As proved in Appendix C.5.2 using an orthogonality argument[3], we have

$$\forall J \in [\![L-1]\!], \quad \forall p \in [\![J, L-1]\!], \quad \langle \mathbf{B}_{J-1} - \mathbf{B}_J, \mathbf{B}_p \rangle = 0. \tag{7.30}$$

Hence:

$$\begin{aligned} \forall J \in [\![L-1]\!], \quad R_J^2 := \|\mathbf{A} - \mathbf{B}_J\|_F^2 &= \|(\mathbf{B}_0 - \mathbf{B}_1) + \ldots + (\mathbf{B}_{J-1} - \mathbf{B}_J)\|_F^2 \\ &= \sum_{i=1}^{J} \|\mathbf{B}_{i-1} - \mathbf{B}_i\|_F^2 + 2 \sum_{i=1}^{J} \sum_{i'>i} \langle \mathbf{B}_{i-1} - \mathbf{B}_i, \mathbf{B}_{i'-1} - \mathbf{B}_{i'} \rangle \\ &\overset{(7.30)}{=} \|\mathbf{B}_0 - \mathbf{B}_1\|_F^2 + \ldots + \|\mathbf{B}_{J-1} - \mathbf{B}_J\|_F^2. \end{aligned}$$

Since $R_0 := 0$, the above implies that for each $J \in [\![L-1]\!]$.

$$R_J^2 = R_{J-1}^2 + \|\mathbf{B}_{J-1} - \mathbf{B}_J\|_F^2, \tag{7.31}$$

which is tighter than the triangular inequality used in (7.26). By (7.29) and the inequality $(a+b)^2 \leq 2(a^2+b^2)$, we obtain $R_J^2 \leq 3R_{J-1}^2 + 2E^{\beta_s}(\mathbf{A})^2$, as desired. □

7.8 Numerical experiments

We now illustrate the empirical behaviour of the proposed hierarchical algorithm for Problem (7.1). All methods are implemented in Python 3.9.7 using the `PyTorch 2.2.1` package. Experiments are conducted on an Intel(R) Xeon(R) Gold 6338 CPU @ 2,00 GHz (32 cores, 125 GB RAM), in float-precision. Since all the target matrices in the experiment are of real values, we implemented all algorithms with real entries (instead of complex ones as in Algorithms 7.1 and 7.3 to 7.5).

7.8.1 Hierarchical algorithm *vs.* with existing methods

We consider Problem (7.1) associated with the square dyadic butterfly architecture with $L = 10$ factors. The target matrix is the Hadamard matrix of size 1024×1024. We compare the following methods.

- **Hierarchical algorithm (Algorithm 7.4 and Algorithm 7.5)**: We use the permutation $\sigma = (5, 2, 1, 3, 4, 7, 6, 8, 9)$, which corresponds to a balanced factor-bracketing tree as illustrated in Figure 6.2b. In line 3 of Algorithm 7.1, the best low-rank approximation is computed via truncated SVD: following [213], we compute the full[4] SVD $\mathbf{UDV}^\top$ of the submatrix $\mathbf{A}[R_P, C_P]$

[3] This argument has been used in [250] to prove (7.4). We adapt their argument to our context for the self-containedness of this chapter.

[4] One can further optimize the algorithm by running truncated SVD instead of full SVD, as discussed in Section 6.5.2.

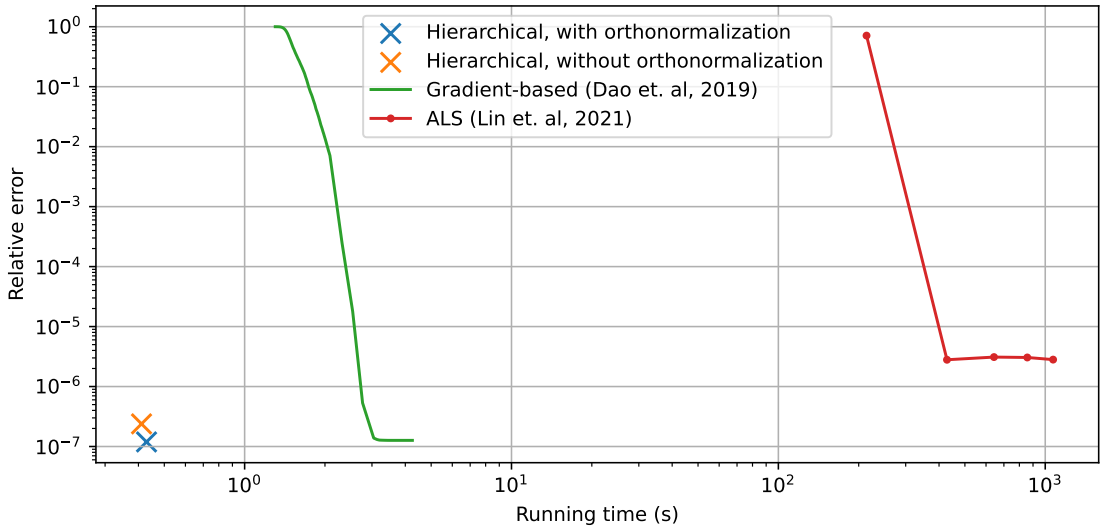


Figure 7.3: Relative approximation errors defined as $\|\mathbf{A} - \hat{\mathbf{A}}\|_F / \|\mathbf{A}\|_F$ *vs.* running time of the different algorithms. The target matrix $\mathbf{A}$ is the Hadamard matrix of size 1024×1024, and $\hat{\mathbf{A}}$ is the computed approximation for Problem (7.1) associated with the square dyadic butterfly architecture.

where the diagonal entries of $\mathbf{D}$ are the singular values in decreasing order, and we set $\mathbf{H} = \mathbf{U}[:, r]\mathbf{D}^{1/2}$ and $\mathbf{K} = \mathbf{D}^{1/2}\mathbf{V}[:, r]^\top$ with $r := [\![|P|]\!]$. Note that we used Algorithm 7.5 instead of Algorithm 7.7 because we know that the considered architecture β is not redundant[5].

- **Gradient-based method** [74]: Using the parameterization $\mathbf{B} = \mathbf{X}_1 \dots \mathbf{X}_L$ for a butterfly matrix $\mathbf{B} \in \mathcal{B}^\beta$, this method uses (variants of) gradient descent to optimize all nonzero entries $\mathbf{X}_\ell[i, j]$ for $(i, j) \in \mathrm{supp}(\mathbf{S}_{\pi_\ell})$ and $\ell \in [\![L]\!]$ to minimize (7.1). We use the protocol of [74]: we perform 100 iterations of ADAM[6] [197], followed by 20 iterations of L-BFGS [244][7].
- **Alternating least squares** [241]: At each iteration of this iterative algorithm, we optimize the nonzero entries of a given factor $\mathbf{X}_\ell$ for some $\ell \in [\![L]\!]$ while fixing the others, by solving a linear regression problem.

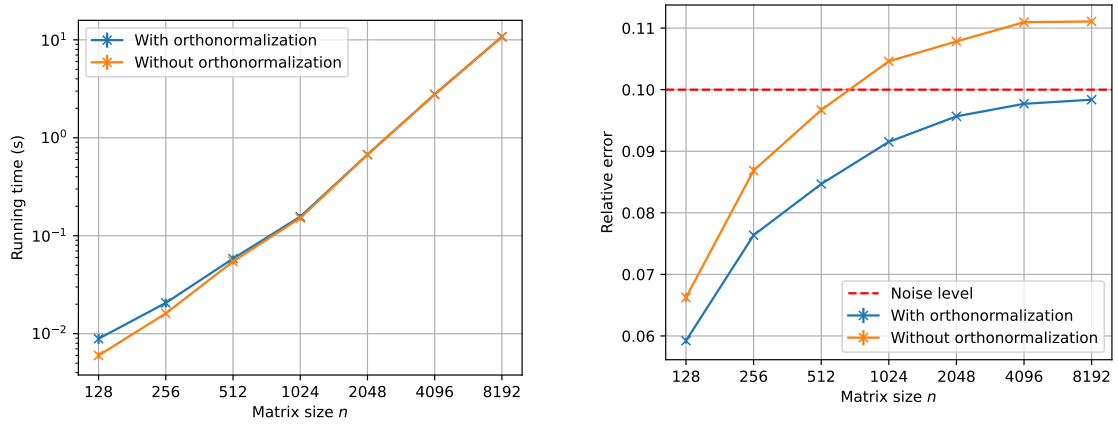
Figure 7.3 shows that the different methods find an approximate solution nearly up to machine precision[8], but hierarchical algorithms are several orders of magnitude faster than the gradient-based method [74] and ALS [241]. Algorithm 7.4 is also faster than Algorithm 7.5 since it does not perform the additional orthonormalization operations.

[5] With a redundant architecture β, we would have to use Algorithm 7.7, incurring an overhead compared to Algorithm 7.5 due to line 5, but it should be negligeable according to Theorem 7.2.

[6] The learning rate is set as 0.1, and we choose $(\beta_1, \beta_2) = (0.9, 0.999)$.

[7] L-BFGS terminates when the norm of the gradient is smaller than 10^{-7}.

[8] Yet, hierarchical algorithms and gradient-based methods are more accurate than ALS by more than one order of magnitude.



(a) Running time *vs.* size n

(b) Relative error $\frac{\|\mathbf{A}-\hat{\mathbf{A}}\|_F}{\|\mathbf{A}\|_F}$ *vs.* size n

Figure 7.4: Running time and the relative approximation errors *vs.* the matrix size n, for Algorithm 7.4 (without orthonormalization) and Algorithm 7.5 (with orthonormalization), for the instance of Problem (7.1) described in Section 7.8.2 with $r = 4$. We show mean and standard deviation on the error bars over 10 repetitions of the experiment.

7.8.2 To orthonormalize or not to orthonormalize?

We now study in practice the impact of the orthonormalization operations in the hierarchical algorithm, in terms of running time and approximation error, at different scales of the matrix size $n \times n$ with $n \in \{128, 256, \ldots, 8192\}$. We consider Problem (7.1) of size $n \times n$ associated with a chainable architecture $\beta = (\boldsymbol{\pi}_\ell)_{\ell=1}^{L}$ defined as the unique chainable architecture of depth $L = 4$ that minimizes the number of parameters $\|\beta\|_0$ under the constraint that:

- Each factor is of size $(a_\ell b_\ell d_\ell, a_\ell c_\ell d_\ell) = (n, n)$
- $\mathcal{B}^\beta$ contains some dense matrices: $\boldsymbol{\pi}_1 * \boldsymbol{\pi}_2 * \boldsymbol{\pi}_3 * \boldsymbol{\pi}_4 = (1, n, n, 1)$;
- In the complementary low-rank characterization of $\mathcal{B}^\beta$, the rank constraint on the submatrices is $r \geq 2$, i.e., $\mathbf{q}(\beta) = (r, r, r)$. We do not choose $r = 1$ because otherwise, the orthonormalization operations are equivalent to some rescaling.

The considered target matrix is $\mathbf{A} = \tilde{\mathbf{A}} + \epsilon \frac{\|\tilde{\mathbf{A}}\|_F}{\|\mathbf{E}\|_F} \mathbf{E}$, where $\tilde{\mathbf{A}} := \mathbf{X}_1\mathbf{X}_2\mathbf{X}_3\mathbf{X}_4$, the entries of $\mathbf{X}_\ell \in \Sigma^{\boldsymbol{\pi}_\ell}$ for $\ell \in [\![4]\!]$ are i.i.d sampled from the uniform distribution in the interval $[0, 1]$, $\mathbf{E}$ is an i.i.d centered Gaussian matrix with the standard deviation 1, and $\epsilon \geq 0$ is the noise level fixed as $\epsilon = 0.1$ in our experiment. The permutation σ for the hierarchical algorithm is $\sigma = (2, 1, 3)$, which corresponds to the *balanced* factor-bracketing tree of $[\![4]\!]$, defined as the binary tree where the nodes of a same level have the same cardinal.

Figure 7.4a shows that the difference in running time between the hierarchical algorithm with (Algorithm 7.4) and without (Algorithm 7.3) orthonormalization

is negligible, in the regime of large matrix size $n \geq 512$. This means that, asymptotically, the time of orthonormalization operations is not the bottleneck, which is coherent with our complexity analysis given in Theorem 7.2.

In terms of the approximation error, Figure 7.4b shows that the hierarchical algorithm *with* orthornormalization (Algorithm 7.4) returns a smaller (i.e., better) approximation error. Moreover, the relative error with orthonormalization error is always smaller than the relative noise level $\epsilon = 0.1$ (cf. Appendix C.7 for other values of ϵ), which is not the case of the hierarchical algorithm without orthonormalization (Algorithm 7.3). In conclusion, besides yielding error guarantees of the form (7.3), the orthonormalization operations in our experiments also lead to better approximation in practice.

7.8.3 Comparison of error bounds

We compare numerically the error bound (7.4) from [250] to the new error bound from Theorem 7.3[9]. We show that there exists some target matrices $\mathbf{A}$ for which the bound (7.4) can be very large compared to the one of Theorem 7.4.

Protocol. Consider the square dyadic butterfly architecture $\beta = (\boldsymbol{\pi}_\ell)_{\ell=1}^L = \big((2^{\ell-1}, 2, 2, 2^{L-\ell})\big)_{\ell=1}^L$, which is indeed chainable, with $\mathbf{q}(\beta) = (1, \dots, 1)$. Define the target matrix $\mathbf{A} = \tilde{\mathbf{A}} + \epsilon \frac{\|\tilde{\mathbf{A}}\|_F}{\|\tilde{\mathbf{E}}\|_F}$ in Problem (7.1) where

$$\begin{aligned} \tilde{\mathbf{A}} &= \mathbf{H} \odot (\mathbf{1}_{n\times n} - \mathbf{S}), \\ \tilde{\mathbf{E}} &= \mathbf{E} \odot \mathbf{S}, \end{aligned}$$

where $\mathbf{H} \in \mathcal{B}^\beta$ is the Hadamard matrix of size $n \times n$ with $n = 2^L$, $\mathbf{E}$ is an i.i.d. random Gaussian matrix of size $n \times n$ with the standard deviation 1. Here, $\mathbf{S} \in \{0,1\}^{n\times n}$ is defined as:

$$\forall (i,j) \in [\![n]\!]^2, \quad \mathbf{S}[i,j] = \begin{cases} 1 & \text{if } (i,j) \in R_P \times C_P \\ 0 & \text{otherwise} \end{cases}, \tag{7.32}$$

where we can choose any $P \in \mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}_1}, \mathbf{S}_{\boldsymbol{\pi}_2 * \dots * \boldsymbol{\pi}_L})$. In our experiment, we choose $R_P \times C_P = \{1, 1 + n/2\} \times [\![1, n/2]\!]$. We apply Algorithm 7.5 with one realization of $\mathbf{A}$ and the identity permutation $\sigma = \texttt{id}$ as inputs.

Results. We compare the error bound from Theorem 7.3 with the bound (7.4) from [250] computed as $\sqrt{|\beta|}\epsilon_0 \|\mathbf{A}\|_F$, see (7.4) for the definition of ϵ_0. By design of $\mathbf{A}$, the numerical value of ϵ_0 is expected to be large, because at the first-level factorization, the submatrix $\mathbf{A}[R_P, C_P]$ cannot be well approximated by a rank-one matrix in the procedure of Algorithm 7.1. In Figure 7.5, we observe that for a

[9] Note that the error bound from Corollary 7.1 cannot be evaluated numerically because we do not know how to compute the minimal approximation error.

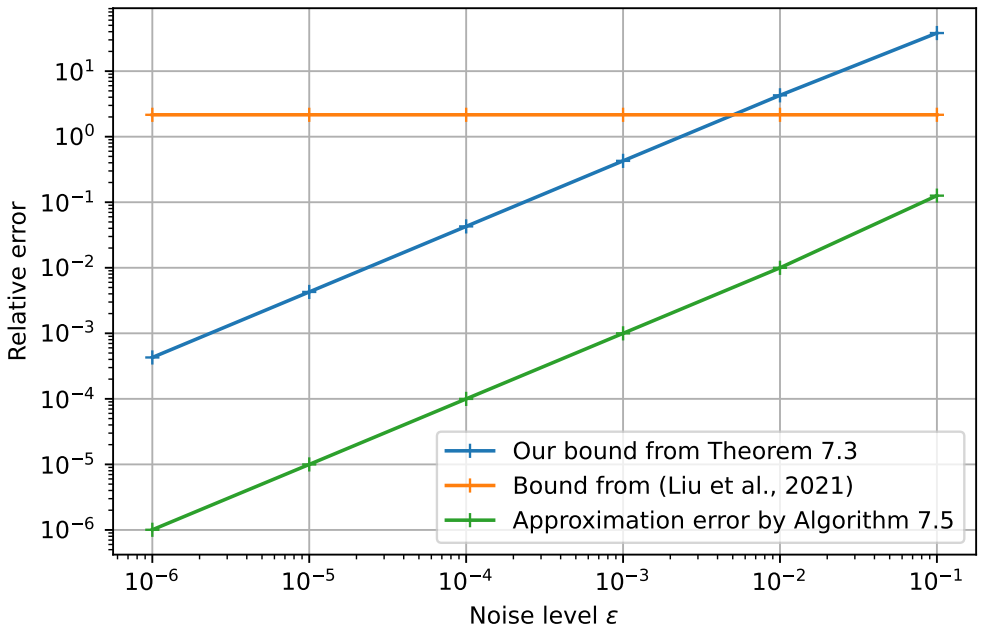


Figure 7.5: Values of the error bounds *vs.* the noise level ϵ. We display the relative error, i.e., we divide the value of the error bound by the norm of the target matrix $\mathbf{A}$. The bound in blue is the one from Theorem 7.3. The bound in orange is the one from (7.4).

sufficiently small noise level ϵ, the error bound from Theorem 7.3 is much smaller than (7.4): even though the constant $C_n = \mathcal{O}(\log n)$ in error bound (7.4) grows slowly with respect to the matrix size n, this example shows that the bound (7.4) is pessimistic since it is proportional to $\epsilon_0 \|\mathbf{A}\|_F$.

7.9 Conclusion

We proposed a general definition of (deformable) butterfly architectures, together with a hierarchical algorithm for the problem of (deformable) butterfly factorization (7.1), endowed with new guarantees on the approximation error of the type (7.3), under the condition that the associated architecture β satisfies a so-called *chainability* condition. The proposed algorithm involves some novel orthonormalization operations in the context of butterfly factorization. We discuss some perspectives of this work.

Tightness of the error bound. The constants C_β in Corollary 7.1 grow *exponentially* with the depth $L = |\beta|$ of the architecture, but linearly with the matrix size n as shown in Table 7.2. Note that the quasi-linear constant C_n in the existing bound (7.4) is not comparable with the constants C_β in Corollary 7.1, due to the presence of ϵ_0 in the bound (7.4), whereas the constants C_β in Corollary 7.1 controls the ratio between the approximation error and the minimal error. A natural

question is whether the constants C_β in Corollary 7.1 are tight for an error bound of the type (7.3). If not, can the bounds for the proposed be algorithm be sharpened by a refined theoretical analysis, or is there another algorithm that yields a smaller constant C_β in the error bound?

Randomized algorithms for low-rank approximation. Algorithms 7.3 to 7.5 need to access all the elements of the target matrix $\mathbf{A} \in \mathbb{C}^{m\times n}$. Thus, the complexity of all algorithms is at least $\mathcal{O}(mn)$. This complexity, however, fails to scale for large m, n (e.g., up to 10^6). Assuming that the target matrix admits a butterfly factorization associated with β, i.e., $\mathbf{A} \in \mathcal{B}^\beta$, is it possible to recover the butterfly factors of $\mathbf{A}$, with a faster algorithm, ideally of complexity $\mathcal{O}(\|\beta\|_0)$? Note that this question was already considered in [233, 234] where randomized algorithms for low-rank approximation [153, 240] are leveraged in the context of butterfly factorization. The question is therefore whether we can still prove some theoretical guarantees of the form (7.3) for butterfly algorithms with such algorithms.

Algorithms beyond the chainability assumption. Although chainability is a sufficient condition for which we can design an algorithm with guarantees on the approximation error, it is natural to ask whether it is also a necessary condition. There exist, in fact, non-chainable architectures for which we can still build an algorithm yielding an error bound (7.3). For instance, this is the case of *arbitrary* architectures of depth $L = 2$ (see Lemma 7.8), or architectures that satisfy a *transposed* version of Definition 7.5, as detailed in Appendix C.8. Therefore, chainability in the sense of Definition 7.5 is not necessary for theoretical guarantees of the form (7.3). However, we conjecture that it is *necessary* to either consider the definition of chainability in the sense of Definition 7.5, or the transpose version of it, in order to preserve the axioms of stability by matrix multiplication (Proposition 7.1) and associativity (Lemma 7.3), which are at the core for our proofs in Section 7.7. In other words, to find a more general condition on β, we need to develop a new proof technique that does not rely on either property. These new techniques, if possible, might allow us to deal with architectures that are neither chainable nor transposed-chainable, such as the Kaleidoscope architecture (cf. Table 7.1).

Chapter 8

Butterfly factorization by algorithmic identification of rank-one submatrices

The previous chapter proposed a quasi-optimal hierarchical algorithm to construct a fast algorithm for an approximate rapid evaluation of the considered matrix, provided that it approximately admits a butterfly factorization associated with some structured *fixed-support constraints* encoded by an architecture β. However, in general, matrices associated with commonly used fast transforms, such as the DFT matrix, admit such a butterfly factorization *up to some specific row or column permutations*. When these permutations are not known, it becomes necessary to approach the problem of butterfly factorization by considering such permutations as part of the optimization problem. This chapter proposes a heuristic to address this more general problem, without any analytical assumption on the entries of the target matrix, for the case of square dyadic butterfly factorizations.

8.1 Introduction

Finding a fast algorithm associated with butterfly factorization when row and column permutations are *not known* is formalized as the following minimization problem:

$$\inf_{(\mathbf{X}_\ell)_{\ell=1}^L, \mathbf{P}, \mathbf{Q}} \|\mathbf{A} - \mathbf{Q}^\top \mathbf{X}_1 \dots \mathbf{X}_L \mathbf{P}\|_F, \tag{8.1}$$

where $\mathbf{P}, \mathbf{Q}$ are permutation matrices. Chapters 6 and 7 focused on instances of Problem (8.1) where, recalling (7.9), the sparse factors satisfy the fixed-support constraint $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$, but with *fixed and known permutations* $\mathbf{P}, \mathbf{Q}$. For such instances, we showed in Chapter 7 that there is an efficient hierarchical algorithm (Algorithm 7.5) that finds quasi-optimal butterfly factors $(\mathbf{X}_\ell)_{\ell=1}^L$ with guarantees on the approximation error (Corollary 7.1), provided that the architecture β

The material of this chapter is based on [386], in collaboration with Gilles Puy, Elisa Riccietti, Patrick Pérez and Rémi Gribonval.

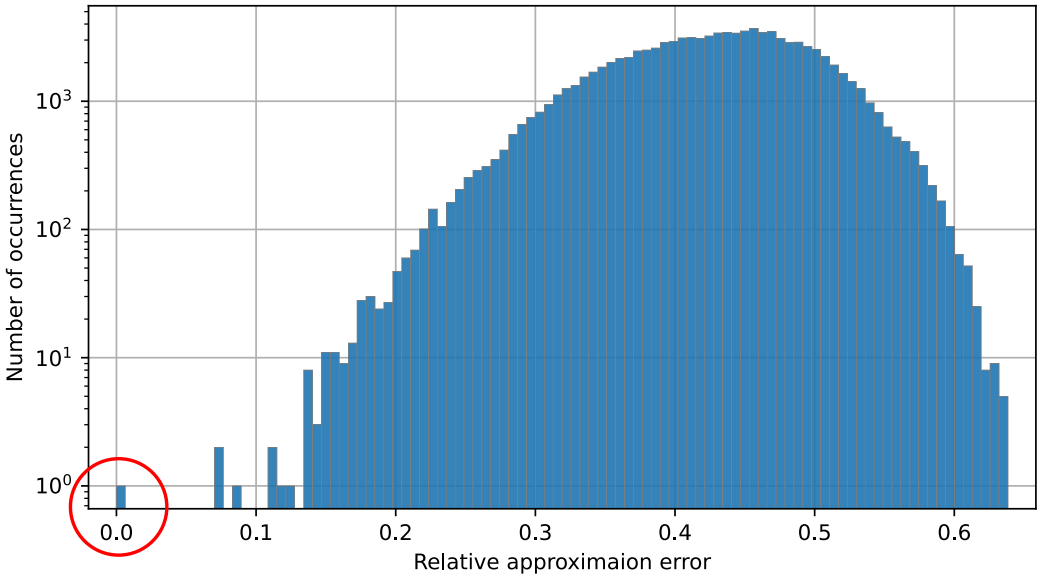


Figure 8.1: Histogram of relative errors when solving Problem (8.1) corresponding to the square dyadic butterfly factorization (Chapter 6) with unknown permutations, by exhaustive search on all possible pairs of row and column permutations, as explained in Section 8.3. The target matrix is of size 8×8, and is assumed to admit exactly a square dyadic butterfly factorization up to some unknown row and column permutations. As indicated by the red circle, only one pair of row and column permutation yields a zero error, out of all the possible permutations.

describing the sparsity patterns of the butterfly factors satisfies the chainability condition (Definition 7.5).

But when the permutations $\mathbf{P}, \mathbf{Q}$ are not known, Problem (8.1) is conjectured to be difficult. On the one hand, if we enumerate all possible permutations to solve Problem (8.1) with fixed permutations, we numerically find that only a small proportion of permutations yields a small approximation error, as illustrated in Figure 8.1. This illustrates the *necessity* of identifying the appropriate permutations in order to solve Problem (8.1). But on the other hand, searching exhaustively for all permutations is not tractable, even if we take into account certain permutation equivalences with respect to Problem (8.1), as discussed in Section 8.2 below.

Motivation for considering the problem with unknown permutations. Some matrices associated with commonly used fast transforms, such as the DFT matrix, does not admit exactly a butterfly factorization $\mathbf{X}_1 \dots \mathbf{X}_L$ under the fixed-support constraint $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$, for a given architecture β of length L, but it admits a factorization $\mathbf{Q}^\top \mathbf{X}_1 \dots \mathbf{X}_L \mathbf{P}$ with $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$ for some row and column permutations $\mathbf{P}, \mathbf{Q}$. We illustrate this in the example of the DFT matrix as follows.

Lemma 8.1. *Denote* $\mathbf{A}$ *the DFT matrix of size* $n \times n$ *where* $n := 2^L$*, and* $\beta := (\pi_\ell)_{\ell=1}^L := ((2^{\ell-1}, 2, 2, 2^{L-\ell}))_{\ell=1}^L$ *the square dyadic butterfly architecture of length* L*. Denote* $\mathbf{P}$ *the bit-reversal permutation matrix (cf. Section 3.2.2). Then:* $\mathbf{A} \notin \mathcal{B}^\beta$*, but* $\mathbf{AP} \in \mathcal{B}^\beta$*, where we recall that the notation* $\mathcal{B}^\beta$ *is defined in (7.2).*

Proof. The proof of $\mathbf{AP} \in \mathcal{B}^\beta$ is already given in Section 3.2.2. The proof of $\mathbf{A} \notin \mathcal{B}^\beta$ is deferred to Section 8.2.1 below. □

When the target matrix $\mathbf{A}$ admits a butterfly factorization associated with an architecture β up to some row and column permutations, addressing directly the problem $\min_{\mathbf{B} \in \mathcal{B}^\beta} \|\mathbf{A} - \mathbf{B}\|_F$ without taking into account these permutations can lead to an arbitrarily high approximation error, as illustrated in Figure 8.1, whereas the infimum value for such a matrix $\mathbf{A}$ is null in Problem (8.1). More generally, when $\mathbf{A}$ cannot be well approximated by $\mathbf{X}_1 \dots \mathbf{X}_L$ with $(\mathbf{X}_\ell)_{\ell=1}^L \in \Sigma^\beta$, considering row and column permutations can offer more flexibility to the factorization model, which can lead to a better approximation of the target matrix by a product of sparse factors. This explains why we want to study Problem (8.1) in this chapter.

Considered approach: partition the matrix into low-rank submatrices. In order to find the optimal permutations in Problem (8.1), we can rely on the fact that a matrix admitting a butterfly factorization satisfies the *complementary low-rank property* [234] (Definition 3.7), that is, there are index blocks (Definition 3.5) that define several partitions of the matrix indices for which the rank of the submatrices at each of these blocks is low (see Corollary 7.2 from Chapter 7). It is then sufficient to identify these partitions in order to address Problem (8.1). Identifying the index blocks for which the rank of the submatrices is low can be done *analytically* when the matrix entries are expressed by a smooth kernel operator $(\mathbf{x}, \boldsymbol{\omega}) \mapsto K(\mathbf{x}, \boldsymbol{\omega})$ evaluated on some parameters $\{\mathbf{x}_i\}_{i=1}^n$, $\{\boldsymbol{\omega}_j\}_{j=1}^n$. For instance, this is the case of matrices associated with certain integral operators [40] or special function transforms [289] (see Section 3.4.2 for more details). However, when the considered target matrix does not have an analytic form, or when it is not accessible, the literature does not, to the best of our knowledge, propose a method for identifying these partitions.

Contribution. We therefore propose a heuristic based on alternating spectral clustering of rows and columns, in order to identify partitions of the target matrix into low-rank submatrices *without relying on analytical assumptions*. In terms of applications, this heuristic enables us to *algorithmically* verify if a linear operator (for which a fast algorithm is not known) satisfies the complementary low-rank property, so that it allows an appropriate approximation of the associated matrix by a product of butterfly factors. This chapter introduces this heuristic for the specific case of *square dyadic butterfly factorization*. Generalization of the method to any chainable architectures (Definition 7.5) is left to future work.

Outline. Section 8.2 recalls the low-rank property of butterfly matrices used to identify optimal permutations in the proposed heuristic. Section 8.3 numerically shows the necessity of identifying these permutations. This motivates our heuristic explained in Section 8.4, and studied experimentally in Section 8.5. Related work and perspectives are discussed in Sections 8.6 and 8.7.

8.2 Problem formulation

In this chapter, using the framework of Chapter 7, β denotes the architecture (cf. Definition 7.3) associated with the *square dyadic butterfly factorization* of length $L \geq 2$, defined as:

$$\beta := (\boldsymbol{\pi}_\ell)_{\ell=1}^{L}, \quad \boldsymbol{\pi}_\ell := (2^{\ell-1}, 2, 2, 2^{L-\ell}) \quad \forall \ell \in [\![L]\!]. \tag{8.2}$$

We recall the following notations from the previous chapter: for any sparsity pattern $\boldsymbol{\pi} = (a, b, c, d)$, we denote $\mathbf{S}_{\boldsymbol{\pi}} := \mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d$, and $\Sigma^{\boldsymbol{\pi}}$ is defined in Definition 7.2; the set Σ^β and $\mathcal{B}^\beta$ are defined in (7.2) and (7.9); the operator $*$ for a chainable pair of patterns is defined in Definition 7.4, which is associative in the sense of Lemma 7.3.

In order to apply the results from Chapter 7, we claim the following result.

Lemma 8.2. *The square dyadic butterfly architecture β is a non-redundant chainable architecture (cf. Definitions 7.5 and 7.6), with $\mathbf{q}(\beta) = (1, \dots, 1)$.*

Proof. Denote $(a_\ell, b_\ell, c_\ell, d_\ell) := (2^{\ell-1}, 2, 2, 2^{L-\ell})$ for any $\ell \in [\![L]\!]$. Let $\ell \in [\![L-1]\!]$. By Definition 7.4, $(\boldsymbol{\pi}_\ell, \boldsymbol{\pi}_{\ell+1})$ is chainable because $a_\ell c_\ell / a_{\ell+1} = 1 = b_{\ell+1} d_{\ell+1} / d_\ell$ and $a_{\ell+1}/a_\ell = 2$, $d_\ell / d_{\ell+1} = 2$. Moreover, it is not redundant by Definition 7.6, because $q(\boldsymbol{\pi}_\ell, \boldsymbol{\pi}_{\ell+1}) = 1 < 2 = \min(b_\ell, c_{\ell+1})$. □

The problem that we address in this chapter is Problem (8.1) under the constraint $(\mathbf{X}_\ell)_{\ell=1}^{L} \in \Sigma^\beta$, with unknown permutations $\mathbf{P}, \mathbf{Q}$, and without any assumption on the target matrix $\mathbf{A}$. The rest of this section explains our approach, which relies on the low-rank characterization of square dyadic butterfly matrices based on the results of Chapter 7.

8.2.1 Low-rank property of square dyadic butterfly matrices

Butterfly matrices associated with a chainable architecture are characterized in Corollary 7.2, by bounding the rank of a polynomial number of their submatrices at specific index blocks. We apply these results to the case of the square dyadic butterfly architecture.

Lemma 8.3. *For the square dyadic architecture β, we have $\mathcal{B}^{\beta} = \bigcap_{\ell=1}^{L-1} \mathcal{B}^{\beta_\ell}$, where we recall the notation introduced in Definition 7.7:*

$$\forall \ell \in [\![L-1]\!], \quad \mathcal{B}^{\beta_\ell} := \{\mathbf{XY} \,|\, (\mathbf{X}, \mathbf{Y}) \in \Sigma^{\pi_1 * \ldots * \pi_\ell} \times \Sigma^{\pi_{\ell+1} * \ldots * \pi_L}\}.$$

Proof. This is an application of Corollary 7.2 since the architecture β is chainable by Lemma 8.2. □

Lemma 8.4. *With the same notations as in Lemma 8.3, for any $\ell \in [\![L-1]\!]$:*

$$\mathbf{A} \in \mathcal{B}^{\beta_\ell} \iff \operatorname{rank}(\mathbf{A}[R, C]) \leq 1 \quad \forall (R, C) \in P_{L-\ell}^{row} \times P_{\ell}^{col},$$

where, recalling the notation from Definition 7.1:

$$\begin{aligned} P_{L-\ell}^{row} &:= \{R_P \,|\, P \in \mathcal{P}(\mathbf{S}_{\pi_1 * \ldots * \pi_\ell}, \mathbf{S}_{\pi_{\ell+1} * \ldots * \pi_L})\} \\ &= \left\{ \{i + (k-1)2^{L-\ell}\}_{k \in [\![2^\ell]\!]} \,|\, i \in [\![2^{L-\ell}]\!] \right\}, \\ P_{\ell}^{col} &:= \{C_P \,|\, P \in \mathcal{P}(\mathbf{S}_{\pi_1 * \ldots * \pi_\ell}, \mathbf{S}_{\pi_{\ell+1} * \ldots * \pi_L})\} \\ &= \left\{ \{(i-1)2^{L-\ell} + k\}_{k \in [\![2^{L-\ell}]\!]} \,|\, i \in [\![2^\ell]\!] \right\}. \end{aligned} \tag{8.3}$$

Proof. This lemma is an application of Lemma 7.2, because $(\pi_1 * \cdots * \pi_\ell, \pi_{\ell+1} * \ldots * \pi_L)$ is chainable for any $\ell \in [\![L-1]\!]$, by chainability of β and by Lemma 7.5. It remains to prove the explicit formula for the partitions $P_{L-\ell}^{\text{row}}$ and P_{ℓ}^{col}. By Lemma 7.4 and by definition of β, we have for any $\ell \in [\![L-1]\!]$: $\pi_1 * \ldots * \pi_\ell = (1, 2^\ell, 2^\ell, 2^{L-\ell})$, and $\pi_{\ell+1} * \ldots * \pi_L = (2^\ell, 2^{L-\ell}, 2^{L-\ell}, 1)$. Then, to prove (8.3), one can use the definition $\mathbf{S}_{\pi} = \mathbf{I}_a \otimes \mathbf{1}_{b \times c} \otimes \mathbf{I}_d$ for any pattern $\pi = (a, b, c, d)$, and the fact that, for any $P \in \mathcal{P}(\mathbf{S}_{\pi_1 * \ldots * \pi_\ell}, \mathbf{S}_{\pi_{\ell+1} * \ldots * \pi_L})$, we have $R_P = \operatorname{supp}(\mathbf{S}_{\pi_1 * \ldots * \pi_\ell}[:, i])$, $C_P = \operatorname{supp}(\mathbf{S}_{\pi_{\ell+1} * \ldots * \pi_L}[i, :])$ for any $i \in P$, by Definition 7.1. □

By Proposition C.1, $P_{\ell+1}^{\text{row}}$ and $P_{\ell+1}^{\text{col}}$ defined in Lemma 8.4 are partitions of $[\![n]\!]$ that are finer (Definition 3.4) than P_{ℓ}^{row} and P_{ℓ}^{col}, respectively, for $\ell \in [\![L-2]\!]$. This defines two corresponding cluster trees (Definition 3.3), denoted $T_{\beta}^{\text{row}}, T_{\beta}^{\text{col}}$, of depth $L-1$ and with root node $[\![n]\!]$, for which the ℓ-th level corresponds to P_{ℓ}^{row} and P_{ℓ}^{col}, respectively, for $\ell \in [\![L-1]\!]$.

Remark 8.1. *One can show, using Lemma 8.4, that $T_{\beta}^{row}, T_{\beta}^{col}$ for the square dyadic butterfly architecture β satisfies the following conditions:*

- *$T_{\beta}^{row}, T_{\beta}^{col}$ are dyadic, i.e., each non-leaf node has exactly two children;*
- *the left and right child of each node $\{a_1, a_2, \ldots, a_{n/2^\ell}\}$ of T_{β}^{row} at level $\ell \in [\![0, L-1]\!]$ $(a_1 < \ldots < a_{n/2^\ell})$ are $\{a_1, a_3, \ldots, a_{n/2^\ell - 1}\}$ and $\{a_2, a_4, \ldots, a_{n/2^\ell}\}$;*
- *the left and right child of each node $\{b_1, b_2, \ldots, b_{n/2^\ell}\}$ of T_{β}^{col} at level $\ell \in [\![0, L-1]\!]$ $(b_1 < \ldots < b_{n/2^\ell})$ are $\{b_1, b_2, \ldots, b_{n/2^{\ell+1}}\}$ and $\{b_{n/2^{\ell+1}+1}, b_{n/2^{\ell+1}+2}, \ldots, b_{n/2^\ell}\}$.*

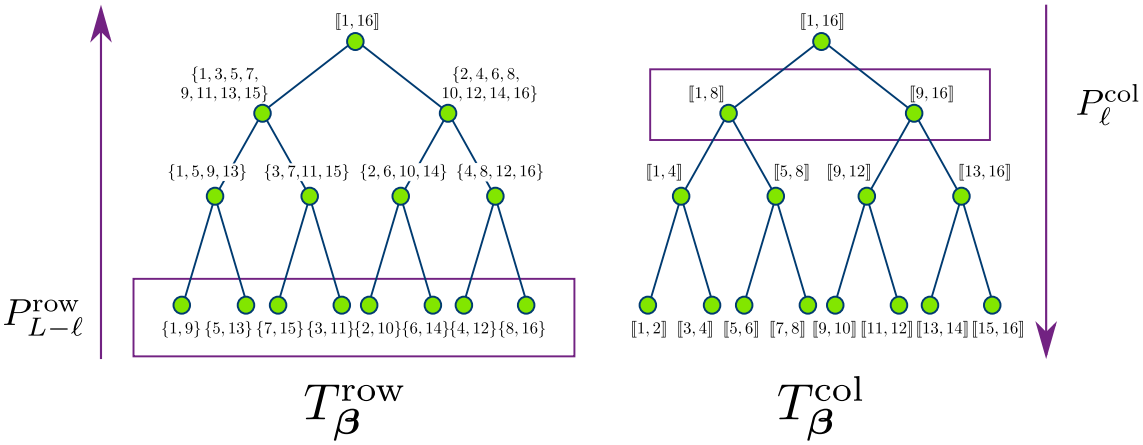


Figure 8.2: Illustration of the cluster trees T_β^{row}, T_β^{col} for which a square dyadic butterfly matrix $\mathbf{A} \in \mathcal{B}^\beta$ of size 16×16 satisfies the complementary low-rank property.

We illustrate these trees in Figure 8.2.

This two cluster trees yield the complementary low-rank characterization of a square dyadic butterfly matrix.

Proposition 8.1. *Let $n := 2^L$, and consider the square dyadic architecture β of length $L \geq 2$. Then, for any $\mathbf{A} \in \mathbb{C}^{n\times n}$, $\mathbf{A} \in \mathcal{B}^\beta$, if, and only if, $\mathbf{A}$ satisfies exactly the complementary low-rank property for $(T_\beta^{row}, T_\beta^{col})$ with rank-one submatrices (cf. Definition 3.7), where $T_\beta^{row}, T_\beta^{col}$ are the cluster trees defined in Proposition C.1, and described in Remark 8.1.*

Proof. This is a direct application of Corollary C.2 because β satisfies the conditions of Corollary C.2 by Lemma 8.2. □

Proof of Lemma 8.1. Before continuing, we can use these results to prove that $\mathbf{A} \notin \mathcal{B}^\beta$ when $\mathbf{A}$ is the DFT matrix of size $n \times n$, as claimed in Lemma 8.1.

Proof. By definition, the entries of $\mathbf{A}$ at index $k, l \in [\![n]\!]$ is $(e^{-i\frac{2\pi}{n}})^{(k-1)(l-1)}$, so

$$\begin{pmatrix} \mathbf{A}[1,1] & \mathbf{A}[1,2] \\ \mathbf{A}[1+n/2,1] & \mathbf{A}[1+n/2,2] \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

The rank of this submatrix is 2. But $R := \{1, 1+n/2\} \in P_{L-1}^{\mathrm{row}}$ as defined in Lemma 8.4, and $\{1,2\} \subseteq C := [\![1, n/2]\!] \in P_1^{\mathrm{col}}$. This means that the rank of $\mathbf{A}[R, C]$ is at least 2. By Lemma 8.4, $\mathbf{A} \notin \mathcal{B}^{\beta_1}$, and by Lemma 8.3, $\mathbf{A} \notin \mathcal{B}^\beta$. □

8.2.2 Approach to recover unknown permutations

In the following, when the permutation matrices $\mathbf{P}, \mathbf{Q}$ are fixed, we choose to apply Algorithm 6.1 introduced in Chapter 6, with input matrix $\mathbf{QAP}^\top$, in order to

find an approximate solution to Problem (8.1). Denoting $(\hat{\mathbf{X}}_1, \dots, \hat{\mathbf{X}}_L)$ the outputs of Algorithm 6.1 with input matrix $\mathbf{QAP}^\top$, the corresponding approximation error is

$$\hat{E}^\beta(\mathbf{QAP}^\top) := \|\mathbf{QAP} - \hat{\mathbf{X}}_1 \dots \hat{\mathbf{X}}_L\|_F = \|\mathbf{A} - \mathbf{Q}^\top \hat{\mathbf{X}}_1 \dots \hat{\mathbf{X}}_L \mathbf{P}\|_F,$$

because permutation matrices are orthonormal and they preserve the Frobenius norm. However, the difficult case is when the permutations are not known.

We now explain our approach based on the complementary low-rank characterization of a square dyadic butterfly matrix to address this difficult case. For that, let us introduce the following notations.

- For any cluster tree T whose root is $[\![n]\!]$, and for any permutation $\sigma : [\![n]\!] \to [\![n]\!]$, we define the cluster tree $\sigma(T)$ obtained by permuting the indices of each node in T according to σ.
- For any permutation matrix $\mathbf{P}$ of size $n \times n$, we denote $\sigma_{\mathbf{P}}$ its corresponding permutation in $[\![n]\!]$.
- For any cluster tree T^{row} of row indices in $[\![n]\!]$, there are several permutation matrices $\mathbf{Q}$ for which $\sigma_{\mathbf{Q}}(T^{\text{row}}_\beta) = T^{\text{row}}$. This defines equivalence classes of row permutations $[\mathbf{Q}_{T^{\text{row}}}]$.
- Similarly, for any given cluster tree T^{col}, two column permutations $\mathbf{P}_1$, $\mathbf{P}_2$ are in the same equivalence class $[\mathbf{P}_{T^{\text{col}}}]$ if, and only if, $\sigma_{\mathbf{P}_1}(T^{\text{col}}_\beta) = \sigma_{\mathbf{P}_2}(T^{\text{col}}_\beta) = T^{\text{col}}$.

Suppose that the target matrix in Problem (8.1) is of the form $\mathbf{A} := \mathbf{Q}^\top \tilde{\mathbf{A}} \mathbf{P}$, where $\tilde{\mathbf{A}} \in \mathcal{B}^\beta$ and $\mathbf{P}$, $\mathbf{Q}$ are two unknown arbitrary permutation matrices. By Proposition 8.1, $\tilde{\mathbf{A}}$ satisfies the complementary low-rank property for the trees $(T^{\text{row}}_\beta, T^{\text{col}}_\beta)$. Hence, the target matrix $\mathbf{A} = \mathbf{Q}^\top \tilde{\mathbf{A}} \mathbf{P}$ also satisfies this property, but for the trees $T^{\text{row}} := \sigma_{\mathbf{Q}}(T^{\text{row}}_\beta)$ and $T^{\text{col}} := \sigma_{\mathbf{P}}(T^{\text{col}}_\beta)$. If we can reconstruct $(T^{\text{row}}, T^{\text{col}})$ from the observation of $\mathbf{A}$, then Problem (8.1) can be solved by choosing any pair $(\hat{\mathbf{P}}, \hat{\mathbf{Q}}) \in [\mathbf{P}_{T^{\text{col}}}] \times [\mathbf{Q}_{T^{\text{row}}}]$ and applying Algorithm 6.1 to the matrix $\hat{\mathbf{Q}}\mathbf{A}\hat{\mathbf{P}}^\top$. Indeed, such a choice is sufficient to guarantee that $\hat{\mathbf{Q}}\mathbf{A}\hat{\mathbf{P}}^\top$ satisfies the complementary low-rank property for T^{row}_β and T^{col}_β, hence, $\hat{\mathbf{Q}}\mathbf{A}\hat{\mathbf{P}}^\top \in \mathcal{B}^\beta$ by Proposition 8.1. This is because the matrices $\hat{\mathbf{P}}^\top$ and $\hat{\mathbf{Q}}^\top$ are associated with the inverse permutations $\sigma^{-1}_{\mathbf{P}}$, $\sigma^{-1}_{\mathbf{Q}}$, and by definition of the introduced equivalence classes, we have $\sigma^{-1}_{\mathbf{P}}(T^{\text{col}}) = T^{\text{col}}_\beta$ and $\sigma^{-1}_{\mathbf{Q}}(T^{\text{row}}) = T^{\text{row}}_\beta$.

In conclusion, in order to solve Problem (8.1), it is *sufficient* to identify the trees T^{row} and T^{col} for which the target matrix satisfies the complementary low-rank property. By Lemma 8.3, Lemma 8.4 and Proposition 8.1, this reduces to identifying partitions of $\mathbf{A}$ into rank-one submatrices.

8.3 Necessity of recovering the partitions

We now show empirically that identifying the cluster trees $(T^{\text{row}}, T^{\text{col}})$ for which $\mathbf{A} := \mathbf{Q}^\top \tilde{\mathbf{A}} \mathbf{P}$ ($\tilde{\mathbf{A}} \in \mathcal{B}^\beta$) satisfies the complementary low-rank property is in fact *necessary* to solve Problem (8.1).

Protocol. Consider a matrix $\tilde{\mathbf{A}} \in \mathcal{B}^\beta$ that admits a square dyadic butterfly factorization, with butterfly factors having nonzero entries drawn i.i.d. from the standard normal distribution. Then, for each possible pair $(\tilde{T}^{\text{row}}, \tilde{T}^{\text{col}})$ of dyadic cluster trees of depth $L-1$ that we enumerate, we fix an arbitrary pair $(\tilde{\mathbf{P}}, \tilde{\mathbf{Q}}) \in [\mathbf{P}_{\tilde{T}^{\text{col}}}] \times [\mathbf{Q}_{\tilde{T}^{\text{row}}}]$, and compute an approximate solution to Problem (8.1) via Algorithm 6.1 applied to $\tilde{\mathbf{Q}}\mathbf{A}\tilde{\mathbf{P}}^\top$, yielding an approximation error $\hat{E}^\beta(\tilde{\mathbf{Q}}\mathbf{A}\tilde{\mathbf{P}}^\top)$ for Problem (8.1). The goal is to check whether the only trees yielding a small approximation error for the target matrix $\mathbf{A} := \mathbf{Q}^\top \tilde{\mathbf{A}} \mathbf{P}$ are $T^{\text{row}} := \sigma_{\mathbf{Q}}(T_\beta^{\text{row}})$ and $T^{\text{col}} := \sigma_{\mathbf{P}}(T_\beta^{\text{col}})$. For illustrative purposes, we consider in our experiment the case $n = 8$, which gives 315 possible cluster trees for the enumeration of $\tilde{T}^{\text{row}}$ and $\tilde{T}^{\text{col}}$.

Remark 8.2. *A count of the number of trees shows that this experiment is not tractable for a large size n, which is why we consider $n = 8$. The number u_n of possible* dyadic *cluster trees of depth $L-1$ with $L := \log_2(n)$ for a same root of cardinal n satisfies the recurrence relation $u_n = \frac{1}{2}\binom{n}{n/2}(u_{\frac{n}{2}})^2$ with $u_2 = 1$, since there are $\frac{1}{2}\binom{n}{n/2}$ possible ways to choose values for the two children of the root node, and the left and right subtrees of the root node are, by definition, dyadic cluster trees of depth $L-2$ whose root is of cardinal $n/2$.*

Results. In Figure 8.1, the exhaustive search on all pairs $(\tilde{T}^{\text{row}}, \tilde{T}^{\text{col}})$ of dyadic trees shows that the error $\hat{E}^\beta(\tilde{\mathbf{Q}}\mathbf{A}\tilde{\mathbf{P}}^\top)$ is null *only for one* pair of trees, and that the other pairs fail to solve Problem (8.1), in the sense that there corresponding approximation error is positive. In other words, this illustrates empirically that, in order to solve Problem (8.1), it is *necessary* to identify the appropriate trees T^{row} and T^{col} for which the target matrix satisfies the complementary low-rank property. The formal proof of such a necessity for arbitrary size of matrices is left to future work.

8.4 Alternating spectral clustering

This section introduces Algorithm 8.2 based on alternating spectral clustering to identify the trees $T^{\text{row}} := \sigma_{\mathbf{Q}}(T_\beta^{\text{row}})$ and $T^{\text{col}} := \sigma_{\mathbf{P}}(T_\beta^{\text{col}})$ for which the matrix $\mathbf{A} := \mathbf{Q}^\top \tilde{\mathbf{A}} \mathbf{P}$ ($\tilde{\mathbf{A}} \in \mathcal{B}^\beta$) satisfies the complementary low-rank property. To de-

scribe our approach for Problem (8.1), we start by studying the problem

$$\min_{\mathbf{M}\in\mathcal{B}^{\beta_\ell},\mathbf{P},\mathbf{Q}} \|\mathbf{A}-\mathbf{Q}^\top\mathbf{M}\mathbf{P}\|_F^2, \tag{8.4}$$

for each $\ell \in [\![L-1]\!]$, since $\mathcal{B}^\beta = \bigcap_{\ell=1}^{L-1}\mathcal{B}^{\beta_\ell}$ by Lemma 8.3. By Lemma 8.4, this problem is equivalent to

$$\min_{\{R_i\}_{i=1}^{n/2^\ell},\{C_j\}_{j=1}^{2^\ell}} \sum_{i=1}^{n/2^\ell}\sum_{j=1}^{2^\ell} \min_{\mathbf{x},\mathbf{y}} \|\mathbf{A}[R_i,C_j]-\mathbf{x}\mathbf{y}^*\|_F^2, \tag{8.5}$$

where $\{R_i\}_{i=1}^{n/2^\ell}$, $\{C_j\}_{j=1}^{2^\ell}$ are row and column partitions into subsets of equal cardinal, and $\min_{\mathbf{x},\mathbf{y}}\|\mathbf{A}[R,C]-\mathbf{x}\mathbf{y}^*\|_F^2$ is the best rank-one approximation error of $\mathbf{A}[R,C]$. The symbol $*$ denotes the conjugate transpose of a matrix. Such a problem can be seen as a kind of *biclustering* problem, in the sense that we want to cluster rows and columns of the matrix $\mathbf{A}$ *simultaneously*. Related work to this problem is further discussed in Section 8.6 below.

Row clustering. For an alternating optimization to work, it is necessary to be able to address the subproblem (8.5) when one of the two partitions is known. Therefore, without loss of generality, let us minimize find a row partition $\{R_i\}_{i=1}^{n/2^\ell}$ that minimizes (8.5) when fixing a given column partition $\{C_j\}_{j=1}^{2^\ell}$.

To achieve this, we draw inspiration from methods based on *spectral clustering* for *subspace clustering* [348]. Define the graph $\mathcal{G}_j$ for $j \in [\![2^\ell]\!]$ as follows:

- the n nodes of the graph $\mathcal{G}_j$ are the n rows of $\mathbf{A}$ restricted to the columns C_j, denoted $\{\mathbf{A}[k,C_j]\}_{k=1}^n$;
- the weights of the edges of $\mathcal{G}_j$ are given by the similarity matrix $\mathbf{W}_j \in \mathbb{R}^{n\times n}$ defined by

$$\mathbf{W}_j[k,k'] := \left|\left(\frac{\mathbf{A}[k,C_j]}{\|\mathbf{A}[k,C_j]\|_2}\right)^*\left(\frac{\mathbf{A}[k',C_j]}{\|\mathbf{A}[k',C_j]\|_2}\right)\right|^\alpha \in [0,1] \quad \forall k,k' \in [\![n]\!], \tag{8.6}$$

  with a parameter $\alpha > 0$ controlling the contrast between weights. In other words, each weight is the cosine similarity raised to the power α.

Intuitively, a group of rows restricted to columns C_j is interconnected by high-value weights in the graph $\mathcal{G}_j$ when the corresponding rows are correlated, which is the case when they form a rank-one submatrix. Conversely, two uncorrelated rows should be connected by an edge with a low-value weight. Thus, by solving a minimal cut problem on the graph $\mathcal{G}$ with similarity matrix $\mathbf{W} := \sum_{j=1}^{2^\ell}\mathbf{W}_j$, the resulting clustering of nodes should yield a row clustering with small error in the subproblem (8.5).

Algorithm 8.1 Alternating spectral clustering for the subproblem (8.5) for a given $\ell \in [\![L-1]\!]$, during N_{iter} iterations.

Require: $\mathbf{A}, \ell, \alpha > 0$, random `seed`

$\{C_j\}_{j=1}^{2^\ell} \leftarrow$ random partition of $[\![n]\!]$ according to `seed`
for $k = 1, \ldots, N_{\text{iter}}$ **do**
  $\{R_i\}_{i=1}^{n/2^\ell} \leftarrow$ row clustering by fixing $\{C_j\}_{j=1}^{\ell}$ with parameter α
  $\{C_j\}_{j=1}^{2^\ell} \leftarrow$ column clustering by fixing $\{R_i\}_{i=1}^{n/2^\ell}$ with paramater α
end for
$E_\ell \leftarrow \sum_{i,j} \min_{\mathbf{x},\mathbf{y}} \|\mathbf{A}[R_i, C_j] - \mathbf{x}\mathbf{y}^*\|_F^2$
return $E_\ell, \{R_i\}_{i=1}^{n/2^\ell}, \{C_j\}_{j=1}^{2^\ell}$

Concretely, a spectral clustering [350] of the graph $\mathcal{G}$ with a similarity matrix $\mathbf{W}$ is performed by computing the eigenvector decomposition of the unnormalized graph Laplacian matrix $\mathbf{L} := \mathbf{D} - \mathbf{W}$, where $\mathbf{D}$ is the degree matrix whose diagonal entries are $\mathbf{W}(1 \ldots 1)^\top$. To ensure that row clusters are of the same cardinal, the k-means clustering step on the spectral embeddings is implemented according to the method from [35].

Alternating optimization. When the column partition is no longer fixed, Algorithm 8.1 proposes to address the subproblem (8.5) via *alternating clustering*. A column partition is randomly initialized, and at each iteration, we perform a spectral clustering of the rows by fixing the column partition of the previous iteration. Then, vice versa, we swap the role of the rows and the columns. Algorithm 8.1 does not admit guarantees of success, and may require several random initializations of the initial column partition in order to provide a solution with small error.

Final heuristic for Problem (8.1). Going back to Problem (8.1) for a target matrix $\mathbf{A} := \mathbf{Q}^\top \tilde{\mathbf{A}} \mathbf{P}$ ($\tilde{\mathbf{A}} \in \mathcal{B}^\beta$), the proposed heuristic described in Algorithm 8.2 applies Algorithm 8.1 to return a solution to each subproblem (8.5) for $\ell \in [\![L-1]\!]$. If each subproblem (8.5) is solved correctly (yielding a null error), and if the identified partitions form valid cluster trees $(T^{\text{row}}, T^{\text{col}})$, in the sense that they satisfy the axioms of a cluster tree (cf. Definition 3.3), then, by construction, $\mathbf{A}$ satisfies the complementary low-rank property for $(T^{\text{row}}, T^{\text{col}})$. We then solve Problem (8.1) via Algorithm 6.1 with input $\hat{\mathbf{Q}}\mathbf{A}\hat{\mathbf{P}}^\top$, by fixing $(\hat{\mathbf{P}}, \hat{\mathbf{Q}}) \in [\mathbf{P}_{T^{\text{col}}}] \times [\mathbf{Q}_{T^{\text{row}}}]$.

Complexity. The complexity of spectral clustering is $\mathcal{O}(n^3)$ in general when there are n elements to cluster. Therefore, the complexity of Algorithms 8.1 and 8.2 is at least $\mathcal{O}(n^3)$.

Algorithm 8.2 Heuristic for Problem (8.1) via the identification of T^{row}, T^{col}.

Require: $\mathbf{A}, \{\alpha_k\}_{k=1}^K, \{\texttt{seed}_m\}_{m=1}^M, \mathcal{T}$
for $\ell = 1, \dots, L-1$ **do**
  for $k = 1, \dots, K$, $m = 1, \dots, M$ **do**
    $E^{k,m}, P^{\text{row}}_{k,m}, P^{\text{col}}_{k,m} \leftarrow$ Algorithm 8.1 $(\mathbf{A}, \ell, \alpha_k, \texttt{seed}_m)$
  end for
  $(\tilde{k}, \tilde{m}) \leftarrow \arg\min\{E^{k,m}\}_{k,m}$
  $\{R_i^{(\ell)}\}_i, \{C_j^{(\ell)}\}_j \leftarrow P^{\text{row}}_{\tilde{k},\tilde{m}}, P^{\text{col}}_{\tilde{k},\tilde{m}}$
end for
Verify that $\{R_i^{(\ell)}\}_i$ for $\ell \in [\![L-1]\!]$ form a valid cluster tree T^{row}
Verify that $\{C_j^{(\ell)}\}_j$ for $\ell \in [\![L-1]\!]$ form a valid cluster tree T^{col}
if one of the two trees is not valid **then**
  return `failure`
else
  $(\hat{\mathbf{P}}, \hat{\mathbf{Q}}) \leftarrow$ any pair of permutations in $[\mathbf{P}_{T^{\text{col}}}] \times [\mathbf{Q}_{T^{\text{row}}}]$
  $(\hat{\mathbf{X}}_\ell)_{\ell=1}^L \leftarrow$ Algorithm 6.1 $\left(\hat{\mathbf{Q}}\mathbf{A}\hat{\mathbf{P}}^\top, \mathcal{T}\right)$
  return `success`, $\hat{\mathbf{P}}, \hat{\mathbf{Q}}, (\hat{\mathbf{X}}_\ell)_{\ell=1}^L$
end if

8.5 Experiments

We evaluate the empirical performance of the proposed heuristic (Algorithm 8.2) for approximating the target matrix

$$\mathbf{A} := \mathbf{Q}^\top \tilde{\mathbf{A}} \mathbf{P} + \epsilon(\|\tilde{\mathbf{A}}\|_F / \|\mathbf{N}\|_F),$$

where $\mathbf{P}$, $\mathbf{Q}$ are random permutation matrices, $\mathbf{N}$ is a random matrix with i.i.d. entries following a standard normal distribution, $\epsilon \geq 0$ controls the relative noise level, and $\tilde{\mathbf{A}}$ corresponds to either a random orthogonal butterfly matrix defined by [297], or the DFT matrix. We apply Algorithm 8.2 with $M = 5$ random seeds and $\{\alpha_k\}_{k=1}^K := \{10^p\}_{p\in\{-2,-1,0,1,2\}}$, on 20 instances of the problem for each $\epsilon \in \{0, 0.01, 0.03, 0.1\}$ and $n \in \{2^L\}_{L\in\{2,\dots,7\}}$. The success rate is computed as the proportion of executions of Algorithm 8.2 returning `success`, out of the 20 instances. We do not consider $n > 128$, because Algorithm 8.2 has a cubic complexity in n. Typically, an execution takes a few minutes for $n = 64$, and up to half-an-hour for $n = 128$.

Results for random orthogonal butterfly matrix. When $\tilde{\mathbf{A}}$ is a random orthogonal butterfly matrix, for each considered noise levels ϵ and matrix size n, Algorithm 8.2 has a success rate of 100 % out of 20 instances of the problem. This means that, when repeated with a sufficient amount of α and random seeds, Algorithm 8.1 can solve *independently* each subproblem (8.5) for $\ell \in [\![L-1]\!]$, and

Table 8.1: Success rate of Algorithm 8.2 on 20 problem instances with unknown permutation.

Noise level ϵ	0	0.01	0.03	0.1
Random orthogonal butterfly ($n = 128$)	100 %	100 %	100 %	100 %
Discrete Fourier transform ($n = 128$)	100 %	95 %	90 %	50 %

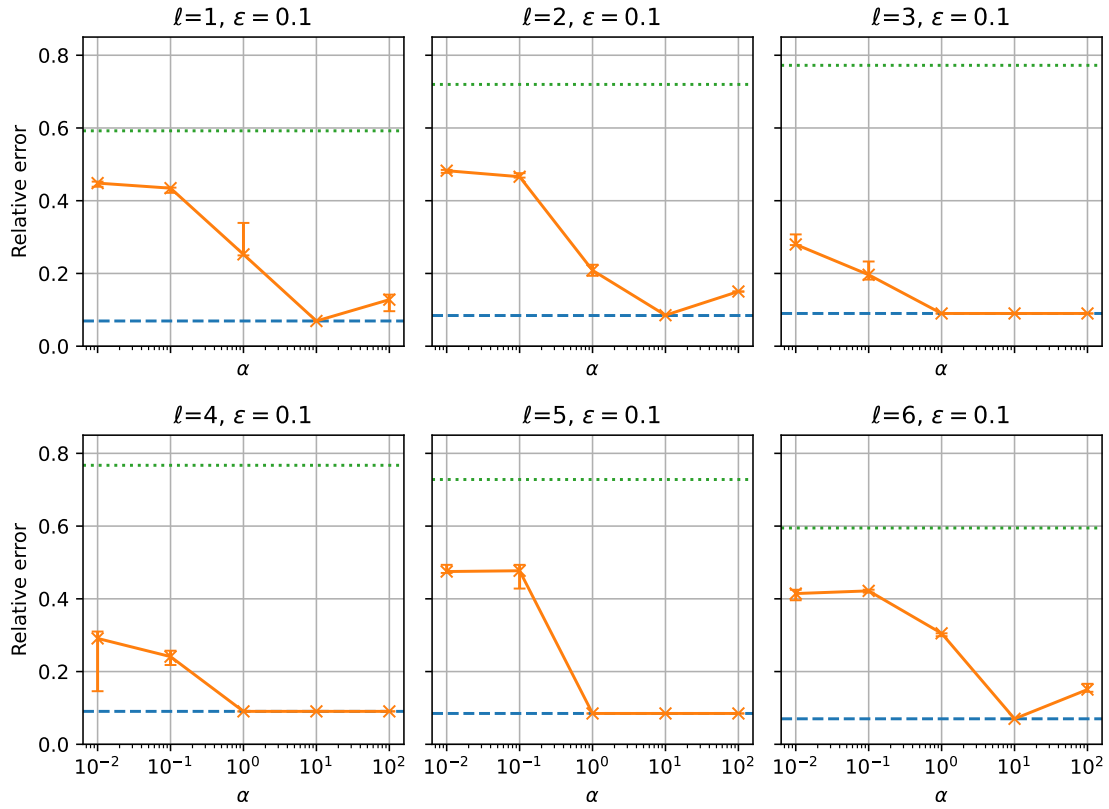


Figure 8.3: Performance of Algorithm 8.1 for each subproblem (8.5) with $n = 50$ iterations and $m = 5$ random seeds for the initialization. The target matrix is a random orthogonal butterfly matrix of size $n = 128$ (orange, full line), with noise level $\epsilon = 0.1$. Error bars show extrema, crosses indicate median values. Blue, dashed line: approximation error when knowing the cluster trees $(T^{\text{row}}, T^{\text{col}})$. Green, dotted line: minimal approximation error out of 1000 random sampling of partitions, where we solve the subproblem (8.5) while fixing the partitions.

the returned partitions on all subproblems form valid cluster trees T^{row} and T^{col}. Figure 8.3 illustrates the results of Algorithm 8.1 repeated with several values of α and several random seeds: for each subproblem (8.5) for a given ℓ, there exists at least one value of α for which alternating spectral clustering yields the same error as the one that we would obtain if the partitioning was known in advance.

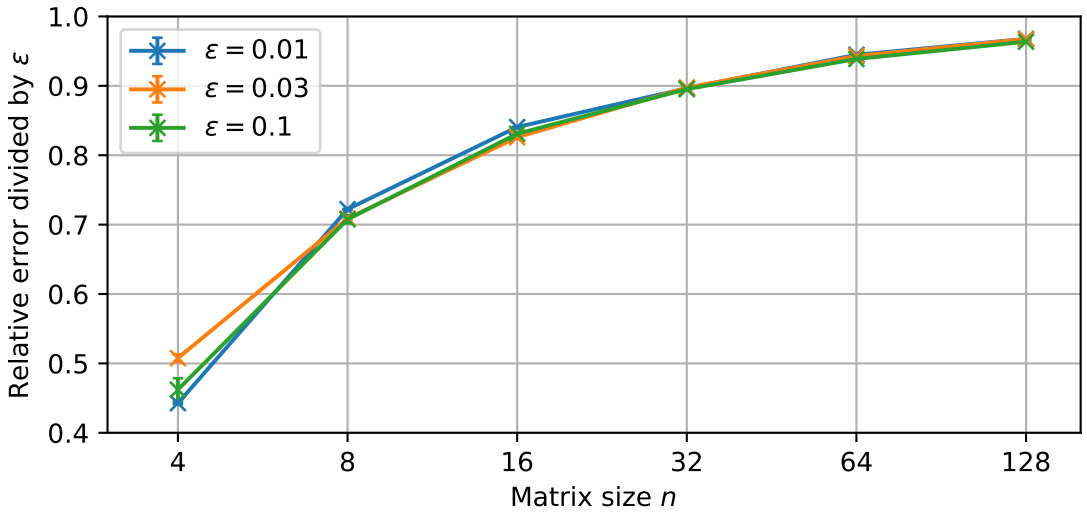


Figure 8.4: Relative approximation error of Algorithm 8.2 divided by ϵ, for the approximation of a noisy permuted random orthogonal butterfly matrix with noise level ϵ (crosses show average values, error bars show standard deviations).

When successful, Algorithm 8.2 returns the same approximation error as the error $\hat{E}^{\beta}(\mathbf{QAP}^{\top})$ that we would obtain if the permutations $\mathbf{P}, \mathbf{Q}$ were fixed and known. In the noiseless case, the relative error reaches machine precision. In the noisy case, Figure 8.4 shows that the relative error is smaller than the relative noise level ϵ.

Results for the DFT matrix. When $\tilde{\mathbf{A}}$ is the DFT matrix, Algorithm 8.2 also has a success rate of 100 %, for any noise levels $\epsilon \in \{0, 0.01, 0.03, 0.1\}$ and $n \leq 64$. For $n = 128$, the success rate is 100 % for the noiseless case, but deteriorates in the noisy case, as illustrated in Table 8.1. Future work could improve the robustness of the method to noise.

8.6 Related work

The subproblem (8.5) is an instance of the more general problem where, given a matrix $\mathbf{A}$, we want to find a row partition P^{row} and a column partition P^{col} that minimize the sum over $(R, C) \in P^{\text{row}} \times P^{\text{col}}$ of the best low-rank approximation of $\mathbf{A}[R, C]$. More precisely, given a number of row clusters $p \in [\![m]\!]$, a number of column clusters $q \in [\![m]\!]$, some rank parameters $\mathbf{R} \in \mathbb{N}^{p \times q}$ and a matrix $\mathbf{A}$ of size $m \times n$, the problem is:

$$\min_{\{R_i\}_{i=1}^{p}, \{C_j\}_{j=1}^{q}} \sum_{i=1}^{p} \sum_{j=1}^{q} \min_{\mathbf{B}, \text{rank}(\mathbf{B}) \leq \mathbf{R}[i,j]} \|\mathbf{A}[R_i, C_j] - \mathbf{B}\|_F, \tag{8.7}$$

where $P^{\text{row}} := \{R_i\}_{i=1}^{p}$ and $P^{\text{col}} := \{C_j\}_{j=1}^{q}$ are partitions of $[\![m]\!]$ and $[\![n]\!]$, respectively[1]. To the best of our knowledge, we did not find studies on this general problem formulation. However, we can relate this problem to three important problems that are documented in the literature.

Subspace clustering. The problem of subspace clustering [347] is the instance of the problem (8.7) where the number of row clusters is set as $p = 1$, i.e., $P^{\text{row}} := \{[\![m]\!]\}$. In other words, this is the problem of fitting a union of q linear subspaces of dimension $r_1, \ldots, r_q$ for some rank parameters $(r_j)_{j=1}^{q}$ to a set of n points in an m-dimensional space. It is a generalization of the classical principal component analysis where the points are assumed to be drawn from a single subspace (which is the case $q = 1$). The problem is in general challenging, because it requires simultaneously looking for optimal assignments of the points to different clusters, and for optimal linear subspaces that fit the points in each cluster. A review of the different methods for this problem can be found in [347,348]. Among them, some methods use spectral clustering algorithms [350] on an affinity matrix of size $n \times n$ that measures the pairwise similarity between the n points, in such a way that, ideally, the similarity is high between points in the same group and low between points of different groups. Typically, this can be done by fitting a local subspace to each point in a certain way, and measuring the similarity between points by comparing their associated subspaces, e.g., using their angle [95,96,129,346,366,381]. The proposed affinity matrix in (8.6) follows precisely this principle.

Multi-view subspace clustering. When the row partition P^{row} is still fixed in the problem (8.7), but has $p \geq 2$ clusters, the objective function becomes a sum of p terms where each of them is a single-view subspace clustering problem, with the constraint that the column partition $P^{\text{col}} := \{C_j\}_{j=1}^{q}$ is shared across these p problems. One can therefore adapt methods from single-view subspace clustering to the multi-view case, by making the clustering of the columns consistent across the different views, using various kinds of regularization like in [117,205]. The proposed heuristic in Section 8.4 is a naive way to perform multi-view spectral clustering, where we simply aggregate the affinity matrices of each view, and perform spectral clustering on the sum of these affinity matrices. Future work can integrate more sophisticated methods for multi-view subspace clustering in the proposed alternating optimization heuristic.

Co-clustering. The co-clustering problem [158] can be formulated as follows:

$$\min_{\mathbf{C} \in \mathbb{C}^{p,q}, \mathbf{U} \in \{0,1\}^{m \times p}, \mathbf{V} \in \{0,1\}^{n \times q}} \|\mathbf{A} - \mathbf{U}\mathbf{C}\mathbf{V}^{\top}\|_F,$$

[1] A problem variation is the one where we further add the constraint that the row and column clusters are balanced, i.e., they are of the same cardinal.

where the rows of $\mathbf{U}$ and $\mathbf{V}$ have exactly one entry equal to one. It can be seen as the instance of the problem (8.7) where all the rank parameters $\{\mathbf{R}[i,j]\}_{i,j}$ are equal to one, and the matrix $\mathbf{B}$ in the term $\min_{\mathbf{B},\text{rank}(\mathbf{B})\leq 1} \|\mathbf{A}[R_i, C_j] - \mathbf{B}\|_F$ is constrained to be of the form $\lambda \mathbf{1}_{|R_i|\times|C_j|}$ for some scalar λ. This problem generalizes the k-means problem in the sense that it considers simultaneous clustering of the rows and the columns. The alternating optimization proposed in Section 8.4 takes inspiration from the alternating least square strategy proposed in [262] for the co-clustering problem: it sequentially performs k-means to cluster rows after fixing the column partition, and vice versa. Other recent methods based on optimal transport [208,310] has been applied to the co-clustering problem. Extension of these methods to the problem (8.7) is an interesting future research direction.

8.7 Conclusion

We have proposed a heuristic designed to identify the partitions of a matrix into low-rank submatrices. This allows a square dyadic butterfly factorization without requiring analytical assumptions about the target matrix. We now discuss some perspectives.

Theoretical guarantees, hardness of the problem. On the one hand, experiments in Section 8.5 show that the proposed heuristic works well when the target matrix is assumed to admit a square dyadic butterfly factorization up to permutations in the noiseless setting. We therefore ask whether the proposed heuristic admits some kind of theoretical guarantees in the noiseless setting. Establishing such guarantees can be based, for instance, on the existing guarantees for the subspace clustering problem [96]. On the other hand, in complement to the previous question, one interesting future direction is to clarify whether the problems (8.7), (8.5) or (8.1) are NP-hard, based on existing hardness results for the k-means problem [7,10,304] or the subspace clustering problem [128].

Robustness, scalability issues. The current heuristic lacks some robustness to noise, as shown in Section 8.5. Indeed, Algorithm 8.2 proposes to solve the different subproblems (8.5) *independently* for $\ell \in [\![L-1]\!]$. However, if the resolution of one of the subproblems fails with Algorithm 8.1, then the heuristic might fail as well to construct valid cluster trees. Therefore, the current method can be further improved by addressing *conjointly* the different subproblems (8.5) for $\ell \in [\![L-1]\!]$, by enforcing the clusterings to be consistent across the different subproblems, in such a way that they yield valid cluster trees. Moreover, the current heuristic does not scale well due to its cubic complexity with respect to the matrix size. The scalability can be improved for instance by considering existing methods for large scale spectral clustering. Overcoming the robustness and scalability issues of the proposed heuristic would eventually enable the search for butterfly factorization of linear operators used in signal processing or machine learning, such

as Fourier transforms on graphs [322] or neural network layers [72], in order to accelerate their evaluation.

Chapter 9

Efficiency of butterfly sparse matrix multiplication on GPU: where do we stand?

Accelerating the inference and training of deep neural networks is a major challenge, given their constantly growing resource requirements. At the very heart of this challenge is the acceleration of matrix multiplication on GPUs, which is one of the main operations in the forward and the backward pass of a deep neural network. One key approach that aims to address this challenge consists in enforcing *sparsity* constraints on certain weight matrices in the model, in order to achieve various trade-offs between performance and model complexity. This chapter studies the efficiency *in practice* of GPU implementations for multiplying by a sparse matrix that admits a butterfly structure, as described in the previous chapters. One of our contributions is to benchmark existing GPU implementations for butterfly sparse matrix multiplication, and improve them in certain settings with a novel CUDA kernel.

9.1 Introduction

As detailed in Section 4.5, many works pointed out that replacing the weight matrices $\mathbf{W}$ of neural networks by butterfly matrices $\mathbf{W} = \mathbf{B}_1 \dots \mathbf{B}_L$ with butterfly factors $(\mathbf{B}_\ell)_{\ell=1}^L$, and training the nonzero weights of each $\mathbf{B}_\ell$ by gradient descent, leads to networks with much fewer parameters than their dense counterpart, while having comparable accuracies on some learning tasks [71,72,241,342].

However, to the best of our knowledge, there are almost no reports of numerical results concerning the *time efficiency* of these butterfly networks. The only results we could find are from [72], where some butterfly networks are reported to be twice faster to train than their dense counterparts for image classification and

The material of this chapter is based on an on-going work in collaboration with Antoine Gonon, Pascal Carrivain and Quoc-Tung Le.

language modeling, and [113] that reports an acceleration of $\mathbf{X} \mapsto \mathbf{W}^{-1}(\mathbf{K} \odot \mathbf{W}\mathbf{X})$ where $\mathbf{K}$ is some dense weight matrix, $\odot$ is the element-wise multiplication, and $\mathbf{W}$ is the DFT matrix (which admits a butterfly factorization, cf. Section 3.2.2), for dimensions of $\mathbf{W}$ larger than 4096. Our replication of the frameworks of the literature (Appendix D.2) revealed that, contrary to expectations, current implementations consistently make the forward pass of butterfly networks *slower* than dense counterparts in *half-precision*. This challenges previous assumptions about their practical utility.

The most fundamental operation when multiplying by a butterfly matrix $\mathbf{W} = \mathbf{B}_1 \dots \mathbf{B}_L$ is the multiplication by a single factor $\mathbf{B}_\ell$. Under the framework of Chapter 7, any butterfly factor $\mathbf{B}$ is associated with a sparsity pattern described by a tuple $\boldsymbol{\pi} := (a, b, c, d) \in (\mathbb{N}^*)^4$, and the case $a = 1$ or $d = 1$ are the most fundamental ones since having an efficient implementation for these cases would translate into an efficient implementation for the general case, as detailed below.

Contributions. The main contributions of this work are:

1. *To benchmark the efficiency of current GPU algorithms in PyTorch for the multiplication by a single butterfly factor*, for $a = 1$ or $d = 1$.
2. To release a *new open-source CUDA kernel that improves previous implementations in float-precision* (typically $\times 1.2$ faster when improving).
3. To find out that breaking the PyTorch convention by placing the batch size in *last dimension of the input PyTorch tensor rather than in first*[1] is a more favorable memory layout for sparse algorithms as it *substantially accelerates* both the generic sparse algorithm of PyTorch and our new kernel, while it has no impact on the dense algorithm.

Outline. Section 9.2 presents some preliminaries to study implementations of butterfly matrix multiplication. Section 9.3 describes existing GPU implementations in PyTorch. Section 9.4 describes the new CUDA kernel. Section 9.5 benchmarks existing GPU implementations on PyTorch, and our new kernel, for the multiplication with a single butterfly factor. Section 9.6 illustrates how the gains achieved by our new kernel translate at coarser granularity levels, when inserting butterfly factors into a butterfly factorization, or into a vision transformer. Section 9.7 discusses perspectives. Experimental details are deferred to Appendix D.

9.2 Preliminaries

We introduce in Chapter 7 a mathematical framework to describe the butterfly supports in Definition 7.2. To the best of our knowledge, this framework cap-

[1] *Many* operations (fully-connected layers, convolutional layers, batch norm, attention module, etc.) in PyTorch requires the batch size to be the first dimension of the input tensor.

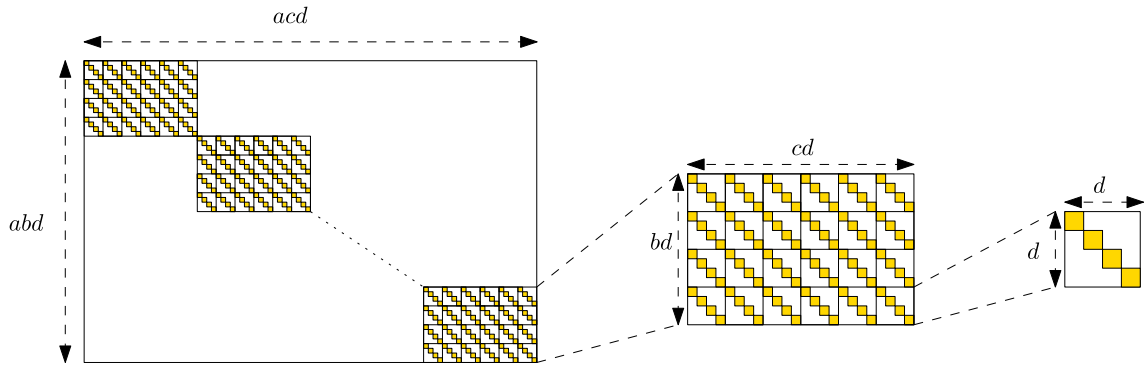


Figure 9.1: A π-butterfly factor with $\pi = (a, b, c, d)$ is a block-diagonal matrix with a diagonal blocks, each block itself is a block matrix composed by $b \times c$ diagonal matrices of size $d \times d$. In particular, each of the a diagonal blocks *follows the sparsity pattern* $(1, b, c, d)$ (case $a = 1$). Therefore, the multiplication in the general case (arbitrary a) can be reduced to the case $a = 1$ by performing in parallel the a multiplications by each diagonal block.

tures all the variants of butterfly factorizations that have been empirically tested for deep neural networks in the literature [71, 72, 74, 113, 241, 342], as detailed in Table 7.1.

According to Definition 7.2 and as illustrated in Figure 1.1, a π-butterfly factor is *sparse* and *structured*, in the sense that it has a support of the form $\mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d$ where $\pi = (a, b, c, d)$. Indeed, in comparison to the dense matrix of the same size $abd \times acd$, the butterfly factor has at most $abcd$ nonzero entries, which yields a sparsity ratio $\frac{abcd}{a^2bcd^2} = \frac{1}{ad}$.

Butterfly matrices in neural networks. The typical application of butterfly matrices we have in mind is to replace fully-connected layers in neural networks in order to accelerate their inference. Our primary focus is on vision transformers (ViTs) [87] as we find that the computational cost of fully-connected layers is significant in such architectures: depending on the size of the ViT, from 30% to 60% of the total time in a forward pass is spent in fully-connected layers (see Appendix D.3 for details). We will not consider convolutional layers, as we did not find the existing method for integrating butterfly factorization [241] satisfactory (see Appendix D.4 for details).

Sparsity patterns (a, b, c, d) with $a = 1$ or $d = 1$ are foundational ones. This chapter focuses on butterfly sparsity patterns (a, b, c, d) satisfying either $a = 1$ or $d = 1$ for three reasons. First, a possible algorithm for the general case (a, b, c, d) is to use a times in parallel a specialized implementation for $a = 1$ on each of the a diagonal blocks with sparsity patterns $(1, b, c, d)$, cf. Figure 9.1. Second, to the best of our knowledge, the only algorithm specialized to $a = 1$ [72] is based on an efficient implementation in the case $d = 1$. Last, the two cases $a = 1$ and

$d = 1$ appear in every architecture β that are chainable (Definition 7.5) and for which Σ^β can express dense matrices[2], which covers all the architectures used in practice in neural networks [71, 72, 74, 113, 241, 342]. In particular, $a = 1$ or $d = 1$ covers all such architectures of length $L = 2$, which is typically the length of architectures used in [72, 113].

9.3 Existing PyTorch implementations for $a = 1$ or $d = 1$

We now introduce the following baseline PyTorch implementations for the matrix multiplication by a single butterfly factor:

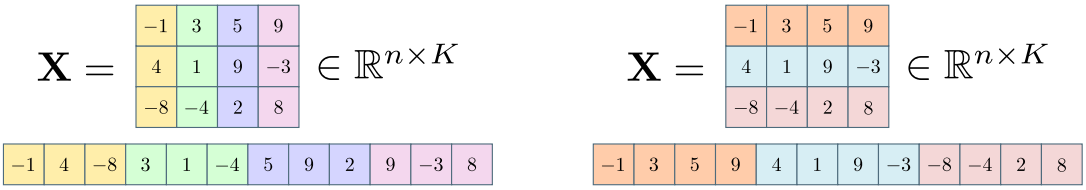
1. the dense matrix multiplication algorithm (simply called `dense`) where the sparse structure of the butterfly factor is not leveraged;
2. the generic sparse matrix multiplication algorithm (simply called `sparse`) in PyTorch, which leverages the sparsity of the butterfly factor by storing it in the CSR format, but that does not leverage its structure;
3. the butterfly matrix multiplication proposed in [72], called `monarch`, which leverages both the sparsity and the structure of the butterfly factor.

In the following, the batch size is denoted by K, the input dimension is n and the output dimension is m. Note that in the transformer architecture, the input tensor is a three-dimensional array where the dimensions are the data batch size, the number of tokens, and the embedding dimension. In the following, the batch size K will be the "effective" batch size defined as the product of the data batch size times the number of tokens. Therefore, the considered inputs will be a batch of K vectors.

***Batch-size-first* vs. *batch-size-last*.** Before detailing each of these algorithms, we emphasize that each exists in two variants: *batch-size-first* and *batch-size-last*. Consider the matrix multiplication $\mathbf{Y} = \mathbf{B}\mathbf{X} \in \mathbb{R}^{m\times K}$ where $\mathbf{B} \in \mathbb{R}^{m\times n}$ and $\mathbf{X} \in \mathbb{R}^{n\times K}$. The two different settings *batch-size-first* and *batch-size-last* correspond to two different memory layouts for the PyTorch tensors encoding the matrices $\mathbf{X}$ and $\mathbf{Y}$ respectively:

- In *batch-size-first*, which is the *default PyTorch convention*, the PyTorch tensors `input_bsf` and `output_bsf` are tensors of shape (K, n) and (K, m) respectively, in such a way that the slices `input_bsf`$[k]$ and `output_bsf`$[k]$ for $0 \leq k \leq K-1$ contains the k-th column of $\mathbf{X}$ and $\mathbf{Y}$. Entries of PyTorch tensors are always saved in *row-major* order, i.e., the entries of the slices `input_bsf`$[k]$ or `output_bsf`$[k]$ are stored next to each other in memory.

[2] i.e., denoting $\beta = (\boldsymbol{\pi}_\ell)_{\ell=1}^L$, $(\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_L) = (1, m, n, 1)$ for some integers m, n, cf. Lemma 7.4



(a) `input_bsf` (b) `input_bsl`

Figure 9.2: Illustration of the memory layout in *batch-size-first* and *batch-size-last* for a same input matrix $\mathbf{X} \in \mathbb{R}^{n \times K}$, where $n = 3$ and $K = 4$ in the example. The array at the bottom shows entries that are contiguous in memory when saved in the tensor `input_bsf` or `input_bsl`. K is the batch size while n is the input dimension.

Consequently, each column of $\mathbf{X}$ or $\mathbf{Y}$ (that corresponds to each input/output vector in the batch of vectors) are stored contiguously in memory in their respective PyTorch tensor `input_bsf` or `output_bsf`.

- The opposite (non-standard) convention is called *batch-size-last*[3]. In this convention, the matrices $\mathbf{X}$ and $\mathbf{Y}$ are stored in tensors `input_bsl` and `output_bsl` of shape (n, K) and (m, K) respectively, where the slices `input_bsl`$[i]$ for $0 \leq i \leq n-1$ or `output_bsl`$[i]$ for $0 \leq i \leq m-1$ contains the i-th row of $\mathbf{X}$ or $\mathbf{Y}$. Again, since PyTorch tensors are in row-major order, each row of $\mathbf{X}$ or $\mathbf{Y}$ (that corresponds to the vector containing the i-th coordinate of each input/output vector in the batch) are stored contiguously in memory in their respective PyTorch tensor `input_bsl` and `output_bsl`. See Figure 9.2 for an illustration of the two different memory layouts.

We now detail the baseline algorithms for butterfly matrix multiplication.

`dense` and `sparse`. The default PyTorch implementation of a linear layer is in *batch-size-first*: `torch.nn.functional.linear(input_bsf, weight)`, with a tensor `weight` of shape (m, n) saving $\mathbf{B}$. This is the `sparse` algorithm when $\mathbf{B}$ is stored in the CSR format, and the `dense` algorithm when $\mathbf{B}$ is stored in the usual dense format. To the best of our knowledge, there is no default implementation in *batch-size-last* for `dense` and `sparse`, because the default PyTorch convention is *batch-size-first*. Therefore, for *batch-size-last*, we propose to use the implementation `torch.matmul(weight, input_bsl)` with `weight` of shape (m, n) saving $\mathbf{B}$, because we find out that it is the fastest among the different alternatives we tried (cf. Appendix D.5).

`monarch` for block-diagonal matrix with dense sub-blocks ($d = 1$). A butterfly factor $\mathbf{B} \in \Sigma^{\pi}$ with sparsity pattern $\boldsymbol{\pi} = (a, b, c, 1)$ is a block-diagonal matrix

[3]We introduce this terminology by analogy with the recent PyTorch optimization *channels-last* that moves the channels dimension to the last position for input tensors of convolutional layers.

with a dense blocks $(\mathbf{B}_i)_{i=1}^a$ of size $b \times c$, cf. Figure 9.1. This implies a straightforward algorithm for efficient matrix multiplication: perform in parallel the multiplications with the dense diagonal submatrices $\mathbf{B}_i$ using a dense multiplication algorithm (see Appendix D.6 for a pseudo-code). The `monarch` algorithm [72] is an efficient GPU implementation of this in *batch-size-first*, using the `torch.bmm` (batched matrix multiplication) routine of PyTorch. We adapt their implementation to the *batch-size-last* setting.

`monarch` for block-diagonal matrix up to permutations ($a = 1$). A butterfly factor $\mathbf{B} \in \Sigma^{\boldsymbol{\pi}}$ with sparsity pattern $\boldsymbol{\pi} = (1, b, c, d)$ is block-diagonal *up to certain row and column permutations*, with d dense submatrices $(\mathbf{B}_i)_{i=1}^d$ of size $b \times c$ for the diagonal blocks after permutations. We detail these permutations in the following lemma.

Lemma 9.1. *For any integer b, c, d:*

$$\mathbf{P}_{b,d}{}^\top (\mathbf{1}_{b\times c} \otimes \mathbf{I}_d) \mathbf{P}_{c,d} = \mathbf{I}_d \otimes \mathbf{1}_{b\times c},$$

where $\mathbf{P}_{p,q}$ denotes the (p, q) perfect shuffle *of $r := pq$ [343] defined as:*

$$\mathbf{P}_{p,q} := \begin{pmatrix} \mathbf{I}_r[\{1 + qj\}_{j\in[\![0,p-1]\!]}, :] \\ \mathbf{I}_r[\{2 + qj\}_{j\in[\![0,p-1]\!]}, :] \\ \vdots \\ \mathbf{I}_r[\{q + qj\}_{j\in[\![0,p-1]\!]}, :] \end{pmatrix}. \tag{9.1}$$

Consequently, denoting $\boldsymbol{\pi} = (1, b, c, d)$ and $\boldsymbol{\pi}' = (d, b, c, 1)$:

$$\forall \mathbf{B} \in \Sigma^{\boldsymbol{\pi}}, \quad \mathbf{P}_{b,d}{}^\top \mathbf{B} \mathbf{P}_{c,d} \in \Sigma^{\boldsymbol{\pi}'}.$$

Proof. The first equality is a consequence of the general result given in [343], which claims that the Kronecker product commutes up to some perfect shuffle permutation matrices. The second equality comes simply from the fact that $\mathrm{supp}(\mathbf{X}) \subseteq \mathrm{supp}(\mathbf{S})$ implies that $\mathrm{supp}(\mathbf{PXQ}) \subseteq \mathrm{supp}(\mathbf{PSQ})$ for any matrix $\mathbf{X}$, binary matrix $\mathbf{S}$, and permutation matrices $\mathbf{P}, \mathbf{Q}$. □

Therefore, *the case $a = 1$ corresponds to the case $d = 1$ up to permutations*, because

$$\mathbf{B} = \mathbf{P}_{b,d} \mathbf{B}' \mathbf{P}_{c,d}{}^\top \quad \text{with} \quad \mathbf{B}' := \mathbf{P}_{b,d}{}^\top \mathbf{B} \mathbf{P}_{c,d},$$

where $\mathbf{B}'$ is a butterfly factor associated with $\boldsymbol{\pi}' = (d, b, c, 1)$, i.e., it is a block-diagonal matrix with dense blocks of size $b \times c$ (case $a = 1$). Hence, the multiplication $\mathbf{X} \mapsto \mathbf{BX}$ can be decomposed sequentially into three steps: $\mathbf{X} \mapsto \mathbf{P}_{c,d}{}^\top \mathbf{X}$, $\mathbf{X} \mapsto \mathbf{B}'\mathbf{X}$, and $\mathbf{X} \mapsto \mathbf{P}_{b,d}\mathbf{X}$. In *batch-size-first*, the `monarch` algorithm [72] implements the permutation operations $\mathbf{X} \mapsto \mathbf{P}_{c,d}{}^\top \mathbf{X}$ and $\mathbf{X} \mapsto \mathbf{P}_{b,d}\mathbf{X}$ via tensor reshaping and transposition on the input tensor, and uses the same algorithm as in

$d = 1$ for the operation $\mathbf{X} \mapsto \mathbf{B}'\mathbf{X}$ (see Appendix D.6 for a pseudo-code). Note that their implementation for the permutation operations on the input tensor involve real memory rearrangements, and not simply a mere reindexing of matrix enumerators, meaning that they can have a non-negligible cost in practice. For the *batch-size-last* setting, we adapt the implementation from [72] originally provided only in *batch-size-first*.

9.4 New CUDA implementation

Let $\mathbf{X} \in \mathbb{R}^{n \times K}$ be the input matrix and $\mathbf{B} \in \Sigma^{\boldsymbol{\pi}}$ be a butterfly factor of size $m \times n$ with sparsity pattern $\boldsymbol{\pi} = (a, b, c, d)$, where $m = abd$ and $n = acd$. The goal is to compute $\mathbf{Y} = \mathbf{B}\mathbf{X} \in \mathbb{R}^{m \times K}$. We propose a new implementation called `kernel`, for both cases $a = 1$ and $d = 1$.

Remark 9.1. *When $d = 1$, the butterfly factor $\mathbf{B}$ is block-diagonal with dense sub-blocks (Figure 9.1). Improving `monarch` in this scenario is challenging since it exclusively relies on highly competitive NVIDIA routines for multiplying each dense sub-block. However, in the case $a = 1$, the proposed implementation `kernel` aims at reducing the computation time of the permutation operations involved in `monarch` discussed above.*

For the case $a = 1$, consider a set I of rows in $\mathbf{B}$ sharing the same sparsity pattern, i.e., the same set of indices corresponding to nonzero entries. Then, as shown in Figure 9.3, there is a subset J of column indices for $\mathbf{B}$ such that the only nonzero columns of $\mathbf{B}[I, :]$ are those indexed by J. Therefore, the restriction of the output $\mathbf{Y} = \mathbf{B}\mathbf{X}$ to the rows indexed by I, denoted by $\mathbf{Y}[I, :]$, is equal to the multiplication of the *dense* submatrices $\mathbf{B}[I, J]$ and $\mathbf{X}[J, :]$. The `monarch` algorithm calls dense NVIDIA routines to perform $\mathbf{B}[I, J]\mathbf{X}[J, :]$ for each different set I, but before that, the dense submatrix $\mathbf{X}[J, :]$ is required to be stored contiguously in memory in the input tensor, because the dense NVIDIA routines cannot be called from PyTorch with non-contiguous entries. Therefore, `monarch` explicitly permutes the entries in the input tensors by performing tensor *reshaping and transposition* to ensure that entries of $\mathbf{X}[J, :]$ are stored contiguously in memory (cf. Algorithm D.2).

Main design feature: `kernel` encodes permutations in the reading phase. In total, `monarch` reads twice the entries of $\mathbf{X}[J, :]$ (once to perform the permutation operations, and another time to perform the dense multiplication $\mathbf{B}[I, J]\mathbf{X}[J, :]$), and writes them once (after permutations in order to make them contiguous in memory). In contrast, `kernel` only reads the entries of $\mathbf{X}[J, :]$ once, and avoids rewriting them. This is possible by implementing the butterfly matrix multiplication in CUDA, but not in PyTorch. Once the entries of $\mathbf{X}[J, :]$ are loaded, they remain in memory for the multiplication $\mathbf{B}[I, J]\mathbf{X}[J, :]$. In other words, the initial reading is directly used to perform the multiplication with $\mathbf{B}[I, J]$.

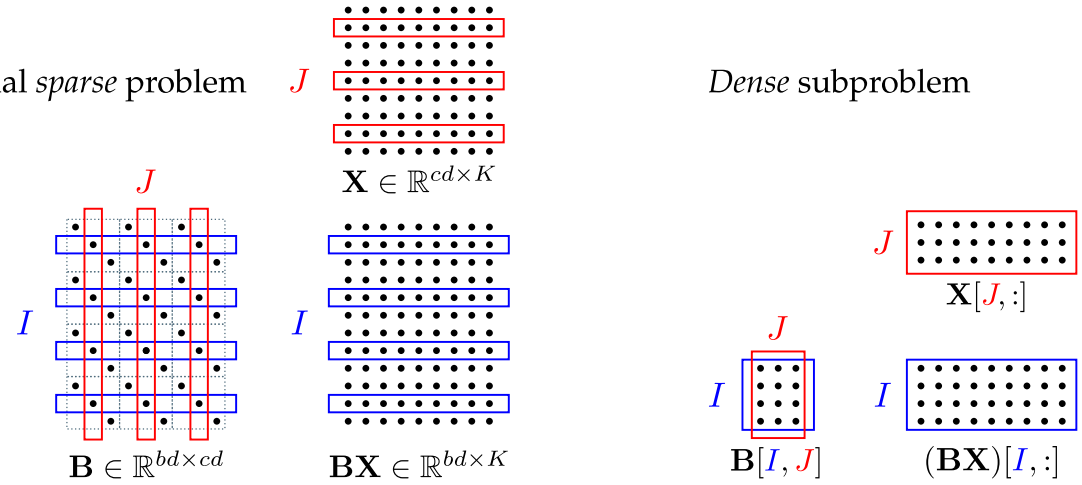


Figure 9.3: The figure depicts a butterfly factor with sparsity pattern $\boldsymbol{\pi} = (1,4,3,3)$ (case $a = 1$). Each set I of rows in $\mathbf{B}$ with the *same sparsity pattern* has only a subset J of nonzero columns. Consequently, only the rows J of $\mathbf{X}$ should be considered for multiplication, and computing the rows I of the output reduces to a dense multiplication: $(\mathbf{BX})[I,:] = \mathbf{B}[I,J]\mathbf{X}[J,:]$. The proposed implementation `kernel` avoids read and write operations on the entries of $\mathbf{X}[J,:]$ that were required in `monarch` in order to make them contiguous in memory and call dense NVIDIA implementations.

In practice, `kernel` implements the classical tile matrix multiplication algorithm[4] [29,225,283–285], where each block of threads computes in parallel a single tile of the output. However, for the specific case of $a = 1$, a *non-standard* selection of tiles is enforced: the tiles are chosen with *non-consecutive rows* to adapt to the sparsity pattern of $\mathbf{B}$. Each tile exclusively corresponds to rows of $\mathbf{B}$ with the same sparsity pattern, evenly spaced by d in this context.

Despite not relying on NVIDIA routines for the dense multiplication part, the current implementation of `kernel` proves to be the fastest in many cases in *float-precision*, see Section 9.5 below. In *half-precision*, the results in Section 9.5 suggest potential optimization opportunities. The open-source code release of `kernel` allows a plug-and-play scenario, enabling users to experiment with their preferred algorithms for the dense multiplication $\mathbf{B}[I,J]\mathbf{X}[J,:]$ and see if it improves the current implementation.

***Batch-size-last* is expected to be favorable when $a = 1$.** The efficiency of read and write operations relies on accessing *adjacent* elements in memory. Therefore, an efficient implementation requires the entries of $\mathbf{X}[J,:]$ and $\mathbf{Y}[I,:]$ to be stored contiguously in memory. In our situation (Figure 9.3), the subset I and J contains non-consecutive integers. This means that efficient access to adjacent elements in memory is only possible if the entries of each *row* of $\mathbf{X}$ and $\mathbf{Y}$ are adjacent in memory. Therefore, *batch-size-last* is expected to be more favorable than *batch-size-first*, for the multiplication by a butterfly factor when $a = 1$.

[4]The classical optimizations that are used in our implementation are detailed in Appendix D.7.

Table 9.1: Percentage out of 300 cases where `algo1` is *faster* than the `algo2` (denoted by `algo1` $<$ `algo2`), and the median acceleration factor in such cases (that is, the median ratio $\frac{\text{time of algo2}}{\text{time of algo1}}$). FP32 denotes *float-precision* and FP16 denotes *half-precision*.

	$\texttt{dense} < \texttt{sparse}$		$\texttt{monarch} < \min\begin{pmatrix}\texttt{dense}\\ \texttt{sparse}\end{pmatrix}$	
	Batch-size-first	***Batch-size-last***	***Batch-size-first***	***Batch-size-last***
$d = 1$, fp16	100% ($\times$114)	100% ($\times$13.0)	74% ($\times$8.2)	75% ($\times$7.9)
$a = 1$, fp16	100% ($\times$143)	100% ($\times$13.2)	62% ($\times$3.1)	60% ($\times$4.1)
$d = 1$, fp32	88% ($\times$7.0)	42% ($\times$2.4)	90% ($\times$12.6)	90% ($\times$6.7)
$a = 1$, fp32	91% ($\times$7.8)	43% ($\times$2.5)	86% ($\times$9.3)	86% ($\times$4.5)

	$\min\begin{pmatrix}\texttt{kernel}\\ \texttt{monarch}\end{pmatrix} < \min\begin{pmatrix}\texttt{dense}\\ \texttt{sparse}\end{pmatrix}$		$\texttt{kernel} < \min\begin{pmatrix}\texttt{monarch}\\ \texttt{dense}\\ \texttt{sparse}\end{pmatrix}$	
	Batch-size-first	***Batch-size-last***	***Batch-size-first***	***Batch-size-last***
$d = 1$, fp16	78% ($\times$7.6)	76% ($\times$7.7)	4.2% ($\times$1.07)	1.3% ($\times$1.1)
$a = 1$, fp16	64% ($\times$3.1)	64% ($\times$4.0)	1.4% ($\times$1.08)	21% ($\times$1.2)
$d = 1$, fp32	96% ($\times$11.3)	94% ($\times$6.8)	37% ($\times$1.2)	30% ($\times$1.2)
$a = 1$, fp32	96% ($\times$7.1)	94% ($\times$6.1)	25% ($\times$1.2)	85% ($\times$1.3)

9.5 Benchmarking the multiplication by a single butterfly factor

Currently, and to the best of our knowledge, no GPU efficiency benchmark exists for butterfly matrices, even at the most basic level of the multiplication with a *single butterfly factor*. This section addresses this gap, and identifies situations where the new kernel implementation is faster than previous implementations.

Protocol. The benchmark is run on nearly 600 sparsity patterns $(a, b, c, d) \in A \times B \times B \times A$ satisfying ($b = c$ or $b = 4c$ or $c = 4b$) and ($a = 1$ or $d = 1$). We consider

$$A := \{1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128\},$$
$$B := \{48, 64, 96, 128, 192, 256, 384, 512, 768, 1024\}.$$

This corresponds to dimensions of the butterfly factor $\mathbf{B} \in \mathbb{R}^{abd \times acd}$ that could be used in ViTs. The sparsity patterns are divided into two categories: $(x, b, c, 1)$ (case $d = 1$) and $(1, b, c, x)$ (case $a = 1$). The cases $a = 1$ and $d = 1$ are closely

related, since the pattern $(x, b, c, 1)$ corresponds to $(1, b, c, x)$ up to permutations by Lemma 9.1. Further details on the protocol are given in Appendix D.1.

Our benchmark shows four main results.

9.5.1 *Batch-size-last* substantially improves `sparse` and `kernel`, but does not change `monarch` nor `dense`

Figure 9.4 shows that in *float-precision*, the implementation in *batch-size-last* is approximately ten times *faster* than the one in *batch-size-first* for `sparse`, and twice *faster* for `kernel` in the case $a = 1$, while this has no impact on `monarch` nor `dense`. The results are similar in *half-precision* (Appendix D.8). While we cannot explain this acceleration for `sparse` since the code is not public, we explained in Section 9.4 that this is expected for `kernel` due to a favorable memory layout of the matrices in this case.

The acceleration of `sparse` in *batch-size-last* substantially improves in *float-precision* the generic baseline $\min($`dense`, `sparse`$)$ that considers the best out of the generic `dense` and `sparse` algorithms. Indeed, the execution time of `dense` does not change between *batch-size-first* and *batch-size-last* (Figure 9.4), while `sparse` becomes faster than `dense` in *batch-size-last*, cf. `dense` $<$ `sparse` in Table 9.1.

The acceleration of `kernel` in *batch-size-last* when $a = 1$ improves the situation by becoming the fastest implementation in 20% and 60% *additional cases* in *half-precision* and *float-precision* compared to *batch-size-first*, respectively, cf. `kernel` $<$ $\min($`monarch`, `dense`, `sparse`$)$ in Table 9.1. See Figure 9.5 for an example of sparsity patterns where `kernel` improves over existing algorithms.

9.5.2 Implementations dedicated to butterfly are faster

The column $\min($`kernel`, `monarch`$) < \min($`dense`, `sparse`$)$ in Table 9.1 shows that algorithms specialized to butterfly sparsity (`kernel` and `monarch`), are most of the time *faster* than generic ones (`dense` and `sparse`). For small matrix size, `kernel` and `monarch` are slower than `dense`. As the matrix size increases, `kernel` and `monarch` becomes faster than `dense`. This behaviour is illustrated in Figure 9.5 where we quantify the asymptotic behavior of the algorithms by fitting a power law $\texttt{time} = kx^{\nu}$ as a function of x, benchmarking either on the patterns $(1, b, c, x)$ or $(x, b, c, 1)$. We find experimentally that all the sparse algorithms (`sparse`, `monarch`, `kernel`) scale asymptotically *linearly* with x, whereas the dense algorithm scales *quadratically* with x, as expected by the theory[5]. The asymptotic analysis also illustrates the benefit of a dedicated butterfly sparse implementation compared to generic ones: `kernel` and `monarch` have a smaller proportional constant k in the power law than `sparse` up to one order of magnitude (Figure 9.5).

[5]These theoretical complexities are justified by the fact that the butterfly factor of sparsity pattern $(1, b, c, x)$ or $(x, b, c, 1)$ has bcx nonzero entries, and is of size $bx \times cx$.

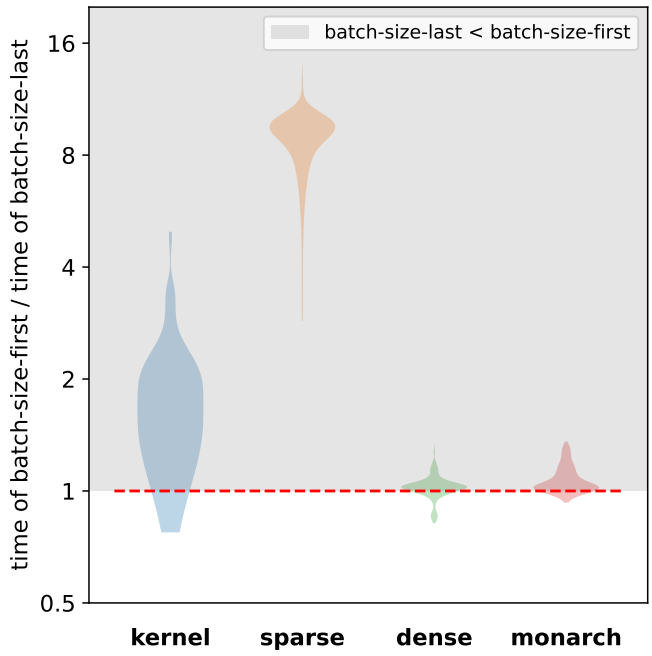


Figure 9.4: Violinplot of the ratios $\frac{\text{time of } \textit{batch-size-first}}{\text{time of } \textit{batch-size-last}}$ in 300 cases, for experiments in *float-precision* with $a = 1$. We observe similar behavior for *half-precision* as detailed in Figure D.2.

9.5.3 `monarch` $a = 1$ is consistently slower than $d = 1$

The sparsity patterns $(x, b, c, 1)$ (case $d = 1$) and $(1, b, c, x)$ (case $a = 1$) correspond to matrices of the same dimensions, with the same number of nonzeros. Therefore, the *theoretical* complexity of the dense and sparse algorithm are the same for these two tuples. In *practice*, we indeed observe that `sparse` and `dense`, that are agnostic to the sparsity pattern (a, b, c, d), have the *same execution times* for such a pair of tuples (Appendix D.9).

In contrast, we find the original implementation of `monarch` to be consistently slower for $a = 1$ compared to $d = 1$: Figure 9.6a shows that `monarch` for the tuple $(1, b, c, x)$ (case $a = 1$) is typically $\times 1.5$ and $\times 2.5$ times slower, in *float-precision* and *half-precision* respectively, compared to the tuple $(x, b, c, 1)$ (case $d = 1$). This gap between $a = 1$ and $d = 1$ can be interpreted as the *cost* for the memory rearrangements induced by the permutations operations in the case $a = 1$ for `monarch` (Section 9.2). Consequently, the proportion of tuples for which `monarch` $<$ min(`sparse`, `dense`) is smaller for $a = 1$ compared to $d = 1$, cf. Table 9.1.

9.5.4 The kernel improves the case $a = 1$

For $d = 1$, `monarch` only relies on dense NVIDIA implementations, which makes it very competitive and hard to improve. For $a = 1$, `monarch` performs memory

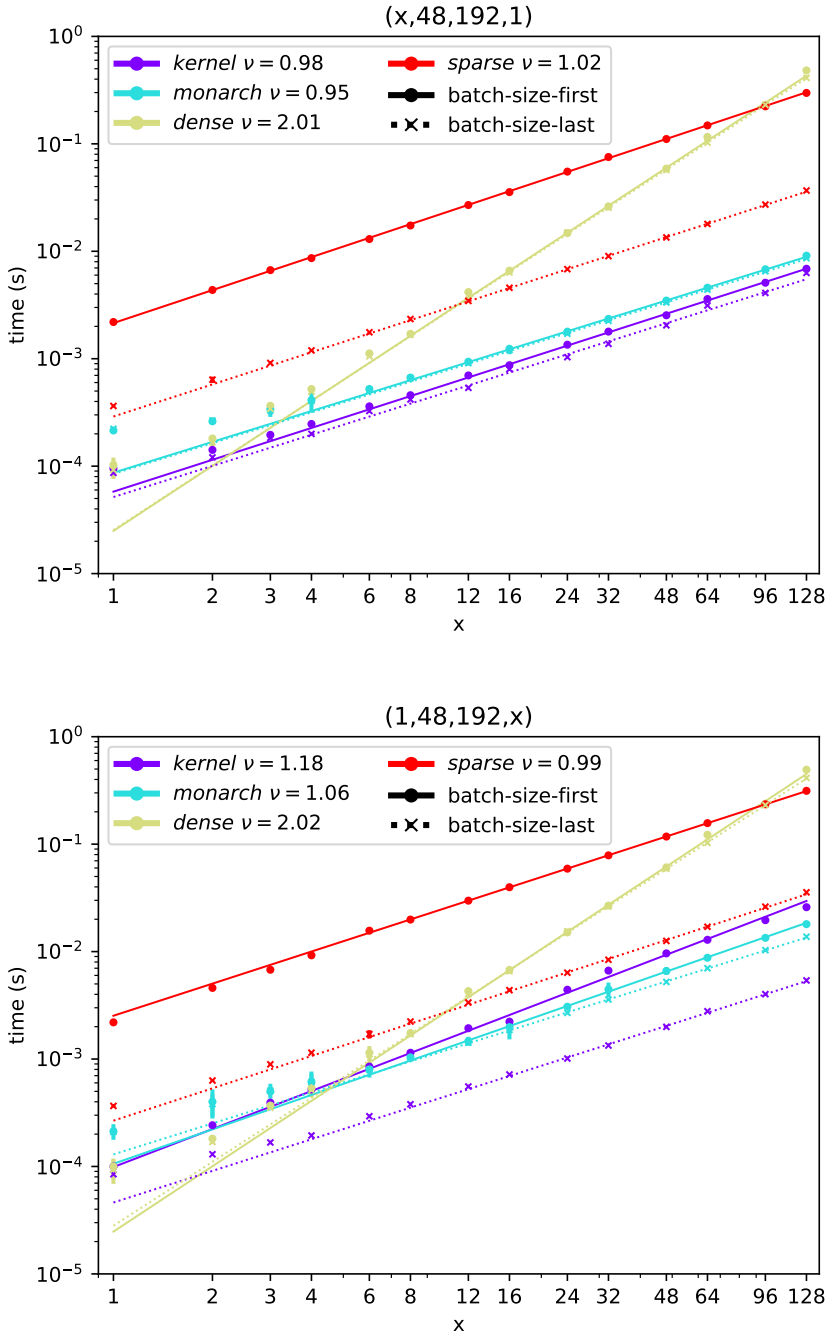
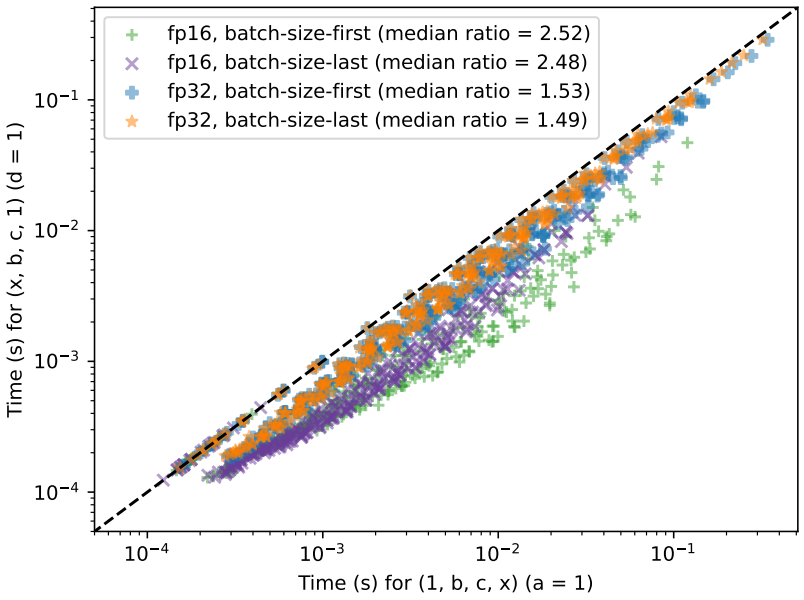
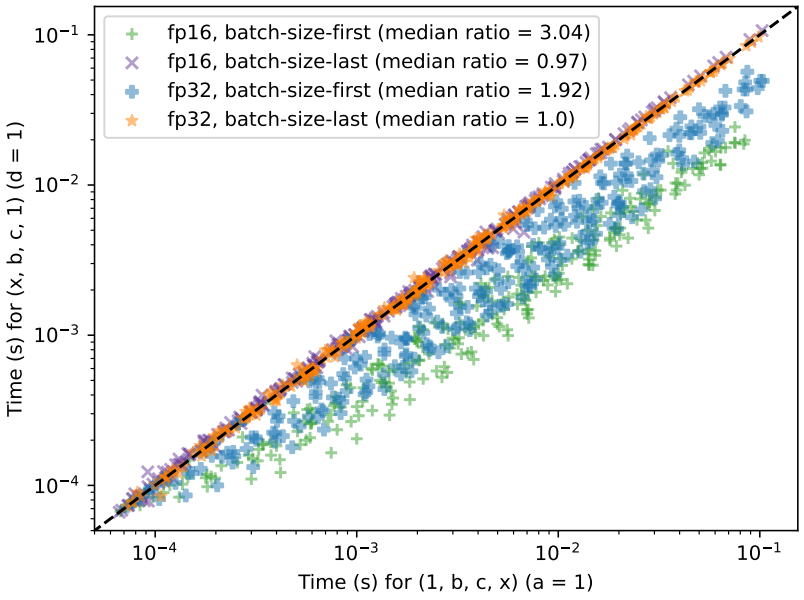


Figure 9.5: Median execution times (s) for matrix multiplication by a single butterfly factor $\mathbf{B}$ with sparsity pattern $(x, 48, 192, 1)$ (top) and with sparsity pattern $(1, 48, 192, x)$ (bottom), in *float-precision*. *Batch-size-first* is represented by circles while *batch-size-last* is represented by crosses. A power law $\texttt{time} = kx^{\nu}$ is fitted to the seven points with largest values of x of each curve. A solid line represents *batch-size-first* while a dotted line represents *batch-size-last*.



(a) `monarch`



(b) `kernel`

Figure 9.6: Median execution time (s) for a tuple $(1, b, c, x)$ (case $a = 1$) versus the one $(x, b, c, 1)$ (case $d = 1$). The median ratio corresponds to the median of $\frac{\text{time of } (1,b,c,x)}{\text{time of } (x,b,c,1)}$ over all possible b, c, x.

arrangements of the entries in the input tensor in order to reduce the case $a = 1$ to the case $d = 1$. As explained in Section 9.4, `kernel` avoids such memory operations.

Float-precision. In *batch-size-first*, the proposed `kernel` implementation already improves 25% of the 300 tested cases $a = 1$ compared to all other algorithms (Table 9.1). It gets substantially better in *batch-size-last* as it benefits from the memory layout of input tensor as explained in Section 9.4. This now makes it better than all baselines in 85% of the same 300 cases where $a = 1$. Beyond enhancing scenarios where `monarch` already outperformed other implementations (see Figure 9.5 for example), the improvements extend to cases where generic baselines min(`dense`, `sparse`) remained the superior choices, despite lacking specialization for butterfly sparsity. This can be seen in Table 9.1 by comparing columns `monarch` $<$ min(`dense`, `sparse`) *vs.* min(`kernel`, `monarch`) $<$ min(`dense`, `sparse`), where min(`kernel`, `monarch`) is faster than min(`dense`, `sparse`) in a broader range of cases than `monarch` alone.

Half-precision. As shown in Table 9.1, `kernel` in *half-precision* is not (yet) able to achieve the same acceleration observed in *float-precision*. First, replicating the exact approach used in *float-precision* for *half-precision* is currently not possible, as CUDA does not yet support vectorized four-by-four operations in *half-precision*: we can only perform two-by-two operations using `half2`, whereas `float4` is available for four-by-four operations in *float-precision*. Second, the `dense` algorithm is very competitive in *half-precision* due to the use of highly optimized NVIDIA routines for dense matrix multiplication such as TensorCores. This also makes `monarch` very competitive in the case $d = 1$ since it *only relies on the NVIDIA routines in this case* (no additional permutations). Despite all that, we still manage to improve 21% of the 300 tested cases for $a = 1$ for *half-precision* in the *batch-size-last* setting, see `kernel` $<$ min(`monarch`, `dense`, `sparse`) in Table 9.1.

9.6 Benchmarking at coarser granularities

This section verifies if the acceleration provided by the proposed `kernel` implementation for a single butterfly factor $\mathbf{B}$ is preserved at *coarser* granularities, when a butterfly factor is inserted into a butterfly factorization $\mathbf{W} = \mathbf{B}_1 \dots \mathbf{B}_L$ (Section 9.6.1), and when it is inserted into a neural network (Section 9.6.2).

9.6.1 Multiplication by a butterfly matrix

For a butterfly matrix $\mathbf{W} = \mathbf{B}_1 \dots \mathbf{B}_L$, is it better to directly compute $\mathbf{WX}$ with a dense implementation, by directly storing $\mathbf{W}$ as a dense matrix (ignoring the sparse factorization), or is it better to sequentially perform the multiplication with each factor using specialized sparse algorithms?

Protocol. We focus on butterfly matrices $\mathbf{W} = \mathbf{B}_1\mathbf{B}_2$ associated with an architecture (π_1, π_2) of length $L = 2$, where π_1 is of the form $(1, b, c, d)$ and π_2 is of the form $(a, b, c, 1)$, and for which the values of π_1 and π_2 correspond to the ones benchmarked in Section 9.5 (see Appendix D.10 for more details). The dimensions of $\mathbf{W}$ are chosen to match the ones of weight matrices in ViTs (e.g., ViT-S/16, B/16, L/16).

Results. In *float-precision*, Table 9.2 shows that `kernel` improves over the baseline min(`monarch`, `dense`) in 72% of the 460 tested architectures in the *batch-size-last* setting (typically $\times 1.16$). For the *batch-size-first* setting, `kernel` still improves over min(`monarch`, `dense`) in 27% of the 460 tested architectures (typically $\times 1.10$).

In *half-precision*, we did not find an architecture among the tested ones for which `kernel` improves over both `monarch` and `dense`. This is somehow expected, because in the benchmark for a single butterfly factor (Section 9.5), even though the `kernel` improves over all previous implementations on some patterns in the case of $a = 1$, it does not improve in many cases when $d = 1$ (Table 9.1).

This motivates a `hybrid` approach (see Appendix D.10 for details) where the sparse multiplication $\mathbf{X} \mapsto \mathbf{B}_1\mathbf{B}_2\mathbf{X}$ uses `kernel` for $a = 1$ ($\mathbf{B}_1$), and `monarch` for $d = 1$ ($\mathbf{B}_2$). While this approach does not do better than `kernel` alone compared to min(`monarch`, `dense`) in *batch-size-first*, it significantly improves in *batch-size-last* with *float-precision*. Moreover, we find architectures in *half-precision* for which `hybrid` is faster than both `monarch` and `dense` in *batch-size-last*. This is the case for 33% architectures out of 24 for the size 1024×4096 (typically $\times 1.08$), and 21% architectures out of 24 for the size 4096×1024 (typically $\times 1.06$).

9.6.2 Butterfly matrices in a neural network

Section 9.6.1 identified architectures for which the associated butterfly matrix multiplication is accelerated with the `kernel` implementation, and for which the sizes correspond to those encountered in weight matrices of ViT. So how do the observed gains translate when inserting butterfly matrices in neural networks?

Protocol. We benchmark various components of a ViT-S/16 architecture: a linear layer with bias, an MLP with non-linear activation and/or normalization layers, a multi-head attention module, etc. As in [72], we replace by a butterfly matrix the weight matrices of linear layers in feed-forward network modules, and the projection matrices for keys, queries and values in multi-head attention modules. We focus on *float-precision*, since the case of *half-precision* still requires more optimization (Section 9.6.1), and we also focus on *batch-size-first*, since the insertion of butterfly matrices in the *batch-size-last* setting would *a priori* require a careful implementation of a ViT in *batch-size-last*[6].

[6] As *batch-size-first* is the default convention of PyTorch, optimized implementations for the rest of the operations present in neural networks are only available in this setting for the mo-

Table 9.2: Percentage of butterfly architectures (π_1, π_2) for which the multiplication $\mathbf{X} \mapsto \mathbf{B}_1\mathbf{B}_2\mathbf{X}$ is faster with `kernel` than both `monarch` and `dense`. We consider between 24 and 90 different butterfly architectures for each size $m \times n$ of the product $\mathbf{B}_1\mathbf{B}_2$. In parenthesis is the median acceleration factor ($\frac{\min(\text{time of } \texttt{dense}, \text{time of } \texttt{monarch})}{\text{time of } \texttt{kernel}}$) computed only in the cases where `kernel` is faster than `monarch` and `dense`.

	`kernel` $<$ min(`dense`, `monarch`), in *float-precision*	
$m \times n$	***Batch-size-first***	***Batch-size-last***
384×384	52% ($\times$1.06)	60% ($\times$1.20)
768×768	21% ($\times$1.09)	72% ($\times$1.20)
1024×1024	25% ($\times$1.05)	63% ($\times$1.16)
384×1536	64% ($\times$1.17)	85% ($\times$1.25)
768×3072	46% ($\times$1.10)	73% ($\times$1.21)
1024×4096	38% ($\times$1.14)	46% ($\times$1.22)
1536×384	4% ($\times$1.03)	78% ($\times$1.12)
3072×768	0% (N/A)	72% ($\times$1.11)
4096×1024	0% (N/A)	67% ($\times$1.05)
All sizes	27% ($\times$1.10)	72% ($\times$1.16)

Results. Table 9.3 shows the ranking (smaller is better): `kernel` $<$ `monarch` $<$ `dense` over all the different submodules. This offers promising perspectives to achieve similar accelerations in *half-precision* and in *batch-size-last*.

9.7 Conclusion

This work presented the first GPU benchmark for butterfly sparse matrix multiplication with sparsity patterns (a, b, c, d) satisfying $a = 1$ or $d = 1$. In such cases, we also introduced a novel kernel that circumvents costly memory operations in the fastest existing implementation for $a = 1$. The proposed kernel is especially beneficial when deviating from PyTorch convention by placing the batch dimension in last. Such a convention also improves generic sparse algorithm with the CSR format, without changing the time for the dense algorithm, which opens interesting perspectives of adopting this convention to promote efficient sparse algorithm on GPU. Our numerical results show that the proposed kernel improves over all existing implementations in *float-precision* for various butterfly sparsity patterns. We now discuss some perspectives of this work.

ment. It remains an open question whether similar optimizations can be extended to the *batch-size-last* setting (this would probably require going at the CUDA-level, as discussed further in Appendix D.11).

Table 9.3: Acceleration of various subparts of a ViT-S/16 where weight matrices are replaced by butterfly matrices associated with the following architectures: $(1, 192, 48, 2), (2, 48, 192, 1)$ for the size $n \times n$, $(1, 768, 192, 2), (6, 64, 64, 1)$ for the size $4n \times n$, $(1, 768, 192, 2), (6, 64, 64, 1)$ for the size $4n \times n$.

	Float-precision, batch-size-first	
	$\frac{\text{time of } \texttt{monarch}}{\text{time of } \texttt{dense}}$	$\frac{\text{time of } \texttt{kernel}}{\text{time of } \texttt{dense}}$
Linear $n \times n$	0.82	**0.50**
Linear $n \times n$ + bias	0.97	**0.66**
Linear $4n \times n$	0.80	**0.78**
Linear $4n \times n$ + bias	0.93	**0.90**
Linear $n \times 4n$	0.91	**0.58**
Linear $n \times 4n$ + bias	0.94	**0.61**
Feed-forward network	0.91	**0.77**
Multi-head attention	0.87	**0.79**
Block	0.90	**0.78**
Butterfly ViT-S/16	0.89	**0.78**

Improvement for *half-precision*. It is an open question whether the case of *half-precision* could be improved. As explained in Section 9.4, after the reading phase, the proposed `kernel` implementation performs dense multiplication between submatrices $\mathbf{B}[I, J]$ and $\mathbf{X}[J, :]$ without relying on NVIDIA routines. Future work can study whether using such routines into our kernel can improve the performance in *half-precision*. Moreover, a future potential release of `half4` which allows four-by-four vectorization operations in *half-precision* could further improve the proposed `kernel` implementation that relies currently on `half2` for two-by-two vectorization operations.

Extension to arbitrary sparsity patterns. A natural extension is to study butterfly factors with arbitrary sparsity patterns, and not only to the case $a = 1$ or $d = 1$. As explained in Section 9.2, by Definition 7.2, the general case for pattern $\boldsymbol{\pi} = (a, b, c, d)$ with arbitrary $a > 1$ can be implemented by parallelizing our implementation for the case $a = 1$ on the different diagonal blocks of pattern $(1, b, c, d)$, so future work can study how to extend our proposed implementation to perform batch matrix multiplication with several butterfly matrices of pattern $(1, b, c, d)$.

Further exploration of *batch-size-last*. Future work can clarify whether other PyTorch operations can be implemented in *batch-size-last* without loosing efficiency compared to existing *batch-size-first* implementations. Our discussion in Appendix D.11 shows promising perspectives towards this goal. This would

typically allow to insert butterfly matrices in a deep neural network with various PyTorch operations in the *batch-size-last* setting, which could lead to interesting acceleration of neural network with butterfly matrices, given the observed improvement of the `kernel` implementation in our benchmark for the *batch-size-last* setting (cf. Table 9.1). For further acceleration, one can also envision kernel fusion to optimize the implementation of a linear linear with a butterfly weight matrix *and a bias*, i.e., $\mathbf{x} \mapsto \mathbf{W}\mathbf{x} + \mathbf{b}$ for a butterfly matrix $\mathbf{W}$ of size $m \times n$ and a bias $\mathbf{b} \in \mathbb{R}^m$.

Comparaison to butterfly multiplication on other hardware. Finally, this work offers a baseline for comparisons of butterfly implementations on other hardwares: CPU, Intelligence Processing Unit (IPU), FPGA, etc. The butterfly sparsity is structured and efficient implementation on other hardwares can leverage this known structured pattern, e.g., as studied in [321] for IPU.

Chapter 10

Conclusion

This thesis studied challenges related to data annotation frugality and computational efficiency in deep learning. First, we studied representation learning via self-supervision for image data, which does not require any annotation for the images. Second, motivated by neural network compression, we studied butterfly sparse matrix factorization, which is a specific sparse matrix factorization inspired from the fast Fourier transform that enables rapid matrix-vector multiplication. We summarize our contributions, discuss their broader impact, and present some overall perspectives of this work.

10.1 Summary of the contributions

Self-supervised learning for visual representations. In Chapter 5, we proposed a unification of several self-supervised objective functions under the framework of rotation-invariant kernels (or dot-product kernels). In the pretext task where the learned representations are invariant to some specific image transformations, the loss of information due to collapse during pretraining can be avoided by adding a generic regularization loss in the objective function. Assuming that embeddings are ℓ_2-normalized, this proposed regularization minimizes an estimator of the maximum mean discrepancy between the embedding distribution and the uniform distribution on the hypersphere, associated with a certain rotation-invariant kernels. Several methods (contrastive, uniformity-based, information-maximization) can be unified under this framework: their objective function includes the minimization of this maximum mean discrepancy for different choices of rotation-invariant kernel. Under this framework, the question of choosing a good regularization for self-supervision can be reduced to the question of choosing a good kernel. In practice, we identified a rotation-invariant kernel never used before in self-supervised learning that yields competitive results, with reduced computational cost during pretraining compared to VICReg [15], due to the kernel trick.

Decomposition algorithms for butterfly factorization. We proposed new decomposition algorithms for butterfly factorization in three problem variations.

In Chapter 6, given a matrix $\mathbf{A}$ of size $n \times n$ with $n = 2^L$ ($L \geq 2$) that admits a square dyadic butterfly factorizations $\mathbf{X}_1 \dots \mathbf{X}_L$ (Definition 6.2), we asked whether the problem of recovering the butterfly factors $\mathbf{X}_1, \dots, \mathbf{X}_L$ is well-posed or not. We showed that these factors are indeed *essentially unique*, up to some unavoidable scaling ambiguities, and we proposed a hierarchical factorization algorithm that recovers these unique factors, with guarantees. The procedure is to recover the partial products of these factors via successive two-layer matrix factorizations, by applying truncated SVDs on some specific submatrices, until all the factors are recovered. The algorithm has a time complexity bounded by $\mathcal{O}(n^2)$, and enables fast $\mathcal{O}(n \log n)$ matrix-vector multiplication by $\mathbf{A}$. Experimentally, it outperforms gradient-based methods in terms of computation time and precision of the approximation.

In Chapter 7, we considered the problem $\min_{(\mathbf{X}_\ell)_{\ell=1}^L} \|\mathbf{A} - \mathbf{X}_1 \dots \mathbf{X}_L\|_F$ where the fixed-support constraints on the factors $\mathbf{X}_1, \dots, \mathbf{X}_L$ are more general: each factor has a support included in the one of $\mathbf{I}_a \otimes \mathbf{1}_{b \times c} \otimes \mathbf{I}_d$ for a certain $\boldsymbol{\pi} := (a, b, c, d) \in \mathbb{N}^4$. In this context, a sequence of patterns $\boldsymbol{\beta} := (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ describing the fixed-support constraints on the factors $(\mathbf{X}_\ell)_{\ell=1}^L$ is called a butterfly architecture, and we identified a new condition on butterfly architectures, called *chainability*, that covers many variants of the butterfly factorization used in deep learning applications. Importantly, for any butterfly factorization problem associated with a chainable architecture, we can construct an extension of the hierarchical butterfly endowed with *error guarantees*: we proved that the ratio of the approximation error by the best approximation error is bounded by a constant independent of the target matrix. As a consequence, we derived an analytical characterization of matrices admitting a butterfly factorization associated with a chainable architecture, in terms of low-rank properties of certain submatrices equivalent to a generalization of the *complementary low-rank property*.

In Chapter 8, we considered the problem $\min_{(\mathbf{X}_\ell)_{\ell=1}^L, \mathbf{P}, \mathbf{Q}} \|\mathbf{A} - \mathbf{Q}^\top \mathbf{X}_1 \dots \mathbf{X}_L \mathbf{P}\|_F$ where $\mathbf{X}_1, \dots, \mathbf{X}_L$ are butterfly factors associated with the square dyadic butterfly architecture, and $\mathbf{P}, \mathbf{Q}$ are unknown permutation matrices part of the optimization problem. We empirically showed that it is *necessary* to recover the right pair of permutations in order to solve the problem, in the sense that there is only one pair (up to some equivalences) that yields a small approximation error among all the possible pairs. For the identification of the optimal permutations, we relied on the complementary low-rank characterization of butterfly matrices, and proposed a heuristic based on alternating subspace clustering to identify row and column partitions of the target matrix yielding low-rank submatrices. We evaluated numerically the method for the butterfly decomposition of the DFT matrix and random orthogonal butterfly matrices. This validation was effective up to a specific matrix size, beyond which the method demonstrated reduced robustness to noise and became computationally prohibitive due to its cubic time complexity.

Implementing butterfly sparse matrix multiplication on GPU. In Chapter 9, we provided the first benchmark of various GPU implementations of the sparse matrix multiplication with the butterfly structure, in terms of time efficiency. Our benchmark reveals that the previous implementation specialized to the butterfly sparsity [72] involves some costly memory transfer operations for permuting elements in the input tensor, which can take up to half of the total time for matrix multiplication, in our experimental setting. We therefore proposed a new CUDA implementation that avoids these costly memory transfers, via a better management of the different levels of the GPU memory. In most of our tested cases, this new implementation improves over all existing methods in float-precision (with a $\times 1.2$ speed-up), but not always in half-precision.

Discussion about perspectives. Short-term perspectives related to the different contributions have been discussed at the conclusion of each corresponding chapter. Therefore, the next sections discuss more general perspectives related to this work.

10.2 Perspectives for self-supervised learning

In this section we comment about the general effectiveness of the pretext task considered in Chapter 5 for self-supervised learning, in terms of its dependence to the design of image transformations and to the tuning of the hyperparameters in the pretraining loss. We then discuss the possibility of using the proposed kernel framework to reduce the computational complexity of other pretraining losses in the literature.

Dependence on the design of image transformations. In the pretext task studied in Chapter 5, the quality of the learned representations heavily depends on the design of the image transformations that are applied to different views of the same image. The transformations should be adapted to the structure of the images in the pretraining dataset, or reciprocally, the data should be well-curated with respect to the considered image transformations that we apply during pretraining. For instance, random image cropping applied on images with multiple objects would not be appropriate, since there would be a risk that the representations of two different objects are learned to be similar in the latent space. To address this issue, recent work proposed various strategies to correctly handle image cropping in complex scenes with multiple objects [301,361]. Overall, future work could further explore how to better design image transformations adapted to images from real world settings, in order to enhance self-supervised learning methods on large-scale uncurated dataset.

Tuning the coefficients of pretraining losses without supervision. The pretraining objective functions discussed in Chapter 5 (SimCLR [50], AUH [354],

VICReg [15] and the proposed method SFRIK) are parameterized by certain coefficients, which need to be tuned carefully as hyperparameters in order to yield high-quality learned representations. In the experiments of Chapter 5, we follow the main practice of the literature where the best coefficients for each pretraining loss are chosen by evaluation on a *labeled* validation set, by kNN-classification or linear probing for instance. In this case, the pipeline is not completely unsupervised because it needs labels for hyperparameter tuning. In order to further avoid the dependence on data annotations, which can be costly at large-scale as discussed in Chapter 1, future work should compare the performance of different self-supervised methods by tuning their respective hyperparameters *in an unsupervised manner*, using for instance the rank criterion proposed in [119].

Reducing the computational costs of other pretraining losses. As discussed in Section 5.5, on the perspectives of the introduced kernel framework that unifies various regularization losses in self-supervision, future work could use kernel approximation techniques to reduce the time and memory complexity to compute the objective function, especially when the batch size is large. This approach could also be applied to other pretraining losses in the literature, such as the proposed sigmoid loss in language-image pretraining from [377]. This loss takes the form of a double sum $\sum_{i\in I}\sum_{i'\in I}\mathcal{K}_{t,b}(\mathbf{x}_i,\mathbf{y}_j)$ for a certain kernel function $\mathcal{K}_{t,b}$ parameterized by some scalar parameters t, b, and a batch of text and images embeddings $\{(\mathbf{x}_i,\mathbf{y}_i)\}_{i\in I}$. The complexity for computing directly this double sum is quadratic with respect to the batch size $|I|$, which is prohibitive at large batch sizes like $|I| = 32768$ as in [377]. Note that [377] computes the double sum by chunks, which allows them to consider large batch size, but still at the cost of necessitating large computational resources (SigLIP is trained during 5 days with 32 TPUv4 chips). In this context, one might wonder if it is possible to approximate this loss function, using some kernel approximation techniques, to allow a linear computational complexity with respect to the batch size.

10.3 Perspectives for butterfly factorization

Chapter 1 motivated our study on butterfly factorization for its potential impact in deep neural network compression. Based on the insights of this thesis, we now provide some partial answers to the question: to what extent can we compress deep neural networks via butterfly factorization to achieve good performance-efficiency trade-off, compared to other techniques for parameter reduction?

10.3.1 Performance of butterfly sparse neural networks trained from scratch

Comparing previous works to important baselines. As mentioned in Section 1.3.4, previous works that propose to train a butterfly sparse neural network from

Table 10.1: Results of our reproduction of butterfly sparse neural networks of the literature. The accuracies correspond to the ImageNet validation set, after training on the ImageNet training set using the experiment protocol of the original papers [72, 241]. The time of a single forward pass is measured for a batch size of 128, in *half-precision*. We use the implementation provided in each paper. We provide a new low-rank baseline that was lacking in previous studies.

	Parameters	Accuracy Top-1 / Top-5	Forward pass (ms)
Simple ViT-S/16 [25]			
Dense	21.9 M	75.5/92.0	14.2
Monarch [72]	9.6 M	73.2/91.1	23.3
Low-rank baseline (new)	9.6 M	73.6/91.2	13.0
Simple ViT-B/16 [25]			
Dense	86.4 M	75.9/91.7	36
Monarch [72]	36.8 M	75.4/91.7	49.3
Low-rank baseline (new)	36.8 M	75.1/91.8	27.1

scratch [72, 241] did not compare the obtained performance to some important baselines, such as using a low-rank parameterization of the weight matrices instead of a butterfly parameterization. Therefore, in Table 10.1, we reproduce the experiment from [72] and add a comparison to this low-rank baseline. We observe that, on ImageNet classification with a vision transformer [87], the low-rank baseline achieves a higher top-1 and top-5 accuracy than the butterfly sparse neural network with the Monarch architecture [72] for the ViT-S/16, while they have the same number of parameters. Similarly, it achieves a higher top-5 accuracy for the ViT-B/16. Meanwhile, using the implementation for butterfly sparse multiplication from [72], the forward pass in half-precision of a butterfly sparse neural is *slower* than the low-rank baseline *and the dense network*. In conclusion, we observe that the current performance-efficiency achieved by a butterfly sparse neural network trained from scratch as proposed in [72] is not satisfying.

Exploring various butterfly architectures and permutations. What are the hopes to obtain a butterfly sparse neural network trained from scratch with better accuracy than the low-rank baseline?

First, thanks to the framework of Chapter 7, we now know that there exist *many choices of chainable architectures* to compress a given weight matrix, so there might exist one that yields better accuracy than the butterfly architectures previously used in the literature. But without a specific prior knowledge about the considered learning task, selecting an architecture yielding good accuracy among all the possible choices is not straightforward. One could use neural architecture

search techniques to explore the different trade-offs of various butterfly architectures, but this technique can be costly in terms of computational resources.

Second, in previous works [72], weight matrices are parameterized by a butterfly factorization $\mathbf{X}_1 \dots \mathbf{X}_L$ associated to a certain architecture, *with fixed row and column permutations* during training. A more flexible approach is to explore different row and column permutations during training, by considering a parameterization $\mathbf{Q}^\top \mathbf{X}_1 \dots \mathbf{X}_L \mathbf{P}$ with permutations $\mathbf{P}, \mathbf{Q}$ part of the training parameters. But in general, optimizing an objective function with respect to some permutations is difficult, because of the discrete nature of the set of permutations. Note that the exploration of different permutations in a butterfly sparse neural network during training would be analogous to the exploration of different sparse supports in dynamic pruning methods[1] [268].

In conclusion, improving the performance of a butterfly sparse neural network trained from scratch might necessitate the exploration of various butterfly architectures and permutations. Unless there is a computationally efficient way to do this exploration, the alternative way is to first train a dense network, and then decompose certain weight matrices via butterfly factorization.

10.3.2 Decomposition of pretrained dense weight matrices

What are the chances that the pretrained weight matrices in a neural network can be accurately approximated by a product of butterfly factors?

Failure of previous decomposition models. Previous methods [72,241] decompose a pretrained weight matrix $\mathbf{W}$ by considering the problem $\min_{(\mathbf{X}_\ell)_{\ell=1}^L} \|\mathbf{W} - \mathbf{X}_1 \dots \mathbf{X}_L\|_F$, where $\mathbf{X}_1, \dots, \mathbf{X}_L$ are associated with a certain butterfly architecture. However, they did not compare the obtained approximation error to a low-rank baseline. Therefore, we perform such a comparison for the first time in Table 10.2. We follow the protocol from [72] where we decompose the weight matrices at several layers of a ViT-B/16, pretrained on ImageNet-1k in a supervised manner, via butterfly factorization associated with a Monarch architecture. We observe that the obtained approximation error is larger than the best low-rank decomposition, for a rank that yields the same compression rate in terms of number of parameters after decomposition. This means that the decomposition model from [72] is worse than the low-rank baseline. One possible explanation for this observation is that permutation symmetries of neural networks[2] were not taken into account

[1] In dynamic pruning methods, during training, some weights of the neural network can be fixed to zero following a certain pruning criterion during a certain number of training iterations, and reintroduced later as a trainable parameter if they become relevant according to a certain regrowing criterion.

[2] Permutation symmetries refer to a reshuffling of the neurons in the same layer of a network. For instance, considering a multilayer perceptron without bias, if $\mathbf{W}_\ell$ and $\mathbf{W}_{\ell+1}$ are the weight matrices of two consecutive layers, then setting $\mathbf{W}_\ell \leftarrow \mathbf{P}\mathbf{W}_\ell$ and $\mathbf{W}_{\ell+1} \leftarrow \mathbf{W}_{\ell+1}\mathbf{P}^\top$ for any permutation matrix $\mathbf{P}$ does not change the function realized by the network.

Table 10.2: Relative approximation error of weight matrices after decomposition via butterfly factorization, for a ViT-B/16 pretrained on ImageNet-1k in a supervised manner. For a weight matrix $\mathbf{W}$ of size $m \times n$, we consider the problem $\min_{\mathbf{X}_1,\mathbf{X}_2} \|\mathbf{W} - \mathbf{X}_1\mathbf{X}_2\|_F$ where $\mathbf{X}_1, \mathbf{X}_2$ are butterfly factors associated with a chainable architecture β. For Monarch butterfly factorization [72], the architecture is $\beta = ((1, m/4, \min(m,n)/4, 4), (4, \min(m,n), n/4, 1))$. For the low-rank decomposition, the architecture is $\beta = ((1, m, r, 1), (1, r, n, 1)$. We choose $r = \min(m,n)/4$ in order to have the same number of parameters for both architectures, which is $(m+n)\min(m,n)/4$. The relative error is the approximation error (in Frobenius norm) obtained by the hierarchical algorithm from Chapter 7, divided by $\|\mathbf{W}\|_F$. The considered weight matrices are those of the feed-forward network (FFN) module and the multi-head attention module, at the first and the last transformer block.

	Relative approximation error	
	Monarch [72]	Low-rank
First transformer block		
First layer of FFN	0.564	**0.413**
Second layer of FFN	0.529	**0.407**
Query projection	0.036	**0.003**
Key projection	0.039	**0.003**
Value projection	0.506	**0.364**
Last transformer block		
First layer of FFN	0.677	**0.626**
Second layer of FFN	0.670	**0.597**
Query projection	0.422	**0.293**
Key projection	0.446	**0.297**
Value projection	0.587	**0.548**

in the problem formulation for butterfly factorization, as we now detail.

Considering permutation symmetries in neural networks. According to the analytical characterization of butterfly matrices provided in Chapter 7, a matrix $\mathbf{A}$ can be written as a product of butterfly factors associated with a chainable architecture β, i.e., $\mathbf{A} \in \mathcal{B}^\beta$ (Definition 7.5), if and only if, it satisfies the generalized complementary low-rank property (Definition 7.8) associated with β. But by definition, this complementary low-rank property is *not stable by permutation of rows and columns*, in the sense that, in general, for a matrix $\mathbf{A}$ satisfying such a complementary low-rank property, the permuted matrix $\tilde{\mathbf{A}} := \tilde{\mathbf{Q}}^\top \mathbf{A} \tilde{\mathbf{P}}$ for some arbitrary permutation matrices $\tilde{\mathbf{P}}, \tilde{\mathbf{Q}}$ does not necessarily satisfy the same complementary low-rank property. Therefore, due to permutation symmetries in the parameterization of a neural network, in order to well approximate a weight ma-

trix $\mathbf{W}$ at a certain layer by a product of butterfly factors associated with β, it is necessary to consider the problem $\min_{(\mathbf{X}_\ell)_{\ell=1}^L, \mathbf{P}, \mathbf{Q}} \|\mathbf{W} - \mathbf{Q}^\top \mathbf{X}_1 \dots \mathbf{X}_L \mathbf{P}\|_F$ with unknown permutations $\mathbf{P}, \mathbf{Q}$. Indeed, as opposed to $\min_{(\mathbf{X}_\ell)_{\ell=1}^L} \|\mathbf{W} - \mathbf{X}_1 \dots \mathbf{X}_L\|_F$, the value for $\min_{(\mathbf{X}_\ell)_{\ell=1}^L, \mathbf{P}, \mathbf{Q}} \|\mathbf{W} - \mathbf{Q}^\top \mathbf{X}_1 \dots \mathbf{X}_L \mathbf{P}\|_F$ does not change if we replace $\mathbf{W}$ by $\tilde{\mathbf{W}} := \tilde{\mathbf{Q}}^\top \mathbf{W} \tilde{\mathbf{P}}$ for arbitrary permutation matrices $\tilde{\mathbf{P}}, \tilde{\mathbf{Q}}$. This ensures that the problem formulation for the decomposition does not depend on a specific parameterization of the neural network, and is the same for any permutation-equivalent parameterizations.

To the best of our knowledge, the work in Chapter 8 provides the first method to empirically address the butterfly factorization problem with unknown permutations, *without any analytical assumption* on the entries of the target matrix, which is typically the scenario when decomposing weight matrices of a given pretrained neural network. However, the current limitations of the method (lack of robustness to noise, issue with time complexity) makes it challenging to apply to weight matrices in neural networks at the moment. Therefore, a better understanding of the factorization problem with unknown permutations is necessary for further consideration of its applications to neural network compression, and some research directions were proposed in Section 8.7.

Decomposition restricted to some calibration data. An alternative problem formulation is to restrict the decomposition problem to a specific batch of input vectors. Given a target matrix $\mathbf{A}$ and an input matrix $\mathbf{Y}$, the problem is to minimize $\|\mathbf{AY} - \mathbf{Q}^\top \mathbf{X}_1 \dots \mathbf{X}_L \mathbf{PY}\|_F$, where $\mathbf{X}_1, \dots, \mathbf{X}_L$ are butterfly factors associated with an architecture β, and $\mathbf{P}, \mathbf{Q}$. Such an approach has been typically considered in other post-training compression methods, such as quantization methods [179,227,278], pruning methods [111,112,178] or low-rank compression methods [49].

10.3.3 Time-efficiency of butterfly sparse matrix multiplication

One of the main interests of the butterfly sparsity is its *structure* that takes the form of a Kronecker product, which can potentially lead to efficient implementations for real-time acceleration of the matrix multiplication. To what extent is it possible to obtain a competitive implementation for matrix multiplication using the butterfly structure compared, for instance, to the low-rank structure or the dense implementation?

Gap with the dense implementation in half-precision on GPUs. It is important to note that general matrix multiply (GEMM) operations in half-precision are highly optimized on modern GPUs, such as NVIDIA A100 GPUs with TensorCores. This makes the implementation of the matrix multiplication by a dense matrix or a low-rank matrix highly competitive, since a low-rank matrix can be

decomposed into two smaller dense matrices. Currently, the matrix multiplication by a butterfly matrix is, in general, *slower* than the multiplication by a low-rank matrix with as many parameters in half-precision, as suggested in Table 10.1, so there is a margin for improvement. In order to reduce this gap, our work in Chapter 9 proposed a new implementation specific to the butterfly sparsity that avoids costly memory transfer involved in the implementation from [72]. We showed speedups upon previous implementations in float-precision, but not always in half-precision. Nevertheless, there might exist some opportunities for further improvements of the proposed implementation in half-precision, as discussed in Section 9.7.

Implementation on specialized hardwares. In a frugal setting with limited computational budget, one could be interested in the implementation of the butterfly sparse matrix multiplication on other more specialized hardware, such as FPGAs or IPUs. We expect the implementation on such hardware to be efficient in terms of computation time and energy consumption, as suggested by recent works [104, 321] that implemented butterfly sparse matrix multiplication for some specific butterfly architectures. For more flexibility, future work could extend these implementations in order to consider arbitrary sparsity patterns $\mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d$ for general $(a, b, c, d) \in \mathbb{N}^4$, and compare them with the provided benchmark for GPU implementations in Chapter 9.

10.3.4 Summary

To summarize, by revisiting some previous works [72, 241], we show for the first time that the current performance-efficiency trade-off of butterfly sparse neural networks achieved so far are not satisfying, notably by comparison to the low-rank baseline. Nevertheless, the contributions of the thesis offer some new perspectives for improving this trade-off, for instance by taking into account unknown permutations for butterfly factorization.

Butterfly factorization for other applications. As suggested in Section 8.7, the decomposition algorithms for butterfly factorization developed in Chapters 6 to 8 could be applied to other applications than deep learning, like the construction of fast algorithms for the graph Fourier transform [215, 217, 315, 322]. Indeed, the graph Fourier transform can be seen as an extension of the DFT, in the sense that the DFT is the graph Fourier transform associated with a specific graph adjacency matrix. But as shown in Section 3.4, the DFT matrix satisfies the complementary low-rank property that allows a decomposition via butterfly factorization. Therefore, it is natural to ask whether the graph Fourier transform also satisfies a form of complementary low-rank property, so that it can be well decomposed via butterfly factorization, in order to accelerate operations in graph signal processing.

Part III

Appendices

Appendix A

Appendices for Chapter 5

A.1 Extended related work

We further discuss some related works referenced in Chapter 5.

A.1.1 Reminders on kernel mean embeddings

The idea of *kernel mean embedding* is to encode a probability distribution in an RKHS $\mathcal{H}$. Denoting $\mathcal{K} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ the reproducing kernel of $\mathcal{H}$ defined on some space $\mathcal{X}$, the kernel mean embedding of a probability distribution $\mathbb{Q}$ defined on $\mathcal{X}$ is

$$\mu_{\mathbb{Q}} := \int_{\mathcal{X}} \mathcal{K}(\mathbf{u}, \cdot) d\mathbb{Q}(\mathbf{u}) \in \mathcal{H}. \tag{A.1}$$

In other words, the kernel mean embedding mapping $\mathbb{Q} \mapsto \mu_{\mathbb{Q}}$ transforms a probability distribution into an element in $\mathcal{H}$. As an application, this allows one to quantify the divergence between probabilities using the norm $\|\cdot\|_{\mathcal{H}}$ associated with $\mathcal{H}$. Given two probability distributions $\mathbb{Q}_1, \mathbb{Q}_2$ defined on $\mathcal{X}$, one can indeed quantify their divergence by

$$\|\mu_{\mathbb{Q}_1} - \mu_{\mathbb{Q}_2}\|_{\mathcal{H}} = \left\| \int_{\mathcal{X}} \mathcal{K}(\mathbf{u}, \cdot) d\mathbb{Q}_1(\mathbf{u}) - \int_{\mathcal{X}} \mathcal{K}(\mathbf{u}, \cdot) d\mathbb{Q}_2(\mathbf{u}) \right\|_{\mathcal{H}},$$

which is precisely the MMD between $\mathbb{Q}_1$ and $\mathbb{Q}_2$ defined in (5.4). We refer the reader to [274] for a complete survey on kernel mean embeddings.

A.1.2 Sample-contrastive criterion

Given a batch of embeddings $\mathbf{Z}_I := \{\mathbf{z}_i\}_{i \in I}$ (that are not necessarily normalized), the general *sample-constrative criterion* of [120] is defined by:

$$\ell_c(\mathbf{Z}_I) = \sum_{i,i' \in I,\, i \neq i'} (\mathbf{z}_i^\top \mathbf{z}_{i'})^2.$$

According to [120], this criterion is minimized in many contrastive learning methods like [156]. In the case where the embeddings are normalized, this sample-contrastive criterion can be derived from the proposed generic uniformity loss ℓ_u defined by (5.7) with the *quadratic kernel* $\mathcal{K}(\mathbf{u}, \mathbf{v}) = (\mathbf{u}^\top \mathbf{v})^2$ where $\mathbf{u}, \mathbf{v} \in \mathcal{S}^{q-1}$, as claimed in Section 5.1. Indeed, since $\|\mathbf{z}_i\|_2 = 1$ for all $i \in I$:

$$\ell_c(\mathbf{Z}_I) = \sum_{i,i' \in I} (\mathbf{z}_i^\top \mathbf{z}_{i'})^2 - \sum_{i \in I} (\mathbf{z}_i^\top \mathbf{z}_i)^2 = |I|^2 \ell_u(\mathbf{Z}_I) - |I|,$$

where $|I|$ is the batch size. Therefore, the sample-contrastive criterion in the normalized case is an estimator of the MMD associated with the quadratic kernel between the embedding distribution and the uniform distribution on the hypersphere.

A.1.3 Kernel dependence maximization

We further explain the positioning of our work with respect to [231], which proposes a self-supervised learning method based on *kernel dependence maximization*, using the Hilbert-Schmidt Independence Criterion (HSIC) [142].

Hilbert-Schmidt independence criterion. This measures the dependence between two random variables $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$ using two RKHS $\mathcal{F}$ on $\mathcal{X}$ with kernel k and $\mathcal{G}$ on $\mathcal{Y}$ with kernel l, in order to capture nonlinear correlations. It is defined as the squared MMD associated with the reproducing kernel of the tensor product space $\mathcal{F} \otimes \mathcal{G}$ between the joint probability distribution $\mathbb{P}_{X,Y}$ and the product $\mathbb{P}_X \mathbb{P}_Y$ of marginal probability distributions:

$$\mathrm{HSIC}(X, Y) := \|\mu_{\mathbb{P}_{X,Y}} - \mu_{\mathbb{P}_X \mathbb{P}_Y}\|^2_{\mathcal{F} \otimes \mathcal{G}},$$

where $\mathbb{Q} \mapsto \mu_{\mathbb{Q}}$ is the kernel mean embedding mapping defined by (A.1).

Kernel dependence maximization for self-supervision. The self-supervised learning loss in [231] is defined as:

$$\mathcal{L}_{\text{SSL-HSIC}} := -\mathrm{HSIC}(Z, Y) + \gamma \sqrt{\mathrm{HSIC}(Z, Z)},$$

where Z encodes embeddings of transformed images, and Y encodes image identity as the index of the original image (before transformation) in the training dataset. By maximizing $\mathrm{HSIC}(Z, Y)$, the backbone learns image representations that are invariant to image transformations. To avoid collapse, high-variance representations are penalized by minimizing $\mathrm{HSIC}(Z, Z)$. This is similar to previous information-maximization methods [15, 375], with the difference that they take into account nonlinear correlations using kernels.

Positioning. Although both our approach and the one in [231] view self-supervised learning as a *kernel method*, we highlight here a main distinction between the two works. Both approaches use the MMD, but they do not use it to measure the same quantity. As explained above, [231] use the MMD to measure dependency between random variables (like Z and Y), while the regularization loss we propose uses the MMD to measure the divergence between the embedding distribution and the uniform distribution on the hypersphere. As explained in Sections 5.1 and 5.2, this kernel approach for self-supervised learning is new in the literature and allows the unification of several previous self-supervised learning methods as illustrated in Table 5.1.

Note that when the image identity Y is a one-hot encoding, [231] shows that

$$-\mathrm{HSIC}(Z, Y) = C\left(-\mathbb{E}_{(Z_1,Z_2)\sim\mathrm{pos}}[k(Z_1, Z_2)] + \mathbb{E}_{Z_3}\mathbb{E}_{Z_4}[k(Z_3, Z_4)]\right), \tag{A.2}$$

where $C > 0$ is a constant, (Z_1, Z_2) is a positive pair of embeddings, i.e., they are embeddings of two transformations of the same original image, and (Z_3, Z_4) is a pair of independent embeddings. In other words, $\mathrm{HSIC}(Z, Y)$ is proportional to the sum of an alignment term $-\mathbb{E}_{(Z_1,Z_2)\sim\mathrm{pos}}[k(Z, Z')]$ and an energy term $\mathbb{E}_{Z_3}\mathbb{E}_{Z_4}[k(Z_3, Z_4)]$, similarly to (5.3) combined with (5.7), which yields the proposed loss (5.2). Our work shows that, if $k(\cdot, \cdot)$ is a rotation-invariant kernel on the hypersphere, then the energy term $\mathbb{E}_{Z_3}\mathbb{E}_{Z_4}[k(Z_3, Z_4)]$ is precisely the MMD between the embedding distribution and the uniform distribution on the hypersphere, cf. (5.6). However there are two differences between the maximization of $\mathrm{HSIC}(Z, Y)$ and the minimization of the proposed loss (5.2). First, the alignment term and the energy term in (A.2) are quantified with the *same* kernel $k(\cdot, \cdot)$, which is not the case in (5.2) where the alignment term is quantified by the ℓ_2-distance between embeddings (equivalent to the linear kernel when the embeddings are normalized), and the uniformity term (5.7) is quantified by another rotation-invariant kernel. Second, the loss (5.2) is a *weighted* sum between the alignment loss (5.3) and the uniformity loss (5.7) controlled by the hyperparameter λ that tunes the balance between the two terms, which is not the case of (A.2).

A.1.4 Regularization loss of SimCLR, AUH and VICReg

In SimCLR, AUH, and VICReg, each image $\mathbf{x}_i$ in a batch $\{\mathbf{x}_i\}_{i\in I}$ is augmented into two different views $\mathbf{x}_i^{(1)}$ and $\mathbf{x}_i^{(2)}$, which are encoded into two embeddings $\mathbf{z}_i^{(1)}$ and $\mathbf{z}_i^{(2)}$. These embeddings are normalized in SimCLR, AUH, but not in VICReg. This yields two batches of embeddings $\mathbf{Z}_I^{(v)} := \{\mathbf{z}_i^{(v)}\}_{i\in I}$ for $v \in [\![2]\!]$. The four methods share the same form of loss function:

$$\ell(\mathbf{Z}_I^{(1)}, \mathbf{Z}_I^{(2)}) := \lambda\, \ell_a(\mathbf{Z}_I^{(1)}, \mathbf{Z}_I^{(2)}) + \mu\, \ell_r(\mathbf{Z}_I^{(1)}, \mathbf{Z}_I^{(2)}), \tag{A.3}$$

for some scalars $\lambda, \mu > 0$, where ℓ_a is the *alignment loss* defined by (5.3) (which is the same for all the four methods), and ℓ_r is the *regularization loss* specific to each method.

SimCLR. The regularization loss in SimCLR is:

$$\ell_r(\mathbf{Z}_I^{(1)}, \mathbf{Z}_I^{(2)}) = \frac{1}{2|I|} \sum_{v=1}^{2} \sum_{i \in I} \log \left(\sum_{v'=1}^{2} \sum_{i' \in I} \mathbb{1}_{[(v,i) \neq (v',i')]} \exp \left({\mathbf{z}_i^{(v)}}^\top \mathbf{z}_{i'}^{(v')} / \tau \right) \right),$$

where $\tau > 0$ is a hyperparameter of the method called the temperature, and $\mathbb{1}_{[(v,i) \neq (v',i')]}$ is equal to 1 if $(v,i) \neq (v',i')$, and 0 otherwise. The scalars λ, μ are fixed at $\lambda = \frac{1}{\tau}$ and $\mu = 1$.

Alignment & Uniformity. The regularization loss in AUH is:

$$\ell_r(\mathbf{Z}_I^{(1)}, \mathbf{Z}_I^{(2)}) = \frac{1}{2|I|^2} \sum_{v=1}^{2} \log \left(\sum_{i \in I} \sum_{i' \in I} \exp(-t \|\mathbf{z}_i^{(v)} - \mathbf{z}_{i'}^{(v)}\|_2^2) \right),$$

where $t > 0$ is a hyperparameter called the scale of the RBF kernel. The scalar λ is tuned as a hyperparameter and μ is fixed at $\mu = 1$.

VICReg. As introduced in Section 3.3, the regularization loss in VICReg is:

$$\ell_r(\mathbf{Z}_I^{(1)}, \mathbf{Z}_I^{(2)}) = \frac{1}{2} \left[v(\mathbf{Z}_I^{(1)}) + v(\mathbf{Z}_I^{(2)}) \right] + \frac{1}{2\mu} \left[c(\mathbf{Z}_I^{(1)}) + c(\mathbf{Z}_I^{(2)}) \right],$$

where μ is the scalar from (A.3). Here, $v(\cdot)$ and $c(\cdot)$ are respectively the variance and covariance terms defined by (5.9). Both λ and μ are tuned as hyperparameters.

A.2 Theoretical results

We provide proofs and more details about the theoretical results in Chapter 5.

A.2.1 Proof of Lemma 5.1

Consider a rotation-invariant kernel $\mathcal{K}(\mathbf{u}, \mathbf{v})$ defined on the hypersphere $\mathcal{S}^{q-1}$ such that:

$$\mathcal{K}(\mathbf{u}, \mathbf{v}) = \sum_{\ell=0}^{+\infty} b_\ell P_\ell(q; \mathbf{u}^\top \mathbf{v}), \quad \forall \mathbf{u}, \mathbf{v} \in \mathcal{S}^{q-1},$$

with weights $b_\ell \geq 0$ and $P_\ell(q; \cdot)$ the *Legendre polynomial* of order ℓ in dimension q. The proof of Lemma 5.1 relies on an orthonormal system of *spherical harmonics*. Let $\langle f, g \rangle_{(q)} := \int_{\mathcal{S}^{q-1}} f g d\sigma_{q-1}$ be the inner product in the space of continuous functions defined on $\mathcal{S}^{q-1}$ and, for each $\ell \in \mathbb{N}$, consider $\{Y_{\ell,k} \mid k \in [\![N(q,\ell)]\!]\}$ an orthonormal basis of spherical harmonics of order ℓ in dimension q (homogeneous harmonic polynomials in q variables restricted to $\mathcal{S}^{q-1}$, see e.g. [277]

for more details), where $N(q,\ell)$ denotes the dimension of this space, which is by [277, Exercise 6, §3]:

$$N(q,\ell) = \begin{cases} q & \text{for } \ell = 1, \\ \frac{(2\ell+q-2)(\ell+q-3)!}{\ell!\,(q-2)!} & \text{for } \ell \geq 2. \end{cases} \tag{A.4}$$

By the addition theorem [277, Theorem 2, §1]:

$$\sum_{k=1}^{N(q,\ell)} Y_{\ell,k}(\mathbf{u}) Y_{\ell,k}(\mathbf{v}) = \frac{N(q,\ell)}{|\mathcal{S}^{q-1}|} P_\ell(q; \mathbf{u}^\top \mathbf{v}), \quad \mathbf{u},\mathbf{v} \in \mathcal{S}^{q-1}. \tag{A.5}$$

Hence, the kernel $\mathcal{K}(\mathbf{u},\mathbf{v})$ can be rewritten as:

$$\mathcal{K}(\mathbf{u},\mathbf{v}) = \sum_{\ell=0}^{+\infty} \sum_{k=1}^{N(q,\ell)} \frac{b_\ell |\mathcal{S}^{q-1}|}{N(q,\ell)} Y_{\ell,k}(\mathbf{u}) Y_{\ell,k}(\mathbf{v}).$$

Since $\{Y_{\ell,k} \mid \ell \in \mathbb{N}, k \in [\![N(q,\ell)]\!]\}$ is an orthonormal system for the inner product $\langle \cdot,\cdot \rangle_{(q)}$, and since $Y_{0,1}$ is constant on $\mathcal{S}^{q-1}$, we have for any integer ℓ and $k \in [\![N(q,\ell)]\!]$ that:

$$\int_{\mathcal{S}^{q-1}} Y_{\ell,k} d\sigma_{q-1} = \frac{1}{Y_{0,1}} \langle Y_{\ell,k}, Y_{0,1} \rangle_{(q)} = \begin{cases} \frac{1}{Y_{0,1}} & \text{if } \ell = 0, k = 1 \\ 0 & \text{otherwise} \end{cases}.$$

Moreover $Y_{0,1} = 1/\sqrt{|\mathcal{S}^{q-1}|}$, because $1 = \langle Y_{0,1}, Y_{0,1} \rangle_{(q)} = \int_{\mathcal{S}^{q-1}} Y_{0,1}^2 d\sigma_{q-1} = Y_{0,1}^2 |\mathcal{S}^{q-1}|$. Therefore, the kernel mean embedding of the uniform distribution on the hypersphere $\mathbb{U} := \sigma_{q-1}/|\mathcal{S}^{q-1}|$ associated with the kernel $\mathcal{K}$ is:

$$\begin{aligned} \forall \mathbf{v} \in \mathcal{S}^{q-1}, \int_{\mathcal{S}^{q-1}} \mathcal{K}(\mathbf{u},\mathbf{v}) d\mathbb{U}(\mathbf{u}) &= \int_{\mathcal{S}^{q-1}} \sum_{\ell=0}^{+\infty} b_\ell P_\ell(q; \mathbf{u}^\top \mathbf{v}) \frac{d\sigma_{q-1}(\mathbf{u})}{|\mathcal{S}^{q-1}|} \\ &= \sum_{\ell=0}^{+\infty} \int_{\mathcal{S}^{q-1}} \sum_{k=1}^{N(q,\ell)} \frac{b_\ell}{N(q,\ell)} Y_{\ell,k}(\mathbf{u}) Y_{\ell,k}(\mathbf{v}) d\sigma_{q-1}(\mathbf{u}) \\ &= \sum_{\ell=0}^{+\infty} \frac{b_\ell}{N(q,\ell)} \sum_{k=1}^{N(q,\ell)} \left[\int_{\mathcal{S}^{q-1}} Y_{\ell,k}(\mathbf{u}) d\sigma_{q-1}(\mathbf{u}) \right] Y_{\ell,k}(\mathbf{v}) \\ &= b_0 \frac{1}{Y_{1,0}} Y_{1,0} = b_0, \end{aligned}$$

where we inverted series and integral in the second equation using the dominated convergence theorem: the series $\sum_{\ell=0}^{+\infty} b_\ell P_\ell(q; \mathbf{u}^\top \mathbf{v})$ converges for every $\mathbf{u}$, $\mathbf{v}$, and for any L, $|\sum_{\ell=0}^{L} b_\ell P_\ell(q; \mathbf{u}^\top \mathbf{v})| \leq \sum_{\ell=0}^{L} |b_\ell P_\ell(q; \mathbf{u}^\top \mathbf{v})| \leq \sum_{\ell=0}^{+\infty} b_\ell = \sum_{\ell=0}^{+\infty} b_\ell P_\ell(q;1)$, because $|P_\ell(q;\cdot)| \leq 1$ for all ℓ by [277, Lemma 2, §8], $P_\ell(q;1) = 1$ for all ℓ by [277, §9], and $\sum_{\ell=0}^{+\infty} b_\ell P_\ell(q;1) < +\infty$ is integrable on $\mathcal{S}^{q-1}$. This yields the first claim of Lemma 5.1.

Consider now any probability distribution $\mathbb{Q}$ defined on the hypersphere. The kernel mean embedding of $\mathbb{Q}$ is simply rewritten as:

$$\begin{aligned}\forall \mathbf{v} \in \mathcal{S}^{q-1},\ & \int_{\mathcal{S}^{q-1}} \mathcal{K}(\mathbf{u},\mathbf{v}) d\mathbb{Q}(\mathbf{u}) \\ &= \int_{\mathcal{S}^{q-1}} \sum_{\ell=0}^{+\infty} b_\ell P_\ell(q; \mathbf{u}^\top \mathbf{v}) d\mathbb{Q}(\mathbf{u}) \\ &= b_0 \int_{\mathcal{S}^{q-1}} P_0(q; \mathbf{u}^\top \mathbf{v}) d\mathbb{Q}(\mathbf{u}) + \int_{\mathcal{S}^{q-1}} \sum_{\ell=1}^{+\infty} b_\ell P_\ell(q; \mathbf{u}^\top \mathbf{v}) d\mathbb{Q}(\mathbf{u}) \\ &= b_0 + \int_{\mathcal{S}^{q-1}} \tilde{\mathcal{K}}(\mathbf{u},\mathbf{v}) d\mathbb{Q}(\mathbf{u}),\end{aligned}$$

because the Legendre polynomial of order 0 is the constant function equal to 1 (see the closed form expression of $P_0(q;\cdot)$ in Theorem 5.1 in the main text) and $\int_{\mathcal{S}^{q-1}} d\mathbb{Q} = 1$. This ends the proof of Lemma 5.1.

A.2.2 Legendre expansion of rotation-invariant kernels

We show that the kernel weights b_ℓ in the Legendre expansion (5.5) of the RBF kernel and the generalized distance kernel decay with a rate at least polynomial with respect to ℓ.

RBF kernel. The RBF kernel is defined as:

$$\mathcal{K}_{\text{RBF}}(\mathbf{u},\mathbf{v}) = e^{-\sigma\|\mathbf{u}-\mathbf{v}\|_2^2} = e^{-2\sigma(1-\mathbf{u}^\top\mathbf{v})} \quad \text{for } \mathbf{u},\mathbf{v} \in \mathcal{S}^q,$$

where $\sigma > 0$ is the scale of the RBF kernel. Denote $\varphi(t) := e^{-2\sigma(1-t)}$ for $t \in [-1,1]$. Since the RBF kernel is positive definite and rotation-invariant, by Theorem 5.1, there exist weights $b_\ell \geq 0$, $\ell \in\in \mathbb{N}$, such that:

$$\varphi(t) = e^{-2\sigma(1-t)} = \sum_{\ell=0}^{+\infty} b_\ell P_\ell(q;t) \quad \text{for } t \in [-1,1]. \tag{A.6}$$

The Legendre polynomials $P_\ell(q;\cdot)$ are orthogonal on the interval $[-1,1]$ with respect to the weight function $(1-t^2)^{\frac{q-3}{2}}$, see e.g. [277]:

$$\int_{-1}^{1} P_n(q;t) P_m(q;t) (1-t^2)^{\frac{q-3}{2}} dt = 0 \quad \text{for } m \neq n.$$

Moreover, by [277, Exercise 3, §2]:

$$\int_{-1}^{1} (P_n(q;t))^2 (1-t^2)^{\frac{q-3}{2}} dt = \frac{|\mathcal{S}^{q-1}|}{|\mathcal{S}^{q-2}|} \frac{1}{N(q,n)} \quad \text{for any } n.$$

We multiply (A.6) by $P_\ell(q;t)(1-t^2)^{\frac{q-3}{2}}$ and integrate the equation on $[-1,1]$:

$$\begin{aligned}\int_{-1}^{1}\varphi(t)P_\ell(q;t)(1-t^2)^{\frac{q-3}{2}}dt &= \int_{-1}^{1}\sum_{n=0}^{+\infty}b_nP_n(q;t)P_\ell(q;t)(1-t^2)^{\frac{q-3}{2}}dt\\ &= \sum_{n=0}^{+\infty}b_n\int_{-1}^{1}P_n(q;t)P_\ell(q;t)(1-t^2)^{\frac{q-3}{2}}dt\\ &= b_\ell\frac{|\mathcal{S}^{q-1}|}{|\mathcal{S}^{q-2}|}\frac{1}{N(q,\ell)},\end{aligned}$$

where the inversion between series and integral is justified by the dominated convergence theorem: the series $\sum_{n=0}^{+\infty}b_nP_n(q;t)P_\ell(q;t)(1-t^2)^{\frac{q-3}{2}}$ converges for every t, and for any N, $|\sum_{n=0}^{N}b_nP_n(q;t)P_\ell(q;t)(1-t^2)^{\frac{q-3}{2}}| \le \sum_{n=0}^{+\infty}b_n(1-t^2)^{\frac{q-3}{2}} := g(t)$ since $|P_n(q;\cdot)| \le 1$ for any n by [277, Lemma 2, §8], and g is integrable on $[-1,1]$ because $\sum_{n=0}^{+\infty}b_n < +\infty$. Hence:

$$b_\ell = N(q,\ell)\frac{|\mathcal{S}^{q-2}|}{|\mathcal{S}^{q-1}|}\int_{-1}^{1}\varphi(t)P_\ell(q;t)(1-t^2)^{\frac{q-3}{2}}dt.$$

By the Rodrigues rule [277, Exercise 1, §2], since φ has continuous derivatives of all orders on $[-1,1]$, we have:

$$b_\ell = N(q,\ell)\frac{|\mathcal{S}^{q-2}|}{|\mathcal{S}^{q-1}|}\frac{\Gamma(\frac{q-1}{2})}{2^\ell\Gamma(\ell+\frac{q-1}{2})}\int_{-1}^{1}\varphi^{(\ell)}(t)(1-t^2)^{\ell+\frac{q-3}{2}}dt, \quad \ell\in\mathbb{N},$$

where $\varphi^{(\ell)}$ is the ℓ-th derivative of φ, which is $\varphi^{(\ell)}(t) = e^{-2\sigma}(2\sigma)^\ell e^{2\sigma t}$. We now show that the weights b_ℓ decay very fast with respect to ℓ. We bound the integral:

$$\int_{-1}^{1}\varphi^{(\ell)}(t)(1-t^2)^{\ell+\frac{q-3}{2}}dt = \int_{-1}^{1}e^{-2\sigma}(2\sigma)^\ell e^{2\sigma t}(1-t^2)^{\ell+\frac{q-3}{2}}dt \le \int_{-1}^{1}(2\sigma)^\ell dt = 2(2\sigma)^\ell.$$

Hence:

$$b_\ell \le 2N(q,\ell)\frac{|\mathcal{S}^{q-2}|}{|\mathcal{S}^{q-1}|}\frac{\Gamma(\frac{q-1}{2})}{2^\ell\Gamma(\ell+\frac{q-1}{2})}(2\sigma)^\ell.$$

Denote $(a)_n := \Gamma(n+a)/\Gamma(a)$ the Pochhammer symbol defined for any integer n and any scalar a. By the Stirling approximation of the Gamma function [327, (25.15)], the asymptotic behavior of $(a)_n$ when n goes to infinity is:

$$(a)_n \sim \frac{\sqrt{2\pi}}{\Gamma(a)}e^{-n}n^{a+n-1/2} \quad \text{as } n\to\infty.$$

Moreover, for a fixed dimension q, the asymptotic behavior of $N(q,\ell)$ defined by (A.4) when ℓ goes to infinity is:

$$N(q,\ell) \sim \frac{2}{(q-2)!}\ell^{q-2} \quad \text{as } \ell\to\infty.$$

Therefore, the asymptotic behavior of b_ℓ as ℓ goes to infinity is:

$$b_\ell = \mathcal{O}\left(\sigma^\ell e^\ell \ell^{q/2-1-\ell}\right) \quad \text{as } \ell\to+\infty.$$

Generalized distance kernel. For $\frac{q-1}{2} < s < \frac{q+1}{2}$, the generalized distance kernel on the hypersphere $\mathcal{S}^{q-1}$ is defined in [36, Section 5] as:

$$\mathcal{K}_{\mathrm{gd}}^{(s)}(\mathbf{u},\mathbf{v}) := 2V_{q-1-2s}(\mathcal{S}^{q-1}) - \|\mathbf{u}-\mathbf{v}\|_2^{2s-q+1} \quad \text{for } \mathbf{u},\mathbf{v} \in \mathcal{S}^{q-1},$$

where

$$\begin{aligned} V_{q-1-2s}(\mathcal{S}^{q-1}) :&= \int_{\mathcal{S}^{q-1}} \int_{\mathcal{S}^{q-1}} \|\mathbf{u}-\mathbf{v}\|_2^{2s-q+1} d\sigma_{q-1}(\mathbf{u}) d\sigma_{q-1}(\mathbf{v}) \\ &= 2^{2s-1} \frac{\Gamma(q/2)\Gamma(s)}{\sqrt{\pi}\Gamma((q-1)/2+s)}. \end{aligned}$$

Following [36, Section 5], the Legendre expansion of the generalized distance kernel $\mathcal{K}_{\mathrm{gd}}^{(s)}$ is:

$$\mathcal{K}_{\mathrm{gd}}^{(s)}(\mathbf{u},\mathbf{v}) = V_{q-1-2s}(\mathcal{S}^{q-1}) + \sum_{\ell=1}^{+\infty} \alpha_\ell^{(s)} N(q,\ell) P_\ell(q; \mathbf{u}^\top \mathbf{v}), \tag{A.7}$$

$$\alpha_\ell^{(s)} := -V_{q-1-2s}(\mathcal{S}^{q-1}) \frac{((q-1)/2-s)_\ell}{((q-1)/2+s)_\ell}, \quad \ell \geq 1. \tag{A.8}$$

The kernel weights indeed decay polynomially with respect to ℓ, because according to [36, Section 5], the asymptotic behavior of the $\alpha_\ell^{(s)}$ is:

$$\alpha_\ell^{(s)} \sim 2^{2s-1} \frac{\Gamma(q/2)\Gamma(s)}{\sqrt{\pi}\Gamma((q-1)/2-s)} \ell^{-2s} \quad \text{as } \ell \to +\infty.$$

A.2.3 Proof of Proposition 5.1

Consider a rotation-invariant kernel $\tilde{\mathcal{K}}(\mathbf{u},\mathbf{v}) := \sum_{\ell=1}^{+\infty} b_\ell P_\ell(q; \mathbf{u}^\top \mathbf{v})$ defined on $\mathcal{S}^{q-1}$ such that $b_\ell \geq 0$ for $\ell \in \mathbb{N}^*$, with $b_1, b_2 > 0$. To show the connection between SFRIK and VICReg, we construct a high-dimensional *feature map* $\Phi : \mathcal{S}^{q-1} \to \ell_2(\mathbb{N})$, where $\ell_2(\mathbb{N})$ denotes the space of square-summable sequences with its canonical inner product $\langle \cdot, \cdot \rangle_{\ell_2}$, such that $\langle \Phi(\mathbf{u}), \Phi(\mathbf{v}) \rangle_{\ell_2} = \tilde{\mathcal{K}}(\mathbf{u},\mathbf{v})$ for any $\mathbf{u},\mathbf{v} \in \mathcal{S}^{q-1}$.

One way to construct such a feature map is to consider an orthonormal system of *spherical harmonics*. For any integer ℓ, denote $\{Y_{\ell,k}\}_{\ell=1}^{N(q,\ell)}$ an orthonormal basis of spherical harmonics of order ℓ in dimension q. By the addition theorem [277, Theorem 2, §1], cf. (A.5), the kernel $\tilde{\mathcal{K}}(\mathbf{u},\mathbf{v})$ admits the decomposition:

$$\begin{aligned} \tilde{\mathcal{K}}(\mathbf{u},\mathbf{v}) = \sum_{\ell=1}^{+\infty} b_\ell P_\ell(q; \mathbf{u}^\top \mathbf{v}) &= \sum_{\ell=1}^{+\infty} \sum_{k=1}^{N(q,\ell)} \frac{b_\ell |\mathcal{S}^{q-1}|}{N(q,\ell)} Y_{\ell,k}(\mathbf{u}) Y_{\ell,k}(\mathbf{v}) \\ &= \langle \Phi(\mathbf{u}), \Phi(\mathbf{v}) \rangle_{\ell_2}, \end{aligned}$$

where

$$\Phi := \left(\sqrt{\frac{b_\ell |\mathcal{S}^{q-1}|}{N(q,\ell)}} \Phi_\ell \right)_{\ell=1}^{+\infty} \quad \text{with} \quad \Phi_\ell : \begin{cases} \mathcal{S}^{q-1} \to \mathbb{R}^{N(q,\ell)} \\ \mathbf{u} \mapsto (Y_{\ell,k}(\mathbf{u}))_{k=1}^{N(q,\ell)} \end{cases} \quad \forall \ell \in \mathbb{N}^*.$$

Then, the MMD in (5.6) between any probability distribution $\mathbb{Q}$ defined on the hypersphere and the uniform distribution $\mathbb{U}$ on the hypersphere can be written as the norm $\|\cdot\|_2$ in $\ell_2(\mathbb{N})$ of the generalized moment of $\mathbb{Q}$ with the mapping Φ:

$$\begin{aligned} \mathrm{MMD}^2(\mathbb{Q},\mathbb{U}) &= \mathbb{E}_{\mathbf{z},\mathbf{z}'\sim\mathbb{Q}}[\tilde{\mathcal{K}}(\mathbf{z},\mathbf{z}')] \\ &= \mathbb{E}_{\mathbf{z},\mathbf{z}'\sim\mathbb{Q}}[\langle \Phi(\mathbf{z}), \Phi(\mathbf{z}') \rangle_{\ell_2}] \\ &= \left\langle \mathbb{E}_{\mathbf{z}\sim\mathbb{Q}}[\Phi(\mathbf{z})], \mathbb{E}_{\mathbf{z}'\sim\mathbb{Q}}[\Phi(\mathbf{z}')] \right\rangle_{\ell_2} = \|\mathbb{E}_{\mathbf{z}\sim\mathbb{Q}}\left[\Phi(\mathbf{z})\right]\|_{\ell_2}^2 \\ &= \sum_{\ell=1}^{+\infty} \frac{b_\ell |\mathcal{S}^{q-1}|}{N(q,\ell)} \|\mathbb{E}_{\mathbf{z}\sim\mathbb{Q}}\left[\Phi_\ell(\mathbf{z})\right]\|_2^2 . \end{aligned} \tag{A.9}$$

We now explain how to construct explicitly an orthonormal basis of spherical harmonics $\{Y_{\ell,k} \mid \ell \in \mathbb{N}^*,\ k \in [\![N(q,\ell)]\!]\}$, based on the following theorem.

Theorem A.1 (from [11, Theorem 5.25]). *For any order $\ell \in \mathbb{N}$ and any dimension $q \geq 3$, the family*

$$\{Y'_{\ell,k}\}_{k=1}^{N(q,\ell)} := \left\{ \mathbf{u} \mapsto \partial_1^{\alpha_1}\partial_2^{\alpha_2}\ldots\partial_q^{\alpha_q} \|\mathbf{u}\|_2^{2-q} \mid \alpha_1+\alpha_2+\ldots+\alpha_q = \ell \text{ and } \alpha_1 \leq 1 \right\} \tag{A.10}$$

is a (non-orthonormal) basis of the space of spherical harmonics of order ℓ in dimension q, where α_j ($j \in [\![q]\!]$) are nonnegative integers, and $\partial_j^{\alpha_j}$ denotes the α_j-th partial derivative with respect to the j-th coordinate.

Typically, we construct the orthonormal basis $\{Y_{\ell,k}\}_{k=1}^{N(q,\ell)}$ by orthonormalizing the basis $\{Y'_{\ell,k}\}_{k=1}^{N(q,\ell)}$ of Theorem A.1 using, e.g., the *Gram-Schmidt* procedure. For $\ell \in \mathbb{N}^*$, denote:

$$\Phi'_\ell :\ \mathcal{S}^{q-1} \to \mathbb{R}^{N(q,\ell)},\ \mathbf{u} \mapsto (Y'_{\ell,k}(\mathbf{u}))_{k=1}^{N(q,\ell)}.$$

Then, for each $\ell \in \mathbb{N}^*$, there exists a lower triangular matrix $\mathbf{M}_\ell$ such that:

$$\Phi_\ell(\mathbf{u}) = \mathbf{M}_\ell \Phi'_\ell(\mathbf{u}), \quad \text{for all } \mathbf{u} \in \mathcal{S}^{q-1}. \tag{A.11}$$

Remark that it is possible to compute explicitly the entries of the matrices $\mathbf{M}_\ell$ for $\ell \in \mathbb{N}^*$, because there exists a closed-form expression for the inner product $\langle Y'_{\ell,k}, Y'_{\ell,k'} \rangle_{(q)}$ for any ℓ, k, k': indeed, the function $Y'_{\ell,k}$ for any ℓ, k is a polynomial defined on the hypersphere, and the integral of any monomial with respect to the measure σ_{q-1} on the hypersphere $\mathcal{S}^{q-1}$ admits a closed-form expression given by [356, Section 3].

By injecting (A.11) in (A.9), we obtain:

$$\mathrm{MMD}^2(\mathbb{Q}, \mathbb{U}) = \sum_{\ell=1}^{+\infty} \frac{b_\ell |\mathcal{S}^{q-1}|}{N(q,\ell)} \left\| \mathbf{M}_\ell \mathbb{E}_{\mathbf{z}\sim\mathbb{Q}} \left[\Phi'_\ell(\mathbf{z}) \right] \right\|_2^2 .$$

This yields the claim of Proposition 5.1 by remarking with Theorem A.1 that the families

$$\{Y'_{1,k}\}_{k=1}^{N(q,1)} = \{\mathbf{u} \mapsto \mathbf{u}[j] \mid j \in [\![q]\!]\},$$
$$\{Y'_{2,k}\}_{k=1}^{N(q,2)} = \{\mathbf{u} \mapsto \mathbf{u}[j]\mathbf{u}[j'] \mid 1 \le j < j' \le q\} \cup \left\{ \mathbf{u} \mapsto (\mathbf{u}[j])^2 - \frac{1}{q} \mid j \in [\![2, q]\!] \right\}$$

are bases of the space of spherical harmonics of order 1 and 2 in dimension q.

A.3 Experimental setting

In the interest of reproducible research, we give more details about the setting of our experiments presented in Section 5.4.

A.3.1 IN20% dataset description

The datasets used in our experiments include a subset of 20% of ImageNet-1k as in [124]. This reduced dataset, denoted IN20%, contains all the 1000 classes of ImageNet-1k, but we keep only 260 images per class. The 260 images extracted are the same as those extracted in the official implementation of OBoW (`https://github.com/valeoai/obow`). In Section 5.4.2, we also use *another* 20% subset of the ImageNet-1k train set as a separate validation set for hyperparameter tuning (see Appendix A.3.4 below). The construction of this validation set is based on the code of OBoW.

A.3.2 Image augmentations

We follow the same image augmentation pipeline as in [15]. Our experiments include the following image augmentations implemented by PyTorch using the methods from `torchvision.transforms`:

- `RandomResizedCrop(224, scale=(0.08, 1.0))`: crop a random area of the image between 8% and 100% of the total area, and resize it to an image of size 224×224;
- `RandomHorizontalFlip()`: flip horizontally an image;
- `ColorJitter(brightness=0.4, contrast=0.4, saturation=0.2, hue=0.1)`: randomly change brightness, contrast, saturation and hue of an image by a factor randomly sampled in respectively $[0.6, 1.4]$, $[0.6, 1.4]$, $[0.8, 1.2]$ and $[-0.1, 0.1]$.

- `RandomGrayscale()`: convert an image into grayscale.

We also use image augmentations implemented by PIL, as in VICReg's code available at https://github.com/facebookresearch/vicreg:

- `GaussianBlur()`: blur an image using a Gaussian kernel with standard deviation uniformly sampled in $[0.1, 2.0]$;
- `Solarization()`: invert all pixel values above a threshold, which is 130.

In our experiments, the first image view is obtained by composing the following random augmentations: random cropping resized to 224×224, random horizontal flip applied with probability 0.5, random color jittering applied with probability 0.8, random grayscale conversion applied with probability 0.2, random Gaussian blur applied with probability 0.1, and random solarization applied with probability 0.2. The second view is obtained by composing the same random augmentations as the first view, except that Gaussian blur is applied every time (probability 1), and solarization is never applied (probability 0).

A.3.3 Evaluation protocol

We describe the *downstream tasks* on which self-supervision methods are evaluated in our experiments of Section 5.4.

Linear probing on IN20% and IN100%. Following, e.g., [15], the weights of the backbone (ResNet-18 or ResNet-50) are frozen and a linear layer followed by a softmax on top of the backbone is trained in a supervised setting on a training set. Then the model is evaluated on a test set. The training set is either IN20% or IN100%, but with labels. The test set is the validation set of ImageNet-1k. The linear layer is trained using an SGD optimizer with momentum parameter equal to 0.9 during 100 epochs. We apply a weight decay of 10^{-6}. The learning rate follows a cosine decay scheduling. The batch size is fixed at 256. Training images are augmented by composing a random cropping of an area between 8% and 100% of the total area resized to 224×224, and a random horizontal flip of probability 0.5. Images at test time are resized to 256×256, and cropped at the center with a size 224×224. The initial learning rate is tuned as a hyperparameter, and we report the top-1 accuracy on the validation set of ImageNet-1k obtained after the last training epoch, along with the corresponding top-5 accuracy. The code that we use for linear probing on IN20% or IN100% is adapted from [15] available at https://github.com/facebookresearch/vicreg.

Linear probing on Places205. We use the code of [124], available at https://github.com/valeoai/obow, for the evaluation by linear probing on Places205. The weights of the backbone (ResNet-50) pretrained on IN100% are frozen and a linear prediction layer is trained for the classification task on Places205. We note

that a batch normalization layer with *non*-learnable scale and bias parameters is added at the output of the backbone in [124]. The linear prediction layer is trained with an SGD optimizer with a 0.9 momentum parameter during 28 epochs. The weight decay is 10^{-4}. The batch size is 256. The learning rate decreases by a factor of 10 at epoch 10 and epoch 20. We use the same image augmentation pipeline for training and testing as in linear probing on IN100%. The initial learning rate is tuned as a hyperparameter, and we report the top-1 accuracy on the validation set of Places205 obtained after the last training epoch, along with the corresponding top-5 accuracy.

Linear classification on VOC2007. After pretraining a ResNet backbone, we use the VISSL library [138] to extract features of VOC2007 images resized to 224×224 by taking the output of the last average pooling layer of the pretrained ResNet backbone. We then learn a linear SVM with LIBLINEAR [105] on top of these features to predict the presence or the absence of a given class in the test images. An average precision score is then computed for each class after a 3-fold cross-validation, and we report the mean score over all classes as the mean average precision (mAP).

Semi-supervised learning. After pretraining a ResNet backbone, we fine-tune this backbone and the linear classifier on the ImageNet-1k classification task with only 1% or 10% of the labeled data. The labeled images that are considered in these subsets are the ones used in the official code of SimCLR available at https://github.com/google-research/simclr. We use an SGD optimizer with momentum parameter equal to 0.9 during 20 epochs, without weight decay. The batch size is fixed at 256. The learning rates of the backbone and the linear classifier follow a cosine decay scheduling with different initial learning rates. These initial learning rates are tuned as hyperparameters. We report the top-1 accuracy on the validation set of ImageNet-1k obtained after the last training epoch, along with the corresponding top-5 accuracy. We use the same image augmentation pipeline for training and testing as in linear probing on IN100%. The code that we use for semi-supervised learning with few labels of IN100% is the one of [15] available at https://github.com/facebookresearch/vicreg.

Weighted kNN classification. We follow the usual protocol of [45, 358]. We compute the normalized representations $f_\theta(\mathbf{x}_i)$ of the images $\mathbf{x}_i$, $i \in [\![N]\!]$, in the training set. The label of an image $\mathbf{x}_{\text{test}}$ in the test set is predicted by a weighted vote of its k nearest neighbors $\mathcal{N}_k$ in the representation space: the class c gets a score of $w_c := \sum_{i \in \mathcal{N}_k} \exp(f_\theta(\mathbf{x}_i)^\top f_\theta(\mathbf{x}_{\text{test}})/0.07) \mathbb{1}_{[c_i=c]}$ where $f_\theta(\mathbf{x}_{\text{test}})$ is normalized, c_i is the class of $\mathbf{x}_i$, and $\mathbb{1}_{[c_i=c]}$ is equal to 1 if $c_i = c$, and 0 otherwise. We report the kNN classification top-1 accuracy for $k = 20$. The image augmentation pipeline for both training and testing is the following one: images are resized to 256×256, and cropped at the center with a size 224×224. The

code that we use for kNN classification is the one of [45] available at https://github.com/facebookresearch/dino.

A.3.4 Hyperparameters when pretraining on IN20%

We describe in detail our hyperparameter tuning protocol for the experiments in Section 5.4.2 where the backbone is pretrained on IN20%.

Hyperparameter tuning on a separate validation set

For a rigorous tuning, it is important that the dataset used for the final evaluation remains unseen during pretraining and hyperparameter tuning. For each pretraining method, we pretrain on the IN20% training set (blue subset in Figure A.1) a backbone for each choice of hyperparameters. These backbones are then evaluated by weighted kNN classification on a *separate validation set*, which is another 20% subset of the ImageNet-1k train set (purple subset in Figure A.1), and we select the hyperparameters yielding the highest top-1 accuracy on this kNN evaluation. Then, we tune the learning rate for the linear probing evaluation, again on our separate validation set (purple subset in Figure A.1). Finally, we use the model trained with the best learning rate discovered for linear probing evaluation on the usual ImageNet-1k validation set (red subset in Figure A.1), which has never been seen during hyperparameter tuning.

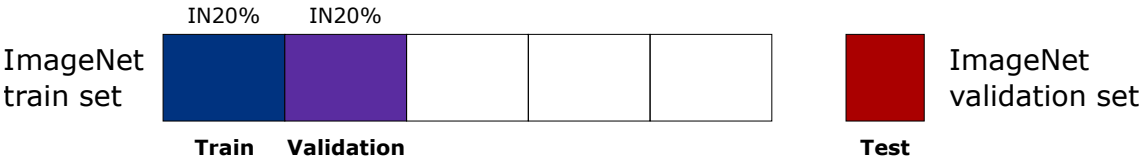


Figure A.1: ImageNet-1k dataset split for hyperparameter tuning in experiments of Section 5.4.2 on IN20%.

Hyperparameter values for linear probing on IN20%

We report the *values of the optimal hyperparameters* found after our hyperparameter tuning on a separate validation set, for each pretraining experiment on IN20% with a ResNet-18 presented in Section 5.4.2. These hyperparameters yield the evaluation results reported in Section 5.4.2 for linear probing on the usual ImageNet-1k validation set. We recall that the hyperparameters specific to each self-supervision method were introduced in Appendix A.1.4.

SimCLR. For each embedding dimension, we fix the batch size at 2048, and tune the temperature τ and the base learning rate *base_lr* for pretraining with SimCLR. Then, we tune the initial learning rate *lr_head* for linear probing on IN20%. The optimal hyperparameters are shown in Table A.1.

Table A.1: **Hyperparameter choice for SimCLR** pretrained on IN20% with a ResNet-18 during 100 epochs, evaluated by linear probing on IN20%.

Dimension	Temperature	*base_lr*	*lr_head*
$q = 1024$	0.15	1.0	0.2
$q = 2048$	0.15	1.0	0.2
$q = 4096$	0.15	1.0	0.2
$q = 8192$	0.15	0.8	0.2

AUH. For each embedding dimension, we fix the batch size at 2048, and tune the alignment weight λ, the scale of the RBF kernel t, and the base learning rate *base_lr* for pretraining with AUH. Then we tune the initial learning rate *lr_head* for linear probing on IN20%. The optimal hyperparameters are shown in Table A.2.

Table A.2: **Hyperparameter choice for AUH** pretrained on IN20% with a ResNet-18 during 100 epochs, evaluated by linear probing on IN20%.

Dimension	Alignment weight λ	Scale t	*base_lr*	*lr_head*
$q = 1024$	400	2.5	1.0	10.0
$q = 2048$	1000	2.5	1.0	4.0
$q = 4096$	2000	2.5	1.0	1.0
$q = 8192$	3000	2.5	1.0	2.0

VICReg. For each embedding dimension, we fix the batch size at 2048, and we tune the alignment weight λ, the variance weight μ, and the base learning rate *base_lr* for pretraining with VICReg. Then, we tune the initial learning rate *lr_head* for linear probing on IN20%. The optimal hyperparameters are shown in Table A.3.

Table A.3: **Hyperparameter choice for VICReg** pretrained on IN20% with a ResNet-18 during 100 epochs, evaluated by linear probing on IN20%.

Dimension	Alignment weight λ	Variance weight μ	*base_lr*	*lr_head*
$q = 1024$	4	10	0.4	0.2
$q = 2048$	4	4	0.7	1.0
$q = 4096$	10	10	0.6	0.2
$q = 8192$	10	10	0.7	0.2

SFRIK, batch size 2048. For each embedding dimension, we fix the batch size at 2048, and tune the alignment weight λ in (5.2), the kernel weights b_ℓ ($\ell \in \{2, 3\}$)

in (5.8), and the base learning rate *base_lr* for pretraining with SFRIK. Without loss of generality, the first kernel weight b_1 in (5.8) is fixed at $b_1 = 1$. Then, we tune the initial learning rate *lr_head* for linear probing on IN20%. The optimal hyperparameters are shown in Table A.4.

Table A.4: **Hyperparameter choice for SFRIK** pretrained on IN20% with a ResNet-18 during 100 epochs, evaluated by linear probing on IN20%.

Order	Dimension	Alignment weight λ	Kernel weights (b_1, b_2, b_3)	*base_lr*	*lr_head*
$L = 1$	$q = 8192$	10000	(1, 0, 0)	0.4	10.0
$L = 2$	$q = 1024$	400	(1, 40, 0)	1.0	2.0
	$q = 2048$	400	(1, 40, 0)	1.0	4.0
	$q = 4096$	1000	(1, 40, 0)	1.0	4.0
	$q = 8192$	2000	(1, 20, 0)	1.0	10.0
$L = 3$	$q = 8192$	4000	(1, 40, 40)	1.2	1.0

SFRIK, batch size 4096. We fix the dimension at $q = 8192$, and the batch size at 4096, and tune the alignment weight λ in (5.2), the kernel weights b_ℓ ($\ell \in \{2, 3\}$) in (5.8), and the base learning rate *base_lr* for pretraining with SFRIK. Without loss of generality, the first kernel weight b_1 in (5.8) is fixed at $b_1 = 1$. Then, we tune the initial learning rate *lr_head* for linear probing on IN20%. The optimal hyperparameters are: $\lambda = 4000$, $(b_1, b_2, b_3) = (1, 20, 0)$, *base_lr* $= 0.8$, *lr_head* $=$ 1.0. This yields a top-1 accuracy of 46.3 for linear probing on IN20% (vs. 47.5 when the batch size is 2048), meaning that it is not necessary to use a batch size larger than 2048 in SFRIK to obtain a better performance.

Hyperparameters for the transfer learning on Places205 and VOC2007

The values of hyperparameters for pretraining methods on IN20% are the same as above. The learning rate for linear probing on Places205 is fixed at 0.01.

Hyperparameters for the ablation on the kernel choice

We detail the experimental setting of our ablation study on the kernel choice for the generic uniformity term (5.7) in Table 5.4 of Section 5.4.2. We follow the protocol of Section 5.4.2, with an embedding dimension of $q = 8192$. The training loss is $\lambda \ell_a(\mathbf{Z}_I{}^{(1)}, \mathbf{Z}_I{}^{(2)}) + 0.5(\ell_u(\mathbf{Z}_I{}^{(1)}) + \ell_u(\mathbf{Z}_I{}^{(2)}))$, where ℓ_u is the loss (5.7) with an RBF or a general distance kernel.

RBF kernel. The kernel is $\mathcal{K}(\mathbf{u}, \mathbf{v}) = e^{-t\|\mathbf{u}-\mathbf{v}\|_2^2}$. We fix the batch size at 2048, and tune the alignment weight λ, the scale of the RBF kernel t, and the base learning rate *base_lr* for pretraining. Then, we tune the initial learning rate *lr_head* for

linear probing on IN20%. The optimal hyperparameters are: $\lambda = 100$, $t = 2$, $base\_lr = 1.0$, $lr\_head = 10$.

Generalized distance kernel. The kernel is $\mathcal{K}(\mathbf{u}, \mathbf{v}) = C - \|\mathbf{u} - \mathbf{v}\|_2^p$, where we fixed $C = 0$ because the value of this constant does not change the gradients of the optimization problem, and $p = 2$ because we verified empirically that choosing $p < 2$ yields poor results. We fix the batch size at 2048, and tune the alignment weight λ, and the base learning rate $base\_lr$ for pretraining. Then, we tune the initial learning rate lr_head for linear probing on IN20%. The optimal hyperparameters are: $\lambda = 10000$, $base\_lr = 0.6$, $lr\_head = 10$.

A.3.5 Hyperparameters when pretraining on IN100%

Table A.5 reports the selected hyperparameters for pretraining ResNet-50 on IN100% with SFRIK in Section 5.4.3.

Tuning protocol. Since hyperparameter tuning is costly on IN100%, we pretrain several ResNet-50 for different values of kernel weights b_ℓ, alignment weight λ and base learning rate $base\_lr$, and pause the pretraining after 50 epochs. We evaluate the obtained backbones on kNN classification (top-1 accuracy, $k = 20$), and select the best performing backbones. Then we continue pretraining these selected backbones until reaching epoch 200 or 400. Finally we select the hyperparameters that yield the highest top-1 accuracy for linear probing on IN100% after 200 or 400 epochs of pretraining. Because of the conclusions of Appendix A.4.1 below, our hyperparameter tuning follows the default practice in the literature on self-supervised learning where hyperparameters are selected by measuring the performance on the validation set of ImageNet-1k, which is the same set for the final evaluation by linear probing. Note that we verified experimentally a posteriori that the hyperparameters obtained by our tuning protocol are similar to the ones obtained from a tuning on a smaller dataset like STL-10 [64]. This means that an alternative tuning protocol is to tune the hyperparameters on STL-10, and generalize these hyperparameters to pretrain SFRIK on IN100%.

Table A.5: **Hyperparameter choice for SFRIK** pretrained on IN100% with a ResNet-50 during 200 or 400 epochs.

Dimension	Order	Epoch	Alignment weight λ	Kernel weights (b_1, b_2, b_3)	$base\_lr$
$q = 8192$	$L = 2$	200	4000	(1, 20, 0)	0.4
	$L = 3$	200	4000	(1, 40, 40)	0.4
$q = 16384$	$L = 2$	200	20000	(1, 40, 0)	0.4
$q = 32768$	$L = 2$	200	40000	(1, 40, 0)	0.4
	$L = 3$	200	40000	(1, 40, 40)	0.4
$q = 8192$	$L = 2$	400	4000	(1, 20, 0)	0.4

Values for hyperparameters. Tables A.6 and A.7 give the optimal hyperparameters found for linear probing on IN100%, linear probing on Places205, and semi-supervised learning with limited labels of IN100% when evaluating pretrained ResNet-50 backbones with SFRIK and VICReg. The hyperparameters that are tuned for evaluation are: the initial learning rate *lr_head* of the linear layer in linear probing; and the initial learning rate *lr_backbone* and *lr_head* for respectively the backbone and the linear layer in semi-supervised learning. The reported hyperparameters in the two tables yield the evaluation results reported in Section 5.4.3.

Table A.6: **Hyperparameter tuning for linear probing on IN100% and Places205** for SFRIK and VICReg pretrained on IN100% with a ResNet-50.

		lr_head	
Method	Epochs	IN100%	Places205
VICReg† ($q = 8192$)	200	0.02	0.01
SFRIK ($L = 2, q = 8192$)	200	1.0	0.01
SFRIK ($L = 3, q = 8192$)	200	2.0	0.01
SFRIK ($L = 2, q = 16384$)	200	0.3	0.01
SFRIK ($L = 2, q = 32768$)	200	1.0	0.01
SFRIK ($L = 3, q = 32768$)	200	0.4	0.01
SFRIK ($L = 2, q = 8192$)	400	2.0	0.01

Table A.7: **Hyperparameter tuning for semi-supervised learning** for SFRIK and VICReg pretrained on IN100% with a ResNet-50.

		Semi-supervised, 1% labels		Semi-supervised, 10% labels	
Method	Epochs	*lr_backbone*	*lr_head*	*lr_backbone*	*lr_head*
VICReg† ($q = 8192$)	200	0.02	0.2	0.2	0.04
SFRIK ($L = 2, q = 8192$)	200	0.004	1.6	0.02	0.4
SFRIK ($L = 3, q = 8192$)	200	0.002	1.0	0.01	0.2
SFRIK ($L = 2, q = 16384$)	200	0.004	1.4	0.04	0.2
SFRIK ($L = 2, q = 32768$)	200	0.004	1.0	0.04	0.1
SFRIK ($L = 3, q = 32768$)	200	0.004	1.0	0.02	0.1
SFRIK ($L = 2, q = 8192$)	400	0.004	1.4	0.02	0.2

A.3.6 Computational resources

Pretrainings of ResNet-18 on IN20% with a batch size of 2048 (respectively 4096) are performed with 4 (respectively 8) NVIDIA Tesla V100 GPUs with 32GB of memory each. Pretrainings of ResNet-50 on IN100% are performed with 8 NVIDIA Tesla V100 GPUs with 32GB of memory each. The total amount of compute used for this work is around 50000 GPU hours.

A.3.7 Public resources

We acknowledge the use of the following public resources, during the course of the experimental work of Chapter 5:

- VICReg official code [15] ..... MIT License
- DINO official code [45] ..... Apache License 2.0
- OBoW official code [124] ..... Apache License 2.0
- SwAV official code [44] ..... CC BY-NC 4.0
- SimCLR official code [50] ..... Apache License 2.0
- VISSL code [138] ..... MIT License
- ImageNet-1k dataset [78] .....
- Places 205 dataset [394] ..... Attribution CC BY
- VOC2007 dataset [103] .....

A.4 Additional experimental results

We provide in this appendix other experimental results to complement Section 5.4.

A.4.1 Hyperparameter tuning without a separate validation set

A common practice in the self-supervised learning literature, e.g., [15, 50], is to select the hyperparameters by measuring the performance on the validation set of ImageNet-1k (red dataset in Figure A.1) instead of a separate validation dataset (purple dataset in Figure A.1). In this paragraph, we verify whether this less rigorous practice changes the conclusion of the experiments in Section 5.4.2.

In Table A.8, we report the evaluation of the different backbones pretrained on IN20% after tuning each method *directly on the validation set* of ImageNet-1k, which is the same dataset used for evaluation in linear probing. By comparison with Table 5.2, which follows the more rigorous hyperparameter tuning protocol described in Appendix A.3.4, we observe that although the absolute figures of merit slightly vary if we use the less rigorous protocol instead of the more rigorous one, the conclusion of the experiments in Section 5.4.2 does not change. This gives an empirical justification to this common practice, which is why we adopt this practice on the most costly experiments with pretraining on IN100%.

Table A.8: **Linear probing on IN20% (top-1 accuracy)** at different embedding dimensions q. All methods were pretrained on IN20% with a ResNet-18 for 100 epochs. Hyperparameters specific to each method and the learning rate are tuned on the *same* dataset as the one used for evaluation in linear probing, which is less rigorous than tuning the hyperparameters on a separate validation set as described in Appendix A.3.4. Symbol † indicates models that we retrained ourselves.

	SimCLR†	AUH†	VICReg†	SFRIK		
				$L=1$	$L=2$	$L=3$
$q=1024$	45.2	45.2	40.8	-	44.2	-
$q=2048$	45.8	45.6	44.1	-	45.5	-
$q=4096$	46.3	46.8	44.9	-	47.0	-
$q=8192$	46.2	46.8	46.0	27.5	47.0	**47.6**

A.4.2 Other pretraining methods on IN100% with ResNet-50

In complement to Table 5.5, Table A.9 reports evaluation results for linear probing on IN100% and semi-supervised learning with few labels of IN100% of different ResNet-50 pretrained on IN100% with other state-of-the-art methods than the ones presented in Table 5.5. As mentioned in Section 5.4.3, we observe that, similarly to SFRIK, both BYOL and SwAV with multi-crop have a performance drop compared to VICReg on semi-supervised learning with 1% of labels, even though they perform better on linear probing on IN100%.

Table A.9: **Linear probing on IN100%, semi-supervised learning with few labels of IN100% (top-1 and top-5 accuracy).** All methods have been pretrained on IN100% with a ResNet-50 during the reported number of epochs.

Method	Epochs	Linear probing		Semi-supervised			
		IN100%		1% labels		10% labels	
		Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
SimCLR [50]	1000	68.3	89.0	48.3	75.5	65.6	87.8
OBoW [124]	200	73.8	-	-	82.9	-	90.7
BYOL [143]	1000	74.3	91.6	53.2	78.4	68.8	89.0
SwAV (with multi-crop) [44]	800	75.3	-	53.9	78.5	70.2	89.9
Barlow Twins [375]	1000	73.2	91.0	55.0	79.2	69.7	89.3
VICReg [15]	1000	73.2	91.1	54.8	79.4	69.5	89.5

Appendix B

Appendices for Chapter 6

B.1 Proof of Lemma 6.1

We begin with a simple lemma [126, Chapter 3].

Lemma B.1. *Let Σ be any set of pairs of factors, and $(\mathbf{X}, \mathbf{Y}) \in \Sigma$. We have*

$$(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Sigma) \iff (\mathbf{X}, \mathbf{Y}) \in \bigcap_{\substack{\Sigma' \subseteq \Sigma \\ (\mathbf{X},\mathbf{Y}) \in \Sigma'}} \mathcal{U}(\Sigma').$$

Proof. Let $(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Sigma)$, and consider $\Sigma' \subseteq \Sigma$ such that $(\mathbf{X}, \mathbf{Y}) \in \Sigma'$, as well as $(\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \in \Sigma'$ such that $\bar{\mathbf{X}}\bar{\mathbf{Y}}^\top = \mathbf{X}\mathbf{Y}^\top$. Since $\Sigma' \subseteq \Sigma$, we have $(\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \in \Sigma$, and because $(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Sigma)$, $(\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \sim (\mathbf{X}, \mathbf{Y})$. Moreover, $(\mathbf{X}, \mathbf{Y}) \in \Sigma'$, hence $(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Sigma')$. This is true for every $\Sigma' \subseteq \Sigma$, proving one implication. The converse is true considering the case $\Sigma' := \Sigma$. □

Lemma 6.1 is then derived from the following trivial but crucial observation.

Lemma B.2. *Let Σ be any set of pairs of factors, and $\mathbf{X}$, $\mathbf{Y}, \bar{\mathbf{Y}}$ such that $(\mathbf{X}, \mathbf{Y}), (\mathbf{X}, \bar{\mathbf{Y}}) \in \Sigma$. If $\mathbf{X}\mathbf{Y}^\top = \mathbf{X}\bar{\mathbf{Y}}^\top$ and $\mathrm{colsupp}(\bar{\mathbf{Y}})$, $\mathrm{colsupp}(\mathbf{Y})$ do not have the same cardinality, then $(\mathbf{X}, \mathbf{Y}) \not\sim (\mathbf{X}, \bar{\mathbf{Y}})$ and hence $(\mathbf{X}, \mathbf{Y}) \notin \mathcal{U}(\Sigma)$ and $(\mathbf{X}, \bar{\mathbf{Y}}) \notin \mathcal{U}(\Sigma)$. This is true in particular if there exists an index $i \in [\![k]\!]$ for which $\mathbf{X}[:, i] = \mathbf{0}$, $\mathbf{Y}[:, i] = \mathbf{0}$, $\bar{\mathbf{Y}}[:, i] \neq \mathbf{0}$, and $\mathbf{Y}[:, j] = \bar{\mathbf{Y}}[:, j]$ for all $j \neq i$.*

Proof of Lemma 6.1. We first show $\mathcal{U}(\Sigma^{\mathrm{S}}) \subseteq \mathtt{IC}^{\mathrm{S}}$ by contraposition. Let $(\mathbf{X}, \mathbf{Y}) \in \Sigma^{\mathrm{S}}$, and suppose that $\mathrm{colsupp}(\mathbf{X}) \neq \mathrm{colsupp}(\mathbf{Y})$. Up to matrix transposition, we can suppose without loss of generality that $\mathrm{colsupp}(\mathbf{Y})$ is not a subset of $\mathrm{colsupp}(\mathbf{X})$, so there is $i \in [\![k]\!]$ such that $\mathbf{X}[:, i] = \mathbf{0}$ and $\mathbf{Y}[:, i] \neq \mathbf{0}$. Define $\bar{\mathbf{Y}}$ a right factor such that $\bar{\mathbf{Y}}[:, i] = \mathbf{0}$ and $\bar{\mathbf{Y}}[:, [\![k]\!] \backslash \{i\}] = \mathbf{Y}[:, [\![k]\!] \backslash \{i\}]$. By construction, $\mathrm{supp}(\bar{\mathbf{Y}}) \subseteq \mathrm{supp}(\mathbf{Y})$, so $(\mathbf{X}, \bar{\mathbf{Y}}) \in \Sigma^{\mathrm{S}}$. Applying Lemma B.2 to $\Sigma = \Sigma^{\mathrm{S}}$, we

obtain $(\mathbf{X}, \mathbf{Y}) \notin \mathcal{U}(\Sigma^S)$. We now show $\mathcal{U}(\Sigma^S) \subseteq \mathtt{MC}^S$, also by contraposition. Let $(\mathbf{X}, \mathbf{Y}) \notin \mathtt{MC}^S$, and assume $\mathrm{colsupp}(\mathbf{X}) \neq \mathrm{colsupp}(\mathbf{S}_{\mathrm{left}})$. The reasoning would be symmetric if we supposed $\mathrm{colsupp}(\mathbf{Y}) \neq \mathrm{colsupp}(\mathbf{S}_{\mathrm{right}})$. By the previously shown inclusion $\mathcal{U}(\Sigma^S) \subseteq \mathtt{IC}^S$, we also assume $\mathrm{colsupp}(\mathbf{X}) = \mathrm{colsupp}(\mathbf{Y})$ without loss of generality. To conclude we treat three cases:

1. If $\mathrm{colsupp}(\mathbf{S}_{\mathrm{right}}) \not\subseteq \mathrm{colsupp}(\mathbf{S}_{\mathrm{left}})$ then we can fix $i \in [\![k]\!]$ such that $\mathbf{S}_{\mathrm{left}}[:, i] = \mathbf{0}$ and $\mathbf{S}_{\mathrm{right}}[:, i] \neq \mathbf{0}$. This means $\mathbf{X}[:, i] = \mathbf{Y}[:, i] = \mathbf{0}$. Setting $\bar{\mathbf{Y}} \in \Sigma^{\mathbf{S}_{\mathrm{right}}}$ such that $\bar{\mathbf{Y}}[:, i] = \mathbf{S}_{\mathrm{right}}[:, i]$ and $\bar{\mathbf{Y}}[:, [\![k]\!] \backslash \{i\}] = \mathbf{Y}[:, [\![k]\!] \backslash \{i\}]$, we build an instance as in Lemma B.2 with $\Sigma = \Sigma^S$, to show $(\mathbf{X}, \mathbf{Y}) \notin \mathcal{U}(\Sigma^S)$.

2. If $\mathrm{colsupp}(\mathbf{S}_{\mathrm{left}}) \not\subseteq \mathrm{colsupp}(\mathbf{S}_{\mathrm{right}})$, then the same arguments yield $(\mathbf{X}, \mathbf{Y}) \notin \mathcal{U}(\Sigma^S)$.

3. There remains the case $\mathrm{colsupp}(\mathbf{S}_{\mathrm{left}}) = \mathrm{colsupp}(\mathbf{S}_{\mathrm{right}})$. Since $\mathrm{colsupp}(\mathbf{X}) \subsetneq \mathrm{colsupp}(\mathbf{S}_{\mathrm{left}})$, let us fix $i \in \mathrm{colsupp}(\mathbf{S}_{\mathrm{left}})$ such that $\mathbf{X}[:, i] = \mathbf{0}$. Then, $\mathbf{S}_{\mathrm{left}}[:, i] \neq 0$, $\mathbf{S}_{\mathrm{right}}[:, i] \neq 0$, and $\mathbf{X}[:, i] = \mathbf{Y}[:, i] = \mathbf{0}$. Again, we construct $\bar{\mathbf{Y}} \in \Sigma^{\mathbf{S}_{\mathrm{right}}}$ with $\bar{\mathbf{Y}}[:, i] = \mathbf{S}_{\mathrm{right}}[:, i]$, $\bar{\mathbf{Y}}[:, [\![k]\!] \backslash \{i\}] = \mathbf{Y}[:, [\![k]\!] \backslash \{i\}]$, and we obtain an instance as in Lemma B.2 with $\Sigma = \Sigma^S$, showing that $(\mathbf{X}, \mathbf{Y}) \notin \mathcal{U}(\Sigma^S)$.

□

The following key proposition will be useful in the lifting approach presented below.

Proposition B.1. *For any pair of supports* S*, we have:* $\mathcal{U}(\Sigma^S) = \mathcal{U}(\mathtt{IC}^S) \cap \mathtt{MC}^S$.

Proof. The direct inclusion is immediate by Lemmas 6.1 and B.1. For the inverse inclusion, let $(\mathbf{X}^*, \mathbf{Y}^*) \in \mathcal{U}(\mathtt{IC}^S) \cap \mathtt{MC}^S$, and $(\mathbf{X}, \mathbf{Y}) \in \Sigma^S$ such that $\mathbf{X}\mathbf{Y}^\top = \mathbf{X}^*\mathbf{Y}^{*\top}$. The goal is to show $(\mathbf{X}, \mathbf{Y}) \sim (\mathbf{X}^*, \mathbf{Y}^*)$. Denote $J = \mathrm{colsupp}(\mathbf{X}) \cap \mathrm{colsupp}(\mathbf{Y})$. Define $(\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \in \mathtt{IC}^S$ such that

$$(\bar{\mathbf{X}}[:, J], \bar{\mathbf{Y}}[:, J]) = (\mathbf{X}[:, J], \mathbf{Y}[:, J]) \quad \text{and} \quad (\bar{\mathbf{X}}[:, [\![k]\!] \backslash J], \bar{\mathbf{Y}}[:, [\![k]\!] \backslash J]) = (\mathbf{0}, \mathbf{0}).$$

Since $\bar{\mathbf{X}}\bar{\mathbf{Y}}^\top = \mathbf{X}\mathbf{Y}^\top = \mathbf{X}^*\mathbf{Y}^{*\top}$, and $(\mathbf{X}^*, \mathbf{Y}^*) \in \mathcal{U}(\mathtt{IC}^S)$, we have $(\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \sim (\mathbf{X}^*, \mathbf{Y}^*)$. But $\mathrm{colsupp}(\mathbf{X}^*) = \mathrm{colsupp}(\mathbf{S}_{\mathrm{left}})$ and $\mathrm{colsupp}(\mathbf{Y}^*) = \mathrm{colsupp}(\mathbf{S}_{\mathrm{right}})$, because $(\mathbf{X}^*, \mathbf{Y}^*) \in \mathtt{MC}^S$. Hence, $J = \mathrm{colsupp}(\bar{\mathbf{X}}) = \mathrm{colsupp}(\mathbf{X}^*) = \mathrm{colsupp}(\mathbf{S}_{\mathrm{left}})$ and similarly $J = \mathrm{colsupp}(\mathbf{S}_{\mathrm{right}})$. This necessarily yields $\bar{\mathbf{X}} = \mathbf{X}$ and $\bar{\mathbf{Y}} = \mathbf{Y}$. In conclusion, $(\mathbf{X}, \mathbf{Y}) = (\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \sim (\mathbf{X}^*, \mathbf{Y}^*)$. □

B.2 Lifting procedure

We start by claiming two technical lemmas, whose proof is left to the reader. Recall that $\mathtt{IC}^S$, $\mathtt{MC}^S$ and Γ^S are defined in (6.8), (6.9), and (6.11).

Lemma B.3. *Let* S *be any pair of supports satisfying* $\mathrm{colsupp}(\mathbf{S}_{left}) = \mathrm{colsupp}(\mathbf{S}_{right})$*, and denote* $\mathcal{S} := \varphi(\mathrm{S})$*. Then:* $(\mathbf{X}, \mathbf{Y}) \in \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}} \iff \varphi(\mathbf{X}, \mathbf{Y}) \in \mathtt{MI}^{\mathcal{S}}$*, where* $\mathtt{MI}^{\mathcal{S}}$ *denotes the set of tuples* $\mathcal{C} \in \Gamma^{\mathcal{S}}$ *with maximal index support*[a]*:*

$$\mathtt{MI}^{\mathcal{S}} := \{\mathcal{C} \in \Gamma^{\mathcal{S}} \mid \forall i \in [\![k]\!],\ \boldsymbol{\mathcal{C}}_i = \mathbf{0} \implies \boldsymbol{\mathcal{S}}_i = \mathbf{0}\}. \tag{B.1}$$

[a] By analogy with column support, the index support of $\mathcal{C} = (\boldsymbol{\mathcal{C}}_i)_{i=1}^r$ is the subset of indices $i \in [\![k]\!]$ such that $\boldsymbol{\mathcal{C}}_i \neq \mathbf{0}$.

Lemma B.4. *The application* φ *is invariant to column rescaling:* $\varphi(\mathbf{X}, \mathbf{Y}) = \varphi(\bar{\mathbf{X}}, \bar{\mathbf{Y}})$ *for any equivalent pair of factors* $(\mathbf{X}, \mathbf{Y}) \sim (\bar{\mathbf{X}}, \bar{\mathbf{Y}})$*. Moreover, the application* φ *restricted to* $\mathtt{IC}^{\mathrm{S}}$*, denoted* $\varphi_{\mathrm{S}} \colon \mathtt{IC}^{\mathrm{S}} \to \Gamma^{\mathcal{S}}$ *is surjective, and injective up to equivalences, in the sense that* $(\mathbf{X}, \mathbf{Y}) \sim (\bar{\mathbf{X}}, \bar{\mathbf{Y}})$ *for any* $(\mathbf{X}, \mathbf{Y}), (\bar{\mathbf{X}}, \bar{\mathbf{Y}}) \in \mathtt{IC}^{\mathrm{S}}$ *such that* $\varphi_{\mathrm{S}}(\mathbf{X}, \mathbf{Y}) = \varphi_{\mathrm{S}}(\bar{\mathbf{X}}, \bar{\mathbf{Y}})$*.*

We now define the operator that sums the r matrices of a tuple $\mathcal{C}$ as

$$\mathcal{A} : \mathcal{C} = (\boldsymbol{\mathcal{C}}_i)_{i=1}^r \mapsto \sum_{i=1}^r \boldsymbol{\mathcal{C}}_i.$$

This operator corresponds to a *lifting operator* in the terminology of [61]. For any set $\Gamma \subseteq (\mathbb{C}^{m \times n})^r$ of r-tuples of rank-one matrices, denote:

$$\mathcal{U}(\Gamma) := \{\mathcal{C} \in \Gamma \mid \forall \mathcal{C}' \in \Gamma,\ \mathcal{A}(\mathcal{C}) = \mathcal{A}(\mathcal{C}') \implies \mathcal{C} = \mathcal{C}'\}.$$

The previous lemmas lead to the following theorem characterizing essential uniqueness of the factorization $\mathbf{A} := \mathbf{X}\mathbf{Y}^\top$ in Σ^{S} as the identifiability of the rank-one contributions $\varphi(\mathbf{X}, \mathbf{Y})$ from $\mathbf{A}$ with the constraint set $\Gamma^{\mathcal{S}}$.

Theorem B.1. *For any pair of supports* S *such that* $\mathrm{colsupp}(\mathbf{S}_{left}) = \mathrm{colsupp}(\mathbf{S}_{right})$*, denoting* $\mathcal{S} := \varphi(\mathrm{S})$*, we have:* $(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Sigma^{\mathrm{S}}) \iff \varphi(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Gamma^{\mathcal{S}}) \cap \mathtt{MI}^{\mathcal{S}}$*.*

Proof. By Lemma B.4, one verifies that $(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\mathtt{IC}^{\mathrm{S}}) \iff \varphi(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Gamma^{\mathcal{S}})$ hence

$$\begin{aligned}(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Sigma^{\mathrm{S}}) &\overset{\text{Proposition B.1}}{\iff} (\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\mathtt{IC}^{\mathrm{S}}) \cap \mathtt{MC}^{\mathrm{S}} \\ &\overset{\text{Lemma B.4}}{\iff} \varphi(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Gamma^{\mathcal{S}}) \text{ and } (\mathbf{X}, \mathbf{Y}) \in \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}} \\ &\overset{\text{Lemma B.3}}{\iff} \varphi(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Gamma^{\mathcal{S}}) \cap \mathtt{MI}^{\mathcal{S}}.\end{aligned}$$

□

Corollary B.1. *For any pair of supports* S *such that* $\mathrm{colsupp}(\mathbf{S}_{left}) = \mathrm{colsupp}(\mathbf{S}_{right})$, *denoting* $\mathcal{S} := \varphi(\mathrm{S})$, *we have:* $\mathcal{U}(\Sigma^{\mathrm{S}}) = \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}} \iff \mathtt{MI}^{\mathcal{S}} \subseteq \mathcal{U}(\Gamma^{\mathcal{S}})$.

Proof. Suppose $\mathtt{MI}^{\mathcal{S}} \subseteq \mathcal{U}(\Gamma^{\mathcal{S}})$. Let $(\mathbf{X}, \mathbf{Y}) \in \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}}$. By Lemma B.3, $\varphi(\mathbf{X}, \mathbf{Y}) \in \mathtt{MI}^{\mathcal{S}}$, which implies by assumption that $\varphi(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Gamma^{\mathcal{S}}) \cap \mathtt{MI}^{\mathcal{S}}$. By Theorem B.1, $(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Sigma^{\mathrm{S}})$. This shows $\mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}} \subseteq \mathcal{U}(\Sigma^{\mathrm{S}})$ and the converse inclusion holds by Lemma 6.1.

Now suppose $\mathcal{U}(\Sigma^{\mathrm{S}}) = \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}}$. Let $\mathcal{C} \in \mathtt{MI}^{\mathcal{S}}$. By Lemma B.4, there is $(\mathbf{X}, \mathbf{Y}) \in \mathtt{IC}^{\mathrm{S}}$ such that $\varphi(\mathbf{X}, \mathbf{Y}) = \mathcal{C}$. Let $i \in \mathrm{colsupp}(\mathbf{S}_{\mathrm{left}})$. By assumption, $i \in \mathrm{colsupp}(\mathbf{S}_{\mathrm{right}})$ hence $\boldsymbol{\mathcal{S}}_i \neq \mathbf{0}$. Since $\mathcal{C} \in \mathtt{MI}^{\mathcal{S}}$, we have $\mathbf{X}[:, i]\mathbf{Y}[:, i]^\top = \boldsymbol{\mathcal{C}}_i \neq 0$, which means that $i \in \mathrm{colsupp}(\mathbf{X})$. This is true for any $i \in \mathrm{colsupp}(\mathbf{S}_{\mathrm{left}})$, so $\mathrm{colsupp}(\mathbf{X}) = \mathrm{colsupp}(\mathbf{S}_{\mathrm{left}})$. By definition of $\mathtt{IC}^{\mathrm{S}}$, $\mathrm{colsupp}(\mathbf{X}) = \mathrm{colsupp}(\mathbf{Y})$, and by assumption, $\mathrm{colsupp}(\mathbf{S}_{\mathrm{left}}) = \mathrm{colsupp}(\mathbf{S}_{\mathrm{right}})$, hence $(\mathbf{X}, \mathbf{Y}) \in \mathtt{MC}^{\mathrm{S}}$. Since we supposed $\mathcal{U}(\Sigma^{\mathrm{S}}) = \mathtt{IC}^{\mathrm{S}} \cap \mathtt{MC}^{\mathrm{S}}$, we get $(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Sigma^{\mathrm{S}})$. By Theorem B.1, $\mathcal{C} = \varphi(\mathbf{X}, \mathbf{Y}) \in \mathcal{U}(\Gamma^{\mathcal{S}})$. □

B.3 Proof of Proposition 6.1

Proof. By Corollary B.1, it is enough to show that $\mathtt{MI}^{\mathcal{S}} \subseteq \mathcal{U}(\Gamma^{\mathcal{S}})$ if, and only if, $\mathcal{S} := \varphi(\mathrm{S})$ has disjoint rank-one supports.

Sufficiency is immediate: given $\mathcal{C} := (\boldsymbol{\mathcal{C}}_i)_{i=1}^k \in \mathtt{MI}^{\mathcal{S}}$ and $\mathbf{A} := \mathcal{A}(\mathcal{C})$, the entries of $\boldsymbol{\mathcal{C}}_i$ $(1 \leq i \leq k)$ can be directly identified from the submatrix $\mathbf{A} \odot \boldsymbol{\mathcal{S}}_i$, because the rank-one supports in the tuple $\mathcal{S}$ are pairwise disjoint.

For necessity, suppose there are $i, i' \in [\![k]\!]$, $i \neq i'$ such that $\mathrm{supp}(\boldsymbol{\mathcal{S}}_i) \cap \mathrm{supp}(\boldsymbol{\mathcal{S}}_{i'}) \neq \emptyset$. Let (p, q) be an index in this intersection. Denote $\mathbf{E}^{(p,q)} \in \mathbb{B}^{n \times m}$ the canonical binary matrix full of zeros except a one at index (k, l). Observe that $\mathcal{A}(\mathcal{C}) = \sum_{j \in [\![k]\!] \setminus \{i, i'\}} \boldsymbol{\mathcal{S}}_j = \mathcal{A}(\bar{\mathcal{C}})$, but $\mathcal{C} \neq \bar{\mathcal{C}}$ with $\mathcal{C}, \bar{\mathcal{C}} \in \mathtt{MI}^{\mathcal{S}} \subseteq \Gamma^{\mathcal{S}}$ defined as

$$\forall j \in [\![k]\!], \quad \boldsymbol{\mathcal{C}}_j = \begin{cases} \mathbf{0} & \text{if } \boldsymbol{\mathcal{S}}_j = \mathbf{0} \\ \mathbf{E}^{(p,q)} & \text{if } j = i \\ -\mathbf{E}^{(p,q)} & \text{if } j = i' \\ \boldsymbol{\mathcal{S}}_j & \text{otherwise} \end{cases}, \quad \bar{\boldsymbol{\mathcal{C}}}_j = \begin{cases} \mathbf{0} & \text{if } \boldsymbol{\mathcal{S}}_j = \mathbf{0} \\ 2\mathbf{E}^{(p,q)} & \text{if } j = i \\ -2\mathbf{E}^{(p,q)} & \text{if } j = i' \\ \boldsymbol{\mathcal{S}}_j & \text{otherwise} \end{cases}.$$

This shows $\mathcal{C}, \bar{\mathcal{C}} \neq \mathcal{U}(\Gamma^{\mathcal{S}})$, hence, $\mathtt{MI}^{\mathcal{S}} \not\subseteq \mathcal{U}(\Gamma^{\mathcal{S}})$, which ends the proof by contraposition. □

B.4 Complexity bounds of Algorithm 6.1 with full SVDs

Unbalanced tree. At the (unique) non-leaf node of level $\ell \in \{0, \ldots, L-2\}$, the algorithm computes the best rank-one approximation of n submatrices of size

$2 \times n/2^{\ell+1}$. Using a full SVD costs of the order of $2 \times n/2^{\ell+1} \times 2 = 2 \times n/2^{\ell}$, hence the total cost (with unbalanced tree and full SVD) is

$$\sum_{\ell=0}^{L-2} n \times 2 \times \frac{n}{2^{\ell}} = 4(1-2^{-L+1})n^2 = 4\left(1-\frac{2}{n}\right)n^2 = \mathcal{O}(n^2).$$

Balanced tree. At each of the 2^{ℓ} non-leaf nodes of level $\ell \in \{0, \dots, \log_2(L)-1\}$, the best rank-one approximation of n square submatrices of size $\sqrt{n^{1/2^{\ell}}}$ is computed: using a full SVD on each submatrix costs of the order of $(\sqrt{n^{1/2^{\ell}}})^3 = n^{3/2^{\ell+1}}$. Since $2^{\ell} \leq L/2 = \log_2(n)/2$ and $n^{1+3/2^{\ell+1}} \leq n^{1+3/4} = n^{7/4}$ for $\ell \geq 1$, the total cost with balanced tree and full SVD is

$$\begin{aligned}\sum_{\ell=0}^{\log_2(L)-1} 2^{\ell} \times n \times n^{3/2^{\ell+1}} &= n^{5/2} + \sum_{\ell=1}^{\log_2(L)-1} 2^{\ell} n^{1+3/2^{\ell+1}} \\ &\leq n^{5/2} + \sum_{\ell=1}^{\log_2(\log_2(n))-1} \frac{\log_2(n)}{2} n^{7/4} \\ &= \mathcal{O}(n^{5/2}).\end{aligned}$$

B.5 Proof of Lemma 6.5

Proof. Denote $\mathcal{S} := \varphi(\mathbf{S}_{\pi_r * \dots * \pi_s}, \mathbf{S}_{\pi_{s+1} * \dots * \pi_t}{}^{\top})$, and $I_j := \{(j-1)n/2^s + 1, \dots, jn/2^s\}$ for $j \in [\![2^s]\!]$. The right support $\mathbf{S}_{\pi_{s+1} * \dots * \pi_t}{}^{\top}$, which is equal to $\mathbf{S}_{\pi_{s+1} * \dots * \pi_t}$ by Lemma 6.3, is block diagonal with blocks $\mathbf{1}_{2^{t-s} \times 2^{t-s}} \otimes \mathbf{I}_{2^{L-t}}$ of size $n/2^s \times n/2^s$. Hence, one the one hand, for $i \in I_j$ and $i' \in I_{j'}$ with $j, j' \in [\![2^{\ell}]\!]$, $j \neq j'$, the rank-one supports $\boldsymbol{\mathcal{S}}_i$ and $\boldsymbol{\mathcal{S}}_{i'}$ are disjoint. One the other hand, the columns supports $\{\mathrm{supp}((\mathbf{S}_{\pi_r * \dots * \pi_s})[:, i]),\ i \in I_j\}$ are pairwise disjoint for each $j \in [\![2^{\ell}]\!]$, by definition of $\mathbf{S}_{\pi_r * \dots * \pi_s}$. This means that for $i, i' \in I_j$ ($j \in [\![2^{\ell}]\!]$), the rank-one supports $\boldsymbol{\mathcal{S}}_i$ and $\boldsymbol{\mathcal{S}}_{i'}$ are also disjoint, when $i \neq i'$. This ends the proof. □

B.6 Proof of Lemma 6.7

Proof. (i) ⇒ (ii): consider r, s, t such that the partial products are made of a single factor. (ii) ⇒ (i): Given $s \in [\![L-1]\!]$, we show by backward induction that $\mathbf{X}_{[\![r,s]\!]}$ has no zero column for each $r \in [\![s]\!]$.

- By assumption, $\mathbf{X}_{[\![s,s]\!]} = \mathbf{X}_s$ has no zero column.
- Let $r \in [\![2, s]\!]$, and suppose that $\mathbf{X}_{[\![r,s]\!]}$ has no zero column. For $i \in [\![n]\!]$, the i-th column of $\mathbf{X}_{[\![r-1,s]\!]} = \mathbf{X}_{r-1}\mathbf{X}_{[\![r,s]\!]}$ is a linear combination of columns $\{(\mathbf{X}_{r-1})[:, j] \mid j \in \mathrm{supp}((\mathbf{X}_{[\![r,s]\!]})[:, i])\}$. By Lemma 6.6, the supports of the rank-one contributions in $\varphi(\mathbf{X}_{r-1}, \mathbf{X}_{[\![r,s]\!]}{}^{\top})$ are pairwise disjoint, hence the

column supports $\{\mathrm{supp}((\mathbf{X}_{r-1})[:,j]) \mid j \in \mathrm{supp}((\mathbf{X}_{[\![r,s]\!]})[:,i])\}$ are pairwise disjoint. But $\mathrm{supp}((\mathbf{X}_{[\![r,s]\!]})[:,i])$ is not empty as $\mathbf{X}_{[\![r,s]\!]}$ has no zero column. As $\mathbf{X}_{r-1}$ is also not empty, the i-th column of $\mathbf{X}_{[\![r-1,s]\!]}$ is non-zero, and this is true for each $i \in [\![n]\!]$, which ends the induction.

A similar induction shows that $\mathbf{X}_{[\![s+1,t]\!]}$ has no zero row for each $2 \leq s+1 \leq t \leq L$. □

Appendix C

Appendices for Chapter 7

C.1 Proof for results in Section 7.3

This section is devoted to the proof of Theorem 7.1. To that end we first introduce the following technical lemma.

Lemma C.1. *Consider support constraints* $(\mathbf{L}, \mathbf{R})$ *satisfying the conditions of Theorem 7.1. For any* $(\mathbf{X}, \mathbf{Y})$ *such that* $\mathrm{supp}(\mathbf{X}) \subseteq \mathbf{L}, \mathrm{supp}(\mathbf{Y}) \subseteq \mathbf{R}$*, we have:*

$$\forall P \in \mathcal{P}(\mathbf{L}, \mathbf{R}), \quad (\mathbf{XY})[R_P, C_P] = \mathbf{X}[R_P, P]\mathbf{Y}[P, C_P], \tag{C.1}$$

where R_P, C_P *are defined as in Definition 7.1.*

Proof. For any pair of matrices $(\mathbf{X}, \mathbf{Y}) \in \mathbb{C}^{m\times r} \times \mathbb{C}^{r\times n}$:

$$\mathbf{XY} = \sum_{i=1}^{r} \mathbf{X}[:, i]\mathbf{Y}[i, :] = \sum_{P\in\mathcal{P}(\mathbf{L},\mathbf{R})} \mathbf{X}[:, P]\mathbf{Y}[P, :]. \tag{C.2}$$

For any $P, \tilde{P} \in \mathcal{P}(\mathbf{L}, \mathbf{R})$ such that $P \neq \tilde{P}$, $\mathbf{U}_P$ and $\mathbf{U}_{\tilde{P}}$ are disjoint by assumption on $\mathbf{L}, \mathbf{R}$. Since $\mathrm{supp}(\mathbf{X}[:, \tilde{P}]\mathbf{Y}[\tilde{P}, :]) \subseteq \mathbf{U}_{\tilde{P}}$, we have $(\mathbf{X}[:, \tilde{P}]\mathbf{Y}[\tilde{P}, :])[R_P, C_P] = \mathbf{0}$. Hence, by (C.2):

$$(\mathbf{XY})[R_P, C_P] = (\mathbf{X}[:, P]\mathbf{Y}[P, :])[R_P, C_P] = \mathbf{X}[R_P, P]\mathbf{Y}[P, C_P].$$

□

The following proof of Theorem 7.1 is mainly taken from [212], but we additionally compute the infimum value of Problem (7.5).

Proof for Theorem 7.1. In this proof, we use the simple shorthand $\mathcal{P}$ for $\mathcal{P}(\mathbf{L}, \mathbf{R})$, and we denote $\Sigma := \{(\mathbf{X}, \mathbf{Y}) \mid \mathrm{supp}(\mathbf{X}) \subseteq \mathbf{L}, \mathrm{supp}(\mathbf{Y}) \subseteq \mathbf{R}\}$. Recall that $(U_i)_{i=1}^r := \varphi(\mathbf{X}, \mathbf{Y})$ where φ is the lifting operator defined in (6.10).

Let $(\mathbf{X}, \mathbf{Y}) \in \Sigma$. Then, $\operatorname{supp}(\mathbf{X}[:, P]\mathbf{Y}[P, :]) \subseteq \mathbf{U}_P$ for any $P \in \mathcal{P}$, hence $\operatorname{supp}(\mathbf{XY}) \subseteq \bigcup_{P\in\mathcal{P}} \mathbf{U}_P$, and $(\mathbf{XY}) \odot \overline{\mathbf{U}_\mathcal{P}} = \mathbf{0}$ where we denote $\overline{\mathbf{U}_\mathcal{P}} := (\llbracket m \rrbracket \times \llbracket n \rrbracket) \setminus (\bigcup_{P\in\mathcal{P}} \mathbf{U}_P)$. Moreover, by assumption, $\mathbf{U}_P$ and $\mathbf{U}_{\tilde{P}}$ are disjoint for any $P, \tilde{P} \in \mathcal{P}$ such that $P \neq \tilde{P}$, so:

$$\begin{aligned}
\|\mathbf{A} - \mathbf{XY}\|_F^2 &= \left(\sum_{P\in\mathcal{P}} \|(\mathbf{A} - \mathbf{XY}) \odot \mathbf{U}_P\|_F^2 \right) + \|(\mathbf{A} - \mathbf{XY}) \odot \overline{\mathbf{U}_\mathcal{P}}\|_F^2 \\
&= \left(\sum_{P\in\mathcal{P}} \|(\mathbf{A} - \mathbf{XY})[R_P, C_P]\|_F^2 \right) + \|\mathbf{A} \odot \overline{\mathbf{U}_\mathcal{P}}\|_F^2 \\
&= \left(\sum_{P\in\mathcal{P}} \|\mathbf{A}[R_P, C_P] - (\mathbf{XY})[R_P, C_P]\|_F^2 \right) + \|\mathbf{A} \odot \overline{\mathbf{U}_\mathcal{P}}\|_F^2 \\
&\overset{(C.1)}{=} \left(\sum_{P\in\mathcal{P}} \|\mathbf{A}[R_P, C_P] - \mathbf{X}[R_P, P]\mathbf{Y}[P, C_P]\|_F^2 \right) + \|\mathbf{A} \odot \overline{\mathbf{U}_\mathcal{P}}\|_F^2.
\end{aligned} \tag{C.3}$$

Since $\mathcal{P}$ is a partition, this implies that each terms in the sum $\sum_{P\in\mathcal{P}} \|\mathbf{A}[R_P, C_P] - \mathbf{X}[R_P, P]\mathbf{Y}[P, C_P]\|^2$ involves columns of $\mathbf{X}$ and rows of $\mathbf{Y}$ that are not involved in other terms of the sum. Moreover, we remark that for any $P \in \mathcal{P}$, the matrix $\mathbf{X}[R_P, P]\mathbf{Y}[P, C_P]$ is of rank at most $|P|$. In other words, minimizing the right-hand-side of (C.3) with respect to $(\mathbf{X}, \mathbf{Y}) \in \Sigma$ is equivalent to minimize each term of the sum for $P \in \mathcal{P}$, which is the problem of finding the best rank-$|P|$ approximation of $\mathbf{A}[R_P, C_P]$. This yields the claimed equation (7.7). □

Remark C.1. *We can go further and provide a closed-form expression for the value of the infimum (7.7). In general, the distance between a matrix $\mathbf{A}$ to the set of matrices of rank at most k is given by $\|\mathbf{A}\|_F^2 - \sum_{j=1}^k \sigma_j^2(\mathbf{A})$ where $\sigma_j(\cdot)$ is the jth largest eigenvalue of a matrix. Injecting this expression in (7.7) yields:*

$$\begin{aligned}
\inf_{(\mathbf{X},\mathbf{Y})\in\Sigma} \|\mathbf{A} - \mathbf{XY}\|_F^2 &= \left(\sum_{P\in\mathcal{P}} \min_{\mathbf{B}, \operatorname{rank}(\mathbf{B})\leq|P|} \|\mathbf{A}[R_P, C_P] - \mathbf{B}\|_F^2 \right) + \|\mathbf{A} \odot \overline{\mathbf{U}_\mathcal{P}}\|_F^2 \\
&= \sum_{P\in\mathcal{P}} \left(\|\mathbf{A}[R_P, C_P]\|_F^2 - \sum_{j=1}^{|P|} \sigma_j^2(\mathbf{A}[R_P, C_P]) \right) + \|\mathbf{A} \odot \overline{\mathbf{U}_\mathcal{P}}\|_F^2.
\end{aligned}$$

This gives the following closed-form expression:

$$\inf_{(\mathbf{X},\mathbf{Y})\in\Sigma} \|\mathbf{A} - \mathbf{XY}\|_F^2 = \|\mathbf{A}\|_F^2 - \sum_{P\in\mathcal{P}} \sum_{j=1}^{|P|} \sigma_j^2(\mathbf{A}[R_P, C_P]). \tag{C.4}$$

C.2 Proof for results in Section 7.4

C.2.1 Proof of Proposition 7.1

Proof. Let $q = q(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$. By Definition 7.4, $a_1 \mid a_2$ and $d_2 \mid d_1$, so there are two integers r, s such that $a_2 = ra_1$ and $d_1 = sd_2$. Since $a_1c_1/a_2 = b_2d_2/d_1 = q$ by

Definition 7.4, this yields $c_1 = a_2q/a_1 = qr$, $b_2 = d_1q/d_2 = qs$. Thus,

$$\begin{aligned}
\mathbf{S}_{\pi_1}\mathbf{S}_{\pi_2} &= \left(\mathbf{I}_{a_1} \otimes \mathbf{1}_{b_1 \times qr} \otimes \mathbf{I}_{d_1}\right)\left(\mathbf{I}_{a_2} \otimes \mathbf{1}_{qs \times c_2} \otimes \mathbf{I}_{d_2}\right) \\
&= \left(\mathbf{I}_{a_1} \otimes \mathbf{1}_{b_1 \times r} \otimes \mathbf{1}_{1 \times q} \otimes \mathbf{I}_{d_1}\right)\left(\mathbf{I}_{a_2} \otimes \mathbf{1}_{q \times 1} \otimes \mathbf{1}_{s \times c_2} \otimes \mathbf{I}_{d_2}\right) \\
&= \left[(\mathbf{I}_{a_1} \otimes \mathbf{1}_{b_1 \times r}) \otimes \mathbf{1}_{1 \times q} \otimes \mathbf{I}_{d_1}\right]\left[\mathbf{I}_{a_2} \otimes \mathbf{1}_{q \times 1} \otimes (\mathbf{1}_{s \times c_2} \otimes \mathbf{I}_{d_2})\right] \\
&\overset{(\star)}{=} \left(\mathbf{I}_{a_1} \otimes \mathbf{1}_{b_1 \times r}\right) \otimes \left(\mathbf{1}_{1 \times q}\mathbf{1}_{q \times 1}\right) \otimes \left(\mathbf{1}_{s \times c_2} \otimes \mathbf{I}_{d_2}\right) \\
&= \left(\mathbf{I}_{a_1} \otimes \mathbf{1}_{b_1 \times r}\right) \otimes \left(q\mathbf{1}_{1 \times 1}\right) \otimes \left(\mathbf{1}_{s \times c_2} \otimes \mathbf{I}_{d_2}\right) \\
&= q\left(\mathbf{I}_{a_1} \otimes \mathbf{1}_{b_1 s \times rc_2} \otimes \mathbf{I}_{d_2}\right) \\
&= q\mathbf{S}_{\pi_1 * \pi_2} \quad \left(\text{because } \frac{b_1 d_1}{d_2} = b_1 s \text{ and } \frac{a_2 c_2}{c_1} = rc_2\right).
\end{aligned}$$

We can use the equality $(\mathbf{A} \otimes \mathbf{C} \otimes \mathbf{E})(\mathbf{B} \otimes \mathbf{D} \otimes \mathbf{F}) = (\mathbf{AB}) \otimes (\mathbf{CD}) \otimes (\mathbf{EF})$ in $(\star)$ because, according to our conditions for chainability, the sizes of $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}$ in $(\star)$ make the matrix products $\mathbf{AB}, \mathbf{CD}$ and $\mathbf{EF}$ well-defined. □

C.2.2 Proof of Lemma 7.2

The proof is based on Lemma 7.8 introduced in Section 7.5.

Proof. Let $\beta := (\pi_1, \pi_2)$ be a pair of chainable patterns. A fortiori, by Lemma 7.8, $(\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})$ satisfies the condition of Theorem 7.1. Therefore, for any matrix $\mathbf{A}$, we have $\mathbf{A} \in \mathcal{B}^\beta$ if, and only if:

$$\begin{aligned}
&\min_{(\mathbf{X},\mathbf{Y}) \in \Sigma^\beta} \|\mathbf{A} - \mathbf{XY}\|_F^2 = 0 \\
\overset{(7.7)}{\iff} &\sum_{P \in \mathcal{P}(\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})} \min_{\mathbf{B}, \mathrm{rank}(\mathbf{B}) \leq |P|} \|\mathbf{A}[R_P, C_P] - \mathbf{B}\|_F^2 + \sum_{(i,j) \notin \mathrm{supp}(\mathbf{S}_{\pi_1}\mathbf{S}_{\pi_2})} \mathbf{A}[i,j]^2 = 0 \\
\overset{(7.12)}{\iff} &\sum_{P \in \mathcal{P}(\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})} \min_{\mathbf{B}, \mathrm{rank}(\mathbf{B}) \leq |P|} \|\mathbf{A}[R_P, C_P] - \mathbf{B}\|_F^2 + \sum_{(i,j) \notin \mathrm{supp}(\mathbf{S}_{\pi_1 * \pi_2})} \mathbf{A}[i,j]^2 = 0 \\
\iff &\begin{cases} \mathrm{rank}(\mathbf{A}[R_P, C_P]) \leq |P|, & \forall P \in \mathcal{P}(\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2}) \\ \mathbf{A} \in \Sigma^{\pi_1 * \pi_2} \end{cases}.
\end{aligned}$$

This proves (7.13). For any $P \in \mathcal{P}(\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})$, we have $|R_P| = b_1, |C_P| = c_2$, because by Definition 7.2, the support of any column in $\mathbf{S}_{\pi_1}$ and any row in $\mathbf{S}_{\pi_2}$ is of cardinal b_1 and c_2, respectively. Moreover, we have: $\sum_{P \in \mathcal{P}(\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})} |P| \, \mathbf{U}_P = \sum_{\mathbf{U}_i \in \varphi(\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})} \mathbf{U}_i = \mathbf{S}_{\pi_1}\mathbf{S}_{\pi_2} = \mathbf{S}_{\pi_1}\mathbf{S}_{\pi_2} = q\mathbf{S}_{(\pi_1 * \pi_2)}$, where the first equality comes from Definition 7.1, the second equality simply comes from the rank-one decomposition of matrix multiplication, and the third equality comes from Proposition 7.1. This implies $|P| = q$ for any $P \in \mathcal{P}(\mathbf{S}_{\pi_1}, \mathbf{S}_{\pi_2})$. □

C.2.3 Proof for Lemma 7.3

Proof. Denote $\pi_\ell = (a_\ell, b_\ell, c_\ell, d_\ell)$ for $\ell \in [\![3]\!]$. Let us show that π_1 and $\pi_2 * \pi_3$ are chainable and $q(\pi_1, \pi_2 * \pi_3) = q(\pi_1, \pi_2)$. Since (π_2, π_3) is chainable, by definition of $*$ (Definition 7.4), we have:

$$(\tilde{a}, \tilde{b}, \tilde{c}, \tilde{d}) = \tilde{\pi} := \pi_2 * \pi_3 = \left(a_2, \frac{b_2 d_2}{d_3}, \frac{a_3 c_3}{a_2}, d_3\right).$$

We then verify that $(\pi_1, \tilde{\pi})$ satisfies the conditions of Definition 7.4:

1. By chainability of (π_1, π_2), we have $q(\pi_1, \pi_2) := a_1 c_1 / a_2 = b_2 d_2 / d_1 \in \mathbb{N}$. Therefore: $a_1 c_1 / \tilde{a} = a_1 c_1 / a_2 = q(\pi_1, \pi_2) = b_2 d_2 / d_1 = \tilde{b}\tilde{d} / d_1 \in \mathbb{N}$. This means that $q(\pi_1, \pi_2 * \pi_3) = q(\pi_1, \pi_2)$.
2. By chainability of (π_1, π_2), we have $a_1 \mid a_2$. Since $\tilde{a} = a_2$ by definition, we have $a_1 \mid \tilde{a}$.
3. By chainability of (π_1, π_2), we have $d_2 \mid d_1$, and by chainability of (π_2, π_3), we have $d_3 \mid d_2$. Thus, $d_3 \mid d_1$, hence $\tilde{d} \mid d_1$ because $\tilde{d} = d_3$ by definition.

In conclusion, $(\pi_1, \pi_2 * \pi_3)$ is chainable with $q(\pi_1, \pi_2 * \pi_3) = q(\pi_1, \pi_2)$. Computing $\pi_1 * (\pi_2 * \pi_3)$ explicitly by (7.11) gives $\pi_1 * (\pi_2 * \pi_3) = \left(a_1, \frac{b_1 d_1}{d_3}, \frac{a_3 c_3}{a_1}, d_3\right)$. Similarly, we can show that $(\pi_1 * \pi_2, \pi_3)$ is also chainable with $q(\pi_1 * \pi_2, \pi_3) = q(\pi_2, \pi_3)$, and we can indeed verify that $(\pi_1 * \pi_2) * \pi_3 = \pi_1 * (\pi_2 * \pi_3)$ using (7.11). □

C.2.4 Proof of Lemma 7.4

We give the explicit formula for $(\pi_1 * \ldots * \pi_L)$ in (7.14).

Proof. Let us show that $\pi_1 * \ldots * \pi_L = \left(a_1, \frac{b_1 d_1}{d_L}, \frac{a_L c_L}{a_1}, d_L\right)$, for each chainable $\beta = (\pi_\ell)_{\ell=1}^L = (a_\ell, b_\ell, c_\ell, d_\ell)_{\ell=1}^L$ of depth $L \geq 2$. The proof is an induction on $L \geq 2$.

- If $L = 2$, the result comes from (7.11).
- Let $L \geq 2$, and assume that the statement holds for any chainable architecture of depth L. Consider a chainable architecture $\beta := (\pi_\ell)_{\ell=1}^{L+1} = (a_\ell, b_\ell, c_\ell, d_\ell)_{\ell=1}^{L+1}$ of depth $L+1$. By the induction hypothesis, we have $\pi_1 * \ldots * \pi_L = \left(a_1, \frac{b_1 d_1}{d_L}, \frac{a_L c_L}{a_1}, d_L\right)$. Therefore,

$$\begin{aligned}
\pi_1 * \ldots * \pi_L * \pi_{L+1} &= \left(a_1, \frac{b_1 d_1}{d_L}, \frac{a_L c_L}{a_1}, d_L\right) * (a_{L+1}, b_{L+1}, c_{L+1}, d_{L+1}) \\
&= \left(a_1, \frac{b_1 d_1 d_L}{d_L d_{L+1}}, \frac{a_{L+1} c_{L+1}}{a_1}, d_{L+1}\right) \\
&= \left(a_1, \frac{b_1 d_1}{d_{L+1}}, \frac{a_{L+1} c_{L+1}}{a_1}, d_{L+1}\right).
\end{aligned}$$

□

C.2.5 Proof for Lemma 7.5

Proof. By (7.14), we have:

$$\begin{aligned}(a,b,c,d) = \boldsymbol{\pi} := \boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s &= \left(a_r, \frac{b_r d_r}{d_s}, \frac{a_s c_s}{a_r}, d_s\right), \\ (a',b',c',d') = \boldsymbol{\pi}' := \boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t &= \left(a_{s+1}, \frac{b_{s+1} d_{s+1}}{d_t}, \frac{a_t c_t}{a_{s+1}}, d_t\right).\end{aligned} \tag{C.5}$$

We show the chainability of $(\boldsymbol{\pi}, \boldsymbol{\pi}')$ by verifying the conditions of Definition 7.4:

1. By chainability of $(\boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1})$, $q(\boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1}) = a_s c_s / a_{s+1} = b_{s+1} d_{s+1} / d_s \in \mathbb{N}$. This means that $ac/a' = a_s c_s / a_{s+1} = q(\boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1}) = b_{s+1} d_{s+1} / d_s = b'd'/d$ and $q(\boldsymbol{\pi}, \boldsymbol{\pi}') = q(\boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1}) \in \mathbb{N}$.

2. By chainability of $\boldsymbol{\beta}$, $a_\ell \mid a_{\ell+1}$ for all $\ell \in [\![L-1]\!]$, so $a_r \mid a_{s+1}$ because $r \leq s$, hence $a \mid a'$ because $a = a_r$ and $a' = a_{s+1}$.

3. By chainability of $\boldsymbol{\beta}$, $d_{\ell+1} \mid d_\ell$ for all $\ell \in [\![L-1]\!]$, so $d_t \mid d_s$ because $s \leq t$, hence $d' \mid d$ because $d' = d_t$ and $d = d_s$.

□

C.2.6 Proof for Lemma 7.6

Proof. The explicit formulas for $(a,b,c,d) = \boldsymbol{\pi} := \boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s$ and $(a',b',c',d') = \boldsymbol{\pi}' := \boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t$ are given in (C.5). By Lemma 7.5, $(\boldsymbol{\pi}, \boldsymbol{\pi}')$ is chainable, with $q(\boldsymbol{\pi}, \boldsymbol{\pi}') = q(\boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1})$. To show the non-redundancy of $(\boldsymbol{\pi}, \boldsymbol{\pi}')$, it remains to show $q(\boldsymbol{\pi}, \boldsymbol{\pi}') < \min(b, c')$. Let us show $q(\boldsymbol{\pi}, \boldsymbol{\pi}') < b$. Because $\boldsymbol{\beta}$ is not redundant, by Definition 7.6, we have $q(\boldsymbol{\pi}_\ell, \boldsymbol{\pi}_{\ell+1}) < b_\ell$ for any $\ell \in [\![L-1]\!]$. But $q(\boldsymbol{\pi}_\ell, \boldsymbol{\pi}_{\ell+1}) = b_{\ell+1} d_{\ell+1} / d_\ell$ by Definition 7.4. Therefore, $b_{\ell+1} d_{\ell+1} < b_\ell d_\ell$ for any $\ell \in [\![L-1]\!]$. Thus, since $r \leq s$, we have $b_{s+1} d_{s+1} < b_r d_r$. A fortiori, $\frac{b_{s+1} d_{s+1}}{d_s} < \frac{b_r d_r}{d_s}$. But by (C.5), $\frac{b_r d_r}{d_s} = b$ and $\frac{b_{s+1} d_{s+1}}{d_s} = \frac{b'd'}{d} = q(\boldsymbol{\pi}, \boldsymbol{\pi}')$. In conclusion, $q(\boldsymbol{\pi}, \boldsymbol{\pi}') = \frac{b_{s+1} d_{s+1}}{d_s} < \frac{b_r d_r}{d_s} = b$. A similar argument yields $q(\boldsymbol{\pi}, \boldsymbol{\pi}') < c'$. This ends the proof. □

C.3 Details on the orthonormalization operations in Section 7.6

The goal of this section is to clarify the nature of the orthonormalization operations introduced in Algorithm 7.5, which involve the procedure described in Algorithm 7.6. We start by giving a formal definition of orthonormality for a $\boldsymbol{\pi}$-butterfly factor (Appendix C.3.1). Then we describe important properties of such orthonormal butterfly factors (Appendix C.3.2), in order to explain the nature of Algorithm 7.6 (Appendix C.3.3).

C.3.1 Definition of orthonormal butterfly factors

In order to introduce a notion of orthonormality for π-butterfly factors, we start by the following remark.

Lemma C.2. *Let $\pi := (a,b,c,d)$ be a pattern, and q be an integer. Assume that there exist another pattern π' such that (π,π') is chainable with $q(\pi,\pi') = q$. Then, the partition $\mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\pi'})$ is the same for any π' such that (π,π') is chainable with $q(\pi,\pi') = q$. This partition does not depend on π' and is equal to*

$$\{I_{t,k}\}_{t,k} := \left\{ \{k + (t-1)dq + (j-1)d\}_{j\in[\![q]\!]} \mid (t,k) \in [\![ac/q]\!] \times [\![d]\!] \right\}. \quad \text{(C.6)}$$

Remark C.2. *If the pattern π' is such that the pair (π,π') is chainable with $q(\pi,\pi') = q$, then $c/q \in \mathbb{N}$, because $q = q(\pi,\pi') = ac/\tilde{a}$ and $a \mid \tilde{a}$.*

Proof. We will prove that $\mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\tilde{\pi}}) = \{I_{t,k}\}_{t,k}$ for any pattern $\tilde{\pi}$ such that $(\pi,\tilde{\pi})$ is chainable with $q(\pi,\tilde{\pi}) = q$. For this, we fix such a pattern $\tilde{\pi}$, and we claim that it is sufficient to prove that, for any $P \in \mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\tilde{\pi}})$, there exists a unique $(k,t) \in [\![d]\!] \times [\![ca/q]\!]$ such that $P = I_{k,t}$, because $\{I_{t,k}\}_{t,k}$ and $\mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\tilde{\pi}})$ have the same cardinal: indeed, by Lemma 7.2, $|P| = q$ for any $P \in \mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\tilde{\pi}})$, so the cardinal of $\mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\tilde{\pi}})$ is acd/q (since there are acd columns in $\mathbf{S}_\pi$), which is precisely the cardinal of the set (C.6).

Define the partition of $[\![acd]\!]$ into ac/q consecutive intervals of length dq as:

$$\forall t \in [\![ac/q]\!], \quad I_t := [\![(t-1)dq + 1, tdq]\!].$$

By chainability of $(\pi,\tilde{\pi})$, we have $acd = \tilde{a}\tilde{b}\tilde{d}$, $ac/q = \tilde{a}$, and $dq = \tilde{b}\tilde{d}$. In other words: $\{I_t\}_{t\in[\![ac/q]\!]} = \{(t-1)\tilde{b}\tilde{d} + 1, t\tilde{b}\tilde{d}\}_{t\in[\![\tilde{a}]\!]}$ is a partition of $[\![\tilde{a}\tilde{b}\tilde{d}]\!]$ into $\tilde{a}$ consecutive intervals of length $\tilde{b}\tilde{d}$. Because of the block diagonal structure of $\mathbf{S}_{\tilde{\pi}} = \mathbf{I}_{\tilde{a}} \otimes \mathbf{1}_{\tilde{b}\times\tilde{c}} \otimes \mathbf{I}_{\tilde{d}}$, two rows $\mathbf{S}_{\tilde{\pi}}[i,:]$ and $\mathbf{S}_{\tilde{\pi}}[i',:]$ cannot have the same support if the two indices i, i' belong each to two different $I_t, I_{t'}$ for $t \neq t'$. A fortiori, indices belonging to different $I_t, I_{t'}$ cannot be in the same equivalence class $P \in \mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\tilde{\pi}})$, by Definition 7.1.

Then, for each $t \in [\![ac/q]\!]$, the interval I_t of cardinal dq can be partitionned into q subsets of integers equally spaced by a distance d as:

$$I_t = [\![(t-1)dq + 1, tdq]\!] = \bigcup_{k\in[\![d]\!]} \{k + (t-1)dq + (j-1)d\}_{j\in[\![q]\!]} = \bigcup_{k\in[\![d]\!]} I_{t,k}. \quad \text{(C.7)}$$

For a given $t \in [\![ac/q]\!]$, if $i \in I_{t,k}$ and $i' \in I_{t,k'}$ for some $k \neq k'$, then $(i \mod d) = k \neq k' = (i' \mod d)$ by construction. By the structure $\mathbf{S}_\pi = \mathbf{I}_a \otimes \mathbf{1}_{b\times c} \otimes \mathbf{I}_d$, this yields $\mathbf{S}_\pi[:,i] \neq \mathbf{S}_\pi[:,i']$. A fortiori, indices belonging to different $I_{t,k}, I_{t,k'}$ for $k \neq k'$ cannot be in the same equivalence class $P \in \mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\tilde{\pi}})$, by Definition 7.1.

In conclusion, this means that for any $P \in \mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\tilde{\pi}})$, there exist $(t,k) \in [\![ac/q]\!] \times [\![d]\!]$ such that $P \subseteq I_{t,k}$. But $|P| = q = |I_{t,k}|$, so $P = I_{t,k}$. We conclude

the proof by remarking that such a pair (t,k) is unique, because $\{I_{t,k}\}_{t,k}$ forms a partition of $[\![acd]\!]$. □

We can prove the following analogous result using the same argument.

Lemma C.3. *Let $\boldsymbol{\pi} := (a,b,c,d)$ be a pattern, and q be an integer. Assume that there exist another pattern $\boldsymbol{\pi}'$ such that $(\boldsymbol{\pi}',\boldsymbol{\pi})$ is chainable with $q(\boldsymbol{\pi}',\boldsymbol{\pi}) = q$. Then, the partition $\mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}'},\mathbf{S}_{\boldsymbol{\pi}})$ is the same for any $\boldsymbol{\pi}'$ such that $(\boldsymbol{\pi}',\boldsymbol{\pi})$ is chainable with $q(\boldsymbol{\pi}',\boldsymbol{\pi}) = q$.*

Lemmas C.2 and C.3 lead to the following definition.

Definition C.1. *Let $\boldsymbol{\pi}$ be a pattern, and q be an integer, define:*

1. *$\mathcal{P}_c(\boldsymbol{\pi},q) := \mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}},\mathbf{S}_{\tilde{\boldsymbol{\pi}}})$ for any pattern $\tilde{\boldsymbol{\pi}}$ such that $(\boldsymbol{\pi},\tilde{\boldsymbol{\pi}})$ is chainable with $q(\boldsymbol{\pi},\tilde{\boldsymbol{\pi}}) = q$ if such a $\tilde{\boldsymbol{\pi}}$ exists.*
2. *$\mathcal{P}_r(\boldsymbol{\pi},q) := \mathcal{P}(\mathbf{S}_{\tilde{\boldsymbol{\pi}}},\mathbf{S}_{\boldsymbol{\pi}})$ for any pattern $\tilde{\boldsymbol{\pi}}$ such that $(\boldsymbol{\pi},\tilde{\boldsymbol{\pi}})$ is chainable with $q(\boldsymbol{\pi},\tilde{\boldsymbol{\pi}}) = q$ if such a $\tilde{\boldsymbol{\pi}}$ exists.*

We can then define the notion of orthonoramlity for a butterfly factor as follow.

Definition C.2 (q-column/row-orthonormal butterfly factor)**.** *Given an integer q and a pattern $\boldsymbol{\pi}$ such that $\mathcal{P}_c(\boldsymbol{\pi},q)$ (resp. $\mathcal{P}_r(\boldsymbol{\pi},q)$) is well-defined, a butterfly factor $\mathbf{A} \in \Sigma^{\boldsymbol{\pi}}$ is* q-column-orthonormal *(resp.* q-row-orthonormal*) if the submatrix $\mathbf{A}[:,P]$ is orthonormal by column (resp. $\mathbf{A}[P,:]$ is orthonormal by row) for each $P \in \mathcal{P}_c(\boldsymbol{\pi},q)$ (resp. $P \in \mathcal{P}_r(\boldsymbol{\pi},q)$).*

Remark C.3. *Another equivalent definition of column or row orthonormality is to require $\mathbf{A}[R_P,P]$ (resp. $\mathbf{A}[P,C_P]$) to be column (resp. row)-orthonormal*[1]*. Indeed, in the case of column-orthonormality, the submatrix $\mathbf{A}[:,P]$ has the form: $\mathbf{A}[:,P] = \begin{pmatrix} \mathbf{A}[R_P,P] \\ \mathbf{0} \end{pmatrix}$ up to some row permutation. Thus, $\mathbf{A}[:,P]$ is column-orthonormal if, and only if, $\mathbf{A}[R_P,P]$ is column-orthonormal. The same reasoning holds in the case of row-orthonormality.*

C.3.2 Properties of orthonormal butterfly factors

We introduce properties related to orthonormal butterfly factors.

Preservation of the inner product

Similarly to classical orthonormal matrices, orthonormal butterfly factors preserve the inner product in the following sense.

[1] R_P, C_P are defined as in Definition 7.1

Lemma C.4. *Consider a chainable pair (π, π'). Then, for any $\mathbf{X} \in \Sigma^{\pi}$ that is $q(\pi, \pi')$-column-orthonormal, and any $\mathbf{Y}_1, \mathbf{Y}_2 \in \Sigma^{\pi'}$:*

$$\langle \mathbf{X}\mathbf{Y}_1, \mathbf{X}\mathbf{Y}_2 \rangle = \langle \mathbf{Y}_1, \mathbf{Y}_2 \rangle$$

where $\langle \cdot, \cdot \rangle$ denotes the usual Frobenius inner product of two matrices.

Proof. By Lemma 7.8, for $i \in \{1, 2\}$, we have $\mathrm{supp}(\mathbf{X}\mathbf{Y}_i) \subseteq \cup_{P \in \mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\pi'})} R_P \times C_P$ where $\{R_P \times C_P\}_P$ are pairwise disjoint. Hence:

$$\langle \mathbf{X}\mathbf{Y}_1, \mathbf{X}\mathbf{Y}_2 \rangle = \sum_{P \in \mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\pi'})} \langle (\mathbf{X}\mathbf{Y}_1)[R_P, C_P], (\mathbf{X}\mathbf{Y}_2)[R_P, C_P] \rangle. \tag{C.8}$$

Moreover, $\mathbf{S}_{\pi'} = \bigcup_{P \in \mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\pi'})} P \times C_P$ and $\{P \times C_P\}_P$ are pairwise disjoint, because $\mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\pi'})$ is a partition. Hence:

$$\langle \mathbf{Y}_1, \mathbf{Y}_2 \rangle = \sum_{P \in \mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\pi'})} \langle \mathbf{Y}_1[P, C_P], \mathbf{Y}_2[P, C_P] \rangle. \tag{C.9}$$

By Definition C.2 and Remark C.3, $\mathbf{X}[R_P, P]$ is orthonormal by column for any $P \in \mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\pi'})$, so:

$$\begin{aligned}
&\langle (\mathbf{X}\mathbf{Y}_1)[R_P, C_P], (\mathbf{X}\mathbf{Y}_2)[R_P, C_P] \rangle \\
&\quad \overset{(C.1)}{=} \langle \mathbf{X}[R_P, P]\mathbf{Y}_1[P, C_P], \mathbf{X}[R_P, P]\mathbf{Y}_2[P, C_P] \rangle \\
&\quad = \mathrm{Tr}(\mathbf{Y}_1[P, C_P]^\top \mathbf{X}[R_P, P]^\top \mathbf{X}[R_P, P]\mathbf{Y}_2[P, C_P]) \\
&\quad = \mathrm{Tr}(\mathbf{Y}_1[P, C_P]^\top \mathbf{Y}_2[P, C_P]) \\
&\quad = \langle \mathbf{Y}_1[P, C_P], \mathbf{Y}_2[P, C_P] \rangle,
\end{aligned} \tag{C.10}$$

where Tr denotes the trace operator. This yields:

$$\langle \mathbf{X}\mathbf{Y}_1, \mathbf{X}\mathbf{Y}_2 \rangle \overset{(C.8)+(C.10)}{=} \sum_{P \in \mathcal{P}(\mathbf{S}_\pi, \mathbf{S}_{\pi'})} \langle \mathbf{Y}_1[P, C_P], \mathbf{Y}_2[P, C_P] \rangle \overset{(C.9)}{=} \langle \mathbf{Y}_1, \mathbf{Y}_2 \rangle,$$

which ends the proof. □

We have a similar result to Lemma C.4, but for row-orthonormal butterfly factors.

Lemma C.5. *Consider a chainable pair (π', π). Then, for any $\mathbf{X}_1, \mathbf{X}_2 \in \Sigma^{\pi'}$, and any $\mathbf{Y} \in \Sigma^{\pi}$ that is $q(\pi', \pi)$-row-orthonormal, we have:*

$$\langle \mathbf{X}_1\mathbf{Y}, \mathbf{X}_2\mathbf{Y} \rangle = \langle \mathbf{Y}_1, \mathbf{Y}_2 \rangle.$$

The following lemmas are consequences of Lemma C.4 and Lemma C.5.

Lemma C.6. *Consider a chainable architecture $\beta := (\pi_1, \pi_2, \pi_3)$. Then, for any $\mathbf{X} \in \Sigma^{\pi_1}$ that is $q(\pi_1, \pi_2)$-column-orthonormal, any $\mathbf{Y} \in \Sigma^{\pi_2}$, and any $\mathbf{Z} \in \Sigma^{\pi_3}$ that is $q(\pi_2, \pi_3)$-row-orthonormal, we have:*

$$\langle \mathbf{X}\mathbf{Y}_1\mathbf{Z}, \mathbf{X}\mathbf{Y}_2\mathbf{Z} \rangle = \langle \mathbf{Y}_1, \mathbf{Y}_2 \rangle$$

Proof. By Lemma 7.2, $\mathbf{Y}_i\mathbf{Z} \in \Sigma^{\pi_2 * \pi_3}$ for $i \in \{1, 2\}$, and by Lemma 7.3, $(\pi_1, \pi_2 * \pi_3)$ is chainable with $q(\pi_1, \pi_2 * \pi_3) = q(\pi_1, \pi_2)$. Therefore, by Lemma C.4: $\langle \mathbf{X}\mathbf{Y}_1\mathbf{Z}, \mathbf{X}\mathbf{Y}_2\mathbf{Z} \rangle = \langle \mathbf{Y}_1\mathbf{Z}, \mathbf{Y}_2\mathbf{Z} \rangle$. Applying Lemma C.5 this time with π_2 and π_3 yields $\langle \mathbf{Y}_1\mathbf{Z}, \mathbf{Y}_2\mathbf{Z} \rangle = \langle \mathbf{Y}_1, \mathbf{Y}_2 \rangle$. □

Lemma C.7. *Under the same assumptions as Lemma C.6: $\|\mathbf{X}\mathbf{Y}\mathbf{Z}\|_F = \|\mathbf{Y}\|_F$.*

Proof. This is immediate by taking $\mathbf{Y}_1 = \mathbf{Y}_2 = \mathbf{Y}$ in Lemma C.6. □

Stable under matrix multiplication

Similarly to classical orthonormal matrices, orthonormal butterfly factors also enjoy a form of stability under matrix multiplication, in the following sense.

Lemma C.8. *Consider a chainable pair (π_1, π_2).*

1. *If $\mathbf{A}_1 \in \Sigma^{\pi_1}$ is $q(\pi_1, \pi_2)$-column-orthonormal and $\mathbf{A}_2 \in \Sigma^{\pi_2}$ is q_2-column-orthonormal for some integer q_2, then the product $\mathbf{A}_1\mathbf{A}_2 \in \Sigma^{\pi_1 * \pi_2}$ is q_2-column-orthonormal.*
2. *If $\mathbf{A}_1 \in \Sigma^{\pi_1}$ is q_1-row-orthonormal for some integer q_1 and $\mathbf{A}_2 \in \Sigma^{\pi_2}$ is $q(\pi_1, \pi_2)$-row-orthonormal, then the product $\mathbf{A}_1\mathbf{A}_2 \in \Sigma^{\pi_1 * \pi_2}$ is q_1-row-orthonormal.*

The proof of Lemma C.8 needs the two following lemmas.

Lemma C.9. *Let $\mathbf{A} \in \Sigma^{\pi}$ be a q-column-orthonormal butterfly factor, with $\pi := (a, b, c, d)$. For any subset of column indices of the form $I_t = [\![(t-1)dq + 1, tdq]\!]$ for $t \in [\![ac/q]\!]$, the submatrix $\mathbf{A}[:, I_t]$ is a column-orthonormal matrix.*

Proof. By Lemma C.2, $\mathcal{P}_c(\pi, q) = \{I_{t,k}\}_{t,k}$ as defined in (C.6). Fix $t \in [\![a]\!]$. As stated in (C.7), $I_t = \bigcup_{k \in [\![d]\!]} I_{t,k}$, where $\{I_{t,k}\}_k$ are pairwise disjoint. Because of the structure $\mathbf{S}_\pi = \mathbf{I}_a \otimes \mathbf{1}_{b \times c} \otimes \mathbf{I}_d$, for any $k, k' \in [\![d]\!]$ such that $k \neq k'$ and $i \in I_{t,k}$, $i' \in I_{t,k'}$, the column supports $\mathbf{S}_\pi[:, i]$ and $\mathbf{S}_\pi[:, i']$ are pairwise disjoint. Therefore, a fortiori, if i, i' are two indices belonging to two different $I_{t,k}$, $I_{t,k'}$ for $k \neq k'$, then $\mathbf{A}[:, i]$ and $\mathbf{A}[:, i']$ are orthogonal. And if i, i' belong to the same $I_{t,k}$ for a given k, then $\mathbf{A}[:, i]$ and $\mathbf{A}[:, i']$ are also orthogonal, and they are of unit norm, since $\mathbf{A}[:, I_{t,k}]$ is an orthonormal matrix by column. Thus, $\mathbf{A}[:, I_t]$ is a column-orthonormal matrix. □

Lemma C.10. *Consider a pattern π_2 and an integer q_2 such that $\mathcal{P}_c(\pi_2, q_2)$ is well-defined. Then, for any π_1 such that (π_1, π_2) is chainable, $\mathcal{P}_c(\pi_1 * \pi_2, q_2)$ is also well-defined and $\mathcal{P}_c(\pi_2, q_2) = \mathcal{P}_c(\pi_1 * \pi_2, q_2)$.*

Proof. Let us show that $\mathcal{P}_c(\pi_1 * \pi_2, q_2)$ is well-defined. Since $\mathcal{P}_c(\pi_2, q_2)$ is well-defined, by Definition C.1, there exists a pattern π such that (π_2, π) is chainable with $q(\pi_2, \pi) = q_2$. In particular, this means that the architecture (π_1, π_2, π) is chainable, and by Lemma 7.5, $(\pi_1 * \pi_2, \pi)$ is chainable with $q(\pi_1 * \pi_2) = q_2$. By Definition C.1, $\mathcal{P}_c(\pi_1 * \pi_2, q_2)$ is well-defined. The equality $\mathcal{P}_c(\pi_2, q_2) = \mathcal{P}_c(\pi_1 * \pi_2, q_2)$ can be verified using (C.6) in Lemma C.2 and (7.11) in Definition 7.4. □

Proof for Lemma C.8. We will prove the claim for the column-orthonormal case. The case with row-orthonormal factors can be dealt with similarly. Let $P \in \mathcal{P}_c(\pi_1 * \pi_2, q_2)$, and let us show that $(\mathbf{A}_1\mathbf{A}_2)[:, P]$ is orthonormal by column. By Lemma C.10 and Lemma C.2:

$$\begin{aligned}\mathcal{P}_c(\pi_1 * \pi_2, q_2) &= \mathcal{P}_c(\pi_2, q_2) \\ &= \left\{ \{k + (t-1)d_2q_2 + (j-1)d_2\}_{j \in [\![q_2]\!]} \mid (t,k) \in [\![d_2]\!] \times [\![a_2c_2/q_2]\!] \right\},\end{aligned}$$

so there exists a pair $(t,k) \in [\![d_2]\!] \times [\![a_2c_2/q_2]\!]$ such that $P = \{\{k + (t-1)d_2q_2 + (j-1)d_2\}_{j \in [\![q_2]\!]}$. Then, we remark that:

$$P \subseteq [\![(t-1)d_2q_2 + 1, td_2q_2]\!] \subseteq [\![uc_2d_2 + 1, (u+1)c_2d_2]\!] := J_u, \tag{C.11}$$

where u is the quotient in the Euclidean division of $t-1$ by c_2/q_2, with $0 \leq u \leq a_2 - 1$, because $0 \leq t-1 \leq a_2c_2/q_2 - 1$. By the structure $\mathbf{S}_{\pi_2} = \mathbf{I}_{a_2} \otimes \mathbf{1}_{b_2 \times c_2} \otimes \mathbf{I}$, this means that the support of each column of $\mathbf{S}_{\pi_2}$ indexed by $P \subseteq J_u$ is included in $I_u := [\![ub_2d_2 + 1, (u+1)b_2d_2]\!]$. Therefore:

$$(\mathbf{A}_1\mathbf{A}_2)[:, P] = \mathbf{A}_1(\mathbf{A}_2[:, P]) = \mathbf{A}_1[:, I_u]\mathbf{A}_2[I_u, P]$$

On the one hand, the submatrix $\mathbf{A}_2[I_u, P]$ is orthonormal by column since $P \in \mathcal{P}_c(\pi_2, q_2)$ and $\mathbf{A}_2$ is a q_2-column-orthonormal butterfly factor. On the other hand, by chainability of (π_1, π_2), denoting $q_1 = q(\pi_1, \pi_2)$, we have $b_2d_2/d_1 = q_1$, and $a_2 = a_1c_1/q_1$. In other words, $I_u = [\![ud_1q_1 + 1, (u+1)d_1q_1]\!]$ for any $u \in [\![0, a_1c_1/q_1 - 1]\!]$. By Lemma C.9, $\mathbf{A}_1[:, I_u]$ is orthonormal by column. In conclusion, $(\mathbf{A}_1\mathbf{A}_2)[:, P]$ is orthonormal by column, because it is the product of two orthonormal matrices by column, which ends the proof. □

C.3.3 Explanations for Algorithm 7.6

We are now ready to precisely describe the goal of Algorithm 7.6, in light of the following lemma.

Lemma C.11. *Consider a non-redundant chainable pair* $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$*, and* $(\mathbf{X}_1, \mathbf{X}_2) \in \Sigma^{\boldsymbol{\pi}_1} \times \Sigma^{\boldsymbol{\pi}_2}$*. Denote* $q := q(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$*. Then:*

- *Algorithm 7.6 with input* $u = \texttt{column}$ *returns* $(\tilde{\mathbf{X}}_1, \tilde{\mathbf{X}}_2) \in \Sigma^{\boldsymbol{\pi}_1} \times \Sigma^{\boldsymbol{\pi}_2}$ *such that* $\tilde{\mathbf{X}}_1$ *is a* q*-column-orthonormal and* $\tilde{\mathbf{X}}_1\tilde{\mathbf{X}}_2 = \mathbf{X}_1\mathbf{X}_2$*;*
- *Algorithm 7.6 with input* $u = \texttt{row}$ *returns* $(\tilde{\mathbf{X}}_1, \tilde{\mathbf{X}}_2) \in \Sigma^{\boldsymbol{\pi}_1} \times \Sigma^{\boldsymbol{\pi}_2}$ *such that* $\tilde{\mathbf{X}}_2$ *is a* q*-row-orthonormal and* $\tilde{\mathbf{X}}_1\tilde{\mathbf{X}}_2 = \mathbf{X}_1\mathbf{X}_2$*.*

Proof. We only prove the first point, as the second point can be addressed similarly. For any $(\tilde{\mathbf{X}}_1, \tilde{\mathbf{X}}_2) \in \Sigma^{\boldsymbol{\pi}_1} \times \Sigma^{\boldsymbol{\pi}_2}$, since $\mathrm{supp}(\mathbf{X}_1\mathbf{X}_2)$ and $\mathrm{supp}(\tilde{\mathbf{X}}_1\tilde{\mathbf{X}}_2)$ are both included in $\bigcup_{P \in \mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}_1}, \mathbf{S}_{\boldsymbol{\pi}_2})} R_P \times C_P$ where $\{R_P \times C_P\}_P$ are pairwise disjoint, we have:

$$\begin{aligned}\mathbf{X}_1\mathbf{X}_2 = \tilde{\mathbf{X}}_1\tilde{\mathbf{X}}_2 &\iff \forall P \in \mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}_1}, \mathbf{S}_{\boldsymbol{\pi}_2}), \quad (\mathbf{X}_1\mathbf{X}_2)[R_P, C_P] = (\tilde{\mathbf{X}}_1\tilde{\mathbf{X}}_2)[R_P, C_P] \\ &\iff \forall P \in \mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}_1}, \mathbf{S}_{\boldsymbol{\pi}_2}), \quad (\mathbf{X}_1\mathbf{X}_2)[R_P, C_P] = \tilde{\mathbf{X}}_1[R_P, P]\tilde{\mathbf{X}}_2[P, C_P],\end{aligned}$$

where the second equivalence comes by Lemma C.1 and by chainability of $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$. Moreover, by Remark C.3, $\tilde{\mathbf{X}}_1 \in \Sigma^{\boldsymbol{\pi}_1}$ is q-column-orthonormal if, and only if, $\tilde{\mathbf{X}}_1[R_P, P]$ is orthonormal by column for each $P \in \mathcal{P}_c(\boldsymbol{\pi}_1, q) = \mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}_1}, \mathbf{S}_{\boldsymbol{\pi}_2})$.

The output $(\tilde{\mathbf{X}}_1, \tilde{\mathbf{X}}_2)$ of Algorithm 7.6 is constructed by computing the QR-decomposition of $\mathbf{X}[R_P, P] = \mathbf{QR}$, and setting $\tilde{\mathbf{X}}[R_P, P] = \mathbf{Q}$, $\tilde{\mathbf{Y}}[P, C_P] = \mathbf{R}\mathbf{Y}_2[P, C_P]$, which is possible because $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ is assumed to be non-redundant. Thus, $\tilde{\mathbf{X}}[R_P, P]$ is orthonormal by column by construction, and $\tilde{\mathbf{X}}[R_P, P]\tilde{\mathbf{Y}}[P, C_P] = \mathbf{QR}\mathbf{Y}_2[P, C_P] = \mathbf{X}_1[R_P, P]\mathbf{Y}_2[P, C_P] = (\mathbf{X}_1\mathbf{X}_2)[R_P, C_P]$. This yields the claim of the lemma. □

In other words, Algorithm 7.6 construct orthonormal submatrices. Recalling the notations form Definition 7.1, if $u = \texttt{column}$, then the submatrices $\tilde{\mathbf{X}}[R_P, P]$ for $P \in \mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}_1}, \mathbf{S}_{\boldsymbol{\pi}_2})$ in line 5 of Algorithm 7.6 become orthonormal by column. Indeed, they are set to $\mathbf{Q}$ where $(\mathbf{Q}, \mathbf{R})$ is the QR-decomposition[2] of submatrices of $\mathbf{X}[R_P, P]$. The orthonormality by column of $\tilde{\mathbf{X}}[R_P, P]$ is very important in the proof of our error bound of Algorithm 7.5. Likewise, if $u = \texttt{row}$, then the submatrices $\tilde{\mathbf{Y}}[P, C_P]$ for $P \in \mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}_1}, \mathbf{S}_{\boldsymbol{\pi}_2})$ in line 14 become orthonormal by row.

C.3.4 Proof of (7.27)

We use Lemma C.11 and the following result.

[2]In this context, a QR-decomposition of a matrix $\mathbf{A} \in \mathbb{C}^{m \times n}, m \geq n$ is a pair of matrices $(\mathbf{Q}, \mathbf{R})$ in which $\mathbf{Q} \in \mathbb{C}^{m \times n}$ is orthonormal by column and $\mathbf{R}^{n \times n}$ is an upper triangular matrix.

Lemma C.12. *For a given iteration $J \in [\![L-1]\!]$ in Algorithm 7.5, recall the notation $\mathbf{X}_{left}^{(J)}$ and $\mathbf{X}_{right}^{(J)}$ given at (7.22) and (7.23), and denote (r,t) as defined in line 9 in Algorithm 7.5. Then:*

1. $\mathbf{X}_{left}^{(J)} \in \Sigma^{\pi_1 * \ldots * \pi_{r-1}}$ *is $q(\pi_{r-1}, \pi_r)$-column-orthnormal;*
2. $\mathbf{X}_{right}^{(J)} \in \Sigma^{\pi_{t+1} * \ldots * \pi_L}$ *is $q(\pi_t, \pi_{t+1})$-row-orthonormal.*

Proof. Fix $J \in [\![L-1]\!]$. Denote $(\tilde{\mathbf{X}}_{P_k})_{k=1}^{J}$ the factors before and after the orthnormalization operations (lines 10-15), with $(P_k)_{k=1}^{J}$ the elements in `partition` as defined in line 4 and updated at line 7. Denote:

$$\forall k \in [\![J]\!], \quad \pi_{P_k} := \pi_{r'} * \ldots * \pi_{t'} \quad \text{with} \quad (r', t') := (\min P_k, \max P_k). \tag{C.12}$$

By Lemma C.11, after the first "for loop" (lines 10-12), due to the assignments in line 11 of Algorithm 7.5, each butterfly factor $\mathbf{X}_{P_k} \in \Sigma^{\pi_{P_k}}$ for $k \in [\![j-1]\!]$ becomes $q(\pi_{P_k}, \pi_{P_{k+1}})$-column-orthonormal. Consequently, by Lemma C.8, the product $\mathbf{X}_{\text{left}}^{(J)} = \mathbf{X}_{P_1} \ldots \mathbf{X}_{P_{j-1}} \in \Sigma^{\pi_{P_1} * \ldots * \pi_{P_{j-1}}}$ is $q(\pi_{P_{j-1}}, \pi_{P_j})$-column-orthonormal. By Lemma 7.5 and by construction of $(P_k)_{k=1}^{J}$, we have $\pi_{P_1} * \ldots * \pi_{Pj-1} = \pi_1 * \ldots * \pi_{r-1}$, and $q(\pi_{P_{j-1}}, \pi_{P_j}) = q(\pi_{r-1}, \pi_r)$, which yields the first claim. The second claim is obtained similarly. □

Proof of (7.27). Consider two cases:

1. if $J = 1$, then we are in the case where $(r,t) = (1, L)$, so by convention given after (7.22) and (7.23), $\mathbf{X}_{\text{left}}^{(1)}$ and $\mathbf{X}_{\text{right}}^{(1)}$ are both the identity matrices. The equality holds trivially in this case.
2. If $J > 1$, by Lemma C.12, $\mathbf{X}_{\text{left}}^{(J)}$ is $q(\pi_{r-1}, \pi_r)$-column-orthnormal and $\mathbf{X}_{\text{right}}^{(J)}$ is $q(\pi_t, \pi_{t+1})$-row-orthonormal. By Lemma 7.5 and chainability of $\boldsymbol{\beta} = (\pi_\ell)_{\ell=1}^{L}$, $(\pi_1 * \ldots * \pi_{r-1}, \pi_r * \ldots * \pi_t)$ is chainable with $q(\pi_1 * \ldots * \pi_{r-1}, \pi_r * \ldots * \pi_t) = q(\pi_{r-1}, \pi_r)$, and $(\pi_r * \ldots * \pi_t, \pi_{t+1} * \ldots * \pi_L)$ is chainable with $q(\pi_r * \ldots * \pi_t, \pi_{t+1} * \ldots * \pi_L) = q(\pi_t, \pi_{t+1})$. Moreover, by the construction of Algorithm 7.5, both $\mathbf{X}_{[\![r,t]\!]}$ and $\mathbf{X}_{[\![r,s]\!]}\mathbf{X}_{[\![s+1,t]\!]}$ belong to $\Sigma^{(\pi_r * \ldots * \pi_t)} = \Sigma^{\pi_{P_j}}$. Therefore, the proof for this case can be concluded using Lemma C.7.

□

C.4 Complexity of hierarchical algorithms

This section is devoted to prove Theorem 7.2. We analyze the complexity of each of the components involved in the proposed hierarchical algorithms.

C.4.1 Complexity of Algorithm 7.1

This algorithm essentially performs several low-rank approximations that are typically computing using truncated SVD.

Lemma C.13. *Consider* $(\mathbf{L}, \mathbf{R})$ *that satisfy the condition of Theorem 7.1. Then, for any matrix* $\mathbf{A}$*, the complexity of Algorithm 7.1 with inputs* $(\mathbf{A}, \mathbf{L}, \mathbf{R})$ *is*

$$\mathcal{O}\left(\sum_{P\in\mathcal{P}(\mathbf{L},\mathbf{R})}|P||R_P||C_P|\right).$$

Remark C.4. *In practice, best low-rank approximations in Algorithm 7.1 via truncated SVDs can be computed in parallel. This can decrease the complexity up to*

$$\mathcal{O}(\max_{P\in\mathcal{P}(\mathbf{L},\mathbf{R})}|P||R_P||C_P|)$$

when parallelizing across $|\mathcal{P}(\mathbf{L}, \mathbf{R})|$ *processes.*

Proof. The algorithm performs the best rank-$|P|$ approximation of a submatrix of size $|R_P| \times |C_P|$ for each $P \in \mathcal{P}(\mathbf{L}, \mathbf{R})$ and the complexity of the truncated SVD at order k for an $m \times n$ matrix is $\mathcal{O}(kmn)$ [153]. □

We apply this complexity analysis to the case where $(\mathbf{L}, \mathbf{R})$ are butterfly supports corresponding to a chainable pair of patterns.

Lemma C.14. *For any chainable pair of patterns* $(\boldsymbol{\pi}, \boldsymbol{\pi})$ *with* $\boldsymbol{\pi} = (a, b, c, d)$ *and* $\boldsymbol{\pi}' = (a', b', c', d')$*, denoting* $q := q(\boldsymbol{\pi}, \boldsymbol{\pi}')$*, the complexity of Algorithm 7.1 with inputs* $(\mathbf{A}, \mathbf{S}_{\boldsymbol{\pi}}, \mathbf{S}_{\boldsymbol{\pi}})$ *is*

$$C(\boldsymbol{\pi}, \boldsymbol{\pi}') = \mathcal{O}(qa'c'bd). \tag{C.13}$$

Proof. By chainability of $(\boldsymbol{\pi}, \boldsymbol{\pi})$, we have $q = ac/a'$. By Lemmas 7.8 and C.13, $C(\boldsymbol{\pi}, \boldsymbol{\pi}') = \mathcal{O}\left(\frac{acd}{q}|P||R_P||C_P|\right)$. By Lemma 7.2, $C(\boldsymbol{\pi}, \boldsymbol{\pi}') = \mathcal{O}(acdbc')$. Since $q = ac/a'$ by Definition 7.4, $C(\boldsymbol{\pi}, \boldsymbol{\pi}') = \mathcal{O}(qa'c'bd)$. □

C.4.2 Complexity of Algorithms 7.3 and 7.4

These two algorithms are based on Algorithm 7.1.

Lemma C.15. *Consider a non-redundant chainable architecture* β *and a matrix* $\mathbf{A}$ *of size* $m \times n$*. With the same notations as in Theorem 7.2, the complexity of Algorithms 7.3 and 7.4 with inputs* β*,* $\mathbf{A}$ *and any factor-bracketing tree* $\mathcal{T}$ *is at most:*

- $\mathcal{O}(\|\mathbf{q}(\beta)\|_1 M_\beta N_\beta)$ *in the general case;*
- $\mathcal{O}(\|\mathbf{q}(\beta)\|_1 mn)$ *if* β *is non-redundant.*

Proof. Since Algorithm 7.3 performs $(L-1)$ factorizations of the form of Problem (7.15) using Algorithm 7.1, its complexity is equal to the sum of the complexity of each of these $(L-1)$ factorizations.

Fix $1 \leq r \leq s < t \leq L$. By Lemma 7.5, $q_s := q(\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t) = q(\boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1})$. By (7.14):

$$\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s = \left(a_r, \frac{b_r d_r}{d_s}, \frac{a_s c_s}{a_r}, d_s\right),$$
$$\boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t = \left(a_{s+1}, \frac{b_{s+1} d_{s+1}}{d_t}, \frac{a_t c_t}{a_{s+1}}, d_t\right).$$

Therefore, by Lemma C.14: $C(\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t) = \mathcal{O}(q_s a_t c_t b_r d_r)$, which is upper bounded by $\mathcal{O}(q_s M_\beta N_\beta)$, by definition of M_β, N_β. Therefore, the overall complexity of Algorithm 7.3/Algorithm 7.4 is upper bounded by:

$$\sum_{s=1}^{L-1} q_s M_\beta N_\beta = \|\mathbf{q}(\beta)\|_1 M_\beta N_\beta. \tag{C.14}$$

Moreover, if β is assumed to be non-redundant, then by Definition 7.6, we have $a_\ell c_\ell < a_{\ell+1} c_{\ell+1}$ and $b_\ell d_\ell > b_{\ell+1} d_{\ell+1}$ for all $\ell \in [\![L-1]\!]$, hence, $a_1 c_1 < \ldots < a_L c_L$ and $b_1 d_1 > \ldots > b_L d_L$. Consequently:

$$\forall \ell \in [\![L-1]\!], \quad \begin{cases} a_\ell c_\ell < a_L c_L = n/d_L \leq n \\ b_\ell d_\ell < b_1 d_1 = m/a_1 \leq m \end{cases}. \tag{C.15}$$

Thus, we have: $M_\beta = \max_{\ell \in [\![L]\!]} a_\ell c_\ell \leq m$. Similarly, $N_\beta \leq n$, which ends the proof. □

C.4.3 Complexity of Algorithm 7.6

We now analyze the complexity of the construction of orthonormal butterfly factors in Algorithm 7.6.

Lemma C.16. *The complexity of Algorithm 7.6 with input $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \mathbf{X}, \mathbf{Y}, u)$ for any $u \in \{\texttt{column}, \texttt{row}\}$ and any non-redundant chainable pair $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ is*

$$\mathcal{O}(q(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)(\|\boldsymbol{\pi}_1\|_0 + \|\boldsymbol{\pi}_2\|_0)),$$

where we recall the notation from Lemma 7.1.

Proof. We only consider the case $u = \texttt{column}$, since the other case can be dealt with similarly. Denote $q = q(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)$ and $\boldsymbol{\pi}_\ell = (a_\ell, b_\ell, c_\ell, d_\ell)$ for $\ell \in \{1, 2\}$. At each iteration of Algorithm 7.6:

- the complexity of line 4 is $\mathcal{O}(|R_P||P|^2) = \mathcal{O}(b_1 q^2)$, since the complexity of QR-decomposition of a matrix of size $m \times n$ is $\mathcal{O}(\min(m,n)mn)$ [131, Section 5.2];

- the complexity of line 5 is $\mathcal{O}(|R_P||P|) = \mathcal{O}(b_1 q)$;
- the complexity of line 6 is $\mathcal{O}(|P|^2|C_P|) = \mathcal{O}(c_2 q^2)$.

Overall, the complexity of each iteration is $\mathcal{O}(q^2(b_1 + c_2))$. Since there are $a_1c_1d_1/q = a_2b_2d_2/q$ equivalence classes, the over complexity is:

$$\begin{aligned}\mathcal{O}\left(\frac{a_1c_1d_1}{q}q^2b_1 + \frac{a_2b_2d_2}{q}q^2c_2\right) &= \mathcal{O}(q(a_1b_1c_1d_1 + a_2b_2c_2d_2)) \\ &= \mathcal{O}(q(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2)(\|\boldsymbol{\pi}_1\|_0 + \|\boldsymbol{\pi}_2\|_0)).\end{aligned}$$

□

C.4.4 Complexity of Algorithm 7.5

We prove the second point of Theorem 7.2.

Proof. Assume that β is not redundant. By Lemma C.16 and the inequality in (C.15), the complexity for each call of Algorithm 7.6 in lines 11 and 14 of Algorithm 7.5 is bounded by $\mathcal{O}(mn\|\mathbf{q}(\beta)\|_\infty)$. At the J-th iteration for some $J \in [\![L-1]\!]$, there are at most $(J-1)$ calls of (7.6), so the total complexity for the orthonormalization operations accross all the $|\beta| - 1$ iterations is at most $\mathcal{O}(|\beta|^2 mn\|\mathbf{q}(\beta)\|_\infty)$. Therefore, the complexity of Algorithm 7.5 is given by

$$\mathcal{O}\left((|\beta|^2\|\mathbf{q}(\beta)\|_\infty + \|\mathbf{q}(\beta)\|_1)mn\right).$$

□

C.4.5 Complexity of Algorithm 7.7

Proof of the third point of Theorem 7.2. Denote β' the output of Algorithm 7.2 with input β. Compared to Algorithm 7.5, Algorithm 7.7 performs Algorithm 7.5 on β' (if $|\beta'| > 1$) and recovers original factors by using Algorithm 7.1.

If $|\beta'| = 1$, then we simply set $\mathbf{X}'_1 = \mathbf{A} \odot \mathbf{S}_{\pi_1 * \ldots * \pi_L}$. Thus, the complexity of this step is at most $O(mn)$. Otherwise, the complexity of Algorithm 7.5 applied to β' is given by:

$$\mathcal{O}\left((|\beta'|^2\|\mathbf{q}(\beta')\|_\infty + \|\mathbf{q}(\beta')\|_1)mn\right).$$

Thanks to Proposition 7.2, we have: $|\beta'| \leq |\beta|$ and $\mathbf{q}(\beta')$ is a sub-vector of $\mathbf{q}(\beta)$. As a consequence, $\|q(\beta')\|_1 \leq \|\mathbf{q}(\beta)\|_1, \|q(\beta')\|_\infty \leq \|\mathbf{q}(\beta)\|_\infty$. Thus,

$$(|\beta'|^2\|q(\beta')\|_\infty + \|q(\beta')\|_1)mn \leq (|\beta|^2\|\mathbf{q}(\beta)\|_\infty + \|\mathbf{q}(\beta)\|_1)mn.$$

In both cases ($|\beta'| = 1$ and $|\beta'| > 1$), the complexity of finding $(\mathbf{X}'_\ell)$ is at most $O\left((|\beta|^2\|\mathbf{q}(\beta)\|_\infty + \|\mathbf{q}(\beta)\|_1)mn\right)$.

To obtain the factors $(\mathbf{X}_\ell)_{\ell=1}^{L}$ from $(\mathbf{X}'_\ell)_{\ell=1}^{L'}$, we need to use Algorithm 7.1 on support constraints of the form $(\mathbf{S}_{\pi_r * \ldots * \pi_s}, \mathbf{S}_{\pi_{s+1} * \ldots * \pi_t})$. This is exactly similar to the analysis of Lemma C.15. Using the same argument, we can bound the complexity of this step by: $O(\|\mathbf{q}(\beta)\|_1 M_\beta N_\beta)$. The proof is concluded by taking the sum of the two bounds. □

C.5 Proof for results in Section 7.7

C.5.1 Proof for (7.28)

The proof is based on the following result.

Lemma C.17. *Consider a non-redundant chainable architecture $\beta := (\boldsymbol{\pi}_\ell)_{\ell=1}^{L}$, and integers (r, s, t) such that $1 \leq r \leq s < t \leq L$. Denote $\mathbf{X}$ any butterfly factor in $\Sigma^{\pi_1 * \ldots * \pi_{r-1}}$ that is $q(\boldsymbol{\pi}_{r-1}, \boldsymbol{\pi}_r)$-column-orhtonormal if $r > 1$, otherwise $\mathbf{X}$ is the identity matrix of size $a_1 b_1 d_1$. Denote $\mathbf{Z}$ any butterfly factor in $\Sigma^{\pi_{t+1} * \ldots * \pi_L}$ that is $q(\boldsymbol{\pi}_t, \boldsymbol{\pi}_{t+1})$-column-orhtonormal if $t < L$, otherwise $\mathbf{Z}$ is the identity matrix of size $a_L c_L d_L$. Then, for any $\mathbf{Y} \in \Sigma^{\pi_r * \ldots * \pi_t}$:*

$$E^{\beta_s}(\mathbf{XYZ}) = \inf_{\mathbf{Y}_1, \mathbf{Y}_2} \{\|\mathbf{Y} - \mathbf{Y}_1 \mathbf{Y}_2\|_F \mid (\mathbf{Y}_1, \mathbf{Y}_2) \in \Sigma^{\pi_r * \ldots * \pi_s} \times \Sigma^{\pi_{s+1} * \ldots * \pi_t}\}. \quad \text{(C.16)}$$

Proof of (7.28). By Lemma C.12, $\mathbf{X}_{\text{left}}^{(J)} \in \Sigma^{\pi_1 * \ldots * \pi_{r-1}}$ is $q(\boldsymbol{\pi}_{r-1}, \boldsymbol{\pi}_r)$-column-orthnormal, and $\mathbf{X}_{\text{right}}^{(J)} \in \Sigma^{\pi_{t+1} * \ldots * \pi_L}$ is $q(\boldsymbol{\pi}_t, \boldsymbol{\pi}_{t+1})$-row-orthonormal. There are two cases to consider on the value of $J \in [\![L-1]\!]$:

- If $J = 1$, then $(r, t) = (1, L)$, so $\mathbf{X}_{\text{left}}^{(1)}, \mathbf{X}_{\text{right}}^{(1)}$ are simply identity matrices, and the claim of the lemma is true by Definition 7.7.
- Otherwise, $J > 1$, and $\mathbf{X}_{[\![r,t]\!]} \in \Sigma^{\pi_r * \ldots * \pi_t}$. This means that we can apply Lemma C.17 with $(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) = (\mathbf{X}_{\text{left}}^{(J)}, \mathbf{X}_{[\![r,t]\!]}, \mathbf{X}_{\text{right}}^{(J)})$. This proves (7.28) by remarking that $(\mathbf{X}_{[\![l,s]\!]}, \mathbf{X}_{[\![s+1,r]\!]})$ are indeed an optimal solution to the problem in the right-hand side of (C.16).

□

Before proving Lemma C.17, we need a technical result.

Corollary C.1. *Consider a chainable pair of patterns $(\boldsymbol{\pi}, \tilde{\boldsymbol{\pi}})$. For any $\mathbf{A} \in \Sigma^{\boldsymbol{\pi}}$ that is $q(\boldsymbol{\pi}, \tilde{\boldsymbol{\pi}})$-column-orthonormal, the submatrix $\mathbf{A}[:, T_k]$ where $T_k = \text{supp}(\mathbf{S}_{\tilde{\pi}}[:, k])$ is an orthonormal matrix by column, for any column index k of $\mathbf{S}_{\tilde{\pi}}$.*

Proof. Denote $\boldsymbol{\pi} = (a, b, c, d)$ and $\tilde{\boldsymbol{\pi}} = (\tilde{a}, \tilde{b}, \tilde{c}, \tilde{d})$. By the structure $\mathbf{S}_{\tilde{\boldsymbol{\pi}}} = \mathbf{I}_{\tilde{a}} \otimes \mathbf{1}_{\tilde{b}\times\tilde{c}} \otimes \mathbf{I}_{\tilde{d}}$, one can verify that T_k is a subset of an interval $I_t := [\![1 + (t-1)\tilde{b}\tilde{d}, t\tilde{b}\tilde{d}]\!] = [\![1 + (t-1)dq, tdq]\!]$ for a certain $t \in [\![\tilde{a}]\!]$, where the equality $\tilde{b}\tilde{d} = dq$ comes from the chainability of $(\boldsymbol{\pi}, \tilde{\boldsymbol{\pi}})$. By Lemma C.9, $\mathbf{A}[:, I_k]$ is an orthonormal matrix by column, so $\mathbf{A}[:, T_k]$ is also orthonormal by column.

□

Proof for Lemma C.17. Denote $\mathbf{S}_{r,t} := S_{\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_t}$ for any $1 \le r \le t \le L$, and $q_s := q(\boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1})$. Let us compute the left-hand side (LHS) and the right-hand side (RHS) of (C.16) separately.

- **LHS:** By Lemma 7.8, $\mathbf{S}_{\boldsymbol{\pi}_1 * \ldots \boldsymbol{\pi}_s}$ and $\mathbf{S}_{\boldsymbol{\pi}_1 * \ldots \boldsymbol{\pi}_s}$ satisfy Theorem 7.1. Moreover, by Lemma 7.5, $q(\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_L) = q_s$. Hence, by (C.4):

$$E^{\beta_s}(\mathbf{XYZ})^2 = \underbrace{\|\mathbf{XYZ}\|_F^2}_{=\|\mathbf{Y}\|_F^2} - \sum_{P \in \mathcal{P}(\mathbf{S}_{1,s}, \mathbf{S}_{s+1,L})} \sum_{k=1}^{q_s} \sigma_k^2((\mathbf{XYZ})[R_P^1, C_P^1])$$

  where we denote $R_P^1 := \mathrm{supp}(\mathbf{S}_{1,s}[:, i])$ and $C_P^1 := \mathrm{supp}(\mathbf{S}_{s+1,L}[i, :])$ for any $i \in P$, and $\sigma_k(\cdot)$ is the k-th largest eigenvalue of a matrix. The equality $\|\mathbf{XYZ}\|_F^2 = \|\mathbf{Y}\|_F^2$ comes by orthonormality of the butterfly factor $\mathbf{X}$, $\mathbf{Z}$ and Lemma C.7.

- **RHS:** Similarly, $\inf\{\|\mathbf{Y} - \mathbf{Y}_1\mathbf{Y}_2\|_F \mid (\mathbf{Y}_1, \mathbf{Y}_2) \in \Sigma^{\boldsymbol{\pi}_r * \ldots \boldsymbol{\pi}_s} \times \Sigma^{\boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_t}\}$ is equal to

$$\|\mathbf{Y}\|_F^2 - \sum_{P \in \mathcal{P}(\mathbf{S}_{r,s}, \mathbf{S}_{s+1,t})} \sum_{k=1}^{q_s} \sigma_k^2(\mathbf{Y}[R_P^2, C_P^2]),$$

  where we denote $R_P^2 := \mathrm{supp}(\mathbf{S}_{r,s}[:, i])$, $C_P^2 := \mathrm{supp}(\mathbf{S}_{s+1,t}[i, :])$ for any $i \in P$.

We now remark that $\mathcal{P}(\mathbf{S}_{1,s}, \mathbf{S}_{s+1,L}) = \mathcal{P}(\mathbf{S}_{r,s}, \mathbf{S}_{s+1,t}) = \mathcal{P}_c(\boldsymbol{\pi}_s, q_s)$. Indeed, on the one hand:

$$\begin{aligned} \mathcal{P}(\mathbf{S}_{1,s}, \mathbf{S}_{s+1,L}) &= \mathcal{P}_c(\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_s, q(\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_s, \boldsymbol{\pi}_{s+1} * \ldots * \boldsymbol{\pi}_L)) \\ &= \mathcal{P}_c(\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_s, q_s) \\ &= \mathcal{P}_c(\boldsymbol{\pi}_s, q_s) \end{aligned} \tag{C.17}$$

by Definition C.1, Lemma 7.5 and Lemma C.10. And on the other hand, we can show similarly that $\mathcal{P}(\mathbf{S}_{l,s}, \mathbf{S}_{s+1,r}) = \mathcal{P}_c(\boldsymbol{\pi}_s, q_s)$.

Therefore, in order to finish the proof, it is sufficient to prove that the spectrum (the set of nonzero eigenvalues, including their multiplicities) of $\mathbf{Y}[R_P^2, C_P^2]$ is identical to the one of $(\mathbf{XYZ})[R_P^1, C_P^1]$, for each $P \in \mathcal{P}_c(\boldsymbol{\pi}_s, q_s)$. By assumption, $\mathrm{supp}(\mathbf{Y}) \subseteq \mathbf{S}_{r,t} = \bigcup_{P \in \mathcal{P}_c(\boldsymbol{\pi}_s, q_s)} R_P^2 \times C_P^2$ where $\{R_P^2 \times C_P^2\}_P$ are pairwise disjoint. Thus:

$$\mathbf{XYZ} = \sum_{P \in \mathcal{P}_c(\boldsymbol{\pi}_s, q_s)} \mathbf{X}[:, R_P^2]\mathbf{Y}[R_P^2, C_P^2]\mathbf{Z}[C_P^2, :]. \tag{C.18}$$

Looking at the support of each term for $P \in \mathcal{P}_c(\boldsymbol{\pi}_s, q_s)$ in the sum, we have:

$$\begin{aligned} & \operatorname{supp}(\mathbf{X}[:, R_P^2]\, \mathbf{Y}[R_P^2, C_P^2]\, \mathbf{Z}[C_P^2, :]) \\ \subseteq\ & \operatorname{supp}(\mathbf{S}_{1,r-1}[:, R_P^2]\, \mathbf{1}_{|R_P^2| \times |C_P^2|}\, \mathbf{S}_{t+1,L}[C_P^2, :]) \\ =\ & \operatorname{supp}(\mathbf{S}_{1,r-1}[:, R_P^2]\, \mathbf{1}_{|R_P^2| \times 1}\, \mathbf{1}_{1 \times |C_P^2|}\, \mathbf{S}_{t+1,L}[C_P^2, :]). \end{aligned}$$

Note that for any $i \in P$, we have $\mathbf{S}_{1,r-1}[:, R_P^2]\mathbf{1}_{|R_P^2| \times 1} = \mathbf{S}_{1,r-1}\mathbf{S}_{r,s}[:, i] = \mathbf{S}_{1,s}[:, i]$ and $\mathbf{1}_{1 \times |C_P^2|}\mathbf{S}_{t+1,L}[C_P^2, :] = \mathbf{S}_{s+1,t}[i, :]\mathbf{S}_{t+1,L} = \mathbf{S}_{s+1,L}[i, :]$. Therefore:

$$\operatorname{supp}(\mathbf{X}[:, R_P^2]\mathbf{Y}[R_P^2, C_P^2]\mathbf{Z}[C_P^2, :]) \subseteq \operatorname{supp}(\mathbf{S}_{1,s}[:, i]\mathbf{S}_{s+1,L}[i, :]) = R_P^1 \times C_P^1.$$

As a consequence, the supports of the summands in the right-hand side of (C.18) are pairwise disjoint. Thus, for any $P \in \mathcal{P}_c(\boldsymbol{\pi}_s, q_s)$, the product $\mathbf{X}[:, R_P^2]\mathbf{Y}[R_P^2, C_P^2]\mathbf{Z}[C_P^2, :]$ is equal to $\begin{pmatrix} (\mathbf{XYZ})[R_P^1, C_P^1] & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}$ up to some permutation of rows and columns. This means that $\mathbf{X}[:, R_P^2]\mathbf{Y}[R_P^2, C_P^2]\mathbf{Z}[C_P^2, :]$ and $(\mathbf{XYZ})[R_P^1, C_P^1]$ share the same spectrum.

But by Corollary C.1, $\mathbf{X}[:, R_P^2]$ is orthonormal by column, because $(\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_{r-1}, \boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s)$ is chainable with $q(\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_{r-1}, \boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_s) = q(\boldsymbol{\pi}_{r-1}, \boldsymbol{\pi}_r)$ by Lemma 7.5, the butterfly factor $\mathbf{X}$ is assumed to be $q(\boldsymbol{\pi}_{r-1}, \boldsymbol{\pi}_r)$-column orthonormal, and $R_P^2 = \operatorname{supp}(\mathbf{S}_{r,s}[i, :])$ for some $i \in P$ by definition. Similarly, $\mathbf{Z}[C_P^2, :]$ is orthonormal by row. In conclusion, $\mathbf{Y}[R_P^2, C_P^2]$ shares the same spectrum with $\mathbf{X}[:, R_P^2]\mathbf{Y}[R_P^2, C_P^2]\mathbf{Z}[C_P^2, :]$, which itself shares the same spectrum as the submatrix $(\mathbf{XYZ})[R_P^1, C_P^1]$. This ends the proof. □

C.5.2 Proof for (7.30)

Proof. For each $J \in [\![L-1]\!]$, we denote $(\mathbf{X}_{[\![J,J]\!]}, \mathbf{X}_{[\![J+1,L]\!]}, \mathbf{X}_{[\![J,L]\!]})$ the matrices at line 17 of Algorithm 7.5, and reuse the notation $\mathbf{X}_{\text{left}}^{(J)}$ as in (7.22). In particular, this means that

$$\begin{aligned} \forall J \in [\![L-1]\!], \quad \mathbf{P}_J &= \mathbf{X}_{\text{left}}^{(J)}\mathbf{X}_{[\![J,J]\!]}\mathbf{X}_{[\![J+1,L]\!]}, \\ \mathbf{P}_{J-1} &= \mathbf{X}_{\text{left}}^{(J)}\mathbf{X}_{[\![J,L]\!]}. \end{aligned} \tag{C.19}$$

When σ is the identity permutation, the values of `partition` at the J-th iteration ($J \in [\![L-1]\!]$) is $\{[\![1,1]\!], [\![2,2]\!], \ldots, [\![J,J]\!], [\![J+1,L]\!]\}$. Denoting $(\hat{\mathbf{X}}_\ell)_{\ell=1}^L$ the output of Algorithm 7.5, we remark that the list `factors` obtained at the end of the J-th iteration (cf. Line 13 - Algorithm 7.5) is a tuple of the form:

$$(\hat{\mathbf{X}}_1, \ldots, \hat{\mathbf{X}}_{J-1}, \mathbf{X}_{[\![J,J]\!]}, \mathbf{X}_{[\![J+1,L]\!]}). \tag{C.20}$$

Indeed, the value of the first $J-1$ factors in the list `factors` are column-orthonormal butterfly factors (Definition C.2), due to the orthonormalization operations at the

J-th iteration. Therefore, their values during orthonormalization operations in the next iterations $J+1, J+2, \ldots, L-1$ do not change anymore, which means that they are equal to $\hat{\mathbf{X}}_1, \ldots, \hat{\mathbf{X}}_{J-1}$.

Therefore, $\mathbf{X}_{\text{left}}^{(J)} = \prod_{\ell=1}^{J-1} \hat{\mathbf{X}}_\ell$ for any $J \in [\![L-1]\!]$. In particular, for any $p > J$, we have $\mathbf{X}_{\text{left}}^{(t)} = \mathbf{X}_{\text{left}}^{(J)} \hat{\mathbf{X}}_J \ldots \hat{\mathbf{X}}_{t-1}$. Combining this with (C.19) yields

$$\forall J \in [\![L-1]\!], \quad \forall p \geq J, \quad \mathbf{P}_p = \mathbf{X}_{\text{left}}^{(J)} \left(\prod_{\ell=J}^{p-1} \hat{\mathbf{X}}_\ell \right) \mathbf{X}_{[\![p,p]\!]} \mathbf{X}_{[\![p+1,L]\!]}. \tag{C.21}$$

Fix $J \in [\![L-1]\!]$. Denote

$$\begin{aligned} &\mathbf{B}_{J-1} := \mathbf{X}_{[\![J,L]\!]}, \\ &\mathbf{B}_J := \mathbf{X}_{[\![J,J]\!]} \mathbf{X}_{[\![J+1,L]\!]}, \\ &\mathbf{B}_p := \left(\prod_{\ell=J}^{p-1} \mathbf{X}_\ell \right) \mathbf{X}_{[\![p,p]\!]} \hat{\mathbf{X}}_{[\![p+1,L]\!]} \quad \forall p > J. \end{aligned} \tag{C.22}$$

On the one hand, by Lemma C.12, $\mathbf{X}_{\text{left}}^{(J)} \in \Sigma^{(\pi_1 * \ldots * \pi_{J-1})}$ is $q(\boldsymbol{\pi}_{J-1}, \boldsymbol{\pi}_J)$-column-orthonormal. On the other hand, considering $\mathcal{P} := \mathcal{P}(\mathbf{S}_{\boldsymbol{\pi}_J}, \mathbf{S}_{\boldsymbol{\pi}_{J+1} * \ldots * \boldsymbol{\pi}_L})$, we have

$$\begin{aligned} &\operatorname{supp}(\mathbf{S}_{\boldsymbol{\pi}_J * \ldots * \boldsymbol{\pi}_L}) = \bigcup_{P \in \mathcal{P}} R_P \times C_P, \\ &\forall P, P' \in \mathcal{P}, \quad (R_P \times C_P) \cap (R_{P'} \times C_{P'}) = \emptyset. \end{aligned} \tag{C.23}$$

Hence, for any $p \geq J$:

$$\begin{aligned} \langle \mathbf{B}_{J-1} - \mathbf{B}_J, \mathbf{B}_p \rangle = 0 &\overset{(C.22)+(C.19)}{\iff} \langle \mathbf{X}_{\text{left}}^{(J)} (\mathbf{B}_{J-1} - \mathbf{B}_J), \mathbf{X}_{\text{left}}^{(J)} \mathbf{B}_p \rangle = 0 \\ &\overset{\text{Lemma C.4}}{\iff} \langle \mathbf{B}_{J-1} - \mathbf{B}_J, \mathbf{B}_p \rangle = 0 \\ &\overset{(C.23)}{\iff} \sum_{P \in \mathcal{P}} \left\langle \left(\mathbf{B}_{J-1} - \mathbf{B}_J \right) [R_P, C_P], \mathbf{B}_p [R_P, C_P] \right\rangle = 0. \end{aligned}$$

Therefore, for proving (7.30), it is sufficient to show:

$$\langle (\mathbf{B}_{J-1} - \mathbf{B}_J)[R_P, C_P], \mathbf{B}_p[R_P, C_P] \rangle = 0, \quad \forall P \in \mathcal{P}, p \geq J. \tag{C.24}$$

Let $P \in \mathcal{P}$, and denote the (column) range of a matrix $\mathbf{M}$ by $\operatorname{range}(\mathbf{M})$. The proof relies on the following two steps. First, for $t > J$, by (C.22), $\mathbf{B}_t = \mathbf{X}_J \mathbf{C}_t$ for some $\mathbf{C}_t \in \Sigma^{\boldsymbol{\pi}_{J+1} * \ldots * \boldsymbol{\pi}_L}$. By Lemma C.1, $\mathbf{B}_t[R_P, C_P] = \mathbf{X}_J[R_P, P] \mathbf{C}_t[P, C_P]$. Hence, $\operatorname{range}(\mathbf{B}_t[R_P, C_P]) \subseteq \operatorname{range}(\mathbf{X}_J[R_P, P])$. Similarly, we can show $\operatorname{range}(\mathbf{B}_J[R_P, C_P]) \subseteq \operatorname{range}(\mathbf{X}_{[\![J,J]\!]}[R_P, P])$. But $\operatorname{range}(\mathbf{X}_{[\![J,J]\!]}[R_P, P]) = \operatorname{range}(\mathbf{X}_J[R_P, P])$ if and only if $\operatorname{rank}(\mathbf{X}_{[\![J,J]\!]}[R_P, P]) = |P|$, because $\mathbf{X}_{[\![J,J]\!]}$ is orthonormalized into $\mathbf{X}_J$ at the next iteration $J+1$ by design of Algorithm 7.6. Thus, we have:

$$\forall p \geq J, \quad \operatorname{range}(\mathbf{B}_p[R_P, C_P]) \subseteq \operatorname{range}(\mathbf{X}_{[\![J,J]\!]}[R_P, P]). \tag{C.25}$$

provided that $\operatorname{rank}(\mathbf{X}_{[\![J,J]\!]}[R_P, P]) = |P|$.

Second, by definition of $(\mathbf{X}_{[\![J,J]\!]}, \mathbf{X}_{[\![J+1,L]\!]}, \mathbf{X}_{[\![J,L]\!]})$ at line 17 of Algorithm 7.5, the product $(\mathbf{X}_{[\![J,J]\!]}\mathbf{X}_{[\![J+1,L]\!]})[R_P, C_P] = \mathbf{B}_J[R_P, C_P]$ is the best rank-$|P|$ approximation of $\mathbf{X}_{[\![J,L]\!]}[R_P, C_P] = \mathbf{B}_{J-1}[R_P, C_P]$ (cf. line 3 in Algorithm 7.1). This means that the range of the residual $(\mathbf{B}_{J-1} - \mathbf{B}_J)[R_P, C_P]$ is orthogonal to the range of $\mathbf{B}_J[R_P, C_P]$, which is equal to the range of $\mathbf{X}_{[\![J,J]\!]}[R_P, P]$, provided that the rank of $\mathbf{B}_{J-1}[R_P, C_P]$ is at least $|P|$. In this case, we necessarily have $\operatorname{rank}(\mathbf{X}_{[\![J,J]\!]}[R_P, P]) = |P|$, so we obtain (C.24) using (C.25). Otherwise, when the rank of $\mathbf{B}_{J-1}[R_P, C_P]$ is at most $|P| - 1$, the residual $(\mathbf{B}_{J-1} - \mathbf{B}_J)[R_P, C_P]$ is null, so we still obtain (C.24). This ends the proof. □

C.6 On the generalization of the complementary low-rank property

We show in this section that the generalized complementary low-rank property associated with a chainable β given in Definition 7.8 coincides, under some assumption on β, with the classical definition of the complementary low-rank property given in Definition 3.7.

In the classical definition, a cluster tree (Definition 3.3)yields a hierarchical partitioning of a given set of indices. Similarly, the following proposition shows that, under some appropriate conditions, a chainable architecture β also yields a hierarchical partitioning of the row and column indices, which leads to two cluster trees.

Proposition C.1. *Consider a chainable architecture $\beta = (\boldsymbol{\pi}_\ell)_{\ell=1}^L$ where $\boldsymbol{\pi}_\ell := (a_\ell, b_\ell, c_\ell, d_\ell)$ for $\ell \in [\![L]\!]$. Denoting $\mathbf{S}_{r,t} := S_{\boldsymbol{\pi}_r * \ldots * \boldsymbol{\pi}_t}$ for any $1 \leq r \leq t \leq L$ and recalling the notation from Definition 7.1, define for all $\ell \in [\![L-1]\!]$:*

$$\begin{aligned} P_{L-\ell}^{row} &:= \{R_I \,|\, I \in \mathcal{P}(\mathbf{S}_{1,\ell}, \mathbf{S}_{\ell+1,L})\}, \\ P_\ell^{col} &:= \{C_I \,|\, I \in \mathcal{P}(\mathbf{S}_{1,\ell}, \mathbf{S}_{\ell+1,L})\}. \end{aligned} \tag{C.26}$$

Assume that $a_1 = d_L = 1$. Then:

- *$\{P_\ell^{row}\}_{\ell=1}^{L-1}$ and $\{P_\ell^{col}\}_{\ell=1}^{L-1}$ are partitions of $[\![m]\!]$ and $[\![n]\!]$, respectively;*
- *for each $\ell \in [\![L-2]\!]$, $P_{\ell+1}^{row}$ and $P_{\ell+1}^{col}$ are finer than P_ℓ^{col} and P_ℓ^{row}, respectively.*

Consequently, $\{P_\ell^{row}\}_{\ell=1}^{L-1}$ and $\{P_\ell^{col}\}_{\ell=1}^{L-1}$ yield two cluster trees, denoted T_β^{row} and T_β^{col}, of depth $L-1$ with root node $[\![m]\!]$ and $[\![n]\!]$, respectively.

Remark C.5. *For $\ell \in [\![L-2]\!]$, we have $\boldsymbol{\pi}_{\ell+1} * \ldots * \boldsymbol{\pi}_L = \left(a_{\ell+1}, b_{\ell+1}d_{\ell+1}, \frac{a_L c_L}{a_{\ell+1}}, 1\right)$ by (7.14), since $d_L = 1$ by assumption. Therefore, there are $a_{\ell+1}$ nodes of cardinal $\frac{a_L c_L}{a_{\ell+1}}$ at*

level ℓ in T_β^{col}. Each of them have $\frac{a_{\ell+2}}{a_{\ell+1}}$ children, because $a_{\ell+1} \mid a_{\ell+2}$ by chainability of $(\boldsymbol{\pi}_{\ell+1}, \boldsymbol{\pi}_{\ell+2})$. Similarly, there are $d_{\ell+1}$ nodes of cardinal $\frac{b_1 d_1}{d_{\ell+1}}$ at level ℓ in T_β^{row}. Each of them have $\frac{d_{\ell+1}}{d_\ell}$ children.

Proof. We only do the proof for $\{P_\ell^{\text{col}}\}_{\ell=1}^{L-1}$, since the proof is similar for the row partitions $\{P_\ell^{\text{row}}\}_{\ell=1}^{L-1}$. First, fix $\ell \in [\![L-1]\!]$, and we show that P_ℓ^{col} is a partition of $[\![n]\!]$ for any $\ell \in [\![L-1]\!]$, where $n := a_L c_L d_L$ by definition. By (7.14): $\boldsymbol{\pi}_{\ell+1} * \ldots * \boldsymbol{\pi}_L = \left(a_{\ell+1}, \frac{b_{\ell+1} d_{\ell+1}}{d_L}, \frac{a_L c_L}{a_{\ell+1}}, d_L\right) = \left(a_{\ell+1}, b_{\ell+1} d_{\ell+1}, \frac{a_L c_L}{a_{\ell+1}}, 1\right)$ since $d_L = 1$ by assumption. Therefore, by Definition 7.2:

$$P_\ell^{\text{col}} = \{\{k + (j-1)c\}_{k=1}^{c} \mid j \in [\![a_{\ell+1}]\!]\} \quad \text{with} \quad c := \frac{a_L c_L}{a_{\ell+1}}, \tag{C.27}$$

which is indeed a partition of $[\![n]\!]$ since $n = a_L c_L$.

Second, let $\ell \in [\![L-2]\!]$, and let us show that $P_{\ell+1}^{\text{col}}$ is finer than P_ℓ^{col}. Since $(\boldsymbol{\pi}_{\ell+1}, \boldsymbol{\pi}_{\ell+2})$ is chainable, $a_{\ell+1} \mid a_{\ell+2}$. Denoting $\gamma := a_{\ell+2}/a_{\ell+1}$, by (C.27):

$$P_{\ell+1}^{\text{col}} = \left\{ \left\{ k + (j-1)\frac{c}{\gamma} \right\}_{k=1}^{\frac{c}{\gamma}} \mid j \in [\![\gamma a_{\ell+1}]\!] \right\} \quad \text{with} \quad c := \frac{a_L c_L}{a_{\ell+1}}. \tag{C.28}$$

For $j \in [\![a_{\ell+1}]\!]$:

$$\begin{aligned} \{k + (j-1)c\}_{k=1}^{c} &= \bigcup_{i=1}^{\gamma} \left\{ k + (i-1)\frac{c}{\gamma} + (j-1)c \right\}_{k=1}^{\frac{c}{\gamma}} \\ &= \bigcup_{j'=\gamma(j-1)+1}^{\gamma j} \left\{ k + (j'-1)\frac{c}{\gamma} \right\}_{k=1}^{\frac{c}{\gamma}} \end{aligned} \tag{C.29}$$

with the change of variable $j' = \gamma(j-1) + i$ for $i \in [\![\gamma]\!]$. By (C.27), (C.28) and (C.29), we conclude that $P_{\ell+1}^{\text{col}}$ is finer than P_ℓ^{col}. □

Therefore, under the same assumptions of Proposition C.1, the general definition of the complementary low-rank property given in Definition 7.8 coincides with the classical one given in Definition 3.7.

Corollary C.2. *Under the same assumptions of Proposition C.1, for any matrix* $\mathbf{A}$*, the following are equivalent:*

- $\mathbf{A}$ *satisfies the generalized complementary low-rank property (Definition 3.7) associated with* β*;*
- $\mathbf{A}$ *satisfies* exactly *the classical complementary low-rank property (Definition 3.7) for* $(T_\beta^{row}, T_\beta^{col})$ *defined in Proposition C.1, where the rank on the product block partition associated with* $T_\beta^{row}(L-\ell)$ *and* $T_\beta^{col}(\ell)$ *is* $q(\boldsymbol{\pi}_\ell, \boldsymbol{\pi}_{\ell+1})$ *for* $\ell \in [\![L-1]\!]$*.*

Proof. Since β is chainable, by assumption $a_1 = d_L = 1$ and (7.14), we have $\boldsymbol{\pi}_1 * \ldots * \boldsymbol{\pi}_L = (1, m, n, 1)$. Therefore, by Definition 7.8, a matrix $\mathbf{A}$ satisfies the general complementary low-rank property associated with β if, and only if, $\operatorname{rank}(\mathbf{A}[R_P, C_P]) \leq q(\boldsymbol{\pi}_\ell, \boldsymbol{\pi}_{\ell+1})$ for each $P \in \mathcal{P}(\mathbf{S}_{1,\ell}, \mathbf{S}_{\ell+1,L})$ and $\ell \in [\![L-1]\!]$. By Proposition C.1, this is precisely a reformulation of the classical complementary low-rank property for the trees $(T_\beta^{\text{row}}, T_\beta^{\text{col}})$, because by definition, $\mathcal{P}(\mathbf{S}_{1,\ell}, \mathbf{S}_{\ell+1,L})$ is the product block partition associated with $T^{\text{row}}(L-\ell)$ and $T^{\text{col}}(\ell)$ for each $\ell \in [\![L-1]\!]$. □

C.7 Additional experiments

In Figure 7.4b, we observed that the approximation error obtained by the hierarchical algorithm *with* orthonormalization operations (Algorithm 7.5) is always smaller than the noise level $\epsilon = 0.1$, as opposed to the one obtained without orthonormalization. In fact, we have the same observation for any values of $\epsilon \in \{0.01, 0.03, 0.1, 0.3\}$, as shown in Figure C.1.

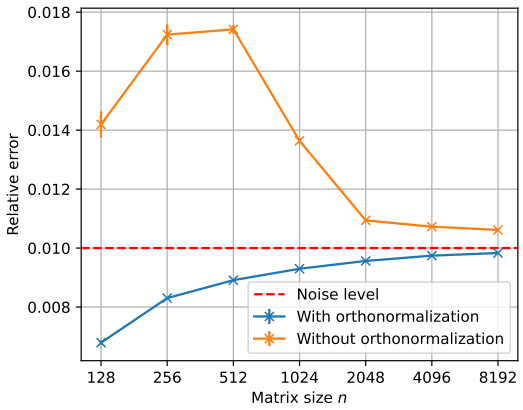
C.8 Is chainability necessary for an error bound of the form (7.3)?

Let us illustrate that the chainability condition is not necessary to obtain an error bound of the form (7.3), by considering the notion of *transpose-chainability*. If $\mathbf{X}$ is a $\boldsymbol{\pi}$-factor where $\boldsymbol{\pi} = (a, b, c, d)$, then $\mathbf{X}^\top$ is a $\boldsymbol{\pi}^\top$-factor where $\boldsymbol{\pi}^\top := (a, c, b, d)$. Therefore, if it is possible to find an approximate solution with an error bound of (7.3) to Problem (7.1) associated with $\beta = (\boldsymbol{\pi}_\ell)_{\ell=1}^L$, then we can also obtain an approximate solution with the same theoretical guarantee to Problem (7.1) associated with $\beta^\top = (\boldsymbol{\pi}_L^\top, \ldots, \boldsymbol{\pi}_1^\top)$. This is typically done by:

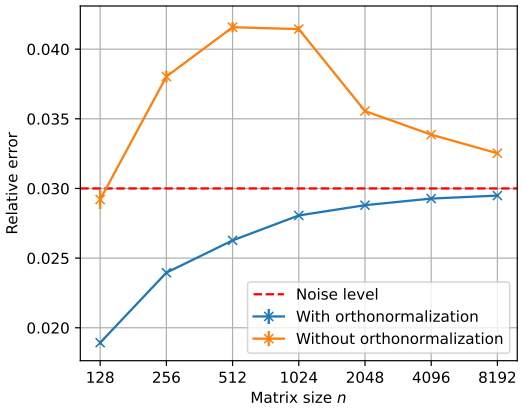
1. finding an approximate solution $(\mathbf{X}_1, \ldots, \mathbf{X}_L) \in \Sigma^\beta$ to (7.1) for β and $\mathbf{A}^\top$;
2. constructing $(\mathbf{X}_L{}^\top, \ldots, \mathbf{X}_1{}^\top) \in \Sigma^{\beta^\top}$, since $(\mathbf{X}_1 \ldots \mathbf{X}_L)^\top = \mathbf{X}_L{}^\top \ldots \mathbf{X}_1{}^\top$.

Since the error bound (7.3) is guaranteed for any chainable β, as shown in Corollary 7.1, it is also guaranteed for any *transpose-chainable* architecture, defined as an architecture $\beta^\top$ such that β is chainable. In general, a chainable architecture is not transpose-chainable, and vice-versa, as shown in the following example:

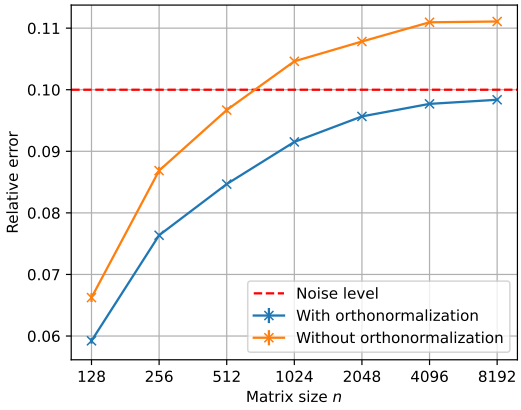
Example C.1. *The pair $(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2) = ((1,2,2,2),(2,2,2,1))$ is chainable but not transpose-chainable, and the pair $(\boldsymbol{\pi}_2{}^\top, \boldsymbol{\pi}_1{}^\top)$ is transpose-chainable but not chainable.*



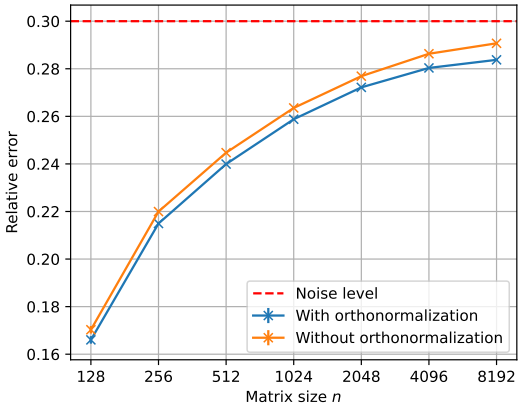
(a) $\epsilon = 0.01$



(b) $\epsilon = 0.03$



(c) $\epsilon = 0.1$



(d) $\epsilon = 0.3$

Figure C.1: Relative approximation errors *vs.* the matrix size n, for Algorithm 7.4 (without orthonormalization) and Algorithm 7.5 (with orthonormalization), for the instance of Problem (7.1) described in Section 7.8.2 with $r = 4$. We show mean and standard deviation on the error bars over 10 repetitions of the experiment.

Appendix D

Appendices for Chapter 9

D.1 Details on the experiments

All the experiments of Section 9.5 (Figure D.2, Figure 9.4, Table 9.1, Figure 9.6, Figure D.3, Figure 9.5) are done on a single NVIDIA A100-PCIE-40GB GPU on an Intel(R) Xeon(R) Silver 4215R CPU @ 3.20GHz with 377G of memory. The version of `pytorch` and `pytorch-cuda` are 2.0.1 and 11.7.

Time measurements are done using the tool `torch.utils.benchmark.Timer` from PyTorch. The medians are computed on at least 10 measurements of 10 runs. All the interquartile range (IQR) are at least 3 times smaller than the median.

In all our experiments for matrix multiplication, we set the batch size to $K = 128 \times 196 = 25088$ for the input matrix $\mathbf{X} \in \mathbb{R}^{n \times K}$, a choice typically aligned with the ViT architecture, where this quantity corresponds to the number of tokens per sequence (192) multiplied by the number of sequences in a batch of images (128). When dealing with a batch of images in neural networks, we choose as batch size $K = 128$. The nonzero entries of any butterfly factor $\mathbf{B} \in \mathbb{R}^{abd \times acd}$ with sparsity pattern (a, b, c, d) are drawn i.i.d. uniformly in $[-\frac{1}{\sqrt{c}}, \frac{1}{\sqrt{c}}]$, as for the initialization chosen for training in [72], while the entries of the inputs $\mathbf{X} \in \mathbb{R}^{n \times K}$ are drawn i.i.d. according to a standard normal distribution $\mathcal{N}(0, 1)$.

D.2 Reproduction of butterfly sparse neural networks found in the literature

Table D.1 shows the result of our reproduction of different setups of the literature [72, 241], for training a butterfly sparse ViT for ImageNet classification [78]. We observe that: (i) butterfly sparse neural networks indeed have accuracies comparable to their dense counterparts; and that (ii) in *half-precision*, the forward pass is slower with butterfly matrices compared to its dense counterpart.

Table D.1: Results of our reproduction of butterfly sparse neural networks of the literature. The accuracies correspond to the ImageNet validation set, after training on the ImageNet training set using the experiment protocol of the original papers [25, 72, 241]. The time of a single forward pass is measured for a batch size of 128, in *half-precision*. We use the implementation provided in each paper.

	Parameters	Accuracy Top-1 / Top-5	Forward pass (ms)
ResNet-50			
Dense	25.6 M	76.2 / 93.0	27.1
DeBut [241]	13.4 M	74.2 / 92.1	101.7
Simple ViT-S/16 [25]			
Dense	21.9 M	75.5/92.0	14.2
Monarch [72]	9.6 M	73.2/91.1	23.3
Simple ViT-B/16 [25]			
Dense	86.4 M	75.9/91.7	36
Monarch [72]	36.8 M	75.4/91.7	49.3

D.3 Time spent in linear layers in vision transformers

Table D.2 shows that at least between 31% and 61% of the time spent on the forward pass of a vision transformer is dedicated to the computation of fully-connected layers. Note that this is *only a lower bound* since we only estimate the time of fully-connected layers in the feed-forward network modules and we do not include those in the multi-head attention module in our estimation. The details are given below.

Estimating the time for fully-connected layers in transformer. The transformer architecture is composed of a sequence of transformer blocks, where each block contains a multi-head attention module and a feed-forward network module. The feed-forward network module is an MLP with one hidden layer, involving two fully-connected layers. We propose the following methodology to estimate the time spent on computing the output of these fully-connected layers during a forward pass of the transformer architecture. We extract all the fully-connected layers appearing in feed-forward network modules of the considered transformer, and stack them sequentially without the biases to obtain an MLP without activations between the layers. We measure the forward time of the obtained MLP, and compare it to the total forward time of the transformer network. The forward time of this MLP is our estimation of the time effectively spent on computing fully-connected layers in the feed-forward network modules of the transformer.

Table D.2: Median execution times (ms) of the forward pass in a ViT, and the forward pass in an MLP containing only all the linear layers involved in the feed-forward network modules of the ViT. The latter is reported with its ratio over the first. FP16 is *half-precision*, FP32 is *float-precision*.

Architecture	fp16 (s)		fp32 (s)	
	Complete	Linear in FFNs	Complete	Linear in FFNs
ViT-S/16	0.014	0.0046 (31%)	0.090	0.04 (46%)
ViT-B/16	0.036	0.015 (42%)	0.30	0.16 (54%)
ViT-L/16	0.11	0.050 (46%)	1.0	0.58 (58%)
ViT-H/14	0.31	0.16 (53%)	2.6	1.6 (61%)

Experimental settings. The architecture ViT-S/16 corresponds to the one in [376], while the architecture ViT-B/16, ViT-L/16 and ViT-H/14 correspond to those in [87]. Input images are of size 224×224. In *float-precision*, the PyTorch implementation of ViT architecture are taken from `https://github.com/lucidrains/vit-pytorch/blob/main/vit_pytorch/simple_vit.py`. In *half-precision*, the considered implementation of the transformer architecture uses FlashAttention [73] to compute the scaled dot product attention[1]. The MLP containing only the linear layers of the feed-forward modules in the transformer architecture is implemented using `torch.nn.Sequential` and `torch.nn.Linear`. Experiments are done on a single A100-40GB GPU on AMD EPYC 7742 64-Core Processor. Measurements are done using the PyTorch tool `torch.utils.benchmark.Timer` for benchmarking. The image batch size is set at 128.

Results. Table D.2 shows that, across various ViT sizes, the proportion of the computation time solely dedicated to linear layers in feed-forward network modules varies between 31% and 53% in *half-precision*, and 46% and 61% in *float-precision*. This proportion increases with the architecture size. In conclusion, there is a non-negligeable part of the computation dedicated to fully-connected layers during the forward pass of a ViT. Note that the time for the fully-connected layers in the multi-head attention module are not taken into account in our measurements, so our estimation is *only a lower bound* on the time effectively dedicated to all fully-connected layers in transformer architectures.

[1] like in `https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html`

Table D.3: Median execution times (ms) of the unfolding operations for the convolutional layers appearing in the last convolutional layers in a ResNet-50 architecture, which are indicated by the name of the corresponding bottleneck block using the notations from the original paper [163], and by their kernel size and their number of output channels. This time is compared to the time of a complete forward pass of the convolutional layer using the default PyTorch implementations.

Bottleneck name	Kernel size, output channels	*Half-precision*	
		`torch.nn.Conv2d` (ms)	Unfolding (ms)
`conv5_1`	1×1 kernel, 512	0.30	1.1
	3×3 kernel, 512	0.13	0.99
	1×1 kernel, 2048	0.17	0.65
`conv5_2` or `conv5_3`	1×1 kernel, 512	0.15	0.73
	3×3 kernel, 512	0.21	1.2
	1×1 kernel, 2048	0.17	0.65

Bottleneck name	Kernel size, output channels	*Float-precision*	
		`torch.nn.Conv2d` (ms)	Unfolding (ms)
`conv5_1`	1×1 kernel, 512	0.46	1.2
	3×3 kernel, 512	0.19	1.0
	1×1 kernel, 2048	0.28	0.67
`conv5_2` or `conv5_3`	1×1 kernel, 512	0.30	0.77
	3×3 kernel, 512	0.35	1.2
	1×1 kernel, 2048	0.28	0.67

D.4 Unfolding convolutional layers

This section explains why the current existing way to insert a butterfly factorization in convolutional layers [241] is not satisfactory.

For a convolutional layer, there is no interest in imposing butterfly sparsity on the matrix of a 2D-kernel $\mathbf{K}$ corresponding to a given pair of input and output channels. Indeed, it is typically of very small dimension (7×7 in ResNets). Instead, [241] proposes to replace another matrix $\mathbf{W}$ which is *deduced* from the 2D-kernel matrices between each pair of input and output channels, by concatenations. The output of the convolutional layer can be computed in three steps:

(i) unfold the input $\mathbf{X}$ into $\tilde{\mathbf{X}}$, involving data copy and reshaping operations;

(ii) computing the matrix multiplication $\tilde{\mathbf{Y}} := \mathbf{W}\tilde{\mathbf{X}}$, where $\mathbf{W}$ is constrained to be a butterfly matrix;

(iii) folding $\tilde{\mathbf{Y}}$ into $\mathbf{Y}$, involving reshaping operations.

Results. In practice, we find in Table D.3 that *the single unfolding operation of step (i)*, as currently implemented in [241], *is already slower than the whole forward pass of the convolutional layer* when it is done in a standard way. Consequently, this unfolding operation is a bottleneck that prevents this approach to accelerate a convolutional layer using the butterfly structure. More details on the experimental settings are given below.

Experimental settings. The unfolding operation of step (i) is implemented using `torch.nn.functional.unfold`, as in the implementation provided by [241]. Table D.3 measures the time for these unfolding operations for the last convolutional layers of a ResNet-50 architecture [163], as proposed in [241]. This time is compared to the time for performing directly the computation of the convolutional layer using the default implementation `torch.nn.Conv2d`. Experiments are done on a single A100-40GB GPU on AMD EPYC 7742 64-Core Processor. The image batch size is set at 128.

D.5 Sparse versus dense GPU matrix multiplication algorithms on PyTorch

Figure D.1 compares different possible implementations in PyTorch for multiplying $\mathbf{Y} = \mathbf{W}\mathbf{X} \in \mathbb{R}^{n\times K}$ for a sparse or dense matrix $\mathbf{W} \in \mathbb{R}^{n\times n}$ and an input matrix $\mathbf{X} \in \mathbb{R}^{n\times K}$. The Pytorch tensor `weight` saves the entries of $\mathbf{W}$ in row-major, and the tensor `weight_transpose` saves the entries of $\mathbf{W}^\top$ in row-major. The entries of $\mathbf{X}$ are saved either in a tensor `input_bsf` of shape (K, n) in the *batch-size-first* setting, or in a tensor `input_bsl` of shape (n, K) in the *batch-size-last* setting. Below are the details of the experiments.

Choice for the matrices X and W. The matrix $\mathbf{W}$ has random values drawn i.i.d. uniformly in $[0, 1]$, and $\mathbf{X}$ has i.i.d. entries according to a standard normal distribution $\mathcal{N}(0, 1)$. When $\mathbf{W}$ is taken to be sparse, the tensor `weight` is stored in the *CSR format* (via $\mathbf{W}$`.to_csr()`) after 95% of its coefficients have been set to zero (with the location of these coefficients chosen uniformly at random). The matrix dimension n is taken to be every power of two between 16 and 8192. The batch size is set to $K = 25088 = 128 \times 196$.

Benchmarked implementations. The PyTorch implementations that are benchmarked are:

- `torch.matmul(weight, input_bsl)`, denoted as $W@X$ in the legend of Figure D.1;
- `torch.matmul(input_bsf,weight_transpose)` denoted as $X@W$ in the legend of Figure D.1;

- `torch.matmul(input_bsf, torch.t(weight))` denoted as $X@T^\top$ in the legend of Figure D.1;
- `torch.matmul(weight, torch.t(input_bsf))` denoted as $W@X^\top$ in the legend of Figure D.1;
- `torch.nn.functional.linear(input_bsf, weight)` denoted as `F.linear`(X, W) in the legend of Figure D.1.

Details on the measurements. The experiments are done on a single A100-40GB GPU and AMD EPYC 7742 64-Core Processor. Measurements are done in *half-precision* and *float-precision*. The IQR are at least 100 times smaller than the median times, and we report the medians.

Results.

(i) The fastest sparse algorithm is `torch.matmul(weight,input_bsl)`, where `input_bsl` is of shape (n, K). This is the only implementation that *cannot be used in a neural network on PyTorch*. Indeed, most of the PyTorch operations assumes that the batch dimensions of the input tensor are at the first positions. This means that any multiplication coming after another PyTorch operation will receive an input with the batch size as the first dimension, which excludes the use of this implementation. The reason why `torch.matmul(weight,input_bsl)` is the fastest sparse implementation may be because of memory layouts. We cannot be certain of anything since the cuSparse routine is not public.

(ii) All the dense implementations have similar times of execution.

(iii) Both in *half-precision* and *float-precision*, the fastest sparse implementation is one order of magnitude faster than the other implementations.

(iv) While the fastest sparse implementation is comparable to the dense implementations in *float-precision*, and even slightly better for n large enough, it is one order slower than the dense implementation in *half-precision*. Once again, the code is not public so we cannot know why for certain, but this may be because the dense implementation leverages TensorCores in *half-precision* while the sparse implementation cannot because of the sparse structure. TensorCores, only available in *half-precision*, is typically very efficient when there are dense sub-blocks of size 16×16.

D.6 Pseudo-code for `monarch` algorithm

Algorithm D.1 and Algorithm D.3 are pseudo-codes corresponding to the algorithms in [72] in the cases $a = 1$ and $d = 1$, respectively, in the *batch-size-first*. Our adaptation to *batch-size-last* is straightforward and is not detailed here.

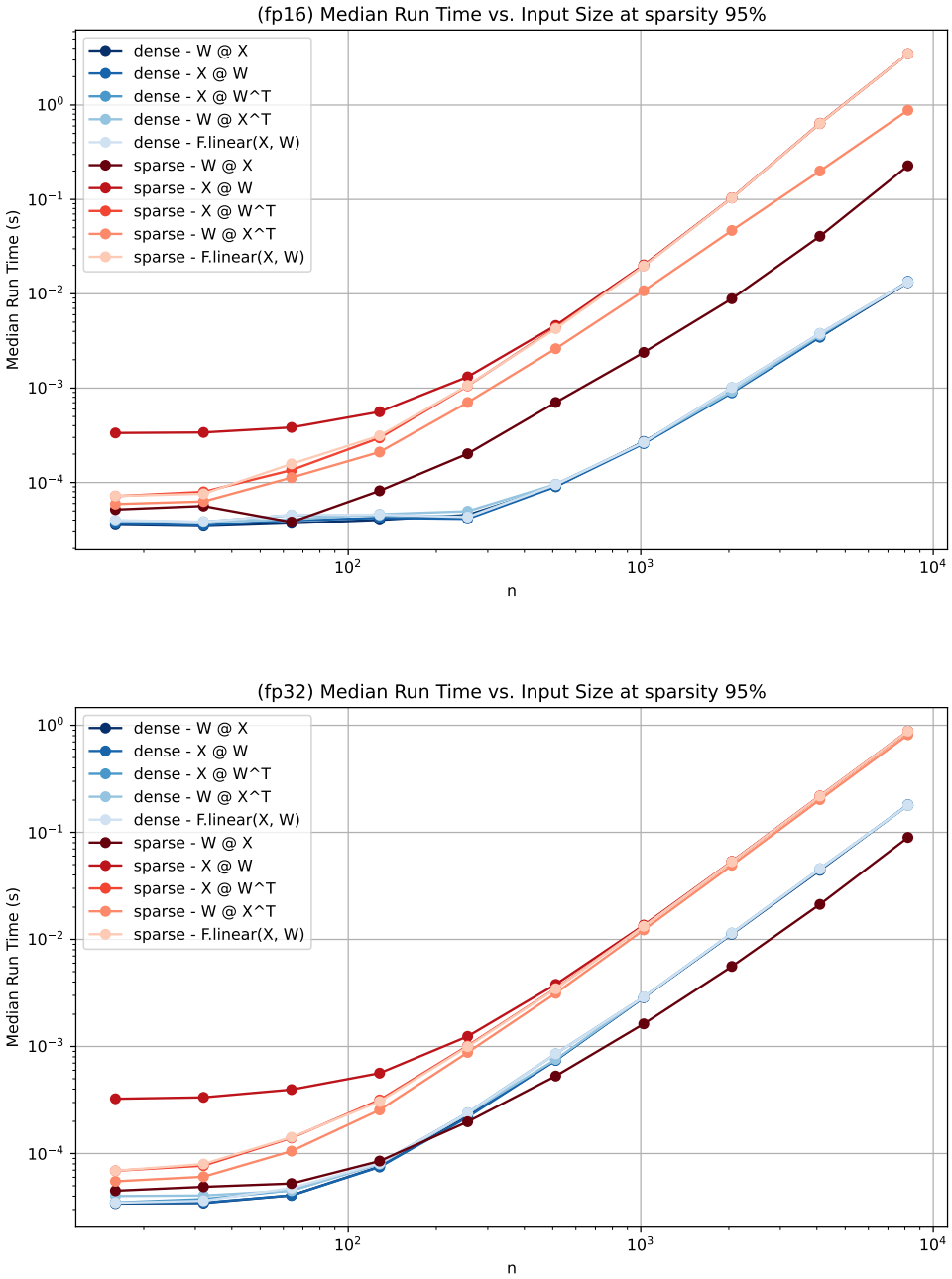


Figure D.1: The experiments are described in Appendix D.5. When **W** is taken to be sparse, the legend indicates “sparse", and "dense" otherwise. The times in *half-precision* (fp16) and *float-precision* (fp32) are reported on the top and bottom graph, respectively.

D.7 Details on the kernel implementation

The proposed implementation use *vectorization* as soon as an operation can be vectorized. Concretely, the `float4` and `half2` vector types are used to mutualize

Algorithm D.1 `monarch` algorithm for the multiplication by a π-butterfly factor with $\pi = (1, b, c, d)$ (case $a = 1$, in the *batch-size-first* setting)

Require: Tensor `blocks` of shape (d, b, c), tensor `input_bsf` of shape (K, cd)
Ensure: Tensor `output_bsf` of shape (K, bd)
`output_bsf` $\leftarrow$ Algorithm D.2 $(c, d,$ `input_bsf`$)$
`output_bsf` $\leftarrow$ Algorithm D.3 (`blocks`, `output_bsf`)
`output_bsf` $\leftarrow$ Algorithm D.2 $(b, d,$ `input_bsf`$)$
return `output_bsf`

Algorithm D.2 Perfect shuffle permutation $\mathbf{P}_{p,q}$, cf. (9.1) in Lemma 9.1, in the *batch-size-first* setting

Require: $p \in \mathbb{N}^*$, $q \in \mathbb{N}^*$, `tensor` of size (K, pq)
Ensure: Shuffled `tensor` of size (K, pq)
1: $r \leftarrow pq$
2: `tensor` $\leftarrow$ reshape `tensor` into size (K, p, q).
3: `tensor` $\leftarrow$ transpose `tensor` into size (K, q, p).
4: `tensor` $\leftarrow$ reshape `tensor` into size (K, r)
5: **return** `tensor`

read and write operations [29, 283–285]. An *epilogue* [283] is also implemented to avoid writing in global memory in a disorganized way. Indeed, after having accumulated the output in registers, each thread has specific rows and columns of the output to write to global memory, and may finish its computation before the others. To avoid that, the epilogue starts to write in the shared memory, in a disorganized way, and then organize the writing from shared to global memory. Another implemented optimization is *double buffering* [29, 225, 283, 284]: a thread block is always both computing the output of a tile, and loading the next tile from global to shared memory. This allows us to hide the latency of loading from the global memory.

D.8 Execution times in *batch-size-first* versus *batch-size-last* in *half-precision*.

Figure D.2 contains the boxplots of the ratios $\frac{\text{time of } \textit{batch-size-first}}{\text{time of } \textit{batch-size-last}}$ in *half-precision* for each algorithm studied in Section 9.5, complementing the information provided in *float-precision* in Figure 9.4.

D.9 Asymmetry $a = 1$ versus $d = 1$

Figure D.3 shows that the sparse and dense algorithms have the same execution times for sparsity patterns $(x, b, c, 1)$ (case $d = 1$) and $(1, b, c, x)$ (case $a = 1$). This

Algorithm D.3 `monarch` algorithm for the multiplication by a π-butterfly factor with $\pi = (a, b, c, 1)$ (case $d = 1$, in the *batch-size-first* setting)

Require: Tensor `blocks` of shape (a, b, c), tensor `input_bsf` of shape (K, ac)
Ensure: Tensor `output_bsf` of shape (K, ab)
  `output_bsf` $\leftarrow$ reshape `input_bsf` into size (a, K, c)
  `output_bsf` $\leftarrow$ `torch.bmm(output_bsf, blocks)`
  `output_bsf` $\leftarrow$ reshape `output_bsf` of size (a, K, b) into size (K, ab)
  return `output_bsf`

is expected since these algorithms are agnostic to the sparsity pattern (a, b, c, d), and going from the tuple $(x, b, c, 1)$ to $(1, b, c, x)$ preserves the dimension of the matrices and the number of nonzeros. This constrasts with the implementations specialized to butterfly sparsity, for which we observed an asymmetry between $a = 1$ and $d = 1$ in Figure 9.6.

D.10 Benchmarking butterfly matrices (Section 9.6.1)

Details on the tested architectures. Denote Π the set of sparsity patterns $\pi = (a, b, c, d)$ for which the multiplication of an associated butterfly factor is benchmarked in Section 9.5. For a given size $m \times n$ in Table 9.2, we benchmark all the butterfly architectures (π_1, π_2) satisfying the three conditions:

1. $\pi_1, \pi_2 \in \Pi$,
2. π_1 corresponds to a factor with output dimension equal to m, and π_2 corresponds to a factor with input dimension equal to n, and
3. the total number of nonzero in the sparsity pattern π_1 and π_2 does not exceed nm.

This yields 27 architectures for the size 384×384, 54 architectures for the sizes 1536×384 and 384×1536, 71 architectures for the size 768×768, 90 architectures for the sizes 3072×768 and 768×3072, and 24 architectures for each of the sizes 1024×1024, 4096×1024 and 1024×4096.

Side note: time for successive multiplications is the sum of individual times. The sequential sparse algorithm (`sequential`) time is closely aligned with the sum of times for each sparse factor. For `monarch`, the median ratio of the time for `sequential` over the sum of the times for each factor is 0.91 (max 1.02) across almost 2300 tested cases; for `kernel`, it is 1.01 (max 1.05).

Results for `hydrib`. Table D.4 reports the comparison of `hybrid` *vs.* the baselines `monarch` and `dense` for the multiplication $\mathbf{X} \mapsto \mathbf{B}_1\mathbf{B}_2\mathbf{X}$ where we use `kernel` for

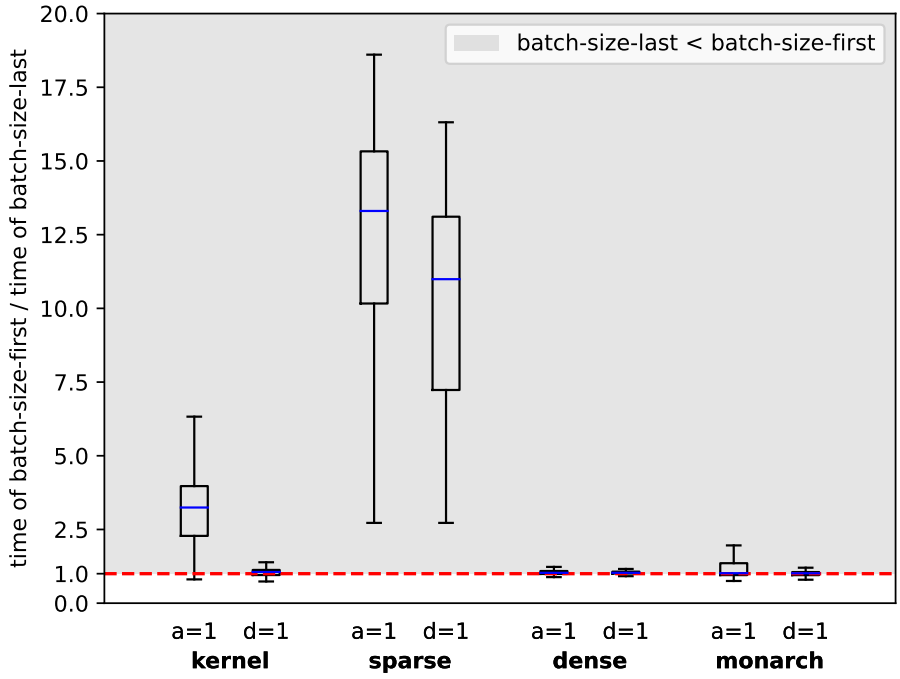


Figure D.2: Boxplots of the ratios $\frac{\text{time of } \textit{batch-size-first}}{\text{time of } \textit{batch-size-last}}$ in 600 cases in *half-precision*.

$\mathbf{B}_1$ (sparsity pattern with the case $a = 1$) and `monarch` for $\mathbf{B}_2$ (sparsity pattern with the case $d = 1$).

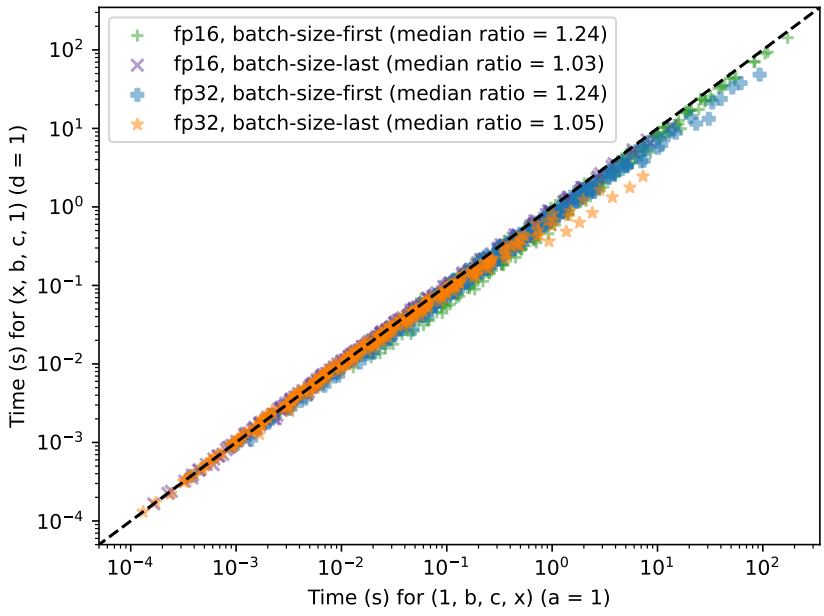
D.11 Implementing *batch-size-last* for different PyTorch operations

To what extent PyTorch operations can be implemented in *batch-size-last*? For this discussion, we consider the following PyTorch operations:

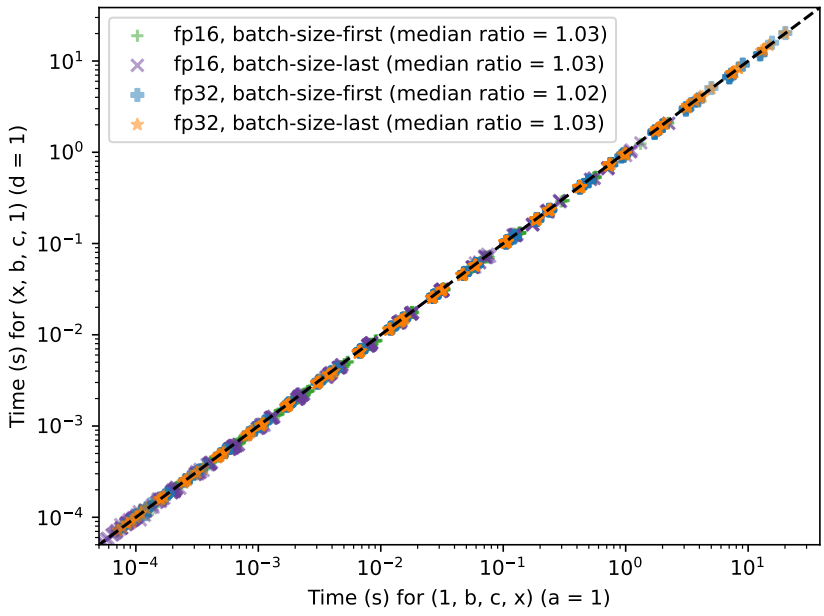
- GELU: `torch.nn.GELU()`;
- Linear: `torch.nn.Linear(in_features, out_features, bias=False)`;
- Linear + bias: `torch.nn.Linear(in_features, out_features, bias=True)`;
- LayerNorm: `torch.nn.LayerNorm(dim)`.

We compare the time of these operations in *batch-size-first vs. batch-size-last*.

GELU. This operation does not suppose any batch order so `torch.nn.GELU()` can be used both to implement *batch-size-first* and *batch-size-last*.



(a) `sparse`



(b) `dense`

Figure D.3: Median execution time (s) for a tuple $(1, b, c, x)$ (case $a = 1$) and the symmetric $(x, b, c, 1)$ (case $d = 1$).

Table D.4: Percentage of butterfly architectures (π_1, π_2) for which the multiplication $\mathbf{X} \mapsto \mathbf{B}_1\mathbf{B}_2\mathbf{X}$ is faster with `hybrid` than both `monarch` and `dense`. We consider between 24 and 90 different butterfly architectures for each size $m \times n$ of the product $\mathbf{B}_1\mathbf{B}_2$. In parenthesis is the median acceleration factor $(\frac{\min(\text{time of dense}, \text{time of monarch})}{\text{time of hybrid}})$ computed only in the cases where `hybrid` is faster than `monarch` and `dense`.

	`hybrid` $<$ min(`dense`, `monarch`)			
	Float-precision		*Half-precision*	
$m \times n$	***Batch-size-first***	***Batch-size-last***	***Batch-size-first***	***Batch-size-last***
384×384	15% ($\times$1.01)	52% ($\times$1.13)	0% (N/A)	0% (N/A)
768×768	13% ($\times$1.01)	87% ($\times$1.16)	0% (N/A)	0% (N/A)
1024×1024	13% ($\times$1.01)	92% ($\times$1.14)	0% (N/A)	0% (N/A)
384×1536	50% ($\times$1.02)	91% ($\times$1.12)	0% (N/A)	0% (N/A)
768×3072	21% ($\times$1.09)	99% ($\times$1.07)	0% (N/A)	27% ($\times$1.10)
1024×4096	17% ($\times$1.20)	100% ($\times$1.11)	0% (N/A)	33% ($\times$1.08)
1536×384	0% (N/A)	87% ($\times$1.11)	0% (N/A)	0% (N/A)
3072×768	0% (N/A)	98% ($\times$1.10)	0% (N/A)	0% (N/A)
4096×1024	0% (N/A)	96% ($\times$1.05)	0% (N/A)	21% ($\times$1.06)

Linear. `torch.nn.Linear` supposes that the first dimensions of the input tensor are the batch dimensions. We implement the *batch-size-last* variant by doing `torch.matmul(weight, input_bsl)` with `weight` a tensor of shape (m, n) saving the entries of $\mathbf{W} \in \mathbb{R}^{m \times n}$, and `input_bsl` a tensor of shape (n, K) saving the entries of $\mathbf{X} \in \mathbb{R}^{n \times K}$ for the forward pass. See Appendix D.5 for the reason behind this choice of implementation.

Linear + bias. The default implementation `torch.nn.Linear` exclusively supports *batch-size-first* and employs a technique called *kernel fusion*, which combines the matrix multiplication and bias addition into an optimized unified kernel. This fusion aims to improve computational efficiency by minimizing memory accesses and intermediate data transfers. While an analogous approach with kernel fusion in *batch-size-last* was not explored, we propose an *unfused* version of `torch.nn.Linear`. Concretely, `torch.nn.functional.linear(input, weight, bias)` in PyTorch is replaced by `torch.nn.functional.linear(input, weight)` + `bias`. this results in separate calls to two kernels, one for adding the bias and the other for matrix multiplication, leading to an "unfused" approach. Achieving a fused kernel would require delving into CUDA-level optimizations. We leave it open for future works.

LayerNorm. The default implementation supposes that the first dimensions of the input tensor are the batch dimensions, and has a dedicated CUDA kernel im-

	Half-precision		*Float-precision*	
	Batch-size-first (ms)	***Batch-size-last*** (ms)	***Batch-size-first*** (ms)	***Batch-size-last*** (ms)
GELU	0.034	0.034	0.060	0.060
Linear	0.20	0.20	1.73	1.70
Linear + bias (default)	0.22	N/A	1.75	N/A
Linear + bias (unfused)	0.38	0.38	1.96	1.94
LayerNorm (default)	0.072	N/A	0.09	N/A
LayerNorm (custom)	0.25	0.25	0.34	0.34

Table D.5: Median execution times (ms) in *float-precision* or *half-precision* of different PyTorch operations in *batch-size-first* or *batch-size-last*.

plementation in *batch-size-first*, which is not available in *batch-size-last*. We did not delve at the CUDA level to do an analog in *batch-size-last*. Instead, we directly implemented an equivalent in PyTorch using `torch.mean` and `torch.var`. Since this is expected to be less efficient than the CUDA implementation, we also compare it to an analogous PyTorch implementation in *batch-size-first* (using `torch.mean` and `torch.var`), that is called *custom* in Table D.5.

Details on the experiments. The experiments are done on a single A100-40GB GPU on AMD EPYC 7742 64-Core Processor. Measurements are done in *half-precision* and *float-precision*. The dimensions for the linear layer is 384×384. Input data $\mathbf{X}$ is a batch of 25088 vectors of size 384. The IQR are at least 100 times smaller than the median times, and we report the medians.

Results. The results in Table D.5 demonstrate that the unfused linear layer with bias, the linear layer without bias, and the GELU exhibit equivalent execution times in both *batch-size-first* and *batch-size-last*. This is a proof of concept that some operations could be performed with *batch-size-last at no additional cost*.

The default linear layer with bias (fused) is however faster than the fused implementations. It is left open whether a fused kernel with the same execution times could be written in *batch-size-last*.

While the custom version of LayerNorm remains invariant during the transition from *batch-size-first* to *batch-size-last*, it exhibits slower performance compared to the original implementation. The question remains open as to whether the original CUDA implementation can be directly modified without compromising efficiency.

Bibliography

[1] S. Abnar, M. Dehghani, B. Neyshabur, and H. Sedghi, *Exploring the limits of large scale pre-training*, in International Conference on Learning Representations, 2022.

[2] M. S. Advani, A. M. Saxe, and H. Sompolinsky, *High-dimensional dynamics of generalization error in neural networks*, Neural Networks, 132 (2020), pp. 428–446.

[3] P. Agrawal, J. Carreira, and J. Malik, *Learning to see by moving*, in IEEE/CVF International Conference on Computer Vision, 2015, pp. 37–45.

[4] A. Ahmed, B. Recht, and J. Romberg, *Blind deconvolution using convex programming*, IEEE Transactions on Information Theory, 60 (2013), pp. 1711–1732.

[5] N. Ailon, O. Leibovitch, and V. Nair, *Sparse linear networks with a fixed butterfly structure: Theory and practice*, in Uncertainty in Artificial Intelligence, PMLR, 2021, pp. 1174–1184.

[6] Z. Allen-Zhu, Y. Li, and Z. Song, *A convergence theory for deep learning via over-parameterization*, in International Conference on Machine Learning, PMLR, 2019, pp. 242–252.

[7] D. Aloise, A. Deshpande, P. Hansen, and P. Popat, *NP-Hardness of Euclidean sum-of-squares clustering*, Machine learning, 75 (2009), pp. 245–248.

[8] J. M. Alvarez and M. Salzmann, *Compression-aware training of deep networks*, in Advances in Neural Information Processing Systems, vol. 30, 2017.

[9] D. Amodei and D. Hernandez, *AI and compute*, 2018. https://openai.com/research/ai-and-compute [Accessed: March, 2024].

[10] P. Awasthi, M. Charikar, R. Krishnaswamy, and A. K. Sinop, *The hardness of approximation of Euclidean k-means*, in International Symposium on Computational Geometry, vol. 34, 2015, pp. 754–767.

[11] S. Axler, P. Bourdon, and R. Wade, *Harmonic function theory*, Springer Science & Business Media, 2013.

[12] S. Bahmani and J. Romberg, *Lifting for blind deconvolution in random mask imaging: Identifiability and convex relaxation*, SIAM Journal on Imaging Sciences, 8 (2015), pp. 2203–2238.

[13] R. Balestriero, M. Ibrahim, V. Sobal, A. Morcos, S. Shekhar, T. Goldstein, F. Bordes, A. Bardes, G. Mialon, Y. Tian, A. Schwarzschild, A. G. Wilson, J. Geiping, Q. Garrido, P. Fernandez, A. Bar, H. Pirsiavash, Y. LeCun, and M. Goldblum, *A cookbook of self-supervised learning*, arXiv preprint arXiv:2304.12210, 2023.

[14] H. Bao, L. Dong, S. Piao, and F. Wei, *BEit: BERT pre-training of image transformers*, in International Conference on Learning Representations, 2022.

[15] A. BARDES, J. PONCE, AND Y. LECUN, *VICReg: Variance-invariance-covariance regularization for self-supervised learning*, in International Conference on Learning Representations, 2022.

[16] A. H. BARNETT, *How exponentially ill-conditioned are contiguous submatrices of the Fourier matrix?*, SIAM Review, 64 (2022), pp. 105–131.

[17] B. BARTOLDSON, A. MORCOS, A. BARBU, AND G. ERLEBACHER, *The generalization-stability tradeoff in neural network pruning*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 20852–20864.

[18] R. BEATSON AND L. GREENGARD, *A short course on fast multipole methods*, Wavelets, multilevel methods and elliptic PDEs, 1 (1997), pp. 1–37.

[19] M. BELKIN, D. HSU, S. MA, AND S. MANDAL, *Reconciling modern machine-learning practice and the classical bias-variance trade-off*, Proceedings of the National Academy of Sciences, 116 (2019), pp. 15849–15854.

[20] G. BELLEC, D. KAPPEL, W. MAASS, AND R. LEGENSTEIN, *Deep rewiring: Training very sparse deep networks*, in International Conference on Learning Representations, 2018.

[21] I. BELLO, W. FEDUS, X. DU, E. D. CUBUK, A. SRINIVAS, T.-Y. LIN, J. SHLENS, AND B. ZOPH, *Revisiting ResNets: Improved training and scaling strategies*, in Advances in Neural Information Processing Systems, vol. 34, 2021, pp. 22614–22627.

[22] Y. BENGIO, A. COURVILLE, AND P. VINCENT, *Representation learning: A review and new perspectives*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 35 (2013), pp. 1798–1828.

[23] Y. BENGIO, P. SIMARD, AND P. FRASCONI, *Learning long-term dependencies with gradient descent is difficult*, IEEE transactions on neural networks, 5 (1994), pp. 157–166.

[24] A. BENICHOUX, E. VINCENT, AND R. GRIBONVAL, *A fundamental pitfall in blind deconvolution with sparse and shift-invariant priors*, in IEEE International Conference on Acoustics, Speech and Signal Processing, 2013, pp. 6108–6112.

[25] L. BEYER, X. ZHAI, AND A. KOLESNIKOV, *Better plain ViT baselines for ImageNet-1k*, arXiv:2205.01580, 2022.

[26] S. BIBIKAR, H. VIKALO, Z. WANG, AND X. CHEN, *Federated dynamic sparse training: Computing less, communicating less, yet learning better*, in AAAI Conference on Artificial Intelligence, vol. 36, 2022, pp. 6080–6088.

[27] C. M. BISHOP, *Pattern recognition and machine learning*, Springer, 2006.

[28] D. BLALOCK, J. J. GONZALEZ ORTIZ, J. FRANKLE, AND J. GUTTAG, *What is the state of neural network pruning?*, in Proceedings of machine learning and systems, vol. 2, 2020, pp. 129–146.

[29] S. BOEHM, *How to optimize a CUDA matmul kernel for cuBLAS-like performance: A worklog*, 2022. https://siboehm.com/articles/22/CUDA-MMM [Accessed: February, 2024].

[30] P. BOJANOWSKI AND A. JOULIN, *Unsupervised learning by predicting noise*, in International Conference on Machine Learning, PMLR, 2017, pp. 517–526.

[31] R. BOMMASANI, D. A. HUDSON, E. ADELI, R. ALTMAN, S. ARORA, S. VON ARX, M. S. BERNSTEIN, J. BOHG, A. BOSSELUT, E. BRUNSKILL, E. BRYNJOLFSSON, S. BUCH, D. CARD, R. CASTELLON, N. CHATTERJI, A. CHEN, K. CREEL, J. Q. DAVIS, D. DEMSZKY, C. DONAHUE, M. DOUMBOUYA, E. DURMUS, S. ERMON, J. ETCHEMENDY, K. ETHAYARAJH, L. FEI-FEI, C. FINN, T. GALE, L. GILLESPIE, K. GOEL, N. GOODMAN, S. GROSSMAN, N. GUHA, T. HASHIMOTO, P. HENDERSON, J. HEWITT, D. E. HO, J. HONG, K. HSU, J. HUANG, T. ICARD, S. JAIN, D. JURAFSKY, P. KALLURI, S. KARAMCHETI, G. KEELING, F. KHANI, O. KHATTAB, P. W. KOH, M. KRASS, R. KRISHNA, R. KUDITIPUDI, A. KUMAR, F. LADHAK, M. LEE, T. LEE, J. LESKOVEC, I. LEVENT, X. L. LI, X. LI, T. MA,

A. MALIK, C. D. MANNING, S. MIRCHANDANI, E. MITCHELL, Z. MUNYIKWA, S. NAIR, A. NARAYAN, D. NARAYANAN, B. NEWMAN, A. NIE, J. C. NIEBLES, H. NILFOROSHAN, J. NYARKO, G. OGUT, L. ORR, I. PAPADIMITRIOU, J. S. PARK, C. PIECH, E. PORTELANCE, C. POTTS, A. RAGHUNATHAN, R. REICH, H. REN, F. RONG, Y. ROOHANI, C. RUIZ, J. RYAN, C. RÉ, D. SADIGH, S. SAGAWA, K. SANTHANAM, A. SHIH, K. SRINIVASAN, A. TAMKIN, R. TAORI, A. W. THOMAS, F. TRAMÈR, R. E. WANG, W. WANG, B. WU, J. WU, Y. WU, S. M. XIE, M. YASUNAGA, J. YOU, M. ZAHARIA, M. ZHANG, T. ZHANG, X. ZHANG, Y. ZHANG, L. ZHENG, K. ZHOU, AND P. LIANG, *On the opportunities and risks of foundation models*, arXiv preprint arXiv:2108.07258, 2021.

[32] F. BORDES, R. BALESTRIERO, AND P. VINCENT, *Towards democratizing joint-embedding self-supervised learning*, arXiv preprint arXiv:2303.01986, 2023.

[33] K. M. BORGWARDT, A. GRETTON, M. J. RASCH, H.-P. KRIEGEL, B. SCHÖLKOPF, AND A. J. SMOLA, *Integrating structured biological data by kernel maximum mean discrepancy*, Bioinformatics, 22 (2006), pp. e49–e57.

[34] S. V. BORODACHOV, D. P. HARDIN, AND E. B. SAFF, *Discrete energy on rectifiable sets*, Springer, 2019.

[35] P. S. BRADLEY, K. P. BENNETT, AND A. DEMIRIZ, *Constrained k-means clustering*, Microsoft Research, Redmond, 20 (2000).

[36] J. BRAUCHART, E. B. SAFF, I. H. SLOAN, AND R. S. WOMERSLEY, *QMC designs: Optimal order quasi Monte Carlo integration schemes on the sphere*, Mathematics of computation, 83 (2014), pp. 2821–2851.

[37] F.-X. BRIOL, A. BARP, A. B. DUNCAN, AND M. GIROLAMI, *Statistical inference for generative models with maximum mean discrepancy*, arXiv preprint arXiv:1906.05944, 2019.

[38] J. BROMLEY, I. GUYON, Y. LECUN, E. SÄCKINGER, AND R. SHAH, *Signature verification using a Siamese time delay neural network*, in Advances in Neural Information Processing Systems, vol. 6, 1993.

[39] S. BUBECK AND M. SELLKE, *A universal law of robustness via isoperimetry*, in Advances in Neural Information Processing Systems, vol. 34, 2021, pp. 28811–28822.

[40] E. CANDES, L. DEMANET, AND L. YING, *A fast butterfly algorithm for the computation of Fourier integral operators*, Multiscale Modeling & Simulation, 7 (2009), pp. 1727–1750.

[41] E. J. CANDES, Y. C. ELDAR, T. STROHMER, AND V. VORONINSKI, *Phase retrieval via matrix completion*, SIAM Review, 57 (2015), pp. 225–251.

[42] E. J. CANDES, T. STROHMER, AND V. VORONINSKI, *Phaselift: Exact and stable signal recovery from magnitude measurements via convex programming*, Communications on Pure and Applied Mathematics, 66 (2013), pp. 1241–1274.

[43] Y. CAO, Z. XIE, B. LIU, Y. LIN, Z. ZHANG, AND H. HU, *Parametric instance classification for unsupervised visual feature learning*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 15614–15624.

[44] M. CARON, I. MISRA, J. MAIRAL, P. GOYAL, P. BOJANOWSKI, AND A. JOULIN, *Unsupervised learning of visual features by contrasting cluster assignments*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 9912–9924.

[45] M. CARON, H. TOUVRON, I. MISRA, H. JÉGOU, J. MAIRAL, P. BOJANOWSKI, AND A. JOULIN, *Emerging properties in self-supervised vision transformers*, in IEEE/CVF International Conference on Computer Vision, 2021, pp. 9650–9660.

[46] Z. CHARLES, K. BONAWITZ, S. CHIKNAVARYAN, B. MCMAHAN, AND B. A. ARCAS, *Federated select: A primitive for communication-and memory-efficient federated learning*, arXiv preprint arXiv:2208.09432, 2022.

[47] G. CHECHIK, V. SHARMA, U. SHALIT, AND S. BENGIO, *Large scale online learning of image similarity through ranking*, Journal of Machine Learning Research, 11 (2010).

[48] L.-C. CHEN, G. PAPANDREOU, I. KOKKINOS, K. MURPHY, AND A. L. YUILLE, *DeepLab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 40 (2017), pp. 834–848.

[49] P. CHEN, H.-F. YU, I. DHILLON, AND C.-J. HSIEH, *Drone: Data-aware low-rank compression for large NLP models*, vol. 34, 2021, pp. 29321–29334.

[50] T. CHEN, S. KORNBLITH, M. NOROUZI, AND G. HINTON, *A simple framework for contrastive learning of visual representations*, in International Conference on Machine Learning, PMLR, 2020, pp. 1597–1607.

[51] T. CHEN, S. KORNBLITH, K. SWERSKY, M. NOROUZI, AND G. E. HINTON, *Big self-supervised models are strong semi-supervised learners*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 22243–22255.

[52] T. CHEN, C. LUO, AND L. LI, *Intriguing properties of contrastive losses*, in Advances in Neural Information Processing Systems, vol. 34, 2021, pp. 11834–11845.

[53] T. CHEN, Z. ZHANG, S. LIU, Y. ZHANG, S. CHANG, AND Z. WANG, *Data-efficient double-win lottery tickets from robust pre-training*, in International Conference on Machine Learning, PMLR, 2022, pp. 3747–3759.

[54] T. CHEN, Z. ZHANG, PENGJUN WANG, S. BALACHANDRA, H. MA, Z. WANG, AND Z. WANG, *Sparsity winning twice: Better robust generalization from more efficient training*, in International Conference on Learning Representations, 2022.

[55] X. CHEN AND K. HE, *Exploring simple Siamese representation learning*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 15750–15758.

[56] Y. CHEN, B. ZHENG, Z. ZHANG, Q. WANG, C. SHEN, AND Q. ZHANG, *Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions*, ACM Computing Surveys (CSUR), 53 (2020), pp. 1–37.

[57] H. CHENG, Z. GIMBUTAS, P.-G. MARTINSSON, AND V. ROKHLIN, *On the compression of low rank matrices*, SIAM Journal on Scientific Computing, 26 (2005), pp. 1389–1404.

[58] Y. CHENG, F. X. YU, R. S. FERIS, S. KUMAR, A. CHOUDHARY, AND S.-F. CHANG, *An exploration of parameter redundancy in deep networks with circulant projections*, in IEEE/CVF International Conference on Computer Vision, 2015, pp. 2857–2865.

[59] M. CHERTI, R. BEAUMONT, R. WIGHTMAN, M. WORTSMAN, G. ILHARCO, C. GORDON, C. SCHUHMANN, L. SCHMIDT, AND J. JITSEV, *Reproducible scaling laws for contrastive language-image learning*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2023, pp. 2818–2829.

[60] S. CHOPRA, R. HADSELL, AND Y. LECUN, *Learning a similarity metric discriminatively, with application to face verification*, in IEEE Computer Society Conference on Computer Vision and Pattern Recognition, vol. 1, 2005, pp. 539–546.

[61] S. CHOUDHARY AND U. MITRA, *Identifiability scaling laws in bilinear inverse problems*, arXiv preprint arXiv:1402.2637, 2014.

[62] S. CHOUDHARY AND U. MITRA, *Sparse blind deconvolution: What cannot be done*, in IEEE International Symposium on Information Theory, 2014, pp. 3002–3006.

[63] C.-Y. CHUANG, J. ROBINSON, Y.-C. LIN, A. TORRALBA, AND S. JEGELKA, *Debiased contrastive learning*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 8765–8775.

[64] A. COATES, A. NG, AND H. LEE, *An analysis of single-layer networks in unsupervised feature learning*, in International Conference on Artificial Intelligence and Statistics, 2011, pp. 215–223.

[65] J. W. COOLEY AND J. W. TUKEY, *An algorithm for the machine calculation of complex Fourier series*, Mathematics of computation, 19 (1965), pp. 297–301.

[66] M. CORDTS, M. OMRAN, S. RAMOS, T. REHFELD, M. ENZWEILER, R. BENENSON, U. FRANKE, S. ROTH, AND B. SCHIELE, *The Cityscapes dataset for semantic urban scene understanding*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2016, pp. 3213–3223.

[67] A. COSSE AND L. DEMANET, *Stable rank-one matrix completion is solved by the level 2 Lasserre relaxation*, Foundations of Computational Mathematics, 21 (2020), pp. 1–50.

[68] E. J. CROWLEY, J. TURNER, A. STORKEY, AND M. O'BOYLE, *A closer look at structured pruning for neural network compression*, arXiv preprint arXiv:1810.04622, 2018.

[69] A. CURTH, A. JEFFARES, AND M. VAN DER SCHAAR, *A U-turn on double descent: Rethinking parameter counting in statistical learning*, in Advances in Neural Information Processing Systems, vol. 36, 2024.

[70] R. DANGOVSKI, L. JING, C. LOH, S. HAN, A. SRIVASTAVA, B. CHEUNG, P. AGRAWAL, AND M. SOLJACIC, *Equivariant self-supervised learning: Encouraging equivariance in representations*, in International Conference on Learning Representations, 2022.

[71] T. DAO, B. CHEN, K. LIANG, J. YANG, Z. SONG, A. RUDRA, AND C. RE, *Pixelated butterfly: Simple and efficient sparse training for neural network models*, in International Conference on Learning Representations, 2022.

[72] T. DAO, B. CHEN, N. S. SOHONI, A. DESAI, M. POLI, J. GROGAN, A. LIU, A. RAO, A. RUDRA, AND C. RÉ, *Monarch: Expressive structured matrices for efficient and accurate training*, in International Conference on Machine Learning, PMLR, 2022, pp. 4690–4721.

[73] T. DAO, D. FU, S. ERMON, A. RUDRA, AND C. RÉ, *Flashattention: Fast and memory-efficient exact attention with IO-awareness*, in Advances in Neural Information Processing Systems, vol. 35, 2022, pp. 16344–16359.

[74] T. DAO, A. GU, M. EICHHORN, A. RUDRA, AND C. RÉ, *Learning fast algorithms for linear transforms using butterfly factorizations*, in International Conference on Machine Learning, PMLR, 2019, pp. 1517–1527.

[75] T. DAO, N. SOHONI, A. GU, M. EICHHORN, A. BLONDER, M. LESZCZYNSKI, A. RUDRA, AND C. RÉ, *Kaleidoscope: An efficient, learnable representation for all structured linear maps*, in International Conference on Learning Representations, 2020.

[76] A. DE VRIES, *The growing energy footprint of artificial intelligence*, Joule, 7 (2023), pp. 2191–2194.

[77] L. DEMANET AND L. YING, *Fast wave computation via Fourier integral operators*, Mathematics of Computation, 81 (2012), pp. 1455–1486.

[78] J. DENG, W. DONG, R. SOCHER, L.-J. LI, K. LI, AND L. FEI-FEI, *ImageNet: A large-scale hierarchical image database*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2009.

[79] M. DENIL, B. SHAKIBI, L. DINH, M. RANZATO, AND N. DE FREITAS, *Predicting parameters in deep learning*, in Advances in Neural Information Processing Systems, vol. 26, 2013.

[80] E. L. DENTON, W. ZAREMBA, J. BRUNA, Y. LECUN, AND R. FERGUS, *Exploiting linear structure within convolutional networks for efficient evaluation*, in Advances in Neural Information Processing Systems, vol. 27, 2014.

[81] R. DESISLAVOV, F. MARTÍNEZ-PLUMED, AND J. HERNÁNDEZ-ORALLO, *Trends in AI inference energy consumption: Beyond the performance-vs-parameter laws of deep learning*, Sustainable Computing: Informatics and Systems, 38 (2023), p. 100857.

[82] A. DEVILLIERS AND M. LEFORT, *Equimod: An equivariance module to improve visual instance discrimination*, in International Conference on Learning Representations, 2023.

[83] J. DEVLIN, M.-W. CHANG, K. LEE, AND K. TOUTANOVA, *BERT: Pre-training of deep bidirectional transformers for language understanding*, in Conference of the North American Chapter of the Association for Computational Linguistics, vol. 1, 2019, pp. 4171–4186.

[84] C. DING, S. LIAO, Y. WANG, Z. LI, N. LIU, Y. ZHUO, C. WANG, X. QIAN, Y. BAI, G. YUAN, X. MA, Y. ZHANG, J. TANG, Q. QIU, X. LIN, AND B. YUAN, *CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices*, in IEEE/ACM International Symposium on Microarchitecture, 2017, pp. 395–408.

[85] C. DOERSCH, A. GUPTA, AND A. A. EFROS, *Unsupervised visual representation learning by context prediction*, in IEEE/CVF International Conference on Computer Vision, 2015, pp. 1422–1430.

[86] X. DONG, S. CHEN, AND S. PAN, *Learning to prune deep neural networks via layer-wise optimal brain surgeon*, in Advances in Neural Information Processing Systems, vol. 30, 2017.

[87] A. DOSOVITSKIY, L. BEYER, A. KOLESNIKOV, D. WEISSENBORN, X. ZHAI, T. UNTERTHINER, M. DEHGHANI, M. MINDERER, G. HEIGOLD, S. GELLY, J. USZKOREIT, AND N. HOULSBY, *An image is worth 16x16 words: Transformers for image recognition at scale*, in International Conference on Learning Representations, 2020.

[88] A. DOSOVITSKIY, J. T. SPRINGENBERG, M. RIEDMILLER, AND T. BROX, *Discriminative unsupervised feature learning with convolutional neural networks*, in Advances in Neural Information Processing Systems, vol. 27, 2014.

[89] P. DUHAMEL AND M. VETTERLI, *Fast Fourier transforms: a tutorial review and a state of the art*, Signal processing, 19 (1990), pp. 259–299.

[90] V. DUTORDOIR, N. DURRANDE, AND J. HENSMAN, *Sparse Gaussian processes with spherical harmonic features*, in International Conference on Machine Learning, PMLR, 2020, pp. 2793–2802.

[91] G. K. DZIUGAITE, D. M. ROY, AND Z. GHAHRAMANI, *Training generative neural networks via maximum mean discrepancy optimization*, in Conference on Uncertainty in Artificial Intelligence, 2015, p. 258–267.

[92] A. EDELMAN, P. MCCORQUODALE, AND S. TOLEDO, *The future fast Fourier transform?*, SIAM Journal on Scientific Computing, 20 (1998), pp. 1094–1114.

[93] M. ELAD, *Sparse and redundant representations: from theory to applications in signal and image processing*, Springer Science & Business Media, 2010.

[94] Y. C. ELDAR, D. NEEDELL, AND Y. PLAN, *Uniqueness conditions for low-rank matrix recovery*, Applied and Computational Harmonic Analysis, 33 (2012), pp. 309–314.

[95] E. ELHAMIFAR AND R. VIDAL, *Clustering disjoint subspaces via sparse representation*, in IEEE International Conference on Acoustics, Speech and Signal Processing, 2010, pp. 1926–1929.

[96] E. ELHAMIFAR AND R. VIDAL, *Sparse subspace clustering: Algorithm, theory, and applications*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 35 (2013), pp. 2765–2781.

[97] S. ELLACOTT, *Aspects of the numerical analysis of neural networks*, Acta numerica, 3 (1994), pp. 145–202.

[98] T. ELSKEN, J. H. METZEN, AND F. HUTTER, *Neural architecture search: A survey*, Journal of Machine Learning Research, 20 (2019), pp. 1–21.

[99] B. ENGQUIST AND L. YING, *A fast directional algorithm for high frequency acoustic scattering in two dimensions*, Communications in Mathematical Sciences, 7 (2009), pp. 327 – 345.

[100] L. ERICSSON, H. GOUK, AND T. HOSPEDALES, *Why do self-supervised models transfer? on the impact of invariance on downstream tasks*, in British Machine Vision Conference, BMVA, 2022, p. 509.

[101] A. ERMOLOV, A. SIAROHIN, E. SANGINETO, AND N. SEBE, *Whitening for self-supervised representation learning*, in International Conference on Machine Learning, PMLR, 2021, pp. 3015–3024.

[102] U. EVCI, T. GALE, J. MENICK, P. S. CASTRO, AND E. ELSEN, *Rigging the lottery: Making all tickets winners*, in International Conference on Machine Learning, PMLR, 2020, pp. 2943–2952.

[103] M. EVERINGHAM, L. VAN GOOL, C. K. WILLIAMS, J. WINN, AND A. ZISSERMAN, *The PASCAL visual object classes (VOC) challenge*, International Journal of Computer Vision, 88 (2010), pp. 303–338.

[104] H. FAN, T. CHAU, S. I. VENIERIS, R. LEE, A. KOURIS, W. LUK, N. D. LANE, AND M. S. ABDELFATTAH, *Adaptable butterfly accelerator for attention-based NNs via hardware and algorithm co-design*, in IEEE/ACM International Symposium on Microarchitecture, 2022, pp. 599–615.

[105] R.-E. FAN, K.-W. CHANG, C.-J. HSIEH, X.-R. WANG, AND C.-J. LIN, *LIBLINEAR: A library for large linear classification*, Journal of Machine Learning Research, 9 (2008), pp. 1871–1874.

[106] W. FEDUS, J. DEAN, AND B. ZOPH, *A review of sparse expert models in deep learning*, arXiv preprint arXiv:2209.01667, 2022.

[107] M. FENN AND G. STEIDL, *FMM and $\mathcal{H}$-matrices: A short introduction to the basic idea*, tech. rep., Department for Mathematics and Computer Science, University of Mannheim, 2002.

[108] S. FOUCART AND H. RAUHUT, *A mathematical introduction to compressive sensing*, Applied and Numerical Harmonic Analysis, Birkhäuser Basel, 2013.

[109] J. FRANKLE AND M. CARBIN, *The lottery ticket hypothesis: Finding sparse, trainable neural networks*, in International Conference on Learning Representations, 2019.

[110] J. FRANKLE, G. K. DZIUGAITE, D. ROY, AND M. CARBIN, *Linear mode connectivity and the lottery ticket hypothesis*, in International Conference on Machine Learning, PMLR, 2020, pp. 3259–3269.

[111] E. FRANTAR AND D. ALISTARH, *Optimal brain compression: A framework for accurate post-training quantization and pruning*, in Advances in Neural Information Processing Systems, vol. 35, 2022, pp. 4475–4488.

[112] E. FRANTAR AND D. ALISTARH, *SparseGPT: Massive language models can be accurately pruned in one-shot*, in International Conference on Machine Learning, PMLR, 2023, pp. 10323–10337.

[113] D. FU, S. ARORA, J. GROGAN, I. JOHNSON, E. S. EYUBOGLU, A. THOMAS, B. SPECTOR, M. POLI, A. RUDRA, AND C. RÉ, *Monarch mixer: A simple sub-quadratic GEMM-based architecture*, in Advances in Neural Information Processing Systems, vol. 36, 2023.

[114] K. FUKUMIZU, F. R. BACH, AND M. I. JORDAN, *Dimensionality reduction for supervised learning with reproducing kernel Hilbert spaces*, Journal of Machine Learning Research, 5 (2004), pp. 73–99.

[115] K. FUKUMIZU, A. GRETTON, X. SUN, AND B. SCHÖLKOPF, *Kernel measures of conditional dependence*, in Advances in Neural Information Processing Systems, vol. 20, 2007.

[116] T. GALE, E. ELSEN, AND S. HOOKER, *The state of sparsity in deep neural networks*, arXiv preprint arXiv:1902.09574, 2019.

[117] H. GAO, F. NIE, X. LI, AND H. HUANG, *Multi-view subspace clustering*, in IEEE/CVF International Conference on Computer Vision, 2015, pp. 4238–4246.

[118] E. GARCÍA-PORTUGUÉS AND T. VERDEBOUT, *An overview of uniformity tests on the hypersphere*, arXiv preprint arXiv:1804.00286, 2018.

[119] Q. GARRIDO, R. BALESTRIERO, L. NAJMAN, AND Y. LECUN, *Rankme: Assessing the downstream performance of pretrained self-supervised representations by their rank*, in International Conference on Machine Learning, PMLR, 2023, pp. 10929–10974.

[120] Q. GARRIDO, Y. CHEN, A. BARDES, L. NAJMAN, AND Y. LECUN, *On the duality between contrastive and non-contrastive self-supervised learning*, in International Conference on Learning Representations, 2023.

[121] Q. GARRIDO, L. NAJMAN, AND Y. LECUN, *Self-supervised learning of split invariant equivariant representations*, in International Conference on Machine Learning, PMLR, 2023, pp. 10975–10996.

[122] M. GEIGER, A. JACOT, S. SPIGLER, F. GABRIEL, L. SAGUN, S. D'ASCOLI, G. BIROLI, C. HONGLER, AND M. WYART, *Scaling description of generalization with number of parameters in deep learning*, Journal of Statistical Mechanics: Theory and Experiment, (2020), p. 023401.

[123] S. GIDARIS, A. BURSUC, N. KOMODAKIS, P. PÉREZ, AND M. CORD, *Learning representations by predicting bags of visual words*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 6928–6938.

[124] S. GIDARIS, A. BURSUC, G. PUY, N. KOMODAKIS, M. CORD, AND P. PÉREZ, *Online bag-of-visual-words generation for self-supervised learning*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 6830–6840.

[125] S. GIDARIS, P. SINGH, AND N. KOMODAKIS, *Unsupervised representation learning by predicting image rotations*, in International Conference on Learning Representations, 2018.

[126] N. GILLIS, *Nonnegative matrix factorization*, SIAM, 2020.

[127] R. GIRSHICK, J. DONAHUE, T. DARRELL, AND J. MALIK, *Rich feature hierarchies for accurate object detection and semantic segmentation*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2014, pp. 580–587.

[128] A. GITLIN, B. TAO, L. BALZANO, AND J. LIPOR, *Improving k-subspaces via coherence pursuit*, IEEE Journal of Selected Topics in Signal Processing, 12 (2018), pp. 1575–1588.

[129] A. GOH AND R. VIDAL, *Segmenting motions of different types by unsupervised manifold clustering*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2007, pp. 1–6.

[130] I. GOHBERG AND V. OLSHEVSKY, *Complexity of multiplication with vectors for structured matrices*, Linear Algebra and Its Applications, 202 (1994), pp. 163–192.

[131] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press, 1996.

[132] A. GONON, L. ZHENG, P. CARRIVAIN, AND Q.-T. LE, *Make inference faster: Efficient GPU memory management for butterfly sparse matrix multiplication*. Preprint, hal-04584450, 2024.

[133] A. GONON, L. ZHENG, C. LALANNE, Q.-T. LE, G. LAUGA, AND C. POULIQUEN, *Sparsity in neural networks can improve their privacy*, arXiv preprint arXiv:2304.10553, 2023.

[134] I. GOODFELLOW, Y. BENGIO, AND A. COURVILLE, *Deep learning*, MIT press, 2016.

[135] I. J. GOODFELLOW, O. VINYALS, AND A. M. SAXE, *Qualitatively characterizing neural network optimization problems*, in International Conference on Learning Representations, 2015.

[136] J. GOU, B. YU, S. J. MAYBANK, AND D. TAO, *Knowledge distillation: A survey*, International Journal of Computer Vision, 129 (2021), pp. 1789–1819.

[137] P. GOYAL, P. DOLLÁR, R. GIRSHICK, P. NOORDHUIS, L. WESOLOWSKI, A. KYROLA, A. TULLOCH, Y. JIA, AND K. HE, *Accurate, large minibatch SGD: Training ImageNet in 1 hour*, arXiv preprint arXiv:1706.02677, 2017.

[138] P. GOYAL, Q. DUVAL, J. REIZENSTEIN, M. LEAVITT, M. XU, B. LEFAUDEUX, M. SINGH, V. REIS, M. CARON, P. BOJANOWSKI, A. JOULIN, AND I. MISRA, *VISSL*. https://github.com/facebookresearch/vissl, 2021.

[139] S. GRAY, A. RADFORD, AND D. P. KINGMA, *GPU kernels for block-sparse weights*, arXiv preprint arXiv:1711.09224, 2017.

[140] L. GREENGARD AND V. ROKHLIN, *A fast algorithm for particle simulations*, Journal of computational physics, 73 (1987), pp. 325–348.

[141] A. GRETTON, K. M. BORGWARDT, M. J. RASCH, B. SCHÖLKOPF, AND A. SMOLA, *A kernel two-sample test*, Journal of Machine Learning Research, 13 (2012), pp. 723–773.

[142] A. GRETTON, O. BOUSQUET, A. SMOLA, AND B. SCHÖLKOPF, *Measuring statistical dependence with Hilbert-Schmidt norms*, in Algorithmic Learning Theory, 2005, pp. 63–77.

[143] J.-B. GRILL, F. STRUB, F. ALTCHÉ, C. TALLEC, P. RICHEMOND, E. BUCHATSKAYA, C. DOERSCH, B. AVILA PIRES, Z. GUO, M. GHESHLAGHI AZAR, B. PIOT, K. KAVUKCUOGLU, R. MUNOS, AND M. VALKO, *Bootstrap your own latent - A new approach to self-supervised learning*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 21271–21284.

[144] S. GRUMBACH, *L'empire des algorithmes: Une géopolitique du contrôle*, Armand Colin, 2022.

[145] M. GU AND S. C. EISENSTAT, *Efficient algorithms for computing a strong rank-revealing QR factorization*, SIAM Journal on Scientific Computing, 17 (1996), pp. 848–869.

[146] H. GUO, Y. LIU, J. HU, AND E. MICHIELSSEN, *A butterfly-based direct integral-equation solver using hierarchical LU factorization for analyzing scattering from electrically large conducting objects*, IEEE Transactions on Antennas and Propagation, 65 (2017), pp. 4742–4750.

[147] Y. GUO, *A survey on methods and theories of quantized neural networks*, arXiv preprint arXiv:1808.04752, 2018.

[148] M. GUTMANN AND A. HYVÄRINEN, *Noise-contrastive estimation: A new estimation principle for unnormalized statistical models*, in International Conference on Artificial Intelligence and Statistics, 2010, pp. 297–304.

[149] W. HACKBUSCH, *A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices*, Computing, 62 (1999), pp. 89–108.

[150] W. HACKBUSCH, *Hierarchical matrices: algorithms and analysis*, Springer, 2015.

[151] W. HACKBUSCH AND S. BÖRM, *Data-sparse approximation by adaptive $\mathcal{H}^2$-matrices*, Computing, 69 (2002), pp. 1–35.

[152] R. HADSELL, S. CHOPRA, AND Y. LECUN, *Dimensionality reduction by learning an invariant mapping*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2006, pp. 1735–1742.

[153] N. HALKO, P.-G. MARTINSSON, AND J. A. TROPP, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Review, 53 (2011), pp. 217–288.

[154] S. M. HALVAGAL, A. LABORIEUX, AND F. ZENKE, *Implicit variance regularization in non-contrastive SSL*, in Advances in Neural Information Processing Systems, vol. 36, 2023, pp. 63409–63436.

[155] S. HAN, H. MAO, AND W. J. DALLY, *Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding*, in International Conference on Learning Representations, 2015.

[156] J. Z. HAOCHEN, C. WEI, A. GAIDON, AND T. MA, *Provable guarantees for self-supervised deep learning with spectral contrastive loss*, in Advances in Neural Information Processing Systems, vol. 34, 2021, pp. 5000–5011.

[157] D. P. HARDIN AND E. B. SAFF, *Minimal Riesz energy point configurations for rectifiable d-dimensional manifolds*, Advances in Mathematics, 193 (2005), pp. 174–204.

[158] J. A. HARTIGAN, *Direct clustering of a data matrix*, Journal of the american statistical association, 67 (1972), pp. 123–129.

[159] B. HASSIBI, D. G. STORK, AND G. J. WOLFF, *Optimal brain surgeon and general network pruning*, in IEEE International Conference on Neural Networks, 1993, pp. 293–299.

[160] T. HASTIE, R. TIBSHIRANI, J. H. FRIEDMAN, AND J. H. FRIEDMAN, *The elements of statistical learning: data mining, inference, and prediction*, Springer, 2009.

[161] K. HE, X. CHEN, S. XIE, Y. LI, P. DOLLÁR, AND R. GIRSHICK, *Masked autoencoders are scalable vision learners*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 16000–16009.

[162] K. HE, H. FAN, Y. WU, S. XIE, AND R. GIRSHICK, *Momentum contrast for unsupervised visual representation learning*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 9729–9738.

[163] K. HE, X. ZHANG, S. REN, AND J. SUN, *Deep residual learning for image recognition*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[164] Y. HE, X. ZHANG, AND J. SUN, *Channel pruning for accelerating very deep neural networks*, in IEEE/CVF International Conference on Computer Vision, 2017, pp. 1389–1397.

[165] O. HENAFF, *Data-efficient image recognition with contrastive predictive coding*, in International Conference on Machine Learning, PMLR, 2020, pp. 4182–4192.

[166] T. HENIGHAN, J. KAPLAN, M. KATZ, M. CHEN, C. HESSE, J. JACKSON, H. JUN, T. B. BROWN, P. DHARIWAL, S. GRAY, C. HALLACY, B. MANN, A. RADFORD, A. RAMESH, N. RYDER, D. M. ZIEGLER, J. SCHULMAN, D. AMODEI, AND S. MCCANDLISH, *Scaling laws for autoregressive generative modeling*, arXiv preprint arXiv:2010.14701, 2020.

[167] D. HERNANDEZ, J. KAPLAN, T. HENIGHAN, AND S. MCCANDLISH, *Scaling laws for transfer*, arXiv preprint arXiv:2102.01293, 2021.

[168] J. HESTNESS, S. NARANG, N. ARDALANI, G. DIAMOS, H. JUN, H. KIANINEJAD, M. M. A. PATWARY, Y. YANG, AND Y. ZHOU, *Deep learning scaling is predictable, empirically*, arXiv preprint arXiv:1712.00409, 2017.

[169] G. HINTON, O. VINYALS, AND J. DEAN, *Distilling the knowledge in a neural network*, arXiv preprint arXiv:1503.02531, 2015.

[170] G. E. HINTON AND R. ZEMEL, *Autoencoders, minimum description length and helmholtz free energy*, in Advances in Neural Information Processing Systems, vol. 6, 1993.

[171] R. D. HJELM, A. FEDOROV, S. LAVOIE-MARCHILDON, K. GREWAL, P. BACHMAN, A. TRISCHLER, AND Y. BENGIO, *Learning deep representations by mutual information estimation and maximization*, in International Conference on Learning Representations, 2019.

[172] T. HOEFLER, D. ALISTARH, T. BEN-NUN, N. DRYDEN, AND A. PESTE, *Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks*, Journal of Machine Learning Research, 22 (2021), pp. 1–124.

[173] A. G. HOWARD, M. ZHU, B. CHEN, D. KALENICHENKO, W. WANG, T. WEYAND, M. ANDREETTO, AND H. ADAM, *MobileNets: Efficient convolutional neural networks for mobile vision applications*, arXiv preprint arXiv:1704.04861, 2017.

[174] Y.-C. HSU, T. HUA, S. CHANG, Q. LOU, Y. SHEN, AND H. JIN, *Language model compression with weighted low-rank factorization*, in International Conference on Learning Representations, 2022.

[175] E. J. HU, YELONG SHEN, P. WALLIS, Z. ALLEN-ZHU, Y. LI, S. WANG, L. WANG, AND W. CHEN, *LoRA: Low-rank adaptation of large language models*, in International Conference on Learning Representations, 2022.

[176] H. HU, Z. SALCIC, L. SUN, G. DOBBIE, P. S. YU, AND X. ZHANG, *Membership inference attacks on machine learning: A survey*, ACM Computing Surveys (CSUR), 54 (2022), pp. 1–37.

[177] T. HUA, W. WANG, Z. XUE, S. REN, Y. WANG, AND H. ZHAO, *On feature decorrelation in self-supervised learning*, in IEEE/CVF International Conference on Computer Vision, 2021, pp. 9598–9608.

[178] I. HUBARA, B. CHMIEL, M. ISLAND, R. BANNER, J. NAOR, AND D. SOUDRY, *Accelerated sparse neural training: A provable and efficient method to find N:M transposable masks*, in Advances in Neural Information Processing Systems, vol. 34, 2021, pp. 21099–21111.

[179] I. HUBARA, Y. NAHSHAN, Y. HANANI, R. BANNER, AND D. SOUDRY, *Accurate post training quantization with small calibration sets*, in International Conference on Machine Learning, PMLR, 2021, pp. 4466–4475.

[180] Y. IDELBAYEV AND M. A. CARREIRA-PERPINÁN, *Low-rank compression of neural nets: Learning the rank of each layer*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 8049–8059.

[181] Y. IOANNOU, D. ROBERTSON, J. SHOTTON, R. CIPOLLA, AND A. CRIMINISI, *Training CNNs with low-rank filters for efficient image classification*, in International Conference on Learning Representations, 2016.

[182] S. IOFFE AND C. SZEGEDY, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, in International Conference on Machine Learning, PMLR, 2015, pp. 448–456.

[183] E. IOFINOVA, A. PESTE, M. KURTZ, AND D. ALISTARH, *How well do sparse imagenet models transfer?*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 12266–12276.

[184] A. ISCEN, G. TOLIAS, Y. AVRITHIS, AND O. CHUM, *Mining on manifolds: Metric learning without labels*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 7642–7651.

[185] M. JADERBERG, A. VEDALDI, AND A. ZISSERMAN, *Speeding up convolutional neural networks with low rank expansions*, in British Machine Vision Conference, BMVA Press, 2014.

[186] S. JAYAKUMAR, R. PASCANU, J. RAE, S. OSINDERO, AND E. ELSEN, *Top-KAST: Top-k always sparse training*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 20744–20754.

[187] Y. JIA, *Learning semantic image representations at a large scale*, tech. rep., University of California, Berkeley, 2014.

[188] C. JIANG, G. LI, C. QIAN, AND K. TANG, *Efficient DNN neuron pruning by minimizing layer-wise nonlinear reconstruction error*, in International Joint Conference on Artificial Intelligence, 2018, pp. 2298–2304.

[189] X. JIN, X. YUAN, J. FENG, AND S. YAN, *Training skinny deep neural networks with iterative hard thresholding methods*, arXiv preprint arXiv:1607.05423, 2016.

[190] L. JING, P. VINCENT, Y. LECUN, AND Y. TIAN, *Understanding dimensional collapse in contrastive self-supervised learning*, in International Conference on Learning Representations, 2022.

[191] T. KAILATH, S.-Y. KUNG, AND M. MORF, *Displacement ranks of matrices and linear equations*, Journal of Mathematical Analysis and Applications, 68 (1979), pp. 395–407.

[192] Y. KALANTIDIS, M. B. SARIYILDIZ, N. PION, P. WEINZAEPFEL, AND D. LARLUS, *Hard negative mixing for contrastive learning*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 21798–21809.

[193] J. KAPLAN, S. MCCANDLISH, T. HENIGHAN, T. B. BROWN, B. CHESS, R. CHILD, S. GRAY, A. RADFORD, J. WU, AND D. AMODEI, *Scaling laws for neural language models*, arXiv preprint arXiv:2001.08361, 2020.

[194] M. KECH AND F. KRAHMER, *Optimal injectivity conditions for bilinear inverse problems with applications to identifiability of deconvolution problems*, SIAM Journal on Applied Algebra and Geometry, 1 (2017), pp. 20–37.

[195] M. KHODAK, N. A. TENENHOLTZ, L. MACKEY, AND N. FUSI, *Initialization and regularization of factorized neural layers*, in International Conference on Learning Representations, 2021.

[196] Y.-D. KIM, E. PARK, S. YOO, T. CHOI, L. YANG, AND D. SHIN, *Compression of deep convolutional neural networks for fast and low power mobile applications*, in International Conference on Learning Representations, 2016.

[197] D. P. KINGMA AND J. BA, *Adam: A method for stochastic optimization*, in International Conference on Learning Representations, 2015.

[198] F. J. KIRÁLY, L. THERAN, AND R. TOMIOKA, *The algebraic combinatorial approach for low-rank matrix completion*, Journal of Machine Learning Research, 16 (2015), pp. 1391–1436.

[199] T. G. KOLDA AND B. W. BADER, *Tensor decompositions and applications*, SIAM Review, 51 (2009), pp. 455–500.

[200] A. KOLESNIKOV, L. BEYER, X. ZHAI, J. PUIGCERVER, J. YUNG, S. GELLY, AND N. HOULSBY, *Big transfer (BiT): General visual representation learning*, in European Conference on Computer Vision, Springer, 2020, pp. 491–507.

[201] A. KOVASHKA, O. RUSSAKOVSKY, L. FEI-FEI, K. GRAUMAN, ET AL., *Crowdsourcing in computer vision*, Foundations and Trends® in computer graphics and Vision, 10 (2016), pp. 177–243.

[202] A. KRIZHEVSKY, *Learning multiple layers of features from tiny images*, tech. rep., University of Toronto, 2009.

[203] A. KRIZHEVSKY, I. SUTSKEVER, AND G. E. HINTON, *ImageNet classification with deep convolutional neural networks*, in Advances in Neural Information Processing Systems, vol. 25, 2012.

[204] J. B. KRUSKAL, *Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics*, Linear algebra and its applications, 18 (1977), pp. 95–138.

[205] A. KUMAR, P. RAI, AND H. DAUME, *Co-regularized multi-view spectral clustering*, in Advances in Neural Information Processing Systems, vol. 24, 2011.

[206] E. KURTIĆ, E. FRANTAR, AND D. ALISTARH, *ZipLM: Inference-aware structured pruning of language models*, in Advances in Neural Information Processing Systems, vol. 36, 2024.

[207] A. KUSUPATI, V. RAMANUJAN, R. SOMANI, M. WORTSMAN, P. JAIN, S. KAKADE, AND A. FARHADI, *Soft threshold weight reparameterization for learnable sparsity*, in International Conference on Machine Learning, PMLR, 2020, pp. 5544–5555.

[208] C. LACLAU, I. REDKO, B. MATEI, Y. BENNANI, AND V. BRAULT, *Co-clustering through optimal transport*, in International Conference on Machine Learning, PMLR, 2017, pp. 1955–1964.

[209] D. H. LE AND B.-S. HUA, *Network pruning that matters: A case study on retraining variants*, in International Conference on Learning Representations, 2021.

[210] Q. LE, T. SARLÓS, A. SMOLA, ET AL., *Fastfood: Approximating kernel expansions in loglinear time*, in International Conference on Machine Learning, vol. 85, PMLR, 2013.

[211] Q.-T. LE, E. RICCIETTI, AND R. GRIBONVAL, *Does a sparse relu network training problem always admit an optimum?*, in Advances in Neural Information Processing Systems, vol. 36, 2023.

[212] Q.-T. LE, E. RICCIETTI, AND R. GRIBONVAL, *Spurious valleys, NP-Hardness, and tractability of sparse matrix factorization with fixed support*, SIAM Journal on Matrix Analysis and Applications, 44 (2023), pp. 503–529.

[213] Q.-T. LE, L. ZHENG, E. RICCIETTI, AND R. GRIBONVAL, *Fast learning of fast transforms, with guarantees*, in IEEE International Conference on Acoustics, Speech and Signal Processing, 2022, pp. 3348–3352.

[214] L. LE MAGOAROU, *Matrices efficientes pour le traitement du signal et l'apprentissage automatique*, PhD thesis, INSA de Rennes, 2016. Written in French.

[215] L. LE MAGOAROU AND R. GRIBONVAL, *Are there approximate fast Fourier transforms on graphs?*, in IEEE International Conference on Acoustics, Speech and Signal Processing, 2016, pp. 4811–4815.

[216] L. LE MAGOAROU AND R. GRIBONVAL, *Flexible multilayer sparse approximations of matrices and applications*, IEEE Journal of Selected Topics in Signal Processing, 10 (2016), pp. 688–700.

[217] L. LE MAGOAROU, R. GRIBONVAL, AND N. TREMBLAY, *Approximate fast graph Fourier transforms via multilayer sparse approximations*, IEEE Transactions on Signal and Information Processing over Networks, 4 (2017), pp. 407–420.

[218] V. LEBEDEV, Y. GANIN, M. RAKHUBA, I. OSELEDETS, AND V. LEMPITSKY, *Speeding-up convolutional neural networks using fine-tuned CP-decomposition*, in International Conference on Learning Representations, 2015.

[219] Y. LECUN, L. BOTTOU, Y. BENGIO, AND P. HAFFNER, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE, 86 (1998), pp. 2278–2324.

[220] Y. LECUN, J. DENKER, AND S. SOLLA, *Optimal brain damage*, in Advances in Neural Information Processing Systems, vol. 2, 1989.

[221] H. LEE, K. LEE, K. LEE, H. LEE, AND J. SHIN, *Improving transferability of representations via augmentation-aware self-supervision*, in Advances in Neural Information Processing Systems, vol. 34, 2021, pp. 17710–17722.

[222] N. LEE, T. AJANTHAN, AND P. H. TORR, *SNIP: Single-shot network pruning based on connection sensitivity*, in International Conference on Learning Representations, 2019.

[223] K. LENC AND A. VEDALDI, *Understanding image representations by measuring their equivariance and equivalence*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2015, pp. 991–999.

[224] H. LI, A. KADAV, I. DURDANOVIC, H. SAMET, AND H. P. GRAF, *Pruning filters for efficient convnets*, in International Conference on Learning Representations, 2017.

[225] X. LI, Y. LIANG, S. YAN, L. JIA, AND Y. LI, *A coordinated tiling and batching framework for efficient GEMM on GPUs*, in Symposium on Principles and Practice of Parallel Programming, 2019, pp. 229–241.

[226] X. LI AND V. VORONINSKI, *Sparse signal recovery from quadratic measurements via convex programming*, SIAM Journal on Mathematical Analysis, 45 (2013), pp. 3019–3033.

[227] Y. LI, R. GONG, X. TAN, Y. YANG, P. HU, Q. ZHANG, F. YU, W. WANG, AND S. GU, *BRECQ: Pushing the limit of post-training quantization by block reconstruction*, in International Conference on Learning Representations, 2021.

[228] Y. LI, K. LEE, AND Y. BRESLER, *Identifiability in bilinear inverse problems with applications to subspace or sparsity-constrained blind gain and phase calibration*, IEEE Transactions on Information Theory, 63 (2016), pp. 822–842.

[229] Y. LI, K. LEE, AND Y. BRESLER, *Identifiability in blind deconvolution with subspace or sparsity constraints*, IEEE Transactions on information Theory, 62 (2016), pp. 4266–4275.

[230] Y. LI, K. LEE, AND Y. BRESLER, *Identifiability and stability in blind deconvolution under minimal assumptions*, IEEE Transactions on Information Theory, 63 (2017), pp. 4619–4633.

[231] Y. LI, R. POGODIN, D. J. SUTHERLAND, AND A. GRETTON, *Self-supervised learning with kernel dependence maximization*, in Advances in Neural Information Processing Systems, vol. 34, 2021, pp. 15543–15556.

[232] Y. LI, K. SWERSKY, AND R. ZEMEL, *Generative moment matching networks*, in International Conference on Machine Learning, PMLR, 2015, pp. 1718–1727.

[233] Y. LI AND H. YANG, *Interpolative butterfly factorization*, SIAM Journal on Scientific Computing, 39 (2017), pp. A503–A531.

[234] Y. LI, H. YANG, E. R. MARTIN, K. L. HO, AND L. YING, *Butterfly factorization*, Multiscale Modeling & Simulation, 13 (2015), pp. 714–732.

[235] Y. LI, H. YANG, AND L. YING, *A multiscale butterfly algorithm for multidimensional Fourier integral operators*, Multiscale Modeling & Simulation, 13 (2015), pp. 614–631.

[236] Y. LI, H. YANG, AND L. YING, *Multidimensional butterfly factorization*, Applied and Computational Harmonic Analysis, 44 (2018), pp. 737–758.

[237] Y. LI, Y. YU, Q. ZHANG, C. LIANG, P. HE, W. CHEN, AND T. ZHAO, *LoSparse: Structured compression of large language models based on low-rank and sparse approximation*, in International Conference on Machine Learning, PMLR, 2023, pp. 20336–20350.

[238] Z. LI, E. WALLACE, S. SHEN, K. LIN, K. KEUTZER, D. KLEIN, AND J. GONZALEZ, *Train big, then compress: Rethinking model size for efficient training and inference of transformers*, in International Conference on Machine Learning, PMLR, 2020, pp. 5958–5968.

[239] T. LIANG, J. GLOSSNER, L. WANG, S. SHI, AND X. ZHANG, *Pruning and quantization for deep neural network acceleration: A survey*, Neurocomputing, 461 (2021), pp. 370–403.

[240] E. LIBERTY, F. WOOLFE, P.-G. MARTINSSON, V. ROKHLIN, AND M. TYGERT, *Randomized algorithms for the low-rank approximation of matrices*, Proceedings of the National Academy of Sciences, 104 (2007), pp. 20167–20172.

[241] R. LIN, J. RAN, K. H. CHIU, G. CHESI, AND N. WONG, *Deformable butterfly: A highly structured and sparse linear transform*, in Advances in Neural Information Processing Systems, vol. 34, 2021, pp. 16145–16157.

[242] S. LING AND T. STROHMER, *Self-calibration and biconvex compressive sensing*, Inverse Problems, 31 (2015), p. 115002.

[243] B. LIU, M. WANG, H. FOROOSH, M. TAPPEN, AND M. PENSKY, *Sparse convolutional neural networks*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2015, pp. 806–814.

[244] D. C. LIU AND J. NOCEDAL, *On the limited memory BFGS method for large scale optimization*, Mathematical programming, 45 (1989), pp. 503–528.

[245] S. LIU, T. CHEN, X. CHEN, L. SHEN, D. C. MOCANU, Z. WANG, AND M. PECHENIZKIY, *The unreasonable effectiveness of random pruning: Return of the most naive baseline for sparse training*, in International Conference on Learning Representations, 2022.

[246] S. LIU, T. CHEN, Z. ZHANG, X. CHEN, T. HUANG, A. K. JAISWAL, AND Z. WANG, *Sparsity may cry: Let us fail (current) sparse neural networks together!*, in International Conference on Learning Representations, 2023.

[247] W. LIU, R. LIN, Z. LIU, L. LIU, Z. YU, B. DAI, AND L. SONG, *Learning towards minimum hyperspherical energy*, in Advances in Neural Information Processing Systems, vol. 31, 2018.

[248] W. LIU, Z. QIU, Y. FENG, Y. XIU, Y. XUE, L. YU, H. FENG, Z. LIU, J. HEO, S. PENG, Y. WEN, M. J. BLACK, A. WELLER, AND B. SCHÖLKOPF, *Parameter-efficient orthogonal finetuning via butterfly factorization*, in International Conference on Learning Representations, 2024.

[249] W. LIU, Y. WEN, Z. YU, M. LI, B. RAJ, AND L. SONG, *Sphereface: Deep hypersphere embedding for face recognition*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2017, pp. 212–220.

[250] Y. LIU, X. XING, H. GUO, E. MICHIELSSEN, P. GHYSELS, AND X. S. LI, *Butterfly factorization via randomized matrix-vector multiplications*, SIAM Journal on Scientific Computing, 43 (2021), pp. A883–A907.

[251] Y. LIU AND H. YANG, *A hierarchical butterfly LU preconditioner for two-dimensional electromagnetic scattering problems involving open surfaces*, Journal of Computational Physics, 401 (2020), p. 109014.

[252] Z. LIU, M. SUN, T. ZHOU, G. HUANG, AND T. DARRELL, *Rethinking the value of network pruning*, in International Conference on Learning Representations, 2019.

[253] I. LOSHCHILOV AND F. HUTTER, *SGDR: Stochastic gradient descent with warm restarts*, in International Conference on Learning Representations, 2017.

[254] C. LOUIZOS, M. WELLING, AND D. P. KINGMA, *Learning sparse neural networks through l_0 regularization*, in International Conference on Learning Representations, 2018.

[255] A. S. LUCCIONI, S. VIGUIER, AND A.-L. LIGOZAT, *Estimating the carbon footprint of BLOOM, a 176B parameter language model*, Journal of Machine Learning Research, 24 (2023), pp. 1–15.

[256] Y. LYU, *Spherical structured feature maps for kernel approximation*, in International Conference on Machine Learning, PMLR, 2017, pp. 2256–2264.

[257] F. MALGOUYRES, *On the stable recovery of deep structured linear networks under sparsity constraints*, in Mathematical and Scientific Machine Learning, PMLR, 2020, pp. 107–127.

[258] F. MALGOUYRES AND J. LANDSBERG, *On the identifiability and stable recovery of deep/multilayer structured matrix factorization*, in IEEE Information Theory Workshop, 2016, pp. 315–319.

[259] F. MALGOUYRES AND J. LANDSBERG, *Multilinear compressive sensing and an application to convolutional linear networks*, SIAM Journal on Mathematics of Data Science, 1 (2019), pp. 446–475.

[260] P.-G. MARTINSSON, *Fast multipole methods*, in Encyclopedia of Applied and Computational Mathematics, Springer Berlin Heidelberg, 2015, pp. 498–508.

[261] P.-G. MARTINSSON, V. ROKHLIN, AND M. TYGERT, *On interpolation and integration in finite-dimensional spaces of bounded functions*, Communications in Applied Mathematics and Computational Science, 1 (2007), pp. 133–142.

[262] V. MAURIZIO, *Double k-means clustering for simultaneous classification of objects and variables*, in Advances in Classification and Data Analysis, Springer, 2001, pp. 43–52.

[263] W. S. MCCULLOCH AND W. PITTS, *A logical calculus of the ideas immanent in nervous activity*, The bulletin of mathematical biophysics, 5 (1943), pp. 115–133.

[264] C. A. MICCHELLI, Y. XU, AND H. ZHANG, *Universal kernels*, Journal of Machine Learning Research, 7 (2006).

[265] E. MICHIELSSEN AND A. BOAG, *Multilevel evaluation of electromagnetic fields for the rapid solution of scattering problems*, Microwave and Optical Technology Letters, 7 (1994), pp. 790–795.

[266] E. MICHIELSSEN AND A. BOAG, *A multilevel matrix decomposition algorithm for analyzing scattering from large structures*, IEEE Transactions on Antennas and Propagation, 44 (1996), pp. 1086–1093.

[267] I. MISRA AND L. V. D. MAATEN, *Self-supervised learning of pretext-invariant representations*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 6707–6717.

[268] D. C. MOCANU, E. MOCANU, P. STONE, P. H. NGUYEN, M. GIBESCU, AND A. LIOTTA, *Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science*, Nature communications, 9 (2018), p. 2383.

[269] M. MOCZULSKI, M. DENIL, J. APPLEYARD, AND N. DE FREITAS, *ACDC: A structured efficient linear layer*, in International Conference on Learning Representations, 2016.

[270] P. MOLCHANOV, A. MALLYA, S. TYREE, I. FROSIO, AND J. KAUTZ, *Importance estimation for neural network pruning*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 11264–11272.

[271] P. MOLCHANOV, S. TYREE, T. KARRAS, T. AILA, AND J. KAUTZ, *Pruning convolutional neural networks for resource efficient inference*, in International Conference on Learning Representations, 2017.

[272] B. MOONS, D. BANKMAN, AND M. VERHELST, *Embedded deep learning*, Springer, 2019.

[273] H. MOSTAFA AND X. WANG, *Parameter efficient training of deep convolutional neural networks by dynamic sparse reparameterization*, in International Conference on Machine Learning, PMLR, 2019, pp. 4646–4655.

[274] K. MUANDET, K. FUKUMIZU, B. SRIPERUMBUDUR, AND B. SCHÖLKOPF, *Kernel mean embedding of distributions: A review and beyond*, Foundations and Trends® in Machine Learning, 10 (2017), pp. 1–141.

[275] T. N. MUNDHENK, D. HO, AND B. Y. CHEN, *Improvements to context based self-supervised learning*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 9339–9348.

[276] M. MUNKHOEVA, Y. KAPUSHEV, E. BURNAEV, AND I. OSELEDETS, *Quadrature-based features for kernel approximation*, in Advances in Neural Information Processing Systems, vol. 31, 2018.

[277] C. MÜLLER, *Analysis of spherical symmetries in Euclidean spaces*, Springer Science & Business Media, 2012.

[278] M. NAGEL, R. A. AMJAD, M. VAN BAALEN, C. LOUIZOS, AND T. BLANKEVOORT, *Up or down? Adaptive rounding for post-training quantization*, in International Conference on Machine Learning, PMLR, 2020, pp. 7197–7206.

[279] M. NEGNEVITSKY, *Artificial intelligence: a guide to intelligent systems*, Pearson education, 2005.

[280] B. NEYSHABUR, R. R. SALAKHUTDINOV, AND N. SREBRO, *Path-SGD: Path-normalized optimization in deep neural networks*, in Advances in Neural Information Processing Systems, vol. 28, 2015.

[281] M. NOROOZI AND P. FAVARO, *Unsupervised learning of visual representations by solving jigsaw puzzles*, in European Conference on Computer Vision, Springer, 2016, pp. 69–84.

[282] M. NOROOZI, H. PIRSIAVASH, AND P. FAVARO, *Representation learning by learning to count*, in IEEE/CVF International Conference on Computer Vision, 2017, pp. 5898–5906.

[283] NVIDIA, *Efficient GEMM in CUDA: documentation*, 2023. https://github.com/NVIDIA/cutlass/blob/main/media/docs/efficient_gemm.md [Accessed: February, 2024].

[284] ——, *Matrix multiplication background user's guide*, 2023. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html [Accessed: February, 2024].

[285] ——, *CUDA C++ programming guide*, 2024. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#c-language-extensions [Accessed: February, 2024].

[286] R. OHIB, B. THAPALIYA, P. GAGGENAPALLI, J. LIU, V. CALHOUN, AND S. PLIS, *SalientGrads: Sparse models for communication efficient and data aware distributed federated training*, arXiv preprint arXiv:2304.07488, 2023.

[287] V. OLSHEVSKY AND A. SHOKROLLAHI, *Matrix-vector product for confluent cauchy-like matrices with application to confluent rational interpolation*, in ACM symposium on Theory of computing, 2000, pp. 573–581.

[288] M. O'NEIL, *A new class of analysis-based fast transforms*, PhD thesis, Yale University, 2007.

[289] M. O'NEIL, F. WOOLFE, AND V. ROKHLIN, *An algorithm for the rapid evaluation of special function transforms*, Applied and Computational Harmonic Analysis, 28 (2010), pp. 203–226.

[290] A. V. D. OORD, Y. LI, AND O. VINYALS, *Representation learning with contrastive predictive coding*, arXiv preprint arXiv:1807.03748, 2018.

[291] M. OQUAB, T. DARCET, T. MOUTAKANNI, H. V. VO, M. SZAFRANIEC, V. KHALIDOV, P. FERNANDEZ, D. HAZIZA, F. MASSA, A. EL-NOUBY, M. ASSRAN, N. BALLAS, W. GALUBA, R. HOWES, P.-Y. HUANG, S.-W. LI, I. MISRA, M. RABBAT, V. SHARMA, G. SYNNAEVE, H. XU, H. JEGOU, J. MAIRAL, P. LABATUT, A. JOULIN, AND P. BOJANOWSKI, *DINOv2: Learning robust visual features without supervision*, Transactions on Machine Learning Research, (2024).

[292] Y. OUALI, C. HUDELOT, AND M. TAMI, *An overview of deep semi-supervised learning*, arXiv preprint arXiv:2006.05278, 2020.

[293] S. J. PAN AND Q. YANG, *A survey on transfer learning*, IEEE Transactions on Knowledge and Data Engineering, 22 (2009), pp. 1345–1359.

[294] V. PAN, *Structured matrices and polynomials: unified superfast algorithms*, Springer Science & Business Media, 2001.

[295] V. Y. PAN AND X. WANG, *Inversion of displacement operators*, SIAM Journal on Matrix Analysis and Applications, 24 (2003), pp. 660–677.

[296] J. Y. PARK, O. BIZA, L. ZHAO, J.-W. VAN DE MEENT, AND R. WALTERS, *Learning symmetric embeddings for equivariant world models*, in International Conference on Machine Learning, PMLR, 2022, pp. 17372–17389.

[297] D. S. PARKER, *Random butterfly transformations with applications in computational linear algebra*, tech. rep., UCLA Computer Science Department, 1995.

[298] O. PARKHI, A. VEDALDI, AND A. ZISSERMAN, *Deep face recognition*, in British Machine Vision Conference 2015, BMVA, 2015.

[299] D. PATHAK, P. KRAHENBUHL, J. DONAHUE, T. DARRELL, AND A. A. EFROS, *Context encoders: Feature learning by inpainting*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2016, pp. 2536–2544.

[300] D. PATTERSON, J. GONZALEZ, Q. LE, C. LIANG, L.-M. MUNGUIA, D. ROTHCHILD, D. SO, M. TEXIER, AND J. DEAN, *Carbon emissions and large neural network training*, arXiv preprint arXiv:2104.10350, 2021.

[301] X. PENG, K. WANG, Z. ZHU, M. WANG, AND Y. YOU, *Crafting better contrastive views for Siamese representation learning*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 16031–16040.

[302] J. PENNINGTON, F. X. X. YU, AND S. KUMAR, *Spherical random features for polynomial kernels*, in Advances in Neural Information Processing Systems, vol. 28, 2015.

[303] A. PINKUS, *Approximation theory of the MLP model in neural networks*, Acta numerica, 8 (1999), pp. 143–195.

[304] A. PYATKIN, D. ALOISE, AND N. MLADENOVIĆ, *NP-Hardness of balanced minimum sum-of-squares clustering*, Pattern Recognition Letters, 97 (2017), pp. 44–45.

[305] Z. QIU, W. LIU, H. FENG, Y. XUE, Y. FENG, Z. LIU, D. ZHANG, A. WELLER, AND B. SCHÖLKOPF, *Controlling text-to-image diffusion by orthogonal finetuning*, in Advances in Neural Information Processing Systems, vol. 36, 2024.

[306] C. M. RADER, *Discrete Fourier transforms when the number of data samples is prime*, Proceedings of the IEEE, 56 (1968), pp. 1107–1108.

[307] A. RADFORD, J. W. KIM, C. HALLACY, A. RAMESH, G. GOH, S. AGARWAL, G. SASTRY, A. ASKELL, P. MISHKIN, J. CLARK, G. KRUEGER, AND I. SUTSKEVER, *Learning transferable visual models from natural language supervision*, in International Conference on Machine Learning, PMLR, 2021, pp. 8748–8763.

[308] A. RAHIMI AND B. RECHT, *Random features for large-scale kernel machines*, in Advances in Neural Information Processing Systems, vol. 20, 2007.

[309] A. RAMESH, M. PAVLOV, G. GOH, S. GRAY, C. VOSS, A. RADFORD, M. CHEN, AND I. SUTSKEVER, *Zero-shot text-to-image generation*, in International Conference on Machine Learning, PMLR, 2021, pp. 8821–8831.

[310] I. REDKO, T. VAYER, R. FLAMARY, AND N. COURTY, *Co-optimal transport*, in Advances in Neural Information Processing Systems, vol. 33, 2020, p. 2.

[311] S. REN, K. HE, R. GIRSHICK, AND J. SUN, *Faster R-CNN: Towards real-time object detection with region proposal networks*, in Advances in Neural Information Processing Systems, vol. 28, 2015.

[312] L. RICE, E. WONG, AND Z. KOLTER, *Overfitting in adversarially robust deep learning*, in International Conference on Machine Learning, PMLR, 2020, pp. 8093–8104.

[313] P. H. RICHEMOND, J.-B. GRILL, F. ALTCHÉ, C. TALLEC, F. STRUB, A. BROCK, S. SMITH, S. DE, R. PASCANU, B. PIOT, AND M. VALKO, *BYOL works even without batch statistics*, arXiv preprint arXiv:2010.10241, 2020.

[314] D. E. RUMELHART, G. E. HINTON, AND R. J. WILLIAMS, *Learning representations by back-propagating errors*, Nature, 323 (1986), pp. 533–536.

[315] A. SANDRYHAILA AND J. M. MOURA, *Discrete signal processing on graphs*, IEEE Transactions on Signal Processing, 61 (2013), pp. 1644–1656.

[316] V. SANH, T. WOLF, AND A. RUSH, *Movement pruning: Adaptive sparsity by fine-tuning*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 20378–20389.

[317] L. SCHMIDT, S. SANTURKAR, D. TSIPRAS, K. TALWAR, AND A. MADRY, *Adversarially robust generalization requires more data*, in Advances in Neural Information Processing Systems, vol. 31, 2018.

[318] I. J. SCHOENBERG, *Positive definite functions on spheres*, Duke Mathematical Journal, 9 (1942), pp. 96–108.

[319] F. SCHROFF, D. KALENICHENKO, AND J. PHILBIN, *Facenet: A unified embedding for face recognition and clustering*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2015, pp. 815–823.

[320] R. SCHWARTZ, J. DODGE, N. A. SMITH, AND O. ETZIONI, *Green AI*, Communications of the ACM, 63 (2020), pp. 54–63.

[321] S.-K. SHEKOFTEH, C. ALLES, AND H. FRÖNING, *Reducing memory requirements for the ipu using butterfly factorizations*, in Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, 2023, pp. 1255–1263.

[322] D. I. SHUMAN, S. K. NARANG, P. FROSSARD, A. ORTEGA, AND P. VANDERGHEYNST, *The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains*, IEEE Signal Processing Magazine, 30 (2013), pp. 83–98.

[323] K. SIMONYAN AND A. ZISSERMAN, *Very deep convolutional networks for large-scale image recognition*, in International Conference on Learning Representations, 2015.

[324] S. P. SINGH AND D. ALISTARH, *Woodfisher: Efficient second-order approximation for neural network compression*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 18098–18109.

[325] A. SMOLA, Z. OVÁRI, AND R. C. WILLIAMSON, *Regularization with dot-product kernels*, in Advances in Neural Information Processing Systems, vol. 13, 2000.

[326] K. SOHN, *Improved deep metric learning with multi-class n-pair loss objective*, in Advances in Neural Information Processing Systems, vol. 29, 2016.

[327] M. R. SPIEGEL, S. LIPSCHUTZ, AND J. LIU, *Mathematical Handbook of Formulas and Tables*, Schaum's Outlines, 2013.

[328] E. STRUBELL, A. GANESH, AND A. MCCALLUM, *Energy and policy considerations for deep learning in NLP*, in Annual Meeting of the Association for Computational Linguistics, 2019, pp. 3645–3650.

[329] C. SUN, A. SHRIVASTAVA, S. SINGH, AND A. GUPTA, *Revisiting unreasonable effectiveness of data in deep learning era*, in IEEE/CVF International Conference on Computer Vision, 2017, pp. 843–852.

[330] X. SUN AND N. P. PITSIANIS, *A matrix version of the fast multipole method*, SIAM Review, 43 (2001), pp. 289–300.

[331] Y. SUN AND Y. LI, *Dice: Leveraging sparsification for out-of-distribution detection*, in European Conference on Computer Vision, Springer, 2022, pp. 691–708.

[332] C. SZEGEDY, W. LIU, Y. JIA, P. SERMANET, S. REED, D. ANGUELOV, D. ERHAN, V. VANHOUCKE, AND A. RABINOVICH, *Going deeper with convolutions*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2015, pp. 1–9.

[333] C. TAI, T. XIAO, Y. ZHANG, X. WANG, AND W. E, *Convolutional neural networks with low-rank regularization*, in International Conference on Learning Representations, 2016.

[334] M. TAN AND Q. LE, *Efficientnet: Rethinking model scaling for convolutional neural networks*, in International Conference on Machine Learning, PMLR, 2019, pp. 6105–6114.

[335] H. TANAKA, D. KUNIN, D. L. YAMINS, AND S. GANGULI, *Pruning neural networks without any data by iteratively conserving synaptic flow*, in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 6377–6389.

[336] Y. TAY, M. DEHGHANI, S. ABNAR, H. CHUNG, W. FEDUS, J. RAO, S. NARANG, V. TRAN, D. YOGATAMA, AND D. METZLER, *Scaling laws vs model architectures: How does inductive bias influence scaling?*, in Conference on Empirical Methods in Natural Language Processing, ACL, 2023, pp. 12342–12364.

[337] Y. TAY, M. DEHGHANI, D. BAHRI, AND D. METZLER, *Efficient transformers: A survey*, ACM Computing Surveys, 55 (2022), pp. 1–28.

[338] Y. TAY, M. DEHGHANI, J. RAO, W. FEDUS, S. ABNAR, H. W. CHUNG, S. NARANG, D. YOGATAMA, A. VASWANI, AND D. METZLER, *Scale efficiently: Insights from pretraining and fine-tuning transformers*, in International Conference on Learning Representations, 2022.

[339] Y. TIAN, X. CHEN, AND S. GANGULI, *Understanding self-supervised learning dynamics without contrastive pairs*, in International Conference on Machine Learning, PMLR, 2021, pp. 10268–10278.

[340] H. TOUVRON, M. CORD, M. DOUZE, F. MASSA, A. SABLAYROLLES, AND H. JÉGOU, *Training data-efficient image transformers & distillation through attention*, in International Conference on Machine Learning, PMLR, 2021, pp. 10347–10357.

[341] M. TYGERT, *Fast algorithms for spherical harmonic expansions, III*, Journal of Computational Physics, 229 (2010), pp. 6181–6192.

[342] K. A. VAHID, A. PRABHU, A. FARHADI, AND M. RASTEGARI, *Butterfly transform: An efficient FFT based neural architecture design*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 12021–12030.

[343] C. F. VAN LOAN, *The ubiquitous Kronecker product*, Journal of Computational and Applied Mathematics, 123 (2000), pp. 85–100.

[344] M. VARMA T, X. CHEN, Z. ZHANG, T. CHEN, S. VENUGOPALAN, AND Z. WANG, *Sparse winning tickets are data-efficient image recognizers*, in Advances in Neural Information Processing Systems, vol. 35, 2022, pp. 4652–4666.

[345] A. VASWANI, N. SHAZEER, N. PARMAR, J. USZKOREIT, L. JONES, A. N. GOMEZ, Ł. KAISER, AND I. POLOSUKHIN, *Attention is all you need*, in Advances in Neural Information Processing Systems, vol. 30, 2017.

[346] R. VIDAL, *Sparse subspace clustering*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, vol. 6, 2009, pp. 2790–2797.

[347] R. VIDAL, *Subspace clustering*, IEEE Signal Processing Magazine, 28 (2011), pp. 52–68.

[348] R. VIDAL, Y. MA, AND S. SASTRY, *Generalized Principal Component Analysis*, Springer New York, 2016.

[349] P. VINCENT, H. LAROCHELLE, Y. BENGIO, AND P.-A. MANZAGOL, *Extracting and composing robust features with denoising autoencoders*, in International Conference on Machine Learning, PMLR, 2008, pp. 1096–1103.

[350] U. VON LUXBURG, *A tutorial on spectral clustering*, Statistics and computing, 17 (2007), pp. 395–416.

[351] A. WANG, A. SINGH, J. MICHAEL, F. HILL, O. LEVY, AND S. BOWMAN, *GLUE: A multi-task benchmark and analysis platform for natural language understanding*, in EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, ACL, 2018, pp. 353–355.

[352] C. WANG, G. ZHANG, AND R. GROSSE, *Picking winning tickets before training by preserving gradient flow*, in International Conference on Learning Representations, 2020.

[353] H. WANG, C. QIN, Y. BAI, AND Y. FU, *Why is the state of neural network pruning so confusing? On the fairness, comparison setup, and trainability in network pruning*, arXiv preprint arXiv:2301.05219, 2023.

[354] T. WANG AND P. ISOLA, *Understanding contrastive representation learning through alignment and uniformity on the hypersphere*, in International Conference on Machine Learning, PMLR, 2020, pp. 9929–9939.

[355] K. Q. WEINBERGER AND L. K. SAUL, *Distance metric learning for large margin nearest neighbor classification*, Journal of Machine Learning Research, 10 (2009).

[356] H. WEYL, *On the volume of tubes*, American Journal of Mathematics, 61 (1939), pp. 461–472.

[357] M. WU, C. ZHUANG, M. MOSSE, D. YAMINS, AND N. GOODMAN, *On mutual information in contrastive learning for visual representations*, arXiv preprint arXiv:2005.13149, 2020.

[358] Z. WU, Y. XIONG, S. X. YU, AND D. LIN, *Unsupervised feature learning via non-parametric instance discrimination*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 3733–3742.

[359] T. XIAO, X. WANG, A. A. EFROS, AND T. DARRELL, *What should not be contrastive in contrastive learning*, in International Conference on Learning Representations, 2021.

[360] X. XIAO, C. LI, AND Y. LEI, *A lightweight self-supervised representation learning algorithm for scene classification in spaceborne SAR and optical images*, Remote Sensing, 14 (2022), p. 2956.

[361] J. XIE, X. ZHAN, Z. LIU, Y. S. ONG, AND C. C. LOY, *Unsupervised object-level representation learning from scene images*, in Advances in Neural Information Processing Systems, vol. 34, 2021, pp. 28864–28876.

[362] S. XIE, J. GU, D. GUO, C. R. QI, L. GUIBAS, AND O. LITANY, *PointContrast: Unsupervised pre-training for 3D point cloud understanding*, in European Conference on Computer Vision, Springer, 2020, pp. 574–591.

[363] Y. XIE, J. WEN, K. W. LAU, Y. A. U. REHMAN, AND J. SHEN, *What should be equivariant in self-supervised learning*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 4111–4120.

[364] Z. XIE, Z. ZHANG, Y. CAO, Y. LIN, J. BAO, Z. YAO, Q. DAI, AND H. HU, *Simmim: A simple framework for masked image modeling*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 9653–9663.

[365] Y. XU, Y. LI, S. ZHANG, W. WEN, B. WANG, Y. QI, Y. CHEN, W. LIN, AND H. XIONG, *TRP: Trained rank pruning for efficient deep neural networks*, in Workshop on Energy Efficient Machine Learning and Cognitive Computing, NeurIPS Edition, 2019, pp. 14–17.

[366] J. YAN AND M. POLLEFEYS, *A general framework for motion segmentation: Independent, articulated, rigid, non-rigid, degenerate and non-degenerate*, in European Conference on Computer Vision, Springer, 2006, pp. 94–106.

[367] J. YANG, K. ZHOU, Y. LI, AND Z. LIU, *Generalized out-of-distribution detection: A survey*, arXiv preprint arXiv:2110.11334, 2021.

[368] K. YE AND L.-H. LIM, *Every matrix is a product of Toeplitz matrices*, Foundations of Computational Mathematics, 16 (2016), pp. 577–598.

[369] L. YING, *Sparse Fourier transform via butterfly algorithm*, SIAM Journal on Scientific Computing, 31 (2009), pp. 1678–1694.

[370] Y. YOU, I. GITMAN, AND B. GINSBURG, *Large batch training of convolutional networks*, arXiv preprint arXiv:1708.03888, 2017.

[371] H. YU AND J. WU, *Compressing transformers: features are low-rank, but weights are not!*, in AAAI Conference on Artificial Intelligence, vol. 37, 2023, pp. 11007–11015.

[372] X. YU, T. LIU, X. WANG, AND D. TAO, *On compressing deep models by low rank and sparse decomposition*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2017, pp. 7370–7379.

[373] X. YUAN AND L. ZHANG, *Membership inference attacks and defenses in neural network pruning*, in USENIX Security Symposium, 2022, pp. 4561–4578.

[374] S. ZAGORUYKO AND N. KOMODAKIS, *Wide residual networks*, in British Machine Vision Conference, BMVA, 2016.

[375] J. ZBONTAR, L. JING, I. MISRA, Y. LECUN, AND S. DENY, *Barlow twins: Self-supervised learning via redundancy reduction*, in International Conference on Machine Learning, PMLR, 2021, pp. 12310–12320.

[376] X. ZHAI, A. KOLESNIKOV, N. HOULSBY, AND L. BEYER, *Scaling vision transformers*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 12104–12113.

[377] X. ZHAI, B. MUSTAFA, A. KOLESNIKOV, AND L. BEYER, *Sigmoid loss for language image pre-training*, in IEEE/CVF International Conference on Computer Vision, 2023, pp. 11975–11986.

[378] X. ZHAI, J. PUIGCERVER, A. KOLESNIKOV, P. RUYSSEN, C. RIQUELME, M. LUCIC, J. DJOLONGA, A. S. PINTO, M. NEUMANN, A. DOSOVITSKIY, L. BEYER, O. BACHEM, M. TSCHANNEN, M. MICHALSKI, O. BOUSQUET, S. GELLY, AND N. HOULSBY, *A large-scale study of representation learning with the visual task adaptation benchmark*, arXiv preprint arXiv:1910.04867,2019.

[379] L. ZHANG, G.-J. QI, L. WANG, AND J. LUO, *AET vs. AED: Unsupervised representation learning by auto-encoding transformations rather than data*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 2547–2555.

[380] R. ZHANG, P. ISOLA, AND A. A. EFROS, *Colorful image colorization*, in European Conference on Computer Vision, Springer, 2016, pp. 649–666.

[381] T. ZHANG, A. SZLAM, Y. WANG, AND G. LERMAN, *Hybrid linear modeling via local best-fit flats*, International Journal of Computer Vision, 100 (2012), pp. 217–240.

[382] X. ZHANG, X. ZHOU, M. LIN, AND J. SUN, *ShuffleNet: An extremely efficient convolutional neural network for mobile devices*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 6848–6856.

[383] X. ZHANG, J. ZOU, K. HE, AND J. SUN, *Accelerating very deep convolutional networks for classification and detection*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 38 (2015), pp. 1943–1955.

[384] R. ZHAO, Y. HU, J. DOTZEL, C. D. SA, AND Z. ZHANG, *Building efficient deep neural networks with unitary group convolutions*, in IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 11303–11312.

[385] L. ZHENG, Q.-T. LE, E. RICCIETTI, AND R. GRIBONVAL, *Code for reproducible research – Fast learning of fast transforms, with guarantees*. https://hal.inria.fr/hal-03552956, 2022.

[386] L. ZHENG, G. PUY, E. RICCIETTI, P. PÉREZ, AND R. GRIBONVAL, *Butterfly factorization by algorithmic identification of rank-one blocks*, arXiv preprint arXiv:2307.00820, 2023.

[387] L. ZHENG, G. PUY, E. RICCIETTI, P. PÉREZ, AND R. GRIBONVAL, *Code for reproducible research – Self-supervised learning with rotation-invariant kernels*. https://hal.inria.fr/hal-03737572/, 2022.

[388] L. ZHENG, G. PUY, E. RICCIETTI, P. PEREZ, AND R. GRIBONVAL, *Self-supervised learning with rotation-invariant kernels*, in International Conference on Learning Representations, 2023.

[389] L. ZHENG, G. PUY, E. RICCIETTI, P. PÉREZ, AND R. GRIBONVAL, *Code for reproducible research - Butterfly factorization by algorithmic identification of rank-one blocks*. https://inria.hal.science/hal-04576156, 2024.

[390] L. ZHENG, E. RICCIETTI, AND R. GRIBONVAL, *Identifiability in two-layer sparse matrix factorization*, arXiv preprint arXiv:2110.01235, 2021.

[391] L. ZHENG, E. RICCIETTI, AND R. GRIBONVAL, *Code for reproducible research – Efficient identification of butterfly sparse matrix factorizations*. https://hal.science/hal-03620052, 2022.

[392] L. ZHENG, E. RICCIETTI, AND R. GRIBONVAL, *Efficient identification of butterfly sparse matrix factorizations*, SIAM Journal on Mathematics of Data Science, 5 (2023), pp. 22–49.

[393] A. ZHOU, Y. MA, J. ZHU, J. LIU, Z. ZHANG, K. YUAN, W. SUN, AND H. LI, *Learning N:M fine-grained structured sparse neural networks from scratch*, in International Conference on Learning Representations, 2021.

[394] B. ZHOU, A. LAPEDRIZA, J. XIAO, A. TORRALBA, AND A. OLIVA, *Learning deep features for scene recognition using Places database*, in Advances in Neural Information Processing Systems, vol. 27, 2014.

[395] J. ZHOU, C. WEI, H. WANG, W. SHEN, C. XIE, A. YUILLE, AND T. KONG, *Image BERT pre-training with online tokenizer*, in International Conference on Learning Representations, 2022.

[396] S. K. ZHOU, H. GREENSPAN, C. DAVATZIKOS, J. S. DUNCAN, B. VAN GINNEKEN, A. MADABHUSHI, J. L. PRINCE, D. RUECKERT, AND R. M. SUMMERS, *A review of deep learning in medical imaging: Imaging traits, technology trends, case studies with progress highlights, and future promises*, Proceedings of the IEEE, 109 (2021), pp. 820–838.

[397] M. ZHU AND S. GUPTA, *To prune, or not to prune: exploring the efficacy of pruning for model compression*, arXiv preprint arXiv:1710.01878, 2017.

[398] Z. ZHU, S. KARNIK, M. A. DAVENPORT, J. ROMBERG, AND M. B. WAKIN, *The eigenvalue distribution of discrete periodic time-frequency limiting operators*, IEEE Signal Processing Letters, 25 (2017), pp. 95–99.